Sistemas Operativos

Jesús Torres <jmtorres@ull.es>

Edición OCW

Tabla de Contenido

Los sistemas operativos son un componente esencial de cualquier sistema informático. Se ejecutan continuamente desde que el sistema se inicia, gestionan los recursos del hardware y ofrecen un entorno para la ejecución de los programas. Por ejemplo, aunque un sistema solo tenga una CPU y una cantidad limitada de memoria —como era común hasta hace unos años— los sistemas operativos modernos son capaces de crear la ilusión de que es capaz de ejecutar múltiples tareas en paralelo y los programadores, por lo general, no tienen que preocuparse por la cantidad de memoria realmente disponible.

Excepto en sistemas muy pequeños —donde no se suelen utilizar sistemas operativos— los programadores desarrollan para el entorno que el sistema operativo ofrece y necesitan solicitarle servicios constantemente, para hacer cualquier programa medianamente útil. Por eso es importante conocer lo que hacen los sistemas operativos, los servicios que ofrecen y cómo utilizarlos.

Ediciones y licencia

Este documento está disponible actualmente en ediciones en formatos HTML y PDF. La primera se puede visitar en el sitio web del proyecto, mientras que la segunda se puede descargar desde ull-esit-sistemas-operativos/ssoo-apuntes.

Código de los ejemplos

En algunos capítulos se enlazan programas de ejemplo para ilustrar en mayor detalle los conceptos tratados. Todos los ejemplos están disponibles en el repositorio ull-esit-sistemas-operativos/ssoo-ejemplos, de donde se pueden descargar.

Para compilar los ejemplos, es necesario disponer de herramientas de desarrollo para C y C++. Por ejemplo, en la distribución Debian de GNU/Linux y derivadas —como Ubuntu o Linux Mint— basta con tener instalados los paquetes **build-essential** y **cmake**. Mientras que en Microsoft Windows hacen falta las **Visual Studio Build Tools**.

Para compilar es necesario hacer lo siguiente desde la línea de comandos:

1. Ir al directorio raíz del repositorio descargado y descomprimido.
2. Ejecutar `cmake -B build` para configurar el proyecto.
3. Ejecutar `cmake --build build` para compilar los ejemplos.

En Microsoft Windows estos comandos deben ejecutarse desde la consola de **Developer Command Prompt**.

En cada sistema solo se compilarán los ejemplos compatibles, que se guardarán en el directorio `build/bin/`, desde donde se pueden ejecutar para probarlos.

El código fuente de los ejemplos está en el directorio `src/`, dentro del subdirectorio numerado con el capítulo correspondiente.

Parte I: Introducción

Antes de profundizar en el funcionamiento de los sistemas operativos modernos, es conveniente definir cuáles son las responsabilidades de un sistema operativo en términos generales. Sin embargo, veremos que esta cuestión no tiene una respuesta sencilla. Las responsabilidades de cualquier sistema operativo dependen del tipo de sistema informático para el que se ha diseñado; por lo que también estudiaremos los diferentes tipos de sistemas informáticos, sus características y cómo estas afectan a los sistemas operativos que se utilizan con ellos.

Finalmente, haremos un breve resumen de la historia de los sistemas operativos. Muchas de las características de los sistemas operativos actuales están condicionadas por decisiones de diseño tomadas al desarrollar sistemas predecesores.

Chapter 1. ¿Qué es un sistema operativo?

Tiempo de lectura: 7 minutos

¿Qué es un sistema operativo? ¿cuáles son sus responsabilidades en el contexto de un sistema informático? ¿cómo cumple con ellas? Estas son algunas de las cuestiones que responderemos en este capítulo. Aunque, cómo veremos, no son preguntas sencillas de responder.

1.1. Definición de sistema operativo

En general no existe una definición universal de lo qué es un **sistema operativo**, aunque si muchas propuestas de diferentes autores:

- Hay quien considera que simplemente es una cuestión del mercado: «lo que nos venden cuando llegamos a una tienda y pedimos un sistema operativo».

  En realidad esta definición no es muy precisa, puesto que las características incluidas pueden variar enormemente de un sistema a otro. Por ejemplo, algunos sistemas operativos apenas alcanzan el megabyte de espacio, careciendo incluso de las aplicaciones más básicas, mientras que otros ocupan gigabytes de espacio, incluyen una interfaz gráfica basada en ventanas y las aplicaciones más comunes que cualquier usuario puede necesitar.

> Aunque pueda parecer lo contrario, la cuestión de qué componentes son parte o no de un sistema operativo no es trivial. Por ejemplo, Microsoft y el Departamento de Justicia de los Estados Unidos se enfrentaron en 1998 por la inclusión del navegador Internet Explorer como parte del sistema operativo Microsoft Windows.
>
> Microsoft afirmaba que ambos productos eran realmente uno solo y que su unión fue el resultado de un proceso de innovación. Mientras tanto, la otra parte alegaba que el navegador era un producto distinto y separado, que no formaba parte del sistema operativo y que todo el asunto restringía la libre competencia en el mercado de los navegadores.
>
> Seguramente en 1998 los argumentos del Departamento de Justicia de los Estados Unidos tenían mucho sentido, ¿pero qué ocurriría si se plantea este mismo asunto en la actualidad?. ¿Concibes que tu móvil o tu ordenador no trajeran de serie un navegador?
>
> Para más información, véase «Caso Estados Unidos contra Microsoft — Wikipedia».

- Una definición mucho más común es que el sistema operativo es «aquel programa que se ejecuta continuamente en el ordenador» —lo que denominamos comúnmente como ***kernel*** o **núcleo** del sistema— siendo el resto: programas del sistema y aplicaciones.

  Sin embargo, en algunos casos esta definición excluye como parte del sistema operativo algunos servicios que intuitivamente solemos considerar dentro del mismo. Por ejemplo, si aplicamos

esta definición a los sistemas operativos de estructura microkernel, no podríamos decir que servicios básicos como la comunicación en red, los sistemas de archivos y la gestión de la memoria son parte del sistema operativo.

Como veremos en el Apartado 8.3, en los sistemas operativos *microkernel* la funcionalidad implementada en el núcleo del sistema es la mínima necesaria. Por lo tanto, según la definición anterior, muchos de los componentes y servicios básicos que damos por supuestos a un sistema operativo no formarían parte del mismo en ese tipo de sistemas.

1.2. Funciones del sistema operativo

Por lo que hemos visto hasta ahora, parece evidente que no es sencillo definir lo que «es» un sistema operativo. Sin embargo, es posible que tengamos más suerte definiéndolo a través de lo que «hace». Es decir, describiendo sus funciones dentro de un sistema informático cualquiera.

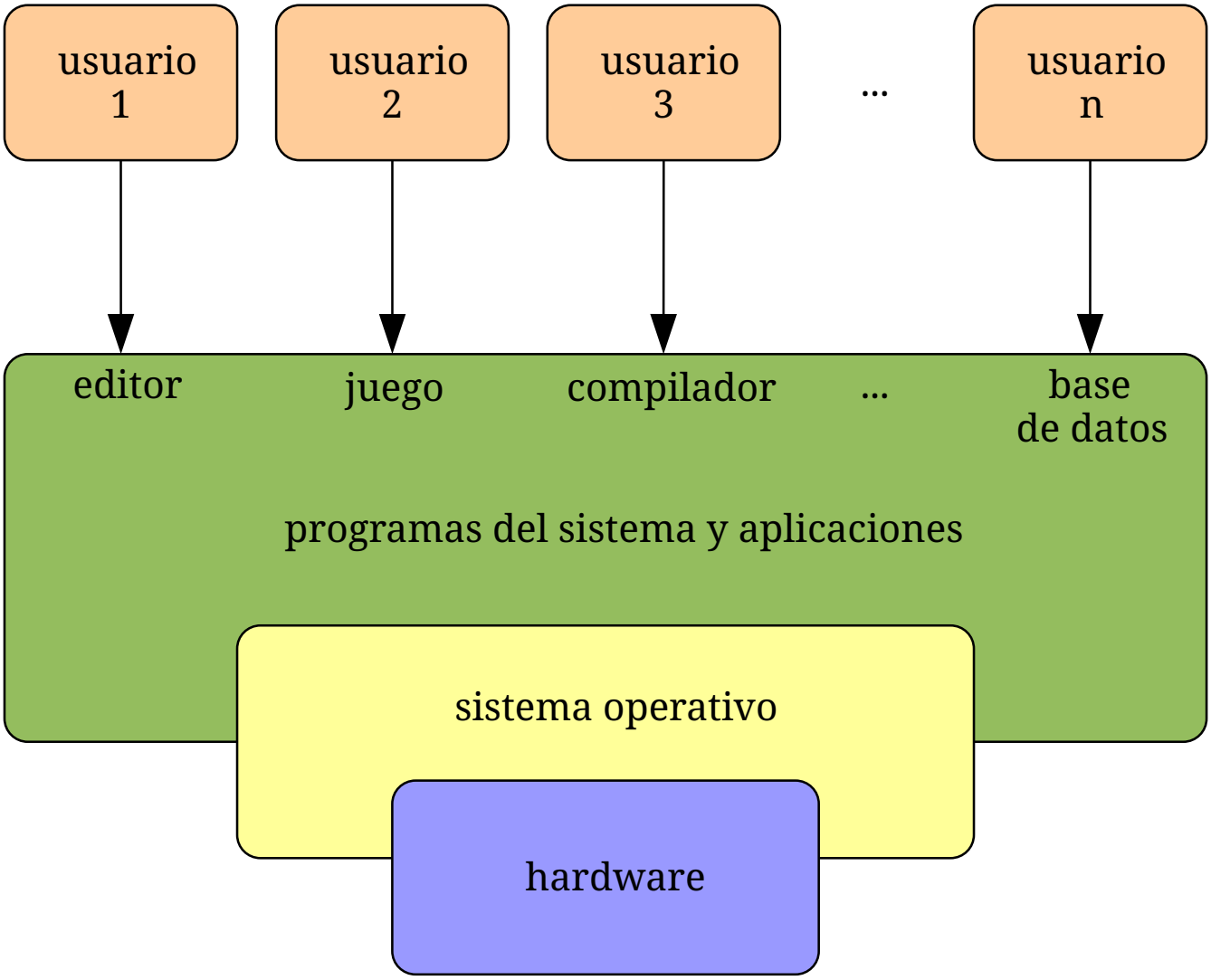


Figura 1. Vista abstracta de los componentes de un sistema informático.

Un **sistema informático** puede ser dividido, *grosso modo*, en cuatro componentes: el hardware, los usuarios, los programas de aplicación y el sistema operativo (véase la Figura 1):

- **Programas de aplicación**. El objetivo fundamental de cualquier sistema informático es

ejecutar programas para resolver los problemas informáticos de los usuarios. Con ese objetivo se construye su hardware y se desarrollan los programas de aplicación —procesadores de textos, hojas de cálculo, compiladores, navegadores de Internet, etc.— que usan los usuarios para resolver sus problemas.

- **Hardware**. El hardware —la CPU, la memoria, los dispositivos de entrada salida, etc.— proporcionan los recursos computacionales del sistema informático. Los programas de aplicación necesitan usar estos recursos computacionales para resolver los problemas informáticos de los usuarios.
- **Sistema operativo**. En un sistema informático las aplicaciones necesitan realizar operaciones comunes, como acceder a los dispositivos de E/S o reservar porciones de la memoria. En lugar de que cada aplicación intente hacerlo por su cuenta, es mucho más sencillo que estas operaciones comunes estén centralizadas en el sistema operativo.

Por lo tanto, el sistema operativo controla, coordina el acceso y asigna los recursos computacionales del hardware a los distintos programas de aplicación.

En realidad esta es solo una de las dos perspectivas desde las que se pueden analizar las funciones del sistema operativo. Es la denominada como: **perspectiva del sistema informático**, mientras que la otra es la **perspectiva del usuario**.

1.2.1. Perspectiva del sistema informático

Un sistema informático tiene múltiples recursos hardware, como son: tiempo de CPU, espacio de memoria, espacio de almacenamiento de archivos, dispositivos de E/S, etc. También tiene recursos software ofrecidos por algunos programas que se ejecutan en el sistema, como son: servicios de red, servicios de impresión, seguridad, etc.—. Estos recursos los necesitan los programas de aplicación para resolver los problemas informáticos de los usuarios.

Dentro del sistema informático, el sistema operativo es el programa más íntimamente relacionado con el hardware y su función es gestionar los recursos hardware y software disponibles, asignarlos a los diferentes programas, resolver los conflictos en las peticiones y hacer que el sistema opere eficientemente para resolver los problemas de los usuarios.

Además, el sistema operativo es el programa encargado del control de la ejecución de los programas de los usuarios, por lo que tiene la tarea de prevenir errores y el uso inadecuado del ordenador.

En resumen, desde la perspectiva del sistema informático, las funciones del **sistema operativo** son:

- Gestionar los recursos computacionales del sistema informático.
- Controlar la ejecución de los programas de usuario y el acceso a los dispositivos de E/S.

Un **sistema operativo**:

- No hace un trabajo directamente útil para los usuarios.
- Pero proporciona un entorno adecuado para que los programas de aplicación lo hagan.

> Los sistemas operativos existen porque es más sencillo crear sistemas informáticos útiles para los usuarios con ellos que sin ellos.

1.2.2. Perspectiva del usuario

Si intentamos definir las funciones del sistema operativo desde nuestra experiencia como usuarios, seguramente haríamos referencia a la interfaz que nos proporciona para utilizar el sistema informático. Sin embargo, debemos tener en cuenta que la interfaz varía con el tipo de sistema, por lo que definir las funciones del sistema operativo desde la perspectiva del usuario es mucho más difícil.

Por ejemplo, los usuarios que se sientan frente a un sistema de escritorio disponen de: monitor, teclado, ratón y una unidad central. Estos sistemas se diseñan buscando la máxima productividad en equipos donde un usuario monopoliza todos los recursos; por lo que el sistema operativo se diseña considerando fundamentalmente la facilidad de uso, poniendo algo de atención en el rendimiento y nada en el aprovechamiento de los recursos.

Esto difiere mucho de otro tipo de sistema informático, donde múltiples usuarios se sientan frente a terminales conectadas a un gran ordenador central. Así todos los usuarios comparten los recursos del sistema informático y pueden intercambiar información entre sí. En este tipo de sistemas el sistema operativo maximiza el aprovechamiento de los recursos con el objeto de garantizar que toda la CPU, memoria y E/S son empleadas de forma eficiente y que ningún usuario utiliza más de lo que le corresponde. Obviamente, en este tipo de sistemas la facilidad de uso está en un segundo plano.

Otros sistemas operativos se diseñan para sistemas informáticos que tienen poca o ninguna interacción con los usuarios. Es, por ejemplo, el caso de los sistemas empotrados de los electrodomésticos.

Todos estos tipos de sistemas tienen interfaces muy diferentes, lo que dificulta obtener una definición única de sistema operativo desde la perspectiva del usuario.

En los tres casos los objetivos con los que se diseña el sistema operativo son opuestos, por lo que seguramente sea diferente «lo que tiene que hacer» cada sistema operativo para alcanzarlos. Sin embargo, en los tres casos el sistema operativo es el responsable de la gestión de los recursos computacionales y del control de los programas, funciones que definimos anteriormente desde la perspectiva del sistema informático y que no cambian de un tipo de sistema a otro.

Chapter 2. Tipos de sistemas operativos

Tiempo de lectura: 23 minutos

Ahora que sabemos que todos los sistemas operativos hacen lo mismo, pero que el «cómo» lo hacen difiere de un tipo de sistema informático a otro, vamos a ver los tipos de sistemas informáticos, las características de los sistemas operativos que los gestionan y cómo han evolucionado a lo largo de la historia.

2.1. Mainframe

Los **ordenadores centrales** o ***mainframes*** fueron los primeros computadores utilizados en muchas aplicaciones comerciales y científicas. Se caracterizan no tanto por la potencia de su CPU como por su: gran capacidad de memoria, gran capacidad de almacenamiento secundario, gran cantidad de dispositivos de E/S y rapidez de estos y alta fiabilidad.

Los *mainframes* pueden funcionar durante años sin problemas ni interrupciones y las reparaciones se realizan sin detener su funcionamiento.

La mayor diferencia entre los superordenadores y los *mainframes* está en que los primeros se centran en resolver problemas limitados por la velocidad de cálculo —lo cual requiere miles de CPU de alto rendimiento— mientras que los segundos se centran en la fiabilidad y en problemas limitados por la E/S —por lo que los *mainframes* suelen tener «solo» entre una y varias docenas de CPU—.

Para más información sobre los *mainframes*, véase «Ordenador central — Wikipedia».

Los *mainframes* aparecieron a finales de la década de los 50 del siglo pasado y han seguido evolucionando hasta la actualidad, por lo que dentro de este tipo de sistemas nos encontramos con varias categorías.

2.1.1. Sistemas de procesamiento por lotes

Los primeros *mainframes* eran enormes máquinas operadas desde una consola y conectados a lectores de tarjetas perforadas, dispositivos de cinta e impresoras.

Para imágenes y más información sobre las tarjetas perforadas, véase «Computer programming in the punched card era — Wikipedia».

El trabajo era preparado por cada programador —normalmente en tarjetas perforadas— y entregado al operador del sistema, que era quién tenía acceso al sistema y la responsabilidad de ejecutar los programas y devolver los resultados al programador correspondiente.

No había sistema operativo y el operador debía cargar y ejecutar cada programa de uno en uno.

Figura 2. Operadora en la consola de un mainframe IBM 705 — Fuente: IBM

Estos sistemas se convirtieron en **sistemas de procesamiento por lotes** o **sistemas en *batch*** cuando se comenzó a utilizar un pequeño programa —llamado **monitor del sistema**— cuya función era cargar y ejecutar sin interrupción un conjunto —o lote— de programas.

Para preparar los lotes, por lo general, el operador cargaba previamente en cinta magnética el conjunto de programas a partir de las tarjetas perforadas proporcionadas por los programadores. Para ello se utilizaba un lector de tarjetas autónomo, independiente del *mainframe*.

Figura 3. Organización de la memoria en sistemas de procesamiento por lotes.

El **monitor del sistema** es un predecesor de los sistemas operativos y tenía las siguientes características:

- Permanecía cargado durante todo el tiempo en la memoria del sistema (véase la Figura 3).
- Su única tarea era cargar y transferir automáticamente la ejecución de un programa al siguiente cuando el anterior terminaba.
- El mayor inconveniente de este tipo de sistemas era que la CPU permanecía mucho tiempo desocupada porque era —y sigue siendo— varios órdenes de magnitud más rápida que los dispositivos de E/S.

Cualquier programa necesita realizar operaciones de E/S para obtener los datos requeridos para sus cálculos —guardados en tarjetas perforadas y unidades de cinta o, si hablamos de hoy en día, en discos duros y memorias USB—. También necesita hacer operaciones de E/S para guardar o imprimir los resultados de esos cálculos.

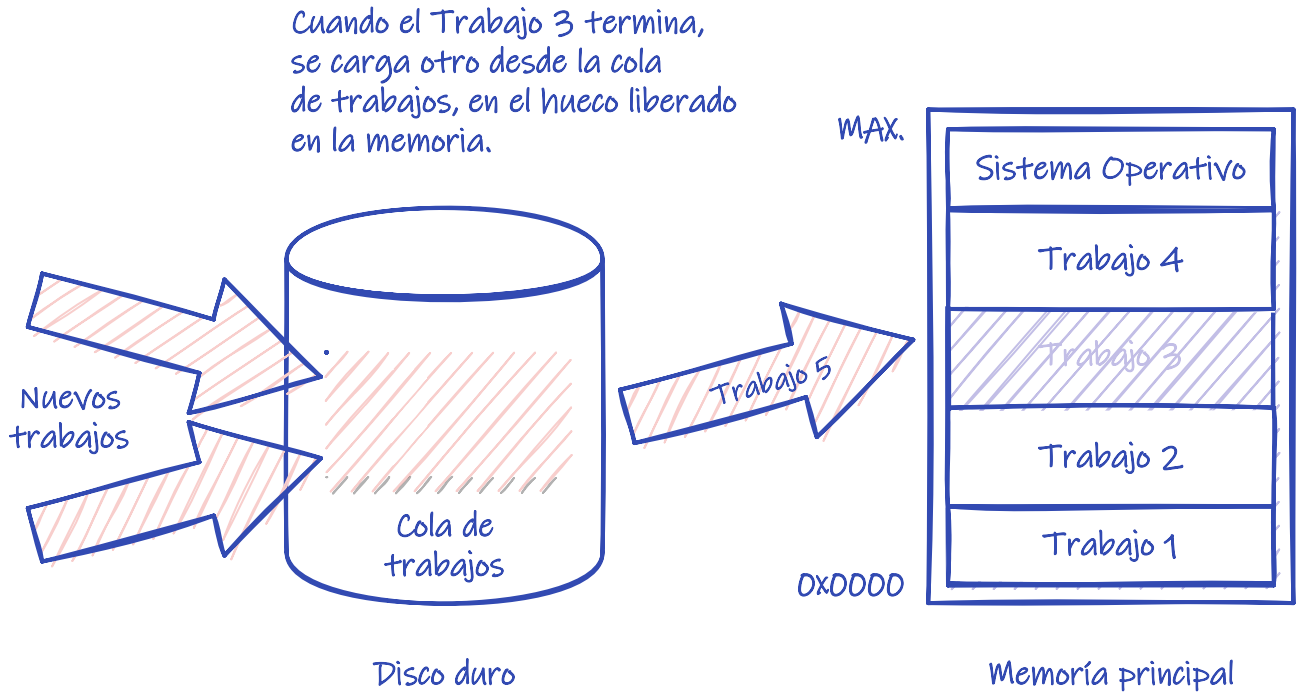
Si solo se puede ejecutar un programa la vez, cuando el programa solicita una operación de E/S, la CPU queda a la espera de que esta termine para continuar con la ejecución del programa, por lo que se pierde tiempo de CPU en no hacer nada. Este desaprovechamiento de la CPU es peor cuanto más rápida es la CPU respecto a los dispositivos de E/S.

2.1.2. Sistemas multiprogramados

La solución al inconveniente de los sistemas de procesamiento por lotes con la E/S fue que los programas no accedieran directamente al dispositivo de E/S, sino que, en su lugar, solicitaran la operación al **monitor del sistema** para que este la solicitara al hardware. Así el sistema operativo —como podemos comenzar a llamarlo— tiene la oportunidad de sustituir el programa en la CPU por otro, mientras la operación de E/S se completa.

Además, con la aparición de la tecnología de los discos magnéticos en la década de los 60 del siglo pasado, los trabajos de los programadores comenzaron a ser almacenados en discos, desde donde eran escogidos por el sistema operativo para su ejecución.

A estos sistemas se los llamó **multiprogramados**.



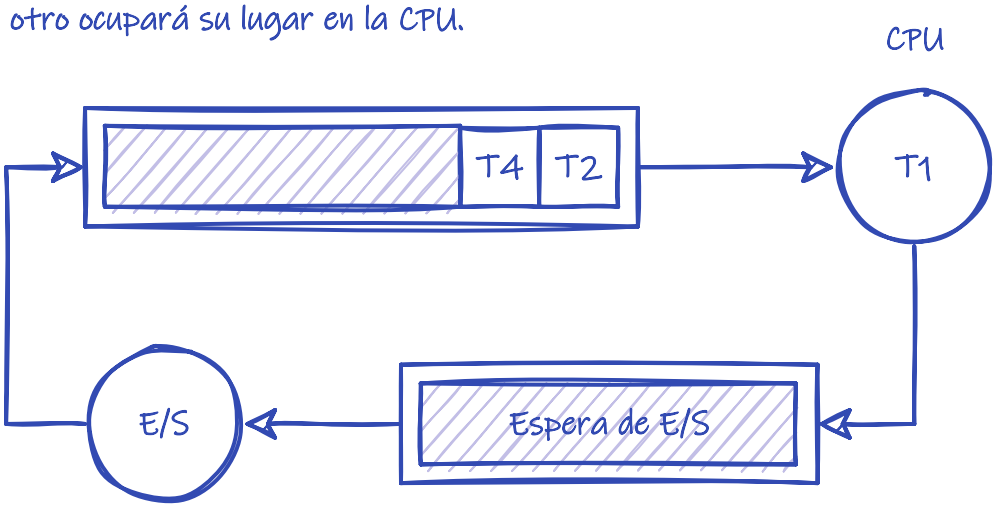


Figura 4. Gestión de trabajos en sistemas multiprogramados.

En los **sistemas multiprogramados** la ejecución de los trabajos funcionaba de la siguiente manera (véase la Figura 4):

1. En el disco magnético se almacenaba una cola donde se iban colocando todos los trabajos que tenían que ser ejecutados.
2. El sistema operativo cargaba varios trabajos en memoria del conjunto de trabajos en la cola en el disco magnético.
3. El sistema operativo cede la CPU a uno de los trabajos en memoria.
4. Cuando el trabajo en la CPU requería usar la E/S se lo pedía al sistema operativo. En lugar de mantener a la CPU ocupada inútilmente, el sistema operativo programaba la operación

de E/S, pero escogía otro trabajo de entre los que estaban en memoria y lo ejecutaba en la CPU.

Cuando la operación de E/S del anterior trabajo terminaba, el programa que ocupaba la CPU no era interrumpido, sino que debía esperar a una nueva oportunidad de ser escogido para ejecutarse en la CPU.

5. Cuando un programa en la CPU terminaba, sus recursos se liberaban, dejando memoria libre. Por lo tanto, el sistema operativo escogía un nuevo trabajo de la cola de trabajos en el disco magnético y lo cargaba en la memoria.

Todo este proceso se repetía mientras hubiera trabajos que ejecutar en la cola de trabajos en el disco.

Para operar de la forma descrita es necesario que el sistema operativo realice tres tareas esenciales:

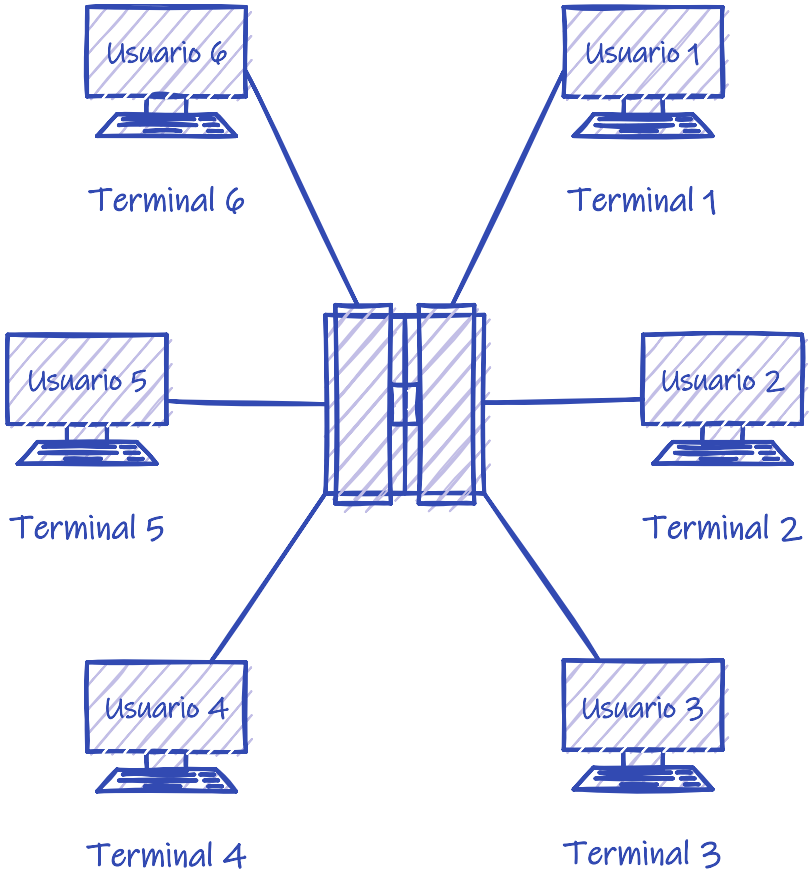
- La **planificación de trabajos**, cuya responsabilidad es seleccionar el siguiente trabajo que será cargado en la memoria principal para mantenerla llena.
- La **planificación de la CPU**, cuya responsabilidad es elegir el siguiente trabajo que será ejecutado en la CPU, de entre los disponibles en la memoria principal.
- La **gestión de la memoria**, cuya responsabilidad es repartir la memoria principal entre los trabajos alojados en la misma.

Un ejemplo de este tipo de sistemas operativos es el IBM OS/360, que fue lanzado en 1966 para utilizarlo en los *mainframes* IBM System/360 (véase el Apartado 3.2).

2.1.3. Sistemas de tiempo compartido

Los sistemas multiprogramados ofrecían un uso más eficiente de la CPU, pero no eran capaces de proporcionar interacción directa con los usuarios. Los programadores seguían teniendo que entregar los trabajos al operador y espera a que este les devolviera los resultados.

Los **sistemas de tiempo compartido** se desarrollaron tras observar que al dar acceso a un grupo de usuarios se podía conseguir un uso más eficiente del sistema, en comparación a cuando solo podía ser utilizado por un usuario a la vez. Esto es debido a que, generalmente, un usuario introduce información de forma continua para luego detenerse durante largos periodos de tiempo, mientras que en un grupo de usuarios, las pausas de uno de ellos se pueden llenar con la actividad de los otros.



No tienen cola de trabajo. Los usuarios indican interactivamente el trabajo que quieren ejecutar haciendo uso de una terminal.

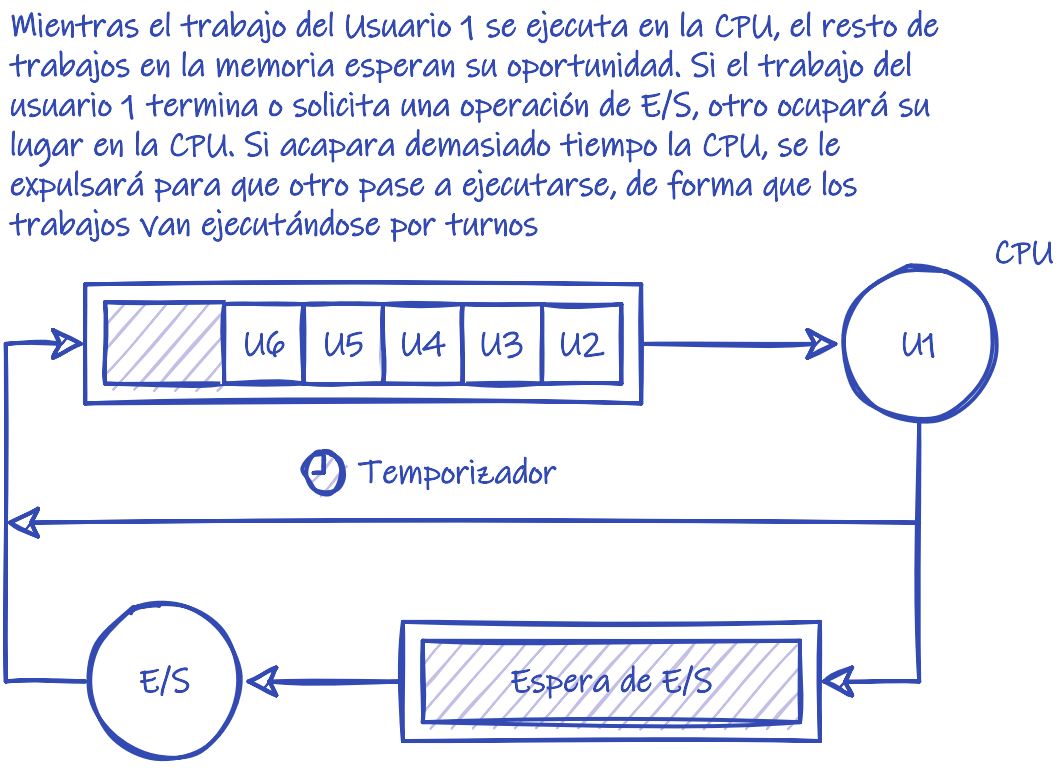


Figura 5. Gestión de trabajos en sistemas de tiempo compartido.

Los **sistemas de tiempo compartido** se caracterizaban por:

- Tener **terminales**, es decir, hardware especializado en hacer de interfaz directa entre los usuarios y el sistema. A través de estas terminales los usuarios podían enviar comandos al

sistema e interactuar con sus trabajos. Podía haber múltiples usuarios al mismo tiempo, pero cada uno solo podía tener un trabajo en ejecución a la vez.

- Usar la **multiprogramación** para tener varios trabajos en la memoria principal al mismo tiempo e intercambiar el trabajo en la CPU cuando este solicitaba una operación de E/S, como ya se venía haciendo en los **sistemas multiprogramados** para hacer un uso más eficiente de la CPU.
- Repartir el tiempo de CPU entre usuarios. El sistema operativo asignaba un tiempo de CPU a cada usuario —denominado **ventana de tiempo** o **cuanto** de CPU—. Cuando este tiempo se agotaba, el sistema intercambiaba el trabajo en la CPU por el de otro usuario en el sistema. La ventana de tiempo era extremadamente pequeña, dando a cada usuario la impresión de que su trabajo nunca se detenía, como si dispusiera de la CPU en exclusiva.

Los sistemas que, como los de tiempo compartido, pueden ser utilizados por varios usuarios simultáneamente se denominan sistemas **multiusuario**.

En los primeros sistemas se usaban **terminales** electromecánicos con un teclado y una impresora, como el Teletype Model 3 (1963). Posteriormente llegaron los terminales electrónicos, que usaban un monitor en lugar de una impresora, como el IBM 3270. En cualquier caso solo disponían del hardware necesario para realizar la tarea de conectar a los usuarios con el ordenador central.

Estos terminales no deben confundirse con las terminales por software que traen algunos sistemas operativos modernos. Las terminales por software o *terminales virtuales* se programan para emular las especificaciones de alguna versión de esas terminales físicas antiguas que hemos comentado.

Los sistemas de tiempo compartido significaron un salto importante en complejidad por diversas razones:

- Como varios trabajos están en la memoria principal al mismo tiempo, el sistema operativo requiere mecanismos de **gestión de la memoria** y **protección**.
- Para tener un tiempo de respuesta razonable, los trabajos deben estar cargados en la memoria principal. Para que quepan más trabajos de los usuarios en la memoria, el sistema operativo debe utilizar técnicas de **memoria virtual** para ejecutar trabajos que no están completamente cargados en la memoria principal.
- Como la CPU debe ser compartida entre todos los trabajos, el sistema operativo necesita mecanismos de **planificación de la CPU**.
- Como varios trabajos pueden tener la necesidad de cooperar y que su ejecución siga cierto orden, el sistema operativo debe proporcionar mecanismos de **sincronización** y **comunicación**.
- Como el sistema debe disponer de un **sistema de archivos** para repartir el espacio en disco y facilitar a los usuarios el acceso y gestión de sus datos, el sistema operativo necesita un componente de **gestión de discos**.

Las primeras versiones de UNIX —lanzado por primera vez en 1970— el sistema operativo VMS

—desarrollado en 1978— para los VAX de Digital Equipment Corportation y el IBM OS/400 —introducido en 1988— utilizado en los minicomputadoras AS/400, son algunos ejemplos de sistemas operativos de tiempo compartido (véase el Apartado 3.3).

Estrictamente hablando, el término **sistemas de tiempo compartido** hace referencia a estos *mainframes* desarrollados a partir de principios de la década de 1970. Así que no es común utilizarlo con *mainframes* modernos.

Los *mainframes* modernos permiten a un mismo usuario ejecutar varios trabajos al mismo tiempo, repartiendo el tiempo de CPU entre todos los trabajo en el sistema y no solo entre los usuarios. Y lo mismo ocurre en la mayor parte de los sistemas operativos de propósito general actuales —utilizados en ordenadores de escritorio, servidores, portátiles y dispositivos móviles— que con el tiempo han copiado muchas características de los **sistemas de tiempo compartido**. Por eso el término actual es **sistema multitarea**, que es mucho más general.

La **multitarea** es un método para tener varios procesos en memoria y ejecutarlos «al mismo tiempo». Generalmente requiere de técnicas de multiprogramación, como las empleadas por los antiguos **sistemas multiprogramados**, y de reparto del tiempo de CPU, como ocurre en los antiguos **sistemas de tiempo compartido**. Por eso se puede decir que ambos tipos de sistemas *mainframe* eran **sistemas multitarea**. Al igual que lo son los *mainframes* modernos y muchos sistemas operativos actuales de escritorio y de dispositivos móviles.

2.2. Sistemas de escritorio

En la década de los 70 del siglo pasado también aparecieron las primeras CPU en microprocesadores y con estas llegaron las **microcomputadoras** o **microordenadores**. Las primeras **microcomputadoras** no incluían teclado ni monitor y se programaban usando interruptores y ledes ubicados en el frontal de la unidad. Pero en torno a 1977 apareció la segunda generación de **microcomputadoras**, que sí incluían estos periféricos de E/S, por lo que eran más fáciles de usar que sus predecesoras. Entonces comenzaron a recibir el nombre de *ordenadores domésticos* y de su mano llegaron los primeros **sistemas operativos de escritorio**.

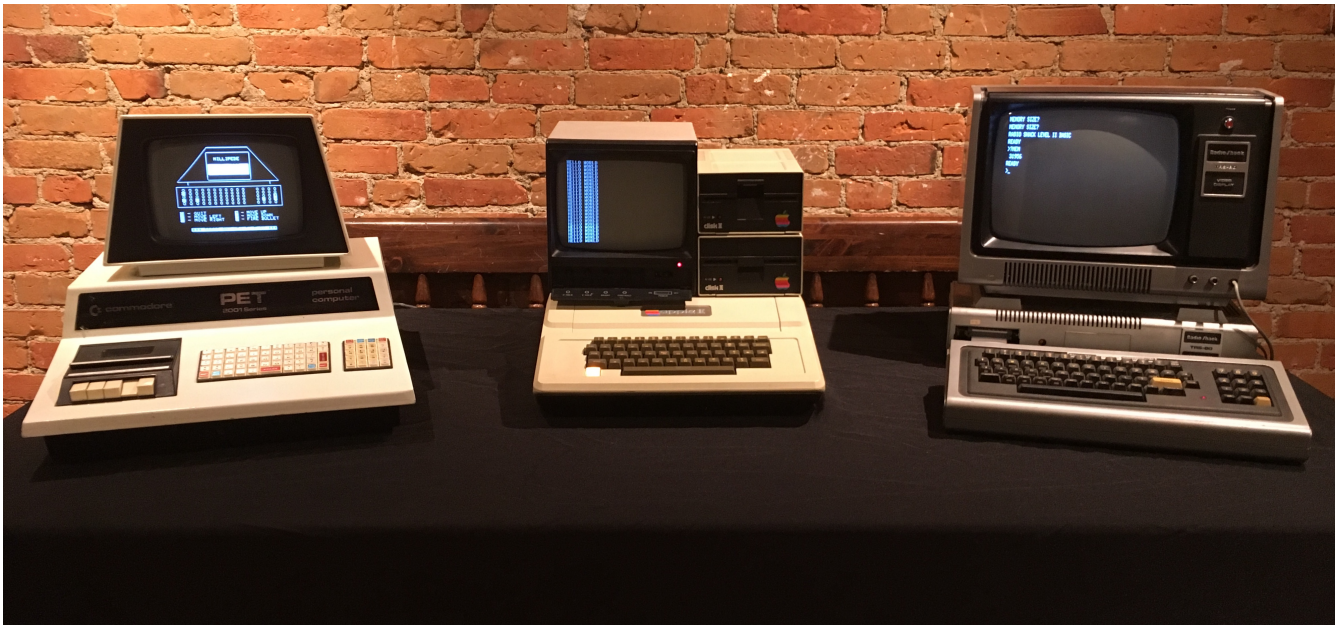


Figura 6. Los tres ordenadores que la revista Byte denominó como la "Trinidad de 1977" de la computación doméstica: el Commodore PET 2001, el Apple II y el TRS-80 Model I — Fuente: Wikipedia

Los *mainframes* y las minicomputadoras de la época siguieron siendo los ordenadores corporativos por excelencia, ya que eran mucho más grandes y potentes, y también costosos.

El término en desuso **minicomputadora** o **miniordenador** hace referencia a máquinas multiusuario de rango medio, entre los *mainframes* y los ordenadores domésticos.

Los primeros **sistemas operativos de escritorio** eran muy básicos. Por ejemplo, en un sistema diseñado para ser utilizado por un único usuario no tiene sentido implementar un sistema de archivos con permisos. Así que, los primeros sistemas operativos de escritorio carecían de esta característica que, sin embargo, ya existía en los sistemas de tiempo compartido de la época. De la misma manera, carecían de otros mecanismos de protección y no eran ni multiusuario ni multitarea.

Pese a estas diferencias, los **sistemas operativos de escritorio** se han beneficiado del desarrollo de los sistemas operativos para *mainframes*. Los sistemas de escritorio actuales son **multiusuario** y **multitarea**; incluyen sistemas de archivos con permisos, autenticación y mecanismos de protección de la memoria —como medidas para proteger los datos de los usuarios— y han incorporado muchas otras características de los sistemas operativos para *mainframe*.

Aunque con el tiempo los sistemas de escritorio han ido adquiriendo características desarrolladas en los *mainframes*, no debemos olvidar que ambos tipos de sistemas se siguen diseñando con objetivos diferentes. Mientras que en los *mainframes* se persigue maximizar la fiabilidad y utilización eficiente de los recursos, en los sistemas de escritorio se maximiza la facilidad de uso y el tiempo de respuesta al usuario, poniendo algo de atención al rendimiento.

Los **sistemas operativos de escritorio** modernos ya nos son «solo de escritorio» ni se ejecutan únicamente en ordenadores domésticos. Se utilizan en un altísimo porcentaje en servidores, superordenadores y hasta en dispositivos móviles. Por eso, en la actualidad, el término

sistema operativo de propósito general es mucho más adecuado.

El **tiempo de respuesta** al usuario se puede considerar como el intervalo de tiempo entre un comando de un usuario —por ejemplo un clic— y la respuesta del sistema a dicho comando. En ocasiones este tiempo se minimiza a costa de un uso menos eficiente de los recursos del sistema, por lo que no es un objetivo deseable para diseñar un *mainframe*. Para más información, véase el Apartado 14.3.

Son muchos los ejemplos de sistemas operativos en esta categoría. Van desde CP/M —lanzado en 1977— hasta los actuales GNU/Linux, Microsoft Windows y Apple macOS, pasando por MS-DOS, IBM OS/2 y todas las versiones anteriores de Microsoft Windows (véase el Apartado 3.4).

2.3. Sistemas de mano

Con el nombre genérico de **sistemas de mano** —del inglés *handheld*— hacemos referencia a las *tablets*, *smartphones*, lectores de libros electrónicos y otro sistemas móviles y portátiles. Los desarrolladores de aplicaciones y sistemas de mano deben enfrentarse a diversos desafíos, originados por el tamaño limitado de los dispositivos y la alimentación mediante el uso de baterías. Debido a esas limitaciones, muchos sistemas de mano tienen poca cantidad de memoria, procesadores lentos —en comparación con sus equivalentes de escritorio— y pantallas más pequeñas.

En el diseño del sistema operativo suele primar la facilidad de uso y buscar un buen equilibrio entre rendimiento y tiempo de vida de la batería.

2.4. Sistemas multiprocesador

Un **sistema multiprocesador** es aquel ordenador hay procesadores interconectados que comparten el bus del sistema, el reloj y, en ocasiones la memoria, y los periféricos.

Hace años esto solo se daba en sistemas con varias CPU, lo que era relativamente común en servidores y sistemas de alto rendimiento para trabajos técnicos o científicos. Sin embargo, en la actualidad cualquier dispositivo digital u ordenador doméstico puede tener una CPU con múltiples núcleos, lo que los convierte en sistemas multiprocesador.

Las principales ventajas de estos sistemas son:

- **Aumentan la cantidad de trabajo realizado**. A mayor número de procesadores, mayor cantidad de trabajo puede realizar el sistema. Sin embargo debemos de tener en cuenta que un sistema con *N* CPU no es un sistema *N* veces más rápido. Cuando varios procesadores cooperan para realizar una tarea, existe cierta pérdida de rendimiento debida a los mecanismos de sincronización requeridos para controlar el acceso a los recursos compartidos por los procesadores.
- **Economía de escala**. Un sistema multiprocesador puede costar menos que múltiples sistemas monoprocesadores conectados para hacer un trabajo equivalente, porque comparten periféricos, almacenamiento, alimentación, etc.

- **Alta disponibilidad**. Con el hardware adecuado el sistema puede ser tolerante al fallo de uno de los procesadores. En caso de fallo el sistema no se detendría, pero sí trabajaría más despacio.

En la actualidad existen dos tipos de sistemas multiprocesador:

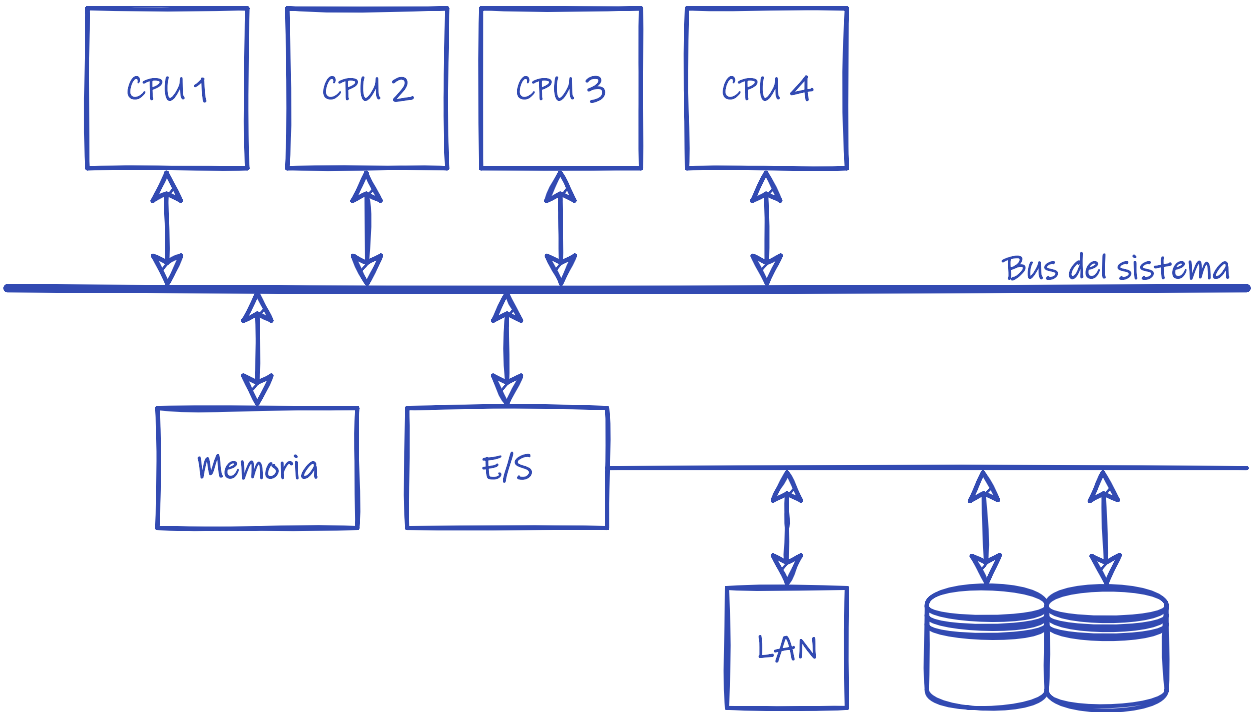


Figura 7. Arquitectura de un sistema de multiprocesamiento simétrico.

- En los **sistemas de multiprocesamiento simétrico** o **SMP** (*Symmetric Multiprocessing*) todos los procesadores son iguales. Todos comparten los mismos recursos, pueden acceder a los mismos dispositivos (véase la Figura 7) y cada uno ejecuta una copia del núcleo del sistema operativo. El sistema operativo debe haber sido diseñado para saber repartir el trabajo entre los procesadores y compartir adecuadamente entre tareas y procesadores el resto de recursos del sistema. Casi todos los sistemas multiprocesador modernos son de este tipo.
- En los **sistemas de multiprocesamiento asimétrico** o **AMP** (*Asymmetric Multiprocessing*) hay un procesador principal y varios secundarios a quienes el principal planifica y entrega las tareas que deben ejecutar. En ocasiones los procesadores secundarios se distinguen del principal por haber sido diseñados para realizar algún tipo concreto de tareas de forma muy eficiente o por estar conectadas a hardware especial. Ejemplos de esto son las GPU, que no son sino procesadores diseñados específicamente para el procesamiento de gráficos, o las CPU de E/S conectadas a discos duros para gestionarlos de forma más eficiente.

Un ejemplo bastante ilustrativo es el de Cell, la CPU de PlayStation 3. Tenía un núcleo principal de propósito general y 8 núcleos optimizados para ejecutar, de forma muy eficiente, operaciones vectoriales. Con la ayuda del sistema operativo, los programas debían envíar tareas matemáticamente intensivas a los procesadores secundarios, si querían extraer el máximo provecho de la arquitectura.

Desarrollar para un sistema así es más complejo. Por lo que, aunque sobre el papel esta arquitectura ofrecía gran rendimiento, aprovecharlo era un verdadero reto para los desarrolladores.

2.5. Sistemas distribuidos

En la actualidad es común el uso de redes para interconectar ordenadores individuales —por ejemplo Internet o la red de área local de una oficina— cada uno equipado con su procesador, su memoria, sus dispositivos de almacenamiento, su fuente de alimentación, etc. En las redes de ordenadores los procesadores de dichos ordenadores se comunican con otros procesadores a través de líneas de comunicación, como: redes Ethernet, líneas telefónicas o wifi. Estos sistemas son comúnmente denominados **sistemas distribuidos**.

Sin entrar en detalles, los sistemas distribuidos pueden ser clasificados en **sistemas cliente-servidor** y **sistemas de redes entre iguales**.

2.5.1. Sistemas cliente-servidor

En los **sistemas cliente-servidor** existen ordenadores que actúan como **servidores** encargados de satisfacer las peticiones generadas por otros ordenadores que actúan como **clientes**.

Este tipo de sistemas han sustituido, en un gran número de casos, a los terminales conectados a *mainframes*, debido a que los sistemas de escritorio son cada vez más potentes y baratos. Concretamente:

- Los terminales han sido sustituidos por sistemas de escritorio que, al disponer de más recursos, son capaces de realizar muchas de las funcionalidades que anteriormente eran manejadas directamente por los *mainframes*.
- Al mismo tiempo estos *mainframes* se han reemplazado por servidores, no muy diferentes a los sistemas de escritorios, pero preparados para atender las peticiones de sus clientes.

Ejemplos de este tipo de sistemas son los servidores de base de datos, que responden a las consultas SQL de los clientes, o los servidores de archivos, que proporcionan una interfaz de sistema de archivos con la que los clientes pueden crear, leer, escribir y borrar archivos en el servidor; de forma similar a como si estuvieran almacenados localmente en el propio cliente.

2.5.2. Sistemas de redes entre iguales

En los **sistemas de redes entre iguales** o **P2P** (*peer-to-peer*) clientes y servidores no se distinguen los unos de los otros. Todos los nodos del sistema son iguales y cada uno puede actuar como cliente o servidor, dependiendo de cuándo piden o proporcionan un servicio.

La ventaja fundamental de este tipo de sistemas es que en los sistemas cliente-servidor el servidor puede ser el cuello de botella del rendimiento, pero en los sistemas de redes entre iguales la carga se distribuye entre todos los nodos de la red. Ejemplos de este tipo de sistemas son las redes BitTorrent y Bitcoin.

Un servidor puede ser el cuello de botella no solo por su potencia sino también por el ancho de banda de su conexión a la red. La potencia del servidor es lo de menos cuando se intenta distribuir en Internet archivos de gran tamaño —por ejemplo imágenes de CD o DVD— pues el problema es que varias descargas simultáneas pueden consumir todo el ancho de banda del servidor durante largos periodos de

tiempo.

2.5.3. Sistemas operativos para sistemas distribuidos

Desde el punto de vista de los sistemas operativos para sistemas distribuidos es posible hacer la siguiente distinción:

- Los **sistemas operativos de red** ofrecen a las aplicaciones que corren sobre ellos servicios de acceso a redes de ordenadores. Por ejemplo, implementan algún mecanismo que permita a diferentes procesos en diferentes ordenadores enviar y recibir mensajes. Además suelen incorporar la opción de proporcionar algunos servicios de red, como la compartición de archivos y dispositivos con otros equipos de la misma red.

  Los ordenadores con sistemas operativos de red son autónomos. Simplemente es que gracias al sistema operativo de red, conocen la existencia de la red y saben usarla para comunicarse con otros ordenadores de la misma.

  Este tipo de sistemas operativos son los más utilizados en los tipos de sistemas distribuidos comentados anteriormente. En la actualidad, la inmensa mayoría de sistemas de escritorio y dispositivos de mano utilizan sistemas operativos de red.

- Los **sistemas operativos distribuidos** crean en el usuario la ilusión de que está en un único ordenador, aunque en realidad el sistema operativo controla todos los ordenadores de la red, dando al usuario acceso transparente a los recursos en todos los equipos de la misma.

  Con este tipo de sistemas operativos el usuario no sabe en qué ordenador se ejecutan sus procesos, donde se almacenan sus archivos, ni qué equipo tiene conectado los distintos periféricos a los que tiene acceso.

Un ejemplo de sistema operativo distribuido es Amoeba, un sistema operativo distribuido de investigación escrito por Andrew S. Tanenbaum en Vrije Universiteit. Para más información, véase el sitio web de Amoeba.

2.6. Sistemas en clúster

Como los sistemas distribuidos, los **sistemas en clúster** interconectar ordenadores individuales. Sin embargo, generalmente se acepta que los **sistemas en clúster** comparten el almacenamiento y estén conectados por medio de una red local, condiciones que no tienen por qué darse en los sistemas distribuidos.

Los **sistemas en clúster** se utilizan para:

- **Obtener servicios con alta disponibilidad**. Para ello un nodo del clúster puede estar ejecutando un servicio mientras otro nodo lo monitoriza. En caso de fallo en el nodo que da el servicio, el que lo monitoriza lo sustituye.

  Si es necesario proporcionar varios servicios, el mecanismo anterior se puede extender repartiendo los servicios entre dos o más nodos y haciendo que se monitoricen entre ellos.

- **Computación de alto rendimiento** o **HPC**. En este caso todos los nodos se utilizan para dar un mismo servicio. Un nodo especial, denominado balanceador de carga, tiene la responsabilidad de repartir el trabajo entre los nodos.

  Este tipo de **sistemas en clúster** se utiliza para realizar trabajos de cálculo muy pesados, como simulaciones —por ejemplo simulación meteorológica, nuclear o de gestión hospitalaria— o romper sistemas de cifrado.

También es muy utilizado en servidores de Internet —como servidores web, correo electrónico o de mensajería instantánea— o servidores de base de datos que deben dar servicio a una gran cantidad de clientes simultáneamente. En estos casos el balanceador de carga realiza su trabajo repartiendo las conexiones de los usuarios entre los servidores del clúster.

2.7. Sistemas de tiempo real

Los **sistemas de tiempo real** se utilizan cuando existen requerimientos estrictos de tiempo en la ejecución de ciertas tareas o en el procesamiento de flujos de datos.

En general se usan frecuentemente en dispositivos de control donde, dentro de unos márgenes estrictos de tiempo, se deben tomar datos de uno o varios sensores, para analizarlos posteriormente y realizar, en consecuencia, alguna acción con algún mecanismo de control. Por ejemplo, se suelen utilizar en sistemas de control industrial, domótica, armamento, automoción —en la inyección electrónica de combustible, sistemas de frenado y de control de tracción— o en dispositivos médicos.

Los **sistemas de tiempo real** están muy relacionados con los **sistemas empotrados**. Estos últimos:

- Se diseñan para realizar tareas muy específicas. No son sistemas de propósito general sino de propósito específico.
- Sus sistemas operativos tienen características muy limitadas y no tienen que tener necesariamente una interfaz de usuario.
- Estos sistemas están tanto en el motor de los automóviles y los robots que los fabrican, como en reproductores de DVD, microondas o dispositivos de red.

Los **sistemas de tiempo real** pueden ser clasificados en **sistemas de tiempo real estricto** y **sistemas de tiempo real flexible**:

- Los **sistemas de tiempo real estricto** o **hard real-time** garantizan que las tareas serán realizadas dentro de unos márgenes estrictos de tiempo.

  Para ello, todas las situaciones imprevistas que puedan ocasionar retrasos en el funcionamiento del sistema operativo deben estar perfectamente delimitadas en tiempo. Por lo tanto, suelen carecer de memoria virtual y de otras abstracciones que aíslen al desarrollador del funcionamiento real del hardware, ya que introducen impredecibilidad.

  Los sistemas de tiempo real estricto no son compatibles con los sistemas de tiempo compartido.

- Los **sistemas de tiempo real flexible** o **soft real-time** son útiles cuando en un sistema operativo convencional hay tareas que tienen mayor importancia que el resto, por lo que deben ser realizadas con mayor prioridad.

  El tiempo real flexible no sirve cuando se tienen tareas con limitaciones precisas de tiempo, porque no hay manera de garantizar que dichas restricciones se van a cumplir. Sin embargo sí es útil para tareas relacionadas con la multimedia, la realidad virtual, los videojuegos, etc. y es compatible con la memoria virtual y otras características presentes en los sistemas de escritorio. Por eso la mayor parte de los sistemas de escritorio actuales soportan tareas de tiempo real flexible.

Chapter 3. Historia de los sistemas operativos

Tiempo de lectura: 19 minutos

La historia de los sistemas operativos se puede dividir en cinco grandes etapas o generaciones, obviamente conectadas con las generaciones de los ordenadores donde funcionaban.

3.1. 1ª Generación (1945-55)

En la primera generación de ordenadores no se utilizaban sistemas operativos.

Sus principales características son:

- Computadoras construidas con electrónica de válvulas de vacío.
- Sin sistema operativo.
- Sin lenguajes de programación. Se programaban directamente en lenguaje máquina.

Algunos ejemplos de ordenadores destacables fueron:

ENIAC (1945)

Se le considera el primer ordenador electrónico digital de propósito general, aunque existe cierta polémica sobre este punto. Lo cierto es que se construyeron otros ordenadores antes que este, pero o no eran de propósito general —como las famosas computadoras Colossus (1944), que fueron diseñadas para ayudar en criptoanálisis— o no eran electrónicos sino electromecánicos —como la computadora Z3 (1941), que usaba relés—.

No era un producto comercial sino un proyecto experimental de defensa que principalmente se diseñó y utilizó para calcular tablas de tiro de artillería destinadas al Laboratorio de Investigación Balística del Ejército de los Estados Unidos.

Z4 (1945) fue el primer ordenador digital comercial, pero era electro-mecánico.

IBM 701 (1953)

Fue el primer *mainframe* de la serie IBM 700, que a la larga se convertiría en un éxito de ventas. Utilizaba tubos de vacío y tarjetas perforadas.

El IBM 7090 —versión transistorizada del 709, que utilizaba válvulas de vacío, como todos los de la serie 700— y el posterior 7094, fueron usados por la NASA para los cálculos de control de las misiones de los programas espaciales Mercury y Gemini y durante la primera etapa del programa Apolo.

3.2. 2ª Generación (1955-64)

En la segunda generación de ordenadores los transistores reemplazaron a las válvulas de vacío.

En lo que respecta a los sistemas operativos:

- Aparecen los monitores del sistema, que se pueden considerar un predecesor de los sistemas operativos.
- Sistema de procesamiento por lotes.
- Se comienzan a utilizar lenguajes de programación, como: ensamblador, FORTRAN y COBOL.

GM-NAA I/O (*General Motors and North American Aviation Input/Output system*) fue el primer sistema operativo. Fue desarrollado por General Motors Research Laboratory en 1956 para el *mainframe* IBM 704 con el fin de automatizar la carga y ejecución de un nuevo trabajo una vez había terminado el anterior. Para su desarrollo se basaron en un monitor del sistema creado en 1955 por programadores de General Motors para el IBM 701.

Figura 8. Instalación de un mainframe IBM 702 — Fuente: Wikipedia

3.3. 3ª Generación (1965-1968)

En la tercera generación se comenzaron a utilizar los circuitos integrados, que fue una invención de finales de la década de 1950.

En lo que respecta a los sistemas operativos:

- Aparecen los sistemas operativos multiprogramados.
- Aparecen más lenguajes de programación.

El ejemplo más destacado de esta época es el IBM OS/360 Fue un sistema operativo desarrollado por IBM para su *mainframe* IBM System/360 (S/360) (véase la Figura 9). Su versión DOS/360 (*Disk Operating System/360*) fue el primer sistema operativo en hacer los discos magnéticos un requisito

para poder operar.

Figura 9. Instalación de un mainframe IBM System/360 — Fuente: IBM

Se anunció en 1964, pero fue lanzado en 1966, con un año de retraso respecto a la fecha prevista originalmente. Los motivos fundamentales fueron ciertos problemas de organización interna de la compañía y la falta de experiencia en proyectos de esa envergadura. Las previsiones iniciales eran de 1 millón de líneas de código y miles de componentes de software.

Algunos autores fechan los inicios de la ingeniería del software en la publicación del libro «The Mythical Man-Month: Essays on Software Engineering», escrito por Frederick Brooks y publicado en 1975. Frederick Brooks se basó en la experiencia adquirida mientras administraba el desarrollo del IBM OS/360, donde era jefe de proyecto.

3.4. 4ª Generación (1965-1980)

La cuarta generación abarca desde mediados de los años 60 hasta finales de la década de los 70. Respecto a los ordenadores, es el resultado del desarrollo de los microprocesadores.

En lo que respecta a los sistemas operativos:

- Aparecen los sistemas operativos de tiempo compartido.
- Aparecen los terminales, los programas interactivos y las máquinas virtuales.

A continuación veremos los ejemplos más representativos de esta época.

3.4.1. MULTICS

MULTICS fue anunciado en 1964, fruto de la colaboración entre el MIT, General Electrics y Bell Labs, como el primer sistema operativo de propósito general.

Figura 10. Mainframe GE-6180 con sistema MULTICS, en torno a 1976 en el MIT — Fuente: Multicians

Fue el primer sistema operativo en proporcionar un sistema de archivos jerárquico, intérprete de comandos implementado como programa de usuario, listas de control de acceso individuales para cada archivo y enlazado dinámico, entre otras características novedosas.

Además experimentó con eliminar la separación entre el espacio de direcciones de los procesos y los archivos. Es decir, como si todos los archivos estuvieran mapeados en memoria, permitiendo a los procesos acceder al contenido de los archivos directamente (véase el Apartado 17.3).

3.4.2. VM/CMS

VM/CMS es un sistema de IBM utilizado en los *mainframes* IBM System/360, System/370, System/390 y zSeries. VM es un hipervisor que se encarga de virtualizar el hardware para crear múltiples máquinas virtuales, dando la sensación de que cada una es un *mainframe* independiente.

Como sistema operativo de las máquinas virtuales, una opción común es CMS, un sistema interactivo y monousuario muy ligero, diseñado para operar fundamentalmente en una máquina virtual de VM. Gracias a VM/CMS, cada usuario tiene la sensación de trabajar en un sistema completamente independiente y seguro.

El desarrollo de VM/CMS comenzó en 1965 y la primera versión estuvo disponible a primeros de 1966. Las versiones actuales se denominan IBM z/VM.

3.4.3. UNIX

UNIX fue desarrollado originalmente por Bell Labs en 1970 para los sistemas PDP-11/20 (véase la Figura 11). La autoría del mismo se le atribuye a un grupo de programadores, liderados por Ken Thompson, que decidieron rehacer el trabajo de MULTICS, pero a menor escala, después de que Bell Labs abandonara el proyecto MULTICS en 1969. Inicialmente se llamó UNICS y fue desarrollado para los sistemas PDP-7.

Figura 11. Dennis Ritchie (de pie) y Ken Thompson (sentado) frente a un PDP-11 y sus dos terminales Teletype 33 — Fuente: Dennis Ritchie

La primera versión de UNIX fue implementada en ensamblador, como era común en la época. Posteriormente, Dennis Ritchie y Brian Kernighan diseñaron un nuevo lenguaje de programación llamado «C», especialmente pensado para que UNIX fuera escrito con él. Eso facilitó que UNIX pudiera ser portado a ordenadores diferentes. Además, gracias al lenguaje C, el código era más conciso y compacto, lo que se tradujo en que se pudieron desarrollar nuevas funcionalidades más rápidamente.

AT&T, la compañía matriz de Bell Labs, no podía competir en la industria de los ordenadores, por lo que puso el código fuente de UNIX a disposición de universidades, compañías privadas y del gobierno de los Estados Unidos. Eso aumentó su difusión y dio resultados inesperados. Por ejemplo, una de las variantes más importantes de UNIX fue BSD, desarrollada por la Universidad de California en Berkeley.

La versión 4.2BSD (*Berkeley Software Distribution*) de esta variante de UNIX fue la primera que incluyó la interfaz de *sockets* para facilitar la comunicación entre procesos a través de Internet y otras redes. Esta interfaz se ha convertido en estándar en prácticamente cualquier sistema operativo.

También implementó y ayudó a difundir el estándar de comunicaciones TCP/IP, base de la actual Internet. Muchos sistemas operativos actuales, tanto libres como privativos, utilizan código de UNIX BSD en sus implementaciones de los protocolos TCP/IP y de diversas utilidades de red.

En la actualidad se considera que hay dos grandes familias de UNIX y las distintas variantes pertenecen a una u otra en función del UNIX del que derivaron originalmente:

- La familia derivada de **AT&T UNIX System V**, en la que se incluyen sistemas operativos no libres, tales como: SCO OpenServer, Oracle/Sun Microsystems Solaris Operating Environment y SCO UnixWare.
- La familia derivada de **UNIX BSD**, en la que se incluyen sistemas libres como: FreeBSD, NetBSD, OpenBSD, Darwin y DragonFly BSD, entre muchos otros.

FreeBSD es el sistema base de algunos sistemas no libres. Por ejemplo, Darwin es el sistema operativo en el que se basan los sistemas operativos de Apple: macOS, IOS, watchOS, tvOS e iPadOS. A su vez Darwin utiliza múltiples elementos de FreeBSD (véase el Apartado 3.5.8).

Otro ejemplo destacable es Orbis OS —el sistema operativo de PlayStation 4— que también está basado en FreeBSD.

3.4.4. VMS

VMS es un sistema operativo de 32 bits diseñado originalmente por Digital Equipment Corporation (DEC) —ahora propiedad de HP— en 1978 para usarlo en minicomputadoras VAX. Posteriormente fue portado a sistemas DEC Alpha e Intel Itanium.

Figura 12. Instalación de VAX 11/780 en 1980 — Fuente: Science and Technology Facilities Council

Las siglas VMS vienen de *Virtual Memory System*, ya que una de sus principales características era explotar el concepto de **memoria virtual**. Este concepto también es muy utilizado en los sistemas operativos modernos. Permite que los procesos se ejecuten aislados, unos de otros, en la memoria principal y sin tener que ser cargados completamente, lo que permite que cada uno consuma menos memoria.

VMS era un sistema multiusuario y multiprocesador que podía distribuir el trabajo entre varias máquinas, lo que le permitía ser tolerante a fallos.

VMS es en cierta medida un ancestro de Microsoft Windows NT (véase el Apartado 3.5.6). Para desarrollar Windows NT, Microsoft contrató a un grupo de desarrolladores de Digital Equipment Corporation. Muchos aspectos del diseño de Windows NT reflejan la experiencia de DEC en VMS.

3.4.5. IBM OS/400

El IBM OS/400 es un sistema utilizado en la familia de minicomputadoras IBM AS/400 —llamada iSeries desde 2006—. Fueron introducidos en el mercado en 1988, pero aún es posible verlos en algunas organizaciones. En 2008 el sistema operativo IBM OS/400 pasó a llamarse IBM i y siguen publicándose nuevas versiones en la actualidad.

3.5. 5ª Generación (desde 1980):

Esta última generación abarca desde la década de 1980 hasta la actualidad.

Respecto a los sistemas operativos:

- Incluye a los sistemas operativos de escritorio y ordenadores personales (PC).
- Aparecen múltiples conceptos nuevos: monousuario, multitarea, distribuidos, paralelos, tiempo real, etc.

Se puede observar una muestra de la interfaz gráfica de usuario de algunos de estos sistemas en el artículo «Operating System Interface Design Between 1981-2009».

3.5.1. CP/M

CP/M (1974) fue el sistema operativo estándar en la primera generación de microcomputadoras. Fue creado por Digital Research, Inc. —fundada por Gary Kildall— para ser el sistema operativo de los microordenadores basados en Intel 8080/85 y Zilog Z80.

Con la elección de MS-DOS por parte de IBM para su IBM PC, CP/M fue perdiendo mercado paulatinamente hasta desaparecer. Sin embargo, la influencia de CP/M en MS-DOS es indudable, en tanto en cuanto 86-DOS, el predecesor de MS-DOS, estaba basado en las ideas de CP/M.

3.5.2. MS-DOS

MS-DOS fue el sistema operativo estándar en la segunda generación de microcomputadoras. No era ni multitarea ni multiusuario. Fue el primer sistema operativo del IBM PC —lanzado en 1981— y durante mucho tiempo fue ampliamente utilizado en toda la plataforma «PC compatible».

MS-DOS fue creado por Seattle Computer Products (SCP) con el nombre de 86-DOS en 1979. Se basaron en ideas de CP/M, pues pretendían ofrecer una versión de CP/M para procesadores Intel 8086. Inicialmente era conocido como QDOS (*Quick and Dirty Operating System*) pero SCP le cambió el nombre en 1980, cuando comenzaron a licenciarlo. Posteriormente Microsoft adquirió el sistema y lo vendió a IBM en 1981 con el nombre de MS-DOS.

Tanto IBM como Microsoft lanzaron versiones de DOS, aunque originalmente IBM solamente validaba y empaquetaba el software de Microsoft. Microsoft lanzaba sus versiones bajo el nombre de MS-DOS, mientras IBM las lanzaba bajo el nombre de IBM PC-DOS.

En PCjs se pueden probar de forma sencilla sistemas operativos y aplicaciones antiguas del IBM PC. Solo hace falta acceder con el navegador y elegir la experiencia que más nos llame la atención.

3.5.3. OS/2

OS/2 fue un sistema operativo creado por Microsoft e IBM para aprovechar las nuevas características de la segunda generación de ordenadores personales de IBM, equipados con procesador Intel 80286. Pero al final terminó siendo desarrollado en exclusiva por IBM.

OS/2 fue pensado como un sucesor con **operación en modo dual** de MS-DOS y de Microsoft Windows 2.0. Fue anunciado en abril y lanzado en diciembre de 1987 como un sistema operativo en modo texto. En la versión 1.1, lanzada en noviembre de 1988, se le añadió interfaz gráfica.

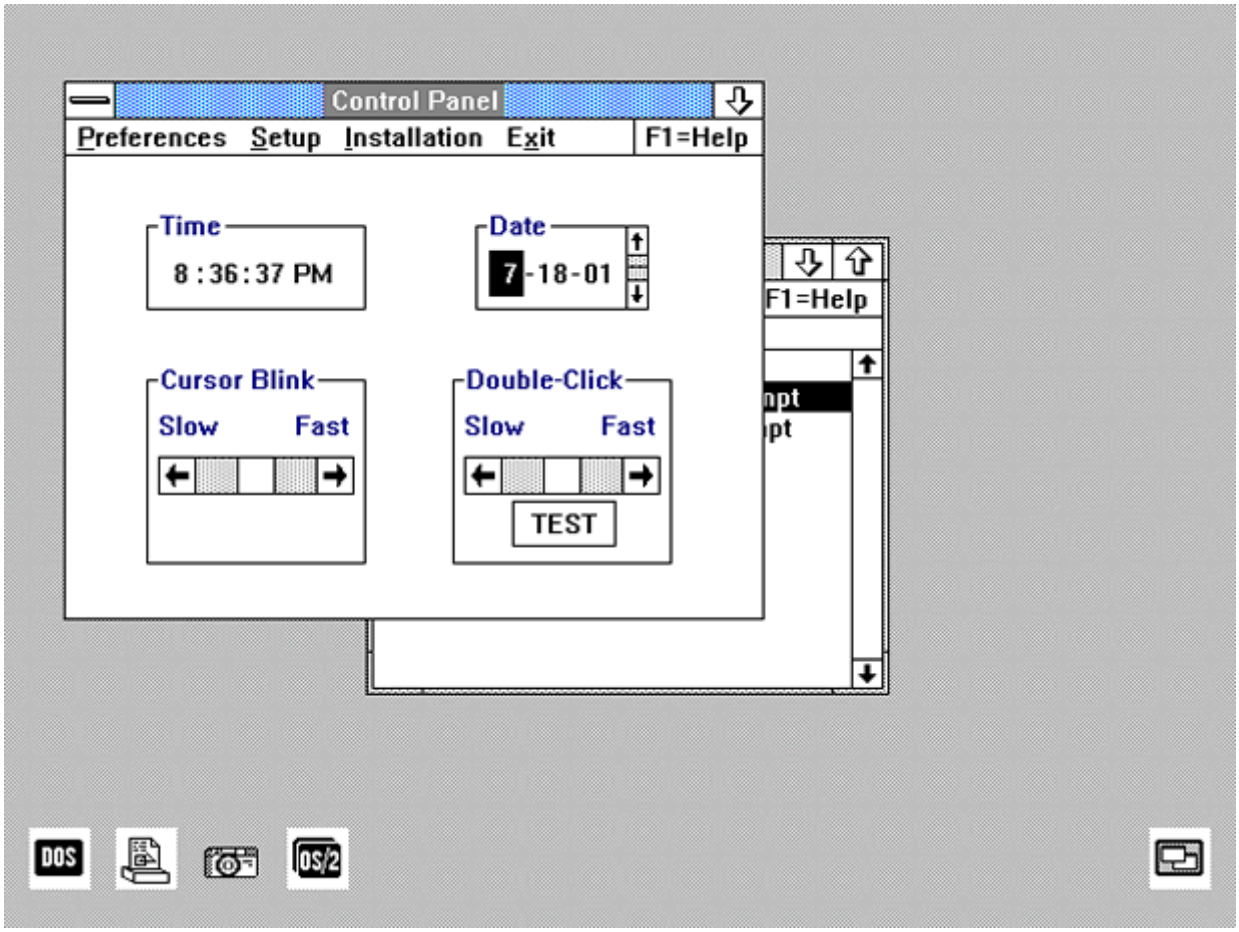


Figura 13. Panel de control de Microsoft-IBM OS/2 1.1 — Fuente: Michal Necasek

En los sistemas con **operación en modo dual** se distingue entre dos modos de ejecución, de tal forma que solo en el modo en el que se ejecuta el código del sistema operativo se pueden realizar operaciones peligrosas. En el otro modo y con menos privilegios, se ejecutan las aplicaciones de usuario. Para más información, véase el Apartado 7.2

La colaboración entre IBM y Microsoft terminó en 1990, entre el lanzamiento de Windows 3.0 y la de OS/2 1.3. El aumento de popularidad de Windows llevó a Microsoft a dejar de centrarse en el desarrollo de OS/2, lo que hizo que IBM se preocupara por los continuos retrasos en el desarrollo de OS/2 2.0. Inicialmente ambas compañías acordaron que IBM tomaría el mantenimiento de OS/2 1.0 y el desarrollo de OS/2 2.0, mientras Microsoft continuaría desarrollando OS/2 3.0, que entonces era conocido como «NT OS/2». Sin embargo, Microsoft finalmente decidió renombrar NT OS/2 como Windows NT, dejando el futuro desarrollo de OS/2 en manos de IBM.

OS/2 Warp 3 fue un sistema completo de 32 bits lanzado en 1994. Le seguiría OS/2 Warp 4, en 1996. Poco después, IBM anunció que OS/2 desaparecería.

3.5.4. Windows 3.x

La familia Windows 3.x de Microsoft Windows fue desarrollada desde 1990 hasta 1994. Windows 3.0 fue la primera versión de éxito de Windows, permitiendo a Microsoft competir con el Macintosh de Apple Computer y el Commodore Amiga.

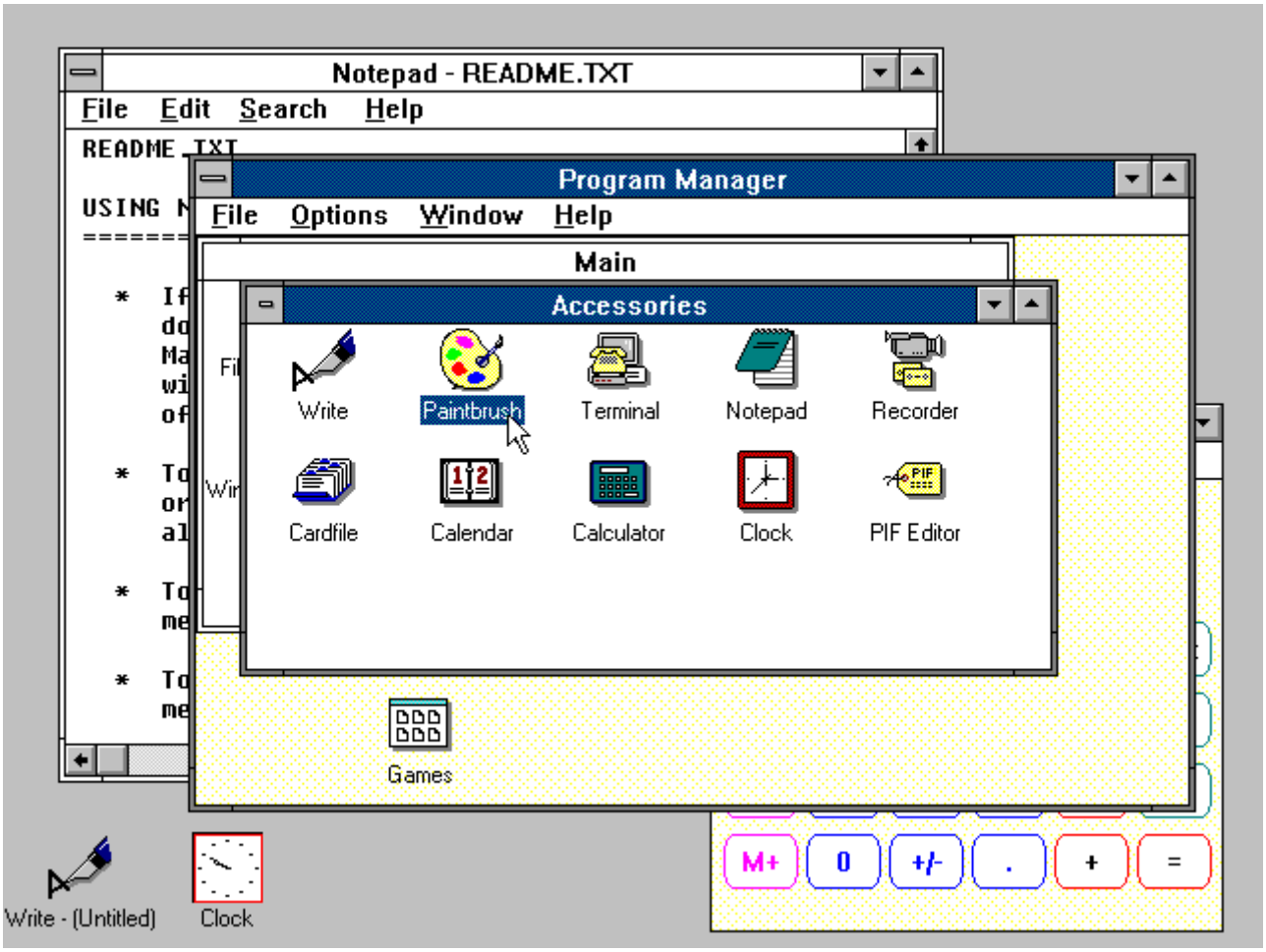


Figura 14. Administrador de programas de Microsoft Windows 3.0 — Fuente: Guidebook

En 1983, Microsoft anunció el desarrollo de Windows, una interfaz gráfica de usuario para su sistema MS-DOS, que se usaba en los IBM PC y compatibles desde 1981. Windows requería una instalación previa de MS-DOS y era iniciado como un programa más, que podía ser terminado en cualquier momento, devolviendo al usuario a la línea de comandos de MS-DOS.

MS-DOS le proporcionaba a Windows controladores de dispositivo para ciertas tareas, como el acceso al CD-ROM o a la interfaz de red. Sin embargo Windows ejecutaba aplicaciones específicas de Windows, almacenadas en un formato ejecutable mucho más complejo que el de los programas de MS-DOS. Además, debido a que MS-DOS no aislaba a las aplicaciones del hardware y no se protegía así mismo de los errores en dichas aplicaciones, Windows disponía de controladores de dispositivo propios, así como sus propios sistemas de gestión de procesos y de memoria. En realidad Windows no se ejecutaba sobre MS-DOS, si no que hacía uso de él. Por ello puede ser considerado como un sistema operativo.

3.5.5. Windows 95, 98, Me

La familia Windows 3.x fue sustituida por una serie de sistemas operativos gráficos híbridos de 16/32 bits.

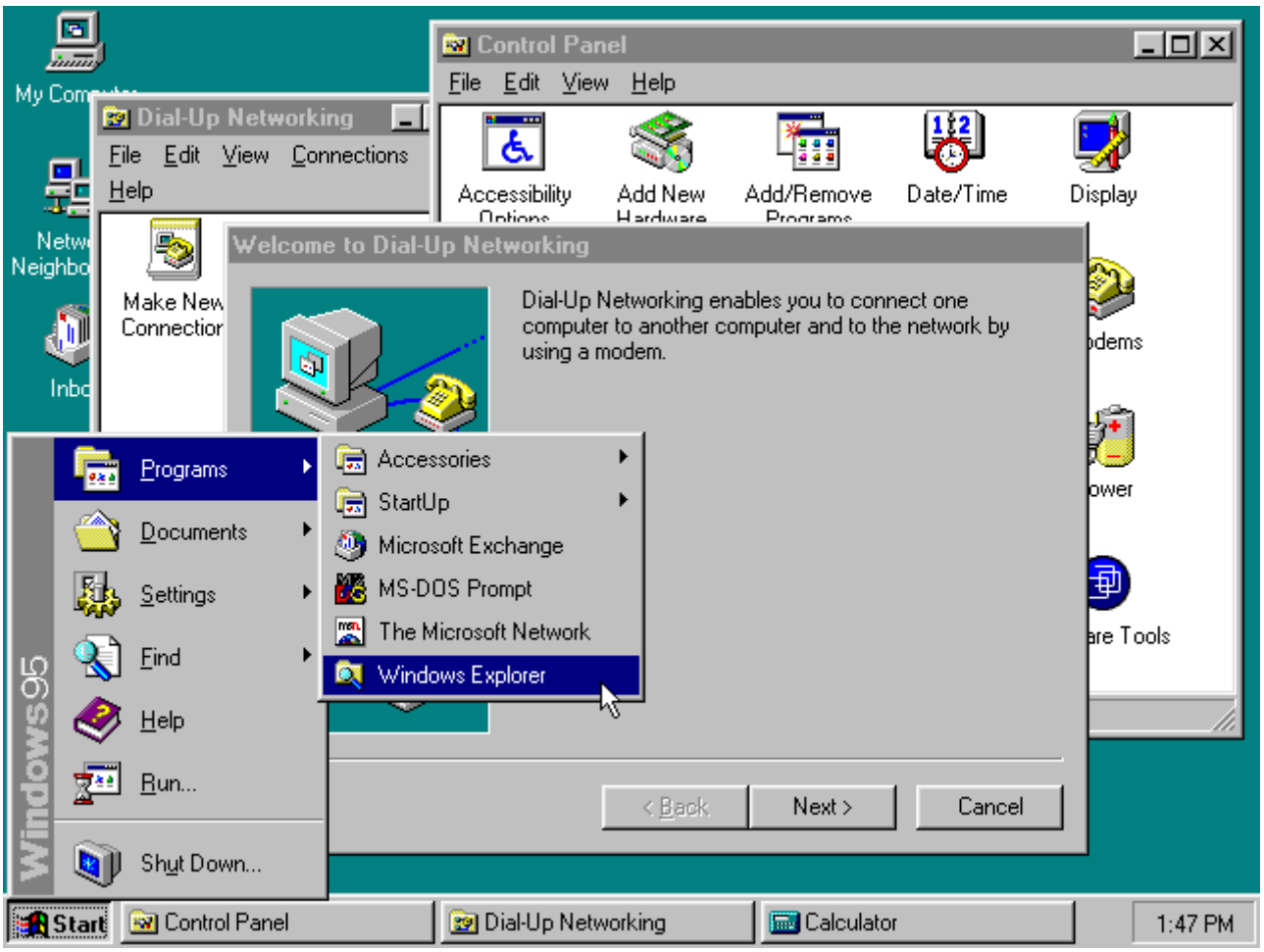


Figura 15. Escritorio de Microsoft Windows 95 — Fuente: Guidebook

Windows 95 fue lanzado en 1995. Fue el primer Windows unido a una versión de MS-DOS específica, aunque este hecho se intentaba mantener oculto. Entre las características de Windows 95 destacan: mejoras significativas en la interfaz de usuario (véase la Figura 15), nombres de archivo de hasta 256 caracteres con conservación de mayúsculas y minúsculas —en MS-DOS el límite era de 8 caracteres para el nombre más 3 de extensión— y multitarea expropiativa para las aplicaciones de 32 bits.

Como veremos en el Apartado 14.1, la planificación expropiativa es una técnica que permite al sistema operativo expulsar de la CPU a los procesos en ciertas circunstancias; como, por ejemplo, que llevan demasiado tiempo utilizando la CPU de forma ininterrumpida.

En la familia Windows 3.x la planificación era cooperativa, es decir, los procesos abandonaban la CPU voluntariamente. Esto ocasionaba problemas con programas que no devolvía la CPU al sistema con la suficiente frecuencia, ya que así el resto de procesos no tenía ocasión de ejecutarse para hacer su trabajo o responder al usuario.

Windows 98 fue lanzado el 25 de junio de 1998. Le siguió Windows Me, el 14 de septiembre de 2000. Windows Me fue la última versión de la familia de sistemas operativos híbridos de 16/32 bits que sucedió a la familia Windows 3.x.

3.5.6. Windows NT, 2000, XP, Vista, 7, 8 y 10

Windows NT fue un sistema operativo de 32 bits. El primero de la familia de sistemas operativos Microsoft Windows actuales.

Su desarrollo empezó en 1988 con el nombre de OS/2 3.0. Cuando Windows 3.0 fue lanzado en mayo de 1990, tuvo tanto éxito que Microsoft decidió cambiar la API del aún en desarrollo NT OS/2 —que era como Microsoft lo llamaba entonces— pasando de ser una versión extendida de la API de OS/2 a una versión extendida de la API de Windows 3.0. Esta decisión causó tensión entre Microsoft e IBM y provocó que finalmente terminara la colaboración.

Una interfaz de programación de aplicaciones o API (del inglés *Application Programming Interface*) es el conjunto de funciones, procedimientos o métodos que ofrece el sistema operativo para ser utilizado por las aplicaciones.

Como hemos comentado anteriormente, Microsoft contrató a un grupo de desarrolladores de Digital Equipment Corporation para crear Windows NT. Por lo que muchos de sus elementos reflejan la experiencia anterior de DEC en VMS.

Windows NT soportaba varias API de distintos sistemas operativos —por ejemplo Win32, POSIX y OS/2 2.1— que eran implementadas como subsistemas encima de una API nativo no documentado públicamente. Esta estructura en subsistemas, fue lo que permitió la adopción tardía de la API de Windows 3.0 como API principal, tal y como hemos comentado.

La primera versión —Windows NT 3.1— lanzada el 13 de julio de 1993, era un sistema operativo *microkernel* (véase el Apartado 3.5.8 un poco más adelante) multiplataforma que corría sobre procesadores x86, DEC Alpha, MIPS 4000 y PowerPC.

Windows NT 4.0 —lanzado en 1996— fue la última versión en soportar plataformas distintas a Intel IA-32. Aunque el desarrollo de Windows 2000 para procesador Alpha continuó un poco más, hasta 1999, cuando Compaq dejó de soportar Windows NT en esa arquitectura. Además Windows NT 4.0 integró en el núcleo más funciones —por ejemplo, parte del subsistema gráfico— para obtener un rendimiento más próximo al de Windows 95 en ese apartado.

Windows 2000 —o Windows NT 5.0— fue lanzado el 17 de febrero de 2000 y fue el primer sistema operativo de la familia NT al que se le eliminaron las siglas del nombre. Fue por motivos de marketing, para favorecer la unificación de las dos familias de sistemas operativos Microsoft Windows de entonces —Windows 9x y Windows NT— alrededor de la tecnología NT.

Windows XP —o Windows NT 5.1— completó en 2001 el proceso de unificación de las dos familias de sistemas operativos Windows. Con su aparición forzó la extinción de la familia Windows 9x, al sustituirla con una versión de Windows XP denominada Windows XP Home Edition, específica para la informática doméstica.

3.5.7. GNU/Linux

GNU/Linux es un sistema operativo libre y, tal vez, el más famoso proyecto de software libre.

El proyecto GNU se inició en 1983, con el fin de desarrollar un sistema operativo estilo UNIX enteramente libre. El proyecto incluía la creación de herramientas de desarrollo de software y aplicaciones de usuario.

Mucho tiempo después, el estudiante universitario finés Linus Torvalds comenzó a desarrollar el núcleo Linux como hobby, mientras estudiaba en la Universidad de Helsinki. Torvalds originalmente usaba Minix, un sistema operativo simplificado escrito por Andrew Tanenbaum para enseñar diseño de sistemas operativos. Sin embargo, el hecho de que Tanenbaum no diera soporte a las mejoras del sistema operativo que eran propuestas por otros desarrolladores, llevó a Torvalds a escribir un sustituto de MINIX.

En 1991, cuando se liberó la primera versión del núcleo Linux, el proyecto GNU había desarrollado todos los componentes necesarios del sistema operativo excepto el núcleo. Torvalds y otros desarrolladores rápidamente adaptaron Linux para que funcionara con los componentes de GNU, creando un sistema operativo completamente funcional que se denomina GNU/Linux.

El núcleo Linux fue licenciado bajo la GNU General Public License (GPL), como el resto del proyecto GNU. Pero Linux no es parte de dicho proyecto. El proyecto GNU tiene su propio núcleo, denominado GNU/Hurd, que lleva 30 años en desarrollo y parece que aún está muy lejos de estar listo.

GNU no es el único sistema operativo que utiliza el núcleo Linux. Android, por ejemplo, es un sistema operativo que usa el núcleo Linux, pero no es GNU.

3.5.8. Mach

Mach es un núcleo de sistema operativo desarrollado en la Universidad Carnegie-Mellon (CMU). El proyecto en la CMU se desarrolló desde 1985 hasta 1994.

Mach explora el concepto que denominamos ***microkernel***. En los sistemas operativos ***microkernel*** solo se implementa en el núcleo del sistema un conjunto mínimo de servicios básicos. El resto de los servicios proporcionados por el sistema operativo se implementan como procesos con menos privilegios.

Por sus ventajas en cuanto a seguridad y fiabilidad, en algún momento se pensó que los *microkernel* dominarían el universo de los sistemas operativos. Sin embargo, el mayor esfuerzo hasta la fecha para conseguirlo es GNU/Hurd, que lleva varias décadas de retraso. Por fortuna, otros sistemas operativos *microkernel* han tenido algo más éxito, como es el caso de QNX o MINIX 3. Mientras que Google parece que lo va a intentar con Google Fuchsia, el posible sustituto de Android.

A mediados de los 90, Apple Computers seleccionó OpenStep como base para el sucesor de su clásico Mac OS. OpenStep es realmente una versión actualizada de NeXTSTEP que era un sistema basado en un núcleo Mach 2.5 con porciones del sistema UNIX BSD de la Universidad de Berkeley. Por tanto, la mezcla de Mach con UNIX BSD de OpenStep es la base del sistema operativo macOS actual de Apple.

Figura 16. Entorno gráfico de OpenStep 4.2 — Fuente: Guidebook

Para ser exactos, la base del sistema operativo macOS es un sistema operativo libre denominado Darwin y desarrollado por Apple. Se trata de un sistema FreeBSD adaptado para correr sobre el núcleo Mach.

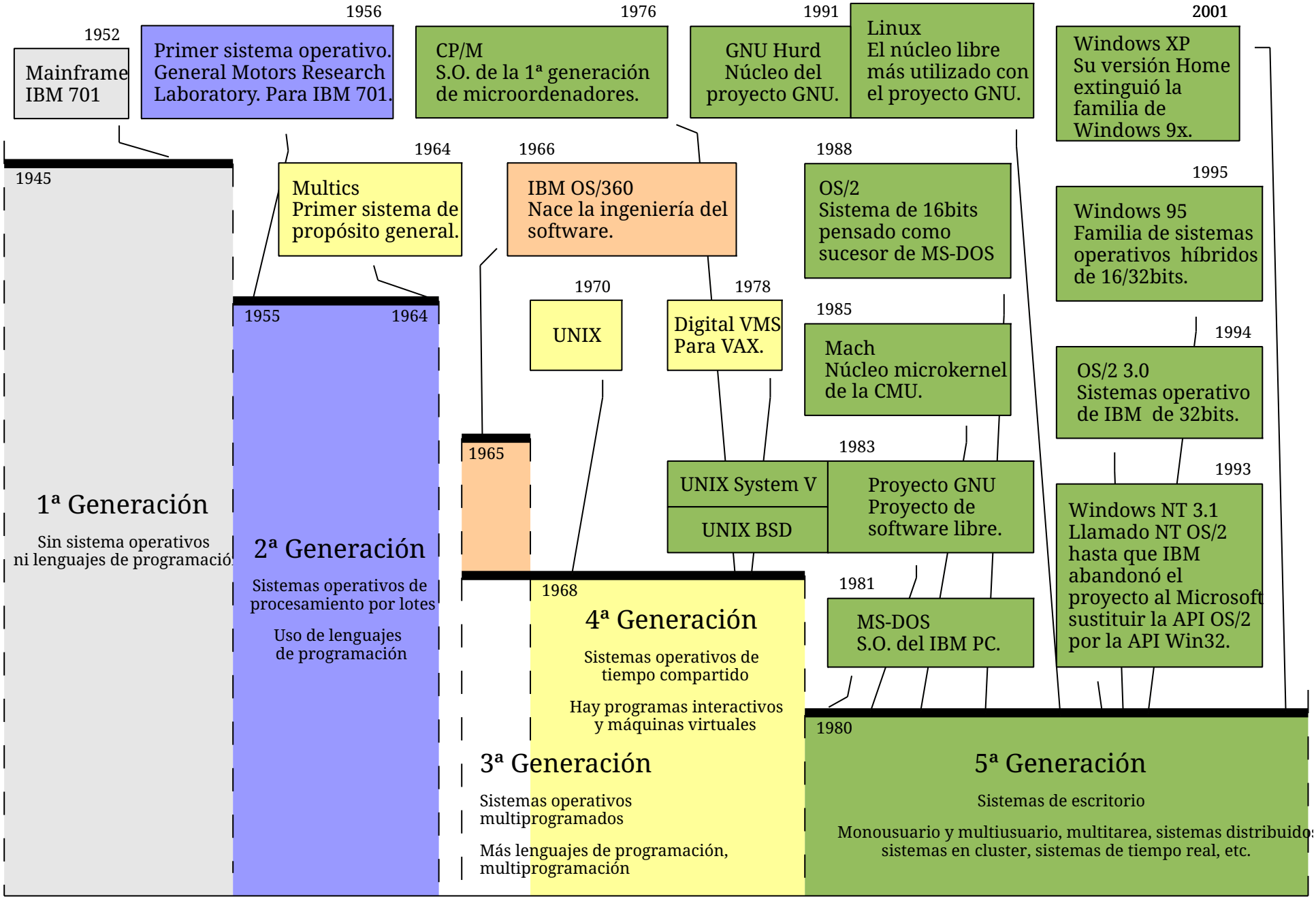


Figura 17. Línea de tiempo de la historia de los sistemas operativos.

Parte II: Organización de los sistemas operativos

El estudio de la organización interna de los sistemas operativos requiere del análisis de tres aspectos diferentes:

- Los componentes del sistema operativo y sus interrelaciones.
- Los servicios que el sistema operativo proporciona a través del funcionamiento coordinado de dichos componentes.
- La interfaz de programación que el sistema operativo ofrece a usuarios y programadores como forma de acceso a dichos servicios.

Finalmente, veremos cómo se categorizan los sistemas operativos según la forma en la que se interconectan sus componentes.

Chapter 4. Componentes del sistema

Tiempo de lectura: 14 minutos

Crear un software tan complejo como un sistema operativo no es sencillo, por ello resulta más práctico dividirlo en piezas más pequeñas especializadas en aspectos concretos de la gestión del sistema.

Cada sistema operativo tiene diferentes componentes con distinto nombre. Lo que veremos en este capítulo es un esquema de los más comunes a la mayoría de sistemas operativos actuales.

4.1. Gestión de procesos

La gestión de los procesos es un elemento central de todo sistema operativo:

- El **proceso** es la unidad de trabajo en cualquier sistema operativo moderno. Es quién realiza las tareas que interesan a los usuarios. Por eso, es a cada proceso al que se le asigna el tiempo de CPU y el resto de recursos del sistema, como por ejemplo: memoria, archivos o dispositivos de E/S abiertos.
- Un **proceso** es un programa en ejecución. Un programa se convierte en proceso cuando las instrucciones del programa son cargadas en la memoria desde el archivo del ejecutable y se le asignan recursos para su ejecución.

Los procesos son entidades activas que necesitan recursos —CPU, memoria, archivos, dispositivos E/S—. Algunos de esos recursos se asignan durante su creación, mientras que otros son solicitados por el proceso durante su ejecución —por ejemplo la memoria, de la que todo proceso necesita cierta cantidad para comenzar, pero que luego puede pedir más dinámicamente durante su ejecución—. Cuando el proceso termina el sistema operativo reclama de estos recursos aquellos que sean reutilizables para otros procesos.

Un **programa** no es un proceso, es una entidad pasiva. Es el contenido de un archivo en disco con las instrucciones que algún día una CPU ejecutará. Un programa no puede hacer ningún tipo de trabajo a menos que sus instrucciones sean ejecutadas por una CPU, pero si eso ocurre, ya no sería un programa sino un proceso.

Aunque varios procesos estén asociados al mismo programa no pueden ser considerados el mismo proceso. La CPU ejecuta las instrucciones de cada proceso una detrás de otra, de manera que para conocer la siguiente instrucción a ejecutar cada proceso tiene un contador de programa que se lo indica a la CPU, así como valores en los registros de la CPU que dependen de la historia pasada del proceso. Aunque varios procesos ejecuten el mismo programa, la secuencia de instrucciones ejecutadas y el estado del proceso en cada momento seguramente sean diferentes. Por lo tanto, no son el mismo proceso.

Por el momento estamos considerando que **proceso** y **trabajo** hacen referencia al

> mismo concepto porque en los sistemas más antiguos (véase el Apartado 2.1) la unidad de trabajo se llamaba **trabajo** mientras que en los sistemas modernos se llama **proceso**, de tal forma que podemos considerar al segundo una evolución del primero.
>
> Sin embargo, mirándolo exclusivamente desde la perspectiva de los sistemas operativos modernos, son dos conceptos diferentes aunque relacionados. En un sistema moderno un trabajo puede ser realizado por un solo proceso o mediante la colaboración de varios.

4.1.1. Responsabilidades de la gestión de procesos

El componente de gestión de procesos es el responsable de las siguientes actividades:

- Crear y terminar procesos.
- Suspender y reanudar los procesos.
- Proporcionar mecanismos para la sincronización de procesos.
- Proporcionar mecanismos para la comunicación entre procesos.
- Proporcionar mecanismos para el tratamiento de interbloqueos.

4.2. Gestión de la memoria principal

La memoria principal es un recurso fundamental para las operaciones de cualquier sistema operativo moderno. Esto es así porque generalmente es el único almacenamiento al que la CPU tiene acceso directo. Para que un programa pueda ser ejecutado debe ser copiado a la memoria principal. Y para que un proceso tenga acceso a datos almacenados en cualquier otro dispositivo de almacenamiento, primero deben ser copiados a la memoria principal.

Para mejorar el aprovechamiento de la CPU y la respuesta al usuario es necesario tener en la memoria varios programas al mismo tiempo. Puesto que dichos programas deben compartir la memoria durante su ejecución, automáticamente existe la necesidad de que el sistema operativo disponga de un componente de gestión de la memoria principal.

4.2.1. Responsabilidad de la gestión de la memoria

El componente de gestión de la memoria debe asumir las siguientes responsabilidades:

- Controlar qué partes de la memoria están actualmente en uso y cuáles no.
- Decidir que procesos —o partes de procesos— añadir o extraer de la memoria cuando hay o falta espacio en la misma.
- Asignar y liberar espacio de la memoria principal según sea necesario.

4.3. Gestión del sistema de E/S

El **sistema de E/S** hace de interfaz con el hardware, oculta las peculiaridades del hardware al resto del sistema.

El sistema de E/S consta de:

- **Un componente de gestión de memoria especializado en E/S**, con soporte para servicios de *buffering*, *caching* y *spooling*. Estos servicios son utilizados por el resto del sistema de E/S.
- **Una interfaz genérica de acceso a los controladores de dispositivo**. Cada dispositivo es diferente, pero los procesos y el resto de componentes del sistema no deben tener necesidad de conocer sus particularidades a la hora de acceder a ellos. Es decir, para acceder a cualquier disco duro el sistema ofrece una misma interfaz, independiente de su marca y modelo. Y lo mismo ocurre con las tarjetas de sonido o con los dispositivos de entrada, como teclados y ratones. Así los programadores pueden acceder a cualquier dispositivo abstrayendo de las particularidades concretas del hardware instalado en cada ordenador.
- **Controladores de dispositivo**, que generalmente son desarrollados por los fabricantes de los dispositivos y son el componente que realmente conoce las peculiaridades específicas del dispositivo. Por tanto, las peticiones que hacen los procesos a la interfaz de E/S genérica la traslada el sistema a los controladores de dispositivo para que estos las conviertan en acciones concretas sobre el hardware del dispositivo.

Interfaz de acceso a dispositivos en sistemas UNIX

Una característica de los sistemas UNIX es que todos los dispositivos de E/S se representan como un archivo en el sistema de archivos. Esto se puede comprobar rápidamente visitando el directorio `/dev` en cualquier sistema GNU/Linux o UNIX BSD, ya que es allí donde suelen estar.

Así no hace falta diseñar y aprender una interfaz diferente para cada tipo de dispositivo. Por ejemplo, no hace falta que el sistema ofrezca funciones para leer y escribir bloques en dispositivos de almacenamiento. Ni para leer la geometría del dispositivo, para configurar modos de trasferencia o de ahorro de energía o para eyectarlo, en el caso de los dispositivos removibles. Tampoco son necesarias funciones específicas para grabar y reproducir sonidos, para configurar la tarjeta de sonido o para leer y establecer los valores del mezclador. Por el contrario, todas las operaciones sobre los dispositivos de E/S se ofrecen a través de una interfaz bien conocida por los desarrolladores, la misma que se utiliza para gestionar archivos convencionales: open(), read(), write(), close() o mmap().

Por ejemplo, `/dev/urandom` es un dispositivo que se utiliza para obtener una secuencia de bytes aleatorios. Para usarlo, solo tenemos que abrirlo y leer de él la cantidad de bytes que necesitemos:

```
int fd = open( "/dev/urandom", O_RDONLY );
if (fd >= 0) {
    char random_data[64];
    ssize_t bytes_read = read( fd, random_data, sizeof(random_data) );   ①

    // ...
```

```
    close(fd);
}
```

① Si read() tiene éxito, `random_data` contendrá datos aleatorios al volver de la llamada al sistema.

Sin embargo, algunos dispositivos tienen funciones de control que no se trasladan bien a esta interfaz basada en archivos. Por ejemplo, si tenemos un lector de DVD o de Blu-ray, las operación read() nos permite leer el contenido del disco en crudo —es decir, bloque a bloque, ignorando el sistema de archivos— pero ¿cómo lo eyectamos o cómo sabemos si hay un disco en el lector?.

Para resolver este tipo de situaciones, los sistemas UNIX incorporan la llamada al sistema ioctl(), que permite enviar comandos u obtener parámetros específicos del dispositivo. La llamada al sistema ioctl() acepta los siguientes argumentos:

- El descriptor de archivo obtenido al abrir el dispositivo.
- El código de petición al dispositivo. Depende del dispositivo, por lo que para conocer los códigos que se soportan es necesario leer la documentación del controlador.
- Un puntero opcional a un búfer en la memoria, para recibir los parámetros solicitados o para enviar datos al dispositivo.

Por ejemplo, así podríamos eyectar el primer lector de CD/DVD-ROM o Blue-ray en Linux:

```
#include <linux/cdrom.h>                                ①

// ...

int fd = open( "/dev/sr0", O_RDONLY | O_NONBLOCK );    ②
if (fd >= 0) {
    ioctl ( fd, CDROMEJECT );                           ③
    close ( fd );
}
```

① Este archivo contiene los códigos IOCTL conocidos por los controladores de dispositivo del tipo lectores de CD-ROM.

② Como en el ejemplo anterior con `/dev/uranom`, es necesario abrir el dispositivo para obtener un descriptor de archivo.

③ En la llamada al sistema ioctl() se indica el código de la petición para eyectar la bandeja.

4.3.1. Buffering

El **buffering** o uso de memoria intermedia es una estrategia en la que se almacenan los datos de manera temporal en una zona de la memoria, llamada búfer.

Consiste en que el controlador indica a un dispositivo que escriba los bloques de datos solicitados

en un búfer. Cuando la escritura del búfer se ha completado, se transfiere su contenido al proceso que hizo la solicitud para que procese los datos. Mientras lo hace, el controlador indica al dispositivo que copie nuevos datos en el búfer.

Por ejemplo, al grabar sonido del dispositivo de sonido del sistema no se entregan las muestras una a una al proceso. En su lugar se graban varios miles de muestras que se escriben en un búfer. Cuando el búfer está lleno, se transfieren todas las muestras al proceso de una sola vez y se siguen grabando muestras en el búfer.

Lo mismo ocurre al reproducir sonido. El proceso no entrega las muestras de sonido de una en una al dispositivo, sino que empaqueta varios miles que se copian al búfer de una sola vez. Entonces el controlador indica al dispositivo que lea las muestras desde ese búfer según lo vaya necesitando.

4.3.2. Caching

En el **caching** el sistema mantiene en la memoria principal una copia de datos leídos o escritos recientemente en los dispositivos de E/S del sistema —por ejemplo, en los discos duros o en las memorias USB—. Esto mejora la eficiencia del sistema si accede con frecuencia a los mismos datos, puesto que el acceso a la memoria principal es más rápido que el acceso a los dispositivos de E/S. La memoria principal es de tamaño limitado, por lo que solo se mantiene copia de los datos utilizados con mayor frecuencia.

4.3.3. Spooling

El **spooling** se utiliza en dispositivos que no admiten el acceso simultáneo de varias aplicaciones a la vez, como es el caso de impresoras y unidades de cinta.

Cuando varias aplicaciones intentan enviar un trabajo a una impresora, el sistema operativo lo intercepta para copiar los datos enviados a un archivo independiente. Cuando una aplicación termina de enviar el trabajo, el archivo correspondiente se mete en una cola de donde son extraídos los trabajos para su impresión de uno en uno. Así no hay acceso simultáneo al dispositivo por parte de varios procesos, mientras que estos pueden entregar el trabajo y continuar con su trabajo sin esperar a que la impresora esté disponible.

4.4. Gestión del almacenamiento secundario

Dentro de los dispositivos de E/S, los dedicados al almacenamiento secundario —como discos duros, memorias USB o lectores de DVD-ROM— merecen un tratamiento especial.

Los programas que se desean ejecutar deben estar en la memoria principal —o almacenamiento primario— pero esta es demasiado pequeña para alojar todos los datos y todos los programas del sistema. Además, incluso aunque pudiera ser así, los datos almacenados en la memoria principal se perderían en caso de que ocurriera un fallo de alimentación. Por eso los ordenadores disponen de un almacenamiento secundario para guardar datos de forma masiva y permanente.

El gestor del almacenamiento secundario utiliza el sistema de E/S para acceder a los dispositivos y ofrecer al sistema servicios más complejos.

4.4.1. Responsabilidades de la gestión del almacenamiento secundario

El gestor del almacenamiento secundario es el responsable de:

- Gestionar el espacio libre en discos duros y resto de dispositivos de almacenamiento secundario.
- Asignar el espacio de almacenamiento.
- Planificar el acceso a los dispositivos, de tal forma que se ordenen las operaciones de forma eficiente.

4.5. Gestión del sistema de archivos

Los ordenadores pueden almacenar información en diferentes tipos de medios físicos —por ejemplo en discos duros magnéticos, CD/DVD-ROM, memorias USB o SSD— cada uno de los cuales tiene características propias. El acceso a cada tipo de medio es controlado por un dispositivo —por ejemplo el controlador de disco o la unidad de DVD-ROM— que también tiene características propias. El sistema de E/S y la gestión del almacenamiento secundario simplifican el acceso a estos dispositivos, pero no lo suficiente como para que sea cómodo usarlos constantemente en cualquier programa.

Para simplificar aún más el acceso al almacenamiento, el sistema operativo proporciona una visión lógica uniforme de todos los sistemas de almacenamiento. Es decir, abstrae las propiedades físicas de los dispositivos de almacenamiento para definir el **archivo**, una unidad de almacenamiento lógico con la que trabajan los procesos para guardar y recuperar datos.

> Un **archivo** o **fichero** es una colección de datos relacionados, identificados por un nombre, que es tratada por el sistema operativo como la unidad de información en el almacenamiento secundario.

Esto quiere decir que, por lo general, para el sistema operativo un archivo no es más que una colección de bytes y lo que ofrece son servicios para leer, escribir, identificar y manipular dicha colección. La organización y formato utilizados para guardar la información —como, por ejemplo, el detalle de cómo se codifica en un archivo una imagen al guardarla en formato JPEG— y lo que se hace dicha información, es algo que generalmente solo incumbe a las aplicaciones, no al sistema operativo.

Los archivos normalmente se organizan en directorios para facilitar su uso y organización.

4.5.1. Responsabilidades de la gestión del sistema de archivos

El sistema de archivos utiliza al gestor del almacenamiento secundario y al sistema de E/S y es responsable de las siguientes actividades:

- Crear y borrar archivos.
- Crear y borrar directorios para organizar los archivos.
- Soportar operaciones básicas para la manipulación de archivos y directorios: lectura y escritura

de datos, cambio de nombre, cambio de permisos, etc.

- Mapear en memoria archivos del almacenamiento secundario.
- Hacer copias de seguridad de los archivos en sistemas de almacenamiento estables y seguros.

4.6. Gestión de red

El componente de red se responsabiliza de la comunicación con otros sistemas interconectados mediante una red de ordenadores —por ejemplo, en Internet o en la red de área local de una oficina—.

4.7. Protección y seguridad

Protección es cualquier mecanismo para controlar el acceso de los procesos y usuarios a los recursos definidos por el sistema.

- Son mecanismos necesarios cuando un sistema informático tiene múltiples usuarios y permite la ejecución concurrente de varios procesos, pues así solo pueden utilizar los recursos aquellos procesos que hayan obtenido la autorización del sistema operativo.
- Permite mejorar la fiabilidad, al permitir detectar los elementos del sistema que no operan correctamente. Un recurso desprotegido no puede defenderse contra el uso —o mal uso— de un usuario no autorizado o incompetente.

Ejemplos típicos de mecanismos de protección son el hardware de direccionamiento de memoria, que se utiliza para que los procesos se ejecuten en su propio espacio de direcciones, y el temporizador, que garantiza que ningún proceso toma el control de la CPU por tiempo indefinido. Además, los registros de los dispositivos de E/S suelen estar protegidos del acceso directo de los usuarios, lo que protege la integridad de los dispositivos. Mientras que en algunos sistemas se pueden establecer permisos sobre los archivos para garantizar que solo los procesos con la debida autorización tengan acceso.

Un sistema puede tener la protección adecuada pero estar expuesto a fallos y permitir accesos inapropiados. Por eso es necesario disponer de mecanismos de **seguridad** que se encarguen de defender el sistema frente a ataques internos y externos.

Eso incluye a virus y gusanos, ataques de denegación de servicio, robo de identidad y uso no autorizado del sistema, entre muchos otros tipos de ataque.

Chapter 5. Servicios del sistema

Tiempo de lectura: 5 minutos

Un sistema operativo proporciona un entorno para la ejecución de programas. Ese entorno debe proporcionar ciertos servicios a los programas y a los usuarios de esos programas. Estos servicios son proporcionados gracias al funcionamiento coordinado de los diferentes componentes del sistema.

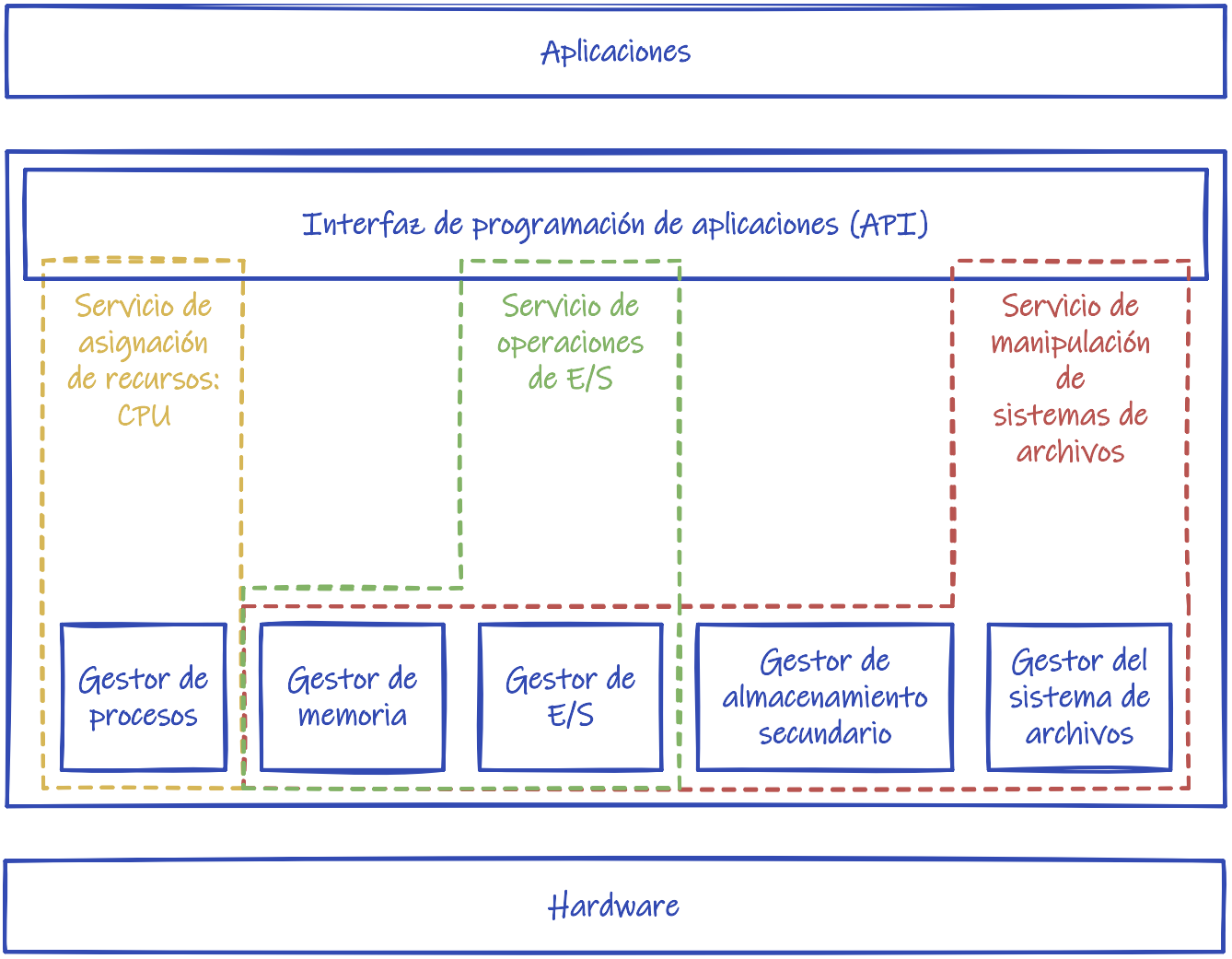


Figura 18. Esquema de la relación entre componentes y servicios.

Aunque cada sistema operativo proporciona servicios diferentes, es posible identificar unas pocas clases comunes.

5.1. Servicios que garantizan el funcionamiento eficiente del sistema

- **Asignación de recursos**. Cuando hay múltiples usuarios o múltiples trabajos ejecutándose los recursos deben ser asignados a cada uno de ellos.

  Ejemplos de estos recursos son la CPU —asignada por el planificador de la CPU del gestor de procesos— la memoria principal —asignada por el gestor de memoria— y el almacenamiento

de archivos —asignada por el sistema de archivos y el gestor del almacenamiento secundario—. Esta asignación debe hacerse con el fin de garantizar la máxima eficacia del sistema.

- **Monitorización**. Es normal querer hacer seguimiento de los recursos que los usuarios usan y en qué cantidad. Esto puede ser útil para facturar a los usuarios por el uso de los recursos —por ejemplo, facturar por el tiempo de CPU— para configurar el sistema mejorando el rendimiento o para limitar cuánto de cada recurso puede usar cada usuario como máximo.
- **Protección y seguridad**. Protección implica asegurar que el acceso a los recursos del sistema está controlado. Por ejemplo, que la información almacenada en un sistema multiusuario solo pueda ser accedida por su propietario o que un proceso no pueda interferir con otro o con el sistema operativo. La seguridad del sistema respecto a los agentes exteriores también es importante. Empieza obligando a los usuarios a autenticarse en él para obtener acceso a los recursos del mismo, pero incluye defender de intentos de acceso inválidos a través de la red.

5.2. Servicios útiles para el usuario

- **Interfaz de usuario**. Los sistemas operativos diseñados para que los usuarios interactúen con ellos deben proporcionar una interfaz de usuario adecuada, que puede tener diferentes formas según el propósito del sistema.
- **Operaciones de E/S**. Un programa puede necesitar realizar operaciones de E/S que pueden involucrar a archivos o a dispositivos de E/S. Por eficiencia y protección un usuario, normalmente los procesos no puede tener acceso directo a los dispositivos; por lo que el sistema operativo debe proporcionar medios para solicitar estas operaciones a los componentes correspondientes del sistema operativo.
- **Manipulación de sistemas de archivos**. Los programas necesitan leer y escribir archivos y directorios, crearlos y borrarlos por nombre, buscar un archivo dado y listar información acerca del mismo.
- **Comunicaciones**. Los procesos necesitan poder intercambiar información entre ellos, tanto si se ejecutan en el mismo ordenador, como en diferentes equipos unidos por una red.
- **Detección de errores**. El sistema operativo necesita tener conocimiento de los posibles errores y para cada tipo de error debe tomar la acción apropiada para asegurar una computación consistente y segura. Por ejemplo, puede haber errores del hardware —como fallos de energía o errores en la memoria— en la E/S —como errores de paridad o falta de papel en la impresora— y en los programas de usuario —como desbordamientos aritméticos o accesos ilegales a la memoria—.

5.3. Interfaz de usuario

La **interfaz de usuario** es un servicio fundamental para todos los sistemas diseñados para que los usuarios interactúen con ellos directamente, por lo que nos vamos a detener un poco más en él.

Las interfaces de usuario pueden ser de diferentes tipos:

- **Interfaz de línea de comandos** o **intérprete de comandos**, que permite que los usuarios introduzcan directamente los comandos que el sistema operativo debe ejecutar. En algunos sistemas este tipo de interfaz se incluye dentro del núcleo, pero en la mayor parte —como MS-

DOS y UNIX— se trata de un programa especial denominado *shell* que se ejecuta cuando un usuario inicia una sesión.

- **Interfaz de proceso por lotes**, en la que los comandos y directivas para controlar dichos comandos se listan en archivos que posteriormente pueden ser ejecutados. Este tipo de interfaz es la utilizada en sistemas no interactivos, como los antiguos sistemas de procesamiento por lotes y los sistemas multiprogramados.

  También suele estar disponible en los sistemas de tiempo compartido y en los sistemas de escritorio modernos, junto con algún otro tipo de interfaz de usuario. Por ejemplo, la *shell* de los sistemas UNIX permite indicar comandos uno a uno —de forma interactiva— pero también permite usar *scripts* —un archivo con una lista de órdenes para que se ejecuten automáticamente de principio a fin—.

- **Interfaz gráfica de usuario** o **GUI** (*Graphical User Interface*) que permite a los usuarios utilizar un sistema de ventanas y menús controlable mediante el ratón.

Puesto que la interfaz de usuario puede variar de un sistema a otro, y de un usuario a otro dentro del mismo sistema, no se suele etiquetar como un componente básico del sistema operativo, sino como un servicio ofrecido por el sistema operativo.

Aparte de la interfaz de usuario, cualquier sistema operativo moderno incluye una colección de **programas del sistema**. El papel de estos programas del sistema es proporcionar un entorno conveniente para la ejecución y desarrollo de programas. Entre los programas del sistema se suelen incluir aplicaciones para manipular archivos y directorios, programas para obtener información sobre el estado del sistema —como la fecha y hora o la memoria y el espacio en disco disponible— herramientas de desarrollo —como intérpretes, compiladores, enlazadores y depuradores— programas de comunicaciones —como clientes de correo electrónico y navegadores web— etc.

Además, muchos sistemas operativos disponen de programas que son útiles para resolver los problemas más comunes de los usuarios. Entre estos programas se suelen incluir: editores de archivos de texto y procesadores de texto, hojas de cálculo, sistemas de base de datos, juegos, etc. A esta colección de aplicaciones se la suele conocer con el término de **utilidades del sistema** o **programas de aplicación**.

Chapter 6. Interfaz de programación de aplicaciones

Tiempo de lectura: 18 minutos

Un sistema operativo proporciona un entorno controlado para la ejecución de programas. Dicho entorno debe proporcionar ciertos servicios que pueden ser accedidos por los programas a través de una **interfaz de programación de aplicaciones** o **API** (*Application Programming Interface*).

6.1. Interfaces de programación de aplicaciones

Algunas de las API disponibles para los desarrolladores de aplicaciones son Windows API y POSIX.

6.1.1. Windows API

Windows API es el nombre que recibe la **interfaz de programación de aplicaciones** de Microsoft Windows, con la que prácticamente tienen que interactuar todas las aplicaciones, de una forma u otra.

Antiguamente se denominaba Win32 API, pero Microsoft ha querido aglutinar bajo una misma denominación las distintas versiones de la API de Windows que han existido, como Win16 —usada en las versiones de 16 bits de Windows— o Win64 —que es la variante de Win32 adaptada a arquitecturas de 64 bits—.

Está compuesta por funciones en C almacenadas, principalmente, en las librerías de enlace dinámico (DLL): `kernel32.dll`, `user32.dll` y `gdi32.dll`. Aunque según se ha ido ampliando la API, se han incorporado otras librerías adicionales.

Provee un conjunto muy amplio de servicios: E/S a archivos y dispositivos, gestión de procesos, hilos y memoria, manejo de errores, registro de Windows, interfaz a dispositivos gráficos —pantallas e impresoras— gestión de ventanas, comunicaciones en red, etc.

6.1.2. POSIX

POSIX (*Portable Operating System Interface for Unix*) es el nombre de una familia de estándares que definen una **interfaz de programación de aplicaciones** para sistemas operativos. Esto permite que un mismo programa pueda ser ejecutado en distintos sistemas operativos, siempre que sean compatibles con POSIX.

El lenguaje C fue diseñado originalmente para implementar sistemas UNIX y por eso la librería estándar de C tenía mucho parecido con la librería del sistema de UNIX. Con el tiempo, al ir añadiendo más funcionalidades, la librería del sistema de los sistemas UNIX de los distintos fabricantes fue divergiendo, haciendo muy complicado desarrollar programas que usasen las características más avanzadas y que a la vez pudieran ejecutarse en varios de ellos. Por eso el IEEE desarrolló el estándar POSIX, que define una API común para todos los UNIX y sistemas estilo UNIX modernos —como es el caso de GNU/Linux—. Así que la práctica totalidad de estos sistemas son compatible POSIX.

Por su origen, la API POSIX es un superconjunto de la API de la librería estándar de C. Por eso en los sistemas POSIX, la librería estándar de C es parte de la librería del sistema, en lugar de dos librerías separadas.

Las funciones POSIX están almacenadas, principalmente, en la librería `libc`. Aunque algunas características pueden estar en otras librerías, como `libm` —la librería matemática— o `libpthread` —la librería de hilos—.

Los desarrolladores del sistema a veces añaden funciones no incluidas en el estándar POSIX, con el objeto de soportar algún tipo de funcionalidad avanzada del sistema. Este es el caso de las diferentes versiones de BSD y la librería del sistema del proyecto GNU —usada generalmente en los sistemas Linux— que incluye sus propias extensiones. Además, el estándar POSIX ha tenido varias revisiones desde la primera —publicada en 1988— cada una de las cuales añade características y funcionalidades adicionales.

Antes de usar extensiones y características avanzadas debemos tener presente que:

- Un programa que solo utilice funcionalidades hasta cierta versión de la API POSIX, podrá ejecutarse en cualquier sistema operativo compatible POSIX que implemente al menos hasta esa versión del estándar.
- Mientras que uno que utilice, por ejemplo, alguna funcionalidad adicional no POSIX de GNU/Linux o macOS, solo podrá compilarse y ejecutarse en GNU/Linux o en macOS, según el caso.

Como la compatibilidad con diferentes sistemas puede ser algo bastante complejo de gestionar para los desarrolladores, los sistemas POSIX ofrecen macros con las que controlar qué funcionalidades del sistema están disponibles para nuestro programa. A estas macros sé las denomina macros de test de características.

Por ejemplo, el siguiente programa en C —disponible en softstack.c— realiza una serie de tareas muy sencillas: crea un archivo, muestra una serie de mensajes por la salida estándar, cierra el archivo y termina. Sin embargo, si no se define la macro `_POSIX_C_SOURCE` puede que no compile, según la versión de la librería del sistema y las opciones del compilador. El motivo es que todas las funciones utilizadas en el programa forman parte del estándar POSIX desde hace tiempo, excepto mkstemp(), que es una función introducida en el estándar POSIX.1-2008. Por lo que si el compilador por defecto compila para una versión anterior del estándar, esta y otras funciones definidas en especificaciones posteriores no están.

```
#define _POSIX_C_SOURCE 200809L                    ①

#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <stdio.h>
#include <stdlib.h>

int main()
```

```
{
    char filename[] = "/tmp/softstack-fileXXXXXX";

    mkstemp(filename);                                  ②

    puts("Antes de abrir el archivo...");
    int fd = open(filename, O_RDWR | O_CREAT);
    puts("Después de abrir el archivo...");
    close(fd);

    return EXIT_SUCCESS;
}
```

① Definir la macro `_POSIX_C_SOURCE` con el valor `200809L` activa que las cabeceras expongan definiciones correspondientes a la especificación POSIX.1-2008.

② Genera un nombre de archivo que no exista en el sistema de archivos usando la plantilla indica por `filename`. Esta función pertenece al estándar POSIX.1-2008.

Este mecanismo tiene la doble ventaja de que:

1. Al tener que incluir las macros sabemos los requisitos mínimos del sistema donde podremos compilar y ejecutar nuestro programa, aunque no estemos preocupados por cumplir con un estándar concreto.
2. Si es un requisito del proyecto que se ejecute en sistemas con un estándar particular, solo tenemos que incluir la macro correspondiente y el compilador no nos dejará usar definiciones no incluidas en la especificación indicada.

Los sistemas POSIX pueden soportar muchas otras macros de test de características, según las especificaciones y extensiones soportadas por la librería del sistema. Las de uso más común en los sistemas GNU son:

- `_POSIX_C_SOURCE`. Según el valor asignado a esta macro se establece la especificación POSIX que debe activarse para el programa. Los valores válidos se indican en la documentación de las macros de test de características.
- `_XOPEN_SOURCE`. Definiendo esta macro se indica la especificación de la Guía de portabilidad X/Open que debe activarse para el programa. Según el valor se activarán ciertas especificaciones POSIX junto con algunas extensiones adicionales.
- `_BSD_SOURCE`. Activa funcionalidades específicas de los sistemas BSD.
- `_DEFAULT_SOURCE`. Activa las especificaciones y extensiones por defecto, por si se diera el caso de que han sido desactivas de alguna manera. Las definiciones por defecto incluyen: POSIX.1-2008, ISO C99 y algunas funcionalidades extra de sistemas UNIX BSD y System V.
- `_GNU_SOURCE`. Activa `_DEFAULT_SOURCE` y extensiones específicas de los sistemas GNU.

6.2. Llamadas al sistema

Para un programa, acceder a los servicios del sistema operativo no es tan sencillo como invocar una función. Para invocar una función, un programa necesita conocer la dirección en la memoria del

punto de entrada de dicha función —es decir, la ubicación de su primera instrucción—. Sin embargo, el código del núcleo del sistema puede estar en cualquier ubicación de la memoria principal. Así que las direcciones de los puntos de entrada a las funciones del núcleo son desconocidas. Además, generalmente, el código y los datos del núcleo están protegidos frente a accesos indebidos (véase el Apartado 7.3). Eso significa que para que un proceso pueda invocar los servicios que necesita hace falta un procedimiento diferente, denominado **llamada al sistema**.

6.2.1. Invocar llamadas al sistema

Generalmente una llamada al sistema se invoca mediante una instrucción específica en lenguaje ensamblador que genera una **excepción** —que no es más que una interrupción lanzada por la propia CPU al detectar instrucciones especiales o un error al ejecutar una instrucción, como una división por 0 o un acceso indebido a ciertas zonas de la memoria—. Por ejemplo, en MIPS e Intel x86 se usa la instrucción `syscall`, que lanza una excepción, haciendo que la CPU salte a una rutina en el código del núcleo del sistema, deteniendo así la ejecución del proceso que la invocó.

Al realizar una llamada, es necesario que el sistema sepa qué operación le está pidiendo el proceso. Esto se suele hacer poniendo un número identificativo de la llamada en un registro concreto de la CPU. Por ejemplo, en Linux para x86 la llamada al sistema open() —que se utiliza para abrir archivos— se identifica con el número 2 o con el 5, según si es en 64 o en 32 bits, respectivamente. Este número se debe guardar en el registro `v0` en MIPS o `eax` en x86, antes de la instrucción `syscall`.

Los números utilizados para identificar cada llamada al sistema dependen del sistema operativo. Mientras que el registro donde se guarda, la instrucción utilizada y el resto de detalles sobre cómo realizar la llamada, dependen también de la arquitectura de la CPU.

6.2.2. Paso de argumentos

Obviamente una llamada al sistema suele requerir más información que la identidad de la llamada. Si, por ejemplo, se quiere abrir un archivo, al menos es necesario indicar su nombre, así como si se abre para leer o para escribir.

En concreto hay tres métodos para pasar parámetros adicionales al identificador a una llamada al sistema:

- **Mediante registros de la CPU**. Consiste en cargar los parámetros de la llamada al sistema en los registros de la CPU antes de realizar la llamada al sistema. Este método es el más eficiente, pero limita el número de parámetros al número de registros disponibles.

  Es utilizado, por ejemplo, en Linux para MIPS (véase el Ejemplo 1) y en la mayoría de sistemas operativos para x86-64.

- **Mediante tabla en memoria** Consiste en copiar los parámetros de la llamada al sistema en una tabla en la memoria principal y luego guardar la dirección de dicha tabla en un registro específico de la CPU, antes de la llamada al sistema. Así no se limita el número de parámetros que pueden ser pasados en cada llamada al sistema.

  Era utilizado por Microsoft Windows 2000 y anteriores. También en Linux para x86 32 bits, cuando el número de parámetros es superior a 6.

- **Mediante la pila del proceso** se insertan los parámetros de la llamada al sistema en la pila del proceso —que también se suele usar para guardar variables locales y, en algunas arquitecturas, los argumentos pasados al llamar a funciones— y el sistema operativo los recupera de allí durante la llamada al sistema. Al igual que en el caso anterior, tampoco limita el número de parámetros que pueden ser pasados en cada llamada al sistema.

  Es utilizado, por ejemplo, en sistemas UNIX BSD y en Windows XP y posteriores para x86 de 32 bits.

Ejemplo 1. Llamada al sistema en Linux MIPS.

Veamos cómo invocar directamente la llamada al sistema write() en Linux para MIPS.

Esta llamada sirve para escribir datos en un archivo. Así que necesita tres argumentos:

- **SIZE**: El número de bytes a escribir.
- **BUFFER**: La dirección de la memoria de la que coger los bytes.
- **FILEDES**: El descriptor que identifica a un archivo abierto donde se van a escribir los datos.

Al terminar devuelve el número de bytes escritos en el archivo, que puede ser inferior a `SIZE`.

El identificador de la llamada al sistema es 4004, según el listado de llamadas al sistema para Linux en MIPS.

```
lw      $a0, FILEDES   ①
la      $a1, BUFFER    ①
lw      $a2, SIZE      ①
li      $v0, 40004     ②
syscall              ③ ④
```

① Cargar cada uno de los 3 argumentos de la llamada al sistema en los registros `a0`, `a1` y `a2`.

② Cargar el identificador de la llamada `write()` en el registro `v0`.

③ Invocar la llamada al sistema. Aunque vemos que es una única instrucción, lo que realmente va a ocurrir es que el sistema operativo va a tomar el control de la CPU para realizar la tarea solicitada. La siguiente instrucción no comenzará a ejecutarse hasta que el sistema operativo no lo decida, por lo que, desde el punto de vista del programa, va a ser como si `syscall` fuera una instrucción más lenta de lo normal.

④ Al ejecutar la siguiente instrucción del código del programa, el registro `v0` contendrá el número de bytes escritos.

En syscalls.s se puede ver un ejemplo completo similar, pero para Linux x86 de 64 bits.

En cualquier caso, sea cual sea el método utilizado, el sistema operativo es responsable de comprobar de manera estricta la validez de los parámetros enviados en la llamada al sistema antes de realizar cualquier operación, puesto que nunca debe confiar en que los procesos hagan su trabajo correctamente. A fin de cuentas, una de las funciones del sistema operativo es el control de

dichos procesos.

6.3. Librería del sistema

Las **llamadas al sistema** proporcionan una interfaz con la que los procesos pueden invocar los servicios que el sistema operativo ofrece. El problema es que como se hacen mediante instrucciones en lenguaje ensamblador (véase el Ejemplo 1) no son demasiado cómodas de utilizar. Así que generalmente los programas no las invocan directamente. En su lugar, lo que hacen es llamar a funciones de la **librería del sistema**, que a su vez son las encargadas de hacer las llamadas al sistema necesarias.

Cuando hablamos anteriormente de Windows API y del estándar POSIX, hablábamos de la interfaz de la **librería del sistema** en esos sistemas operativos.

La **librería del sistema**:

- Es parte del sistema operativo, por lo que se distribuye con él.
- Es una colección de clases o funciones que ofrecen los servicios del sistema operativo a los programas, apoyándose en las llamadas al sistema.

  Algunas funciones de la librería del sistema son traducciones literales de llamadas al sistema —por ejemplo, write() o close()— mientras que otras pueden ser más complejas, hacer más trabajo o mostrar conceptos más abstractos que los usados por el sistema operativo al nivel de llamadas al sistema.

- Constituye la verdadera **interfaz de programación de aplicaciones** del sistema operativo. Es la forma recomendada de solicitar servicios al sistema operativo. Invocar directamente las llamadas al sistema debe ser el último recurso.
- Sus funciones se llaman como cualquier otra. Al igual que el resto de librerías, se carga dentro de la región de memoria asignada al proceso. Por lo tanto, la invocación de las funciones de la librería del sistema se realiza como si fueran cualquier otra función del programa.
- Es muy común que esté implementada en C, lo que permite que tanto los programas en C como en C++ la puedan utilizar directamente.

6.4. Librería estándar

Lenguajes distintos de C y C++ pueden tener difícil usar las funciones de la librería del sistema. Pero de alguna forma deben poder hacerlo, porque sus programadores necesitan acceso a los servicios que ofrece el sistema operativo.

Incluso en C y en C++ puede ser interesante tener acceso a funcionalidades adicionales a las ofrecidas por la API del sistema operativo: estructuras de datos, algoritmos de ordenamiento o búsqueda, funciones para manipular cadenas, funciones matemáticas, etc. También abstracciones de los servicios del sistema, que encajen mejor con las particularidades del lenguaje de programación en cuestión. Por ejemplo, utilizando clases y objetos en lenguajes que soportan

programación orientada a objetos.

Por eso, junto a cada intérprete o compilador de cada lenguaje de programación suele ir una **librería estándar** que ofrece clases o funciones con las que los programas pueden acceder a los servicios del sistema operativo y realizar las tareas más comunes de forma más sencilla.

Estas librerías generalmente no forman parte del sistema operativo, sino de las herramientas de desarrollo de cada lenguaje de programación, y constituyen la **interfaz de programación de aplicaciones** del lenguaje al que acompañan.

La **librería estándar** necesita acceder a los servicios del sistema operativo para, a su vez, dar servicio a los programas que la usan. Es decir, cuando un programa invoca alguna función o método de la librería estándar que lo acompaña, es muy probable que esta necesite invocar uno o más servicios del sistema operativo para atender la petición convenientemente. Para ello la **librería estándar** utiliza la **librería del sistema** que acompaña al sistema operativo, que a su vez realiza las **llamadas al sistema** necesarias.

De archivos a flujos

Un ejemplo del papel de las **librerías estándar** lo podemos encontrar en el acceso a los archivos.

Las llamadas al sistema y la librería del sistema de los sistemas operativos ofrecen funciones básicas para manipular archivos. Los archivos se abren indicando su ruta y, al hacerlo, el sistema operativo devuelve un identificador del archivo abierto (véase open()). Este identificador se puede usar para leer o escribir en bytes el contenido del archivo.

Sin embargo en C, C++ y otros lenguajes, todo lo que son flujos de datos se generalizan en el concepto de flujo o *stream* (véase <stdio.h> e std::iostream). En él se incluye la entrada de teclado y la salida por pantalla, la impresión de documentos, las conexiones de red —potencialmente— y, obviamente, el acceso a archivos y a dispositivos.

Los flujos pueden ser de texto o binarios, lo que implica algunas transformaciones en los datos. Además van ligados al concepto del *buffering*, es decir, que los bytes o caracteres escritos en el flujo no se «envían» inmediatamente, sino que se acumulan en la memoria para ser enviados en bloque.

Todas estas características adicionales las implementa la **librería estándar**. Pero por debajo, al final, los datos tienen que ser escritos en un archivo, una impresora o el monitor, recursos que gestiona el sistema operativo. Por lo tanto, las **librerías estándar** necesitan hacer uso de la **librería del sistema** para comunicarse con el sistema operativo.

Algo que suele ocurrir al crear mayores abstracciones es que se suele perder control y características específicas. Por ejemplo, la llamada al sistema open() con la que se pueden crear archivos permite asignar permisos o crear archivos temporales. Sin embargo, con las interfaces de *streams* de C y C++ no se puede hacer eso, ya que los permisos y la temporalidad

son propiedades de los archivos que no son comunes a todas fuentes de flujos de datos.

Así que en ocasiones puede ser que nos resulte más útil llamar a las funciones de la **librería del sistema**, que usar las facilidades de la **librería estándar**. Sin embargo, debemos valorar que así perdemos portabilidad, ya que ahora nuestro programa ya no podrá usarse allí donde haya un compilador o intérprete de nuestro lenguaje, sino solo en sistemas operativos con una **librería del sistema** compatible.

6.5. Con todas las piezas juntas

En la Figura 19 se ilustra el papel de todos los elementos comentados, con el ejemplo de programas en C y Python, ejecutados en Microsoft Windows, que invocan los métodos fopen() y `file()` de la librería estándar de estos lenguajes, respectivamente.

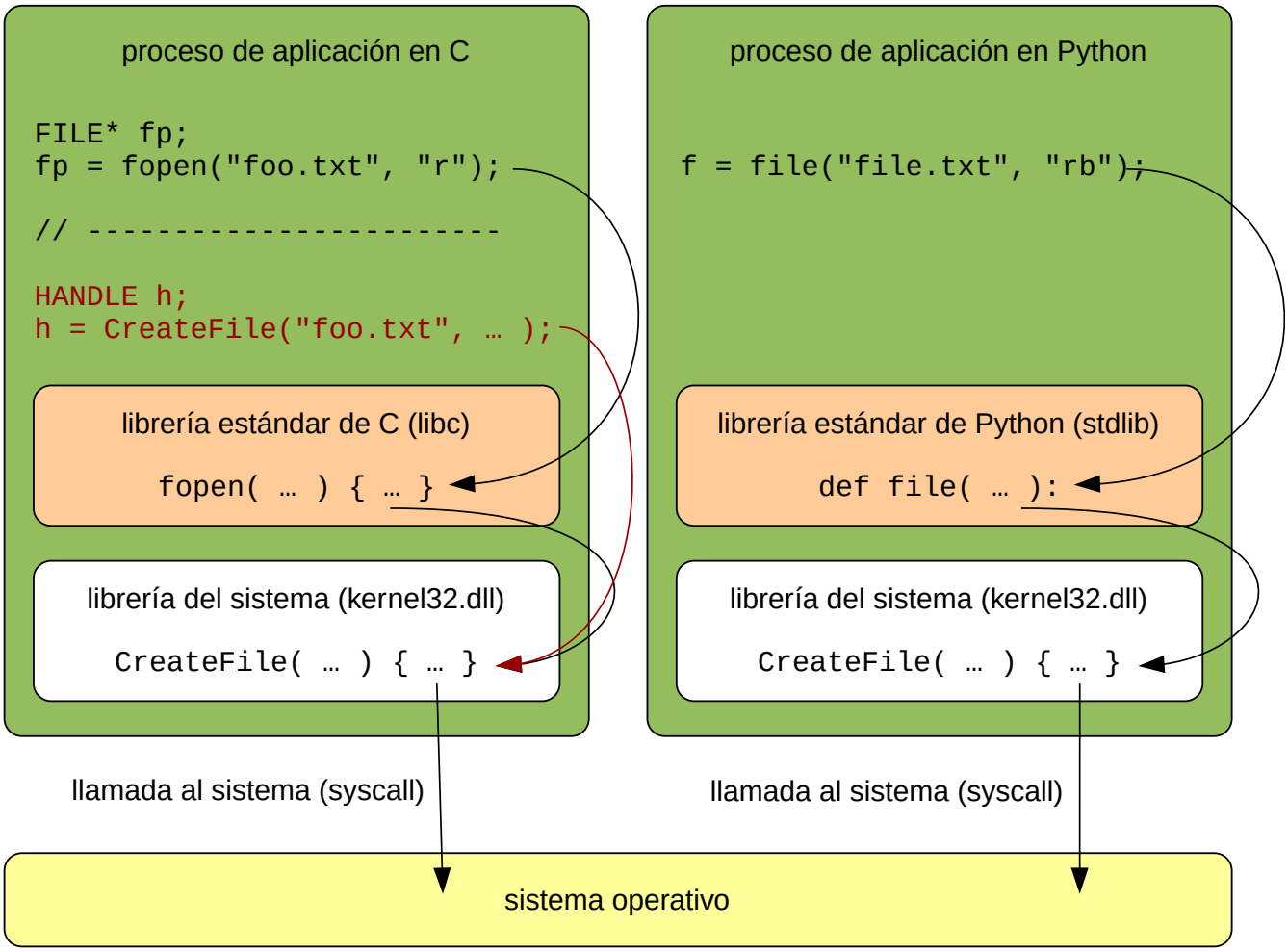


Figura 19. Elementos de la interfaz de programación de aplicaciones en Microsoft Windows.

En ambos casos, la librería estándar llama a la función CreateFile() de la librería del sistema de Windows, que finalmente realiza una llamada al sistema que hace que el sistema operativo tome el control, deteniendo la ejecución del proceso que la solicita. Entonces se realiza la tarea solicitada mediante el funcionamiento coordinado de los diferentes componentes del sistema (véase el Capítulo 4).

El programa en C, puede usar tanto la función fopen() de su librería estándar como llamar directamente a la función CreateFile() de la librería del sistema —marcado en rojo en la Figura

19—. Sin embargo, en el programa en Python no tenemos esa facilidad —al menos directamente—.

Usar directamente las funciones de la librería del sistema desde programas en C o C++ tiene la ventaja de que permite utilizar todas las características del sistema operativo. Por ejemplo, utilizar las opciones adicionales de CreateFile():

```
HANDLE WINAPI CreateFile(
  LPCTSTR lpFileName,                         ①
  DWORD dwDesiredAccess,                      ②
  DWORD dwShareMode,                          ③
  LPSECURITY_ATTRIBUTES lpSecurityAttributes, ④
  DWORD dwCreationDisposition,                ⑤
  DWORD dwFlagsAndAttributes,                 ⑥
  HANDLE hTemplateFile                        ⑦
);
```

① Nombre del archivo.

② Modo de acceso: lectura o escritura.

③ Modo en el que se compartirá el archivo con otros procesos que accedan al mismo tiempo.

④ Permisos del archivo, en caso de crearlo.

⑤ Acción en caso de que el archivo exista o no: siempre crear, solo abrir, truncar si existe, etc.

⑥ Atributos del archivo, en caso de crearlo.

⑦ Archivo abierto del que copiar los atributos para copiarlo en este, en caso de crearlo.

que fopen() no posee:

```
FILE* fopen(
  const char *path, ①
  const char *mode  ②
);
```

① Nombre del archivo.

② Modo de acceso: lectura o escritura.

Sin embargo, debemos tener en cuenta que se pierde portabilidad pues CreateFile() solo está disponible en Microsoft Window, mientras que fopen() viene con la librería estándar de cualquier compilador de C.

En la Figura 20 se puede observar un ejemplo similar en GNU/Linux —un sistema compatible POSIX— pero en esta ocasión con programas en C y C++. En este caso la llamada al sistema es open() y tanto fopen() en C como std::ofstream::open() en C++ la utilizan. Además, ambos lenguajes pueden invocar directamente la librería del sistema —marcado en rojo en la Figura 20— si necesitan alguna característica adicional de la función open().

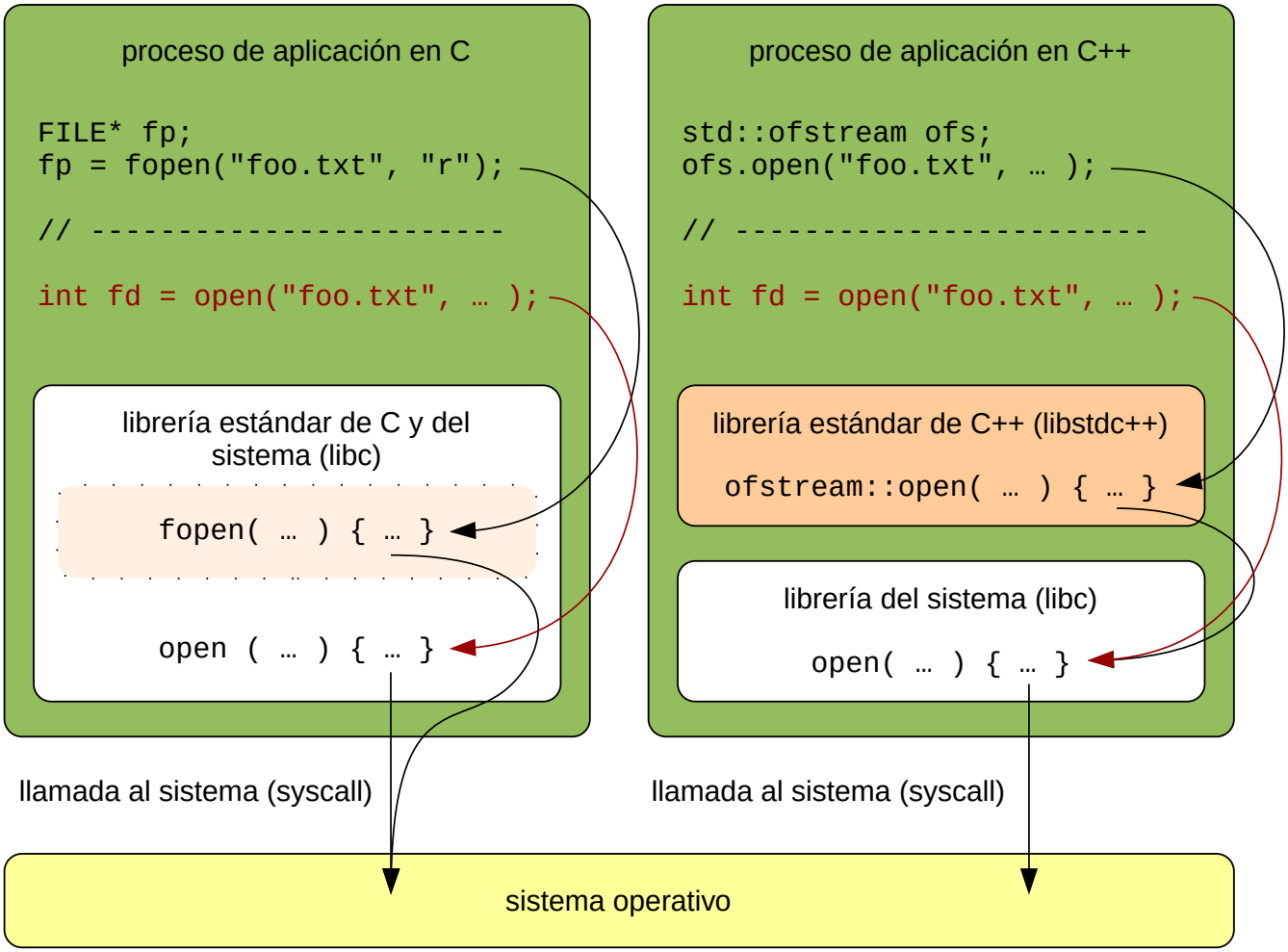


Figura 20. Elementos de la interfaz de programación de aplicaciones en GNU/Linux.

La única diferencia es que en Figura 20 las funciones fopen() y open() están realmente en la misma librería, porque en los sistemas POSIX la librería del sistema y la librería estándar de C pueden ser la misma, dado que el estándar POSIX se diseñó como un superconjunto de la librería estándar de C.

Chapter 7. Operación del sistema operativo

Tiempo de lectura: 17 minutos

Dado que el sistema operativo y los procesos de usuarios comparten los recursos del sistema informático, necesitamos estar seguros de que un error en un programa solo afecte al proceso que lo ejecuta —por ejemplo, que un proceso no puede modificar la memoria de otro proceso o la del núcleo del sistema—. Por eso es necesario establecer mecanismos de protección frente a los errores en los programas que se ejecutan en el sistema.

7.1. Software controlado mediante interrupciones

Antes de entender cómo funcionan estos mecanismos de protección debemos entender que los sistemas operativos modernos pertenecen a un tipo de software que se dice que está controlado mediante interrupciones.

Los sucesos que requieren la atención del sistema casi siempre se indican mediante una interrupción:

- Cuando un proceso comete un error —como una división por cero o un acceso a memoria no válido— o un programa solicita un servicio al sistema operativo a través de una llamada al sistema lo que se genera es una excepción. Esta excepción despierta al sistema operativo para que haga lo que sea más conveniente.
- Cuando un proceso necesita un servicio lo que hace es lanzar una llamada al sistema, que no es más que ejecutar una instrucción que lanza una excepción. Esta excepción despierta al sistema operativo para que atienda la petición.
- Cuando los dispositivos de E/S requieren la atención del sistema operativo —por ejemplo, porque se ha completado una transferencia de datos— se genera una interrupción que despierta al sistema operativo.

Esto funciona así porque el sistema operativo configura la CPU durante el arranque para que si ocurre cualquier interrupción o excepción la ejecución, salte a rutinas en el código del núcleo, con el objeto de darles el tratamiento adecuado.

Si ningún proceso realiza una acción ilegal o pide un servicio, ni ningún dispositivo de E/S pide la atención del sistema, el sistema operativo permanece inactivo esperando a que algo ocurra.

Teniendo todo esto en cuenta podremos entender mejor cómo funciona el modo dual.

7.2. Operación en modo dual

Para proteger el sistema de programas con errores es necesario poder distinguir entre la ejecución del código del sistema operativo y del código de los programas de usuario, de tal forma que el código de los programas de usuario esté más limitado en lo que puede hacer que el del sistema operativo.

El método que utilizan la mayor parte de los sistemas operativos consiste en utilizar algún tipo de

soporte en la CPU que permita diferenciar entre varios modos de ejecución y restringir la utilización de las instrucciones peligrosas —llamadas **instrucciones privilegiadas**— para que solo puedan ser utilizadas en el modo en el que se ejecuta el código del sistema operativo.

7.2.1. Modos de operación

Así que como mínimo son necesarios dos modos de operación diferentes:

- En el **modo usuario**, en el que se ejecuta el código de los procesos de los usuarios. Si se hace un intento de ejecutar una instrucción privilegiada en este modo, el hardware la trata como ilegal y genera una excepción que es interceptada por el sistema operativo, en lugar de ejecutar la instrucción.
- En el **modo privilegiado** —también denominado **modo supervisor**, **modo del sistema** o **modo kernel**— se ejecuta el código de las tareas del sistema operativo. La CPU es la encargada de garantizar que las instrucciones privilegiadas solo pueden ser ejecutadas en este modo.

El modo actual de operación puede venir indicado por un **bit de modo** en alguno de los registros de configuración de la CPU, de tal forma que, si por ejemplo, el bit está a 0, la CPU considera que el código en ejecución opera en modo privilegiado, mientras que si el bit está a 1, el código en ejecución opera en modo usuario.

Comúnmente en el grupo de las **instrucciones privilegiadas** se suelen incluir:

- La instrucción para conmutar al modo usuario desde el modo privilegiado.
- Las instrucciones para acceder a dispositivos de E/S.
- Las instrucciones necesarias para la gestión de las interrupciones. Por ejemplo, para desactivarlas —evitando que se lancen— activarlas y configurarlas.

Niveles de privilegio en procesadores x86

Aunque para operar en modo dual solo se necesita que la CPU admita los dos modos descritos, existen procesadores que soportan más, con la idea de tener mayor control sobre el nivel de privilegio en el que se ejecuta cada componente del sistema.

Es el caso de la arquitectura Intel x86, que soporta 4 modos de operación. El modo 0 es para el software más confiable y el que necesita más privilegios, que generalmente es el núcleo. Mientras que el modo 3 se utiliza para el software menos confiable y que necesita más supervisión, que normalmente son los procesos de usuario.

La idea detrás de tener los modos 1 y 2 es usarlos para controladores de dispositivo o procesos que dan servicio al resto del sistema. Así estos componentes pueden tener mayores privilegios que los procesos de usuario —por ejemplo, los controladores de dispositivo necesitan acceso directo al hardware— pero al mismo tiempo serían supervisados y no podrían afectar al núcleo, que se ejecuta en el modo 0.

Sin embargo, los sistemas operativos con mayor cuota de mercado —incluyendo Microsoft Windows, macOS, Linux y Android— solo utilizan los modos 0 y 3. Los motivos son que los desarrolladores de sistemas no encuentran realmente ninguna ventaja en utilizar más modos y que complica portar el sistema operativo a procesadores donde solo se soporten dos.

En procesadores x86 recientes, que vienen con instrucciones específicas para facilitar la ejecución de máquinas virtuales, se ha incorporado un modo -1, para que el núcleo del sistema operativo virtualizado se ejecute en el modo 0 mientras es supervisado desde el modo -1 por el núcleo del sistema operativo anfitrión.

Para más información, véase «Anillo (seguridad informática) — Wikipedia».

En los procesadores x86 es importante no confundir los **modos real** y **protegido** con el modo dual y los niveles de privilegio de los que estamos hablando.

Por compatibilidad hacia atrás, los procesadores x86 se inician en modo real, donde se comportan como una CPU Intel 8086. En este modo, por ejemplo, solo tienen acceso al primer mega de memoria RAM —ya que los procesadores Intel 8086 solo tenían 20 bits para direcciones de memoria—.

Cuando un sistema operativo moderno arranca, lo primero que hace es iniciar el modo protegido, en el que se activan todas las características de la CPU. Entre otras, el direccionamiento de 32 o 64 bits —según el procesador que sea— y la posibilidad de usar los 4 niveles de privilegio, de los que hemos hablado, para que el núcleo pueda supervisar al resto de componentes.

Para más información, véase «Modo protegido — Wikipedia».

7.2.2. Ejecución de instrucciones

A continuación podemos ver el ciclo de vida de la ejecución de instrucciones en un sistema con modo dual de operación:

1. Inicialmente, al arrancar el ordenador, la CPU se inicia en el modo privilegiado —es decir, en nuestro ejemplo, con el bit de modo a 0—. En este modo se carga el núcleo del sistema operativo e inicia su ejecución.
2. El núcleo del sistema operativo debe cambiar al modo usuario —poniendo el bit de modo a 1— antes de ceder la CPU a un proceso de usuario. Esto ocurre cuando es necesario que un proceso de usuario continúe o inicie su ejecución (véase el Apartado 14.2). Así se asegura que el código de los procesos de usuario siempre se ejecuten en modo usuario, con menos privilegios.
3. La CPU conmuta a modo privilegiado cuando ocurre una interrupción o una excepción —poniendo el bit de modo a 0— antes de comenzar el código del sistema operativo que se encargará de tratarlas.

Esto último es muy importante. Como ya hemos comentado, los sistemas operativos están controlados mediante interrupciones. Al activarse el modo privilegiado cada vez que ocurre una interrupción, podemos estar seguros de que las tareas del sistema operativo se ejecutarán siempre en modo privilegiado.

Cuando se dispone de la protección del modo dual, el hardware se encarga de detectar los errores de ejecución y de notificarlo al sistema operativo mediante excepciones, siendo responsabilidad de este último realizar un tratamiento adecuado de los mismos. Por lo general, si un programa falla de alguna forma —como por ejemplo, intentando utilizar una instrucción ilegal o de acceder a una zona de memoria inválida— el sistema operativo lo hace terminar.

7.3. Protección de la memoria

La memoria principal debe acomodar tanto el sistema operativo como a los diferentes procesos de los usuarios. Por eso la memoria normalmente se divide en dos partes o espacios:

1. La primera parte es el **espacio del núcleo**. Sirve para albergar el núcleo del sistema operativo.

   El sistema operativo puede estar localizado tanto en la parte baja como en la parte alta de la memoria. El factor determinante en la elección es la localización del vector de interrupciones, que es una tabla en la memoria que define las direcciones a las que saltará la CPU en caso de que ocurra una interrupción o una excepción.

   Puesto que en la mayor parte de las arquitecturas este reside en la parte baja de la memoria, normalmente el sistema operativo también se aloja en la parte baja.

2. La segunda parte es el **espacio de usuario** y alberga los procesos de usuario.

Sin embargo, en los sistemas operativos modernos, los procesos no tienen acceso libre a toda memoria física, con el objeto de proteger a los procesos en ejecución y al sistema operativo de posibles errores en cualquiera de ellos:

- El sistema operativo proporciona a cada proceso una «vista» privada de la memoria RAM; de tal forma que el **espacio de usuario** que ve cada proceso es similar al que vería cada uno de ellos si se estuviera ejecutando en solitario (véase la Figura 21).
- A esa «vista» que tiene cada proceso de la memoria es a lo que se denomina **espacio de direcciones virtual** del proceso. Está formada por el conjunto de todas las direcciones que puede generar la CPU para un proceso dado. Por ejemplo, en una CPU de 32 bits el espacio de direcciones virtual tiene 4 GiB, desde la dirección 0x00000000 a 0xFFFFFFFF.
- En los accesos a la memoria principal durante la ejecución del proceso, estas **direcciones virtuales** son convertidas por la CPU en direcciones físicas, antes de ser enviadas a la memoria principal. Por tanto las **direcciones físicas** son las direcciones reales que ve la memoria. Mientras que el **espacio de direcciones físico** es el conjunto de direcciones físicas que corresponden a todas las direcciones virtuales de un espacio de direcciones virtual dado.

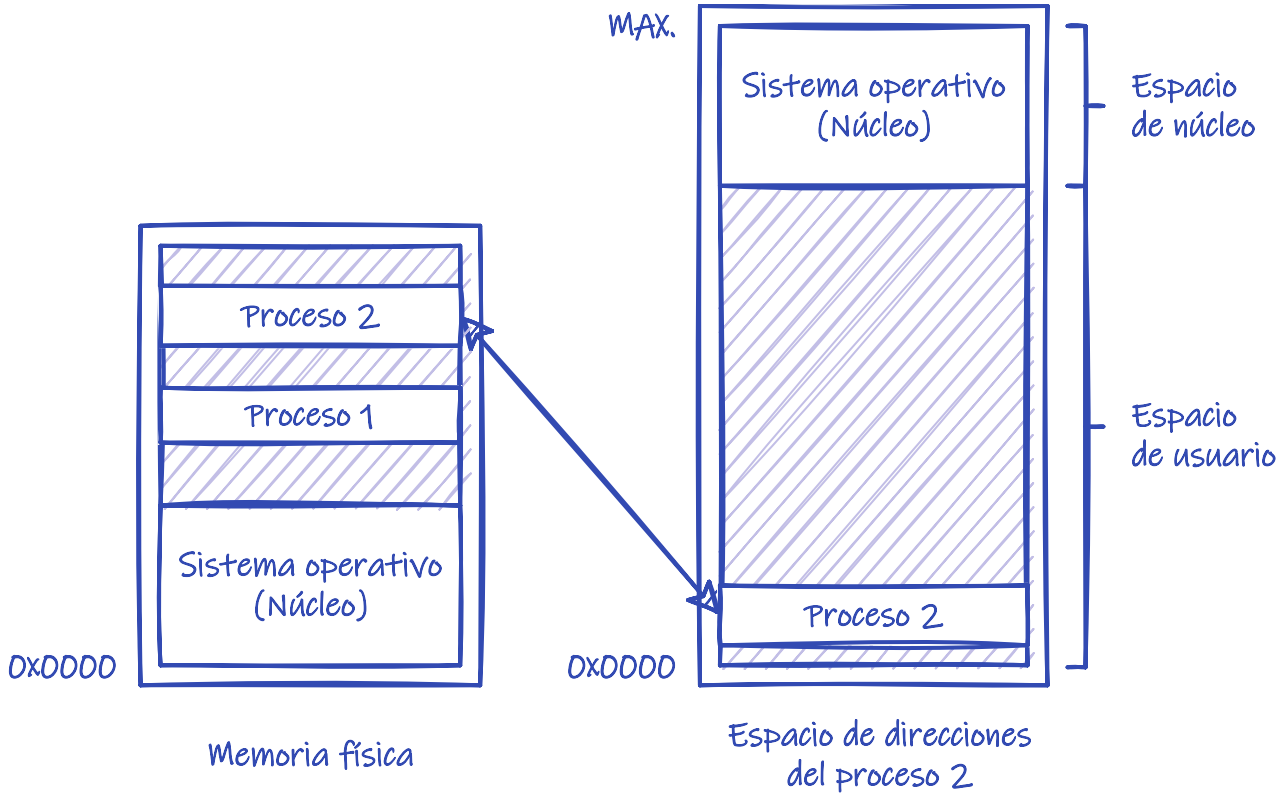


Figura 21. Mapeo de la memoria física en el espacio de direcciones virtual de un proceso.

La conversión de una dirección virtual en una física, la realiza en tiempo de ejecución un componente de la CPU denominado MMU (*Memory-Management Unit*).

Las ventajas de usar esta técnica, desde el punto de vista de la protección de la memoria son:

- Permite el aislamiento de los procesos, creando para cada uno la ilusión de que toda la memoria es para él y evitando que un proceso pueda acceder a la memoria de otros procesos.
- Permite marcar modos de acceso autorizados en las diferentes regiones de la memoria —como por ejemplo lectura, escritura y ejecución— evitando que el código ejecutado en modo usuario tenga acceso a zonas a las que no debería tenerlo. El acceso a la memoria en un modo no autorizado se considera una instrucción privilegiada, por lo que ese tipo de acceso desde el modo usuario siempre genera una excepción. Por ejemplo, si se intenta ejecutar instrucciones en una zona de memoria no marcada con el permiso de ejecución.

7.4. El temporizador

El **temporizador** se configura por el sistema operativo durante el arranque del sistema para interrumpir a la CPU a intervalos regulares. Así, cuando el temporizador interrumpe, el control se transfiere automáticamente al núcleo del sistema. Entonces este puede:

- Conceder más tiempo al proceso en ejecución.
- Detenerlo y darle más tiempo de CPU en el futuro
- Tratar la interrupción como un error y terminar el programa.

El temporizador se utiliza para asegurar que ningún proceso acapara la CPU indefinidamente.

Por ejemplo, un programa mal desarrollado que entra en un bucle infinito, del que no sale jamás.

Obviamente, las instrucciones que pueden modificar el contenido del temporizador son instrucciones privilegiadas.

7.5. Máquinas virtuales

Utilizando las técnicas comentadas anteriormente, el sistema operativo crea a los procesos la ilusión de que se ejecutan en su propio procesador y memoria principal, aunque realmente los comparten. Aun así, los procesos saben que hay un sistema operativo que los supervisa, porque le deben solicitar a él los distintos recursos a través de las llamadas al sistema.

Una máquina virtual también es un proceso en un sistema operativo una máquina real —también llamado sistema operativo anfitrión—. Se utilizan las mismas técnicas para crear la ilusión de que se ejecuta en su propia máquina. Sin embargo, en lugar de llamadas al sistema, el software que gestiona la máquina virtual ofrece una interfaz de hardware virtual. Es decir:

1. El sistema operativo de la máquina virtual intenta acceder al hardware, ya que presupone que se ejecuta en una máquina real.
2. El sistema operativo anfitrión intercepta estos intentos —ya que son instrucciones privilegiadas, prohibidas para los procesos en el modo usuario— y, en lugar de detener el proceso, comunica el suceso al software de gestión de la máquina virtual.
3. El software de gestión de la máquina virtual identifica a qué dispositivo y que intenta hacer el sistema operativo de la máquina virtual en él —para lo que generalmente se utilizan máquinas de estado que simulan el comportamiento del hardware real— y lo transforma en peticiones al sistema operativo anfitrión.

Por ejemplo, los intentos del sistema operativo virtual de acceder a los discos duros del hardware virtual, son convertidos en operaciones sobre un archivo, en un sistema de archivos real que contiene la imagen en disco de la máquina virtual.

7.6. Arranque del sistema

Desde el momento en que el ordenador se pone en marcha hasta que el sistema operativo inicia su ejecución se realizan una serie de operaciones. Estos son los pasos más comunes en el arranque de un sistema:

1. Llega a la CPU una señal de RESET motivada por el encendido del sistema o por un reinicio.
2. La CPU inicializa el contador de programa a una dirección predefinida de la memoria. En esa dirección está el *bootstrap* inicial.

El *bootstrap* es el programa que se encarga en primera instancia del arranque. Debe estar almacenado en una memoria no volátil —ROM o Flash— porque la RAM está en un estado indeterminado en el momento del arranque.

En los PC el *bootstrap* forma parte del *firmware* —sea BIOS o UEFI— de las placas madres.

El término *firmware* viene de que por sus características se sitúa en algún lugar entre el hardware y el software. Concretamente es un componente de software instalado en un dispositivo hardware para encargarse de su control a bajo nivel.

7.6.1. Tareas del bootstrap

El *bootstrap* debe realizar diversas tareas:

1. **Diagnóstico de la máquina** —o *Power-on Self-Test* (POST)—. El *bootstrap* se detiene en este punto si el sistema no supera el diagnóstico.
2. **Inicializar el sistema**. Por ejemplo, configurar los registros de la CPU, inicializar los dispositivos y contenido de la memoria, etc.
3. **Iniciar el sistema operativo**.

Al iniciar el sistema operativo hay que considerar que puede estar en diferentes ubicaciones según el tipo de dispositivo:

- En **consolas de videojuegos, móviles y otros dispositivos empotrados** se almacena el sistema operativo en alguna forma de memoria de solo lectura —ROM o Flash—. Como la ejecución en esas memorias es más lenta que en la RAM, muchas veces el *bootstrap* suele copiar el sistema a la RAM durante el arranque, antes de iniciarlo.
- En **sistemas operativos de gran tamaño** —incluidos los de propósito general— el sistema se almacena en disco.

En los sistemas más antiguos, el *bootstrap* lee de una posición fija del disco —generalmente el bloque 0— el gestor de arranque, lo copia en la memoria y lo ejecuta. Esto es lo que ocurre en los PC más antiguos que utilizan BIOS y particiones MBR.

También se llama MBR a ese bloque 0 del disco donde está el gestor de arranque. De hecho MBR son las siglas de *Master Boot Record* o registro de arranque principal.

Aunque en ocasiones el código de ese bloque inicial de arranque sabe cargar e iniciar el sistema operativo completo, es común que solo sepa donde está el resto del gestor de arranque en el disco, para cargarlo y ejecutarlo. No debemos olvidar que el código cargado por el *bootstrap* debe caber en un solo bloque del disco, que generalmente tiene solo 512 bytes.

En los PC más modernos que utilizan UEFI y particiones GPT, la UEFI tiene la capacidad de leer el sistema de archivo en las particiones para buscar directamente los archivos del gestor de arranque completo. Una vez el *bootstrap* los encuentra, los carga y ejecuta.

En ambos casos, el gestor de arranque completo es el programa que sabe cómo iniciar el sistema operativo así que: explora el sistema de archivos en busca del núcleo del sistema, lo carga e inicia su ejecución.

A partir de esto punto cada sistema operativo prosigue de forma diferente. A modo de ejemplo, veremos como prosigue el arranque en sistemas UNIX en modo texto.

7.6.2. Arranque de sistemas UNIX

Al iniciarse el núcleo del sistema, este realiza una serie de tareas:

1. Configura el sistema para crear un entorno adecuado para la ejecución de los procesos: configuración de interrupciones, configuración de los modos de ejecución —privilegiado y usuario— y de la gestión de la memoria; inicialización de dispositivos y controladores; montaje del sistema de archivos raíz; creación del proceso inactivo —que se ejecutará cuando no haya nada que hacer— etc.
2. Crea el proceso **init** —que por ser el primero tiene PID 1— a partir de la carga del programa `init` almacenado en el sistema de archivos raíz. En los sistemas GNU/Linux actuales el proceso **init** más común es systemd.
3. El planificador de la CPU toma el control de la gestión de la CPU y el núcleo se queda dormido. Puesto que la función del planificador es asignar procesos a la CPU y solo hay uno, el proceso **init**, este es escogido y comienza su ejecución.
4. El proceso **init** lanza los *scripts* encargados de configurar los servicios —también llamados demonios— del sistema. Por ejemplo, para el registro de eventos del sistema, gestión de dispositivos, particiones, impresoras, entre otros.

El proceso **init** también configura el entorno de usuario. Configura las terminales del sistema, inicia un proceso **login** conectado a cada una y se duerme a la espera. Estos procesos **login** son monitorizados por **init** para reiniciarlos en caso de que mueran.

> Aunque, por lo general, un sistema de escritorio tiene una única pareja de teclado y monitor y, por lo tanto, una única terminal real; el sistema suele estar configurado para crear varios terminales virtuales entre los que el usuario puede conmutar usando las combinaciones de teclas adecuadas.

Los procesos **login** se encargan de autenticar a los usuarios y de iniciar y configurar su sesión:

1. Muestran una pantalla de inicio de sesión donde se solicita el nombre del usuario y su contraseña.
2. Autentican al usuario comprobando las credenciales proporcionadas por el mismo.
3. Si la autenticación es positiva, el proceso **login** cambia su identidad actual —generalmente de *root* o administrador del sistema— por la del usuario autenticado, configura la sesión y sustituye su programa actual por el del intérprete de comandos que tiene configurado ese usuario (véase el Apartado 9.7.3.2).

El intérprete de comandos completa la configuración del entorno basándose en sus archivos de configuración, muestra el prompt y queda a la espera del primer comando del usuario.

Chapter 8. Sistemas operativos por su estructura

Tiempo de lectura: 10 minutos

Ya hemos discutido anteriormente acerca de los componentes más comunes en un sistema operativo (véase el Capítulo 4). En esta sección comentaremos cómo se clasifican los distintos sistemas operativos según la organización e interconexión de sus componentes.

8.1. Estructura sencilla

Los sistemas con **estructura sencilla** se caracterizan por:

- No tener una estructura bien definida. Los componentes no están bien separados y las interfaces entre ellos no están bien definidas.
- Son sistemas **monolíticos**, dado que gran parte de la funcionalidad del sistema se implementa en el núcleo.

8.1.1. MS-DOS

Por ejemplo, en el MS-DOS los programas de aplicación podían acceder directamente a toda la memoria y a cualquier dispositivo. Disponiendo de esa libertad un programa erróneo cualquiera podía corromper el sistema completo.

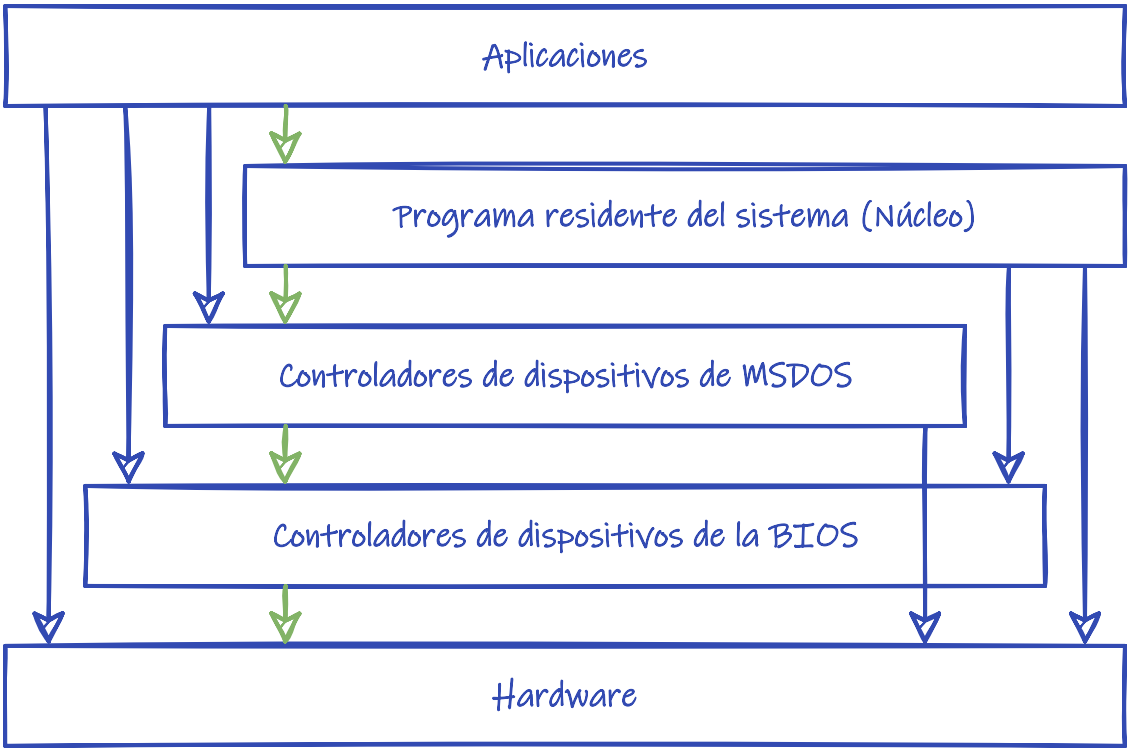


Figura 22. Esquema de la estructura de MS-DOS.

Como el Intel 8086 para el que fue escrito MS-DOS no proporcionaba un modo dual de operación, los diseñadores del sistema no tuvieron más opción que dejar accesible el hardware a los programas de usuario.

8.1.2. UNIX

Otro ejemplo es el de UNIX original, donde sí había una separación clara entre procesos de usuario y código del sistema, pero juntaba un montón de funcionalidad en el núcleo del sistema.

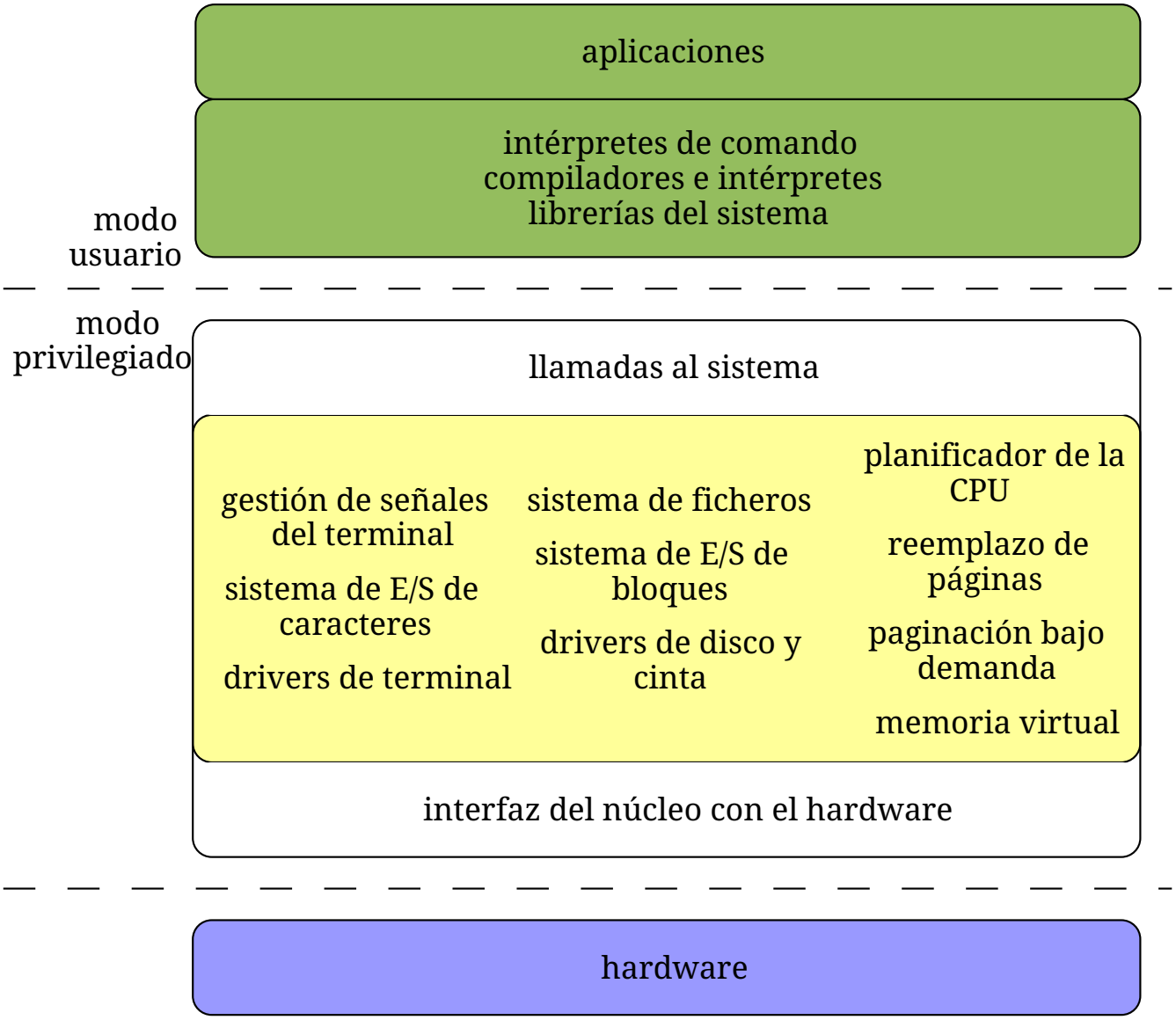


Figura 23. Esquema de la estructura de UNIX.

El núcleo proporciona la planificación de CPU, la gestión de la memoria, el soporte de los sistemas de archivos y muchas otras funcionalidades del sistema operativo. En general se trata de una enorme cantidad de funcionalidad que es difícil de implementar y mantener, si no se compartimenta adecuadamente.

Tanto MS-DOS como UNIX eran originalmente sistemas pequeños y simples, limitados por las funcionalidades del hardware de su época, que fueron creciendo más allá de las previsiones originales. Lo cierto es que con mejor soporte del hardware se puede dividir el sistema operativo en piezas más pequeñas y apropiadas que las del MS-DOS y el UNIX original.

8.2. Estructura en capas

Los sistemas con **estructura en capas** se caracterizan por:

- La funcionalidad se divide en capas, de tal forma que una capa solo utiliza funciones y servicios de la capa inmediatamente inferior y lo hace a través de una interfaz bien

definida.

- Como en la programación orientada a objetos, cada capa oculta a la capa superior los detalles de su implementación. Por ejemplo, las estructuras de datos internas que usa y las operaciones o el hardware de la capa inferior que utiliza.
- Escalan mejor que los sistemas con **estructura sencilla** porque las capas hacen que el código esté mejor compartimentado. Por ejemplo, al corregir un *bug* o añadir una nueva funcionalidad solo hay que preocuparse de su efecto en la capa a la que afecta y no en todo el código del núcleo —siempre que no se altere la interfaz de la capa con el exterior—.
- Ser menos eficiente que la de los sistemas de **estructura sencilla**. En cada capa los argumentos son transformados y los datos necesarios deben de ser transferidos al invocar operaciones en la capa inferior, por lo que cada una añade cierto nivel de sobrecarga al funcionamiento del sistema.
- También son sistemas **monolíticos**, dado que gran parte de la funcionalidad del sistema se implementa en el núcleo, aunque ahora el núcleo esté compartimentado en capas.

Un ejemplo de este tipo de sistemas operativos es el OS/2.

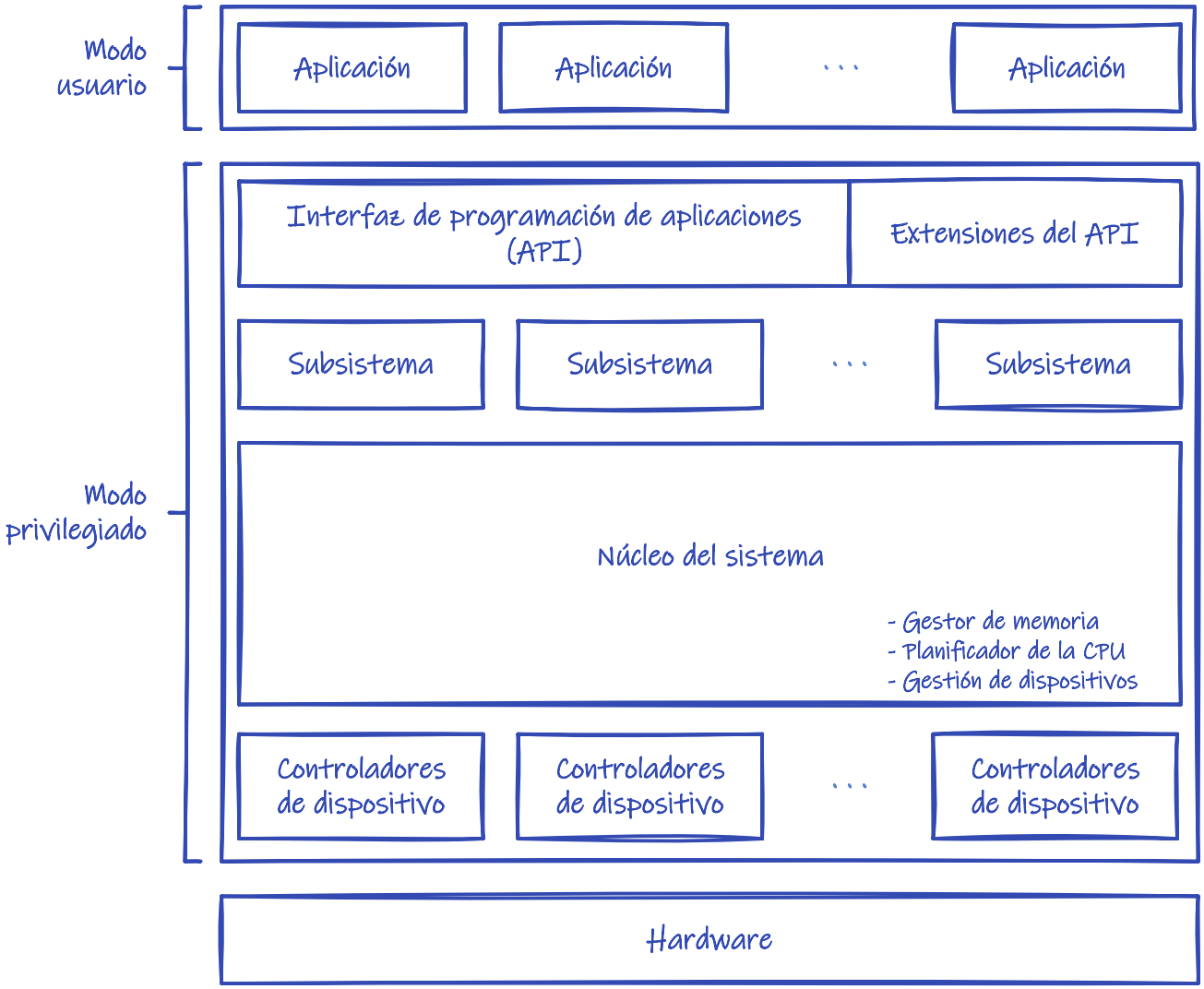


Figura 24. Esquema de la estructura de IBM OS/2.

8.2.1. Dificultades con el diseño

Es importante tener en cuenta que diseñar un sistema con **estructura en capas** no es tan sencillo como pudiera parecer. La definición de las capas y sus funcionalidades debe ser planificada cuidadosamente debido a la restricción, comentada anteriormente, de que una capa solo puede utilizar los servicios de las capas inferiores.

Por ejemplo, el planificador de la CPU suele tener información de los procesos que están en la memoria. Parte de esa información puede ser almacenada en el disco para aumentar la memoria principal disponible. Esto nos debería llevar a pensar que la gestión del almacenamiento secundario debe ir en una capa inferior a la del planificador de la CPU, para que así el segundo pueda pedir al primero que guarde los datos en disco.

Sin embargo, el planificador de la CPU debe asignar la CPU a otro proceso cuando el proceso que actualmente la ocupa solicita alguna operación de E/S —lo típico en multiprogramación—. Como es la gestión del almacenamiento secundario el que debe pedir una operación al planificador de la CPU, ahora el primero debe estar sobre el segundo.

La solución a esta dependencia circular es hacer que ambos componentes estén en la misma capa. Este tipo de dependencias no son raras, ocurre en muchos otros casos, ya que los componentes del sistema operativo suelen depender mucho unos de otros.

Al final, la solución de compromiso es tender hacia sistemas con muy pocas capas donde cada una tiene mucha funcionalidad. Esto limita mucho las ventajas de esta técnica porque no permite compartimentar el núcleo tanto como sería deseable.

8.3. Microkernel

Los sistemas con **estructura microkernel** se caracterizan por:

- Eliminar todos los componentes no esenciales del núcleo e implementarlos como procesos de usuario.
- Un núcleo **microkernel** proporciona funciones mínimas de gestión de procesos y de memoria y algún mecanismo de comunicación entre procesos. Sin embargo, hay que tener en cuenta que hay poco consenso a este respecto, por lo que algunos **microkernel** reales incluyen en el núcleo algunas funcionalidades adicionales.
- El mecanismo de comunicación permite a los procesos de los usuarios solicitar servicios a los componentes del sistema. También sirve para que los componentes del sistema se comuniquen entre sí y se pidan servicio.

Dado que los componentes del sistema están aislados unos de otros —ya que se implementan como procesos de usuario— el mecanismo de comunicación entre procesos es la única forma que tienen los procesos de los usuarios y los componentes, de solicitarles un servicio.

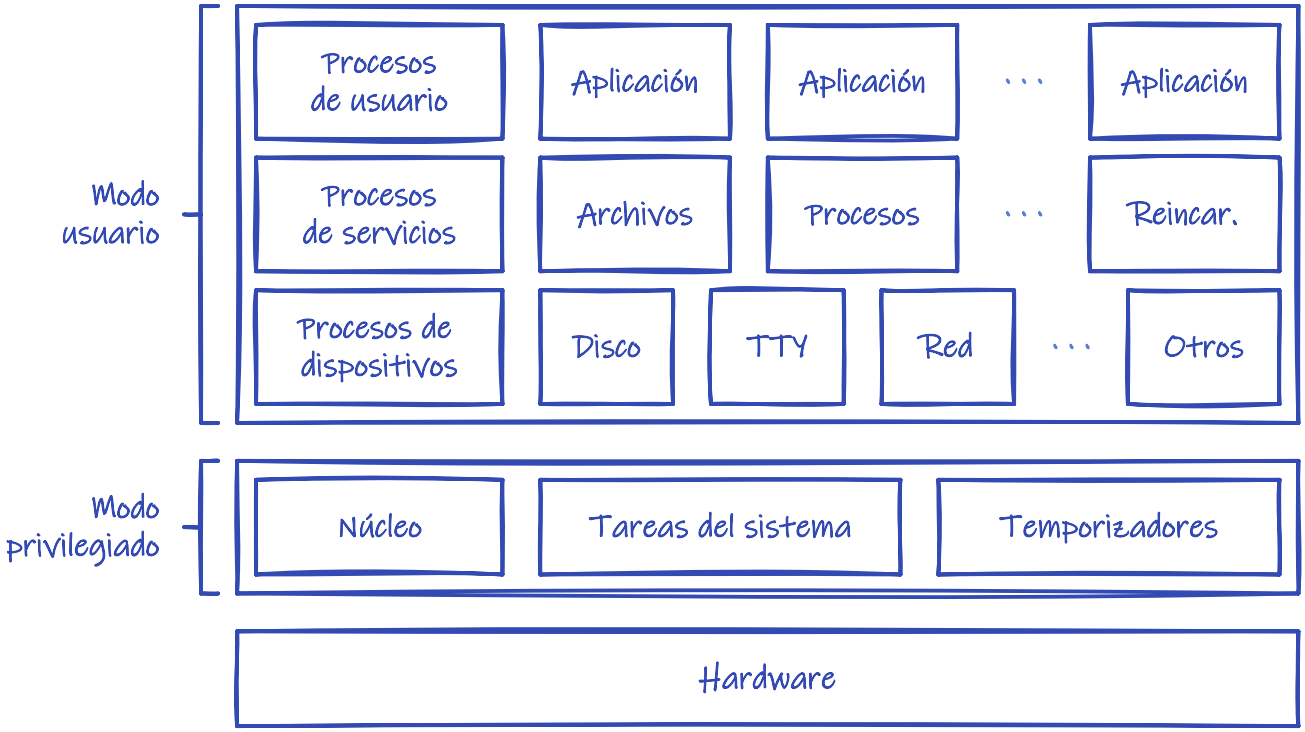


Figura 25. Esquema de la estructura microkernel de MINIX 3.

Generalmente esta comunicación se implementa mediante paso de mensajes (véase el Apartado 9.8.2).

Entre los beneficios de estos sistemas operativos se incluyen:

- **Facilidad a la hora de añadir nuevas funcionalidades**. Los nuevos servicios son añadidos como aplicaciones de nivel de usuario, por lo que no es necesario hacer modificaciones en el núcleo. Desarrollar en el modo privilegiado siempre es más peligrosos que en el modo usuario porque los errores pueden ser catastróficos: bloqueo o caída del sistema, corrupción de datos, etc.
- **Facilidad a la hora de llevar el sistema a otras plataformas**. Puesto que el núcleo es muy pequeño, resulta muy sencillo de portar a otras plataformas.
- **Más seguridad y fiabilidad**. Puesto que la mayor parte de los servicios se ejecutan como procesos separados de usuario, un servicio que falla no puede afectar a otros ni puede ser utilizado para ganar acceso a otros servicios o al núcleo. Además se pueden implementar estrategias para mejorar la tolerancia a fallos, como reiniciar un servicio que ha fallado, como si fuera un programa cualquiera.

8.3.1. Rendimiento

El mayor inconveniente es el pobre rendimiento que puede tener, causado por la sobrecarga que añade el mecanismo de comunicación.

Por ejemplo, Microsoft Windows NT nació con una estructura de **microkernel** en capas donde una parte importante de los servicios eran proporcionados por unos procesos de usuario llamados subsistemas.

El sistema operativo podía mostrar diferentes personalidades o *entornos operativos* —básicamente de OS/2, POSIX y MS-DOS— a través del uso de subsistemas ambientales, que también se ejecutaban como procesos de usuario. Las aplicaciones de Microsoft Windows NT se comunicaban con estos subsistemas utilizando un mecanismo de comunicación denominado LPC (*Local Inter-Process Communication*).

Con esta estructura, la pérdida de rendimiento respecto a Microsoft Windows 95 era tan importante —especialmente en lo relativo a operaciones gráficas— que los diseñadores se vieron obligados a mover más servicios al espacio del núcleo en la versión 4.0. El resultado es que los Windows sucesores a Windows NT 4.0 tienen una arquitectura más monolítica que microkernel, ya que aunque muchos servicios siguen siendo proporcionados por procesos de usuario, esto solo ocurre con aquellos donde el rendimiento no es un factor crítico.

Microsoft Windows XP tiene 280 llamadas al sistema a las que hay que sumar las más de 650 llamadas del subsistema gráfico, que también se aloja en el núcleo desde Microsoft Windows NT 4.0. Mientras que Microsoft Windows NT 3.51 tenía algo menos de 200 llamadas al sistema.

Sin embargo, varios sistemas operativos siguen utilizando núcleos **microkernel**, como QNX o MINIX 3. Ambos son sistemas operativos de tiempo real, que basan en la estructura de **microkernel** su estabilidad como sistema para tareas críticas.

En la Figura 25, por ejemplo, se puede observar un esquema de MINIX 3. El núcleo es muy pequeño —apenas tiene 5000 líneas de código— por lo que la mayor parte de la funcionalidad reside en los procesos de servicios y de controladores de dispositivo.

MINIX 3 es un sistema compatible POSIX. Así que soporta las llamadas al sistema definidas por este estándar, pero estas se convierten en mensajes enviados al servidor correspondiente con la petición, y no en llamadas directas al núcleo. Para que un servidor pueda atender una petición, quizás tenga que enviar peticiones a otros servidores o controladores de dispositivo. Incluso pueden tener que hacer llamadas al núcleo, para solicitar alguna operación privilegiada que no se puede implementar en el modo usuario. Por ejemplo, operaciones de E/S —fundamentales para los controladores de dispositivo— o el acceso a tablas del núcleo —como la tabla de procesos—.

Es este trasiego de mensajes con peticiones y respuestas —y la correspondiente conmutación de procesos en la CPU para ejecutar el proceso que atiende cada mensaje— para resolver una petición de un proceso de usuario, lo que teóricamente justifica el menor rendimiento de los sistemas **microkernel**.

8.4. Estructura modular

Los sistemas con **estructura modular** se caracterizan por:

- Dividir el núcleo en módulos, cada uno de los cuales implementa funciones y servicios concretos y se comunican entre sí a través de una interfaz bien definida.
- Como en la programación orientada a objetos, cada módulo oculta al resto los detalles de

su implementación.

- Todos los módulos pueden llamar a funciones de la interfaz de cualquier otro módulo, a diferencia de los sistemas operativos con **estructura en capas**, donde una capa solo podía usar a la inmediatamente inferior.
- También son sistemas **monolíticos**, dado que gran parte de la funcionalidad del sistema se implementa en el núcleo, aunque ahora el núcleo esté compartimentado en módulos.

Estos núcleos suelen disponer de un pequeño conjunto de componentes fundamentales que se cargan durante el arranque. Posteriormente pueden cargar módulos adicionales, tanto durante la inicialización del sistema como en tiempo de ejecución.

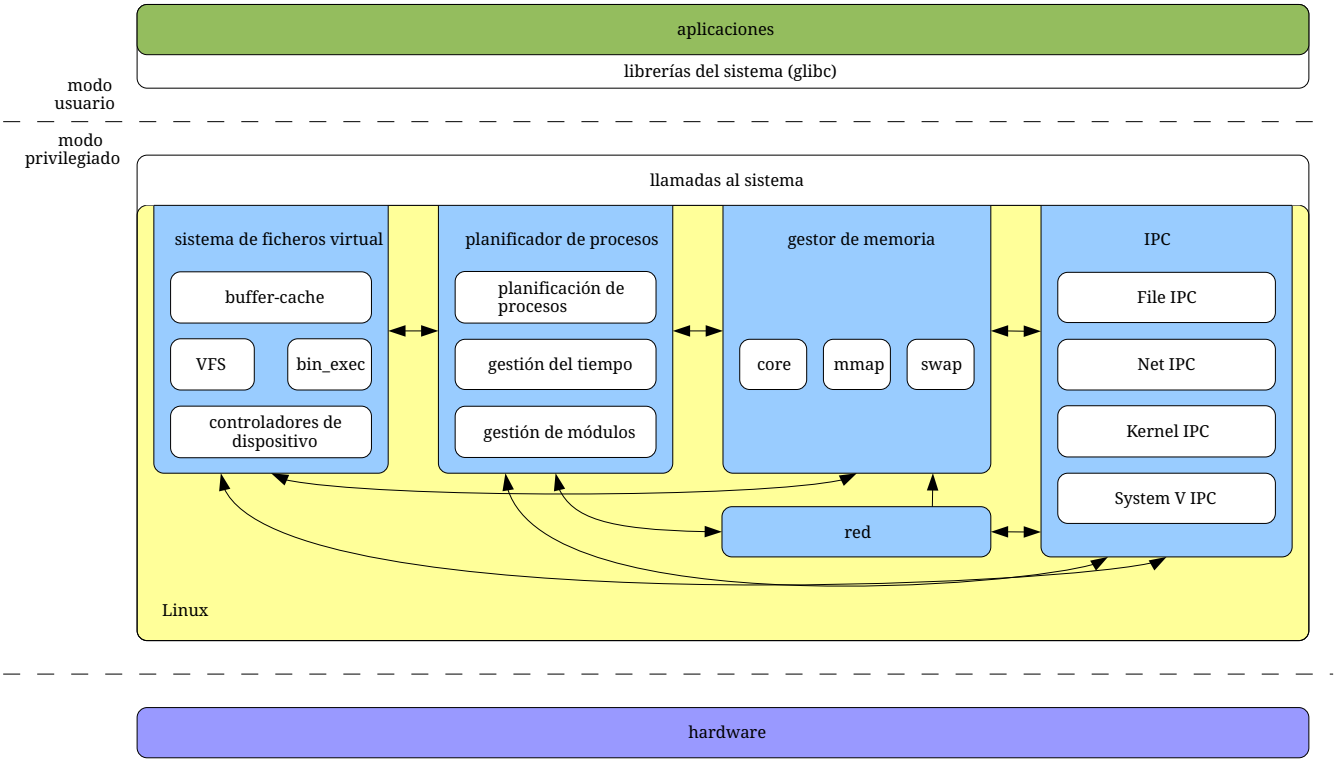


Figura 26. Esquema de la estructura del núcleo Linux.

En este aspecto se asemejan a los núcleos **microkernel**, ya que el módulo principal solo tiene funciones básicas. Sin embargo los núcleos modulares:

- **Son más eficientes** al no necesitar un mecanismo de comunicación, puesto que los componentes se cargan en la memoria destinada al núcleo, por lo que pueden llamarse directamente.
- **Son menos seguros y fiables**, puesto que gran parte de su funcionalidad se ofrece desde el modo privilegiado. Un error en cualquier componente puede comprometer o hacer caer el sistema.

Este tipo de estructura es la utilizada en los UNIX modernos, como Oracle/Sun Microsystems Solaris, FreeBSD, Linux (véase la Figura 26) y macOS.

Parte III: Gestión de procesos

La gestión de procesos es un elemento central de cualquier sistema operativo. Los programas se cargan en la memoria y se convierten en procesos, porque son los procesos los que reciben del sistema operativo los recursos que necesitan para ejecutarse, como: tiempo de CPU, memoria RAM, acceso a archivos o dispositivos; entre otros. Resulta fundamental conocer la **anatomía de los procesos** donde van a ejecutarse nuestros programas y como el sistema operativo hace para asignarles recursos de la mejor forma posible.

Por otro lado, muchas aplicaciones complejas están formadas por diferentes programas y procesos que cooperan entre sí. La **cooperación entre procesos** es la base de cualquier servicio de Internet —como la web o la mensajería instantánea— pero también es muy frecuente entre los procesos de un mismo equipo. Por ejemplo, las distintas aplicaciones de cualquier suite ofimática se tienen que coordinar para compartir datos y contenidos y permitirnos incrustar unos documentos en otros. En los sistemas modernos es común que haya un gestor de impresión, que recibe del resto de procesos los trabajos, los encola y los entrega de forma ordenada a las impresoras conectadas al equipo. E incluso el sencillo editor de texto que usamos para programar nuestros propios programas tiene que cooperar con el compilador, el depurador y la *shell* para ofrecernos buena parte de sus funcionalidades. Cómo pueden comunicarse y coordinarse los procesos, los problemas que surgen y como solucionarlos, son temas que también estudiaremos en esta parte.

Chapter 9. Procesos

Tiempo de lectura: 39 minutos

Los primeros sistemas informáticos solo permitían que un programa se ejecutase cada vez. Dicho programa tenía control completo sobre el sistema y acceso a todos los recursos del mismo. Por el contrario, los sistemas **multitarea** actuales permiten que múltiples programas sean cargados y ejecutados concurrentemente.

Obviamente esta evolución implica un control más fino y la compartimentación de los diversos programas, para que no interfieran unos con otros. Esto, a su vez, conduce a la aparición de la noción de **proceso**, que no es sino la unidad de trabajo en un sistema operativo moderno de tiempo compartido.

Por simplicidad, en este capítulo utilizaremos los términos **trabajo** y **proceso** de forma indistinta. A fin de cuentas tanto los **trabajos** en los antiguos *mainframes* como los **procesos** en los sistemas modernos son la unidad de trabajo en sus respectivos sistemas y el origen de toda actividad en la CPU.

Por último, antes de continuar, es importante señalar que en un sistema operativo hay varios tipos de procesos:

- **Procesos del sistema**. Ejecutan el código del sistema operativo contenido en los **programas del sistema**, que generalmente sirven para hacer tareas del sistema operativo que es mejor mantener fuera del núcleo.
- **Procesos de usuario**. Ejecutan el código contenido en los *programas de aplicación*.

Sin embargo, en lo que resta de capítulo, no estableceremos ninguna distinción entre ellos. En lo que respecta a la gestión de estos procesos en el sistema, no hay ninguna diferencia.

9.1. El proceso

Como ya hemos comentado con anterioridad, un **proceso** es un programa en ejecución (véase el Apartado 4.1 para una definición más completa). Sin embargo, los procesos no solo están compuestos por el código del programa, sino que también son importantes otros elementos:

Segmento de código
: Contiene las instrucciones ejecutables del programa. También es conocido como segmento **text** o **.text**.

Segmento de datos
: Contiene las variables globales y estáticas del programa que se inicializan con un valor predefinido. También es conocido como segmento **.data**.

Segmento BSS
: Contiene las variables globales y estáticas del programa inicializadas a 0 o sin inicialización

explícita. También es conocido como segmento **.bss**.

Pila

Contiene datos temporales, como los parámetros y direcciones de retorno de las funciones y las variables locales.

Montón

Contiene el espacio de la memoria que se asigna dinámicamente durante la ejecución del proceso. También es conocido como **heap**.

Información sobre el estado actual de ejecución

Como el **contador de programa**, los valores de los **registros de la CPU**, el **estado** del proceso y más (véase el Apartado 9.3).

Los **segmentos de código**, **datos** y **BSS** por lo general son secciones dentro del archivo ejecutable que contiene el programa. El resto de elementos los crea el sistema operativo al cargar el programa y crear el proceso.

Como vimos en el Apartado 4.1 varios procesos pueden estar asociados al mismo programa, pero no por eso dejan de ser distintos procesos. Todos tendrán una copia del mismo segmento de código, pero diferente: contador de programa, valores en los registros de la CPU, pila, segmento de datos, montón y demas propiedades.

En la Figura 27 se puede observar la disposición de algunos de estos elementos de un proceso en el espacio de usuario en la memoria.

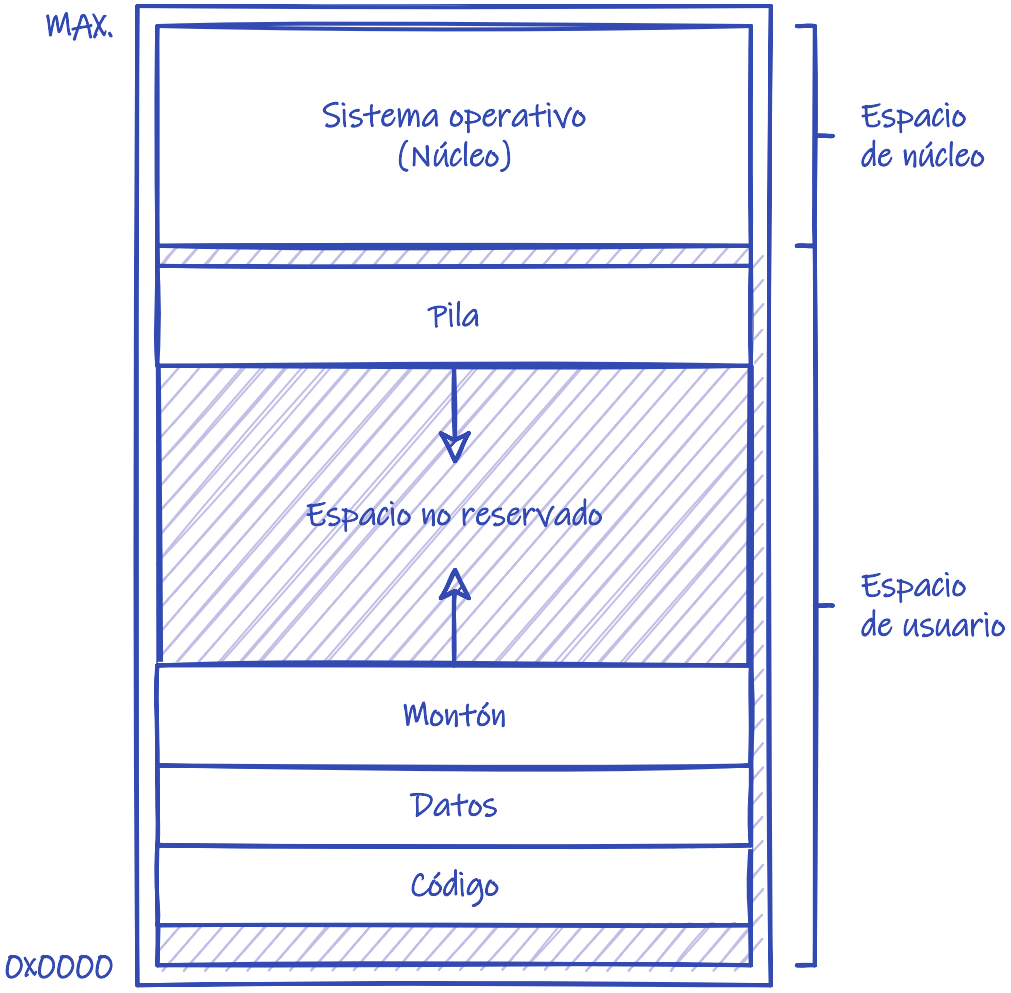


Figura 27. Anatomía de un proceso en memoria.

9.2. Estados de los procesos

Cada proceso tiene un **estado** que cambia a lo largo de su ejecución y que está definido, parcialmente, por la actividad que realiza actualmente el propio proceso.

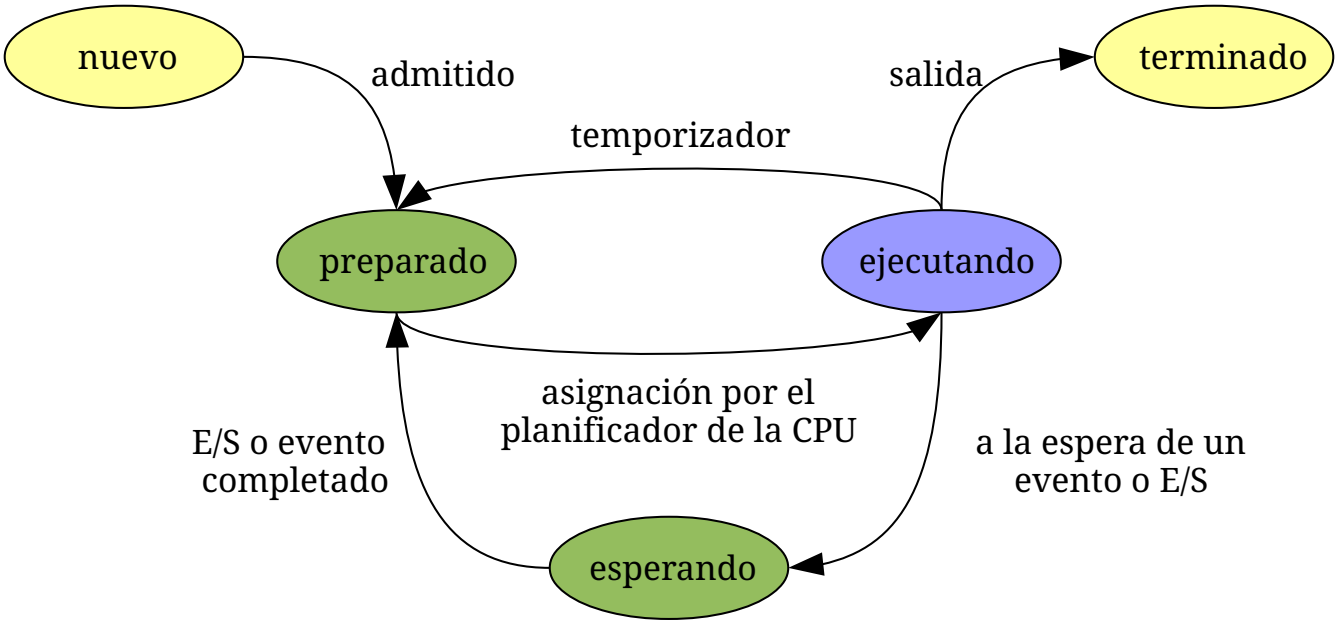


Figura 28. Diagrama de estado de un proceso.

Los estados por los que puede pasar un proceso varían de un sistema operativo a otro, aunque los siguientes son comunes a todos ellos:

Nuevo

El proceso está en proceso de creación. Este estado existe porque la creación de un proceso no es algo instantáneo. Necesita de varias operaciones que pueden tardar tiempo en realizarse, como: reservar memoria libre, cargar el programa en la memoria, inicializar estructuras de datos y configurar el entorno de ejecución.

Ejecutando

El proceso está siendo ejecutado en la CPU. Para eso tiene que haber sido escogido por el planificador de la CPU de entre todos los procesos en estado **preparado**. Solo puede haber un proceso en este estado por CPU en el sistema.

Esperando

El proceso está esperando por algún **evento** como, por ejemplo, que termine una operación de E/S solicitada previamente o que otro proceso termine su ejecución. Múltiples procesos pueden estar en este estado de espera.

Preparado

El proceso está esperando a poder usar la CPU. Múltiples procesos pueden estar en este estado.

Terminado

El proceso ha finalizado su ejecución y espera a que el sistema operativo recupere los recursos que le fueron asignados. Como en el caso del estado **nuevo**, este estado existe porque terminar un proceso no es algo instantáneo.

El diagrama de estados de los procesos, con las transiciones posibles entre ellos, se muestra en la Figura 28.

9.3. Bloque de control de proceso

El **bloque de control de proceso** o **PCB** (*Process Control Block*) es una estructura de datos que representa a cada proceso en el sistema operativo y que guarda información sobre su estado de actividad actual.

En el sistema hay un PCB por proceso y sirve de almacén para cualquier información que puede variar de un proceso a otro:

- **Estado del proceso**. El estado actual del proceso de la lista que hemos visto anteriormente. Por ejemplo: nuevo, preparado, esperando, etc.
- **Contador de programa**. Indica la dirección de la próxima instrucción del proceso que debe ser ejecutada por la CPU. Obviamente, durante el estado **ejecutando** el contador de programa está en el registro correspondiente de la CPU. Su valor se guarda en el PCB al salir el proceso de la CPU para que comience ejecutarse en ella otro proceso.
- **Registros de la CPU**. El valor de los registros de la CPU también forman parte del estado de actividad del proceso. Como en el caso del **contador de programa**, durante el estado

ejecutando los valores están en los registros de la CPU, pero se guardan en el PCB cuando el proceso sale de la CPU para que se ejecute otro proceso.

- **Información de planificación de la CPU**. Incluye la información requerida por el planificador de la CPU. Por ejemplo la prioridad del proceso, punteros a las colas de planificación donde está el proceso, punteros al PCB del proceso padre y de los procesos hijos, etc.
- **Información de gestión de la memoria**. Incluye la información requerida para la gestión de la memoria. Por ejemplo los valores de los registros base y límite que definen el área de la memoria física que ocupa el proceso —en el caso de se use asignación contigua de memoria (véase el Apartado 15.5 o la dirección a la tabla de páginas —en el caso de que se use paginación (véase el Capítulo 16)—.
- **Información de registro**. Aquí se incluye la cantidad de CPU usada, límites de tiempo en el uso de la CPU, estadísticas de la cuenta del usuario a la que pertenece el proceso, estadísticas de la ejecución del proceso, etc.
- **Información de estado de la E/S**. Incluye la lista de dispositivos de E/S reservados por el proceso, la lista de archivos abiertos, etc.

9.4. Colas de planificación

En los sistemas operativos hay diferentes **colas de planificación** para los procesos en distintos **estados**.

Cola de trabajo

Contiene todos los trabajos en el sistema, de manera que cuando entran en el sistema van a esta cola, a la espera de ser escogidos para ser cargados en la memoria y ejecutados. Esta cola existía en los **sistemas multiprogramados**, pero no existe en los sistemas operativos modernos.

Cola de preparados

Contiene a los procesos que están en estado **preparado**. Es decir, procesos cargados en la memoria principal que esperan para usar la CPU. La cola de preparados es generalmente una lista enlazada de PCB, donde cada uno incluye un puntero al PCB del siguiente proceso en la cola.

Colas de espera

Contienen a los procesos que están en estado **esperando**. Es decir, que esperan por un evento concreto, como por ejemplo la finalización de una petición de E/S. Estas colas también suelen ser implementadas como listas enlazadas de PCB y suele haber una por evento, de manera que cuando ocurre algún evento todos los procesos en la cola asociada pasan automáticamente al estado **preparado** y a la **cola de preparados**.

Colas de dispositivo

Son un caso particular de cola de espera. Cada dispositivo de E/S tiene asociada una **cola de dispositivo** que contiene los procesos que están **esperando** por ese dispositivo en particular.

Una manera habitual de representar la planificación de procesos es a través de un diagrama de colas como el de la Figura 29.

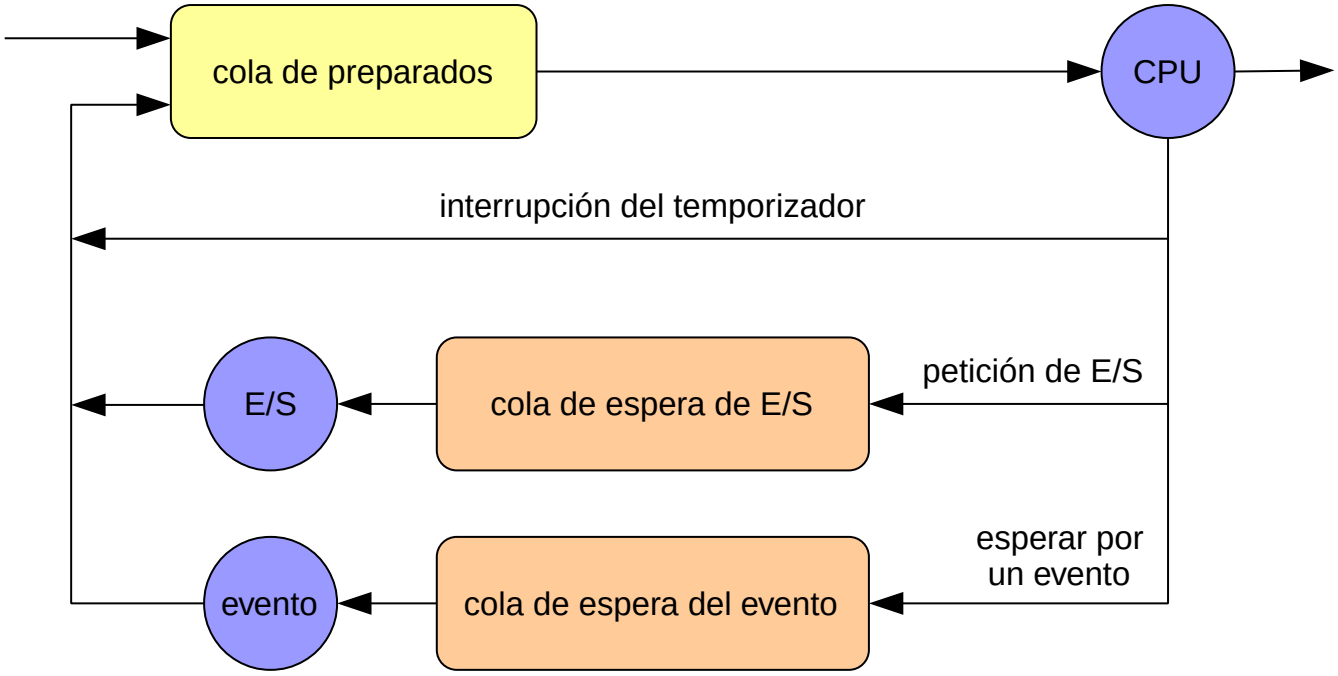


Figura 29. Diagrama de colas de la planificación de procesos.

Analizándolo podemos tener una idea clara del flujo típico de los procesos dentro del sistema:

1. **Un nuevo proceso llega al sistema**. Una vez pasa del estado **nuevo** a **preparado** es colocado en la **cola de preparados**. Allí espera hasta que es seleccionado por el **planificado de la CPU** para su ejecución y se le asigna la CPU. Mientras se ejecuta pueden ocurrir varias cosas:
    - **El proceso solicita una operación de E/S** por lo que abandona la CPU y es colocado en la *cola de dispositivo* correspondiente en estado **esperando**. No debemos olvidar que aunque en nuestro diagrama no exista más que una de estas colas, en un sistema operativo real suele haber una para cada dispositivo.
    - **El proceso puede querer esperar por un evento**. Por ejemplo, puede crear otro proceso y esperar a que termine. En ese caso el proceso hijo es creado, mientras el proceso padre abandona la CPU y es colocado en una **cola de espera** en estado **esperando** hasta que el proceso hijo termine. La terminación del proceso hijo es el evento que espera el proceso padre para salir de la **cola de espera** y entrar en la **cola de preparados** para continuar su ejecución en la CPU cuando sea posible.
    - **El proceso puede ser sacado forzosamente de la CPU**, como resultado de la interrupción del temporizador, que permite determinar cuando un proceso lleva demasiado tiempo ejecutándose, así que es colocado en la **cola de preparados** en estado **preparado**.
2. **Cuando las esperas concluyen, los procesos vuelven a la cola de preparado**, pasando del estado de espera al de preparado.
3. **Los procesos repiten este ciclo hasta que terminan**. En ese momento son eliminados de todas las colas mientras el PCB y los recursos asignados son recuperados por parte del sistema operativo para poder usarlos con otros procesos.

9.5. Planificación de procesos

Durante su ejecución, los procesos se mueven entre las diversas colas de planificación a criterio del sistema operativo. Este proceso de selección debe ser realizado por el **planificador** adecuado:

- El **planificador de largo plazo** o **planificador de trabajos**— selecciona los trabajos desde la cola de trabajos en el almacenamiento secundario —dónde están todos almacenados— y los carga en memoria.

  Este planificador se usaba en los sistemas multiprogramados, donde había cola de trabajos. Los sistemas de tiempo compartido posteriores y los sistemas modernos, carecen de planificador de trabajos, porque los programas se cargan directamente en memoria para ser ejecutados, cuando el usuario lo solicita.

- El **planificador de corto plazo** o **planificador de CPU** selecciona uno de los procesos en la cola de preparados y lo asigna a la CPU. Obviamente este planificador es invocado cuando un proceso en ejecución abandona la CPU, dejándola disponible para otro proceso.
- El **planificador de medio plazo** era utilizado en algunos sistemas para sacar procesos de la memoria cuando escasea y reintroducirlos posteriormente cuando vuelve a haber suficiente memoria libre. A este esquema se le denomina **intercambio** —o ***swapping***.

  Esto era útil en sistemas antiguos donde un proceso tenía que estar cargado completamente en la memoria para poder ejecutarse. Así que si faltaba memoria, se podía suspender un proceso completo, preservar el contenido de su memoria en disco y liberar la memoria ocupada para usarla con otros procesos.

En los sistemas de propósito general modernos no se utiliza **planificador de medio plazo** porque utilizan técnicas de **memoria virtual** (véase el Capítulo 17), que permite mover parte de la memoria de los procesos al disco para liberar memoria, sin tener que suspender su ejecución.

9.6. Cambio de contexto

El **cambio de contexto** es la tarea de asignar la CPU a un proceso distinto al que la tiene asignada en el momento actual. Esto implica salvar el estado del viejo proceso en su PCB y cargar en la CPU el estado del nuevo. Entre la información que debe ser preservada en el PCB se incluyen:

- El **contador de programa**.
- Los **registros de la CPU**.
- El **estado del proceso**.
- La **información de gestión de la memoria**. Por ejemplo, la información necesaria para configurar el espacio de direcciones del proceso.

El cambio de contexto es sobrecarga pura, puesto que no hace ningún trabajo útil mientras se conmuta. Su velocidad depende de aspectos tales como: el número de registros, la velocidad de la memoria y la existencia de instrucciones especiales.

Algunas CPU disponen de instrucciones especiales para salvar y cargar todos los registros de manera eficiente. Esto reduce el tiempo que la CPU está ocupada en los cambios de contexto.

Otra opción es el uso de juegos de registros, como es el caso de los procesadores Sun UltraSPARC e Intel Itanium. Con ellos el juego de registros actual de la CPU se mapea sobre un banco de registros mucho más extenso. Al hacer cambio de contexto, se mapea el juego de registros a otros registros diferentes del banco. Esto permite a la CPU almacenar de forma eficiente el valor de los registros de más de un proceso, sin que en cada cambio de contexto sea necesario copiarlos al PCB del proceso en la memoria principal.

9.7. Operaciones sobre los procesos

En general es necesario que los procesos pueden ser creados y eliminados dinámicamente, por lo que los sistemas operativos deben proporcionar servicios para la creación y terminación de los mismos.

9.7.1. Creación de procesos

Un proceso —denominado **padre**— puede crear múltiples procesos —los **hijos**— utilizando una llamada al sistema específica para la creación de procesos. Cada proceso creado se identifica de manera unívoca mediante un **identificador de proceso** o **PID** (*Process Identifier*), que normalmente es un número entero.

Por ejemplo en sistemas POSIX un programa puede crear otro proceso así:

```
pid_t pid = fork();
```

mientras que en Windows API es así:

```
PROCESS_INFORMATION pi = {0};
if ( CreateProcess( "C:\\Windows\\System32\\charmap.exe", /* ... */, &pi )) ①
{
    DWORD pid = pi.dwhProcessId; ②
    HANDLE handle = pi.hProcess; ③
}
```

① CreateProcess() devuelve TRUE si el proceso se creó con éxito.

② PROCESS_INFORMATION contiene el **identificador de proceso** del nuevo proceso, si CreateProcess() ha tenido éxito.

③ PROCESS_INFORMATION también contiene el manejador del proceso —o *handle* en inglés— que sirve para obtener y manipular el nuevo proceso.

En ambos casos `pid` identifica al nuevo proceso en el sistema. Sin embargo, mientras que los sistemas POSIX ese identificador se puede usar en otras llamadas al sistema para indicar futuras

operaciones sobre el proceso, en Windows lo que se utiliza es el manejador `hProcess` devuelto en PROCESS_INFORMATION.

Obviamente, cada proceso puede obtener del sistema su propio identificador de procesos:

```
/* POSIX API */
pid_t pid = getpid();

/* Windows API */
HANDE handle = GetCurrentProcess();
DWORD pid = GetProcessId( handle );
```

o el de su padre:

```
/* POSIX API */
pid_t parent = getppid();
```

Árbol de procesos

Puesto que cada nuevo proceso puede a su vez crear otros procesos, al final se acaba obteniendo un **árbol de procesos**. En los sistemas POSIX es muy sencillo de ver ejecutando el comando pstree.

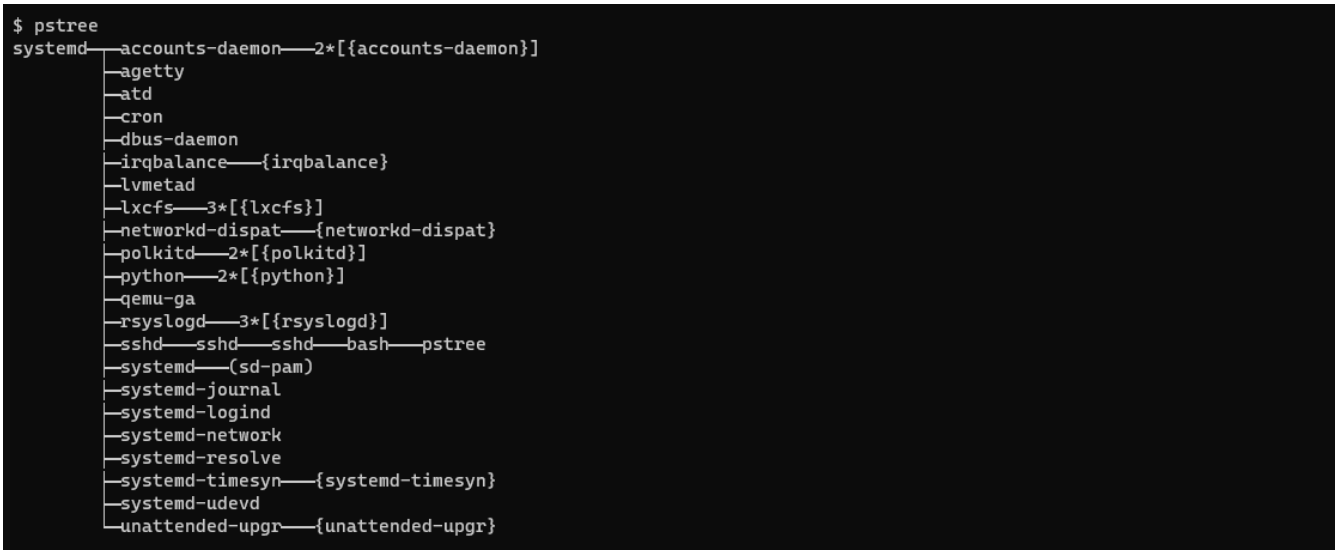


Figura 30. Ejemplo de árbol de procesos mostrador por el comando pstree.

En estos sistemas se conoce como proceso **init** al proceso padre raíz de todos los procesos de usuario. Su PID siempre es 1, ya que es el primer proceso creado por el sistema operativo al terminar la inicialización del núcleo. Por lo tanto, es el responsable de crear todos los otros procesos que son necesarios para el funcionamiento del sistema.

En la Figura 30 se observa que `systemd` es el proceso **init**, como ocurre frecuentemente en muchos sistemas Linux actuales. Anteriormente, lo común es que los sistemas Linux emplearan una implementación de **init** basada en la de los UNIX System V.

Cómo obtienen los procesos hilos los recursos que necesitan

Hay varios aspectos en la creación de los procesos que pueden variar de un sistema operativo a otro. Uno de ellos es cómo obtienen los procesos hilos los recursos que necesitan para hacer su trabajo.

Fundamentalmente existen dos alternativas:

1. Que cada proceso hijo pueda solicitar y obtener los recursos directamente del sistema operativo, compitiendo por los recursos del sistema en las mismas condiciones que el resto de procesos en ejecución. Esta es la opción más común en los sistemas de propósito general actuales, como Microsoft Windows, Android, Linux, macOS, UNIX BSD y muchos otros.
2. Que los procesos hijo solo puedan aspirar a obtener un subconjunto de los recursos de su padre. Esto es interesante en sistemas diseñados para ser muy robustos, ya que evita que un proceso pueda sobrecargar el sistema creando múltiples procesos que consuman demasiada memoria o tiempo de CPU.

En este último caso, el proceso padre puede estar obligado a repartir sus recursos entre los procesos hijo. O puede que el sistema les permita compartir algunos de esos recursos —como memoria o archivos— con algunos de sus hijos.

Cómo pasar parámetros de inicialización a los procesos hijo

Generalmente, el proceso padre suele disponer de algún mecanismo para pasar parámetros de inicialización a sus procesos hijo.

Argumentos de línea de comandos

Por ejemplo, en Windows API un proceso puede usar el segundo argumento de CreateProcess() para indicar al proceso hijo opciones y argumentos de línea de comandos:

```
HANDLE handle = CreateProcess("C:\\holamundo.exe", "/v /s foo.txt bar.png", /* ... */
);
```

Si el proceso hijo está programado en C o C++, podrá acceder a los argumentos `/v`, `/s`, `foo.txt` y `bar.png` a través de los argumentos `argc` y `argv` de la función main() del programa:

```
int main (int argc, char* argv[])
{
    /* . . . */
}
```

de forma que `argv[0]` contendrá `/v`, `argv[2]` contendrá `/s` y así sucesivamente.

> Obviamente, en otros lenguajes de programación se accede de manera diferente a estos argumentos de línea de comandos.

Variables de entorno

Otra forma de pasar parámetros a un proceso hijo es usando las **variables de entorno**, que no son sino variables dinámicas que se pueden crear, leer y modificar durante la ejecución del proceso.

Las **variables de entorno** se gestionan con funciones específicas ofrecidas por la API del sistema operativo:

Tabla 1. Funciones de la API para gestionar variables de entorno.

	POSIX API	Windows API
Leer	getenv()	GetEnvironmentVariable()
Leer todos	environ	GetEnvironmentStrings()
Crear / modificar	setenv()	SetEnvironmentVariable()

por ejemplo, en sistemas POSIX un programa puede leer la variable de entorno `PATH` así:

```
char* path = getenv("PATH");
```

mientras que en Windows API es así:

```
DWORD buffSize = 4096;
TCHAR path[buffSize];
GetEnvironmentVariable("PATH", path, buffSize); ①
```

① El valor de la variable de entorno `PATH` se copia en `path`.

Usando setenv() o SetEnvironmentVariable() de forma similar, cualquier proceso puede crear variables de entorno que serán accesibles a sus procesos hijos, porque por defecto los nuevos procesos heredan un duplicado de las variables de entorno de su proceso padre. Así se pueden pasar parámetros de configuración para alterar el comportamiento de los procesos hijo.

Todas las variantes de sistemas UNIX, así como MS-DOS y todas las versiones de Microsoft Windows soportan variables de entorno.

Herencia de recursos

En algunos sistemas operativos los procesos hijos pueden heredar cierto tipo de recursos del proceso padre, lo que también puede servir para inicializar y alterar el comportamiento del proceso hijo.

Por ejemplo, en los sistemas POSIX todos los archivos abiertos por un proceso son heredados en el mismo estado por sus hijos. Lo interesante es que en estos sistemas muchos recursos se gestionan como archivos. Algunos ejemplos podrían ser: dispositivos de E/S, memoria compartida, tuberías, *sockets* y otros mecanismos de comunicación.

En POSIX todo proceso tiene, por defecto, tres archivos abiertos que corresponden a tres dispositivos de E/S especiales:

- **Entrada estándar**, de donde los procesos leen la entrada del teclado de la terminal.
- **Salida estándar**, donde el proceso escribe para mostrar texto en la pantalla de la terminal.
- **Salida de error**, usada para mostrar errores en la pantalla de la terminal.

Debido a la herencia de los archivos abiertos del proceso padre, todo proceso hijo tiene acceso a estos tres mismos dispositivos. Y a su vez también la tendrán sus hijos y los hijos de estos. De esta manera, todo proceso tiene acceso a los dispositivos de E/S de la terminal donde se ejecuta. Pero también permite a un proceso controlar el destino de la E/S de un proceso hijo —y de los hijos de este—.

Por ejemplo, si antes de crear el proceso hijo sustituye el dispositivo de salida estándar por un archivo real, todo lo que el hijo intente mostrar por pantalla se guardará en dicho archivo, en lugar de mostrarse. Mientras que si lo hace con el dispositivo de entrada estándar, todo lo que pretenda leer de teclado realmente lo leerá de un archivo que el padre puede haber preparado, como si de algún tipo de control remoto se tratara.

Esta misma idea se puede extender a procesos que ofrecen servicios, ya sea a otros procesos del mismo sistema o a redes de ordenadores, como Internet.

Cada conexión con un cliente es como archivo abierto, por lo que los hijos del proceso heredan las conexiones. Así que es común la estrategia de crear un hijo por conexión para que la atienda en nombre del padre, mientras este se encarga de recibir nuevas conexiones.

En Microsoft Windows existe un mecanismo similar pero no por defecto. La función CreateProcess() de Windows API permite indicar si se quiere que el nuevo proceso herede los recursos abiertos. Y también tiene ajustes específicos para la entrada y salida estándar y la salida de error del nuevo proceso.

Qué ocurre con la ejecución del padre

Se suelen contemplar dos posibilidades en términos de la ejecución del padre:

1. Que el padre continúe ejecutándose al mismo tiempo que el hijo. Es lo más común en los sistemas multitarea actuales.
2. Que el padre quede detenido a la espera de que algunos o todos sus hijos terminen. Era lo más frecuente en sistemas monotarea, como MS-DOS.

Cómo se construye el espacio de direcciones de los procesos hijo

En general hay dos posibilidades:

1. Que el espacio de direcciones del proceso hijo sea un duplicado del que tiene el padre. Es decir, que inicialmente el hijo tenga el mismo código y datos que el padre. Es lo que hace fork() en los sistemas POSIX.
2. Que el espacio de direcciones del proceso hijo se cree desde cero y se cargue en él un nuevo programa. Es lo que hace CreateProcess() en Windows. Por eso siempre hay que indicarle el nombre del programa que se quiere ejecutar en el nuevo proceso.

Esto lo veremos con más detalle en el Apartado 9.7.3.

9.7.2. Terminación de procesos

Un proceso termina cuando se lo indica al sistema operativo con la llamada al sistema **exit**. En ese momento puede devolver un valor de estado a su padre.

Esto ocurre en C y C++ incluso si el programa termina usando la sentencia `return` en main(). Lo que ocurre es que es el código, introducido por el compilador, que llamó a main() es el que llama a **exit** usando el valor devuelto por main().

El proceso padre puede esperar a que el hijo termine y recuperar ese valor a través de la llamada al sistema **wait**. Cuando un proceso termina, todos los recursos son liberados, incluyendo: la memoria física y virtual, archivos y dispositivos abiertos, búferes de E/S, etc.

Tabla 2. Funciones de la API para salir, esperar y terminar procesos.

	POSIX API	Windows API
Salir	exit()	ExitProcess()
Esperar (un hijo concreto)	waitpid()	WaitForSingleObject()
Esperar (múltiples hijos)	wait()	WaitForMultipleObject()
Terminar otro proceso	kill()	TerminateProcess()

En todo caso un proceso puede provocar la terminación de otro proceso a través de una llamada al sistema. Por ejemplo, en sistemas POSIX se usa un mecanismo llamado **señales**:

```
kill(pid, SIGTERM);
```

mientras que en Windows API:

```
TerminateProcess(handle);
```

Habitualmente el proceso que invoca estas funciones es el proceso padre, ya que puede que sea el único con permisos para hacerlo.

Los motivos para terminar un proceso hijo pueden ser:

- **El hijo ha excedido el uso de algunos de los recursos reservados**. Obviamente esto tiene sentido cuando los hijos utilizan un subconjunto de los recursos asignados al padre.
- **La tarea asignada al hijo ya no es necesaria**. Por ejemplo, se creó para comprimir un archivo, pero el usuario ha pedido cancelar la operación.
- **El padre termina y el sistema operativo está diseñado para no permitir que el hijo pueda seguir ejecutándose si no tiene un padre**. En esos sistemas, la terminación de un proceso provoca que el sistema operativo inicie lo que se denomina una **terminación en cascada**, en la que termina todos los procesos que cuelgan de dicho proceso.

En sistemas UNIX y estilo UNIX, si un proceso muere a sus hijos no terminan sino que se les reasigna como padre el proceso **init**.

9.7.3. Ejemplos de operaciones con procesos

En C estándar la función system() de la librería estándar permite ejecutar otro proceso, con sus argumentos, esperar a que termine y obtener el valor de estado con el que finalizó el proceso.

```
int status = system("holamundo -v foo.txt");
```

Esta función es portable. Está disponible en cualquier sistema donde haya un compilador de C estándar, pero sus funcionalidades son bastante limitadas. Por ejemplo, no permite que el programa padre continúe su ejecución mientras se ejecuta el hijo, aunque el sistema sea multitarea y ese sea el comportamiento por defecto. Tampoco facilita el control de los recursos que son heredados por el proceso hijo o hacer redirecciones de los dispositivos de E/S estándar.

Como hemos comentado anteriormente, para acceder a todas las funcionalidades ofrecidas por los sistemas operativos, muchas veces es necesario utilizar directamente la librería del sistema.

Windows API

En Windows la librería del sistema ofrece la función CreateProcess(). A diferencia de system(), recibe muchísimos argumentos, ya que permite configurar bastantes aspectos de la creación de un nuevo proceso.

En el Ejemplo 2 se puede ver cómo se usa CreateProcess() para ejecutar un programa y esperar a que termine, de forma similar a como lo hace system().

Ejemplo 2. Crear un proceso usando Windows API

El código fuente completo de este ejemplo está disponible en createprocess.c.

```
STARTUPINFO si = { sizeof(STARTUPINFO) }; ①
PROCESS_INFORMATION pi = {0}; ②

// Crear procesos hijo y comprobar si no se creó con éxito.
if( ! CreateProcess( ③
    NULL, ④
    "C:\\Windows\\System32\\charmap.exe", ④ ⑤
    NULL,
    NULL,
    FALSE, ⑥
    0,
    NULL,  ⑦
    NULL,  ⑧
    &si,
    &pi ))
{
```

```
    fprintf( stderr, "Error (%d) al crear el proceso.\n", GetLastError() ); ⑨
    return EXIT_FAILURE;
}

printf( "[PADRE] El PID del nuevo proceso hijo es: %d\n", pi.dwProcessId );

// Esperar hasta que el hijo termine.
WaitForSingleObject( pi.hProcess, INFINITE ); ⑩

DWORD dwExitCode;
GetExitCodeProcess( pi.hProcess, &dwExitCode ); ⑪
printf( "[PADRE] El valor de salida del proceso hijo es: %d\n", dwExitCode );

// Cerrar los manejadores del proceso y del hilo principal del proceso.
CloseHandle( pi.hProcess ); ⑫
CloseHandle( pi.hThread );
```

① STARTUPINFO sirve para pasar a CreateProcess() parámetros adicionales sobre el inicio de la aplicación, como configurar la redirección de la E/S estándar o características de la primera ventana creada por la aplicación —en aplicaciones con interfaz gráfica—. Si no se va a usar, debe inicializarse a 0, excepto el primer campo que debe contener el tamaño de la estructura.

② PROCESS_INFORMATION sirve para devolver el manejador y el **identificador de proceso** del nuevo proceso. Es común inicializar la estructura a 0.

③ CreateProcess() devuelve `TRUE` o `FALSE`, en función de si ha tenido éxito o no, respectivamente.

④ El primer argumento —`lpApplicationName`— se usa para pasar la ruta del ejecutable, mientras que los argumentos de línea de comando generalmente se pasan por el segundo —`lpCommandLine`—. Si en `lpApplicationName` se indica NULL, se puede pasar todo junto por `lpCommandLine`.

⑤ En `lpCommandLine` indicamos la ruta al ejecutable y los argumentos de la línea de comandos, si hicieran falta.

⑥ Con `bInheritHandles` a `FALSE` señalamos que no queremos que el proceso hijo herede ningún manejador abierto del proceso padre. Estos manejadores son recursos a los que el padre tiene acceso y, si fuera necesario, el hijo también podría tenerlo. Los manejadores pueden representar, por ejemplo, archivos abiertos, tuberías, *sockets* u otros mecanismos de comunicación, procesos o archivos mapeados en memoria, entre muchos otros tipos de recursos.

⑦ Con `NULL` en `lpEnvironment` indicamos que el hijo herede el conjunto de variables de entorno directamente del padre. La otra opción es indicar un nuevo conjunto de variables de entorno.

⑧ `lpCurrentDirectory` sirve para indicar el directorio del trabajo del proceso hijo. Es decir, el directorio respecto al que se resolverán las rutas de archivo relativas. Con `NULL` indicamos que utilice la misma ruta que el proceso padre.

⑨ Si CreateProcess() falla, devuelve `FALSE`. Llamando a GetLastError() obtiene el código que

identifica el motivo del error de la última función utilizada de Windows API.

⑩ Usando WaitForSingleObject() hacemos que el proceso padre se quede en estado **esperando** —sin que pueda seguir ejecutándose— hasta que el proceso hijo termine.

⑪ Cuando el proceso ha terminado, el padre puede conocer su valor de salida. Es decir, el valor usado para terminar en la sentencia `return` de main() o al llamar a ExitProcess() en el programa del proceso hijo. Como convención, el hijo indica con un 0 que terminó con éxito, mientras que con un valor distinto indica que tuvo algún tipo de problema.

⑫ Cuando ya no hace falta obtener información del proceso hijo o manipularlo, es necesario cerrar los manejadores devueltos por CreateProcess(). Así el sistema operativo sabe que las estructuras de datos relacionadas con el proceso hijo ya no son necesarias, por lo que pueden liberarse.

CreateProcess() siempre necesita la ruta a un ejecutable —sea en el primer o en el segundo argumento de la función— porque se utiliza para crear un proceso completamente limpio y ejecutar en él un nuevo programa.

POSIX API

Por el contrario, en los sistemas POSIX se utiliza una estrategia muy diferente. Los nuevos procesos se crean con la llamada fork(), que se encarga de crearlo como una copia del proceso padre.

Ejemplo 3. Crear un proceso en sistemas POSIX

El código fuente completo de este ejemplo está disponible en fork.c.

```
pid_t pid = getpid(); ⑦

// Crear un proceso hijo
pid_t child = fork(); ①

if (child == 0) ②
{
    // Aquí solo entra el proceso hijo
    puts( "[HIJO] ¡Soy el proceso hijo!" );
    printf( "[HIJO] El valor de mi variable 'child' es: %d\n", child ); ②
    printf( "[HIJO] Este es mi PID: %d\n", getpid() ); ④
    printf( "[HIJO] El valor de mi variable 'pid' es: %d\n", pid ); ⑦
    printf( "[HIJO] El PID de mi padre es: %d\n", getppid() ); ⑦

    puts( "[HIJO] Durmiendo 10 segundos..." );
    sleep(10);

    int status = 42;
    printf( "[HIJO] Salgo con %d ¡Adios!\n", status );
    return status; ⑨
}
else if (child > 0) ③ ④
{
```

```
    // Aquí solo entra el proceso padre
    puts( "[PADRE] ¡Soy el proceso padre!" );
    printf( "[PADRE] El valor de mi variable 'child' es: %d\n", child ); ③ ④
    printf( "[PADRE] Este es mi PID: %d\n", getpid() ); ⑦
    printf( "[PADRE] El valor de mi variable 'pid' es: %d\n", pid ); ⑦
    printf( "[PADRE] El PID de mi padre es: %d\n", getppid() );

    puts( "[PADRE] Voy a esperar a que mi hijo termine..." );

    int status;
    wait( &status ); ⑧ ⑨
    printf( "[PADRE] El valor de salida de mi hijo fue: %d\n", WEXITSTATUS(status)
); ⑨

    puts( "[PADRE] ¡Adios!" );
    return EXIT_SUCCESS;
}
else { ⑤
    // Aquí solo entra el padre si no pudo crear el hijo
    fprintf( stderr, "Error (%d) al crear el proceso: %s\n", errno, strerror(
errno) ); ⑥
    return EXIT_FAILURE;
}
```

① El proceso llama a fork() pero al retornar de la llamada vuelven dos procesos: el proceso padre, que es el que llamó originalmente a fork(), y el proceso hijo. Como el proceso hijo es una copia del padre, tiene el mismo código, las mismas variables y los mismos recursos que tenía el padre en el momento de llamar a fork(). La única diferencia es el valor devuelto por fork(), que guardamos en `child`.

② Los dos procesos ejecutan el mismo programa, así que ambos llegan a la línea detrás del fork(). Como queremos que cada proceso haga cosas diferentes, necesitamos que cada uno vaya a ramas distintas del código. Eso se hace comprobando el valor de `child`, porque si vale 0 es que el proceso que actualmente ejecuta el programa es el hijo.

③ Si, por el contrario, el valor de `child` es mayor de 0, el proceso que ejecuta el programa es el padre y el valor de `child` es el PID del proceso hijo creado.

④ Así que el valor de `child` en el padre coincide con el devuelto por getpid() en el hijo.

⑤ Finalmente, si el valor devuelto por fork() es negativo, es que ocurrió un error y el proceso hijo no llegó a crearse.

⑥ En los sistemas POSIX es común que las llamadas al sistema devuelvan un valor negativo para indicar un error. El motivo del error se puede conocer a través de la variable global errno, que siempre guarda el código de identificación del error en la última función invocada de la API POSIX. La función strerror() permite obtener un texto descriptivo de cualquier valor de errno, lo que siempre resulta útil para crear mensajes de error que ayuden a determinar dónde estuvo el problema.

⑦ A modo de ejemplo hemos guardado el PID del proceso en la variable `pid`, antes de la llamada a fork(). Como el proceso hijo es una copia del proceso padre, la variable existe en ambos, pero en el proceso hijo su valor coincide con lo devuelto por getppid() mientras que

en el proceso padre con lo devuelto por getpid().

⑧ wait() hace que el proceso padre interrumpa su ejecución hasta que algún hijo termine y devuelve el estado de salida en `status`.

Debemos asegurarnos de llamar a wait() o waitpid() una vez por cada proceso hijo, en algún momento, porque así es como el sistema sabe que el padre ya no tiene más interés en el proceso y puede liberar su PCB, donde se guarda el estado de salida. No hacerlo genera **procesos zombi** o *defunct*.

⑨ El valor de salida del proceso hijo lo obtiene el proceso padre a través del estado de salida devuelto por wait(). Pero ese estado contiene más información sobre la causa por la que el proceso terminó. Para recuperar el valor de salida se usa la macro `WEXITSTATUS` sobre el estado de salida.

Lo siguiente es un posible resultado de ejecutar el programa anterior en una terminal de Linux, numerado con las anotaciones realizadas al código:

```
$ ./fork
[PADRE] ¡Soy el proceso padre!
[PADRE] El valor de mi variable 'child' es: 2360 ④
[PADRE] Este es mi PID: 2359 ⑦
[PADRE] El valor de mi variable 'pid' es: 2359 ⑦
[HIJO] ¡Soy el proceso hijo!
[PADRE] El PID de mi padre es: 1857
[PADRE] Voy a esperar a que mi hijo termine...
[HIJO] El valor de mi variable 'child' es: 0 ②
[HIJO] Este es mi PID: 2360
[HIJO] El valor de mi variable 'pid' es: 2359 ⑦
[HIJO] El PID de mi padre es: 2359 ⑦
[HIJO] Durmiendo 10 segundos...
[HIJO] Salgo con 42 ¡Adios! ⑨
[PADRE] El valor de salida de mi hijo fue: 42 ⑨
[PADRE] ¡Adios!
```

Aunque pueda parecer algo complejo, esta estrategia facilita la comunicación entre procesos. Es muy sencillo lanzar otro proceso para hacer una tarea en paralelo que tendrá automáticamente una copia de los datos del proceso original.

Como se trata de una copia, las nuevas variables o la modificación de variables existentes que realice cualquiera de los procesos, no serán visibles para el otro. Es decir, después del fork() ambos procesos son completamente independientes. Pero como el proceso hijo hereda el acceso a todo tipo de recursos abiertos por el proceso padre, como: archivos, tuberías, *sockets* o regiones de memoria compartida, entre muchos otros recursos; es muy sencillo crear un canal de comunicación entre ambos procesos, si fuera necesario.

Sin embargo, fork() no proporciona una funcionalidad similar a la de system(). No sirve para crear otro proceso con un programa diferente. Para eso necesitamos exec(), una familia de funciones cuyo propósito es cargar un nuevo programa en el proceso que la invoca.

Ejemplo 4. Ejecutar otro programa en un proceso nuevo en sistemas POSIX

El código fuente completo de este ejemplo está disponible en fork-exec.c.

```
// Crear un proceso hijo
pid_t child = fork(); ①

if (child == 0)
{
    // Aquí solo entra el proceso hijo
    puts( "[HIJO] ¡Soy el proceso hijo!" );
    puts( "[HIJO] Voy a ejecutar el comando 'ls'" );

    /* Hacer otras cosas necesarias antes de ejecutar el programa... */ ④

    execl( "/bin/ls", "ls", "-l", NULL ); ② ③ ⑤

    fprintf( stderr, "Error (%d) al ejecutar el programa: %s\n", errno, strerror
(errno) ); ⑥
    return EXIT_FAILURE; ⑦
}
else if (child > 0)
{
    // Aquí solo entra el proceso padre
    puts( "[PADRE] ¡Soy el proceso padre!" );
    puts( "[PADRE] Voy a esperar a que mi hijo termine..." );

    int status;
    wait( &status ); ⑧
    printf( "[PADRE] El valor de salida de mi hijo fue: %d\n", WEXITSTATUS(status)
);

    puts( "[PADRE] ¡Adios!" );
    return EXIT_SUCCESS;
}
else {
    // Aquí solo entra el padre si no pudo crear el hijo
    fprintf( stderr, "Error (%d) al crear el proceso: %s\n", errno, strerror(
errno) );
    return EXIT_FAILURE;
}
```

① Primero creamos un proceso hijo, donde ejecutaremos el nuevo programa. Si nos diera por llamar directamente a una función de la familia exec(), nuestro programa sería sustituido y no tendríamos ningún control sobre lo que pase después.

② En la rama de código que se va a ejecutar en el hijo —gracias a la comprobación del valor devuelto por fork()— ejecutamos la función de la familia exec() que más nos interese. Esta función no crea otro proceso, sino que carga el programa indicado en el proceso hijo, sustituyendo así a nuestro programa.

③ Todas las funciones de la familia exec() reciben como primer argumento la ruta al ejecutable, pero en `execlp()` en particular, a continuación se indican los argumentos de línea de comandos, tal y como queremos que los reciba el programa en el argumento `argv` de su main(). Es decir, que el programa del comando `/bin/ls` recibirá `ls` y `-l` en `argv[0]` y `argv[1]`, respectivamente. El `NULL` del final indica cuando no hay más argumentos de línea de comandos para pasar.

④ Antes de ejecutar la función exec() se pueden hacer cosas para configurar adecuadamente el proceso donde se ejecutará el nuevo programa. Por ejemplo, cambiar las variables de entorno, redirigir la E/S estándar, cambiar el usuario al que pertenece el proceso —si originalmente se ejecuta con un usuario con ese privilegio— o cerrar archivos abiertos del proceso padre que ha heredado el proceso hijo y que, obviamente, no queremos que se queden abiertos para programas diferentes al nuestro.

⑤ Las funciones exec() no retornan si tienen éxito, porque el programa actual es sustituido por el indicado, que comenzará a ejecutarse de su main().

⑥ Si la función exec() retorna es porque falló y, como es común, el motivo del error está disponible en errno. Un motivo de fallo muy típico es que el ejecutable indicado no exista.

⑦ Si la función exec() retorna, la ejecución del programa en el proceso hijo continúa hasta salir de main(). Generalmente, el proceso hijo no es útil si no puede ejecutar el programa que le hemos indicado. Por eso es importante asegurarnos de que el proceso hijo termina, si exec() falla.

⑧ Mientras todo lo anterior ocurre en el proceso hijo, el proceso padre espera. Cuando el proceso hijo termine, el padre podrá obtener su estado de salir para saber si tuvo éxito o no.

Lo siguiente es un posible resultado de ejecutar el programa anterior en una terminal de Linux, numerado con las anotaciones realizadas al código:

```
$ ./fork-exec
[PADRE] ¡Soy el proceso padre!
[PADRE] Voy a esperar a que mi hijo termine...
[HIJO] ¡Soy el proceso hijo!
[HIJO] Voy a ejecutar el comando 'ls'
total 628 ②
-rwxr--r-- 1 jesus jesus 72640 Sep 16 13:41 fifo-client
-rwxr--r-- 1 jesus jesus 72784 Sep 16 13:41 fifo-server
-rwxr--r-- 1 jesus jesus 20056 Sep 16 13:41 fork
-rwxr-xr-x 1 jesus jesus 19896 Sep 18 13:24 fork-exec
-rwxr--r-- 1 jesus jesus 80744 Sep 16 13:41 mmap
-rwxr--r-- 1 jesus jesus 45712 Sep 16 13:41 pipe
-rwxr--r-- 1 jesus jesus 87024 Sep 16 13:41 shared-memory
-rwxr--r-- 1 jesus jesus 77696 Sep 16 13:41 shared-memory-sync
-rwxr--r-- 1 jesus jesus 19608 Sep 16 13:41 softstack-c
-rwxr--r-- 1 jesus jesus 39328 Sep 16 13:41 softstack-cpp
-rwxr--r-- 1 jesus jesus  9920 Sep 16 13:41 syscall
-rwxr--r-- 1 jesus jesus 40712 Sep 16 13:41 threads-mutex-pthread
-rwxr--r-- 1 jesus jesus 39944 Sep 16 13:41 threads-pthread
[PADRE] El valor de salida de mi hijo fue: 0 ⑧
```

```
[PADRE] ¡Adios!
```

Veamos qué ocurre si la línea de la función exec() fuera:

```
execl( "/bin/ls", "ls", "-l", "/foo", NULL );
```

para intentar ver el contenido del directorio `/foo`, que no existe:

```
$ ./fork-exec
[PADRE] ¡Soy el proceso padre!
[PADRE] Voy a esperar a que mi hijo termine...
[HIJO] ¡Soy el proceso hijo!
[HIJO] Voy a ejecutar el comando 'ls'
ls: cannot access '/foo': No such file or directory ①
[PADRE] El valor de salida de mi hijo fue: 2 ②
[PADRE] ¡Adios!
```

① El comando `ls` se ejecuta, pero falla porque el directorio indicado no existe.

② Por eso el programa, al terminar el proceso, no devuelve 0 si no 2 y es ese el valor que recibe el proceso padre. Esto le permite saber al proceso padre que el comando `ls` no tuvo éxito.

Y finalmente cambiemos la línea de la función exec() así:

```
execl( "/noexists", "ls", "-l", NULL );
```

para que intente ejecutar un programa que no existe:

```
$ ./fork-exec
[PADRE] ¡Soy el proceso padre!
[PADRE] Voy a esperar a que mi hijo termine...
[HIJO] ¡Soy el proceso hijo!
[HIJO] Voy a ejecutar el comando 'ls'
Error (2) al ejecutar el programa: No such file or directory ①
[PADRE] El valor de salida de mi hijo fue: 255 ②
[PADRE] ¡Adios!
```

① exec() falla y se muestra el mensaje de error con el motivo.

② El proceso hijo termina con -1 y así llega ese valor al proceso padre. Al utilizar un valor de salida diferente a los que usa el programa que intenta ejecutar, el padre distingue las terminaciones causadas por errores al llamar a exec() de los errores del propio programa.

Todas las funciones exec() hacen lo mismo. Primero liberan la memoria reservada en el proceso, después cargan el nuevo programa y finalmente inicia la ejecución del programa desde su punto de

entrada. La diferencia entre las distintas funciones está en los argumentos que aceptan. Esa diferencia se puede conocer fijándonos en las letras al final del nombre de cada función:

- **Sin 'p'**, como `execl()` o `execv()`, el primer argumento de la función es la ruta hasta el ejecutable del programa que se quiere ejecutar.
- **Con 'p'**, como `execlp()` o `execvp()`, la función busca el ejecutable como lo hace la *shell*. Es decir, si el primer argumento no contiene ninguna '/' se toma como el nombre del ejecutable y se busca en los directorios listados en la variable de entorno `PATH`. Si el primer argumento contiene alguna '/', se considera una ruta y se busca directamente el ejecutable en ella.
- **Con 'l'**, como `execl()` o `execlp()`, los argumentos de línea de comandos para pasar al programa se indican directamente como argumentos diferentes de la función —por ejemplo `execl("/bin/ls", "ls", "-l", "-a" NULL)`— lo que es ideal cuando el número de argumentos es fijo. La lista de argumentos debe terminar en `NULL`.
- **Con 'v'**, como `execv()` o `execvp()`, los argumentos de la línea de comandos para pasar al programa se indican en un *array* de punteros a cadenas terminadas en '\0', lo que resulta muy práctico si el número de argumentos es desconocido en el momento de compilar. El último elemento del *array* debe apuntar a `NULL`. Por ejemplo:

  ```
  char* argv[] = { "ls", "-l", "-a", NULL };
  execv("/bin/ls", argv);
  ```

- **Con 'e'**, como `execvpe()` o `execle()`, la función admite un argumento adicional para indicar el conjunto de variables de entorno con el que se ejecutará el nuevo programa. Con las otras funciones exec() se conservan las variables de entorno actuales en el proceso que llama a la función.

9.8. Procesos cooperativos

Desde el punto de vista de la cooperación podemos clasificar los procesos en dos grupos:

- Los **procesos independientes**, que no afectan o pueden ser afectados por otros procesos del sistema. Cualquier proceso que no comparte datos —temporales o persistentes— con otros procesos es independiente.
- Los **procesos cooperativos**, que pueden afectar o ser afectados por otros procesos ejecutados en el sistema. Los procesos que comparten datos, sea cual sea la forma en la que lo hacen, siempre son cooperativos.

9.8.1. Motivaciones para la colaboración entre procesos

Hay diversos motivos para proporcionar un entorno que permita la cooperación de los procesos:

- **Compartición de información**. Dado que varios usuarios pueden estar interesados en los mismos bloques de información —por ejemplo, en un archivo compartido— el sistema operativo debe proporcionar un entorno que permita el acceso concurrente a este tipo de recursos.

- **Velocidad de cómputo**. Para que una tarea se ejecute más rápido se puede partir en subtareas que se ejecuten en paralelo. Es importante destacar que la mejora en la velocidad solo es posible si el sistema tiene varios componentes de procesamiento como procesadores —si se quiere acelerar la ejecución en la CPU— o canales E/S —si se quieren acelerar las operaciones de E/S —.
- **Modularidad**. Podemos querer crear nuestro software de forma modular, dividiendo las funciones del programa en procesos separados que se comunican entre sí.
- **Conveniencia**. Incluso un usuario individual puede querer hacer varias tareas al mismo tiempo. Por ejemplo, editar, imprimir y compilar al mismo tiempo.

La ejecución simultánea de procesos cooperativos requiere mecanismos tanto para comunicar unos con otros como para sincronizar sus acciones (véase el Capítulo 13).

9.8.2. Comunicación entre procesos

Para comunicar procesos cooperativos existen diversas aproximaciones, que en general se pueden encajar en alguna de las siguientes estrategias:

Memoria compartida

Método de comunicación en el que los procesos utilizan regiones compartidas de la memoria principal para compartir información.

Paso de mensajes

Método en el que los procesos utilizan funciones del sistema operativo para enviarse mensajes entre ellos, compartiendo información y sincronizando acciones, sin necesidad de compartir memoria.

En la Figura 31 se puede un esquema comparativo entre ambos modelos de comunicación. Veremos cada uno en detalle en el Capítulo 11 y el Capítulo 10, respectivamente.

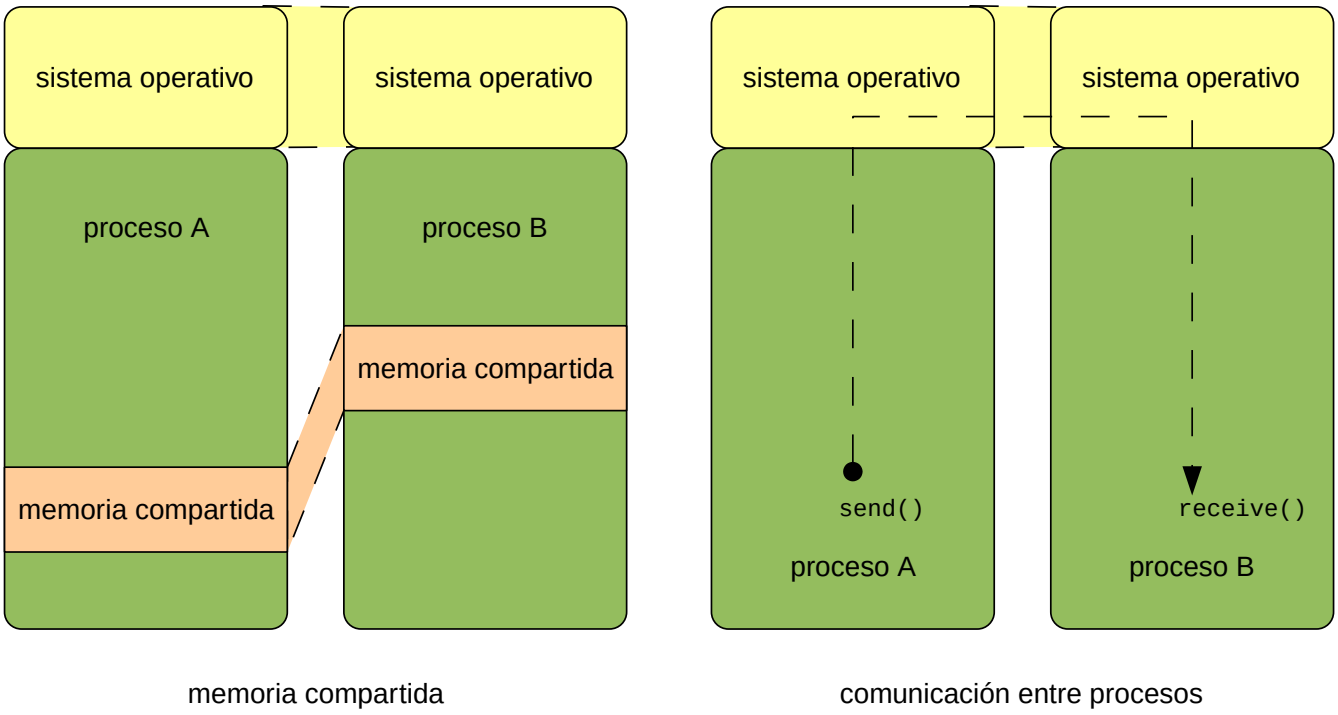


Figura 31. Modelos de comunicación.

Chapter 10. Comunicación mediante paso de mensajes

Tiempo de lectura: 34 minutos

El **paso de mensajes** es un mecanismo que permite a los procesos compartir información y sincronizar sus acciones sin necesidad de compartir recursos —compartir memoria, archivos, etc.—

Esto lo hace especialmente útil en entornos distribuidos, donde los procesos a comunicar residen en ordenadores diferentes conectados a una red, por lo que tiene muy difícil —o incluso imposible— compartir memoria u otros recursos para comunicarse. En este caso, el sistema operativo es el encargado de codificar los mensajes y enviarlos a través de la red para hacerlos llegar a su destinatario. La web —donde un navegador se conecta a un servidor web para obtener contenido— y el resto de servicios de Internet son ejemplos de sistemas de paso de mensajes.

El sistema de paso de mensajes debe ser proporcionado por el sistema operativo que, a diferencia de cuando se usa memoria compartida, se encarga de la sincronización —ya que no existen riesgos en el envío y recepción de mensajes al mismo tiempo— y de establecer el formato que deben tener los datos del mensaje.

En algunas fuentes se sigue haciendo referencia al término **IPC** (*Interprocess Communication*) —o **comunicación entre procesos**— para identificarlo exclusivamente con **sistemas de paso de mensajes**. Sin embargo, la **memoria compartida** y otras técnicas también sirven para «comunicar procesos», por lo que es más adecuado usar el término **IPC** para englobar todas las técnicas conocidas de comunicación entre procesos.

Los sistemas de paso de mensaje de cualquier sistema operativo debe proporcionar al menos dos llamadas al sistema similares a:

- **send(message)** para mandar mensajes a otro proceso.
- **receive(&message)** para recibir mensajes de otro proceso y copiarlo en *message*.

Vamos a hablar de funciones de un sistema de paso de mensajes hipotético. Meros ejemplos para ilustrar las diferentes alternativas. Esto no significa que en los sistemas operativos reales las funciones se llamen así y tengan esos mismos argumentos.

Para que estas llamadas puedan enviar y recibir mensajes entre dos procesos es necesario que haya un **enlace de comunicaciones** entre ambos. No trataremos aquí la implementación física del enlace —que por ejemplo puede ser mediante memoria compartida, un bus hardware o una red de ordenadores— sino de su implementación lógica, es decir, las características de la interfaz que usan las aplicaciones para comunicarse con sus correspondientes operaciones de envío y recepción.

10.1. Tamaño del mensaje

Los diseñadores del sistema operativo deben escoger entre implementar un sistema de paso de mensajes con mensajes de tamaño fijo o mensajes de tamaño variable:

- **Mensajes de tamaño fijo**. La implementación del sistema operativo es muy sencilla, pero el uso de la interfaz por parte de las aplicaciones es mucho más compleja.

  Por ejemplo, para comunicar procesos en un mismo ordenador cada enlace puede tener un búfer de tamaño fijo donde se copia el mensaje enviado y de donde se extrae el mensaje al recibirlo. Esto es muy sencillo de implementar en el sistema operativo. Sin embargo, si el desarrollador de la aplicación quiere enviar algo de mayor tamaño que el tamaño del mensaje, debe trocearlo en varios mensajes para enviarlo y reconstruirlo al recibirlo.

- **Mensajes de tamaño variable**. La implementación del sistema operativo es más compleja, ya que ahora tiene que gestionar la memoria para almacenar mensajes de tamaño variable hasta que son recibidos. Sin embargo, la programación de aplicaciones es más simple, puesto que el programador puede mandar mensajes de cualquier tamaño sin ninguna preocupación

10.1.1. Comunicación orientada a flujos

En algunos sistemas con **mensajes de tamaño variable** no se preserva la separación entre mensajes al recibirlos. Es decir, que los procesos leen un número arbitrario de bytes, donde puede haber parte de un mensaje o varios mensajes al mismo tiempo. Por ejemplo, en esos sistemas el transmisor puede mandar tres mensajes de 16000, 3200 y 100 bytes, pero el receptor leer la secuencia de bytes en bloques de 512 bytes.

A esto se lo denomina **comunicación orientada a flujos** o ***(streams)***. Si usamos este tipo de sistema es importante conservar la separación entre los mensajes recibidos, será nuestra responsabilidad escoger un formato de mensaje adecuado que permita al receptor recuperar dónde comienza y termina un mensaje dentro de la secuencia de bytes.

10.2. Referenciación

Los procesos que se quieran comunicar deben tener una forma de señalarse el uno al otro. Para ello el diseñador del sistema puede elegir que el sistema de paso de mensajes sea con comunicación directa o indirecta.

10.2.1. Comunicación directa

En la **comunicación directa** cada proceso debe nombrar explícitamente al proceso destinatario o receptor de la información. Por ejemplo, ahora las llamadas al sistema básicas podrían ser así:

- **send(A, message)** para mandar un mensaje al proceso identificado como «A».
- **receive(A, &message)** para recibir un mensaje del proceso identificado como «A», copiándolo en «*message*».

De hecho el ejemplo anterior corresponde a un caso de **comunicación directa** con

direccionamiento simétrico, pero existe una variante de ese mismo esquema denominado **direccionamiento asimétrico** donde el receptor puede recibir mensajes de cualquier proceso, de forma que al volver de la llamada recibe el mensaje y la identidad del remitente.

- **send(A, message)** para mandar un mensaje al proceso identificado como «A»
- **receive(&pid, &message)** para recibir un mensaje de cualquier proceso, recibiendo en «*message*» una copia del *message* y en «pid» la identidad del remitente.

En resumen:

- En el **direccionamiento simétrico** tanto el proceso que envía como el que recibe tienen que identificar al otro para comunicarse.
- En el **direccionamiento asimétrico** solo el proceso que envía identifica a que recibe, mientras que este último no tiene que nombrar al remitente. Es el sistema operativo el que informa de quién es el remitente del mensaje que se ha recibido.

Un **enlace de comunicaciones** según este esquema tiene las siguientes características:

- Un enlace se establece automáticamente entre cada par de procesos que quieren comunicarse. Por tanto, los procesos solo necesitan conocer la identidad de los otros para comunicarse.
- Cada enlace se asocia exactamente a dos procesos.
- Entre cada par de procesos solo hay un enlace.

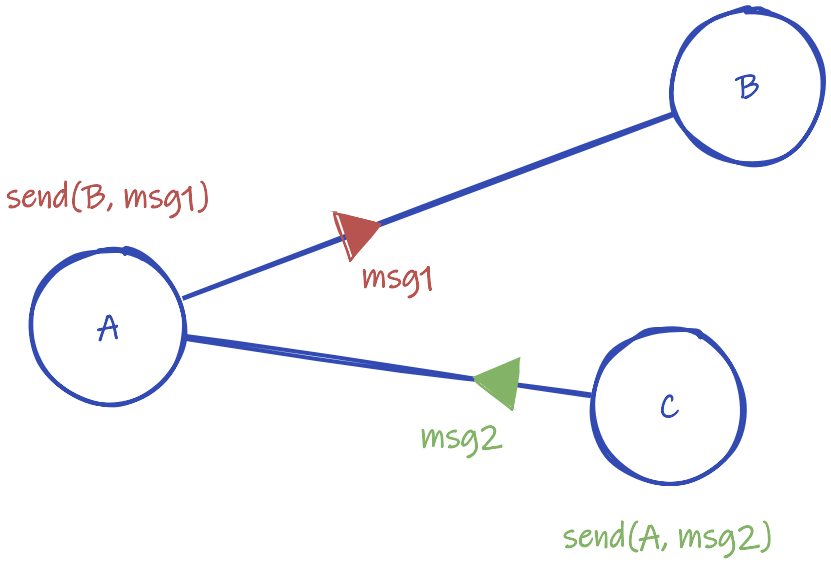


Figura 32. Comunicación directa.

La principal desventaja de este tipo de comunicación es que si cambia el identificador de un proceso hay que actualizar todas las referencias en todos los procesos que se comunican con él. En general cualquier técnica que requiera que los identificadores de los procesos sean establecidos explícitamente en el código de los programas no es deseable, puesto que en muchos sistemas los identificadores de los procesos cambian de una ejecución a otra. Por lo tanto, lo mejor sería disponer de una solución con un nivel adicional de indirección que evite que los procesos usen sus identificadores para comunicarse.

Colas de mensajes en Windows API

En Windows API un hilo puede enviar mensajes a otro hilo usando PostThreadMessage(). Como aún no hemos visto el concepto de hilo, podemos asumir que es equivalente al de proceso.

```
BOOL PostThreadMessage(
    DWORD  idThread, ①
    UINT   Msg, ②
    WPARAM wParam, ③
    LPARAM lParam ③
);
```

① Identificador del hilo.

② Un número entero que identifica el mensaje.

③ Parámetros del mensaje de tipo entero.

Como se puede observar, en las colas de mensajes de Windows API el **tamaño del mensaje es fijo** y con una estructura muy bien definida: un identificador del mensaje y dos enteros que sirven de parámetros opcionales del mensaje.

Para recibir el mensaje el proceso llama a GetMessage():

```
BOOL GetMessage(
    LPMSG lpMsg, ①
    HWND  hWnd,
    UINT  wMsgFilterMin,
    UINT  wMsgFilterMax
);
```

① Puntero a una estructura MSG que a la vuelta contendrá el identificador y los parámetros del mensaje recibido, entre otra información.

Como se puede observar, no se indica de qué hilo o proceso se quiere recibir el mensaje, por lo que se trata de un caso de **comunicación directa asimétrica**. De hecho, si se quiere conocer la identidad del remitente, este tendría que poner su identificador en alguno de los parámetros del mensaje.

El sistema de colas de mensajes de Windows API es una pieza fundamental del entorno gráfico de Microsoft Windows. Con ese fin, el sistema trae un conjunto de mensajes predefinidos, pero podemos definir nuestros propios mensajes para comunicar unos hilos o procesos con otros.

10.2.2. Comunicación indirecta

En la **comunicación indirecta** los mensajes son enviados a **buzones**, ***maillox*** o **puertos** que son

objetos donde los procesos pueden dejar y recoger mensajes.

- **send(P, message)** para mandar un mensaje al puerto «P»
- **receive(P, &message)** para recibir un mensaje del puerto «P».

Un **enlace de comunicaciones** según este esquema tiene las siguientes características:

- Un enlace se establece entre un par de procesos solo si ambos comparten un mismo puerto, dado que cada enlace corresponde con un puerto.
- Un enlace puede estar asociado a más de dos procesos, puesto que múltiples procesos pueden compartir el mismo puerto.
- Entre cada par de procesos en comunicación puede haber varios enlaces, cada uno de los cuales corresponde a un puerto.

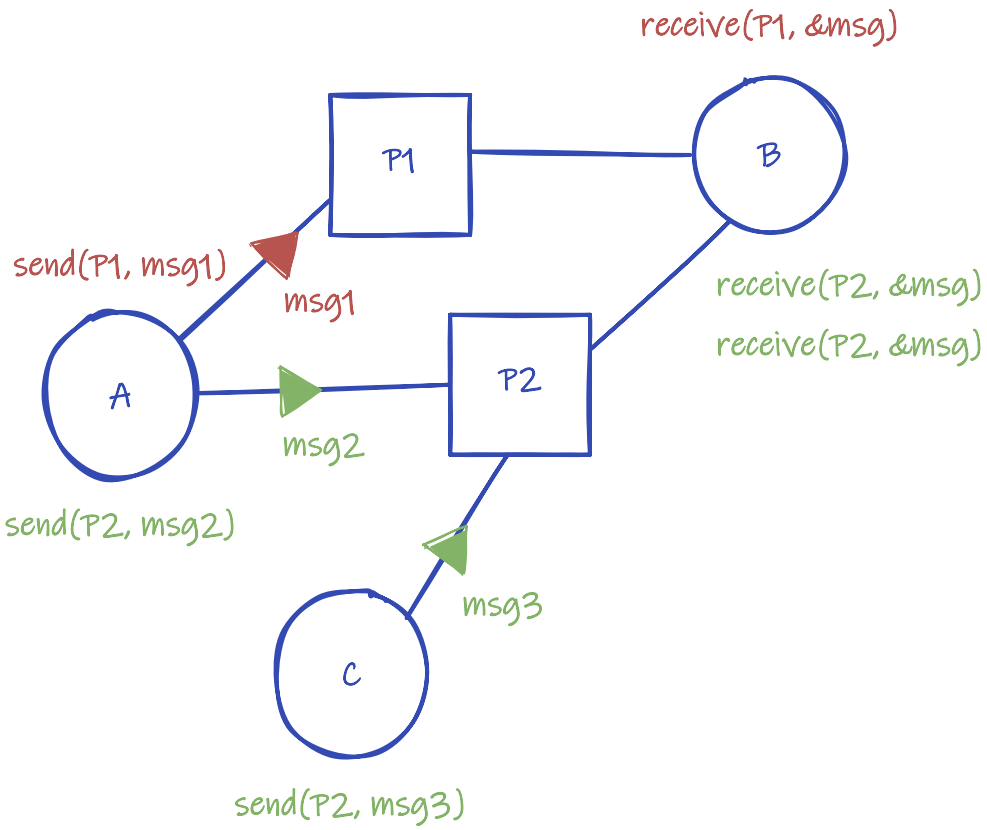


Figura 33. Comunicación indirecta.

Colas de mensajes en sistemas POSIX

El estándar POSIX también define un sistema de colas de mensajes, pero es bastante diferente a la solución en Windows API (véase «mq_overview(7) — Linux Manual»).

Para usarlo, lo primero es abrir o crear —si aún no existe— la cola de mensajes llamando a mq_open():

```
mqd_t mqueue = mq_open(
    "/foo-mqueue",      ①
    O_CREAT | O_RDWR,   ②
```

```
    0644,               ③
    NULL
);
```

① Nombre que identifica la cola de mensajes. Como con los archivos, para que varios procesos puedan acceder a la misma cola, deben indicar el mismo nombre.

② Valores que indican diferentes opciones a la hora de abrir la cola de mensajes. Por ejemplo, usando `O_RDWR` se indica abrir para enviar o recibir y con `O_CREAT` se indica que la cola debe crearse si no existe previamente.

③ Indica los permisos de la cola de mensajes al crearla nueva, de forma similar a los permisos que se aplican a los archivos.

El valor devuelto por mq_open() es el descriptor de la cola de mensajes. Como otros descriptores, se hereda de padres a hijos al usar fork().

Este descriptor se utiliza como primer argumento en funciones posteriores para indicar sobre qué cola queremos realizar la correspondiente operación. Por ejemplo, para enviar un mensaje se utiliza mq_send() así:

```
int mq_send(
    mqueue,                 ①
    (const char *)&message, ②
    sizeof(message),        ③
    0                       ④
);
```

① Descriptor de la cola a la que enviar el mensaje.

② Puntero a la dirección de memoria donde está el mensaje.

③ Tamaño del mensaje en bytes.

④ Prioridad del mensaje. Los mensajes con mayor prioridad se entregarán antes.

Mientras que para recibir un mensaje se utiliza mq_receive()

```
unsigned int msg_prio;

int mq_receive(
    mqueue,                 ①
    (const char *)&message, ②
    sizeof(message),        ③
    &msg_prio               ④
);
```

① Descriptor de la cola de la que recibir el mensaje.

② Puntero a la dirección de memoria donde guardar el mensaje al recibirlo.

③ Tamaño máximo de espacio reservado en `message` para guardar el mensaje.

④ Puntero a variable entera donde devolver la prioridad del mensaje recibido.

Como los mensajes no se dirigen directamente a los procesos, sino a estas entidades llamadas colas de mensajes, se trata de un caso de **comunicación indirecta**. Además, el **tamaño de los mensajes es variable**, aunque limitado por defecto a 8 KiB si no se configura de otra manera.

Si varios procesos intentan recibir de una misma cola de mensajes al mismo tiempo, queda en manos del sistema operativo decidir cuál recibirá el siguiente mensaje que llegue. Por lo general es el primero en ser escogido por el planificador de la CPU para seguir ejecutándose.

Recepción concurrente

Este tipo de comunicación da lugar a algunas situaciones que deben ser resueltas durante el diseño. Por ejemplo, ¿qué ocurre, por ejemplo, si los procesos A, B y C comparten el puerto P1; A manda un mensaje y B y C invocan `receive()` en el puerto P al mismo tiempo?.

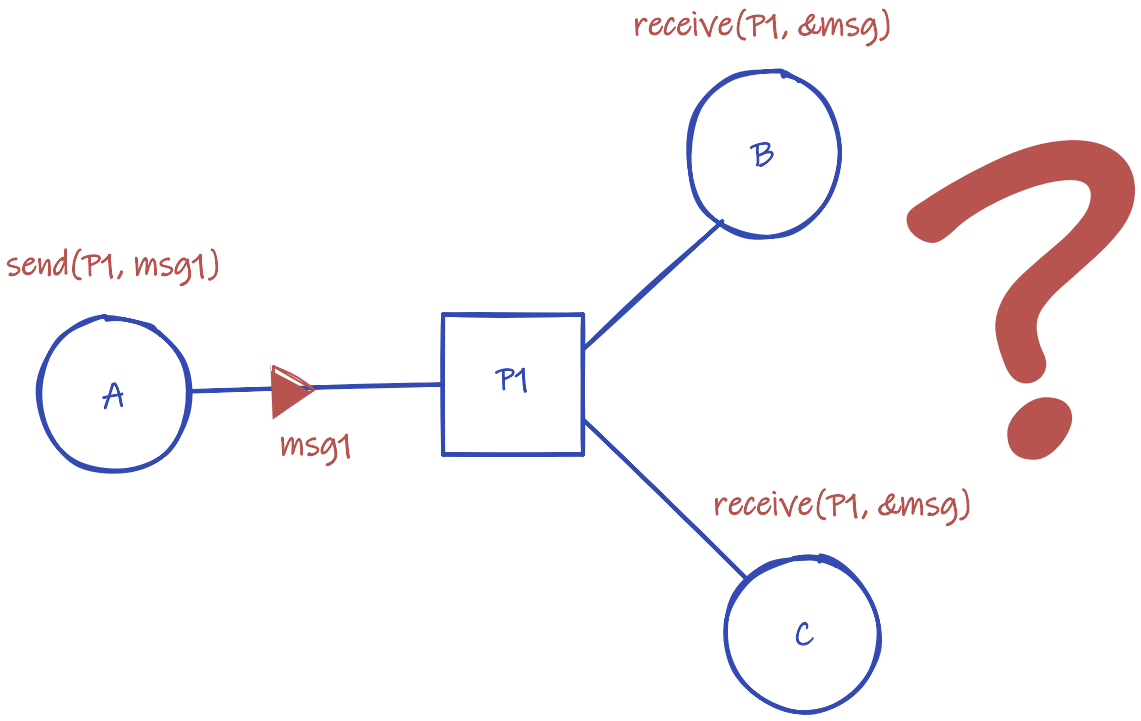


Figura 34. Problema de la recepción concurrente.

La respuesta correcta dependerá de la elección de los diseñadores del sistema:

- No permitir que cada enlace de comunicación —y por tanto cada puerto— esté asociado a más de dos procesos.
- No permitir que más de un proceso puedan ejecutar `receive()` al mismo tiempo. Por ejemplo, en algunos sistemas solo el proceso que crea el puerto tiene permisos para recibir de él. Los sistemas que optan por esta solución suelen disponer de algún mecanismo para que un proceso pueda transferir el permiso de recibir a otros procesos.
- Permitir que el sistema operativo escoja arbitrariamente quién recibe el mensaje si dos o más procesos ejecutan `receive()` al mismo tiempo. La elección puede ser aleatoria, mediante algún algoritmo, por ejemplo, por turnos o el siguiente proceso en obtener la CPU, a criterio del

planificador de la CPU.

10.3. Buffering

Los mensajes intercambiados por enlace de comunicación se almacenan en una cola temporal, a la espera de ser enviados o, tras recibirlos, a la espera de que los reclame el proceso.

Básicamente hay tres formas de implementar dicha cola:

- Con **capacidad cero** o **sin buffering** la cola tiene una capacidad máxima de 0 mensajes, por lo que no puede haber ningún mensaje esperando en el enlace. En este caso el proceso transmisor se bloquea en espera hasta que el receptor recibe el mensaje.
- Con **buffering automático**, donde existe dos opciones:
  - Con **capacidad limitada** la cola tiene una capacidad máxima de *N* mensajes, por lo que si la cola se llena el proceso transmisor se bloquea a la espera de que haya espacio en la cola. Obviamente, mientras la cola no se llene en transmisor puede seguir metiendo mensajes sin bloquearse.
  - Con **capacidad ilimitada** la cola es de longitud potencialmente infinita, lo que permite que el transmisor nunca espere.

Las colas de longitud infinita son imposibles, puesto que los recursos son limitados. En realidad este término hace referencia a colas de longitud variable cuyo máximo viene determinado por la memoria principal disponible, que suele ser lo suficientemente grande como para que podamos considerar que las colas son infinitas.

Buffering en las colas de mensajes POSIX

Las colas de mensajes en sistemas POSIX tienen capacidad limitada. Los límites se configuran al crear la cola, a través del último argumento de mq_open():

```
struct mq_attr attr = {  ①
    .mq_maxmsg = 5,      ②
    .mq_msgsize = 2049   ③
};

mqd_t mqueue = mq_open(
    "/foo-queue",
    O_CREAT | O_RDWR,
    0644,
    &attr ①
);
```

① Estructura con propiedades para la cola cuando esta se crea nueva.

② Una de las propiedades es el número máximo de mensajes en la cola al mismo tiempo.

③ Otra de las propiedades es el tamaño máximo de cada mensaje.

Estos límites tienen unos valores por defecto por si en el lugar de `attr` en mq_open() se indica `NULL`. El estándar POSIX indica que esos valores por defecto dependen de cada sistema operativo, por lo que es necesario ir a la documentación para desarrolladores de cada sistema para conocer los detalles en cada caso concreto.

Por ejemplo, en Linux los valores por defecto son 10 mensajes y 8 KiB por mensaje, siendo estos, además, los valores máximos que admiten esas propiedades. Estos valores máximos y por defecto se pueden cambiar de forma global para todo el sistema, por si tuviéramos interés en valores más altos.

10.4. Operaciones síncronas y asíncronas

La comunicación entre dos procesos tiene lugar por medio de las llamadas `send()` y `receive()`; de tal forma que generalmente la primera se bloquea cuando la cola de transmisión se llena —en función del tipo de *buffering*— mientras que la segunda lo hace cuando la cola de recepción está vacía.

Sin embargo, en lugar de bloquearse, puede que aun proceso le interese ejecutar otras tareas en la CPU. A fin de cuentas las comunicaciones son bastante lentas, por lo que en caso de bloquearse podría estar dejando de aprovechar bastante tiempo de CPU. Incluso puede darse el caso que tengan conexión con otros procesos y que quiera aprovechar para intentar comunicarse con alguno de ellos.

Por eso existen diferentes opciones de diseño a la hora de implementar las llamadas anteriores en función de si se pueden bloquear o no. Concretamente, el paso de mensajes puede ser **síncrono** —con bloqueo— o **asíncrono** —sin bloqueo—.

- **Cuando el envío es asíncrono**, el proceso transmisor nunca se bloquea. Si se llama a `send()` cuando la cola de mensajes esté llena, lo más común es que retorne con un código de retorno que indique que el proceso debe volver a intentar el envío más tarde.
- **Cuando el envío es síncrono**, el proceso transmisor se bloquea cuando no queda espacio en la cola de mensajes y hasta que pueda depositar el mensaje en la misma.
- **Cuando la recepción es asíncrona**, el receptor nunca se bloquea. En caso de que la cola de mensajes esté vacía, el sistema operativo puede indicar al proceso que lo intente más tarde a través de un código de retorno o devolviendo un mensaje vacío.
- **Cuando la recepción es con bloqueo**, el receptor se bloquea cuando no hay mensajes en la cola y hasta que llegue alguno.

Algunos sistemas de paso de mensajes son claramente síncronos o asíncronos. Mientras que otros permiten activar un modo u otro según las necesidades de la aplicación. E incluso los hay que soportan que la transmisión y recepción sean síncronas o asíncronas de manera totalmente independiente.

Comunicaciones asíncronas con colas de mensajes POSIX

Por defecto las colas de mensajes son síncronas, tanto en envío como en recepción. Es decir, si al enviar un mensaje la cola está llena, el proceso transmisor quedará bloqueado en estado **esperando** hasta que haya un hueco libre para depositar el nuevo mensaje. Si al recibir un mensaje la cola está vacía, el receptor quedará bloqueado hasta que otro proceso deposite un mensaje.

Sin embargo, si en el argumento `oflag` de mq_open() un proceso indica la opción `O_NONBLOCK` estas operaciones para ese proceso en esa cola serán asíncronas:

```
mqd_t mqueue = mq_open(
    "foo-mqueue",
    O_RDONLY | O_NONBLOCK, ①
    0644,
    &attr
);
```

① Abrir la cola de mensajes para solo lectura —con `O_RDONLY`— y para comunicaciones asíncronas —con `O_NONBLOCK`—.

Eso quiere decir que las funciones mq_send() y mq_receive(), en lugar de bloquear el proceso en estado de esperando, devolverán -1 y el valor de errno será `EAGAIN`. Así el proceso puede aprovechar el tiempo de CPU del que dispone para realizar otras tareas mientras tanto y volver a intentarlo más tarde.

```
int return_code = mq_receive(mqueue, &message, sizeof(message), &msg_prio);
if ( return_code > 0 ) ①
{
    // Aquí va código para usar el mensaje recibido...
}
else if( return_code < 0 && errno != EAGAIN) ②
{
    // Aquí va código para manejar el error de mq_receive()...
}
```

① Si todo va bien, mq_receive() devuelve el tamaño en bytes del mensaje.

② Si devuelve -1, es que ha ocurrido un error. Pero solo será un error real si el código de error en errno no es `EAGAIN`. Si es `EAGAIN`, se pueden ejecutar otras partes del programa y volver a intentar la recepción más adelante.

Si un proceso debe comunicarse mediante varias colas de mensajes, la comunicación asíncrona también sirve para intentar recibir y enviar de varias colas sin bloquearse en ninguna. Para este caso algunos sistemas ofrecen una alternativa más sencilla y eficiente, que veremos en el Apartado 10.5.1.

10.5. Ejemplos de sistemas de paso de mensajes

10.5.1. Colas de mensajes POSIX

Como hemos comentado a lo largo de capítulo, las colas de mensajes POSIX son un caso de **comunicación indirecta**, con **tamaño de mensaje variable**, *buffering* con **capacidad limitad** y que soporta operaciones **asíncronas**.

Las colas de mensajes son útiles para enviar mensajes de pequeño tamaño entre procesos que se ejecutan en el mismo sistema. Además tienen la posibilidad de asociar a cada mensaje una prioridad, de tal forma que se reciban primero los mensajes de prioridad más alta. Su uso es relativamente común en sistemas de tiempo real, aunque lo más frecuente en los sistemas de propósito general es usar *sockets*.

En message_queue.hpp se puede ver un ejemplo de una clase desarrollada en C++ para utilizar colas de mensajes POSIX. En los distintos métodos se puede ver cómo se utilizan las funciones de la librería del sistema para crear la cola y enviar y recibir mensajes.

En mqueue-server.cpp y mqueue-client.cpp se puede ver un ejemplo de cómo se utiliza la clase en message_queue.hpp. El primero es un programa que muestra la hora del sistema periódicamente. El segundo se puede comunicar con el primero a través de una cola de mensajes para controlarlo. En ejemplo es muy sencillo, así que, por el momento, lo único que puede hacer mqueue-client.cpp es pedirle a mqueue-server.cpp que termine. Aunque no costaría nada añadir otras órdenes, como pedir que cambie la hora del sistema o la periodicidad con la que la muestra.

En Linux los descriptores de colas de mensajes son descriptores de archivo —como también lo son los descriptores de *sockets*, tuberías y los de archivos abiertos con open(), entre otros—. Esta particularidad implica que en Linux, mediante las funciones select(), poll() o epoll(), se pueden monitorizar al mismo tiempo varios descriptores de colas de mensajes, para así saber cuándo se puede enviar o recibir por ellas sin que el proceso se bloquee.

Este comportamiento es específico de Linux. No está contemplado en el estándar POSIX, por lo que otros sistemas POSIX no tienen por qué soportarlo. Así que no es portable.

A continuación se puede ver un ejemplo específico con poll(), aunque las tres funciones se utilizan empleando un patrón similar:

```
mqd_t mqueue1 = mq_open( "/foo-queue", /* ... */ ); ①
mqd_t mqueue2 = mq_open( "/bar-queue", /* ... */ );

struct pollfd fds[] =
{
    {
        .fd = mqueue1, ②
        .events = POLLIN, ③
        .revents = 0
    }, {
```

```
        .fd = mqueue2, ②
        .events = POLLIN | POLLOUT, ③
        .revents = 0
    }
};

while ( !quit_app )
{
    int return_code = poll( fds, 2, -1 ); ④
    if (return_code > 0) ⑤
    {
        if (fds[0].revents & POLLIN) ⑦
        {
            mq_receive( fds[0].fd, /* ... */ );

            // Aquí va código para usar el mensaje recibido...
        }

        if (fds[1].revents & POLLIN) ⑦
        {
            mq_receive( fds[1].fd, /* ... */ );

            // Aquí va código para usar el mensaje recibido...
        }

        if (fds[1].revents & POLLOUT) ⑦
        {
            // Aquí va código para preparar el mensaje a enviar...

            mq_send( fds[1].fd, /* ... */ );
        }
    }
    else if (return_code < 0) ⑥
    {
        // Error en poll().
        // Aquí va código para leer errno y manejar el error...

        quit_app = true;
    }
}
```

① Abrimos o creamos las colas que vamos a utilizar.

② Creamos un *array* de la estructura `pollfd` con un elemento por cola que vamos a monitorizar con poll(). En cada estructura, en el campo `fd`, se indica el descriptor de cada una de las colas de mensajes.

③ Para cada cola hay que utilizar el campo `events` para indicar qué queremos que monitorice poll(). `events` es una máscara de bit donde a cada evento monitorizable le corresponde un bit. Si queremos monitorizar un evento, debemos poner su bit a 1.

Para eso nos podemos ayudar de macros como `POLLIN` y `POLLOUT`. Por ejemplo, para `mqueue1` se

quiere monitorizar cuándo hay mensajes para recibir, por lo que se activa `POLLIN`. Mientras que para `mqueue2` se quiere saber tanto cuándo hay mensajes para recibir como cuándo hay un hueco en la cola para enviar sin bloqueos, por lo que se activan `POLLIN` y `POLLOUT`.

④ Iterativamente se llama a poll() —mientras no queramos que termine la aplicación— que pondrá el proceso en estado **esperando** hasta que ocurra alguno de los eventos que nos interesan. poll() necesita `fds` —el *array* de la estructura `pollfd` que hemos inicializado previamente— el número de elementos en el *array* y el tiempo máximo que debe mantener bloqueado el proceso esperando a que ocurra alguno de los eventos. Con un número negativo en este último campo, se indica que queremos que espere indefinidamente.

⑤ Si poll() tiene éxito, devuelve un número positivo que indica en cuántos descriptores se ha detectado un evento.

⑥ Si poll() devuelve un valor negativo, es que ha ocurrido algún error. El motivo del error se puede conocer comprobando el valor de la variable global errno.

⑦ El campo `revents` es una máscara de bits similar a `events`, pero al retornar de poll() indica qué eventos se han detectado, para cada cola de mensajes en `fds`.
Por ejemplo, en ambas colas se comprueba si `POLLIN` está activo. En caso afirmativo, sabemos que podemos leer un mensaje sin que mq_receive() se bloquee. Igualmente, sabemos si `mqueue2` tiene hueco para enviar un mensaje comprobando si `POLLOUT` está activo. En caso afirmativo, podemos enviar un mensaje con mq_send() sabiendo que no se bloqueará.

10.5.2. Señales en sistemas operativos POSIX

En los sistemas POSIX, una forma más sencilla de comunicar dos procesos del mismo sistema es mediante el envío de una **señal** de uno al otro.

Los procesos pueden mandar señales utilizando la llamada al sistema kill(), que solo requiere el identificador del proceso de destino y el número que identifica la señal.

```
kill(pid, SIGTERM);
```

Como se usa el identificador del proceso, estamos hablando de un mecanismo de **comunicación directa**.

El **tamaño y formato del mensaje es fijo**. Las señales solo pueden portar la información de que ha ocurrido un evento, indicado qué evento es a través del número que identifica la señal.

Cada señal tiene un efecto particular por defecto —que por lo general es matar al proceso— en el proceso que las recibe. Sin embargo, cada proceso puede declarar un **manejador de señal**. Una función del programa que será invocada por el sistema operativo para tratar una señal determinada, interrumpiendo lo que esté haciendo el proceso en ese momento. En ese sentido las señales en POSIX puede interpretarse como una forma de interrupción por software.

Como la ejecución del programa se interrumpe para saltar al **manejador de señal** y luego continuar, hay que tener cuidado con el código de los **manejadores de señal**. Puede causar problemas que, por ejemplo, modifiquen el valor de una variable en un punto donde el resto del código no espera que puedan ocurrir

> cambios. Por eso es recomendable que el código de los **manejadores de señal** sea **reentrante** (véase el Apartado 13.7.1) y que solo invoquen otras funciones **reentrantes** o funciones de la librería del sistema marcadas como seguras en señales.

El **manejador de señal** se puede configurar usando la llamada al sistema signal():

```
signal(
    SIGTERM, ①
    &mi_manejador_de_sigterm ②
);
```

① Identificador de la señal a recibir.

② Puntero al manejador de señal. Es decir, la del programa que será llamada por el sistema operativo cuando llegue la señal `SIGTERM`.

El problema de signal() es que el estándar POSIX permite diferencias que hacen que se pueda comportar de forma distinta en diferentes sistemas operativos. Por ejemplo, qué ocurre al retornar del **manejador de señal** si cuando llegó la señal el proceso estaba ocupado en una llamada al sistema. Como el estándar no especifica nada al respecto, los sistemas BSD optaron porque las llamadas al sistema continuaran como si nada, mientras que en otros sistemas POSIX las llamadas al sistema son interrumpidas como si hubiera ocurrido un error.

Para resolverlo, el estándar recomienda usar sigaction() en su lugar, ya que está descrita de forma más precisa —evitando este tipo de divergencias— y permite que el programador escoja de entre varias opciones el comportamiento que más le convenga:

```
struct sigaction act = {
    .sa_handler = &mi_manejador_de_sigterm, ④
    .sa_sigaction = NULL,   ⑤
    .sa_mask = 0,           ⑥
    .sa_flags = SA_RESTART, ⑦
}

sigaction(
    SIGTERM, ①
    &act,    ②
    NULL     ③
);
```

① Identificador de la señal a recibir.

② Puntero a una estructura de tipo `sigaction` que describe los detalles de como tratar la señal cuando llega al proceso.

③ Puntero a una estructura de tipo `sigaction` donde sigaction() guarda la configuración anterior sobre como tratar la señal indicada.

④ Puntero al manejador de señal para la señal indicada.

⑤ Puntero a un manejador de señal alternativo al de `sa_handler`. Este manejador recibe más información sobre la señal cuando es llamado. Para activar es necesario indicar `SA_SIGINFO` en el campo `sa_flags`.

⑥ Máscara de bits de señales a bloquear durante el manejo de la señal. Cada bit de la máscara identifica a una señal. Deben ponerse a 1 aquellas señales que queremos que estén bloqueadas —es decir, que no se puedan recibir— mientras se ejecuta el manejador de señal porque ha llegado una. Es especialmente útil si se va a usar el mismo manejador para varias señales.

⑦ Opciones de configuración. En el ejemplo se usa `SA_RESTART`, que indica que si la señal llega durante una llamada al sistema, la llamada debe continuar una vez se haya salido del manejador de señal. Como comentamos anteriormente, este es el comportamiento de signal() en los sistemas BSD. El comportamiento por defecto, sin esta opción, es que la llamada al sistema interrumpida falle con el error `EINTR` en errno.

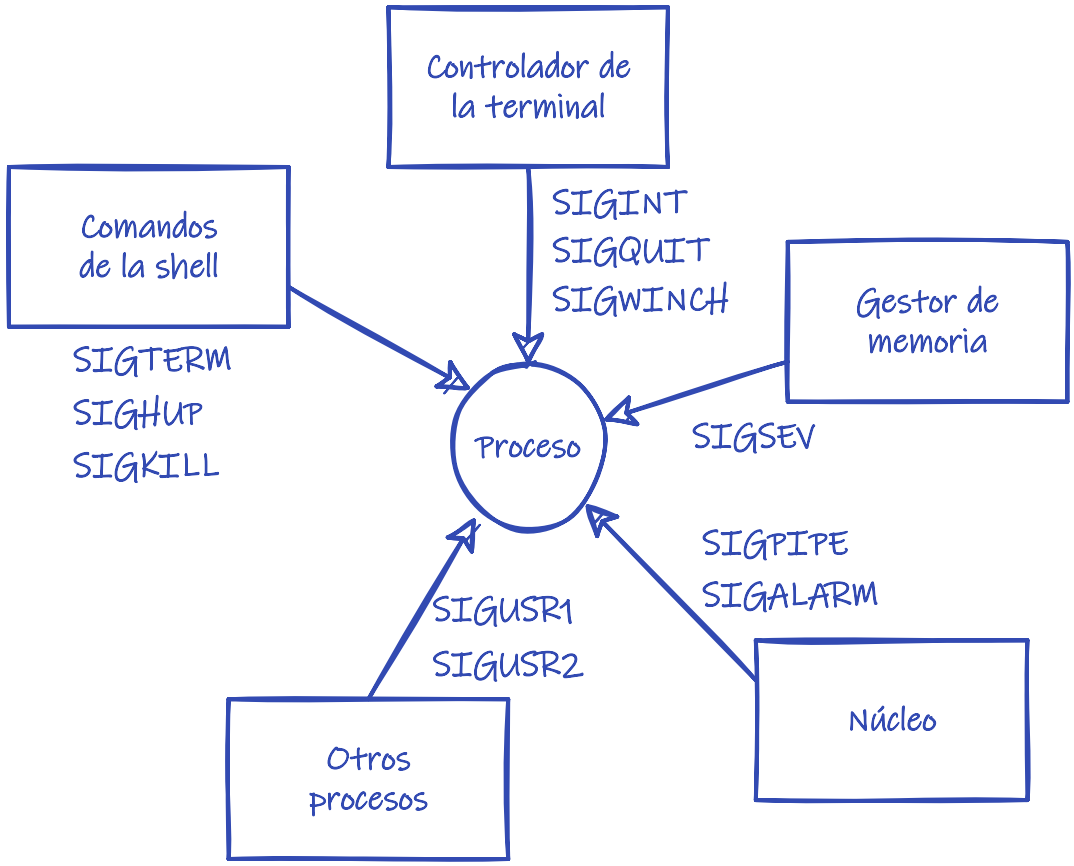


Figura 35. Orígenes más comunes de las señales.

Las señales fueron diseñadas originalmente como un mecanismo para que el sistema operativo notificase a los programas ciertos errores y sucesos críticos. Por ejemplo:

- La señal `HUP` o `SIGHUP` es enviada a cada proceso iniciado desde una sesión de terminal cuando dicha sesión termina —o cuando se usa la combinación de teclas `CTRL` + `D`, que tiene el mismo efecto—.

  En el caso de los servicios del sistema —que, como no son interactivos, no están conectados a ninguna terminal— esta señal suele usarse para indicarles que deben reiniciarse, volviendo a leer sus archivos de configuración, o para que guarden su estado interno en algún sitio conocido del almacenamiento.

- La señal `INT` o `SIGINT` es enviada al proceso que está enganchado a la consola cuando el usuario pulsa el carácter de interrupción —frecuentemente la combinación de teclas `CTRL` + `C`—.
- La señal `TERM` o `SIGTERM` es enviada al proceso cuando debe terminar. Por ejemplo, el sistema operativo envía esta señal a todos los procesos cuando se está apagando el sistema.
- La señal `SEGV` o `SIGSEGV` es enviada a un proceso cuando intenta acceder a una zona de memoria a la que no tiene permiso. Si no se maneja esta señal, el programa termina con el conocido mensaje de **violación de segmento**.

Obviamente hay muchas más señales. Entre todas, el estándar POSIX incluye dos señales —`USR1` y `USR2`— especialmente indicadas para usarlas con el significado que nosotros queramos.

Se puede consultar una lista de las señales del estándar POSIX en «Señales (informática) — Wikipedia». Mientras que la lista completa de señales soportadas en Linux se puede consultar en «signal(7) — Linux Manual».

El ejemplo en mqueue-server.cpp y en otros ejemplos de este capítulo, utiliza señales para manejar `SIGINT`, `SIGTERM` y para mostrar la hora periódicamente. El código dedicado a eso está en timeserver.c y se comparte entre todos los ejemplos. En todos los casos se evita usar `SA_RESTART` porque interesa que el proceso interrumpa lo que esté haciendo cuando llegue una señal para terminar.

En signals.c hay un programa de ejemplo que muestra cómo manejar las señales del sistema y que sirve para ver cómo funcionan. Solo hay que ejecutarlo y luego enviarle señales con el comando `kill` desde otra terminal. En este caso si se usa `SA_RESTART`, porque el programa debe estar esperando la pulsación de una tecla con `getc()` y no nos interesa que deje de hacerlo cuando llegue una señal.

10.5.3. Tuberías

Las **tuberías** son un mecanismo de paso de mensajes de **comunicación indirecta**, **orientada a flujos**, **capacidad limitada** y, generalmente, **comunicación síncrona** —aunque en algunos sistemas operativos también puede soportar asíncrona—.

Conceptualmente, cada tubería tiene dos extremos en los que opera utilizando la misma interfaz que generalmente empleamos para manipular archivos. Es decir, usando funciones como `read()`, `write()` y `close()`, entre otras. Un extremo permite a los procesos en ese extremo escribir en la tubería, mientras el otro extremo permite a los procesos leer de la tubería los datos escritos desde el otro extremo.

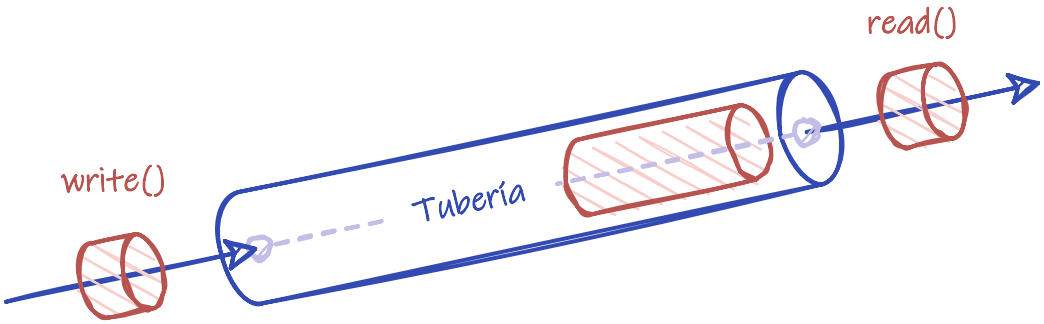


Figura 36. Esquema del concepto de tubería.

El que cada extremo imite ser un archivo, facilita que se puedan usar en muchas de las llamadas al sistema que aceptan un archivo como argumento. Los procesos pueden leer o escribir en un archivo sin saber realmente si están accediendo a una archivo real o se están comunicando con otro proceso mediante una tubería.

Existen dos tipos de tuberías:

- Las **tuberías anónimas** que solo existen en el espacio de direcciones del proceso que las crea, de tal forma que debe heredarse de padres a hijos para que otros procesos puedan tener acceso.
- Las **tuberías con nombre** son públicas al resto del sistema, por lo que teóricamente cualquier proceso con permisos puede abrir una para comunicarse con otros procesos. Por eso se suele utilizar en aplicaciones cliente-servidor, donde un proceso servidor ofrece algún servicio a otros procesos cliente a través de la tubería.

En los sistemas POSIX las **tuberías con nombre** se denominan *FIFO* y tienen presencia en el sistema de archivos como archivos especiales.

Tabla 3. Funciones de la API para manipular tuberías.

	POSIX API	**Windows API**
Crear tubería anónima	pipe()	CreatePipe()
Crear tubería con nombre	mkfifo()	CreateNamedPipe()
Abrir tubería con nombre	open()	CreateFile()
Leer	read()	ReadFile()
Escribir	write()	WriteFile()
Cerrar	close()	CloseHandle()
Destruir tubería con nombre	unlink()	[Automático]

Con fork() es muy sencillo lanzar otros procesos para que ejecuten tareas en paralelo. El proceso hijo tiene acceso a los datos del padre por la forma en la que funciona fork() y gracias a las tuberías anónimas puede comunicar los resultados al padre. En fork-pipe.cpp se puede observar un ejemplo de esto.

Además, el hecho de que cada extremo se comporte como un archivo —uno en modo solo lectura y el otro en modo solo escritura— hace posible redirigir la E/S estándar del proceso hijo. Es decir,

conectar la entrada, la salida estándar o la salida de error a una tubería, desde la que leer lo que el proceso intenta imprimir por la pantalla de la terminal o proporcionarle lo que debe leer, como si fuera desde el teclado. En fork-redir.c se puede ver un ejemplo de cómo ejecutar el comando `ls` y redirigir su salida al proceso padre para contar el número de líneas en lo que el comando quería mostrar por pantalla.

Por otro lado, las tuberías con nombre permiten que un proceso se comunique con cualquier otro, solo con conocer la ruta de la tubería. En fifo-server.c tenemos un ejemplo de un programa que muestra la hora del sistema de forma periódica, mientras espera órdenes de una tubería que sirve de canal de control remoto. Los programas en fifo-client.c y fifo-client.cpp pueden conectarse a esa tubería y mandar el comando que hace terminar fifo-server.c.

10.5.4. Sockets

Mientras que las tuberías son conceptualmente un enlace de comunicación unidireccional que tiene dos extremos, un ***socket*** representa un solo extremo en un enlace de comunicación bidireccional. Para que una pareja de procesos se pueda comunicar son necesarios dos *sockets* —uno en cada proceso— de manera que cada uno de ellos es el medio por el que el proceso accede al enlace de comunicación.

La API de *sockets* fue creada por la Universidad de Berkeley para abstraer el acceso a la familia de protocolos de Internet en el UNIX desarrollado por esa misma universidad. Sin embargo, rápidamente se convirtió en el estándar de facto para la comunicación en red, por lo que todos los sistemas operativos modernos —incluidos los sistemas POSIX y Microsoft Windows— tienen una implementación de la misma.

Pese a sus orígenes en Internet, los *sockets* se diseñaron para ser independientes de la tecnología de red subyacente con la que se implementa el enlace de comunicación. En Linux, por ejemplo, se puede utilizar como interfaz de programación para utilizar dos decenas de familias de protocolos y tecnologías diferentes.

Para crear un *socket* se utiliza la llamada al sistema socket():

```
int sockfd = socket( ①
    AF_UNIX,         ②
    SOCK_DGRAM,      ③
    0
)
```

① En sistemas POSIX la función devuelve un `int` con el descriptor del socket mientras que en Microsoft Windows devuelve un `HANDLE`.

② En el primer argumento se especifica la familia de protocolos. `AF_UNIX` son un tipo de *socket* que solo sirve para comunicar procesos en el mismo sistema, denominado **socket de dominio UNIX**. Otras familias muy comunes son `AF_INET`, que corresponde a la familia de protocolos TCP/IP y `AF_INET6` para los protocolos IPv6.

③ En el segundo argumento se especifica el tipo del *socket*. Cada tipo suele corresponde con un protocolo concreto de la familia elegida. Por ejemplo, los *sockets* `SOCK_DGRAM` son «no orientados a conexión», no fiables y de longitud máxima fija, así que en la familia `AF_INET` estos *sockets*

utilizan UDP. Mientras que los *sockets* `SOCK_STREAM` son orientados a conexión, fiables, bidireccionales y orientados a flujo, por lo que en la familia `AF_INET` utilizan TCP.

Los **sockets de dominio UNIX** solo entregan los mensajes dentro del mismo sistema, así que siempre son confiables, independientemente de si son `SOCK_DGRAM` o `SOCK_STREAM`.

Un *socket* recién creado no tiene un nombre que otro proceso pueda usar para identificarlo y comunicarse con él. Si se va a usar este *socket* solo para enviar mensajes a otro *socket*, esto es no es un problema. Pero para recibir mensajes, el remitente tiene que poder identificar al destinatario.

Para asignar un nombre o dirección a un *socket* se utiliza bind(). La dificultad es que cada familia de protocolos tiene un formato de direcciones diferente, así que hay que tener cuidado de usar el adecuado:

```
struct sockaddr_un addr = {
    .sun_family = AF_UNIX,         ⑤
    .sun_path = "/tmp/foo-socket" ⑥
};

int result_code = bind( ①
    sockfd, ②
    (struct sockaddr*) &addr, ③
    sizeof(addr) ④
)
```

① Como en el resto de llamadas al sistema, en caso de error se devuelve un número negativo y errno contendrá el código del error.

② El descriptor del *socket* al que se le quiere cambiar la dirección.

③ La nueva dirección del *socket* especificada como una estructura adecuada para la familia del *socket*. En *socket* de tipo `AF_UNIX` la estructura debe ser de tipo `sockaddr_un` mientras que en los de tipo `AF_INET` es del tipo `sockaddr_in`.

④ El tamaño en bytes de la estructura con la nueva dirección.

⑤ En la estructura con la dirección, el primer campo siempre es para indicar la familia.

⑥ En los *sockets* de dominio UNIX la dirección es una ruta en el sistema de archivos. Para otras familias, las direcciones se indican de otra manera, por lo que es necesario consultar la documentación.

La API de *sockets* incluye muchas otras funciones:

- listen(), para poner *sockets* tipo `SOCK_STREAM` a la espera de conexiones.
- connect(), para conectar un *socket* tipo `SOCK_STREAM` con otro que esté a la espera de conexiones.
- accept() para que un *socket* tipo `SOCK_STREAM` a la espera de conexiones acepte una solicitud de conexión.
- shutdown() para cerrar uno de los sentidos de una conexión.

- close() para destruir un *socket*.
- send(), sendto() y sendmsg() para enviar mensajes. send() solo se puede utilizar con *sockets* conectados. Mientras que sendto() permiten indicar la dirección del *socket* de destino, por lo que es útil en *sockets* no orientados a conexión `SOCK_DGRAM`.
- recv(), recvfrom() y recvmsg() para recibir mensajes. recvfrom() permite obtener la dirección del *socket* del que llegó el mensaje. Por eso es útil en *sockets* no orientados a conexión `SOCK_DGRAM`.

```
struct sockaddr_un addr;
socklen_t addrlen;

int result_code = recvfrom( ①
    sockfd,          ②
    &message,        ③
    sizeof(message), ④
    0,               ⑤
    (struct sockaddr*) &addr, ⑥
    &addrlen         ⑦
)
```

① En caso de éxito devuelve el número de bytes del mensaje recibido. En caso de error, un -1 y errno contiene el código del error.

② El descriptor del *socket* al que se le quiere cambiar la dirección.

③ Puntero a la dirección de memoria donde está el mensaje.

④ Tamaño del mensaje en bytes.

⑤ Opciones adicionales de configuración.

⑥ Estructura de dirección vacía donde se copiará la dirección del *socket* que remite el mensaje.

⑦ Puntero donde la llamada al sistema copiará el tamaño de la estructura copiada en `addr`.

Las operaciones con *sockets* son síncronas por defecto. Sin embargo, es posible configurarlos en modo asíncrono, para que así cualquiera de estas funciones falle, retornando -1 y código de error `EAGAIN` o `EWOULDBLOCK`, antes de poner el proceso en estado **esperando**.

También se pueden utilizar las funciones select() y poll() para monitorizar varios *sockets* al mismo tiempo, de forma similar a como se hace para colas de mensajes POSIX (véase el Apartado 10.5.1).

En socket-server.cpp y socket-client.cpp se puede observar un ejemplo similar al que usamos con las tuberías y las colas de mensajes, pero empleando *sockets* de dominio UNIX. Ambos programas utilizan la cabecera socket.hpp que incluye un ejemplo de clase en C++ para comunicaciones mediante *sockets*. En los distintos métodos se puede ver cómo se utilizan las funciones de la librería del sistema para crear *sockets*, asignarles dirección y usarlos para enviar y recibir mensajes.

En resumen, los ***sockets*** son un mecanismo de paso de mensajes de **comunicación indirecta**, que admite tanto comunicación **orientada a flujos** como **mensajes de tamaño variable**, *buffering* de **capacidad limitada** y tanto **comunicación síncrona** como **asíncrona**, aunque el comportamiento real final de la interfaz depende de la tecnología de red utilizada.

Chapter 11. Memoria compartida

Tiempo de lectura: 5 minutos

La **memoria compartida** es una estrategia para comunicar procesos donde uno de ellos gana acceso a regiones de la memoria del otro; algo que por lo general el sistema operativo siempre intenta evitar. Por eso, para que pueda haber memoria compartida es necesario que los dos procesos estén de acuerdo en eliminar dicha restricción.

Dos procesos que comparten una región de la memoria pueden intercambiar información simplemente leyendo y escribiendo datos en la misma. Sin embargo debemos tener en cuenta que:

- La estructura de los datos y su localización dentro de la región compartida la determinan los procesos en comunicación y no el sistema operativo, a diferencia de lo que ocurre en los sistemas de paso de mensajes.
- Los procesos son responsables de sincronizarse para no escribir y leer en el mismo sitio de la memoria al mismo tiempo, pues esto puede generar inconsistencias (véase el Capítulo 13) .

Las principales ventajas de la memoria compartida frente a otros mecanismos de comunicación son:

- **Eficiencia**. Puesto que la comunicación tiene lugar a la velocidad de la memoria principal, se trata de un mecanismo tremendamente rápido.
- **Conveniencia**. Puesto que el mecanismo de comunicación solo requiere leer y escribir de la memoria, se trata de un sistema muy sencillo y fácil de utilizar.

Como ocurre con las tuberías (véase el Apartado 10.5.3) la memoria compartida puede ser anónima o con nombre.

11.1. Memoria compartida anónima

La **memoria compartida anónima** solo existe para el proceso que la crea y para sus procesos hijos, que heredan el acceso. Es por tanto, una forma eficiente de comunicar procesos padres e hijos.

En los sistemas POSIX, las funciones y operadores de reserva de memoria como malloc() y new, utilizan internamente la llamada al sistema mmap(). Esta función se puede llamar de la siguiente manera para reservar `length` bytes de memoria.

```
void* p = mmap( ①
    NULL,
    length,     ②
    PROT_READ | PROT_WRITE,      ③
    MAP_ANONYMOUS | MAP_PRIVATE, ④
    -1,
    0
);
```

① Si todo va bien, mmap() devuelve un puntero al primer byte de la memoria reservada.

② Cantidad de memoria a reservar en bytes.

③ Permisos de acceso para la memoria reservada. En este caso, se solicita permitir la lectura y la escritura de la memoria.

④ `MAP_ANONYMOUS` indica que la memoria no está respaldada por ningún archivo, por lo que su contenido inicial será cero. Mientras que `MAP_PRIVATE` establece que la región de memoria es privada.

Lo interesante es que si se cambia `MAP_PRIVATE` por `MAP_SHARED` la región de memoria reservada es memoria compartida:

```
void* p = mmap(
    NULL,
    length,
    PROT_READ | PROT_WRITE,
    MAP_ANONYMOUS | MAP_SHARED, ①
    -1,
    0
);
```

① Memoria anónima y compartida.

Es decir, que al crear un hijo con fork() este tendrá una copia de toda la memoria del proceso padre, excepto esta región en particular, que será la misma que la del padre. Por lo tanto, escribiendo y leyendo en esa región, ambos procesos pueden comunicarse.

En anom-shared-memory.cpp se puede ver un ejemplo muy simple, similar a fork-pipe.cpp pero utilizando memoria compartida para comunicar ambos procesos. Como se puede apreciar, la versión que usa memoria compartida es bastante más sencilla que la que utiliza tuberías.

En Microsoft Windows se puede hacer algo similar con CreateFileMapping():

```
HANDLE hMapFile = CreateFileMapping( ③
    INVALID_HANDLE_VALUE,
    NULL,
    PAGE_READWRITE, ①
    0,
    length,         ②
    NULL
);
```

① Permisos de acceso para la memoria reservada.

② Cantidad de memoria a reservar en bytes.

③ A diferencia de mmap(), CreateFileMapping() crea un objeto de memoria compartida, pero no hace visible esa memoria para nuestro proceso. Para eso hay que llamar a MapViewOfFile() pasándole el manejador `hMapFile` devuelto por CreateFileMapping().

11.2. Memoria compartida con nombre

La **memoria compartida con nombre** es pública para el resto del sistema, por lo que teóricamente cualquier proceso con permisos puede acceder a ella para comunicarse con otros procesos.

Como ocurre en las tuberías con nombre, los **objetos de memoria compartida con nombre** hay que crearlos antes de comenzar a utilizarlos. Para eso los sistemas POSIX ofrecen la función shm_open().

```
int shmfd = shm_open(    ④
    "/foo-shm",          ①
    O_RDWR | O_CREAT,    ②
    0666                 ③
);
```

① Nombre que identifica al objeto de memoria compartida. Como ocurre con los archivos, varios procesos pueden acceder al mismo objeto indicando el mismo nombre.

② Valores que indican diferentes opciones a la hora de abrir el objeto. Por ejemplo, usando `O_RDWR` indicamos que se abra para lectura y escritura. Mientras que con `O_CREAT` se indica que el objeto debe crearse si no existía previamente.

③ Indica los permisos del objeto de memoria compartida al crearlo nuevo, de forma similar a los permisos que se aplican a los archivos en el sistema de archivos.

El valor devuelto por shm_open() es el descriptor del objeto de memoria compartida, que utilizaremos posteriormente con mmap() al reservar una región de la memoria de nuestro proceso donde ese objeto de memoria compartida será visible:

```
void* p = mmap(
    NULL,
    length,                  ②
    PROT_READ | PROT_WRITE,
    MAP_SHARED,              ①
    shmfd,                   ①
    0                        ②
);
```

① Al pasar el descriptor del objeto de memoria compartida, ya no se puede indicar `MAP_ANONYMOUS`.

② Se puede hacer visible para el proceso todo el objeto de memoria compartida o solo una parte. Para esto último se indica el tamaño de la región y el desplazamiento dentro del objeto, que es el último argumento de mmap().

Un objeto de memoria compartida recién creado tiene tamaño 0. Para redimensionarlo se utiliza ftruncate(), que lo que necesita es el descriptor del objeto y el nuevo tamaño.

En shared-memory-server.c y shared-memory-client.cpp se puede ver el ejemplo de un programa que muestra periódicamente la hora del sistema. En este caso controlado por otro mediante memoria compartida. Ambos programas usan la clase definida en shared_memory.hpp para

gestionar el objeto de memoria compartida. Sus métodos muestran de forma práctica cómo utilizar las llamadas al sistema comentadas.

En Microsoft Windows también se utiliza CreateFileMapping() para crear el objeto de memoria compartida con nombre. Simplemente hay que indicar el nombre en el último argumento de la función.

```
HANDLE hMapFile = CreateFileMapping(
    INVALID_HANDLE_VALUE,
    NULL,
    PAGE_READWRITE,
    0,
    length,
    "Global\\FooMemoriaCompartida" ①
);
```

① Nombre del nuevo objeto de memoria compartida.

Chapter 12. Hilos

Tiempo de lectura: 35 minutos

En el modelo de proceso que hemos descrito hasta el momento, cada proceso tiene una única secuencia de instrucciones que se ejecuta en la CPU. Si los procesos solo pueden tener una única secuencia de instrucciones, solo pueden realizar una tarea a la vez. Por ejemplo, en un procesador de textos en un sistema operativo con este modelo de procesos, el usuario nunca podría escribir al mismo tiempo que se comprueba la ortografía. Si queremos hacer varias tareas al mismo tiempo, estamos obligados a crear varios procesos y seleccionar un mecanismo de comunicación para que estos colaboren.

Por eso muchos sistemas operativos modernos han extendido el concepto de proceso para permitir que cada uno tenga múltiples secuencias de instrucciones para ejecutarse en la CPU. A cada una de estas secuencias de instrucciones se las conoce como **hilo** de ejecución. Los procesos con varios hilos pueden realizar varias tareas a la vez.

12.1. Introducción

Desde que introducimos el concepto de **proceso** hemos considerado que es la unidad básica de uso de la CPU. Es decir, que la CPU se asignaba a los procesos, que la usaban para ejecutar sus instrucciones. Sin embargo, en los **sistemas operativos multihilo** es el **hilo** la unidad básica de uso de la CPU.

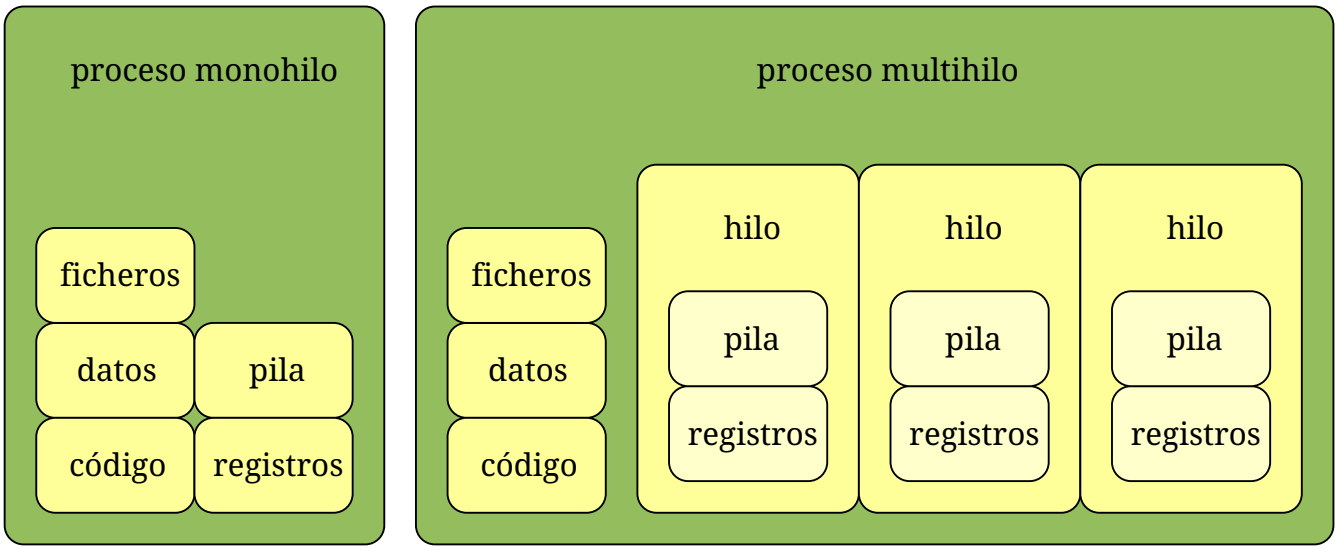


Figura 37. Comparación entre procesos monohilo y proceso multihilo.

Cada hilo tiene una serie de recursos propios dentro del proceso (véase la Figura 37):

- **El identificador del hilo** es único para cada hilo y sirve para identificarlos, de la misma manera que lo hace el identificador de proceso con cada proceso.
- **El contador de programa** es el registro de la CPU que indica la dirección de la próxima instrucción del hilo que debe ser ejecutada por la CPU.
- **Los registros de la CPU**, cuyos valores son diferentes en cada hilo, puesto que, aunque

todos los hilos ejecutan el mismo programa, pueden ejecutar diferentes partes del mismo.

- **La pila** contiene datos temporales como argumentos y direcciones de retorno de las funciones y variables locales. Al igual que ocurre con los registros de la CPU, cada hilo necesita su pila porque recorre el programa de manera diferente.

Sin embargo hay otros recursos del proceso que se comparten entre todos los hilos del mismo (véase la Figura 37):

- **El código del programa**. El programa es el mismo para todos los hilos.
- **Los segmentos BSS y de datos y el montón**. Las secciones de datos diferentes de la pila son accesibles a todos los hilos. Eso quiere decir, por ejemplo, que cualquier hilo puede acceder y modificar una variable global o una asignada dinámicamente mediante malloc() o new.
- **Otros recursos del proceso** como archivos, *sockets*, tuberías y dispositivos abiertos, regiones de memoria compartida, señales, directorio actual de trabajo, entre muchos otros recursos.

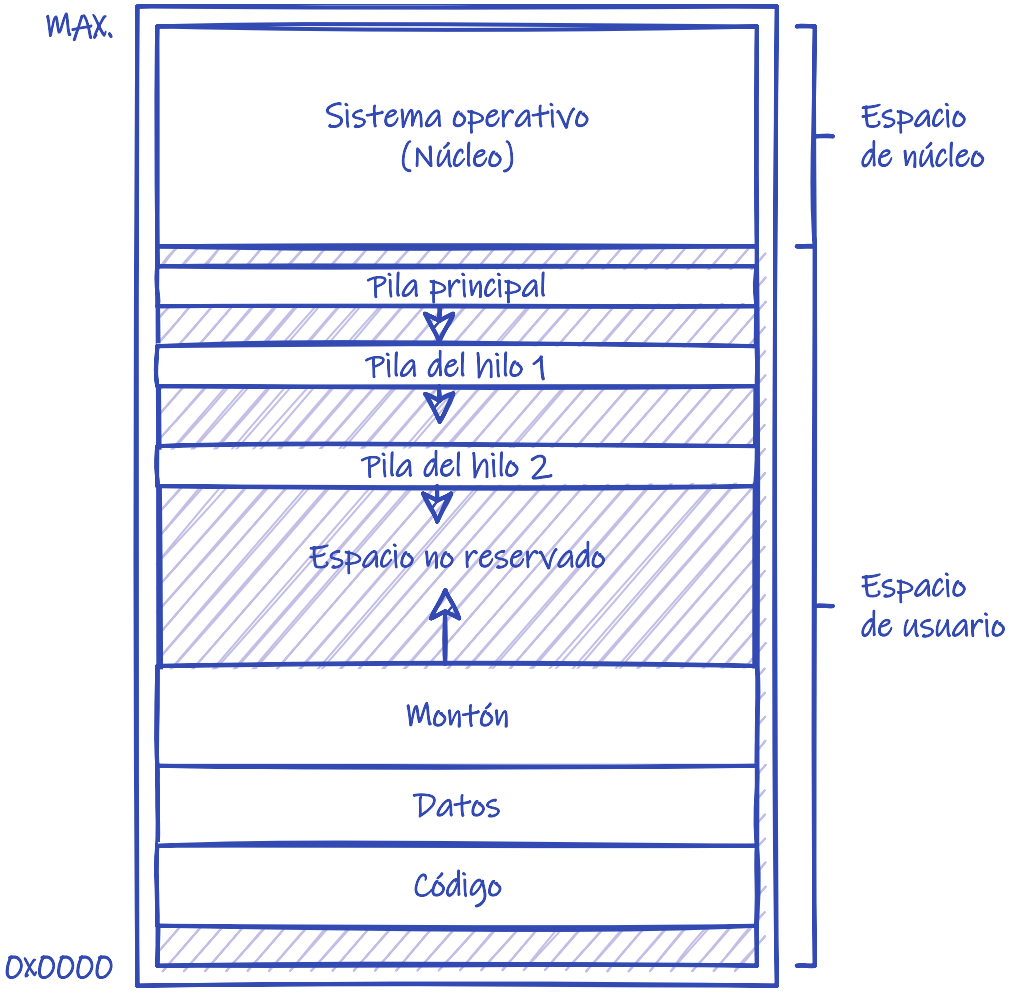


Figura 38. Anatomía de un proceso multihilo en memoria.

En la Figura 38 se puede observar como cambia la disposición de los elementos de un proceso en la memoria cuando es multihilo, respecto a lo que vimos en el Apartado 9.1.

12.2. Beneficios

Son muchos los beneficios que aporta la programación multihilo:

12.2.1. Tiempo de respuesta

Una aplicación multihilo interactiva puede continuar ejecutando tareas aunque uno o varios hilos de la misma estén bloqueados o realizando operaciones muy lentamente, mejorando así el tiempo de respuesta al usuario.

Por ejemplo, un navegador web multihilo puede gestionar la interacción del usuario a través de un hilo, mientras el contenido solicitado se descarga en otro. Para hacer lo mismo en un navegador monohilo habría que utilizar comunicaciones asíncronas, de lo contrario, mientras el proceso está en estado **esperando**, a la espera de que lleguen los datos a través de la red, no puede atender las acciones del usuario.

12.2.2. Compartición de recursos

En sistemas operativos monohilo se pueden crear varios procesos y comunicarlos mediante memoria compartida para conseguir algo similar a lo que ofrecen los sistemas multihilo. Sin embargo, al utilizar hilos, las tareas ejecutadas en ellos comparten los recursos automáticamente, sin que tengamos que hacer nada. Además, los hilos no solo comparten la memoria, sino también otros muchos recursos del proceso. Por lo que los hilos son una forma más conveniente de tener procesos que realizan diferentes actividades al mismo tiempo.

12.2.3. Economía

Reservar memoria y otros recursos para la creación de un proceso es muy costoso. Por eso los sistemas operativos modernos han desarrollado diversas técnicas para que sea lo más eficaz posible.

Aun así, puesto que los hilos comparten los recursos de los procesos a los que pertenecen, son mucho más económicos de crear. También es más económico el cambio de contexto entre ellos, ya que hay que guardar y recuperar menos información al cambiar entre dos hilos de un mismo proceso.

Por ejemplo, en Microsoft Windows crear un proceso puede ser 300 veces más costoso que un hilo. Mientras que en sistemas Linux es 3 veces más lento, por la eficiencia de fork().

12.2.4. Aprovechamiento de las arquitecturas multiprocesador.

En los sistemas multiprocesador diferentes hilos pueden ejecutarse en paralelo en distintos procesadores. Por el contrario, un proceso monohilo solo se puede ejecutar en una CPU a la vez, independientemente de cuantas CPU estén disponibles para ejecutarlo.

12.3. Soporte multihilo

Las **librerías de hilos** proporcionan al programador la interfaz de programación para crear y

gestionar los hilos de su proceso. Hay dos formas de implementar una librería de hilos: en el espacio de usuario o en el núcleo.

12.3.1. Librería de hilos en el espacio de usuario

La librería de hilos se puede implementar en el espacio de usuario, junto al código y los datos del proceso, sin requerir ningún soporte especial por parte del núcleo.

Estos hilos no existen para el núcleo del sistema operativo, solo para el proceso que los ha creado. Por ese motivo se los denomina **hilos de usuario** o **hilos del nivel de usuario**.

Como el código y los datos de la librería residen en el espacio de usuario, invocar una función de la misma se reduce a una simple llamada a una función, evitando el coste de hacer llamadas al sistema.

12.3.2. Librería de hilos en el núcleo

Si la librería de hilos se implementa en el núcleo, es el núcleo del sistema el que se encarga de darles soporte. Por ese motivo se los denomina **hilos de núcleo** o **hilos del nivel de núcleo**.

Aparte del **PCB** que vimos en el Apartado 9.3, cada hilo tiene una estructura llamada **bloque de control del hilo** o TCB (*Thread Control Block*) que representa a cada hilo en el sistema operativo y que guarda información sobre su estado de actividad actual.

En estos sistemas, es el hilo la unidad básica de uso de la CPU. Son los hilos los que se mueven por los estados del Figura 28 y las colas de la Figura 29 y no los procesos. El planificador de la CPU selecciona un hilo para ejecutarse en la CPU de entre todos los que están en el estado **preparado** en el sistema y el **cambio de contexto** asigna la CPU a un hilo distinto al que la tiene asignada en el momento actual.

Por tanto, es en **TCB** —y no en el **PCB**— donde se guarda la información privada del hilo necesaria para la gestión de los estados y para el cambio de contexto, como: los valores de los registros de la CPU y el contador de programa, el estado o la información de planificación de la CPU; además de un puntero al **PCB** al que pertenece el hilo con el resto de la información privada del proceso.

Como el código y los datos de la librería residen en el espacio del núcleo, invocar una función de la misma requiere frecuentemente hacer una llamada al sistema. Obviamente, la librería del sistema ofrece funciones para no tener que hacer la llamada al sistema directamente.

En la actualidad, en los diferentes sistemas operativos se pueden encontrar librerías de ambos tipos. Por ejemplo, la librería de hilos de Windows API se implementa en el núcleo (véase «Using Processes and Threads — Microsoft Docs») mientras que la librería de hilos POSIX Threads —frecuentemente utilizada en los sistemas POSIX— puede ser de ambos tipos, dependiendo solamente del sistema donde se implemente. En Linux y en la mayor parte de los UNIX modernos, POSIX Threads se implementa en el núcleo del sistema.

12.4. Modelos multihilo

Las distintas formas de implementar los hilos comentadas anteriormente —en espacio de usuario o

en el núcleo— no son excluyentes, ya que en un sistema operativo concreto se pueden implementar ambas, una de las dos o ninguna.

A continuación veremos los modelos a los que han dado lugar las distintas combinaciones.

12.4.1. Muchos a uno

En el modelo **muchos a uno** los hilos que ve el proceso se mapean en una única «entidad planificable en la CPU» del núcleo.

Este, por lo general, es el modelo utilizado cuando el núcleo no soporta múltiples hilos de ejecución. En ese caso, la única entidad planificable en la CPU que conoce el núcleo es el proceso, la librería de hilos se implementa en el espacio de usuario —dentro del proceso— y los hilos que ve el proceso son **hilos de usuario** (véase la Figura 39).

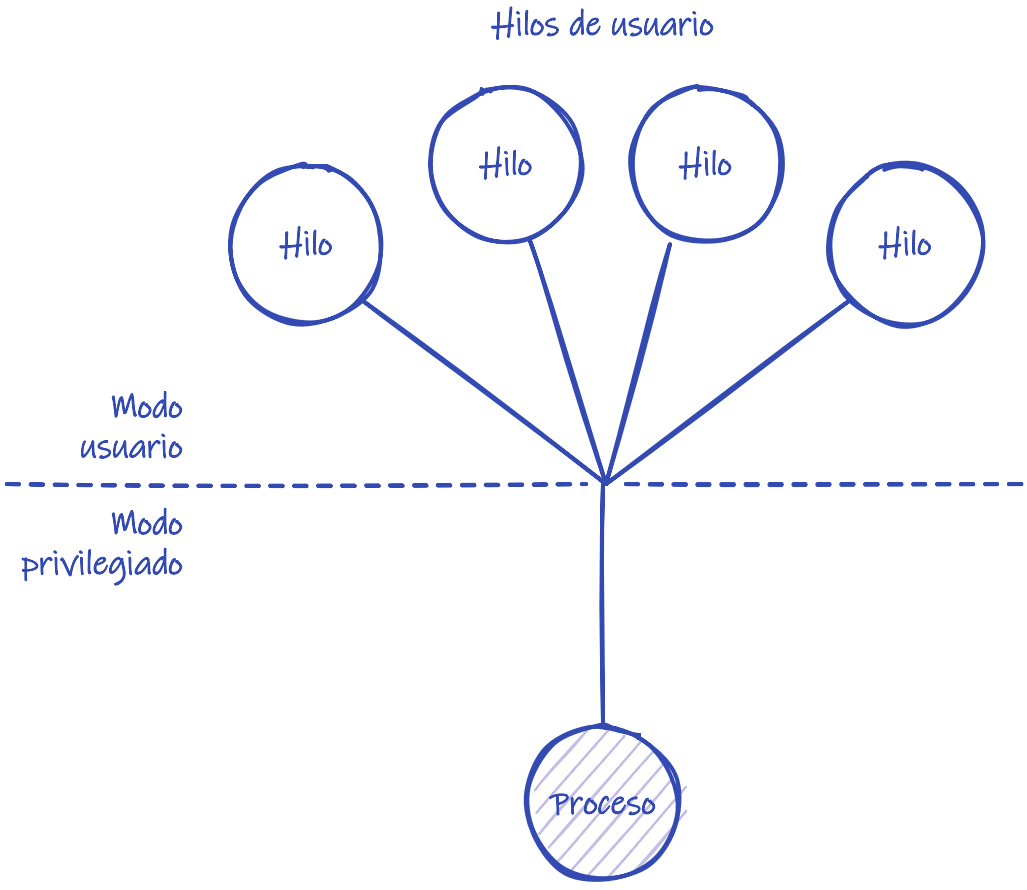


Figura 39. Modelo muchos a uno.

Las principales características de este modelo son:

- La gestión de hilos se hace con una librería en el espacio de usuario, por lo que los hilos se pueden crear de forma rápida y con poco coste. Como hemos visto anteriormente, la invocación de las funciones de la librería se hace por medio de simples llamadas a funciones.
- Si uno de los hilos solicita al sistema operativo una operación que deba ser bloqueada a la espera —por ejemplo, operaciones de E/S sobre archivos, comunicaciones o esperar a que otro proceso termine— todo el proceso es bloqueado, no pudiendo ejecutarse otros hilos

del mismo proceso mientras tanto. Eso significa que si nuestros hilos hacen ese tipo de operaciones, el resultado es como si no tuviéramos hilos.

- Como solo un hilo puede ser asignado al proceso, los hilos de un mismo proceso no se pueden ejecutar en paralelo en sistemas multiprocesador. El planificador de la librería de hilos es el encargado de determinar qué hilo de usuario es asignado al proceso y este solo puede ejecutarse en una única CPU al mismo tiempo.

El problema del bloqueo de procesos puede ser evitado interceptando las llamadas a funciones de la librería del sistema, para evitar el uso de llamadas al sistema que se puedan bloquear y sustituirlas por versiones equivalentes pero asíncronas.

Por ejemplo, si un hilo llamase a las funciones recv() o send() de la librería del sistema, habría que hacer que realmente se invocase una versión diferente que utilizase estas funciones de forma asíncrona. Mientras la operación es ejecutada por el sistema operativo, en lugar de retornar de la función, se llama al planificador de la librería de hilos para que la ejecución continúe con otro hilo del proceso, dejando suspendido el que tiene pendiente la operación. Obviamente, el planificador de la librería de hilos debe estar al tanto de cuándo las operaciones asíncronas son completadas para poder volver a planificar los **hilos de usuario** suspendidos.

Este procedimiento es a todas luces bastante complejo y requiere versiones no bloqueantes de todas las llamadas al sistema —que no siempre existen— así como modificar o interceptar de alguna forma las funciones bloqueantes de la librería del sistema para implementar el comportamiento descrito.

Implementaciones

A este modelo de hilos frecuentemente se lo llama Green Threads. En Java 1.1 era el único modelo soportado, pero debido a sus limitaciones se implementó el soporte del modelo **uno a uno** en versiones posteriores.

Otras implementaciones de este modelo son las fibras de Windows API, Stackless Python y GNU Portable Threads. Estas implementaciones son muy útiles en los sistemas monohilo, de cara a poder ofrecer cierto soporte de hilos a las aplicaciones. Pero también lo son en los sistemas multihilo, ya que debido a su bajo coste en recursos y a su alta eficiencia son ideales cuando la cantidad de hilos a crear —el nivel de concurrencia— va a ser previsiblemente muy alta.

12.4.2. Uno a uno

En el modelo **uno a uno** cada hilo que ve el proceso se mapea en una «entidad planificable en la CPU» diferente del núcleo.

Este, por lo general, es el modelo utilizado cuando el núcleo del sistema operativo soporta hilos de ejecución. En este caso, la librería de hilos se implementa en el núcleo, por lo que las entidades que planifica el núcleo en la CPU son los **hilos de núcleo** y los procesos pueden gestionar estos hilos mediante llamadas al sistema.

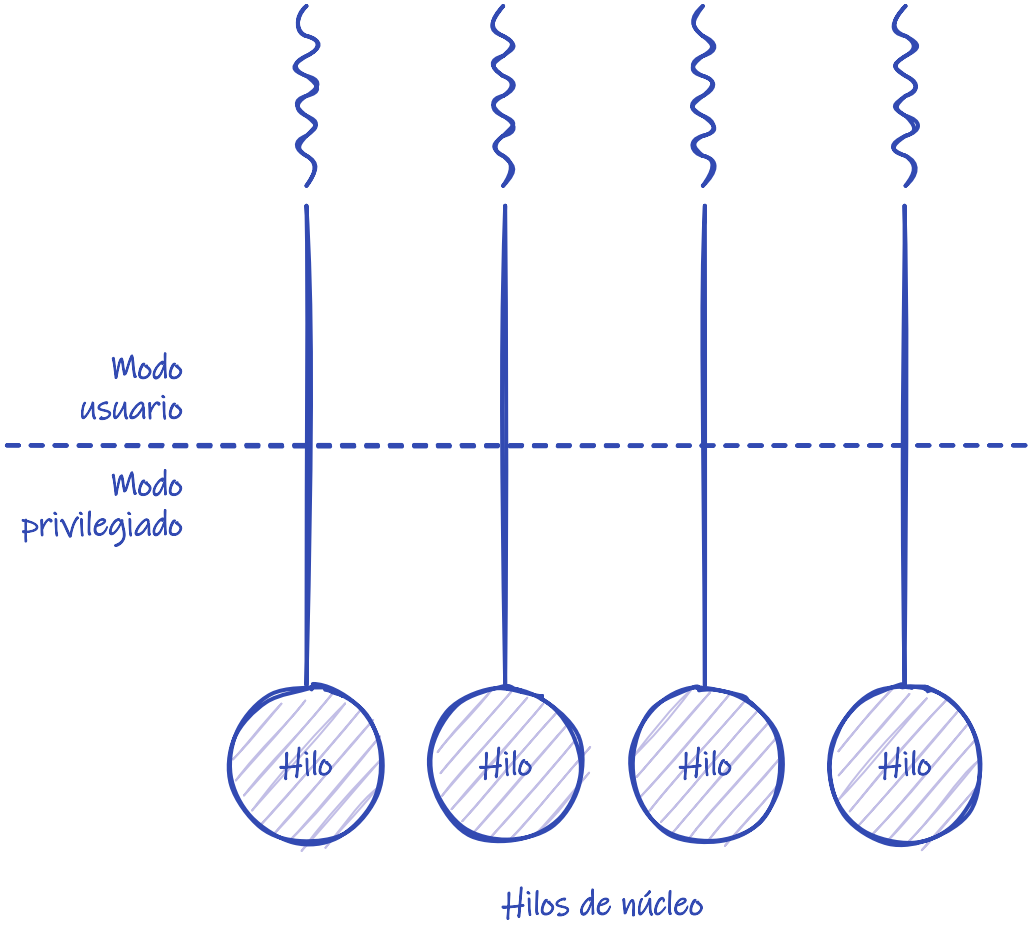


Figura 40. Modelo uno a uno.

Las principales características de este modelo son:

- Permite a otros hilos del mismo proceso ejecutarse aun cuando uno de ellos haga una llamada al sistema que debe bloquearse. El núcleo se encarga de ponerlo en espera y planificar en la CPU a otro de los hilos preparados para ejecutarse de entre todos los existentes en el sistema.
- Permite paralelismo en sistemas multiprocesador, ya que diferentes hilos pueden ser planificados por el núcleo en distintos procesadores.
- Crear un hilo para un proceso implica crear ciertas estructuras de datos en el núcleo. Debido a que la cantidad de memoria disponible para el núcleo suele estar limitada, muchos sistemas restringen la cantidad máxima de **hilos de núcleo** soportados.
- La gestión de los hilos se hace con una librería en el espacio de núcleo, lo que requiere que el proceso haga llamadas al sistema para gestionarlos. Esto siempre es más lento que invocar simplemente una función, como ocurre en el modelo **muchos a uno**.

Este modelo se utiliza en la mayor parte de los sistemas operativos multihilo modernos. Linux, Microsoft Windows —desde Windows 95— Solaris 9 y superiores, macOS y la familia de UNIX BSD; son ejemplos de sistemas operativos que utiliza el modelo **uno a uno**.

12.4.3. Muchos a muchos

En teoría debería ser posible aprovechar lo mejor de los dos modelos anteriores con una librería de hilos en el núcleo, para crear **hilos de núcleo**, y otra en el espacio de usuario, para crear **hilos de usuario**. Así los desarrolladores pueden utilizar la librería de hilos en el espacio de usuario para crear tantos hilos como quieran y que se ejecuten sobre los **hilos de núcleo**.

El planificador de la librería de hilos se encarga de determinar qué hilo de usuario es asignado a qué hilo de núcleo. Mientras que el planificador de la CPU asigna la CPU a alguno de los **hilos de núcleo** del sistema.

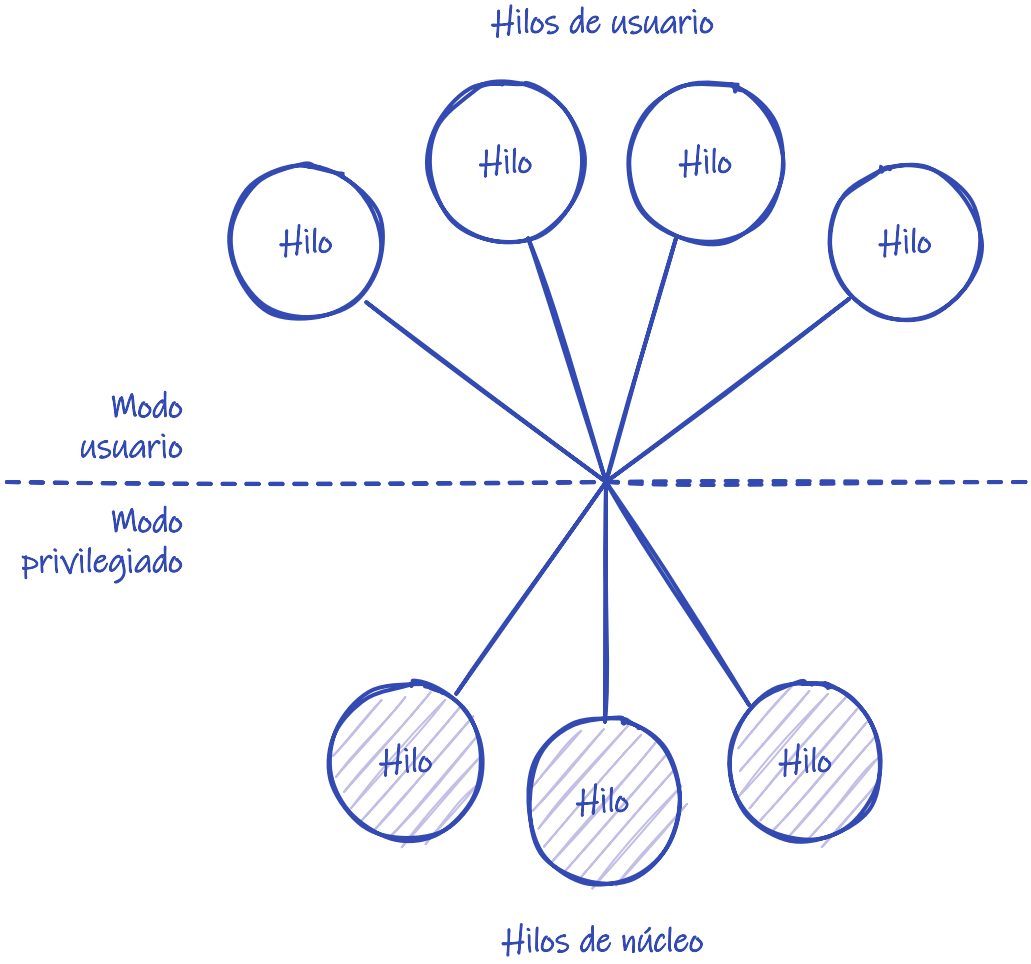


Figura 41. Modelo muchos a muchos.

En el modelo **muchos a muchos** se mapean los **hilos de usuario** en un menor o igual número de **hilos de núcleo** del proceso (véase la Figura 41).

- Permite paralelismo en sistemas multiprocesador, ya que diferentes **hilos de núcleo** pueden ser planificados en distintos procesadores y en cada uno puede ejecutarse cualquier hilo de usuario.
- Permite a otro hilo de usuario del mismo proceso ejecutarse cuando un hilo hace una llamada al sistema que debe bloquearse, puesto que si esto ocurre el correspondiente hilo de núcleo se queda bloqueado. Sin embargo, el resto de los **hilos de usuario** pueden seguir ejecutándose en los otros **hilos de núcleo** del proceso.

Este modelo se soportaba en sistemas FreeBSD y versiones antiguas de NetBSD, así como en UNIX comerciales, como: Solaris 8 y anteriores, IRIX, HP-UX y Tru64 UNIX. También Microsoft Windows —a partir de Windows 7— soporta este modelo gracias a incorporar un mecanismo denominado planificación en modo usuario (véase «User-Mode Scheduling — Microsoft Docs»).

Algunos lenguajes de programación implementan el modelo **muchos a muchos** sobre el modelo **uno a uno** soportado por la mayoría de sistemas operativos modernos. Ese es el caso de Go, Erlang y Elixir.

Activación del planificador

Tanto en el modelo **muchos a muchos** como en el de **dos niveles** es necesario cierto grado de coordinación entre el núcleo y la librería de hilos del espacio de usuario. Dicha comunicación tiene como objeto ajustar dinámicamente el número de **hilos de núcleo** para garantizar la máxima eficiencia.

Uno de los esquemas de comunicación se denomina **activación del planificador** y consiste en que el núcleo informa a la librería de hilos en espacio de usuario que una llamada al sistema va a bloquear un hilo de un proceso. Antes de dicha notificación, el núcleo se encarga de crear un nuevo hilo de núcleo en el proceso y se lo pasa a la librería de hilos en la notificación. Así, el planificador de la librería puede asignarle alguno de los otros **hilos de usuario**, evitando el bloqueo completo del proceso y ajustando el número de **hilos de núcleo** dinámicamente.

Debido a la complejidad del mecanismo descrito anteriormente y a la dificultad de coordinar el planificador de la librería de hilos con el de la CPU para obtener un rendimiento óptimo, sistemas como Linux y Solaris —a partir de la versión 9— han optado finalmente por el modelo **uno a uno**.

Con el objetivo de evitar los problemas derivados del coste de dicho modelo, los desarrolladores de Linux han preferido concentrar sus esfuerzos en conseguir un planificador de CPU más eficiente, así como en reducir los costes de la creación de **hilos de núcleo**.

12.4.4. Dos niveles

Existe una variación del modelo **muchos a muchos** donde, además de funcionar de la forma comentada anteriormente, se permite que un hilo de usuario quede ligado indefinidamente a un único hilo de núcleo, como en el modelo **uno a uno**.

Esta variación se denomina, en ocasiones, modelo de **dos niveles** (véase la Figura 42).

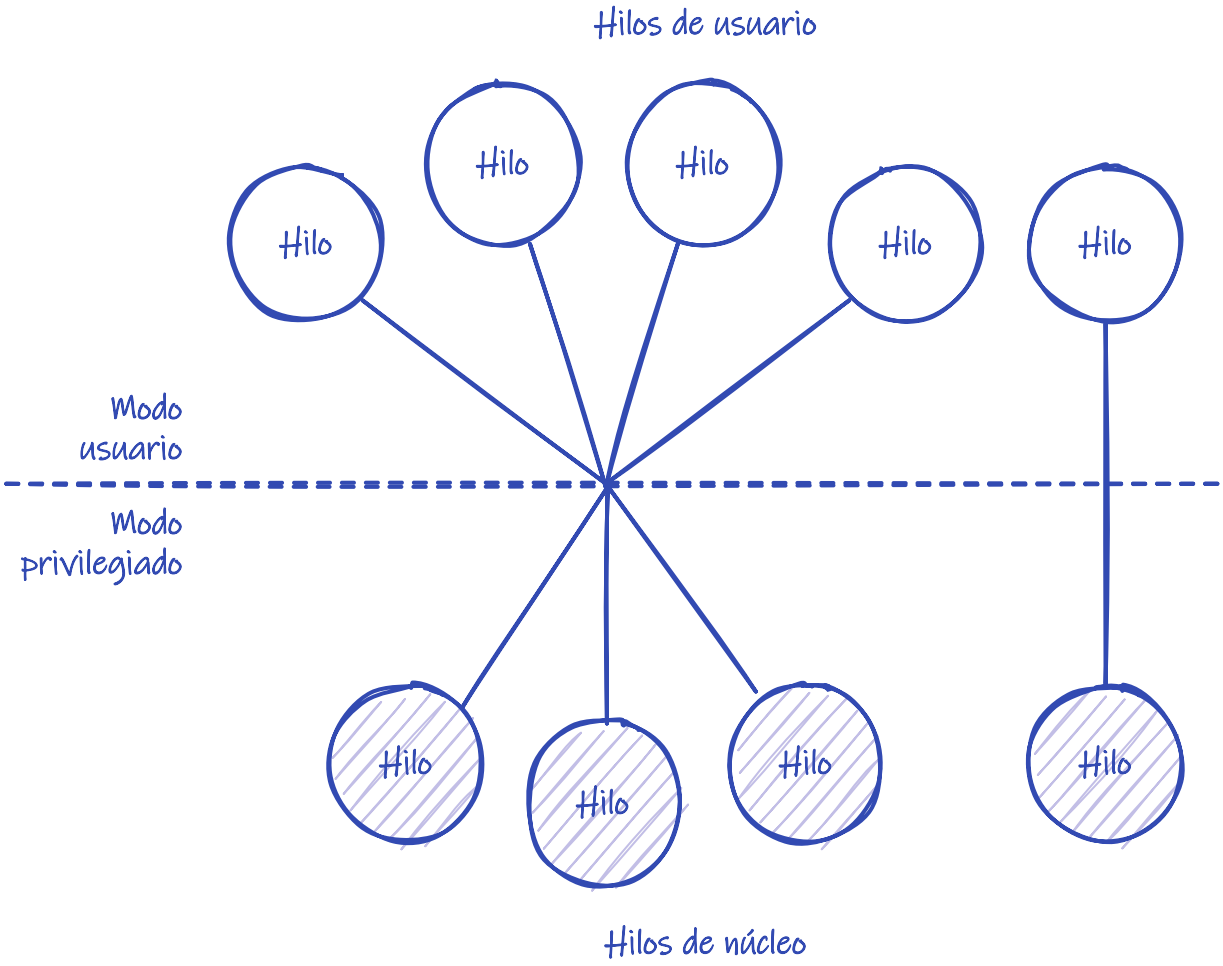


Figura 42. Modelo muchos a uno.

12.5. Operaciones sobre los hilos

Como ocurre con los procesos, es necesario que los hilos pueden ser creados y eliminados dinámicamente, por lo que los sistemas operativos deben proporcionar servicios para la creación y cancelación de los mismos.

12.5.1. Creación de hilos

En un sistema operativo con librería de hilos implementada en el núcleo, todo proceso se crea con un hilo, denominado **hilo principal**. Este es con el que comienza a ejecutarse el programa al entrar en main() y el que provoca la terminación de todo el proceso —incluida la terminación de los otros hilos que existan— al retornar de dicha función.

El **hilo principal** puede crear otros hilos y estos, a su vez, crear los hilos que necesiten. Pero, a diferencia de lo que ocurre con los procesos, no existe una relación de padres a hijos ni se crea un árbol de hilos. Excepto por la característica especial del **hilo principal** de que su finalización significa la terminación del proceso, todos los hilos son iguales entre sí.

En el Ejemplo 5 se puede ver cómo se usa pthread_create() en sistemas POSIX que implementan POSIX Threads para crear varios hilos y esperar a que terminen con pthread_join().

Ejemplo 5. Calcular el factorial de un número con POSIX Threads.

Vamos a calcular el factorial de 122 repartiendo la tarea entre dos hilos con el objeto de paralelizar los cálculos en procesadores multinúcleo.

El código fuente completo de este ejemplo está disponible en pthreads.cpp y, además, permite indicar el número que queramos para calcular el factorial. En threads.cpp se puede estudiar un ejemplo equivalente, pero usando std::thread de la librería estándar de C++, por lo que también compila en sistemas no POSIX, como Microsoft Windows.

```
struct factorial_thread_args ⑨
{
    BigInt number;
    BigInt lower_bound;
    BigInt result;
};

void* factorial_thread (void* arg) ⑦ ⑧ ⑨
{
    std::cout << fmt::format( "Hilo creado: 0x{:x}\n", pthread_self() );    ⑩

    factorial_thread_args* args = static_cast<factorial_thread_args*>(arg); ⑨
    args->result = calculate_factorial( args->number, args->lower_bound );

    return &args->result; ⑫
}

int main()
{
    int return_code;
    pthread_t thread1, thread2; ① ②

    factorial_thread_args thread1_args {
        122,    // El primer hilo calcula el factorial multiplicando
        61,     // desde 122 a 61.
        0
    };

    factorial_thread_args thread2_args {
        60,     // El segundo hilo calcula el factorial multiplicando
        2,      // desde 60 a 2
        0
    };

    int return_code = pthread_create( ①
        &thread1,           ③
        nullptr,            ⑥
        factoria_thread,    ⑦
        &thread_args );     ⑧ ⑨
```

```
    if (return_code) ④
    {
        std::cerr << fmt::format( "Error ({}) al crear el hilo: {}\n",
            return_code, strerror(return_code) ); ⑤
        return EXIT_FAILURE;
    }

    return_code = pthread_create( &thread2, /* ... */ );
    if (return_code)
    {
        // ...
    }

    BigInt* thread1_result, *thread2_result;
    pthread_join( ⑪
        thread1,
        reinterpret_cast<void**>(&thread1_result) ); ⑬
    pthread_join( thread2, reinterpret_cast<void**>(&thread2_result) );

    // Multiplicar ambos resultados para obtener el factorial
    auto result = *thread1_result * *thread2_result;

    std::cout << fmt::format( "El factorial de {} es {}\n",
        number.to_string(), result.to_string() );

    return EXIT_SUCCESS;
}
```

① En POSIX Threads se usa pthread_create() para crear hilos. Devuelve un manejador de tipo `pthread_t` que podemos usar con otras funciones de la API para indicar el hilo que queremos gestionar.

② `pthread_t` no es el equivalente al PID de los hilos. Si el sistema implementa la librería de hilos en el núcleo, por lo general, cada hilo tiene un identificador único; pero POSIX Threads no ofrece una forma de obtenerlo.

③ La variable `pthread_t` se pasa a pthread_create() como puntero para que al retornar, si todo ha ido bien, contenga el manejador del hilo.

④ Si el hilo se puede crear, pthread_create() devuelve 0. En caso contrario devuelve un código de error.

⑤ Los códigos de error son los mismos que hasta ahora veíamos en errno. Así que podemos llamar a strerror() pasando el valor retornado, para obtener un texto descriptivo del error.

⑥ Es opcional pasar a pthread_create() una estructura con atributos tales como: tamaño y posición de la pila, política y parámetros de planificación, entre otros.

⑦ Todo hilo tiene una función principal que será donde comience la ejecución del hilo. Cuando esa función termine, el hilo finalizará.

⑧ Los hilos pueden recibir un argumento en la forma de un puntero a `void*`. Si queremos pasar varios, lo más sencillo es crear una estructura.

⑨ En este ejemplo definimos `factorial_thread_args` para pasar los argumentos a los hilos y lo pasamos como `void *` a la función principal. Allí hacemos un *typecast* para recuperar el puntero a la estructura `factorial_thread_args` y poder acceder a sus campos.

⑩ En cualquier momento se puede llamar a pthread_self() para obtener el manejador `pthread_t` del hilo actual.

⑪ El hilo que invoca pthread_join() se queda dormido hasta que el hilo indicado en el primer argumento termine. Si el hilo principal sale de main() sin esperar a que todos los hilos del proceso terminen, estos mueren inmediatamente, junto con el proceso.

⑫ El hilo puede retornar un resultado mediante un puntero 'void*'. Esto se indica en la sentencia `return` de la función principal del hilo o invocando pthread_exit() para terminar.

⑬ La función pthread_join() acepta un puntero a `void*` para devolver ese valor de retorno al hilo que la invoca.

Como los hilos de POSIX Threads devuelven punteros, es importante no intentar devolver variables locales, ya que se destruirán cuando el hilo termine y el punto devuelto no será válido.

Una alternativa es devolver los resultados a través de la estructura pasada como argumento. Por ejemplo, el campo `result` de la estructura `factorial_thread_args` ofrece una manera más cómoda de obtener el resultado del cálculo de cada hilo.

12.5.2. Cancelación de hilos

La **cancelación** es la operación de terminar un hilo antes de que termine su trabajo. Por ejemplo, en un navegador web un hilo se puede encargar de la interfaz de usuario mientras otros hilos se encargan de descargar las páginas y las imágenes de la misma. Si el usuario pulsa el botón **Cancelar** es necesario que todos los hilos que intervienen en la descarga sean cancelados.

Esto puede ocurrir de dos maneras:

- En la **cancelación asíncrona** el hilo termina inmediatamente. Esto puede causar problemas al no liberarse los recursos reservados en el proceso por parte del hilo Por ejemplo, antes de terminar no se cierran archivos abiertos ni se libera memoria de los que solo este hilo tiene los descriptores de archivo y los punteros, respectivamente.

  Además, si el hilo que termina estaba modificando datos que compartía con otros hilos, estos cambios podrían quedar a medias. Esto puede dejar las estructuras de datos compartidas en un estado inconsistente, causando problemas en otros hilos.

- En la **cancelación en diferido** el hilo comprueba periódicamente cuando debe terminar. Si no se tiene cuidado, los problemas pueden ser similares a los de la **cancelación asíncrona**. La diferencia es que ahora el desarrollador conoce de antemano los puntos donde podría terminar el hilo, lo que da una oportunidad de introducirlos solo donde sea seguro terminar.

Se denomina **fuga de memoria** al error que ocurre cuando un bloque de memoria reservada no se libera durante la ejecución del programa. También pueden ocurrir fugas con otros recursos del sistema operativo, como: archivos, *sockets*, colas de

> mensajes o regiones de memoria compartida.
>
> Generalmente ocurre porque se pierden todas las referencias a un recurso, por lo que ya no hay oportunidad de liberarlo. Por ejemplo, cuando se cancela un hilo que es el único que tiene algunas referencias, sin liberar antes esos recursos.

12.5.3. Cancelación en POSIX Threads

En POSIX Threads un hilo puede solicitar la cancelación de otro hilo usando pthread_cancel().

```
int return_code = pthread_cancel(thread);
```

El hilo identificado por el manejador `thread` será cancelado si está configurado como cancelable. Por defecto todos los hilos son cancelables, pero eso lo puede cambiar el propio hilo llamando a pthread_setcancelstate():

```
int return_code = pthread_setcancelstate(
    PTHREAD_CANCEL_DISABLE, ①
    &oldstate               ②
);
```

① Con `PTHREAD_CANCEL_DISABLE` se desactiva la cancelación en el hilo que llama la función. El otro valor posible es `PTHREAD_CANCEL_ENABLE`.

② La función devuelve a través de un puntero a `int` el valor anterior del estado de cancelación.

El tipo de cancelación se puede configurar con pthread_setcanceltype():

```
int return_code = pthread_setcanceltypr(
    PTHREAD_CANCEL_DEFERRED,    ①
    &oldtype                    ②
);
```

① Con `PTHREAD_CANCEL_DEFERRED` se activa la **cancelación en diferido**, que de todas formas es el tipo de cancelación por defecto. El otro valor posible es `PTHREAD_CANCEL_ASYNCHRONOUS`, que corresponde con la **cancelación asíncrona**.

② La función devuelve a través de un puntero a `int` el valor anterior del tipo de cancelación.

Se pueden cambiar entre estado y tipo de cancelación en cualquier momento, según lo que encaje mejor con las características de las distintas partes del código.

Cancelación asíncrona

Por los motivos comentados anteriormente, no es recomendable la **cancelación asíncrona**, a menos que estemos muy seguros de que no puede causar problemas. Uno de los pocos casos con los que es compatible es en bucles 100% dedicados a ejecutar cálculos en la CPU, como el siguiente:

```
int factorial = 1;
for ( int i = 2; i <= number; i++ )
{
    factorial = factorial * i;
}
```

La **cancelación asíncrona** no se debe usar si el código reserva memoria dinámicamente o solicita otros recursos del sistema operativo, porque el hilo podría terminar en cualquier momento sin liberarlos. Tampoco si se modifican estructuras de datos, porque los cambios pueden quedar a medias.

Por ejemplo, si la cancelación ocurre en medio de una llamada a malloc() o new no hay forma de saber si ocurrió antes de que la memoria fuera reservada o después. Incluso puede haber ocurrido en medio de la operación, dejando en estado inconsistente las estructuras de datos que sirven para seguir la pista de las zonas de memoria reservadas y libres.

El estándar POSIX solo indica que las funciones pthread_setcancelstate() y pthread_setcanceltype() deben ser seguras frente a la **cancelación asíncrona** del hilo. En general, no se puede llamar a otras funciones de la librería del sistema de forma segura en un hilo cancelable asíncronamente.

Cancelación en diferido

Por tanto, la **cancelación en diferido** es la mejor alternativa. Con este tipo de cancelación, la terminación del hilo ocurre en puntos concretos del código.

En la terminología de POSIX Threads a estos puntos se los denomina **puntos de cancelación** y la inmensa mayoría de las llamadas al sistema que puede poner el hilo en estado **esperando** lo son por sí mismas. Por ejemplo, open(), close(), read(), write(), recv(), send(), poll() y sleep(), entre muchas otras (véase la lista en la sección «*Cancellation points*» de la documentación de POSIX Threads). Eso significa que seguramente también sean **puntos de cancelación**, las funciones de la librería del sistema y de la librería estándar del lenguaje que utilizan esas llamadas al sistema.

Sabiendo esto, se puede estudiar cada caso. Si no es seguro permitir la cancelación de un hilo en la invocación de una de estas funciones en nuestro código, se puede usar pthread_setcancelstate() para desactivar temporalmente el mecanismo de cancelación. Por ejemplo, una llamada a printf() como ayuda para depurar, en medio de los pasos para modificar una estructura de datos —como una lista enlazada o una cola— introduce un **punto de cancelación** en lugar poco seguro; porque si el hilo se cancela en ese punto, la estructura de datos quedará en estado inconsistente. La solución es eliminar la llamada a printf() o desactivar temporalmente el mecanismo de cancelación.

De forma inversa, se pueden introducir manualmente puntos de cancelación llamando a pthread_testcancel(). Por ejemplo, el siguiente bucle no hace llamadas al sistema, por lo que no tiene puntos de cancelación:

```
int factorial = 1;
for ( int i = 2; i <= number; i++ )
{
    factorial = factorial * i;
```

```
}
```

Eso significa que ese código para calcular el factorial de `number` podría ejecutarse durante bastante tiempo sin ofrecer una oportunidad para cancelar el hilo; incluso aunque es un código muy seguro desde el punto de vista de la cancelación. La solución es introducir manualmente un punto de cancelación:

```
int factorial = 1;
for ( int i = 2; i <= number; i++ )
{
    factorial = factorial * i;
    pthread_testcancel();   ①
}
```

① Comprobar si se ha solicitado la cancelación del hilo y si es así, cancelar el hilo.

La **cancelación en diferido** también presenta retos desde el punto de vista de evitar las fugas de memoria y de otros recursos cuando un hilo es cancelado. Por ejemplo, supongamos que tenemos una función que abre una tubería, crea un hilo para gestionar los mensajes que llegan y devuelve un puntero a una estructura de datos que se puede usar en otras funciones de la librería —de forma similar a fopen() y `FILE*`—:

```
CONN* conn_open( /* ... */ )
{
    CONN* handler = malloc( sizeof(CONN) );
    if(handler == NULL)
    {
        return NULL;
    }

    handler->fifofd = open( /* ... */ );
    if (fifofd < 0)
    {
        free( handler ); ①
        return NULL;
    }

    int return_code = pthread_create(&handler->thread, /* ... */, handler );
    if (return_code)
    {
        close( handler->fifofd ); ②
        free( handler ); ①
        return NULL;
    }

    return handler;
}
```

① Evitar la **fuga de memoria** si open() o pthread_create() fallan.

② Evitar la fuga del *socket* si pthread_create() falla.

Este código y la forma en que maneja los errores funcionan bien en programas monohilo, porque estamos seguros de que al salir de la función o se completaron todas las etapas o ninguna. Es decir, si alguna de las peticiones al sistema falla, las hechas anteriormente se «deshacen» para evitar la fuga de recursos.

Pero no es correcto en programas multihilo porque open() y close() son **puntos de cancelación**. Si `conn_open()` es llamada desde un hilo y ese hilo es cancelado, el hilo podría terminar a mitad de la función, sin liberar `handler`, creando una **fuga de memoria** que no se liberará hasta que el proceso termine. Si `conn_open()` es llamada en múltiples ocasiones, cada una es una oportunidad para perder memoria.

El código anterior se puede mejorar usando **manejadores de limpieza**. Esos manejadores se organizan en una pila de la que se pueden insertar o extraer llamando a pthread_cleanup_push() y pthread_cleanup_pop(), respectivamente. Cuando el hilo es cancelado, la librería extrae los manejadores de la pila y los va ejecutando en orden, antes de terminar.

```
CONN* conn_open( /* ... */ )
{
    CONN* handler = malloc( sizeof(CONN) );
    if(handler == NULL)
    {
        return NULL;
    }

    pthread_cleanup_push(&free, handler); ①

    handler->fifofd = open( /* ... */ );  ②
    if (fifofd < 0)
    {
        pthread_cleanup_pop(1);     // free(handler) ③
        return NULL;
    }

    pthread_cleanup_push(&cleanup_fd, &handler->fifofd);

    int return_code = pthread_create(&handler->thread, /* ... */, handler );
    if (return_code)
    {
        close( handler->fifofd ); ②
        pthread_cleanup_pop(1);     // free(handler) ③
        return NULL;
    }

    pthread_cleanup_pop(0); ④

    return handler;
```

```
}
```

① Nada más reservar la memoria de `CONN` se añade un **manejador de limpieza** que llamará a `free(handler)` si el hilo va a ser cancelado. Así nos aseguramos que `handler` será liberado si el hilo es cancelado.

② La cancelación solo puede ocurrir en los **puntos de cancelación** que son las llamadas a open() y close().

③ En caso de error, el **manejador de limpieza** ya no hace falta, así que se extrae antes de salir de la función. Se llama a pthread_cleanup_pop() con valor distinto de 0 porque así la función extrae el manejador y lo invoca. A fin de cuentas se sale a causa de un error, por lo que sigue siendo necesario ejecutar `free(handler)` para evitar una **fuga de memoria**.

④ Al terminar la función se extraen todos los manejadores de señal, puesto que ya no hacen falta. El argumento 0 hace que pthread_cleanup_pop() no ejecute el manejador de limpieza extraído.

Ahora `conn_open()` maneja correctamente la cancelación del hilo donde se ejecuta, por lo que puede usarse sin problemas en aplicaciones multihilo.

12.5.4. Cancelación de hilos en lenguajes de alto nivel

El mecanismo de cancelación de hilos descrito funciona razonablemente bien en C, pero no con lenguajes de más alto nivel, como C++, Java o C#. Las librerías de hilos suelen ser librerías en C, que no conocen nada de objetos ni de otras particularidades de esos lenguajes.

Por ejemplo, en C++, antes de terminar un hilo, deberían ser llamados todos los destructores de los objetos locales, para evitar **fugas de memoria** y de otros recursos, datos sin escribir y otros problemas derivados de tener objetos que no se destruyen adecuadamente. Lamentablemente, el mecanismo de cancelación de POSIX Threads —y el de otras librerías de hilos, como la de Windows API— no sabe hacer nada de eso. Cada lenguaje debe implementar su propia solución.

En Java y C#, por ejemplo, cuando un punto de cancelación detecta una petición de cancelación emite la excepción `Thread.Interrupt`, que retrocede por la pila de llamadas, liberando las variables locales hasta salir por el método principal del hilo. A este mecanismo se lo denomina **cancelación coordinada**.

En C++ no se ha incluido un mecanismo de cancelación en el estándar hasta C++20. Antes de C++20, la forma recomendada de implementar la cancelación es pasando a los hilos una variable de tipo `bool` con la que señalarles cuándo deben terminar. El código de los hilos debe comprobar frecuentemente el valor de dicha variable y, llegado el momento, terminar retornando ordenadamente por la función principal del hilo.

En C++20 esta estrategia de **cancelación cooperativa** se ha formalizado e incluido en el estándar al introducir la clase std::jthread. Esta nueva clase de hilo puede pasar a la función principal lo que se llama un ***token* de cancelación** —en lugar de una variable tipo `bool`— que se debe comprobar regularmente para saber si hay que terminar el hilo prematuramente.

Por ejemplo, la función del factorial podría hacer uso de esa funcionalidad para terminar cuando se lo indiquen:

```
void compute_factorial(std::stop_token stoken, int& factorial, int number) ②
{
    factorial = 1;
    for ( int i = 2; i <= number; i++ )
    {
        if(stoken.stop_requested()) return; ④
        factorial = factorial * i;
    }
}

int main()
{
    // ...

    int factorial;
    std::jthread thread(compute_factorial, std::ref(factorial), 122); ① ②

    // ...

    thread.request_stop(); ③

    // ...
}
```

① Crear e iniciar el hilo con std::jthread para calcular el factorial de 122.

② Al crear el hilo se pasa a la función el ***token*** **de cancelación** `stoken`.

③ En algún momento de la ejecución del programa pedimos al hilo que se detenga antes de terminar los cálculos.

④ El código del hilo debe comprobar el ***token*** **de cancelación** regularmente. Si se ha pedido la cancelación, se termina el hilo retornando desde la función principal.

Java y C# han terminado incluyendo también este tipo de **cancelación cooperativa** usando un ***token*** **de cancelación**, debido a los problemas que tienen los desarrolladores para recordar usar correctamente la excepción de la **cancelación coordinada**.

12.6. Otras consideraciones sobre los hilos

12.6.1. Las llamadas al sistema fork() y exec() en procesos multihilo

La llamada al sistema fork() de los sistemas POSIX es anterior a la existencia del concepto de **hilo**. Así que cuando estos aparecieron surgió el problema de si al llamar a fork() en un proceso multihilo:

- El nuevo proceso debía tener un duplicado de todos los hilos.
- O el nuevo proceso debía tener un único hilo copia del que invocó a fork().

Como hemos comentado anteriormente, la llamada al sistema fork() sustituye el programa en

ejecución con un nuevo programa e inicia su ejecución en main(). Esto incluye liberar toda la memoria reservada y la destrucción de todos los hilos del programa original, por lo que duplicar los hilos en el proceso hijo creado por fork(), si luego se va a llamar a exec() parece algo innecesario.

El estándar POSIX establece que si se utiliza fork() en un programa multihilo, el nuevo proceso debe ser creado con un solo hilo, que será una réplica del que hizo la llamada, así como un duplicado completo del espacio de direcciones del proceso.

Sin embargo, algunos sistemas UNIX tienen una segunda llamada no estándar denominada `forkall()`, capaz de duplicar todos los hilos del proceso padre. Obviamente solo resulta conveniente emplearla si no se va a utilizar la llamada exec() a continuación. La inclusión de `forkall()` en el estándar POSIX fue considerada y rechazada.

12.6.2. Manejo de señales en procesos multihilo

En el Apartado 10.5.2 hablamos del uso de las señales como mecanismo de comunicación, pero en general sirven para informar a un proceso del suceso de ciertos eventos.

Existen dos tipos de señales:

- Las **señales síncronas** se deben a alguna acción del propio proceso. Ejemplos de señales de este tipo son `SIGSEV` y `SIGFE`, originadas por accesos ilegales a memoria o divisiones por 0, respectivamente.

  Las señales síncronas son enviadas al mismo proceso que las origina.

- Las **señales asíncronas** son debidas a acciones externas. Un ejemplo de este tipo de señales es la terminación de procesos con teclas especiales como `CTRL` + `C` o `CTRL-D`, que envían al proceso las señales `SIGINT` y `SIGHUP` respectivamente. También lo son las señales enviadas desde otro proceso, como cuando el proceso **init** envía `SIGTERM` al resto de procesos para informales que deben terminar porque el sistema se va a apagar.

Como hemos visto, las señales que llegan a un proceso pueden ser interceptadas por una función definida por el programador llamada **manejador de señal**.

Las señales también son anteriores a los hilos, por lo que cuando aparecieron los hilos se tuvieron que tomar decisiones sobre cómo iban a encajar ambos conceptos. Por ejemplo, decidir cuál de los hilos del proceso, será interrumpido cuando llegue una señal, para ejecutar el manejador de señales.

Señales enviadas por otros hilos

En los sistemas POSIX multihilo se pueden enviar señales a un proceso:

```
kill(pid, SIGTERM);
```

En ese caso uno cualquiera de los hilos podrá ser interrumpido para ejecutar el manejador de señal.

Cada hilo puede enmascarar las señales que considere llamando a pthread_sigmask() Es decir, cada hilo puede elegir qué señales quiere bloquear para no tener que atenderlas:

```
sigset_t set;               ①

sigemptyset(&set);          ②
sigaddset(&set, SIGINT);    ③
sigaddset(&set, SIGUSR1);   ④

pthread_sigmask(SIG_BLOCK, &set, NULL); ⑤
```

① Las máscaras de señales se definen mediante *sets* de señales. El tipo de los *sets* de señales es `sigset_t`, de cuyo tipo real no deberíamos preocuparnos, por portabilidad.

② Para manipular los *sets* se proporcionan una serie de funciones. sigemptyset() es para asegurar que el *set* está vacío.

③ Añadimos al *set* la señal `SIGINT`.

④ Añadimos al *set* la señal `SIGINT`.

⑤ Bloqueamos en el hilo actual las señales en el *set* `set`,es decir, `SIGINT` y `SIGUSR1`.

Así una señal enviada a un proceso interrumpirá a uno de los hilos que no la haya bloqueado.

También se puede enviar una señal a un hilo en concreto usando pthread_kill(). El hilo será interrumpido si no la ha bloqueado.

```
pthread_kill(thread, SIGTERM);
```

Sin embargo, hay que tener en cuenta que el manejo de señales es un recurso del proceso, compartido por todos sus hilos. Esto quiere decir que si la señal está configurada para ser manejada usando la acción por defecto y dicha acción es terminar, terminará todo el proceso, aunque la señal haya sido dirigida a un hilo en concreto.

Señales enviadas por el sistema

Lo que queda por ver es a quién va dirigida una señal, cuando es el sistema quién la envía para notificar un evento:

- Las señales síncronas son causadas por un error en la ejecución, que en un proceso multihilo es debido a la fallida ejecución de un hilo en particular. Por eso estas señales se dirigen al hilo que las causa.
- Las señales asíncronas llegan por causas externas, así que se dirigen al proceso, pudiendo ser entregada a uno de los hilos que no la tenga bloqueada.

La recomendación es elegir un hilo para el manejo de señales asíncronas, de tal forma que sea el único que no las tenga bloqueadas. El resto de hilos deberían bloquear estas señales nada más iniciar su ejecución.

Si se destina un hilo para esta tarea en exclusiva, este puede utilizar sigwait() para bloquearse hasta que llegue una señal. Cuando eso ocurre, la función sigwait() retorna indicando el número de señal recibida —sin necesitar **manejadores de señal**—. Entonces el hilo puede solicitar la cancelación de los otros hilos para, por ejemplo, terminar el proceso.

Esta estrategia facilita el desarrollo de programas que manejen las señales adecuadamente. Al utilizar un hilo para manejar las señales, tenemos a nuestra disposición cualquier función **segura en hilos** y sabemos cómo proteger las variables y estructuras de datos compartidas mediante el uso de mecanismos de **sincromización** (véase el Capítulo 13). Mientras que al trabajar con **manejadores de señal** estamos mucho más limitados, porque el código de estos debe ser **reentrante** y solo pueden usar alguna de las pocas funciones de la librería del sistema marcadas como seguras en señales.

Chapter 13. Sincronización

Tiempo de lectura: 30 minutos

En el Capítulo 11 vimos que varios procesos pueden compartir regiones de la memoria con el objeto de cooperar en las tareas que deben desempeñar. Además, en el Capítulo 12 vimos que en los procesos multihilo todos los hilos comparten el espacio de direcciones del proceso al que pertenecen, lo que significa que pueden acceder al mismo tiempo a las variables globales y a la memoria reservada dinámicamente.

Ambas posibilidades introducen algunos riesgos, puesto que el acceso simultáneo a los datos compartidos puede ocasionar inconsistencias. Así que ha llegado el momento de discutir cómo se puede asegurar la ejecución ordenada de hilos o procesos cooperativos que comparten regiones de la memoria, con el fin de mantener la consistencia de los datos.

En este capítulo hablaremos de hilos y de procesos que comparten memoria indistintamente. En ambos casos el problema es el mismo y las soluciones similares.

13.1. El problema de las secciones críticas

Llamamos **condición de carrera** a la situación en la que varios procesos o hilos pueden acceder y manipular los mismos datos al mismo tiempo —es decir, de forma **concurrente**— y donde el resultado de la ejecución depende del orden particular en el que tienen lugar dichos accesos. Estas situaciones ocurren frecuentemente en los sistemas operativos, puesto que diferentes componentes del mismo manipulan los mismos recursos interfiriendo unos con otros.

13.1.1. Problema del productor-consumidor

Para ilustrarlo, veamos un problema clásico de concurrencia: el **problema del productor-consumidor**.

Supongamos que dos hilos o procesos comparten una región de la memoria que contiene un vector de elementos y un contador con el número de elementos del vector.

El primer hilo realiza varias tareas, que no entraremos a describir. Lo importante es que, como resultado de esas tareas, en ocasiones añade un elemento al vector e incrementa el contador que indica el número de elementos en el vector. Es decir, el primer hilo actúa como un **productor** de elementos del vector.

A continuación mostramos una porción de la función del productor:

```
while(/* ... */)
{
    item = produce_item();

    // Si el vector está lleno, esperar
```

```
    while (count == VECTOR_SIZE);

    // Añadir el elemento al vector
    vector[count] = item;
    ++count;
}
```

El segundo hilo también realiza varias tareas que no describiremos. Pero para realizar esas tareas en ocasiones debe tomar un elemento del vector compartido, decrementando el contador para indicar que ahora hay un elemento menos en el vector. Es decir, el segundo hilo actúa como un **consumidor** de elementos del vector.

A continuación mostramos una porción de la función del consumidor:

```
while(/* ... */)
{
    // Si el vector está vacio, esperar
    while (count == 0);

    // Extraer un elemento del vector
    --count;
    item = vector[count];

    consume_item(item);
}
```

Aunque el **problema del productor-consumidor** parezca artificial, lo cierto es que es muy común.

Por ejemplo, en una herramienta de grabación de audio, el **productor** es un hilo dedicado a obtener bloques de muestras grabadas a través de la API multimedia del sistema operativo. Mientras tanto, otro hilo puede dedicarse a tomar las muestras y realizar diversas transformaciones, como: reducir el ruido, mezclar con otras fuentes de sonido o aplicar algún tipo de efecto digital. Este segundo hilo es el **consumidor**. La manera de conectar ambos es tener un vector compartido, donde se depositan los bloques de muestras cuando llegan y de dónde se extraen para su tratamiento. Así, ambos hilos pueden trabajar a su propio ritmo, de forma casi independiente.

Aunque el código anterior del productor y del consumidor es correcto cuando no coinciden en el tiempo al ejecutarse, no funciona adecuadamente cuando sí lo hacen. El motivo es que los dos hilos comparten la variable `count` y tanto las sentencias `++count` y como `--count` pueden interrumpirse a medias para dejar paso a la ejecución del otro hilo.

Por ejemplo, `++count` podría dividirse por el compilador en las siguientes operaciones, al generar las instrucciones del procesador:

++count

```
registro1 = count;
registro1 = registro1 + 1;
count = registro1;
```

Donde `registro1` representa un registro de la CPU. De forma parecida la sentencia `--count` podría ser implementada de la siguiente manera:

--count

```
registro2 = count;
registro2 = registro2 - 1;
count = registro2;
```

Donde nuevamente `registro2` representa un registro de la CPU. Realmente, aunque `registro1` y `registro2` pueden ser el mismo registro físico, el contenido de los registros se guarda y se recupera durante los cambios de contexto de un hilo al otro, por lo que cada uno ve sus propios valores y no los del otro.

El que las sentencias `++count` y `--count` se ejecute de forma concurrente, es similar a que las instrucciones de lenguaje máquina de ambas sentencias en ambos hilos o procesos se entrelacen en algún orden aleatorio.

Un posible entrelazado de las instrucciones en lenguaje máquina entre hilos, suponiendo que inicialmente `count = 5`, podría ser el siguiente:

```
// Entra ++count
registro1 = count;          // registro1 = 5
registro1 = registro1 + 1;  // registro1 = 6
// Sale ++count y entra --count
registro2 = count;          // registro2 = 5
registro2 = registro2 - 1;  // registro2 = 4
// Sale --count y entra ++count
count = registro1;          // count = 6 ②
// Entra --count
count = registro2;          // count = 4 ① ②
```

① Así llegamos al resultado incorrecto `count = 4`, indicando que hay 4 elementos en el vector cuando realmente hay 5.

② Si invertimos el orden de las sentencias obtendremos el resultado, también incorrecto, `count = 6`.

Como se puede apreciar, hemos llegado a estos valores incorrectos porque hemos permitido la manipulación concurrente de la variable `count`. Según como se entrelacen las instrucciones de `++count` y `--count` en la CPU, el resultado final podría ser: 4, 5 o 6. Pero el único resultado correcto es 5, que es el que obtendremos si ejecutamos las sentencias secuencialmente, sin mezclar las operaciones en las que se dividen.

Ambos hilos se ejecutan de forma concurrente porque o bien, tenemos un sistema multiprocesador o multinúcleo, donde ambos hilos se ejecutan a la vez en procesadores diferentes, o bien, porque tenemos un sistema operativo donde uno de los hilos puede ver interrumpida su ejecución en cualquier momento para asignar la CPU a otro (véase el Apartado 14.1.

13.1.2. Manipular estructuras de datos

Obviamente, el problema comentado no aparece solo en sentencias simples, sino también en bloques de código destinados a hacer tareas complejas, como manipular estructuras de datos.

Por ejemplo, supongamos que `vector` no es un simple *array* de elementos, sino una lista enlazada, de tal forma que ahora extraer un elemento sería así:

```
--count;
item = vector.extract(count);
```

y el método `extract()` tendría que dar los siguientes pasos:

1. Iterar sobre la lista para buscar el nodo en la posición `count`.
2. Al encontrarlo, preservar en variables locales el puntero a ese nodo y al previo.
3. Cambiar en el nodo previo el puntero al siguiente nodo, para que apunte al nodo tras el que queremos extraer. En este momento el nodo a extraer ya no pertenece a la lista enlazada.
4. Extrae el `item` del campo que lo contiene en el nodo.
5. Destruir el nodo.
6. Salir del método retornando el elemento.

Esto genera varios momentos cruciales en torno al paso 3, que puedan dar lugar a **condiciones de carrera**. Por ejemplo, si el hilo es interrumpido tras guardar el puntero al nodo en una variable local y llega otro hilo que extrae —y destruye— antes ese mismo nodo, el puntero ya no es válido —es un *dangling pointer* o referencia colgante— al continuar la ejecución del primer hilo. Y lo mismo ocurre con el puntero al nodo previo o al siguiente, si el hilo es interrumpido y otro hilo destruye antes alguno de ellos.

Los problemas que esto puede causar son diversos, según el momento exacto en el que ocurra. Puede haberlos al intentar leer el elemento guardado en el nodo en el paso 4, porque este último ya no exista. También, al intentar actualizar, en el paso 3, el puntero al siguiente nodo en el nodo previo, porque el nodo previo no exista. Incluso puede que la función termine con aparente normalidad, pero dejando que el nodo previo apunte a un nodo siguiente que no existe. En este último supuesto, la lista quedaría en estado inconsistente y así el problema se lo encontraría el próximo hilo que intente usarla.

13.1.3. Exclusión mutua

Para evitar que estas situaciones lleven a la corrupción de datos y a caídas de servicios y sistemas, debemos asegurarnos que solo un hilo en cada momento puede manipular recursos y variables

compartidas. Por tanto, necesitamos algún tipo de mecanismo de sincronización para que mientras se ejecuta `++count` no se pueda ejecutar `--count` en otro hilo, ni viceversa. O para que mientras un hilo haga un `insert()` o un `extract()` en una lista, otro no pueda utilizar ni estas ni otras funciones de la misma clase.

Para resolver esto, debemos empezar buscando las **secciones críticas** de nuestro código. Una **sección crítica** es una porción del código donde se accede a variables, tablas, listas, archivos y otros recursos compartidos.

Para evitar **condiciones de carrera**, el acceso a las **secciones críticas** debe ser controlado, de manera que cuando un hilo se esté ejecutando en una sección de este tipo ningún otro pueda hacerlo en la suya correspondiente para manipular los mismos recursos. En estos casos se dice que existe **exclusión mutua** entre las **secciones críticas**.

13.1.4. Eventos

Las **condiciones de carrera** son el principal problema del código anterior del productor y el consumidor, pero no el único. En ambos ejemplos se usan bucles para que el hilo espere si el vector está lleno o vacío, antes de continuar. A esta técnica se la denomina **espera ocupada** o **espera activa** y está completamente desaconsejada usarla en código del espacio de usuario, porque contribuye a gastar tiempo de CPU inútilmente.

En su lugar, se recomienda usar mecanismos de sincronización ofrecidos por el sistema operativo; diseñados para que un hilo o proceso notifique eventos a otro, de tal forma que hasta que eso ocurre, el que espera permanezca en estado **esperando**, dejando la CPU para los hilos que la necesitan.

13.2. Sincronización por hardware

Las soluciones ofrecidas por el sistema operativo, para resolver los problemas anteriores, suelen tener que apoyarse en características del hardware. A continuación veremos algunas de esas características, antes de profundizar en los mecanismos ofrecidos por el sistema operativo.

13.2.1. Bloque de las interrupciones

El problema de la sección crítica puede ser resuelto de forma sencilla en un sistema monoprocesador.

Como el núcleo del sistema operativo es un software controlado mediante interrupciones, basta con que los hilos bloqueen las interrupciones mientras se está dentro de la sección crítica. Así, el sistema operativo no puede tomar el control y asignar otro hilo a la CPU, lo que impide que se ejecute otra secuencia de instrucciones que podría modificar los datos compartidos.

Indudablemente esta solución no es práctica en sistema multiprocesador, donde hay varios procesadores ejecutándose a la vez.

13.2.2. Instrucciones atómicas

Todas las CPU modernas disponen de instrucciones para comparar y modificar el contenido de una

variable o intercambiar el contenido de dos variables, de forma **atómica**. El término **atómico** hace referencia a que las operaciones se ejecutan como una unidad ininterrumpible. No importa que varias CPU ejecuten estas instrucciones simultáneamente, puesto el hardware se encargará de que sean ejecutadas secuencialmente en algún orden arbitrario.

Estas instrucciones están disponibles para los programadores de C y C++ a través de tipos especiales. Por ejemplo, en C11 <stdatomic.h> define tipos como: `atomic_bool`, `atomic_uint` o `atomic_char` para declarar variables atómicas de los tipos `bool`, `unsigned int` y `char`, respectivamente. También declara funciones para inicializar, leer, guardar, intercambiar, sumar, restar y realizar operaciones lógicas, de forma atómica sobre estas variables:

```
atomic_int count;
atomic_init(&count, 0); ①
int old_count = atomic_fetch_add(&count, 1); ②
```

① Inicializar el valor de la variable atómica.

② Como un `count++` atómico: incrementa la variable devolviendo el valor previo.

En C++11, <atomic> declara la plantilla std::atomic que ofrece una funcionada similar:

```
std::atomic<int> count {0}; ① ②
int old_count = count++;    ③
```

① También hay un tipo `std::atomic_int` que es equivalente.

② Se usa el constructor para inicializar la variable atómica.

③ Además de soportar los operadores '++' y '--', soporta los métodos std::atomic::fetch_add() y std::atomic::fetch_sub() para sumar y restar devolviendo el valor previo.

La importancia de estas instrucciones está en que pueden ser utilizadas por el sistema operativo para ofrecer soluciones sencillas al problema de la sección crítica. Por ejemplo, **semáforos** o ***mutex***.

13.3. Semáforos

La exclusión mutua en las secciones críticas se asegura utilizando adecuadamente una serie de recursos que para ese fin proporciona el sistema operativo. Estos recursos utilizan internamente instrucciones y otras características de la CPU, incluidas por los diseñadores para resolver este tipo de problemas, que hemos comentado anteriormente. Ese es el caso de los **semáforos**.

Los **semáforos** son un tipo de objetos del sistema operativo que nos permiten controlar el acceso a una sección crítica, por medio de dos primitivas: **acquire** y **release** —o **wait** y **signal**, según el libro de texto que consultemos—. A continuación describimos el mecanismo de funcionamiento:

```
semaphore S(10);    ①

S.acquire()         ②
```

```
// Código de la sección crítica... ③

S.signal();          ④
```

① Crear el **semáforo** `S` inicializado a 10. Un **semáforo** contiene fundamentalmente un contador con el número máximo de hilos que pueden estar ejecutando el código de la sección crítica al mismo tiempo.

② Intentar entrar en la sección crítica:

- Si el contador interno del **semáforo** es mayor que 0, `acquire()` lo decrementa y retorna para que la ejecución continúe.
- Si el contador interno del **semáforo** es igual a 0, `acquire()` saca al hilo de la CPU y lo pone en una cola de espera, suspendiendo así su ejecución. Básicamente, es que hay demasiados hilos dentro de la sección crítica.

③ Aquí iría el código protegido con el **semáforo**. Es decir, el código de la sección crítica en sí.

④ Salir de la sección crítica:

- Si el contador interno del **semáforo** es mayor que 0, `release()` lo incrementa y retorna para que la ejecución continúe.
- Si el contador interno del **semáforo** es igual a 0, `release()` lo incrementa y saca a uno de los hilos en la cola de espera —donde los puso su `acquire()`— para meterlo en la cola de preparados, dejándolo listo para entrar en la CPU. Cuando eso ocurra, ese hilo decrementará el contador interno del **semáforo** y saldrá de su `acquire()`, donde hasta ahora estaba atrapado. Mientras tanto `release()` retorna y la ejecución del hilo que sale del sección crítica continúa.

Para que funcione correctamente, el **semáforo** `S` debe ser el mismo para todos los hilos que tengan secciones críticas en cuya ejecución debe haber **exclusión mutua**. Es decir, el **semáforo** `S` debe estar compartido entre los hilos de la misma manera que las estructuras de datos, variables y otros recursos que protege.

13.3.1. Tipos de semáforos

Tanto el estándar POSIX como Windows API soportan semáforos y ambos admiten dos tipos de semáforos:

- Los **semáforos anónimos** que solo existen en el espacio de direcciones del proceso que los crea, de tal forma que están disponibles para sincronizar hilos del mismo proceso.

  La forma de usarlos para sincronizar procesos diferentes o hilos en procesos diferentes depende del sistema operativo. Con Windows API se pueden heredar de padres a hijos. Mientras que en sistemas POSIX es necesario crear el **semáforo** en una región de **memoria compartida**, que hayamos creado previamente, e indicar un valor distinto de 0 en el argumento `pshared` de sem_init().

- Las **semáforos con nombre** son públicos al resto del sistema, por lo que teóricamente cualquier proceso con permisos puede abrirlos para utilizarlos.

Tabla 4. Funciones de la API para manipular semáforos.

	POSIX API	Windows API
Crear semáforo anónimo	sem_init()	CreateSemaphore()
Crear semáforo con nombre	sem_open()	CreateSemaphore()
Abrir semáforo con nombre	sem_open()	OpenSemaphore()
Operación acquire	sem_wait()	WaitForSingleObject()
Operación release	sem_post()	ReleaseSemaphore()
Cerrar semáforo con nombre	sem_close()	CloseHandle()
Destruir semáforo anónimo	sem_destroy()	[Automático]
Destruir semáforo con nombre	sem_unlink()	[Automático]

13.3.2. Ejemplos del uso de semáforos

En el ejemplo anom-shared-memory.cpp de comunicación mediante memoria compartida, se usa un **semáforo** para que el proceso hijo indique al proceso padre que ha terminado de calcular el factorial y el resultado ya está en la memoria.

En shared-memory-server.c está el ejemplo completo de un programa que muestra periódicamente la hora del sistema y que puede ser controlado remotamente, mediante memoria compartida, con un cliente como el de shared-memory-client.cpp.

Para enviar los mensajes entre el cliente y el servidor, en la memoria compartida se reserva hueco para un búfer en el que el cliente copia el comando que quiere enviar y para dos **semáforos**:

```
struct memory_content
{
    sem_t empty; ①
    sem_t ready; ②
    char command_buffer[MAX_COMMAND_SIZE];
};
```

① Indica cuándo `command_buffer` está vacío, así que se inicializa a 1. El cliente usa sem_wait() en este **semáforo** antes de escribir un nuevo comando en `command_buffer`:

- Si el **semáforo** está a 0, el cliente pasa y escribe el comando. Después llama a sem_post() en `ready`.
- Si el **semáforo** está a 1, el cliente queda bloqueado y tiene que esperar a que el servidor use sem_post() sobre el mismo **semáforo**. El servidor lo hace después de leer el comando para interpretarlo.

② Indica cuándo `command_buffer` tiene un comando, así que se inicializa a 0. El servidor usa sem_wait() en este **semaforo** antes de leer el comando en `command_buffer` para interpretarlo:

- Si el **semáforo** está a 0, el cliente pasa y lee el comando. Después llama a sem_post() en `empty`.

- Si el **semáforo** está a 1, el servidor queda bloqueado y tiene que esperar a que el cliente use sem_post() sobre el mismo **semáforo**. El cliente lo hace después de escribir un nuevo comando en `command_buffer`.

El detalle de cómo cliente y servidor usan ambos semáforos, se puede ver en el código de shared-memory-client.cpp y shared-memory-server.c, respectivamente.

Finalmente, para resolver el **problema del productor-consumidor** tenemos que considerar que:

- Necesitamos un semáforo para que haya **exclusión mutua** entre ambos al insertar y extraer elementos del vector.
- Necesitamos una forma de que el productor espere cuando el vector esté lleno y que el consumidor haga lo mismo cuando el vector esté vacío. Una solución es usar dos semáforos, uno para que cuente el número de elementos en el vector y otro para contar el número de huecos libres:

```
sem_t mutex;       ①
sem_t fill_count;  ②
sem_t empty_count; ③

std::vector<item_t> buffer( VECTOR_SIZE );

void initialize()
{
    sem_init( &mutex, 0, 1 ); ① ④
    sem_init( &mutex, 0, 0 ); ② ④
    sem_init( &mutex, 0, VECTOR_SIZE ); ③ ④
}

void productor()
{
    // ...

    while(/* ... */)
    {
        item_t item = produce_item();

        sem_wait( &empty_count ); ⑤
        sem_wait( &mutex );       ⑦

        vector.push_back(item);

        sem_post( &mutex );       ⑦
        sem_post( &fill_count );  ⑥
    }

    // ...
}

void consumer()
```

```
{
    // ...

    while(/* ... */)
    {
        sem_wait( &fill_count );  ⑥
        sem_wait( &mutex );       ⑦

        item_t item = vector.back();
        vector.pop_back();

        sem_post( &mutex );       ⑦
        sem_post( &empty_count ); ⑤

        consume_item(item);
    }

    // ...
}
```

① **Semáforo** que se encarga de la exclusión mutua. Se inicializa a 1, para que el primer hilo que use sem_wait() pueda entrar.

② **Semáforo** que se encarga de contar huecos ocupados en el vector. Se inicializa a 0, porque al principio no hay ningún elemento.

③ **Semáforo** que se encarga de contar los huecos libres en el vector. Se inicializa a VECTOR_SIZE, porque están todos vacíos.

④ El segundo argumento de sem_init() es `pshared`. Se pone a 0 para indicar que este **semaforo** no se va a compartir entre procesos diferentes.

⑤ Antes de insertar un elemento se decrementa `empty_count`. Así, si vale 0, es que el vector está lleno y el productor se bloquea. El consumidor incrementa `empty_count` tras extraer un elemento y dejar hueco, despertando al productor.

⑥ Antes de extraer un elemento se decrementa `fill_count`. Así, si vale 0, es que el vector está vacío y el consumidor se bloquea. El productor incrementa `fill_count` tras insertar un elemento nuevo, despertando al consumidor.

⑦ El acceso al vector con los elementos es en **exclusión mutua**, así que tanto productor como consumidor deben usar sem_wait() sobre `mutex` antes de acceder a él. Esto decrementa el semáforo, así que solo uno de los dos pasa y ejecuta las líneas siguientes, mientras el otro queda bloqueado. Cuando el que haya pasado termine, debe usar sem_post() sobre `mutex` para incrementar el semáforo y permitir que el otro hilo entre en su **sección crítica**.

13.4. Mutex

Los semáforos inicializados a 1 se denominan ***mutex*** o **semáforos binarios**. Por tanto, aunque un sistema o lenguaje solo soporte **semáforos**, es directo implementar ***mutex***. A la inversa ocurre igual. Si un sistema o lenguaje soporta ***mutex***, es muy sencillo hacer una implementación **semáforos**, si nos hiciera falta.

El estándar POSIX soporta ***mutex*** a través de POSIX Threads. Por defecto solo se pueden utilizar para sincronizar hilos del mismo proceso; pero tienen un atributo para permitir la sincronización entre procesos diferentes, aunque para eso deben ser creados en una región de memoria compartida por dichos procesos.

La API de POSIX Threads no soporta **semáforos** porque, como vimos antes, ya eran parte del estándar POSIX.

Tabla 5. Funciones de la API para manipular mutex.

	C++	POSIX API	Windows API	
Crear	std::mutex	pthread_mutex_init()	InitializeCriticalSection()	CreateMutex()
Abrir				OpenMutex()
Operación acquire	std::mutex:::lock()	pthread_mutex_lock()	EnterCriticalSection()	WaitForSingleObject()
Operación release	std::mutex::unlock()	pthread_mutex_unlock()	LeaveCriticalSection()	ReleaseMutex()
Destruir	[Destructor]	pthread_mutex_destroy()	DeleteCriticalSection()	CloseHandle()

En Windows API hay dos tipo de objetos equiparables a los ***mutex***: los ***mutex*** y las **secciones críticas**. Las **secciones críticas** son más ligeras, pero solo se pueden utilizar para sincronizar hilos del mismo proceso. Mientras que los ***mutex*** de Windows API son objetos más costosos, pero se pueden compartir entre procesos sin utilizar memoria compartida; ya sea mediante herencia al crear un proceso hijo o asignando un nombre al ***mutex***, como ocurre con los **semáforos**.

13.4.1. Ejemplos del uso de mutex

En pthreads-sync-factorial.cpp se puede estudiar el código completo de un ejemplo similar a pthreads.cpp, donde se calculaba el factorial de un número, repartiendo la tarea entre dos hilos, usando la API de POSIX Threads. La diferencia es que ahora los hilos no retornan el resultado, sino que cada uno lo mete en un vector compartido. Al terminar, el hilo principal recorre el vector multiplicando los resultados parciales.

Como ahora ambos hilos acceden a una estructura de datos compartida, esta debe ir protegida por un ***mutex***:

```
pthread_mutex_t mutex;
std::vector<BigInt> partials;
```

Antes de meter un nuevo valor, cada hilo debe adquirir el `mutex`:

```
// Bloquear el mutex y guardar el resultado
pthread_mutex_lock( mutex );   ①
partials.push_back( result );
```

```
pthread_mutex_unlock( mutex ); ②
```

① Adquirir `mutex` antes de entrar en la **sección crítica**.

② Liberar `mutex` para salir de la **sección crítica**. Es importante no olvidarnos de liberar el ***mutex*** al terminar o de lo contrario uno de los hilos quedará dormido indefinidamente, al no poder entrar en la **sección crítica**.

En pthreads-sync-counter.cpp se puede observar un ejemplo más simple donde dos hilos intentan incrementar un contador. El resultado es correcto siempre que cada hilo adquiera el mismo ***mutex*** antes del incremento:

```
// Bloquear el mutex antes de incrementar el contador.
pthread_mutex_lock( &args->mutex );
args->counter++;
// Desbloquear el mutex tras incrementar el contador.
pthread_mutex_unlock( &args->mutex );
```

En threads-sync-counter.cpp y threads-sync-factorial.cpp se puede ver ejemplos equivalentes pero usando std::thread y std::mutex, de la librería estándar de C++.

13.5. Variables de condición

En la solución que dimos al **problema del productor-consumidor** usando **semáforos** (véase el Apartado 13.3.2) empleamos **semáforos** para implementar las esperas del productor y el consumidor cuando el vector está lleno o vacío, respectivamente. Lamentablemente, los *mutex* no se pueden usar de la misma manera para señalar eventos. En su lugar necesitamos otro tipo de objeto llamado **variable de condición**.

Las **variables de condición** soportan tres primitivas principales:

wait(mutex)

Es llamada por un hilo que desea esperar a que ocurra el evento que representa la variable de condición. El hilo debe haber adquirido antes el ***mutex***, es liberado en el momento de poner al hilo en estado **esperando**. Varios hilos pueden llamar a **wait** sobre la misma variable de condición, a la espera de que alguno use **notify**.

notify

Es llamada por un hilo que quiere notificar el suceso de un evento a los hilos que esperan en la variable de condición. Uno de esos hilos es despertado, adquiere el ***mutex*** que liberó al llamar a **wait** y, finalmente, retorna de **wait** para seguir ejecutándose.

notifyAll

Es llamada por un hilo que quiere notificar el suceso de un evento a los hilos que esperan en la variable de condición. Todos los hilos son despertados e intentan adquirir el ***mutex*** que liberaron al llamar a **wait**. Cuando lo consiguen, retornan de **wait** para seguir ejecutándose. Obviamente, si todos hicieron **wait** sobre el mismo ***mutex***, irán retornando de **wait** de uno en uno, porque solo un hilo puede tener el ***mutex*** al mismo tiempo.

Tanto Windows API como el estándar POSIX, a través de POSIX Threads, soportan **variables de condición**.

Tabla 6. Funciones de la API para manipular variables de condición.

	C++	POSIX API	Windows API
Crear	std::condition_variable	pthread_cond_init()	InitializeConditionVariable()
Operación wait	std::condition_variable::wait()	pthread_cond_wait()	SleepConditionVariableCS()
Operación notify	std::condition_variable::notify_one()	pthread_cond_signal()	WakeConditionVariable()
Operación notifyAll	std::condition_variable::notify_all()	pthread_cond_broadcast()	WakeAllConditionVariable()
Destruir	[Destructor]	pthread_cond_destroy()	

Por defecto, las **variables de condición** de POSIX Threads solo se pueden utilizar para sincronizar hilos del mismo proceso; pero tienen un atributo para permitir la sincronización entre procesos diferentes. Obviamente, para eso deben ser creadas en una región de memoria compartida por dichos procesos.

Las **variables de condición** de Windows API solo se pueden utilizar en hilos del mismo procesos. Como alternativa, Windows API soporta **eventos**, que son un tipo de objeto similar a las **variables de condición**, pero que sí se puede utilizar entre hilos de procesos diferentes (véase «Using Event Objects — Microsoft Docs»).

A los **eventos** se les puede asignar un nombre, para que sean accesibles por otros procesos, o heredarse de padres a hijos. Además son más pesados que las **variables de condición** de Windows API, no exigen un ***mutex*** para liberar al invocar su operación **wait**, ni admiten la operación **notifyAll**.

Tabla 7. Funciones de la API para manipular eventos de Windows API.

	Windows API
Crear	CreateEvent()
Abrir	OpenEvent()
Operación wait	WaitForSingleObject()
Operación notify	SetEvent()
Resetear evento	ResetEvent()
Destruir	CloseHandle()

13.5.1. Ejemplos del uso de variables de condición

Vamos a intentar resolver el **problema del productor-consumidor** sin usar **semáforos**. Para lo que, nuevamente, tenemos que considerar que:

- Necesitamos **exclusión mutua** entre ambos hilos al insertar y extraer elementos del vector para evitar **condiciones de carrera**, por tanto usamos un ***mutex*** para proteger la **sección crítica**.
- Necesitamos una forma de que el productor espere cuando el vector está lleno y que el consumidor haga lo mismo cuando el vector está vacío. Para señalar estos eventos necesitamos dos **variables de condición**:

```
std::mutext mutex; ①
std::condition_variable no_full;  ②
std::condition_variable no_empty; ③

std::vector<item_t> buffer( VECTOR_SIZE );

void productor()
{
    // ...

    while(/* ... */)
    {
        item_t item = produce_item();

        std::unique_lock lock{ mutex }; ④ ⑤

        while( buffer.size() == VECTOR_SIZE ) ⑥ ⑩
        {
            no_full.wait( mutex ); ⑪
        }

        vector.push_back( item );  ⑪

        no_empty.notify_one(); ⑦
    } ⑥

    // ...
}

void consumer()
{
    // ...

    while(/* ... */)
    {
        std::unique_lock lock{ mutex }; ④ ⑤

        while( buffer.size() == 0 ) ⑧ ⑩
        {
            no_empty.wait( mutex );  ⑪
        }

        item_t item = vector.back(); ⑪
        vector.pop_back();
```

```
        no_full.notify_one(); ⑨

        consume_item(item);
    } ⑥

    // ...
}
```

① ***Mutex*** que se encarga de la exclusión mutua.

② **Variable de condición** que se encarga de indicar cuando el vector no está lleno.

③ **Variable de condición** que se encarga de indicar cuando el vector no está vacío.

④ Antes de acceder al vector es necesario bloquear `mutex`. Ni siquiera es seguro preguntar por el número de elementos guardados en `vector` sin antes adquirir el ***mutex***, puesto que el otro hilo puede estar modificando `vector` al mismo tiempo.

⑤ Los ***mutex*** se pueden adquirir y liberar con std::mutex:::lock() y cpp_mutex_unlock}, pero esa no es la forma recomendada. Lo recomendado es crear alguno de los objetos *lock* incluidos en la librería estándar. Estos objetos bloquean el ***mutex*** al crearse y lo desbloquean automáticamente al destruirse. Así es complicado que nos olvidemos de desbloquearlo al salir de la función.

⑥ Antes de insertar un elemento se comprueba si hay algún hueco disponible. Si no lo hay, se pone el hilo a la espera en la **variable de condición** `no_full`.

⑦ El consumidor despierta al productor de esa espera tras extraer un elemento, porque es seguro que al hacerlo habrá dejado un hueco.

⑧ Antes de extraer un elemento se comprueba si hay alguno en el vector. Si no lo hay, se pone el hilo a la espera en la **variable de condición** `no_empty`.

⑨ El productor despierta al consumidor de esta espera tras insertar un nuevo elemento.

⑩ El estándar de C++ indica que las esperas en las **variables de condición** son susceptibles de despertar de forma espuria. Es decir, que el hilo puede salir de std::condition_variable::wait() sin que haya habido notificación. Por eso hay que volver a comprobar la condición antes de continuar ejecutando sentencias en la **sección crítica**.

⑪ Cuando los hilos se bloquean en std::condition_variable::wait(), `mutex` es liberado para que el otro hilo pueda entrar y extraer o insertar un elemento. De lo contrario, no podría hacerlo y ambos se quedarían bloqueados indefinidamente —en una situación que se denomina **interbloqueo** o ***deadlock***—. Pero antes de salir de std::condition_variable::wait(), el hilo adquiere de nuevo el `mutex`. Así que el código que inserta y extrae elementos se ejecuta en **exclusión mutua**, tal y como nos interesa.

13.6. Esperas

Muchos de los objetos de sincronización que hemos visto necesitan algún mecanismo para poner en espera a los hilos que los usan. Existen dos alternativas desde el punto de vista de cómo implementar esta espera:

- El sistema operativo puede cambiar el estado del hilo o proceso a **esperado** y moverlo a una

cola de espera asociada al objeto de sincronización, tal y como hemos comentado en varias ocasiones. Entonces el planificador de la CPU escogerá a otro proceso para ser ejecutado.

- El hilo puede iterar comprobando constantemente la condición hasta que se cumple. A esa técnica se la denomina **espera ocupada**.

Este tipo de **espera ocupada** desperdicia tiempo de CPU que otro hilo podría utilizar de forma más productiva, por lo que solo se utiliza en el caso de esperas previsiblemente cortas. Para evitar que las esperas ocupadas sean demasiado largas, los sistemas operativos nunca expulsan de la CPU a hilos que se estén ejecutando dentro de secciones críticas controladas por objetos de sincronización con este tipo de espera, con la idea de que salgan de la sección crítica lo antes posible.

En Windows API, por ejemplo, se puede utilizar InitializeCriticalSectionAndSpinCount() para inicializar un objeto de **sección crítica** donde el hilo que la intenta adquirir itera el número especificado de veces en una **espera ocupada**, comprobando si la sección es liberada, antes de bloquearse en el estado **esperando** si eso no ocurre.

La **espera ocupada** de estos objetos **sección crítica** de Windows API solo ocurre en sistemas multiprocesador, donde el hilo que tiene adquirida la sección puede estar ejecutándose en otro hilo y terminar rápidamente. En sistemas monoprocesador nunca hay **espera ocupada**.

A los ***mutex*** con **espera ocupada** también se los denomina ***(spinlock)***. Los ***spinlocks*** son utilizados frecuentemente para proteger las estructuras del núcleo en los sistemas multiprocesador, cuando la tarea a realizar dentro de la sección crítica en el núcleo requiere poco tiempo y los diseñadores calculan que se desperdicia más tiempo sacando de la CPU al hilo en espera para ejecutar otro en su lugar.

13.7. Funciones reentrantes y seguras en hilos

Todas estas cuestiones sobre la sincronización no solo afectan al código que escribimos sino también a las librerías que podemos utilizar. A la hora de decidir utilizar una librería en un programa multihilo es necesario que tengamos en cuenta los conceptos de **reentrante** y **seguridad de hilos**.

13.7.1. Funciones reentrantes

Una función es **reentrante** puede ser interrumpida en medio de su ejecución y, mientras espera, volver a ser llamada con total seguridad. Obviamente las funciones recursivas deben ser reentrantes para poder llamarse a sí mismas una y otra vez con seguridad.

En el contexto de la programación multihilo, ocurre una reentrada cuando durante la ejecución de una función por parte de un hilo, este es interrumpido por el sistema operativo para planificar posteriormente a otro del mismo proceso que invoca la misma función.

En general una función es reentrante, si:

- No modifica variables estáticas o globales. Si lo hiciera solo puede hacerlo mediante operaciones **leer-modificar-escribir** que sean ininterrumpibles —es decir, atómicas—.

- No modifica su propio código y no llama a otras funciones que no sean reentrantes.

Como hemos mencionado anteriormente, los **manejadores de señal** deben ser funciones **reentrantes**.

13.7.2. Seguridad en hilos

Una función es **segura en hilos** o **thread-safe** si al manipular estructuras compartidas de datos lo hace de tal manera que se garantiza la ejecución segura de la misma por múltiples hilos al mismo tiempo. Obviamente estamos hablando de un problema de secciones críticas, por lo que las funciones lo resuelven sincronizando el acceso a estos datos mediante el uso de **semáforos**, ***mutex*** u otros recursos similares ofrecidos por el sistema operativo.

En ocasiones, ambos conceptos se confunden porque es bastante común que el código reentrante también sea seguro en hilos. Sin embargo es posible crear código reentrante que no sea seguro en hilos y viceversa.

A la hora de usar una función o librería que va a ser llamada desde múltiples hilos, primero debemos consultar la documentación para averiguar si es **segura en hilos**. Si no lo fuera, tendríamos que buscar funciones alternativas o recordar proteger las llamadas a las funciones no seguras con mecanismos de sincronización, para asegurar que solo son invocadas desde un hilo al mismo tiempo.

Esto se aplica tanto a librerías de otros desarrolladores como a la librería estándar del lenguaje que estemos usando y a la librería del sistema.

Seguridad en hilos en C++

La norma general es que las clases de la librería estándar de C++ son seguras frente a múltiples accesos de lectura desde diferentes hilos. Pero si un hilo modifica un objeto, todas las lecturas y escrituras en el mismo objeto por ese y otros hilos deben estar protegidas.

Obviamente, las clases de mecanismos de sincronización y gestión de hilos generalmente ofrecen mayores garantías, para lo que hay que consultar la documentación.

Seguridad en hilos en C

El estándar de C no menciona nada sobre **seguridad en hilos**, por lo que se debe suponer que las funciones de la librería estándar no lo son o consultar la documentación ofrecida por el proveedor de la librería.

Por ejemplo, todas las versiones de la librería estándar de C en Windows actualmente son **seguras en hilos** (véase «Multithreading with C and Win32 — Microsoft Docs»). Pero hasta hace unos años Microsoft ofrecía varias versiones de la librería, algunas **seguras en hilos**, para usar en aplicaciones multihilo, y otras no seguras para usar en aplicaciones monohilo.

Seguridad en hilos en POSIX

La API POSIX es un superconjunto de la API de la librería estándar de C. Por lo que en esos sistemas el estándar POSIX es el que marca qué funciones de la librería estándar de C y del resto de la API

POSIX son **seguras en hilos**.

En los sistemas POSIX, el estándar establece que todas las funciones son seguras excepto algunas muy concretas, que se pueden consultar en el apartado «2.9.1 Thread-Safety» de la especificación. Muchas de esas funciones no se especifican como **seguras en hilos** porque existe alguna alternativa que sí lo es. Por ejemplo, strerror() no es **segura en hilos**, pero strerror_r() tiene una funcionalidad equivalente y sí lo es.

Chapter 14. Planificación de la CPU

Tiempo de lectura: 56 minutos

El **planificador de la CPU** o **planificador de corto plazo** tiene la misión de seleccionar de la **cola de preparados** el siguiente proceso o hilo de núcleo a ejecutar. En dicha cola suelen estar los PCB —o TCB— de todos los procesos —o hilos de núcleo— que esperan una oportunidad para usar la CPU. Aunque se suele pensar en la **cola de preparados** como una cola FIFO, no tiene por qué ser así, como veremos más adelante, ya que existen mejores estrategias para seleccionar la próxima tarea a ejecutar.

En este capítulo hablaremos de procesos y de cómo son seleccionados por el planificador de la CPU. Sin embargo, debemos tener en cuenta que en los sistemas operativos multihilo con la librería de hilos implementada en el núcleo —categoría a la que pertenecen todos los sistemas modernos— la unidad de trabajo de la CPU es el hilo. Así que todo lo que comentemos a partir de ahora sobre la planificación de procesos en la CPU, realmente se aplica a los hilos y no a los procesos en los sistemas operativos modernos.

En cualquier caso, sea cual sea el algoritmo de planificación utilizado, este debe ser muy rápido, ya que es ejecutado con mucha frecuencia —aproximadamente una vez cada 100 milisegundos—.

14.1. Planificación expropiativa

El planificador deben ser invocado necesariamente en los siguientes casos, dado que en ellos la CPU queda libre y es conveniente aprovecharla planificando otro proceso, en lugar de dejarla desocupada:

1. Cuando un proceso pasa de **ejecutando** a **esperando**. Por ejemplo, por solicitar una operación de E/S, esperar a que un hijo termine, esperar en un semáforo, etc.
2. Cuando un proceso termina.

Cuando el planificador de la CPU es invocado solo en los casos anteriores, decimos que tenemos un sistema operativo con **planificación cooperativa** o **no expropiativa**.

En la **planificación cooperativa** cuando la CPU es asignada a un proceso, este la acapara hasta terminar o hasta pasar al estado de **esperando**.

La **planificación cooperativa** no requiere de ningún hardware especial, por lo que en algunas plataformas puede ser la única opción. Por ello estaba presente en los sistemas operativos más antiguos, como Windows 3.x y Mac OS —que no debemos confundir con el actual macOS—.

Sin embargo, las decisiones de planificación también pueden ser tomadas en otros dos casos:

1. Cuando ocurre una interrupción del temporizador, lo que permite detectar si un proceso

lleva demasiado tiempo ejecutándose.

2. Cuando un proceso pasa de **esperando** a **preparado**. Por ejemplo, porque para un proceso ha terminado la operación de E/S por la que estaba esperando.

Cuando el planificador es invocado en los cuatro casos decimos que tenemos **planificación expropiativa** o **apropiativa**.

La **planificación expropiativa** sí requiere de un soporte adecuado por parte del hardware, por lo que se utiliza en los sistemas operativos modernos. Ejemplos de estos sistemas son Microsoft Windows —desde Windows 95— Linux, macOS, y todos los UNIX modernos.

La utilización de un **planificador expropiativo** introduce algunas dificultades adicionales:

- Puesto que un proceso puede ser expropiado en cualquier momento —sin que pueda hacer nada para evitarlo— el sistema operativo debe proporcionar *mecanismos de sincronización* (véase el Capítulo 13) para coordinar el acceso a datos compartidos que podrían estar siendo modificados por el proceso que abandona la CPU y que puede necesitar el que entra en ella.
- ¿Qué ocurre si un proceso va a ser expropiado en el preciso momento en el que se está ejecutando una llamada al sistema? No debemos olvidar que dentro del núcleo se manipulan datos importantes, compartidos por todo el sistema, que deben permanecer consistentes en todo momento.

Para resolver esta cuestión la solución más sencilla es impedir la expropiación dentro del núcleo. Es decir, el cambio de contexto —que sacaría al proceso actual de la CPU y metería al siguiente— no ocurre inmediatamente, sino que se retrasa hasta que la llamada al sistema se completa o se bloquea poniendo al proceso en el estado de *esperando*. Esto permite núcleos simples y garantiza que las estructuras del mismo permanezcan consistentes, pero es una estrategia muy pobre para sistemas de tiempo real o multiprocesador. Exploraremos otras soluciones más adelante (véase el Apartado 14.6).

14.2. El asignador

El **asignador** es el componente que da el control de la CPU al proceso seleccionado por el planificador de corto plazo. Esta tarea implica realizar las siguientes funciones:

- Cambiar el contexto.
- Cambiar al modo usuario.
- Saltar al punto adecuado del programa para continuar la ejecución del proceso.

Puesto que el **asignador** es invocado para cada intercambio de procesos en la CPU, es necesario que el tiempo que tarda en detener un proceso e iniciar otro sea lo más corto posible. Al tiempo que transcurre desde que un proceso es escogido para ser planificado en la CPU hasta que es asignado a la misma se lo denomina **latencia de asignación**.

14.3. Criterios de planificación

Los diferentes algoritmos de planificación de la CPU tienen diversas propiedades que pueden favorecer a una clase de procesos respecto a otra. Por ello es interesante disponer de algún criterio para poder comparar los algoritmos y determinar cuál es el mejor.

Se han sugerido muchos criterios para comparar los algoritmos de planificación de CPU. La elección de uno u otro puede suponer una diferencia sustancial a la hora de juzgar qué algoritmo es el mejor.

A continuación presentamos los criterios más comunes.

14.3.1. Criterios a maximizar

Los algoritmos de planificación son mejores cuanto mayor es su valor para los siguientes criterios.

Uso de CPU

Un buen planificador debería mantener la CPU lo más ocupada posible. El **uso de CPU** es la proporción de tiempo que se usa la CPU en un periodo de tiempo determinado. Se suele indicar en tanto por ciento.

$$\text{Uso de CPU} = 100\frac{\text{Tiempo que la CPU permanece ocupada}}{\text{Tiempo durante el que se toma la medida}}\%$$

Tasa de procesamiento

Cuando la CPU está ocupada es porque el trabajo se está haciendo. Por tanto, una buena medida del volumen de trabajo realizado puede ser el número de tareas o procesos terminados por unidad de tiempo. A dicha magnitud es a la que denominamos como **tasa de procesamiento**.

$$\text{Tasa de procesamiento} = \frac{\text{Numero de procesos terminados}}{\text{Tiempo durante el que se toma la medida}}\text{procesos/s}$$

14.3.2. Criterios a minimizar

Los algoritmos de planificación son mejores cuanto menor es su valor para los siguientes criterios.

Tiempo de ejecución

Es el intervalo de tiempo que transcurre desde que el proceso es cargado hasta que termina.

Tiempo de espera

Es la suma de tiempos que el proceso permanece a la espera en la **cola de preparados**. Esta medida de tiempo no incluye el tiempo de espera debido a las operaciones de E/S.

Tiempo de respuesta

Es el intervalo de tiempo que transcurre desde que se lanza un evento —se pulsa una tecla, se hace clic con el ratón o llega un paquete por la interfaz de red— hasta que se produce la primera respuesta del proceso.

El **tiempo de respuesta** mide el tiempo que se tarda en responder y no el tiempo de E/S. Mientras que el **tiempo de ejecución** sí incluye el tiempo que consumen las operaciones de E/S, por lo que suele estar limitado por la velocidad de los dispositivos E/S.

14.3.3. Elección del criterio adecuado

En función del tipo de sistema o de la clase de trabajos que se van a ejecutar puede ser conveniente medir la eficiencia del sistema usando un criterio u otro. Esto a su vez beneficiará a unos algoritmos de planificación frente a otros, indicándonos cuáles son los más eficientes para nuestra clase de trabajos en particular.

En general podemos encontrar dos clases de trabajos para los que puede ser necesario evaluar la eficiencia del sistema de manera diferente: los trabajos interactivos y los que no lo son.

Sistemas interactivos

En los sistemas interactivos —ya sean sistemas de escritorio o *mainframes* de tiempo compartido— los procesos pasan la mayor parte del tiempo esperando algún tipo de entrada por parte de los usuarios.

En este tipo de sistemas, el **tiempo de ejecución** no suele ser el mejor criterio para determinar la bondad de un algoritmo de planificación, ya que viene determinado en gran medida por la velocidad de la entrada de los usuarios. Por el contrario, se espera que el sistema reaccione lo antes posible a las órdenes recibidas, lo que hace que el **tiempo de respuesta** sea un criterio más adecuado para evaluar al planificador de la CPU.

Generalmente, el **tiempo de respuesta** se reduce cuando el tiempo que pasan los procesos interactivos en la **cola de preparados** también lo hace —tras haber sido puestos ahí por la ocurrencia de algún evento— por lo que también puede ser una buena idea utilizar como criterio el **tiempo de espera**.

Esta selección de criterios no solo es adecuada para los sistemas interactivos, ya que existen muchos otros casos donde es interesante seleccionar un planificador de la CPU que minimice el tiempo de respuesta. Esto, por ejemplo, ocurre con algunos servicios en red, como: sistemas de mensajería instantánea, videoconferencia, servidores de videojuegos, etc.

Sistemas no interactivos

Por el contrario, en los antiguos *mainframes* de procesamiento por lotes y multiprogramados, en los superordenadores que realizan complejas simulaciones físicas y en los grandes centros de datos de proveedores de Internet como Google, lo de menos es el tiempo de respuesta y lo realmente importante es completar cada tarea en el menor tiempo posible. Por eso en ese tipo de sistemas es aconsejable utilizar criterios tales como el **tiempo de ejecución** o la **tasa de procesamiento**.

Promedio o varianza del criterio

Obviamente estos criterios varían de un proceso a otro, por lo que normalmente lo que se busca es optimizar los valores promedios en el sistema.

Sin embargo, no debemos olvidar que en muchos casos puede ser más conveniente optimizar el

máximo y mínimo de dichos valores antes que el promedio. Por ejemplo, en los sistemas interactivos es más importante minimizar la **varianza en el tiempo de respuesta** que el **tiempo de respuesta promedio**, puesto que para los usuarios un sistema con un tiempo de respuesta predecible es más deseable que uno muy rápido en promedio pero con una varianza muy alta.

14.4. Ciclo de ráfagas de CPU y de E/S

El éxito de la planificación de CPU depende en gran medida de la siguiente propiedad que podemos observar en hilos o procesos:

> *La ejecución de un hilo o proceso consiste en ciclos de CPU y esperas de E/S, de forma que alternan entre estos dos estados.*
>
> *La ejecución empieza con una ráfaga de CPU, seguida por una ráfaga de E/S, que a su vez es seguida por otra de CPU y así sucesivamente. Finalmente, la última ráfaga de CPU finaliza con una llamada al sistema —generalmente exit()— para terminar la ejecución del proceso.*

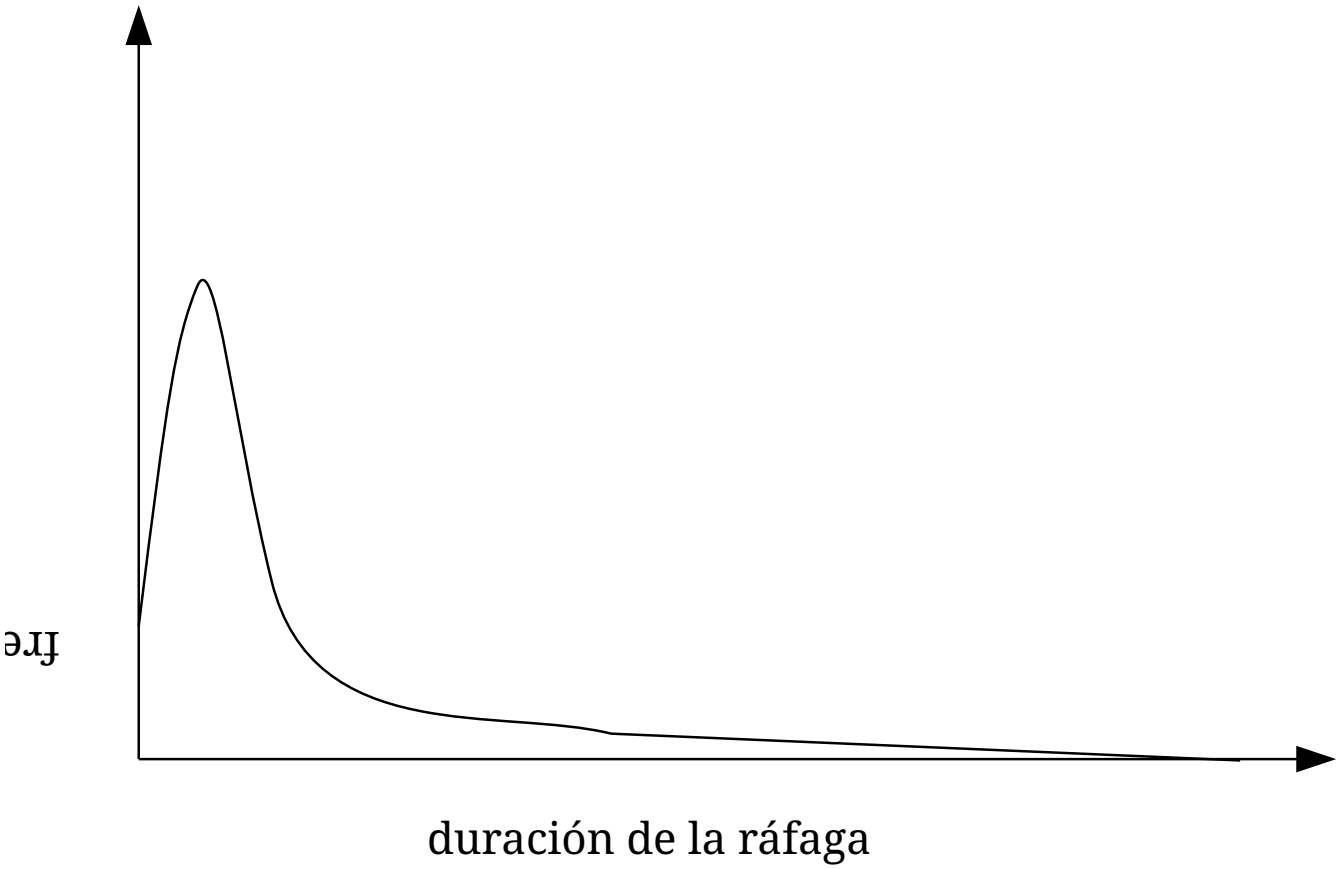


Figura 43. Histograma de los tiempos de las ráfagas de CPU.

La curva que relaciona la frecuencia de las ráfagas de CPU con la duración de las mismas tiende a ser exponencial o hiperexponencial (véase la Figura 43) aunque varía enormemente entre tipos de tareas y sistemas informáticos distintos. Esto significa que los procesos se pueden clasificar entre aquellos que presentan un gran número de ráfagas de CPU cortas o aquellos con un pequeño número de ráfagas de CPU largas.

Concretamente:

- Decimos que un proceso es **limitado por la E/S** cuando presenta muchas ráfagas de CPU cortas, debido a que si es así, es porque pasa la mayor parte del tiempo esperando por la E/S.
- Decimos que un proceso está **limitado por la CPU** cuando presenta pocas ráfagas de CPU largas, debido a que si es así, es porque hace un uso intensivo de la misma y a penas pasa tiempo esperando por la E/S.

Esta distinción entre tipos de procesos puede ser importante en la selección de un algoritmo de planificación de CPU adecuado, puesto que, por lo general el algoritmo escogido debe planificar antes a los procesos limitados por la E/S, evitando así que los procesos limitados por la CPU —que son los que tienden a usarla más tiempo— la acaparen.

Si esto último ocurriera, los procesos limitados por la E/S se acumularían en la **cola de preparados**, dejando vacías las colas de dispositivos. Este fenómeno, que provoca una infrautilización de los dispositivos de E/S, se denomina **efecto convoy**.

Planificar primero a los procesos limitados por la E/S tiene además dos efectos muy positivos:

- Los procesos interactivos son generalmente procesos limitados por la E/S, por lo que planificarlos primero hace que mejore el tiempo de respuesta.
- Generalmente el tiempo de espera promedio se reduce cuando se planifican primero los procesos con ráfagas de CPU cortas. Según las definiciones anteriores, estos procesos son precisamente los limitados por la E/S.

14.5. Algoritmos de planificación de la CPU

A continuación ilustraremos algunos de los algoritmos de planificación de CPU más comunes. Lo haremos considerando que cada proceso tiene una única ráfaga de CPU. Sin embargo, no debemos olvidar que para ser precisos necesitaríamos utilizar muchos más procesos, donde cada uno estuviera compuesto de una secuencia de miles de ráfagas alternativas de CPU y de E/S.

14.5.1. Planificación FCFS

En la planificación **FCFS** (*First Come, First Served*) o **primero que llega, primero servido** la cola es FIFO:

- Los procesos que llegan se colocan al final de la cola que les corresponde.
- El proceso asignado a la CPU se coge siempre del principio de la cola seleccionada.
- Es un algoritmo **cooperativo**, puesto que un proceso mantiene la CPU hasta que decide liberarla voluntariamente —recordemos que la ráfaga de CPU llega a su fin porque el proceso termina o solicita alguna operación que lo lleva el estado **esperando**—.

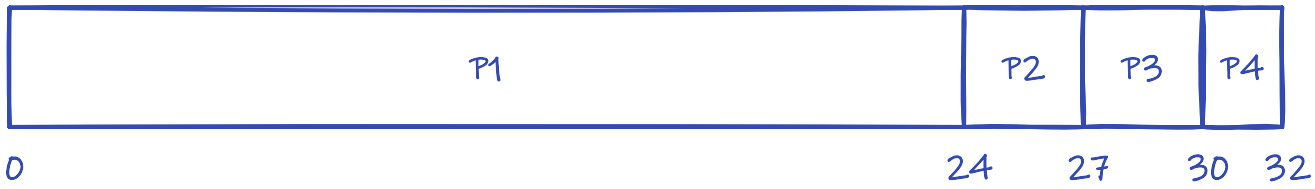
Ilustremos el algoritmo con un ejemplo. Supongamos que 4 procesos llegan a la **cola de preparados** en los tiempos indicados en la Tabla 8. Además, aunque es difícil tener un

conocimiento a priori del tiempo de la ráfaga de CPU de cada proceso, vamos a suponer que también son conocidos.

Tabla 8. Problema de planificación de la CPU mediante algoritmo FCFS.

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	24
P2	1	3
P3	2	3
P4	3	2

En la siguiente figura podemos ver el diagrama de Gantt de la planificación considerando que se utiliza el algoritmo **FCFS**:



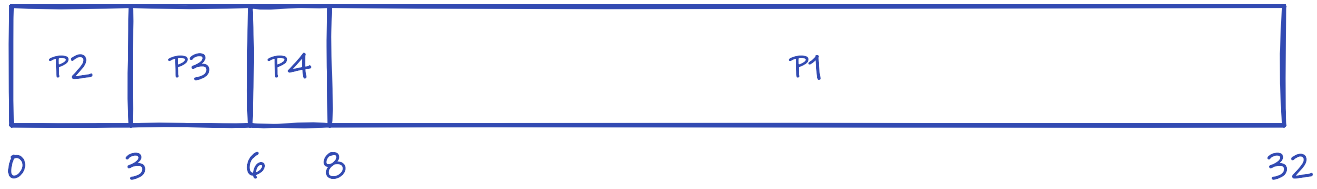
Utilizando el diagrama anterior, podemos calcular fácilmente los **tiempos de espera y de ejecución promedio**:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	24	(0-0)=	0	(24-0)=	24
P2	27	(24-1)=	23	(27-1)=	26
P3	30	(27-2)=	25	(30-2)=	28
P4	32	(30-3)=	27	(32-3)=	29
Tiempos promedio (ms)			18.75		26.75

Lo interesante es que el resultado cambia si los procesos llegan en otro orden. Por ejemplo, P1 podría llegar el último:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P2	0	3
P3	1	3
P4	2	2
P1	3	24

Entonces el resultado de la planificación sería el que se muestra en la siguiente figura:



Y los **tiempos de espera y ejecución promedio** correspondientes serían:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P2	3	(0-0)=	0	(3-0)=	3
P3	6	(3-1)=	2	(6-1)=	5
P4	8	(6-2)=	4	(8-2)=	6
P1	32	(8-3)=	5	(32-3)=	29
Tiempos promedio (ms)			2.75		10.75

Aunque el tiempo total necesario para ejecutar las ráfagas de los 4 procesos, los criterios utilizados reflejan que el algoritmo se comporta mucho mejor en el segundo caso. Por tanto, el algoritmo **FCFS** no garantiza ni **tiempos de espera** ni de **ejecución** mínimos, ya que pueden cambiar variar considerablemente con el orden en el que llegan los procesos.

Además, el algoritmo **FCFS** sufre el llamado **efecto convoy**. Para entenderlo, analicemos lo que está pasando en el ejemplo de la Tabla 8:

1. Al proceso P1 se le asigna la CPU. Durante el tiempo que P1 utiliza la CPU todos los otros procesos terminan sus operaciones de E/S y pasan a la **cola de preparados**. Por tanto, mientras los procesos esperan para utilizar la CPU, los dispositivos de E/S permanecen desocupados.
2. El proceso P1 termina de usar la CPU y pasa a una cola de dispositivos.
3. El resto de procesos P, que tienen ráfagas de CPU cortas, se ejecutan rápidamente y pasan a las colas de dispositivos. Por tanto, la CPU permanecerá vacía hasta que algún proceso termine la operación de E/S solicitada.
4. El proceso P1 pasa a la **cola de preparados** y se le asigna la CPU. Con el tiempo el resto de procesos terminarán sus operaciones y, nuevamente, tienen que esperar en la **cola de preparados** a que el proceso P1 termine de utilizarla.

Esto nos permite llegar a la conclusión de que en cierto orden de llegada la mayor parte de los procesos esperan constantemente detrás de uno para poder realizar su trabajo. Esto reduce la utilización de la CPU y de los dispositivos de E/S por debajo de lo que sería posible, si los procesos más cortos se ejecutasen primero.

14.5.2. Planificación SJF

La planificación **SJF** (*Shortest-Job First*) o **primero el más corto**, consiste en:

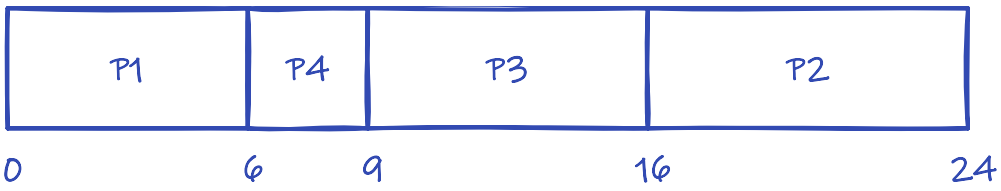
1. Se asocia con cada proceso la longitud de tiempo de su siguiente ráfaga de CPU.

2. Cuando la CPU está disponible, se pone **ejecutando** el proceso de menor ráfaga de CPU.
3. Si dos procesos tienen ráfagas de una misma longitud, se utiliza el algoritmo **FCFS** —entre ellos, el que lleva más tiempo en la **cola de preparados**—.
4. Es un algoritmo **cooperativo**, puesto que un proceso mantiene la CPU hasta que decide liberarla voluntariamente.

Ilustremos el algoritmo con un ejemplo:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	6
P2	1	8
P3	2	7
P4	3	3

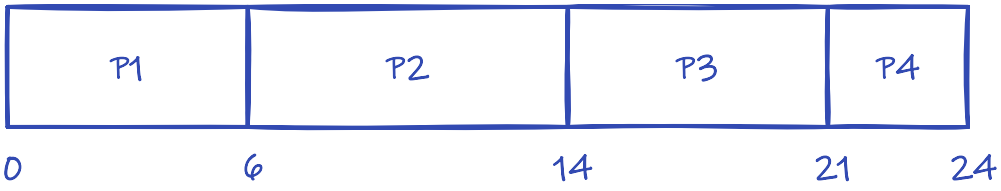
Considerando que se utiliza el algoritmo **SJF** obtendremos el siguiente diagrama de Gantt:



Con los **tiempos de espera y ejecución promedio** correspondientes:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	6	(0-0)=	0	(6-0)=	6
P2	24	(16-1)=	15	(24-1)=	23
P3	16	(9-2)=	7	(16-2)=	14
P4	9	(6-3)=	3	(9-3)=	6
Tiempos promedio (ms)			6.25		12.25

Sin embargo, si hubiéramos utilizado el algoritmo **FCFS**:



Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	6	(0-0)=	0	(6-0)=	6
P2	14	(6-1)=	5	(14-1)=	13
P3	21	(14-2)=	12	(21-2)=	19
P4	24	(21-3)=	18	(24-3)=	21
Tiempos promedio (ms)			8.75		14.75

El algoritmo **SJF** es óptimo en el sentido de que el **tiempo de espera promedio** es mínimo, porque reduce más el tiempo de espera de los procesos cortos y aumenta el de los procesos largos. Además, así se evita el **efecto convoy**.

Sin embargo, la pregunta que debemos hacernos es cómo podemos conocer de antemano la longitud de las ráfagas de CPU de un proceso, para usar esa información durante la planificación. Sin analizar el código, el sistema operativo no puede conocer el tiempo de una ráfaga hasta que esta no termina de ejecutarse.

Por eso el algoritmo SJF se utiliza frecuentemente como planificador de la **cola de trabajos**, donde se puede obligar al usuario a especificar un tiempo de ejecución máximo, al enviar el trabajo a dicha cola. En este caso, los usuarios tenderán a ajustar la estimación de tiempo de ejecución, puesto que los que tengan tiempos más cortos serán priorizados para ser ejecutados antes, frente a los de tiempos más largos.

Para evitar que los usuarios hagan trampas indicando un tiempo de ejecución más corto que el real, con el fin de que se planifique antes su trabajo, se puede utilizar un temporizador para abortar los trabajos que excedan el tiempo de ejecución indicado por el usuario. El error puede ser notificado al usuario para que vuelva a enviar el trabajo con una estimación más realista.

Para utilizar el algoritmo **SJF** en el planificador de la CPU, lo único que se puede hacer es intentar predecir el tiempo de la siguiente ráfaga de CPU. Por ejemplo, se puede utilizar un promedio ponderado exponencial de los tiempos de las de ráfagas de CPU pasadas:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n, \;\; 0 \leq \alpha \leq 1$$

donde:

- τ_{n+1} es la estimación de tiempo de la siguiente ráfaga de CPU
- t_n es el tiempo real de la última ráfaga
- α es el peso relativo del tiempo real de la última ráfaga.
- τ_n es la estimación de tiempo de la última ráfaga.

La expresión es recursiva, dado que τ_n se calcula usando la ecuación con t_{n-1} y τ_{n-1}; que a su vez depende de t_{n-2} y τ_{n-2}, y así sucesivamente.

Si desarrollamos la fórmula sustituyendo los valores, veremos que t_n se pondera con α, t_{n-1} con $(1 - \alpha)\alpha$, t_{n-2} con$(1 - \alpha)^2\alpha$, y así sucesivamente:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \ldots$$

Si α = 1, τ_{n+1} = αt_n, ignorando el resto del histórico. En otro caso, dado que tanto α como 1 - α son menores de 1, cada término sucesivo tiene menor peso que su predecesor, haciendo que los t_n contribuyan menos cuanto más alejados del tiempo actual.

El problema es que todos estos cálculos consumen tiempo de CPU, cuando el planificador debe ser lo más rápido posible, dado que se ejecuta con mucha frecuencia.

14.5.3. Planificación SRTF

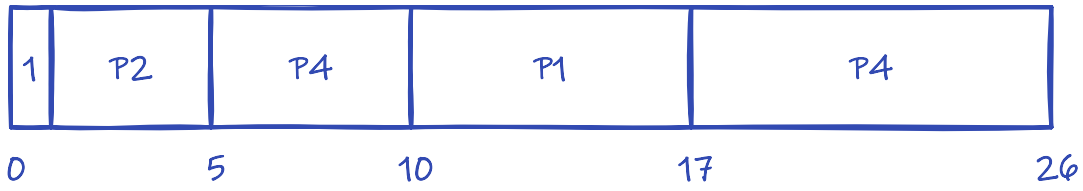
El algoritmo **SJF** es **cooperativo**, pero se puede implementar de forma **expropiativa**, en cuyo caso se llama **SRTF** (Shortest-Remaing-Time First). La diferencia está en lo que ocurre cuando un nuevo proceso llega a la **cola de preparados**.

En **SRTF** se compara el tiempo de la siguiente ráfaga de CPU del nuevo proceso, con el tiempo de ráfaga que le queda al proceso en ejecución. Si la primera magnitud es inferior, el proceso que tiene la CPU es expropiado y sustituido por el nuevo proceso. Mientras que en **SJF** no se hace nada. Se espera que el proceso que actualmente se está ejecutando termine su ráfaga de CPU voluntariamente.

Ilustremos el algoritmo con un ejemplo:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

El resultado es el que se muestra en el siguiente diagrama de Gantt:



Y los tiempos de espera y ejecución promedio correspondientes serían:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	17	(0-0) + (10-1)=	9	(17-0)=	17
P2	5	(1-1)=	0	(5-1)=	4
P3	26	(17-2)=	15	(26-2)=	24
P4	10	(5-3)=	2	(10-3)=	7

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
Tiempos promedio (ms)			6.50		13.00

Es muy complicado predecir cuál de los dos algoritmos será mejor para un conjunto concreto de procesos. Sin embargo, debemos tener en cuenta que —aunque no lo estemos considerando en estos problemas— un algoritmo expropitativo, por lo general, provocará más cambios de contexto en los que se perderá tiempo de CPU.

Los algoritmos expropiativos también suelen ofrecer mejores tiempos de respuesta, puesto que un proceso que llega a la **cola de preparados** puede ser asignado a la CPU sin esperar a que el proceso que se ejecuta en ella actualmente termine su ráfaga de CPU.

14.5.4. Planificación con prioridades

En la **planificación con prioridades** se asocia una prioridad a cada proceso, de tal forma que el de prioridad más alta es asignado a la CPU. En caso de igual prioridad, se utiliza **FCFS**.

Las prioridades se suelen indicar con números enteros en un rango fijo. Por ejemplo [0-7], [0-31], [0-139] o [0-4095]. En algunos sistemas operativos los números más grandes representan mayor prioridad, mientras que en otros son los procesos con números más pequeños los que se planifican primero. En este curso utilizaremos la convención de que a menor valor, mayor prioridad.

Si las prioridades se asignan en relación al tiempo de la próxima ráfaga de CPU, su comportamiento es el mismo que el del **SJF**; por lo que se considera a este último un caso particular de algoritmo de **planificación con prioridades**.

El algoritmo de planificación con prioridades puede ser **expropiativo** o **cooperativo**:

- En el caso **expropiativo**, cuando un proceso llega a la **cola de preparados** su prioridad es comparada con la del proceso en ejecución. Se expropia la CPU si la prioridad del nuevo proceso es superior a la prioridad del proceso que se ejecuta.
- En el caso **cooperativo**, no se toma ninguna decisión cuando llega un proceso a la **cola de preparados**, solo cuando el que tiene asignada la CPU la abandona.

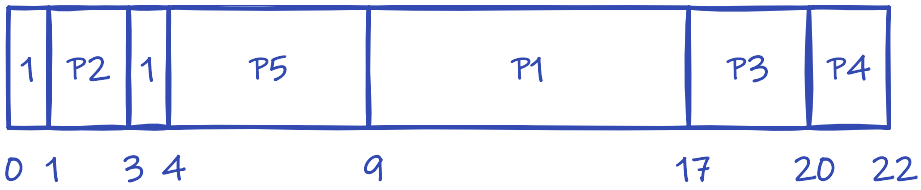
Supongamos que 5 procesos llegan a la cola de preparados en los tiempos indicados en la Tabla 9. Como en los ejemplos anteriores, aunque es difícil tener un conocimiento a priori del tiempo de las ráfagas de CPU, vamos a suponer que son conocidos. Y también que a cada proceso se le asigna, de alguna forma, una prioridad cuando llega a la **cola de preparados**.

Tabla 9. Problema de planificación de la CPU mediante algoritmo de planificación con prioridades.

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)	Prioridad
P1	0	10	3
P2	1	2	1
P3	2	3	4

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)	Prioridad
P4	3	2	5
P5	4	5	2

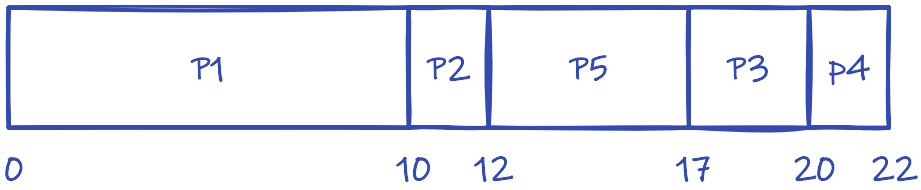
En las condiciones anteriores, si utilizamos el algoritmo de planificación por prioridades expropiativo, obtendremos el diagrama de Gantt de la siguiente figura:



Con los **tiempos de espera y ejecución promedio** correspondientes:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	4	(0-0) + (3-1) + (9-4)=	7	(17-0)=	17
P2	3	(1-1)=	0	(3-1)=	2
P3	20	(17-2)=	15	(20-2)=	18
P4	22	(20-3)=	17	(22-3)=	19
P5	9	(4-4)=	0	(9-4)=	5
Tiempos promedio (ms)			7.80		12.20

Mientras que si utilizamos el algoritmo de planificación por prioridades cooperativo, obtendremos el diagrama de Gantt de la siguiente figura:



Con los **tiempos de espera y ejecución promedio** correspondientes:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	10	(0-0)=	0	(10-0)=	10
P2	12	(10-1)=	9	(12-1)=	11
P3	20	(17-2)=	15	(20-2)=	18
P4	22	(20-3)=	17	(22-3)=	19

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P5	17	(12-4)=	8	(17-4)=	13
Tiempos promedio (ms)			9.80		14.20

Que estos algoritmos ofrezcan mejores o peores resultados que otros, obviamente depende de los criterios utilizados para asignar las prioridades.

Prioridades definidas internamente o externamente

Hay dos maneras de asignar las prioridades:

- **Internamente**. Se utiliza una cualidad medible del proceso para calcular su prioridad. Por ejemplo, límites de tiempo, necesidades de memoria, número de archivos abiertos, tiempo estimado de ráfaga de CPU —como en **SJF**— o la proporción entre esta y el tiempo estimado de ráfaga de E/S.
- **Externamente**. Las prioridades son fijadas por criterios externos al sistema operativo. Por ejemplo, la importancia del proceso para los usuarios, la cantidad de dinero pagada para el uso del sistema u otros factores políticos.

Algunas de estas formas de asignar las prioridades pueden ser fijas, mientras que otras pueden ser variables. Es decir, un criterio externo como es la importancia del proceso para los usuarios, puede dar lugar a una prioridad que se asigna al crear el proceso y que no cambia durante toda su ejecución. Por el contrario, un criterio como el tiempo de ráfaga de CPU puede dar lugar a una prioridad variable, que se ajusta cada vez que se tiene una estimación mejor.

Muerte por inanición

El mayor problema de este tipo de planificación es el **bloqueo indefinido** o **muerte por inanición**. Si hay un conjunto de procesos de alta prioridad demandando CPU continuamente, el algoritmo puede dejar a algunos procesos de menor prioridad esperando indefinidamente.

Una solución a este problema es aplicar mecanismos de **envejecimiento**. Consisten en aumentar gradualmente la prioridad de los procesos que esperan —por ejemplo, 1 unidad cada 15 minutos—. De esta manera los proceso de baja prioridad tarde o temprano tendrán una oportunidad para ejecutarse. Una vez se les asigna la CPU, se restablece su prioridad al valor original.

14.5.5. Planificación RR

El algoritmo **RR** (*Round-Robin*) es similar al **FCFS**, pero añadiendo la expropiación para conmutar entre procesos cuando llevan cierta cantidad de tiempo ejecutándose en la CPU.

Este algoritmo requiere los siguientes elementos:

- Definir una **ventana de tiempo** o **cuanto**, generalmente entre 10 y 100 ms
- Definir la **cola de preparados** como una cola circular, donde el planificador asigna la CPU a cada proceso en intervalos de tiempo de hasta un **cuanto**, como máximo.

Cuando un proceso está en la CPU pueden darse diversos casos:

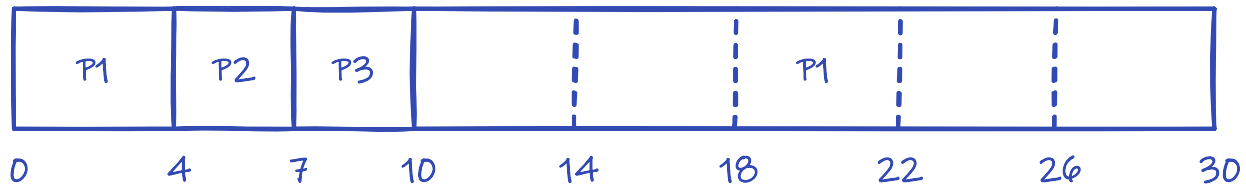
- Que la ráfaga de CPU sea menor que un cuanto. Entonces el proceso liberará la CPU voluntariamente, al terminar la ráfaga.
- Que la ráfaga de CPU sea mayor que un cuanto. El temporizador interrumpirá el proceso al terminar el cuanto e informará al sistema operativo. Este hará el cambio de contexto para asignar la CPU al siguiente proceso y el que abandona la CPU es insertado al final de la **cola de preparados**.

Este algoritmo es **expropiativo**, puesto que los procesos son expropiados por la interrupción del temporizador. Como se puede intuir, originalmente fue diseñado para los **sistemas de tiempo compartido**, para repartir la CPU por igual entre los procesos de los usuarios del sistema.

Ilustremos el algoritmo con un ejemplo con cuanto de 4 ms:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)
P1	0	24
P2	1	3
P3	2	3

El resultado es el que se muestra en el siguiente diagrama de Gantt:



Y los **tiempos de espera y ejecución promedio** correspondientes serían:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	30	(0-0) + (10-4)=	6	(30-0)=	24
P2	7	(4-1)=	3	(7-1)=	6
P3	10	(7-2)=	5	(10-2)=	8
Tiempos promedio (ms)			4.67		12.67

Rendimiento

Cuando se utiliza la planificación **RR** el tamaño del cuanto es un factor clave en la eficiencia del planificador:

- Cuando se reduce el **cuanto**, el **tiempo de respuesta** y el **tiempo de espera promedio** tienden a mejorar. Sin embargo el número de cambios de contexto será mayor, por lo que la ejecución de los procesos será más lenta.

Es importante tener en cuenta que interesa que el **cuanto** sea mucho mayor que el tiempo del cambio de contexto. Si, por ejemplo, el tiempo de cambio de contexto es un 10% del **cuanto**, entonces alrededor del 10% del tiempo de CPU se pierde en cambios de contexto.

- Cuando se incrementa el **cuanto**, el **tiempo de espera promedio** también se incrementa. En el caso extremo en el que el **cuanto** es tan grande que ningún proceso lo agota, el **RR** se convierte en **FCFS**, que suele tener grandes **tiempos de espera promedio**.

  Por otro lado, puede observarse experimentalmente que el **tiempo de ejecución promedio** generalmente mejora cuantos más procesos terminan su próxima ráfaga de CPU dentro de su **cuanto**. Por lo tanto, nos interesa un cuanto grande para que más procesos terminen su siguiente ráfaga dentro del mismo.

Dados tres procesos con una duración cada uno de ellos de 10 unidades de tiempo y cuanto igual a 1, el tiempo de ejecución promedio será de 29 unidades. Sin embargo, si el cuanto de tiempo fuera 10, el tiempo de ejecución promedio caería a 20 unidades de tiempo.

La regla general que siguen los diseñadores es intentar que el 80% de las ráfagas de CPU sean menores que el tiempo de **cuanto**. Se busca así equilibrar los criterios anteriores, evitando que el tiempo de cuanto sea demasiado grande o demasiado corto.

Actualmente se utilizan tiempos de cuanto de entre 10 y 100 ms Estos tiempos son mucho mayores que los tiempos de cambios de contexto, que generalmente son inferiores a 10 µs.

Reparto equitativo del tiempo de CPU

Uno de los inconvenientes del algoritmo **RR** es que no garantiza el reparto equitativo del tiempo de CPU entre los procesos limitados por la E/S y los limitados por la CPU —aunque es mejor que **FCFS**—.

Esto es debido a que los primeros utilizan el procesador durante periodos cortos de tiempo, para bloquearse posteriormente a la espera de que se realice la operación de E/S que han solicitado. Cuando la espera termina, vuelven a la **cola de preparados** donde aguardan a que se les asigne la CPU. Sin embargo, eso no va a ocurrir rápidamente si en el sistema hay procesos limitados por la CPU, pues estos generalmente agotan el **cuanto** antes de ser forzados a volver a la **cola de preparados**.

Así, los procesos limitados por la CPU hacen un mayor uso de la misma, mientras que los limitados por la E/S pueden tener que esperar durante bastante tiempo —aunque menos que si el algoritmo fuera **FCFS**, donde no hay **cuanto**— en la **cola de preparados** antes entrar en la CPU para solicitar una nueva operación de E/S. Esto hace que se desaprovechen los dispositivos de E/S y genera un incremento de la varianza del tiempo de respuesta. Para evitarlo se puede optar por un **planificador de colas multinivel** —para resolver el problema combinando el algoritmo **RR** con otro que priorice adecuadamente los procesos limitados por la E/S (véase el Apartado 14.5.7)— o por la **planificación equitativa** que veremos a continuación.

14.5.6. Planificación equitativa

Hasta el momento hemos hablado de planificadores que se centran en cuál es el proceso más importante para ejecutarlo a continuación. Sin embargo otra opción, desde el punto de vista de la planificación, es dividir directamente el tiempo de CPU entre los procesos. Esto es precisamente lo que hace la **planificación equitativa** (*Fair Scheduling*) que intenta repartir por igual el tiempo de CPU entre los procesos de la cola de preparados.

Por ejemplo, si 4 procesos compiten por el uso de la CPU, el planificador asignaría un 25% del tiempo de la misma a cada uno. Si a continuación un usuario inicia un nuevo proceso, el planificador tendría que ajustar el reparto asignando un 20% del tiempo a cada uno, ya que ahora habría 5 procesos compitiendo por el tiempo de CPU.

El algoritmo de planificación equitativa es muy similar al algoritmo **RR**. Pero, mientras que en este último se utiliza un cuanto de tamaño fijo, en la planificación equitativa la ventana de tiempo se calcula dinámicamente para garantizar el reparto equitativo de la CPU.

Planificación equitativa ponderada

Al igual que en los algoritmos anteriores, en ocasiones puede ser interesante priorizar unos procesos frente a otros, tanto por motivos ajenos al sistema operativo como por motivos internos. Por ejemplo, se puede querer favorecer a los procesos limitados por la E/S para mejorar la eficiencia del sistema, tal y como comentamos en el apartado Apartado 14.4.

La **planificación equitativa** resuelve este problema permitiendo que a los procesos se les asignen pesos y repartiendo proporcionalmente más tiempo de CPU a los procesos con mayor peso. A esta generalización del planificador equitativo se la conoce como **planificador equitativo ponderado**.

Linux, desde la versión 2.6.23, utiliza un tipo de **planificador equitativo ponderado** denominado **CFS** (*Completely Fair Scheduler*) o **planificador completamente equitativo** (véase «Inside the Linux 2.6 Completely Fair Schedule — IBM Developer»). Otro ejemplo es Zircon —el *microkernel* de Google Fuchsia— que está inmerso en cambiar su actual planificador basado en colas multinivel con prioridades a una implementación del **planificador equitativo ponderado** (véase «Zircon Fair Scheduler — Fuchsia Project»).

Implementación

Para ilustrar como funciona este tipo de planificadores, vamos a describir brevemente una posible implementación basada en la de Zircon.

Zircon es un *microkernel* multihilo con modelo uno a uno, por lo que su planificador trabaja con hilos. Es decir, las propiedades que necesita el planificador para hacer su trabajo están vinculadas a los hilos y son estos hilos los que el planificador ordena y selecciona para ser ejecutados en la CPU. Por el contrario, en la versión que vamos a describir los hilos no existen. En su lugar hablaremos de planificar procesos en la CPU, por coherencia con cómo hemos explicado el resto de planificadores de este capítulo.

Este planificador tiene una serie de parámetros ajustables:

- La **granularidad mínima *M***, es la porción mínima de tiempo de CPU que se puede asignar a

cualquier proceso.

- La **latencia de planificación *L***, es el periodo de tiempo en el que todos los procesos deben haber sido planificados aproximadamente una vez en la CPU.

  Lo que hace el planificador es repartir periodos de tiempo L entre todos los procesos que compiten por la CPU, para que todos tengan una oportunidad de ejecutarse en intervalos de al menos L segundos.

Como en otros planificadores, los procesos listos para ejecutarse en un instante dado se almacenan en la cola de preparados. Cada proceso P_i tiene una serie de propiedades:

- **Peso w_i,** que indica el peso relativo del proceso en el reparto del tiempo de CPU. Se trata de un número real en el intervalo (0.0, 1.0].
- **Instante inicial T_{si}** de la ráfaga de CPU del proceso, en el *tiempo virtual* de la CPU.
- **Instante final T_{fi}** de la ráfaga de CPU del proceso, en el *tiempo virtual* de la CPU.
- **Cuanto del tiempo de CPU T_i** en el periodo actual. Como ocurre con el cuanto en la planificación RR, se trata del tiempo máximo que el proceso puede estar de forma continua en la CPU.

El concepto de *tiempo virtual* es muy común en este tipo de planificadores. Es como si cada proceso tuviera su propia dimensión temporal, de forma que el tiempo fluyera más despacio para los procesos con mayor peso w_i. Así, una unidad de tiempo virtual para un proceso muy pesado implica más tiempo real, que la misma unidad para un proceso muy ligero.

Sea t es el instante de tiempo en el que un proceso P_i entra en la cola de preparados, en relación al tiempo virtual el proceso comienza a ejecutarse en la CPU desde ese mismo instante y necesitará más tiempo para terminar cuanto mayor sea su peso. Por lo tanto, podemos calcular el instante inicial T_{si} y el instante final T_{fi} de la siguiente manera:

$$T_{si} = t$$
$$T_{fi} = T_{si} + L / w_i$$

El tiempo virtual no es el tiempo real en la CPU. Por lo tanto, el proceso P_i no entrará en la CPU nada más llegar en t, pero si se colocará en la cola de preparados en orden ascendente en relación al instante final T_{fi}. Es decir, los procesos con mayor peso w_i se colocarán antes en la cola de preparados, con bastante probabilidad.

Cuando un proceso P_i es seleccionado para pasar al estado **ejecutando**, se calcula su cuanto de tiempo. Como el tiempo se asigna en múltiplos de M, primero es necesario saber cuantas unidades de tiempo M tiene el periodo de latencia de planificación L, que es el tiempo máximo que se le podría asignar si el proceso estuviera solo:

$$g_i = \text{floor}\left(\frac{L}{M}\right)$$

A cada proceso le corresponde más tiempo de CPU según su peso en relación al resto de procesos en competición, lo que lleva a calcular el peso relativo r_i como:

$$r_i = \frac{w_i}{W}$$

donde W es la suma de los pesos de todos los procesos en competición por la CPU —en la cola de preparados— más el peso del que se está ejecutando.

$$W = \sum_i w_i : P_i[\text{estado}] \in \{\text{PREPARADO, EJECUTANDO}\}$$

Cuando un proceso entra en la cola de preparados su peso se suma a W, mientras que cuando pasa al estado **esperando** o termina, su peso se resta.

Finalmente, podemos calcular el cuanto del proceso P_i:

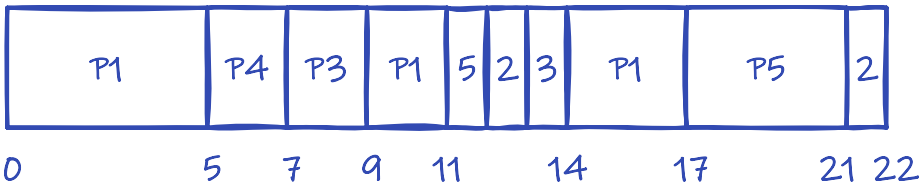
$$T_i = \text{ceil}(g_i \cdot r_i) \cdot M$$

Esta definición asegura que T_i sea un entero múltiplo de la granularidad mínima M, mientras es aproximadamente proporcional al peso del proceso respecto al resto de procesos con los que compite.

Ilustremos el algoritmo con un ejemplo con L = 5 y M = 1:

Proceso	Tiempo de llegada (ms)	Tiempo de ráfaga de CPU (ms)	Peso
P1	0	10	0.3
P2	1	2	0.1
P3	2	3	0.4
P4	3	2	0.5
P5	4	5	0.2

El resultado es el que se muestra en el siguiente diagrama de Gantt:



Como el primer proceso está solo cuando se le asigna la CPU, puede consumir toda la latencia de planificación L en el primer periodo. Sin embargo, ya han llegado el resto de procesos del problema cuando es expulsado de la CPU, por lo que a partir de ese momento tiene que competir con ellos. Entonces, el orden para asignar la CPU viene determinado por el peso y el tiempo de cuanto por la expresión para T_i que vimos anteriormente.

Una vez resuelto el problema, los **tiempos de espera y ejecución promedio** correspondientes serían:

Proceso	Instante de finalización (ms)	Tiempo de espera (ms)		Tiempo de ejecución (ms)	
P1	17	(0-0) + (9-5) + (14-11)=	7	(17-0)=	17
P2	22	(12-1) + (21-13)=	19	(22-1)=	21
P3	14	(13-2)=	11	(14-2)=	12
P4	7	(5-4)=	1	(7-3)=	4
P5	21	(11-5) + (17-12)=	11	(21-4)=	17
Tiempos promedio (ms)			9.80		14.20

14.5.7. Planificación con colas multinivel

Los diseñadores recurren a la **planificación de colas multinivel** cuando quieren combinar las características de varios algoritmos.

En la planificación con colas multinivel se divide la cola de preparados en colas separadas. Los procesos son asignados permanentemente a alguna de dichas colas, cada una de las cuales puede tener un algoritmo de planificación distinto.

La asignación de un proceso a una cola se hace en relación a alguna una característica del proceso. Por ejemplo, si es interactivo o no, su prioridad o su tamaño en memoria. Se hace de esta manera porque se supone que los procesos se pueden clasificar, y que cada clase tiene diferentes requerimientos. Por ejemplo, si los procesos se clasifican en interactivos o no interactivos, los primeros pueden ir a una cola con planificación **RR** mientras los segundos van a una con **FCFS**.

Figura 44. Ejemplo de planificación con colas multinivel.

Una cuestión interesante es cómo seleccionar la cola que debe escoger al siguiente proceso a ejecutar:

- Una opción común en los sistemas actuales es utilizar un **planificador con prioridades**. Es decir, que cada cola tenga una prioridad y así el planificador solo tiene que escoger la cola de

prioridad más alta que no esté vacía.

Por ejemplo, en la Figura 44, mientras un proceso de prioridad 1 esté preparado, no se escoge ningún otro de prioridad inferior. Si este planificador se implementa de forma expropiativa, el proceso que tiene asignada la CPU es expulsado si un proceso entra en una de las colas que tiene mayor prioridad que la suya.

- Otra opción es usar cuantos sobre las colas. Es decir, que a cada cola se le asigne una porción del tiempo de la CPU que debe repartirse entre los distintos procesos en la misma.

  Por ejemplo, un 80% de CPU para la cola de procesos interactivos, con planificación **RR**, y el 20% de CPU restante para la cola de procesos no interactivos, con planificador **FCFS**.

La **planificación de colas multinivel** con un **planificador con prioridades** para escoger la cola adecuada, es con diferencia la opción más común en los sistemas operativos modernos. Sin embargo, en este tipo de **colas multinivel** la asignación de los procesos a las colas es permanente —si la asignación se hace por prioridad, significa que la prioridad es fija—. Mientras que hoy en día es común que los procesos se muevan entre colas según las características del proceso.

14.5.8. Planificación con colas multinivel realimentadas

Para aumentar la flexibilidad de la planificación con colas multinivel se puede permitir a los procesos pasar de una cola a otra. Así se pueden clasificar en colas distintas, procesos con diferentes tiempos de ráfaga de CPU. Por ejemplo, para situar los procesos interactivos o limitados por la E/S en las colas de más alta prioridad, lo que ya hemos discutido que mejora los tiempos de espera y de respuesta y evita el **efecto convoy**.

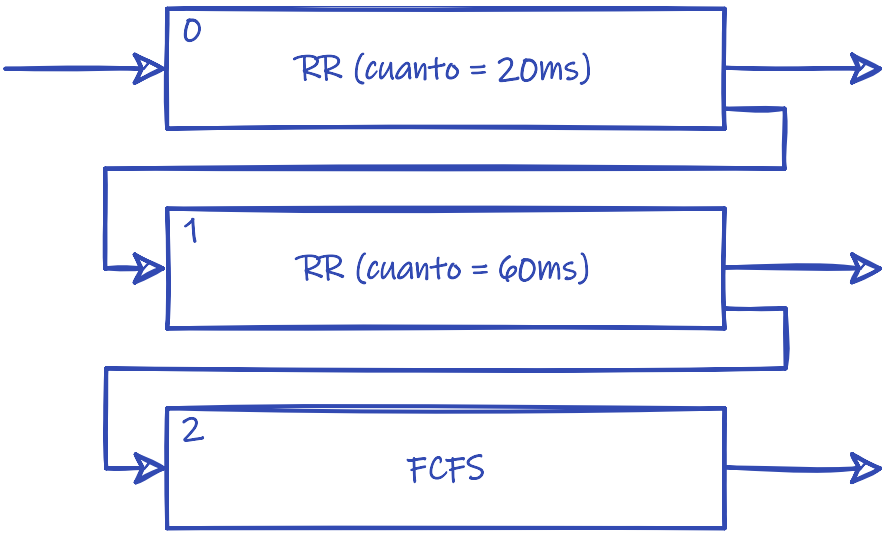


Figura 45. Ejemplo de planificación con colas multinivel realimentadas.

Por ejemplo, supongamos un **planificador de colas multinivel** donde cada cola tiene una prioridad, así que se usa la **planificación con prioridades** para seleccionar la cola. En las colas se usa el algoritmo **RR** para seleccionar el siguiente proceso, siendo el **cuanto** de la cola mayor cuanto menos prioritaria es la cola (véase la Figura 45). Los procesos que llegan nuevos o desde el estado **esperando** lo hacen con la prioridad más alta —que por convención hemos decidido que sea 0— así que se insertan en la cola correspondiente. Mientras que los procesos expropiados por vencimiento del **cuanto** pierde un punto de prioridad, siendo insertados en una cola de prioridad menor.

Con este algoritmo los procesos limitados por E/S suelen ejecutarse la mayor parte del tiempo con prioridades más altas que los limitados por CPU. Por ejemplo, usando los valores del esquema de la Figura 45, los procesos de ráfagas de CPU entre 20 y 80 ms acaban cayendo a la cola de prioridad 1 tras 20 ms de ejecución. Así dejan paso a los procesos con ráfagas menores de 20 ms, que siempre se ejecutan con prioridad 0. Finalmente, los procesos de ráfagas mayores de 80 ms van a la cola FCFS, desde donde solo tendrán acceso a la CPU cuando no haya ningún proceso de los otros tipos en la **cola de preparados**

El **planificador de colas multinivel realimentadas** también se puede utilizar para pasar a colas superiores los procesos que han esperado mucho tiempo en colas inferiores, evitando la **muerte por inanición**, que puede afectar a los sistemas de **planificación de colas multinivel** con **prioridad fija**.

Por ejemplo, el algoritmo **RR virtual** es un caso de **planificador de colas multinivel realimentadas** que resuelve los problemas del **RR**, en cuanto al reparto de la CPU entre procesos limitados por la E/S y limitados por la CPU (véase el Apartado 14.5.5.2).

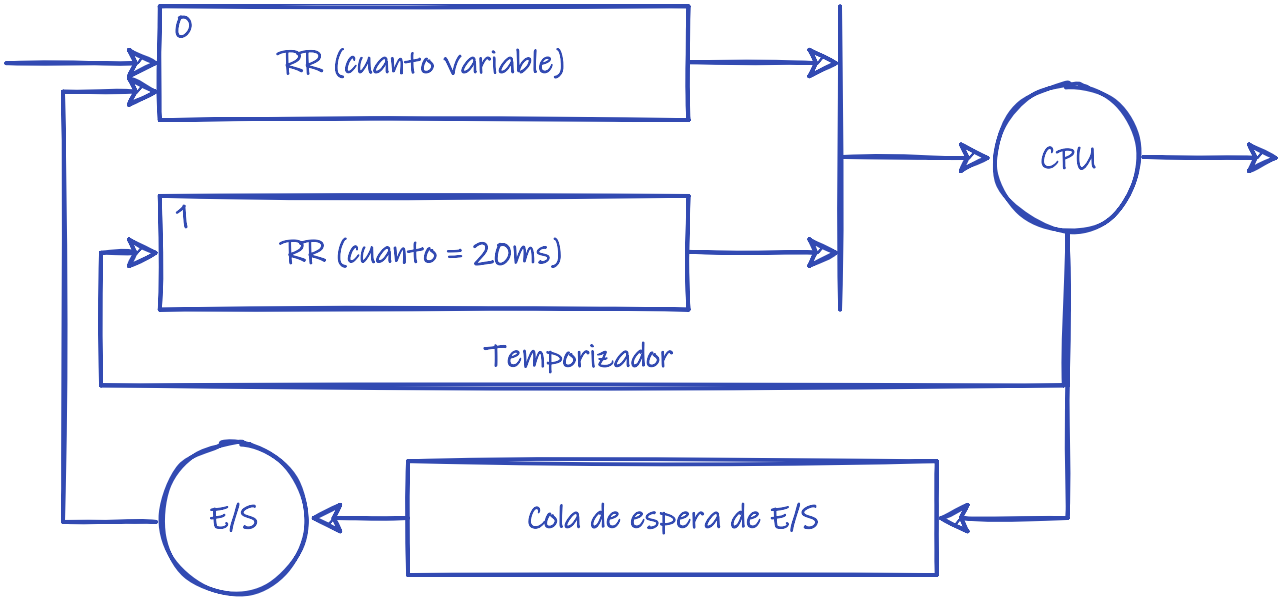


Figura 46. Ejemplo de planificación con RR virtual.

Tal y como se ilustra en la Figura 46, en el **RR virtual** los procesos por lo general tienen prioridad 1. Sin embargo, aquellos que vuelven al estado **preparado** desde **esperando** después de una operación de E/S, obtienen una bonificación en la propiedad que los lleva a tener prioridad 0. Por tanto, los procesos que usan con más frecuencia la E/S, usan más la cola de prioridad más alta, por lo que se les asigna antes la CPU mayor frecuencia.

Esta solución puede llevar a que si hay muchos procesos limitados por la E/S, estos acaparen la CPU y no den oportunidad de ejecutarse a los procesos en la cola de prioridad 1. Para evitarlo, el algoritmo **RR** de la cola de prioridad 0 tiene un **cuanto** variable, de tal forma que cada proceso recibe lo que le queda del **cuanto** de la cola de prioridad 1 tras haber consumido parte en la CPU en la ráfaga anterior. Esto hace que incluso los procesos con ráfagas de CPU más cortas acaben consumiendo su **cuanto** en la cola de prioridad 0 y terminen cayendo a la cola de prioridad 1, dando oportunidad de ejecutarse a otros procesos.

Planificación en Microsoft Windows

Para ilustrar los visto hasta el momento sobre la planificación de la CPU en sistemas operativos modernos, vamos a comentar las principales características de las últimas versiones de Microsoft Windows a este respecto.

Las actuales versiones de sistemas operativos Windows pertenecen a la familia de Microsoft Windows NT; que nació con el sistema operativo Windows NT 3.1 en 1993 y que llega hasta hoy en día con Microsoft Windows 10 y Windows Server 2019 —que se corresponden con la versión 10.0 de dicha familia Windows NT—

El núcleo de la familia Windows NT es multihilo e internamente implementa un algoritmo de planificación expropiativa con colas multinivel realimentadas basado en prioridades.

En Windows las prioridades de los hilos se pueden ver desde dos perspectivas: la de Windows API y la del núcleo. Ambas tienen una organización muy diferente, pero en última instancia, las primeras deben traducirse en las segundas.

Tabla 10. Clases de prioridad en Windows API.

Clase	Valor
REALTIME_PRIORITY_CLASS	0x00000100
HIGH_PRIORITY_CLASS	0x00000080
ABOVE_NORMAL_PRIORITY_CLASS	0x00008000
NORMAL_PRIORITY_CLASS	0x00000020
BELOW_NORMAL_PRIORITY_CLASS	0x00004000
IDLE_PRIORITY_CLASS	0x00000040

Desde el punto de vista de Windows API, todo proceso pertenece a alguna de las 6 clases de prioridad de la Tabla 10. La clase de prioridad de un proceso se puede indicar durante la creación del proceso, a través del argumento `dwCreationFlags` de la función CreateProcess(), o sea puede obtener y cambiar con las funciones GetPriorityClass() y SetPriorityClass(), respectivamente. Por lo general, la clase de prioridad `NORMAL_PRIORITY_CLASS` es la clase por defecto de cualquier proceso nuevo, excepto que se indique otra cosa durante su creación.

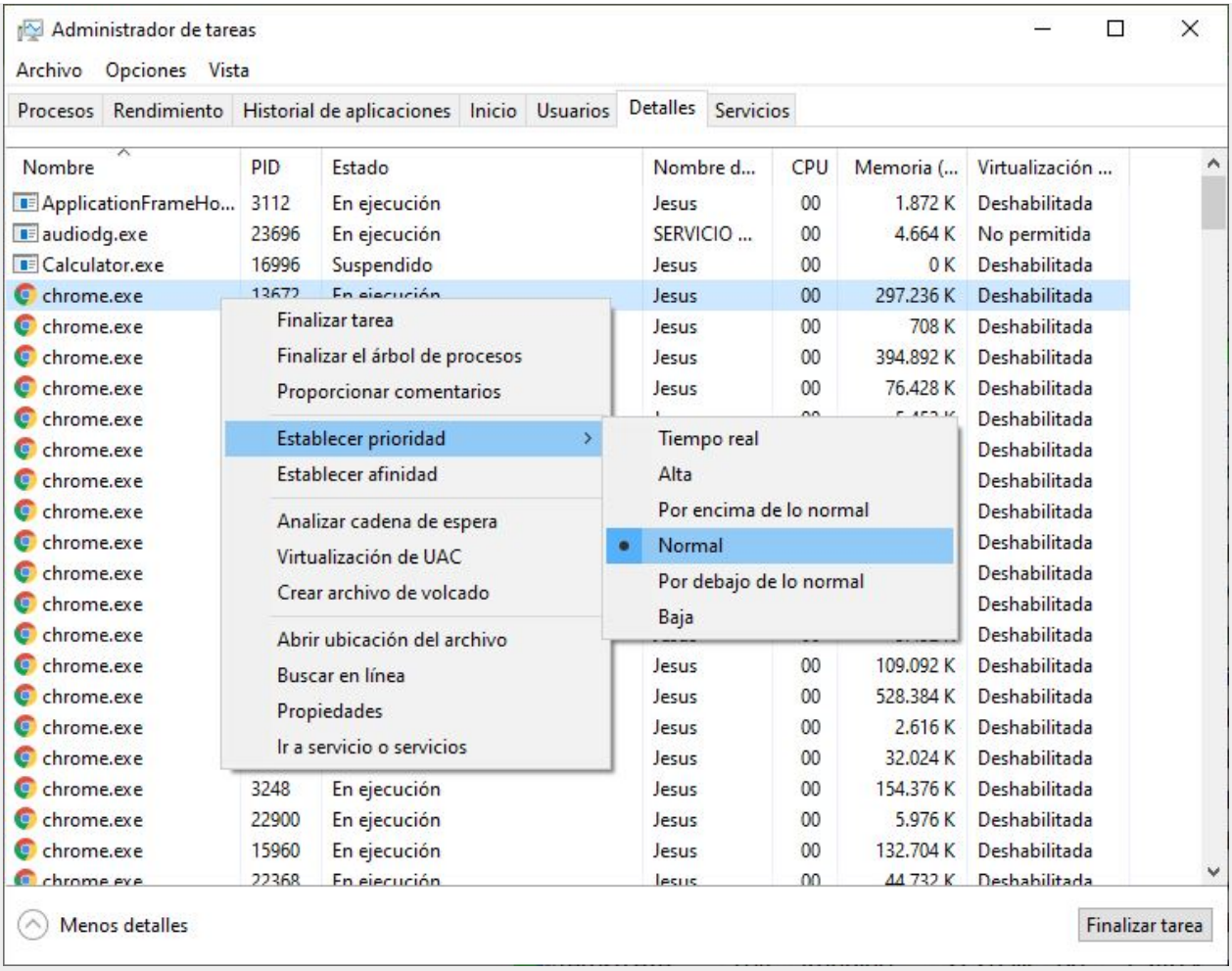


Figura 47. Cambiar la prioridad de un proceso en el **Administrador de tareas**.

Con el **Administrador de tareas** de Windows podemos alterar fácilmente la clase de prioridad de un proceso durante su ejecución (véase la Figura 47).

Tabla 11. Clases de prioridad en Windows API.

Prioridad	Valor
THREAD_PRIORITY_TIME_CRITICAL	15
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_IDLE	15

Al mismo tiempo, cada hilo del sistema tiene alguno de las prioridades de la Tabla 11. La prioridad de un hilo recién creado es `THREAD_PRIORITY_NORMAL`, pero se puede cambiar usando la función SetThreadPriority().

El núcleo de Windows tiene 32 prioridades, siendo 31 la prioridad más alta y 0 la más baja. Estos valores se dividen en dos rangos. El rango de prioridades de tiempo real va de 16 a 31 y solo está disponible para hilos en procesos en la clase de prioridad `REALTIME_PRIORITY_CLASS`.

Mientras que el rango de prioridades dinámicas va de 1 a 15. El nivel 0 está reservado para el sistema y se usa para una rutina especializada en limpiar zonas de memoria liberada por los procesos, poniéndolas a 0.

La prioridad base —o prioridad estática— de cada hilo que ve el núcleo se calcula combinando la prioridad del hilo y la clase de prioridad del proceso al que pertenece.

Tabla 12. Clases de prioridad base en Windows API.

	Clase de prioridad del proceso					
	REALTIME	**HIGH**	**ABOVE NORMAL**	**NORMAL**	**BELOW NORMAL**	**IDLE**
TIME CRITICAL	31	15	15	15	15	15
HIGHEST	26	15	12	10	8	6
ABOVE NORMAL	25	15	11	9	7	6
NORMAL	24	13	10	8	6	4
BELOW NORMAL	23	12	9	7	5	3
LOWEST	22	11	8	6	4	2
IDLE	16	1	1	1	1	1

Esta prioridad base es la prioridad real del hilo, si este tiene una prioridad en el rango de tiempo real —es decir, si el proceso al que pertenece está en la clase de prioridad `REALTIME_PRIORITY_CLASS`—. Mientras que para el resto de hilos, el sistema suma ciertas bonificaciones a la prioridad base para calcular la prioridad dinámica, que es con la que realmente será planificado el hilo. Estas bonificaciones se truncan para que nunca puedan hacer que el hilo se meta en el rango de tiempo real.

La prioridad real la usa el sistema para determinar a qué cola va el hilo cuando va a ser insertado en la **cola de preparados**. Para cada nivel de prioridad hay una cola con algoritmo **RR**, de tal forma que el planificador escoge primero a los hilos con prioridad más alta. Dentro de la misma prioridad la CPU se asigna en turno, dándoles un **cuanto** de tiempo de CPU.

Cuando llega un hilo a la cola de preparados, expropia la CPU al hilo que la tiene asignada si este tiene menor prioridad. Esto puede ocurrir incluso si el hilo a expropiar está en medio de una llamada al sistema, ya que, como cualquier sistema operativo moderno, el núcleo de Windows es expropiable —lo que veremos en el Apartado 14.6.4.2 que ofrece latencias de asignación más bajas que si no lo fuera—.

Respecto al **cuanto**, desde Windows Vista –NT 6.0– no se usa el temporizador para controlarlo sino el contador de ciclos de reloj de la CPU. Así el sistema puede determinar con precisión el tiempo que se ha estado ejecutando un hilo, excluyendo los tiempos dedicados a otras cuestiones, como por ejemplo a manejar interrupciones.

Desde el Intel Pentium las CPU de la familia x86 incorporan un contador de marca de tiempo (*Time Stamp Counter* o TSC) de 64 bits que indica el número de ciclos transcurridos desde el último reinicio del procesador.

Para más información véase «Time Stamp Counter — Wikipedia».

Una característica curiosa es que los hilos expropiados se insertan en la cabeza de su cola —no en el final— y conservan lo que les queda de **cuanto**. Mientras que se insertan por el final con el valor de **cuanto** reiniciado cuando abandonan la CPU por haber agotado el cuanto anterior. Estos últimos, además, pierden un nivel de prioridad si se ejecutaban con una prioridad superior a su prioridad base, a causa de alguna modificación.

Las bonificaciones a los hilos en el rango de prioridades dinámicas vienen determinadas por distintos criterios, escogidos para mejorar el **tiempo respuesta** y el **tiempo de espera** —priorizando los procesos limitados por E/S— evitar la **muerte por inanición** y la **inversión de prioridad**.

Se bonifican los hilos que despiertan tras completar operaciones de E/S con una cantidad que depende del tipo de dispositivo. Por ejemplo, la bonificación es mejor para hilos que han esperado por el teclado o el ratón que para los que esperaron por dispositivos del almacenamiento. También son bonificados los hilos que despiertan de eventos, semáforos y de otros objetos de sincronización. En este último caso, incluso se les ofrece más tiempo de **cuanto** si es el hilo asociado a la ventana de primer plano, con el objetivo de mejorar la respuesta de las aplicaciones interactivas. También se bonifica cualquier hilo que gestione elementos de la interfaz gráfica cuando despierta para responder a eventos del sistema de ventanas.

Para evitar la **muerte por inanición**, el planificador escoge cada segundo unos pocos hilos que llevan esperando aproximadamente 4 segundos, les triplica el **cuanto** y les aumenta la prioridad a 15. Estos hilos recuperan su prioridad base y el cuanto anterior cuando agotan el tiempo de cuanto actual o son expropiados de la CPU

14.6. Planificación de tiempo real

En el Apartado 2.7 discutimos la importancia de los sistemas de tiempo real. A continuación, describiremos las funcionalidades necesarias para soportar la ejecución de procesos en tiempo real dentro de un sistema operativo de propósito general.

14.6.1. Tiempo real estricto

Los sistemas de **tiempo real estricto** son necesarios para realizar tareas críticas que deben ser completadas dentro de unos márgenes de tiempo preestablecidos.

Generalmente las tareas son entregadas al sistema operativo junto con una declaración de las restricciones de tiempo —periodicidad y límite de tiempo— y la cantidad de tiempo que necesitan para ejecutarse. El planificador solo admitirá las tareas si puede garantizar el cumplimiento de las restricciones de tiempo, rechazándolas en caso contrario.

El ofrecer estas garantías requiere que el planificador conozca exactamente el tiempo máximo que se tarda en realizar todas y cada una de las funciones del sistema operativo. Esto es imposible en sistemas con almacenamiento secundario o memoria virtual, ya que introducen variaciones no controladas en la cantidad de tiempo necesario para ejecutar una tarea. Por tanto, el **tiempo real estricto** no es compatible con los sistemas operativos de propósito general, como los sistemas operativos de escritorio modernos.

14.6.2. Tiempo real flexible

La ejecución de procesos de **tiempo real flexible** es menos restrictiva. Tan solo requiere que los procesos críticos reciban mayor prioridad que los que no lo son. Esto puede generar excesos en la cantidad de recursos asignados a los procesos de tiempo real, así como inanición y grandes retrasos en la ejecución del resto de los procesos, pero es compatible con los sistemas de propósito general.

Además nos permite conseguir sistemas de propósito general que soporten multimedia, videojuegos y otras tareas que no funcionan de manera aceptable en un entorno que no implementara tiempo real flexible. Por ello, la mayor parte de los sistemas operativos modernos soportan este tipo de tiempo real.

14.6.3. Implementación del soporte de tiempo real

Implementar el soporte de tiempo real flexible en un sistema operativo de propósito general requiere:

- **Sistema operativo con planificación con prioridades**. Los procesos de tiempo real deben tener la mayor prioridad y ser fija. Es decir, no deben ser afectados por ningún mecanismo de envejecimiento o bonificación, que pueda usarse con los procesos de tiempo no real.
- **Baja latencia de asignación**. Cuanto menor es la latencia, más rápido comenzará a ejecutarse el proceso de tiempo real después de ser seleccionado por el planificador de la CPU.

Mientras que el primer requerimiento es bastante sencillo de conseguir, el segundo es mucho más complejo.

14.6.4. Reducir la latencia de asignación

Muchos sistemas operativos tienen un núcleo no expropiable. Estos núcleos no pueden realizar un cambio de contexto mientras se está ejecutando código del núcleo —por ejemplo, debido a una llamada al sistema— por lo que se ven obligados a esperar hasta que la operación que se esté realizando termine, antes de asignar la CPU a otro proceso. Esto aumenta la **latencia de asignación**, dado que algunas llamadas al sistema pueden ser muy complejas y requerir mucho tiempo para completarse.

Con el objetivo de resolver este problema se han desarrollado diversas alternativas para que el código del núcleo sea expropiable.

Puntos de expropiación

Una posibilidad es introduciendo **puntos de expropiación** en diversos lugares «seguros» dentro del código. En dichos puntos se comprueba si algún proceso de prioridad más alta está en la cola de

preparados. En caso de que sea así, se expropia la CPU al proceso actual y se le asigna al proceso de más alta prioridad.

Debido a la función que realizan los puntos de expropiación, solo pueden ser colocados en lugares seguros del código del núcleo. Es decir, lugares donde no se interrumpe la modificación de estructuras de datos. Sin embargo, esto limita el número de puntos que pueden ser colocados, por lo que la latencia de asignación puede seguir siendo muy alta para algunas operaciones muy complejas del núcleo.

Núcleo expropiable

Otra posibilidad es diseñar un **núcleo completamente expropiable**.

Puesto que en este caso la ejecución de cualquier operación en el núcleo puede ser interrumpida en cualquier momento por procesos de mayor prioridad que el que actualmente tiene asignada la CPU, es necesario proteger las estructuras de datos del núcleo con mecanismos de sincronización. Esto hace que el diseño de un núcleo de estas características sea mucho más complejo.

Microsoft Windows —desde Windows NT— Linux —desde la versión 2.6— Solaris y NetBSD son algunos ejemplos de sistemas operativos con núcleos expropiables. En el caso concreto de Solaris la latencia de asignación es inferior a 1 ms, mientras que con la expropiación del núcleo desactivada esta puede superar los 100 ms

Expropiación en el núcleo de Linux

Lamentablemente, conseguir baja latencia de asignación no tiene coste cero. El hecho de que el núcleo sea expropiable aumenta el número de cambios de contexto, lo que reduce el rendimiento del sistema a cambio de un menor tiempo de respuesta. Esto resulta muy interesante para aplicaciones de tiempo real, multimedia y sistemas de escritorio, pero es poco adecuado para servidores y computación de altas prestaciones.

Por eso desde Linux 2.6 se puede compilar el núcleo con diferentes niveles, de lo expropiable que es el núcleo.

En la configuración por defecto `PREEMPT_NONE`, el núcleo tiene algunos **puntos de expropiación**, de tal forma que es ideal para servidores y sistemas cómputo de altas prestaciones. Con `PREEMPT_VOLUNTARY` —el siguiente nivel— se añaden muchos más **puntos de expropiación** con el objeto de reducir la latencia, mejorando el tiempo de respuesta en sistemas de escritorio.

Finalmente, activando `PREEMPT` el núcleo se vuelve **completamente expropiable** —excepto en algunas secciones críticas—. Esto es ideal para sistemas de escritorio o sistemas empotrados con requisitos de latencia en el rango de los milisegundos.

Inversión de prioridad

Supongamos que en un núcleo completamente expropiable, un proceso de baja prioridad es interrumpido porque hay un proceso de alta prioridad en la cola de preparados. Y que esto ocurre mientras el primero accede a una importante estructura de datos del núcleo.

Durante su ejecución, el proceso de alta prioridad podría intentar acceder a la misma estructura que trataba de manipular el proceso de baja prioridad cuando fue interrumpido. Debido al uso de mecanismos de sincronización, el proceso de alta prioridad se quedaría bloqueado y tendría que abandonar la CPU a la espera de que el de baja, libere el acceso al recurso. Sin embargo, este último tardará en ser asignado a la CPU mientras haya algún otro proceso de alta prioridad en la cola de preparados.

Al hecho de que un proceso de alta prioridad tenga que esperar por uno de baja se le conoce como **inversión de la prioridad**. Para resolverlo se utiliza un **protocolo de herencia de la prioridad**, donde un proceso de baja prioridad hereda la prioridad del proceso de más alta prioridad que espera por un recurso al que el primero está accediendo. En el momento en que el proceso de baja prioridad libere el acceso a dicho recurso, su prioridad retornará a su valor original.

14.7. Planificación en sistemas multiprocesador

Para tratar el problema de la planificación en los sistemas multiprocesador nos limitaremos al caso de los **sistemas homogéneos**. En dichos sistemas los procesadores son idénticos, por lo que, en cualquiera de ellos, puede ejecutar cualquier proceso. Esto es bastante común y simplifica el problema de la planificación.

Un ejemplo de lo contrario a un sistema homogéneo —un sistema heterogéneo— se puede observar en los PC modernos, donde muchos disponen tanto de una CPU como de una GPU, especializada en el procesamiento de gráficos y en las operaciones vectoriales con números enteros y de coma flotante.

Aun así, no debemos olvidar que incluso en el caso de los sistemas homogéneos pueden aparecer limitaciones en la planificación. Por ejemplo, los procesadores SMT (*Simultaneous Multithreading*) permiten la ejecución concurrente de varios hilos de ejecución como si de varias CPU se tratara. Sin embargo, al no disponer cada hilo de una CPU completa, es posible que algunos deban esperar a que algún otro libere unidades de ejecución de la CPU que le son necesarias. Eso debe ser tenido en cuenta por el planificador con el fin de optimizar el rendimiento del sistema.

La tecnología *Hyper-threading* disponible en algunos procesadores de Intel es una implementación de la tecnología *Simultaneous Multithreading*. Permite que cada núcleo de procesador que está presente físicamente, el sistema operativo lo gestione como dos núcleos virtuales —o lógicos— y repartir entre ellos las tareas cuando es posible.

Al margen de estas cuestiones, según el tipo de procesamiento, existen diversas posibilidades a la hora de enfrentar el problema de la planificación en un sistema multiprocesador (véase el Apartado 2.4).

14.7.1. Multiprocesamiento asimétrico

Cuando utilizamos **multiprocesamiento asimétrico** todas las decisiones de planificación, procesamiento de E/S y otras actividades son gestionadas por el núcleo del sistema ejecutándose en un único procesador: el **servidor** o **maestro**. El resto de procesadores se limitan a ejecutar código de usuario, que les es asignado por ese procesador **maestro**.

Este esquema es sencillo, puesto que evita la necesidad de compartir estructuras de datos entre el código que se ejecuta en los diferentes procesadores.

14.7.2. Multiprocesamiento simétrico

Cuando utilizamos **multiprocesamiento simétrico** o **SMP**, cada procesador ejecuta su propia copia del núcleo del sistema operativo y se autoplanifica mediante su propio planificador de CPU. En estos sistemas nos podemos encontrar con varias alternativas.

Con una cola de preparados común

Algunos sistemas disponen de una cola de preparados común para todos los procesadores. Puesto que se mira en una única cola, todos los procesos pueden ser planificados en cualquier procesador.

Este esquema requiere el uso mecanismos de sincronización para controlar el acceso concurrente de los núcleos a las colas. En caso contrario, varios procesadores podrían escoger y ejecutar el mismo proceso a la vez.

Muchos sistemas operativos modernos implementan el esquema SMP con una cola de preparados común. Esto incluye Microsoft Windows NT/2000/XP, Solaris, macOS y versiones anteriores a Linux 2.6.

Es importante recordar que en esos sistemas operativos, lo que se planifica en las distintas CPU usando alguna de estas estrategias, son los hilos y no los procesos.

Sin embargo, esta solución presenta algunos inconvenientes:

- La posibilidad de que un proceso se pueda ejecutar en cualquier CPU —aunque parezca beneficiosa— es negativa desde el punto de vista de que dejan de ser útiles las cachés de los procesadores, penalizando notablemente el rendimiento del sistema. Por eso, la mayoría de los sistemas operativos de este tipo evitan, en lo posible, la migración de procesos de un procesador a otro. A esto se lo conoce como asignar al proceso **afinidad al procesador**.
- Los mecanismos de sincronización requeridos para controlar el acceso a la cola de preparados pueden mantener a los procesadores mucho tiempo desocupados —mientras esperan— en sistemas con un gran número de procesadores y con muchos procesos en la cola de preparados.

Con una cola para cada procesador

Cada vez más sistemas modernos están optando por utilizar el esquema SMP con una cola de preparados por procesador. De esta manera, al no utilizar mecanismos de sincronización, se eliminan los tiempos de espera para acceder a la cola de preparados y escoger un nuevo proceso.

El mayor inconveniente de esta solución es que puede generar desequilibrios entre los procesadores, ya que un procesador puede acabar desocupado —con su cola de preparados vacía— mientras otro está muy ocupado. Con el fin de que esto no suceda, es necesario que el sistema disponga de algunos mecanismos de **balanceo de carga**:

- En la **migración comandada** o *push migration*, una tarea específica —que se ejecuta con menor frecuencia que el planificador de la CPU— estima la carga de trabajo de cada CPU y en caso de

encontrar algún desequilibrio mueve algunos procesos de la cola de preparados de unos procesadores a la de los otros.

- En la **migración solicitada** o *pull migration*, un procesador inactivo extrae de la cola de preparados de un procesador ocupado alguna tarea que esté esperando.

Tanto el planificador de Linux 2.6 y posteriores, como el planificador ULE de FreeBSD, implementan ambas técnicas. Mientras que en Microsoft Windows, a partir de Windows Vista también se utiliza una cola de preparados por procesador, pero solo implementa la **migración solicitada**.

Parte IV: Gestión de la memoria

Para que un programa pueda ejecutarse, su código y datos debe ser cargados en la memoria principal. Por tanto, para poder ejecutar múltiples programas que mantenga ocupada la CPU y el resto de recursos del sistema, es necesario aprovechar todo lo posible la memoria principal. Estudiaremos varias estrategias, aunque sobre todas ellas destaca la **memoria virtual**, implementada por los sistemas modernos, ya que permite ejecutar un programa sin tener que cargarlo completamente en memoria, lo que deja más espacio para cargar y ejecutar otros programas.

Aparte del uso eficaz de la memoria, también consideraremos su **protección**. Como varios procesos se cargan simultáneamente en la memoria principal —a la que tiene acceso directo la CPU— sin mecanismos de protección cualquier proceso podría modificar la memoria del resto de procesos e incluso del sistema operativo. Para evitarlo, en los sistemas modernos los procesos se aíslan unos de otros en sus propios **espacios de direcciones virtuales**. Incluso dentro de la memoria de cada proceso se ajustan los permisos para que solo se puedan ejecutar las regiones que contienen código o solo se pueda escribir en aquellas zonas que almacenan variables que pueden ser modificadas.

Chapter 15. Memoria principal

Tiempo de lectura: 32 minutos

La memoria es un recurso central para el funcionamiento de un sistema operativo moderno, puesto que es el único medio de almacenamiento al que la CPU puede acceder directamente. Por ello, para que un programa pueda ser ejecutado, debe ser cargado en la memoria desde el disco y creadas o modificadas las estructuras internas del sistema operativo necesarias para convertirlo en un proceso. Además, dependiendo de la forma en la que se gestiona la memoria, los procesos —o partes de los mismos— pueden moverse de la memoria al disco —y viceversa— durante su ejecución, con el objetivo de ajustar las necesidades de memoria para mantener el **uso de la CPU** lo más alto posible.

Como comentamos en el Apartado 2.1.2, en los **sistemas multiprogramados** existe una **cola de entrada**, que se define como aquella formada por el conjunto de procesos en disco que esperan para ser cargados en la memoria para su ejecución.

Por tanto, el procedimiento normal de ejecución de un programa en dichos sistemas es:

1. Seleccionar un proceso de la cola de entrada y cargarlo en la memoria.
2. Mientras el proceso se ejecuta, este accede a instrucciones y datos de la memoria.
3. Finalmente, el proceso termina y su espacio en memoria es marcado como disponible.

En los sistemas de propósito general modernos —desde los **sistemas de tiempo compartido** y los primeros **sistemas de escritorio**— no existe **cola de entrada**, por lo que los programas se cargan inmediatamente en memoria cuando los usuarios solicitan su ejecución. Excepto por eso, el procedimiento normal de ejecución de un programa es similar al de los **sistemas multiprogramados**.

15.1. Etapas de un programa de usuario

En la mayor parte de los casos, un programa de usuario debe pasar por diferentes etapas —algunas de las cuales son opcionales— antes de ser ejecutado (véase la Figura 48).

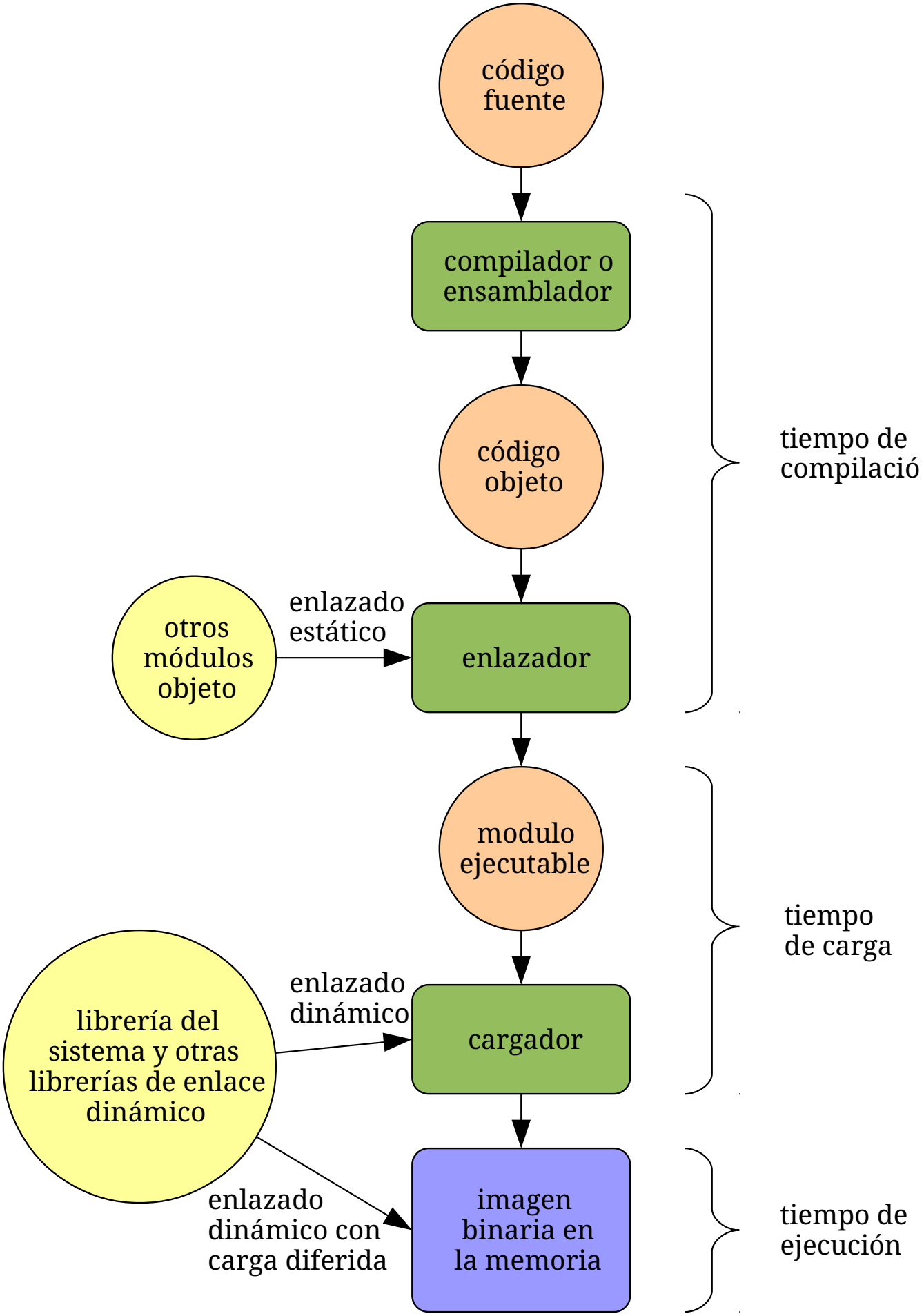


Figura 48. Etapas de procesamiento de un programa de usuario.

Los archivos de **código fuente** del programa son compilados por el compilador, generando un

archivo de **código objeto** —con extensiones `.o` u `.obj`— para cada uno.

Todos los archivos de **código objeto** son unidos por el enlazador para crear el archivo **ejecutable**, en una fase que se denomina **enlazado estático**. En esta fase también se pueden incorporar al ejecutable **librerías de enlace estático** —con extensiones `.a` o `.lib`— con **código objeto** que ha sido empaquetado para ser reutilizado en múltiples ejecutables.

El compilador y el enlazador suelen ser dos programas independientes, aunque en ocasiones el compilador se haga cargo de ambas fases por comodidad. Por ejemplo, en los sistemas GNU el compilador gcc por defecto genera el **código objeto** en archivos temporales, luego invoca al enlazador ld para crear el **ejecutable** y finalmente elimina los archivos temporales.

En proyectos grandes suele ser más interesante usar ambas herramientas por separado para reducir el tiempo de compilación. El compilador genera los archivos de **código objeto**, que se conservan entre compilaciones. Así, cada vez que se quiere generar una nueva versión del ejecutable, solo es necesario compilar los archivos de **código fuente** que hayan cambiado y luego enlazar juntos todos los archivos de **código objeto**.

Al crear el **ejecutable** se pueden guardar en él dependencias respecto a librerías que se enlazarán posteriormente, durante la carga o ejecución del programa, en una fase denominada **enlazado dinámico**.

En el momento en el que se va a ejecutar el programa, cuando está construyendo la imagen binaria del proceso en la memoria; el sistema operativo examina estas dependencias, carga las **librerías de enlace dinámico** indicadas —con extensiones `.so`, `dylib` o `.dll`— y resuelve las referencias del programa sus variables y funciones. Las **librerías de enlace dinámico** contienen **código objeto**, enlazado en un formato especial de **ejecutable** diseñado para contener partes compartidas entre archivos ejecutables.

Este proceso puede ocurrir mientras se carga el **ejecutable** —como se ha descrito— o cuando el programa usa por primera vez un elemento de las **librerías de enlace dinámico**. También es común que el sistema ofrezca funciones para que los programas puedan cargar manualmente e invocar funciones de **librerías de enlace dinámico**. Esto es muy útil para crear programas que se puedan mejorar por medio de extensiones o *plugins*.

Tabla 13. Extensiones de archivos de programas.

	UNIX, Linux y otros sistemas estilo UNIX	macOS	Microsoft Windows
Código objeto	`.o`	`.o`	`.obj`
Librería de enlace estático	`.a`	`.a`	`.lib`
Ejecutable			`.exe`
Librería de enlace dinámico	`.so`	`.so`, `.dylib`	`.dll`

	UNIX, Linux y otros sistemas estilo UNIX	macOS	Microsoft Windows
Formato de ejecutables y librerías de enlace dinámico	Executable and Linkable Format (ELF)	Mach-O	Portable Executable (PE)

15.2. Reubicación de las direcciones

La mayor parte de los sistemas permiten que un proceso de usuario resida en cualquier parte de la memoria física. Así, aunque el espacio de direcciones del sistema comience en `0x00000000`, la primera dirección del proceso de usuario no tiene por qué ser esa.

En cada una de las etapas vistas en el Apartado 15.1 las direcciones pueden representarse de formas distintas, por lo que en cada paso es necesario reasignar las direcciones usadas en una etapa en direcciones de la siguiente.

Por ejemplo, en el código fuente de un programa las direcciones son generalmente simbólicas, como los nombres de las variables y las funciones. A continuación, un compilador suele reasignar esas direcciones simbólicas en **direcciones reubicables** del estilo de «120 bytes desde el comienzo del módulo». Finalmente —el enlazador— que genera el ejecutable— o el cargador —que carga el programa en la memoria— convierte esas **direcciones reubicables** en **direcciones absolutas**, como `0x00210243`.

Por tanto, en cada etapa se traducen las direcciones de un espacio de direcciones en el siguiente. Sin embargo, para que al final el programa pueda ser ejecutado, es necesario que tanto a los datos como a las instrucciones se les reasignen en algún momento a **direcciones absolutas** de la memoria. Esto puede ocurrir en **tiempo de compilación**, **tiempo de carga** o **tiempo de ejecución**

15.2.1. Reubicación en tiempo de compilación

Si durante la compilación o el enlazado se conoce el lugar de la memoria donde va a ser ejecutado el proceso, se puede generar directamente código con **direcciones absolutas** o **código absoluto**.

Eso significa que si en algún momento la dirección de inicio donde es cargado el programa cambia, es necesario recompilar el código fuente del programa para poder ejecutarlo en la nueva ubicación.

Un ejemplo son los ejecutables con formato COM del sistema operativo MS-DOS. Estos ejecutables no eran reubicables, aunque podían ponerse en distintas ubicaciones de la memoria gracias a la segmentación de memoria de la familia Intel x86.

15.2.2. Reubicación en tiempo de carga

Si no se conoce durante la compilación el lugar donde va a residir un programa cuando sea ejecutado, el compilador y el enlazador deben generar ejecutables con **código reubicable**.

En este tipo de código se utilizan **direcciones reubicables**, de manera que se retrasa su asignación a **direcciones absolutas** hasta el momento de la carga del programa. Esto permite que un

programa pueda residir en cualquier parte de la memoria física, cargando los procesos donde más convenga para maximizar el aprovechamiento de la misma.

Para generar **código reubicable**, por lo general, el compilador genera **código independiente de la posición** o **PIC** (*Position-Independent Code*). Este tipo de código se puede ejecutar adecuadamente y sin modificaciones independientemente del lugar de la memoria donde esté ubicado, porque utiliza direcciones relativas.

Lamentablemente, esto puede limitar las características de la CPU que puede utilizar el compilador o, a veces, las instrucciones que usan direcciones absolutas son más rápidas que las que usan direcciones relativas, aunque en los procesadores modernos la diferencia apenas es perceptible.

Por ejemplo, las CPU x86-64 soportan un modo de direccionamiento en el que las direcciones son relativas a la dirección en el contador de programa. Esto simplifica generar código reubicable eficiente. Sin embargo, en las CPU x86 anteriores, las instrucciones de salto podían ser relativas al contador de programa, pero no ocurría así con aquellas destinadas a acceder a los datos del programa.

Cuando no se puede o no es eficiente generar **código independiente de la posición** se puede recurrir al uso de **tablas de reubicación** en tiempo de carga. En este caso el compilador y el enlazador generan:

1. Código con direcciones relativas a cierta dirección fija del ejecutable —como el comienzo de la sección de código— o direcciones absolutas calculadas bajo la suposición de que el ejecutable se va a poder cargar en cierta dirección concreta de la memoria, que suele guardarse en la cabecera del ejecutable.
2. Una **tabla de reubicaciones** que se almacena en el mismo ejecutable. Esta tabla contiene punteros a las ubicaciones en el código del ejecutable de las direcciones que deben reubicarse al cargarlo.

Durante la carga, el cargador del sistema operativo, una vez ha copiado a la memoria el contenido del ejecutable y conoce la ubicación definitiva del programa, recorre la **tabla de reubicaciones** para buscar las **direcciones reubicables** y actualizarlas.

Direcciones en el código objeto en Linux x86-64

Supongamos que en un sistema Linux x86-64 tenemos el siguiente código en un archivo de nombre `test.c`:

```
long i = 10;

int test()
{
    i = 12;
    return 0;
}

int main()
{
```

```
    i = 11;
    return test();
}
```

En lugar de compilarlo para generar el ejecutable final, podemos usar la opción `-c` del compilador gcc para que solo genere el archivo de **código objeto** correspondiente. Después podemos usar el comando objdump para examinar el **código objeto**:

```
$ gcc -c test.c
$ objdump -d test.o

test.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <test>:                                    ①
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 c7 05 00 00 00 00    movq   $0xc,0x0(%rip)       ③
   f:   0c 00 00 00
  13:   b8 00 00 00 00          mov    $0x0,%eax
  18:   5d                      pop    %rbp
  19:   c3                      retq

000000000000001a <main>:                                    ①
  1a:   f3 0f 1e fa             endbr64
  1e:   55                      push   %rbp
  1f:   48 89 e5                mov    %rsp,%rbp
  22:   48 c7 05 00 00 00 00    movq   $0xb,0x0(%rip)       ③
  29:   0b 00 00 00
  2d:   b8 00 00 00 00          mov    $0x0,%eax
  32:   e8 00 00 00 00          callq  37 <main+0x1d>       ②
  37:   5d                      pop    %rbp
  38:   c3                      retq
```

① El archivo de **código objeto** se divide en varias secciones o segmentos. En el **segmento de código** o **.text** se almacenan las funciones compiladas del código original. Cada función tiene una ubicación, que es relativa a su posición dentro del segmento de código en el archivo.

Obviamente, estas no son las direcciones definitivas en las que se ejecutarán estas funciones cuando el ejecutable sea cargado en la memoria, por lo que este **código objeto** generado por el compilador debe ser **código reubicable**.

② `callq` es la instrucción encargada de invocar a `test()` desde `main()`, pero si ejecutamos este código en su estado actual, la ejecución saltaría a la instrucción en 0x37, es decir, a la siguiente tras `callq`. Esto no es un error del compilador, sino que el compilador no ha

puesto la dirección correcta porque aún desconoce cuál será la ubicación definitiva de `test()`.

En x86-64 las instrucciones de salto como `callq` hacen saltos relativos al contador de programa —almacenado en un registro llamando `rip`—. Todas estas instrucciones llevan un desplazamiento que se suma al valor actual de `rip`, que contiene la dirección en la memoria de la siguiente instrucción. Como el compilador no sabe donde estará `test()`, deja este desplazamiento a 0 —por eso los bytes '00 00 00 00' en la codificación de la instrucción `callq`— por lo que el salto será a la siguiente instrucción.

③ En el acceso a la variable `i` para cambiar su valor, ocurre algo parecido. La dirección se escribe relativa al contador de programa `rip`, pero como el compilador no sabe la posición definitiva de la variable, deja el desplazamiento a 0. Por eso el desensamblado indica lo de `0x0(%rip)`, que quiere decir: dirección desplazada 0 respecto a `rip`.

Durante el enlazado, para generar el ejecutable definitivo, es necesario saber dónde están en el código las direcciones que se deben reubicar. Para eso el archivo de **código objeto** lleva una **tabla de reubicación**, que también se puede leer con objdump:

```
$ objdump -r test.o
test.o:     file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
000000000000000b R_X86_64_PC32     i-0x0000000000000008
0000000000000025 R_X86_64_PC32     i-0x0000000000000008
0000000000000033 R_X86_64_PLT32    test-0x0000000000000004
```

En nuestro ejemplo la tabla contiene tres entradas, dos para las dos instrucciones que modifican la variable `i` y una para la instrucción `callq` que sirve para invocar `test()`. La primera columna indica la ubicación en **.text** de la dirección que se debe reubicar. Mientras la segunda columna especifica el tipo de reubicación y la tercera contiene información adicional que hace falta para realizar la reubicación.

Por ejemplo, la primera entrada del ejemplo indica que debe desplazarse en `0x08` la dirección a reubicar en `0x0b` —el acceso a `i` en `test()` para que apunte a la dirección de la variable `i`.

15.2.3. Reubicación en tiempo de ejecución

Si un proceso puede ser movido durante su ejecución de un lugar de la memoria a otro, la reubicación de direcciones debe ser retrasada hasta el momento de la ejecución de cada instrucción del programa.

Para que esto sea posible, necesitamos disponer de hardware especial que suele estar presente en la mayor parte de las CPU modernas, por lo que la inmensa mayoría de los sistemas operativos de propósito general modernos utilizan este método. De él hablaremos en el Apartado 15.3.

15.3. Espacio de direcciones virtual frente a físico

En el Apartado 7.3 vimos en los sistemas operativos modernos, como medida de protección, los procesos no tienen acceso libre a la memoria física.

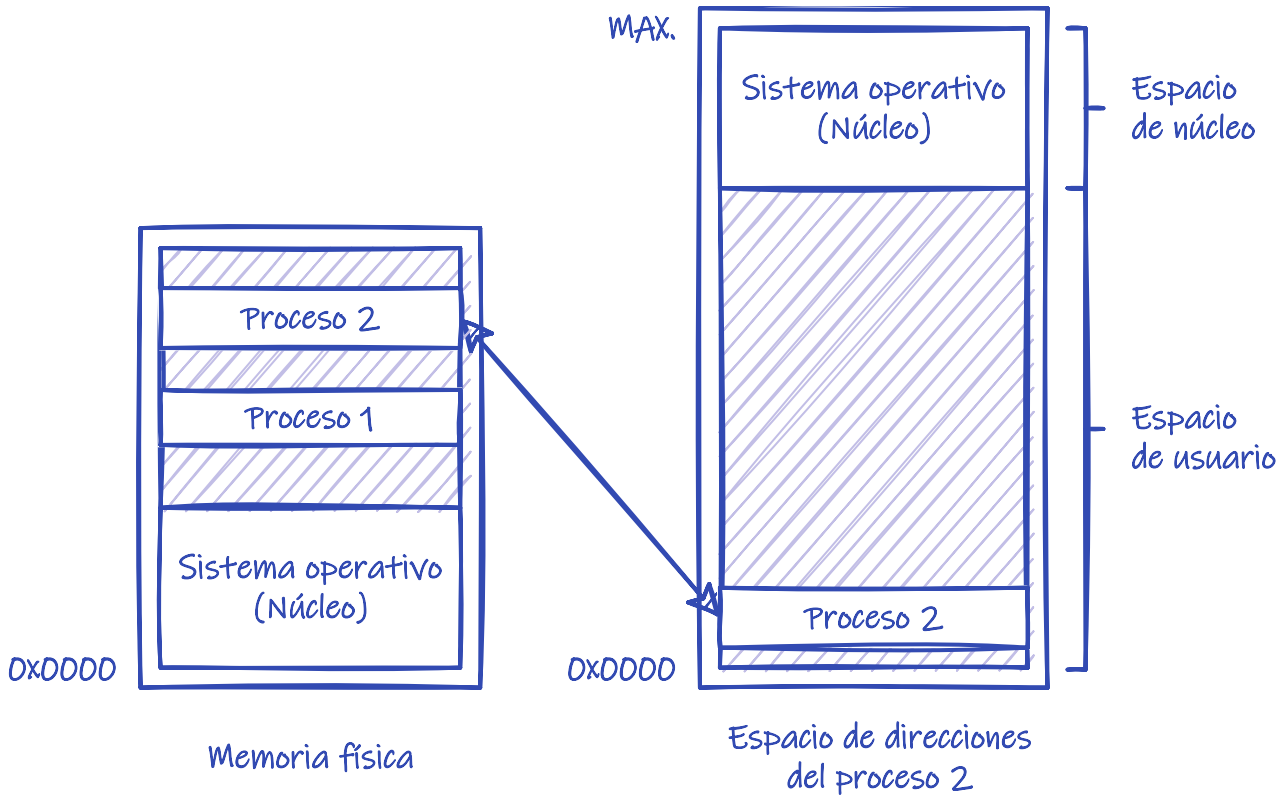


Figura 49. Mapeo de la memoria física en el espacio de direcciones virtual de un proceso.

En lugar de eso el sistema operativo —asistido por la **MMU** (*Memory-Management Unit*)— proporciona a cada proceso un **espacio de direcciones virtual** que ofrece una «vista» privada de la memoria, similar a la que tendrían si cada uno de los procesos estuviera siendo ejecutando en solitario (véase la Figura 49). Es durante los accesos a la memoria principal en tiempo de ejecución, cuando estas **direcciones virtuales** son convertidas por la **MMU** en las **direcciones físicas**, con las que realmente se accede a la memoria. El **espacio de direcciones físico** es el conjunto de direcciones físicas que corresponden a todas las direcciones virtuales de un **espacio de direcciones virtual** dado.

El mecanismo de protección descrito es una forma muy común de **reubicación de las direcciones en tiempo de ejecución**, que está presente en la mayor parte de los sistemas operativos de propósito general modernos. Pero, aparte de la protección de la memoria, algunas otras características de dicho mecanismo son:

- Los procesos pueden ser cargados en cualquier zona libre de la memoria física e incluso movidos de una región a otra durante la ejecución de los procesos, puesto que la transformación de las **direcciones virtuales** en **direcciones físicas** se realiza durante la ejecución de cada instrucción.
- El código generado por el compilador puede ser **código absoluto**, puesto que de antemano se sabe que todas las ubicaciones del espacio de direcciones virtual van a estar disponibles.

  Lo común es que los programas se ubiquen en una dirección fija en la parte baja del espacio de

direcciones virtual. Por ejemplo, empezando en la dirección `0x00400000`, dejando libres los primeros 4 MiB del **espacio de direcciones virtual**.

Los programas pueden ubicarse en cualquier lugar del **espacio de direcciones virtual**, pero no ocurre lo mismo con las **librerías de enlace dinámico**, cuya posible ubicación va a depender del espacio ocupado por el programa y por otras **librerías de enlace dinámico**. Por tanto, como veremos en detalle más adelante, estas librerías deben ser reubicables en tiempo de carga.

- Se puede reducir el consumo de memoria principal compartiendo las regiones de memoria física asignadas al código y los datos de solo lectura de los procesos de un mismo programa.

El código de un programa suele contener direcciones tanto para los saltos como para el acceso a los datos. Al ubicar los programas en las mismas regiones de los espacios de direcciones virtuales de sus procesos, nos estamos asegurando de que el código en memoria de los procesos de un mismo programa es el mismo —pues todos usan las mismas direcciones virtuales absolutas— por lo que se puede compartir la memoria física que ocupan.

Direcciones en ejecutables en Linux x86-64

En Linux cada proceso tiene su propio espacio de direcciones virtual, por lo que los ejecutables pueden generarse para ser cargados en una dirección fija, ya que es seguro que no habrá otro ocupando la misma dirección.

Si la dirección donde se va a ejecutar el programa puede fijarse de antemano, los ejecutables pueden contener **código absoluto**. Por eso en Linux x86-64 el enlazador coge el **código objeto** que va a formar parte del ejecutable, lo reubica asumiendo que el ejecutable se cargará en cierta dirección de la memoria y genera un ejecutable con **código absoluto**, sin **tablas de reubicación**.

Por ejemplo, si compilamos completamente el `test.c` anterior y preguntamos por la **tabla de reubicación**, veremos que está vacía:

```
$ gcc -o test test.c
$ objdump -r test
test.o:     file format elf64-x86-64
```

Si desensamblamos el código, veremos que han cambiado algunas cosas en `main()` y a `test()` respecto a lo que había en el **código objeto**.

```
$ objdump -d test

test.o:     file format elf64-x86-64


Disassembly of section .text:
```

```
...

0000000000001129 <test>:                                        ②
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
    1131:       c7 05 d9 2e 00 00 03    movl   $0x3,0x2ed9(%rip)   ①
    1138:       00 00 00
    113b:       b8 00 00 00 00          mov    $0x0,%eax
    1140:       5d                      pop    %rbp
    1141:       c3                      retq

0000000000001142 <main>:
    1142:       f3 0f 1e fa             endbr64
    1146:       55                      push   %rbp
    1147:       48 89 e5                mov    %rsp,%rbp
    114a:       c7 05 c0 2e 00 00 01    movl   $0x1,0x2ec0(%rip)   ①
    1151:       00 00 00
    1154:       b8 00 00 00 00          mov    $0x0,%eax
    1159:       e8 cb ff ff ff          callq  1129 <test>         ②
    115e:       5d                      pop    %rbp
    115f:       c3                      retq

...
```

① Las instrucciones que acceden a la variable `i` ahora tienen el desplazamiento adecuado respecto al contador de programa `rip` para acceder a dicha variable.

② La instrucción de salto tiene un desplazamiento que sumado al valor de `rip` lleva directamente al comienzo del código de la función `test()`.

15.4. Enlazado dinámico y librerías compartidas

Como hemos comentado anteriormente, fundamentalmente existen dos tipos de enlazado:

- En el **enlazado estático**, las librerías del sistema y otros módulos son combinados por el enlazador para formar la imagen binaria del programa que es almacenada en disco. Algunos sistemas operativos —como MS-DOS— solo soportan este tipo de enlazado.
- En el **enlazado dinámico**, este se pospone hasta la carga o la ejecución_ (véase la Figura 48).

Generalmente el enlazado dinámico ocurre durante la carga del programa:

1. Durante la carga del ejecutable se comprueban las dependencias del mismo. Estas se almacenan en el mismo archivo en disco que dicho ejecutable.
2. Las librerías a enlazar se cargan y ubican en el espacio de direcciones virtual creado para el nuevo proceso.
3. Finalmente, las referencias del programa a las funciones de cada una de las librerías cargadas se actualizan con la dirección en memoria de las mismas. Así la invocación de las funciones por

parte del programa se puede realizar de forma transparente, como si siempre hubieran formado parte del mismo.

Cuando el enlazado se va a realizar en tiempo de ejecución se habla de **enlazado dinámico con carga diferida**. En ese caso el procedimiento es el siguiente:

1. Durante el enlazado estático del ejecutable se pone un *stub* a cada referencia a alguna función de la librería que va a ser enlazada dinámicamente.
2. Si durante la ejecución del programa alguna de dichas funciones es invocada, se ejecuta el *stub*. El *stub* es una pequeña pieza de código que sabe como cargar la librería, si no ha sido cargada previamente, y cómo localizar la función adecuada en la misma.
3. Finalmente, el *stub* se sustituye a sí mismo con la dirección de la función y la invoca. Esto permite que la siguiente ejecución de la función no incurra en ningún coste adicional.

Sin esta habilidad, cada programa en el sistema debería tener, por ejemplo, una copia de la librería del sistema incluida en su ejecutable. Esto significa un desperdicio de espacio libre en disco y de memoria principal. Además, este esquema facilita la actualización de las librerías, puesto que los programas pueden utilizar directamente las versiones actualizadas sin necesidad de volver a ser enlazados.

15.4.1. Reubicación de las direcciones

Durante la compilación de una **librería dinámica** no se conoce la región que va a ocupar, dentro de los espacios de direcciones virtuales de los distintos procesos que la van a utilizar, por lo que es necesario generar **código reubicable**.

Atendiendo a lo visto en Apartado 15.2.2 existen fundamentalmente dos estrategias:

- El compilador puede generar **código independiente de la posición** (PIC). Esto permite reducir el consumo de memoria principal compartiendo las regiones de memoria física asignadas al código de una misma librería en los distintos procesos que la utilizan, pues en todas el código será exactamente el mismo.
- En los sistemas operativos donde no se usa código PIC, el compilador debe generar código reubicable con **tablas de reubicación**, para que la reubicación de las direcciones virtuales se haga en tiempo de carga. Esto aumenta el tiempo de carga de las librerías y solo permite que compartan memoria física partes de la librería que sigan siendo iguales tras la reubicación de las direcciones.

15.4.2. Librerías compartidas

Habitualmente las librerías incluyen información acerca de su versión. Esta información puede ser utilizada para evitar que los programas se ejecuten con versiones incompatibles de las mismas, o para permitir que haya más de una versión de cada librería en el sistema. Así los viejos programas se pueden ejecutar con las viejas versiones de las librerías —o con versiones actualizadas aunque compatibles— mientras los nuevos programas se ejecutan con las versiones más recientes e incompatibles con los viejos programas.

A este sistema se lo conoce como **librerías compartidas**.

Para establecer correctamente la compatibilidad entre librerías y programas, es conveniente y bastante común que los desarrolladores de las **librerías compartidas** utilicen **versionado semántico**. El **versionado semántico** es un convenio por el cual el número de versión suele venir indicado por tres número separados por puntos —como, por ejemplo: 5.3.4— de tal forma que:

- El primer número es incrementado cuando ocurre un cambio drástico en la librería, de tal forma que seguramente los programas hechos para versiones anteriores no van a ser compatibles con la nueva versión.
- El segundo número es incrementado cuando se añade alguna nueva característica o se modifica alguna ya existente, pero el cambio no debe romper los programas hechos para la versión anterior.
- El último número es incrementado al corregir errores, pero no se rompe la compatibilidad con versiones previas.

De esta forma, los desarrolladores pueden enlazar sus programas contra la versión 5 de la librería, por ejemplo, sabiendo que así funcionarán con cualquier versión actualizada o corregida —como la 5.1, 5.2.9 o 5.10.1— pero nunca podrán ejecutarse con versiones que tienen cambios mayores y posiblemente incompatibles —como la versión 6.1 o la 2.10—.

Enlazado dinámico en Linux x86-64

Volviendo al programa de ejemplo `test.c`, podemos obtener fácilmente las librerías compartidas de las que depende usando el comando ldd:

```
$ ldd test
linux-vdso.so.1 (0x00007fffd833e000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f47c20c9000)   ①
/lib64/ld-linux-x86-64.so.2 (0x00007f47c22cf000)                    ②
```

① El ejecutable tiene que estar enlazado con la librería del sistema —llamada `libc` en los sistemas POSIX— para que pueda acceder a los servicios del sistema.

`libc.so.6` es el nombre por el que se buscará el archivo que contiene la librería compartida de la librería del sistema Si se hubiera usado solamente `libc.so`, el ejecutable podría intentar usar cualquier versión de `libc`, pero al incluir el primer número de versión en el nombre, nos aseguramos de que el ejecutable solo usará versiones 6 de la librería del sistema. Así se evita ejecutar el programa con versiones que pueden ser incompatibles.

② La librería ld se encarga de las cuestiones relacionadas con el enlazado y carga dinámica de librerías.

Si estás librerías no estuvieran disponibles al intentar ejecutar el programa, la ejecución fallaría, ya que son un requisito obligatorio. Sin embargo, el enlazado dinámico no es algo que solo pueda configurarse durante la compilación para que ocurra automáticamente durante la carga del ejecutable. Los sistemas operativo modernos proporcionan funciones para que un proceso pueda cargar cualquier y descargar librería durante la ejecución y acceder a sus variables e invocar sus funciones. En los sistemas POSIX estas funciones las

proporciona la librería ld.

Esto es muy útil cuando se quiere que ciertas librerías sean opcionales. Es decir, que no sea necesario tenerlas instaladas en el sistema donde se va a ejecutar el programa, de tal forma que si no están disponibles, el programa funcione con normalidad, pero si están instaladas, el programa aproveche la funcionalidad adicional que proporcionan. Así es como navegadores, editores y otros programas implementan extensiones y *plugins*, con los que el usuario puede extender la funcionalidad del programa original sin recompilarlo. Estas extensiones generalmente se implementan como librerías compartidas que el programa busca y carga durante su ejecución.

Por lo general, la mayor parte de las librerías que usan los programas son enlazadas dinámicamente, pero el enlazador siempre tiene que incluir en el ejecutable algunas librerías enlazadas estáticamente que son importantes para la ejecución del programa. Por eso al ejecutar `objdump -d test` vemos funciones que no estaban ni en `test.c ni en el archivo de código objeto `test.o`. La librería que vemos incluida en el ejecutable es la **librería de tiempo de ejecución** de C.

Todo lenguaje de programación —excepto ensamblador, obviamente— necesita incluir en sus ejecutables una **librería de tiempo de ejecución** o **librería de runtime** con rutinas de bajo nivel que sirven de apoyo para implementar algunas características del lenguaje y del entorno de ejecución.

En Linux y otros sistemas POSIX, la posición dónde debe iniciarse la ejecución se marca con la etiqueta `_start` en el ejecutable. Se trata de código de la **librería de tiempo de ejecución** que inicializa el entorno que espera el programa en C, obtiene del sistema operativo los argumentos de la línea de comandos e invoca a la función `main()` de nuestro programa con ellos. Otras funcionalidades que típicamente implementa la **librería de tiempo de ejecución** son operaciones básicas de gestión de la memoria, manejo de excepciones, comprobaciones de límites en los _arrays_ o comprobaciones dinámicas de tipos —por ejemplo, para implementar el operador `dynamic_cast` en C++—.

Finalmente, no debemos confundir la **librería de tiempo de ejecución** con la **librería estándar**. En lenguajes como C o C++ se distingue claramente entre lo que es el lenguaje de programación y la librería estándar que suele acompañarlos. En ese sentido, la **librería de tiempo de ejecución** implementa lo mínimo requerido para la ejecución de programa en esos lenguajes. Mientras que si además queremos usar funcionalidades de su **librería estándar**, también tendremos que enlazarla, generalmente de forma dinámica.

15.5. Asignación contigua de memoria

Como vimos en el Apartado 7.3, la memoria principal debe acomodar tanto el sistema operativo como a los diferentes procesos de los usuarios.

Normalmente queremos tener varios procesos en la memoria al mismo tiempo. Por tanto, necesitamos considerar de qué formas debemos asignar la memoria disponible a los procesos para que puedan ser cargados en ella. En este apartado estudiaremos la técnica más simple, denominada **asignación contigua de memoria**. Mientras que en capítulos posteriores vemos técnicas más

avanzadas de hacer esta asignación.

En la **asignación contigua de memoria** a cada proceso se le asigna una única sección de memoria contigua. Esto se puede hacer mediante **particionado fijo** o **particionado dinámico**

15.5.1. Particionado fijo

El **particionado fijo** la memoria se divide en varias secciones de tamaño fijo, cada una de las cuales contiene un proceso. Cuando un proceso termina, se carga uno nuevo de la cola de entrada en la partición libre.

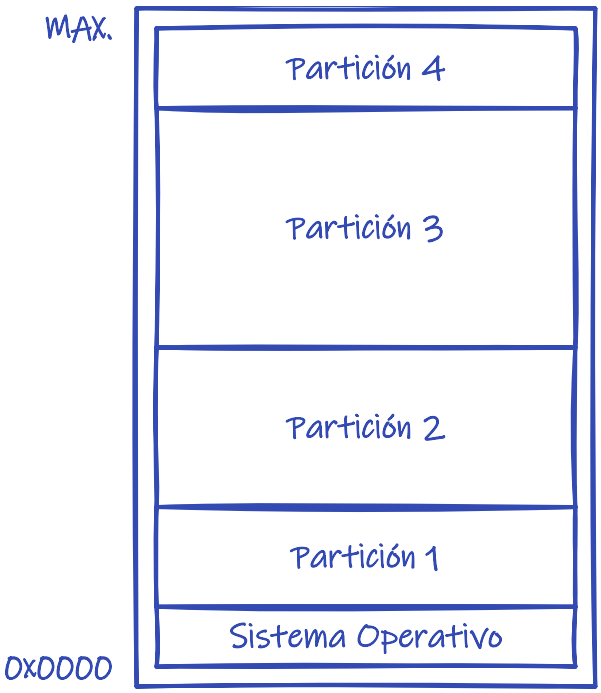


Figura 50. Particionado fijo de la memoria en el IBM OS/360.

Este método fue utilizado originalmente por el IBM OS/360, pero ya no se utiliza

15.5.2. Particionado dinámico

El **particionado dinámico** es una generalización del anterior:

1. El sistema operativo mantiene una tabla indicando qué partes de la memoria están libres y cuáles ocupadas. Inicialmente toda la memoria está libre por lo que es considerada como un gran hueco de memoria disponible.
2. Cuando un proceso llega y necesita memoria, se le busca un hueco lo suficientemente grande para alojarlo. Si se encuentra, solo se le asigna el espacio necesario, que es marcado como ocupado. El resto sigue siendo un hueco libre, aunque de menor tamaño.
3. Si un proceso termina y se crean dos huecos adyacentes, se funden en uno solo.

El **particionado dinámico** se utilizaba fundamentalmente en **sistemas de procesamiento por lotes** y **multiprogramados**. En este último caso, el sistema operativo tenía una **cola de entrada** ordenada por el **planificador de largo plazo** y la recorría asignando memoria a los procesos, hasta que no quedara ningún hueco libre con tamaño suficiente para alojar al siguiente en la cola. Entonces el sistema operativo podía esperar hasta que algunos procesos terminarán y hubiera un

hueco lo suficientemente grande en la memoria, para el siguiente proceso, o podía seguir buscando en la cola de entrada procesos de menores requerimientos, aunque para ello tuviera que saltarse algunos procesos.

En general, en un momento dado el sistema operativo, debe satisfacer una petición de tamaño N con una lista de huecos libres de tamaño variable. Esto no es más que un caso particular del problema clásico de la asignación dinámica de almacenamiento, para el cual hay diversas soluciones:

- En el **primer ajuste** se escoge el primer hueco lo suficientemente grande como para satisfacer la petición. La búsqueda puede ser desde el principio de la lista o desde donde ha terminado la búsqueda anterior.
- En el **mejor ajuste** se escoge el hueco más pequeño que sea lo suficientemente grande para satisfacer la petición. Indudablemente esto obliga a recorrer la lista de huecos completa o a tenerla ordenada por tamaño.
- En el **peor ajuste** se escoge el hueco más grande. Igualmente obliga a buscar en toda la lista de huecos o a tenerla ordenada por tamaño.

Para evaluar qué estrategia es la mejor, se han realizado algunas simulaciones con los siguientes resultados:

- El **primer y el mejor ajuste** son mejores que el peor ajuste en términos de menor tiempo y mayor aprovechamiento del espacio de almacenamiento.
- Si comparamos el **primer y el mejor ajuste** ninguno de ellos destaca sobre el otro en lo que a mejor aprovechamiento del espacio se refiere.
- El **primer ajuste** es normalmente más rápido que el **mejor ajuste**.

15.6. Fragmentación

Las estrategias de asignación de espacio de almacenamiento generalmente sufren de problemas de **fragmentación**. Vamos a comentar brevemente cómo afecta la **fragmentación** a la **asignación contigua de memoria**.

15.6.1. Fragmentación externa

La **fragmentación externa** ocurre cuando hay suficiente espacio libre para satisfacer una petición, pero el espacio no es contiguo. Es decir, el espacio de almacenamiento está fraccionado en un gran número de huecos de pequeño tamaño:

- Afecta tanto a la estrategia del **primer** como del **mejor ajuste**. Siendo el primero mejor en algunos sistemas y el segundo mejor en otros.
- Algunos análisis estadísticos realizados con el **primer ajuste** revelan que incluso con algunas optimizaciones, con N bloques asignados se pierden $0.5N$ por **fragmentación externa** —es decir, un tercio de toda la memoria no es utilizable—. A esto se lo conoce como la regla del 50%.

Existen diversas soluciones a este problema:

- Utilizar técnicas de **compactación**, lo que consiste en mover los procesos para que toda la memoria libre quede en un único hueco de gran tamaño. Sin embargo, esto puede ser muy caro en términos de tiempo y solo puede ser realizado cuando la **asignación de direcciones absolutas se realiza en tiempo de ejecución**.
- La otra solución es permitir que el **espacio de direcciones físico** de un proceso no sea contiguo. Es decir, que la memoria puede ser asignada a un proceso independientemente de donde esté disponible. Existen dos técnicas complementarias que utilizan esta solución: la paginación (véase el Capítulo 16) y la segmentación.

15.6.2. Fragmentación interna

La **fragmentación interna** se origina por la diferencia entre el espacio solicitado y el espacio finalmente asignado.

Supongamos un hueco de espacio libre de 12987 bytes que se va a usar para satisfacer una petición de 12985 bytes. Esto genera un hueco de 2 bytes, pero la cantidad de información que debemos guardar en la lista de huecos para saber que dicho hueco está ahí, es mucho mayor que el tamaño del hueco en sí mismo. Por lo tanto, no nos interesa tener huecos de tamaño arbitrario.

La solución más común es dividir la memoria física en unidades de tamaño fijo y asignarla en múltiplos del tamaño de dichos bloques. Esto hace que, en general, se asigne más memoria de la que realmente se ha solicitado y, por tanto, de la que realmente los procesos van a utilizar. A esto se lo denomina **fragmentación interna**.

15.7. Intercambio

Un proceso debe estar en la memoria para ser ejecutado, pero en algunos sistemas operativos un proceso puede ser sacado de la memoria y copiado a un almacenamiento de respaldo de forma temporal —generalmente un dispositivo de almacenamiento secundario, como un disco— y en algún momento volver a ser traído a la memoria para continuar su ejecución. Al procedimiento descrito se lo denomina **intercambio** o ***swapping***.

15.7.1. Implementación

El **intercambio** se puede implementar de la siguiente manera:

1. La **cola de preparados** contiene todos los procesos que esperan para ser ejecutados en la CPU.
2. Cuando el **planificador de la CPU** decide ejecutar un proceso, llama al **asignador**.
3. El **asignador** comprueba si el siguiente proceso que debe ser ejecutado está en la memoria. Si no lo está y no hay memoria libre, el **asignador** hace que el **gestor de la memoria** intercambie el proceso con alguno de los que sí lo está.
4. Finalmente, el **asignador** ejecuta el resto del cambio de contexto (véase el Apartado 9.6) para entregar la CPU al proceso seleccionado.

Por ejemplo, si a un sistema con **planificación de CPU** basado en prioridad llega a la **cola de preparados** un proceso de alta prioridad, el **gestor de memoria** intercambia algunos procesos de baja prioridad con el de alta prioridad y ejecuta este último. Cuando el proceso de alta prioridad

termina, los de baja prioridad pueden ser intercambiados para continuar su ejecución.

15.7.2. Limitaciones

Sin embargo el **intercambio** presenta algunas limitaciones importantes:

- Si un sistema **reubica las direcciones en tiempo de compilación o carga**, el proceso solo puede ser intercambiado en la misma región de la memoria. Sin embargo, si se utiliza **reubicación en tiempo de ejecución**, entonces el proceso puede ser intercambiado en cualquier región de la memoria, puesto que las **direcciones físicas** son calculadas durante la ejecución.
- El **tiempo de cambio de contexto** en un sistema con **intercambio** puede ser mucho mayor, puesto que incluye el tiempo que se tarda en hacer el intercambio. La mayor parte del tiempo de intercambio es el tiempo de transferencia con el disco, que puede ser de varios cientos de milisegundos, incluso utilizando los discos más rápidos. Esto afecta al **tiempo de cuanto** que siempre debe ser mucho mayor que el tiempo de **cambio de contexto**.
- Un proceso podría disponer de un espacio en memoria de 120 MiB pero estar utilizando solo 2 MiB. Por tanto, es interesante que el sistema operativo conozca con exactitud la memoria utilizada por el proceso —y no la que podría estar utilizando como máximo— para reducir el tiempo de transferencia de los datos al disco durante el intercambio.

  Para eso, el sistema operativo proporciona llamadas al sistema con las que un proceso con requerimientos dinámicos de memoria puede informar del cambio en su necesidad de memoria. Por ejemplo, los sistemas operativos modernos proporcionan llamadas al sistema para reservar y liberar memoria —como malloc() y free() en los sistemas POSIX— gracias a las que el sistema conoce las necesidades reales de los procesos.

- El **intercambio** presenta dificultades cuando el proceso que va a ser sacado de la memoria está esperando por una operación de E/S que accede a la memoria del proceso para leer o escribir datos en ella. Las soluciones podrían ser:
  - No intercambiar procesos con operaciones de E/S síncronas o asíncronas pendientes.
  - Utilizar búferes del sistema operativo en las operaciones de E/S. Por ejemplo, en una operación **write** a un archivo, el sistema operativo copiaría primero los datos a un búfer interno y luego ordenaría la escritura de esos datos. Así el proceso podría ser intercambiado sin problemas. Las transferencias entre los búferes del sistema y la memoria de los procesos serían realizadas, por el sistema operativo, solo cuando los procesos residen en la memoria.

Debido fundamentalmente a que el tiempo de intercambio es muy alto, no se utiliza el intercambio estándar en los sistemas operativos actuales. Lo que sí podemos encontrar en muchos sistemas son versiones modificadas de este mecanismo.

Por ejemplo, en muchas versiones antiguas de UNIX y en los sistemas modernos, el intercambio permanece desactivado y solo se activa cuando la cantidad de memoria usada supera cierto límite. Además, en los sistemas actuales no se intercambian procesos completos si no las porciones menos usadas de cada proceso, como veremos en el Capítulo 17.

Chapter 16. Paginación

Tiempo de lectura: 22 minutos

La traducción entre direcciones virtuales y físicas puede realizarse de diversas maneras. La forma más extendida es la **paginación**, que no es sino un esquema de gestión de la memoria que permite que el espacio de direcciones físico de un proceso no sea continuo, evitando el problema de la **fragmentación externa**.

16.1. Método básico

En la paginación la memoria física se divide en bloques de tamaño fijo denominados **marcos**, mientras que el espacio de direcciones virtual se divide en bloques del mismo tamaño que los marcos, denominados **páginas**. Cuando un proceso va a ser ejecutado, sus páginas son cargadas desde el almacenamiento secundario en marcos libres de la memoria física.

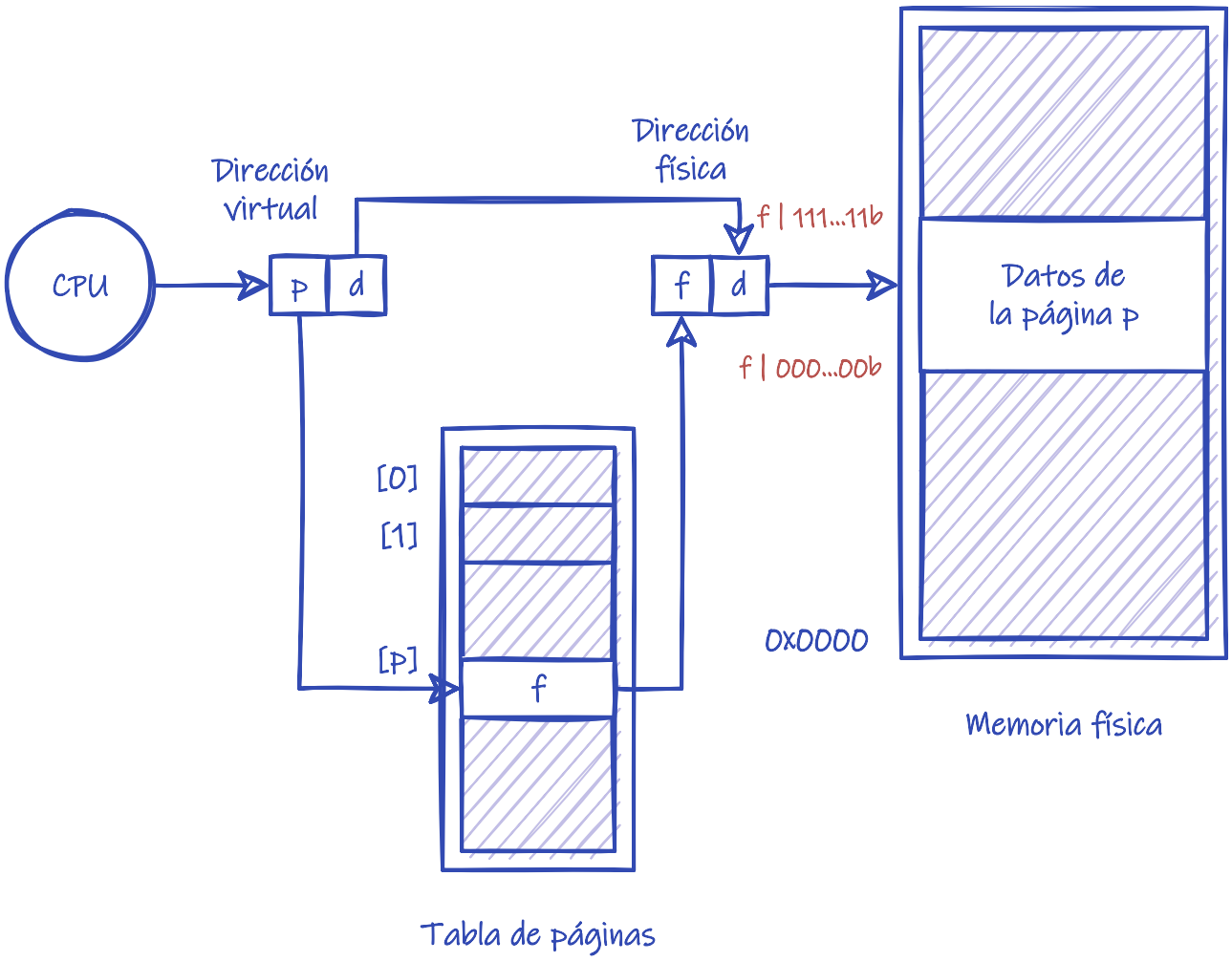


Figura 51. Soporte del hardware para la paginación.

La paginación es una forma de **reubicación de las direcciones en tiempo de ejecución** donde la transformación de las direcciones virtuales en direcciones físicas se realiza de la siguiente manera (véase la Figura 51):

1. Cada dirección virtual generada por la CPU es divida en dos partes: un **número de página** *p* y un **desplazamiento** *d*.
2. El **número de página** es utilizado por la MMU para indexar la **tabla de páginas**, que contiene el **número de marco** *f* de cada **página** en la memoria física.
3. El **número de marco** *f* es combinado con el **desplazamiento** *d* para generar la dirección física que va a ser enviada por el bus de direcciones hacia la memoria.

El tamaño de las **páginas** —y el de los **marcos**— viene definido por el hardware y normalmente es un número entero potencia de 2 que puede variar entre 512 bytes y 16 MiB, dependiendo de la arquitectura. Es decir, si el espacio de direcciones es de 2^m y el tamaño de página es de 2^n, los m - n bits de mayor orden de las direcciones virtuales indican el **número de página**, mientras que los n bits de menor orden indican el **desplazamiento** (véase la Figura 52).

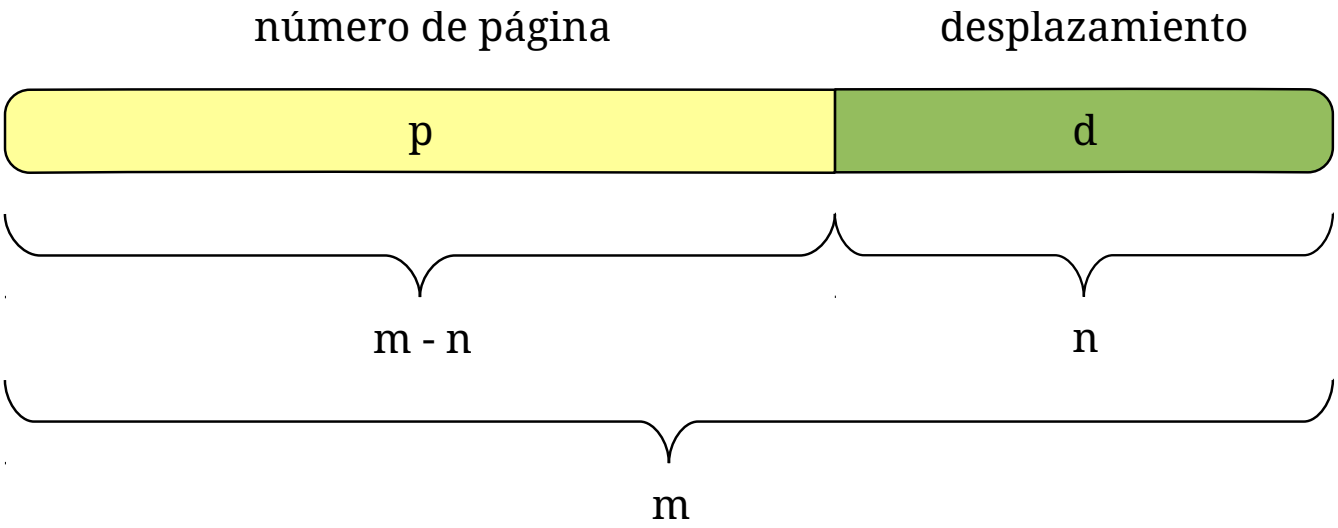


Figura 52. Descomposición de las direcciones virtuales en paginación.

Por ejemplo, en muchos sistemas operativos el tamaño de página es de 4 KiB, por lo que el desplazamiento *n* necesita:

$$n = \log_2 4096 = 12 \text{ bits}$$

Si las direcciones virtuales son de 32 bits, eso deja para el número de página *p*:

$$p = 32 - 12 = 20 \text{ bits}$$

por lo que el espacio de direcciones virtual tiene 2^{20} páginas —es decir, 1 048 576 páginas—.

16.1.1. Desde el punto de vista de los procesos

Cada **página** de un proceso requiere un **marco**. Por tanto, cuando un proceso llega al sistema:

1. Si el proceso requiere *n* **páginas**, el sistema operativo debe escoger *n* **marcos**. Estos **marcos** son tomados de la **lista de marcos libres** que debe mantener el sistema. Puesto que son escogidos de allí donde los haya libres, el **espacio de direcciones físico** puede no ser contiguo, aunque los procesos vean un **espacio de direcciones virtual** contiguo.
2. Los **marcos** seleccionados son asignados al proceso y cada **página** del proceso es cargada en uno de dichos **marcos**.

3. La **tabla de páginas** es actualizada de manera que en la entrada de cada **página** del proceso se pone el número de **marco** correspondiente.

Un aspecto importante de la paginación es la diferencia entre cómo ven los procesos la memoria y como es realmente la memoria física. Cada proceso ve la memoria como un espacio único que lo contiene solo a él. Sin embargo, la realidad es que el programa está disperso por la memoria física, que además puede almacenar a otros programas. Esto es posible porque en cada momento la **tabla de páginas** solo contiene las **páginas** del proceso en ejecución en la CPU.

16.1.2. Desde el punto de vista del sistema operativo

Puesto que el sistema operativo es quién gestiona la memoria física, este debe saber:

- Que **marcos** están asignados y a que **página** de qué proceso o procesos.
- Que **marcos** están disponibles.

Toda esta información generalmente se guarda en una estructura denominada la **tabla de marcos**, que tiene una entrada por cada **marco** de la memoria física.

Además, el sistema operativo debe mantener una copia de la **tabla de páginas** para cada proceso en el **PCB**, igual que mantiene una copia del contador de programa y del contenido de los registros de la CPU. Esta copia es utilizada:

- Por el **asignador** para sustituir la **tabla de páginas** usada por la CPU cuando realiza un **cambio de contexto**. Por lo tanto, el uso de la paginación incrementa el tiempo del cambio de contexto.
- Para la traducción manual de direcciones virtuales en físicas. Por ejemplo, cuando un proceso realiza una llamada al sistema para realizar una operación de E/S y proporciona una dirección como parámetro, dicha dirección debe ser traducida manualmente para producir la dirección física correspondiente, que será comunicada al hardware para realizar la operación.

16.1.3. Tamaño de las páginas

Una decisión de diseño importante es escoger el tamaño de las **páginas** adecuado:

- Con **páginas** más pequeñas esperamos tener menos **fragmentación interna**.
- Con páginas más grandes se pierde menos espacio en la **tabla de páginas**. No olvidemos que cuanto más pequeñas son las **páginas**, más **páginas** son necesarias y, por tanto, más entradas en la **tabla de páginas** se necesitan. Además, la E/S es más eficiente cuanto más datos son transferidos de cada vez.

Los tamaños de **páginas** típicos son 4 y 8 KiB. En un sistema de 32 bits con páginas de 4 KiB —como del que hablamos antes— el espacio de direcciones virtual tiene 1 048 576 páginas. Si se utilizan 4 bytes para cada entrada de la **tabla de páginas** —aunque esto también puede variar— eso significa que cada **tabla de páginas** ocupa 4 MiB de espacio. Mientras que con **páginas** de 8 KiB, la **tabla de páginas** ocuparía 2 MiB de espacios.

También significa que si los 4 bytes de la **tabla de páginas** se utilizan para guardar únicamente el **número de marco**, cada entrada puede direccionar a uno de 2^{32} —o 4 GiB— **marcos** de la memoria física. Si el tamaño de cada **marco** es de 4 KiB —dado que debe coincidir con el tamaño de las

páginas— podemos determinar que el sistema es capaz de direccionar 2^{44} bytes —o 16 TiB— de memoria física, aunque el espacio de direcciones virtual de cada proceso solo le da acceso a un máximo de 4 GiB.

16.2. Soporte hardware de la tabla de páginas

La implementación en hardware de la **tabla de páginas** puede realizarse de diversas maneras.

16.2.1. Almacenada en registros de la CPU

La **tabla de páginas** del proceso actual en la CPU puede alojarse dentro de la propia CPU, en unos registros destinados a tal fin.

Debido a la velocidad de los registros de la CPU, la implementación en registros es la más eficiente. Sin embargo, solo puede ser utilizado para **tablas de páginas** razonablemente pequeñas, ya que alojar tablas de más de 256 entradas en registros es muy costoso.

Por ejemplo, el DEC PDP-11 —para el que se diseñó el primer UNIX— es un ejemplo de sistema con esta implementación. Utilizaba un espacio de direcciones de 16 bits y un tamaño de **páginas** de 8 KiB, por lo que solo necesitaba 8 registros dedicados para alojar toda la **tabla de páginas**.

16.2.2. Almacenada en memoria

La otra opción es alojar la **tabla de páginas** del proceso actual en la memoria, normalmente en un formato definido por la CPU.

En los sistemas modernos se utilizan **tablas de páginas** de un millón de entradas o más, que difícilmente pueden alojarse en registros dentro de la CPU. Por eso, los sistemas actuales almacenan la **tabla de páginas** del proceso actualmente en ejecución, en la memoria. Eso permite disponer de **tablas de páginas** de gran tamaño, aunque a costa de necesitar dos accesos a la memoria física por cada acceso a una dirección virtual.

Para que la MMU pueda conocer la ubicación de la **tabla de páginas** durante la traducción de las direcciones, la CPU debe disponer de un registro —el **PTBR** (*Page-Table Base Register*)— donde se guarda la dirección de la **tabla de páginas** actual.

Además, esto tiene la ventaja de que el **cambio de contexto** es más rápido —respecto al uso de registros para almacenar la tabla de páginas— puesto que solo es necesario cargar un único registro más —el **PTBR**— durante el mismo.

16.2.3. TLB

La solución al retraso originado por el acceso a la tabla de páginas, cuando esta está en la memoria, pasa porque el sistema disponga de una pequeña caché de traducciones en hardware llamada **TLB** (*Translation Look-aside Buffer*).

La **TLB** es una memoria asociativa de alta velocidad. Cada entrada de la **TLB** tiene dos partes: la **clave** —o etiqueta— y el valor. Cuando a la **TLB** se le entrega un elemento, este es comparado simultáneamente con todas las claves. Si se produce alguna coincidencia, la memoria devuelve el

valor de la entrada correspondiente.

Debido a la forma que tienen de operar, son rápidas pero muy caras de fabricar. Por ello, el número de entradas es bajo, normalmente entre 64 y 1024.

Uso básico de la TLB

La **TLB** es utiliza con la **tabla de páginas** de la siguiente manera:

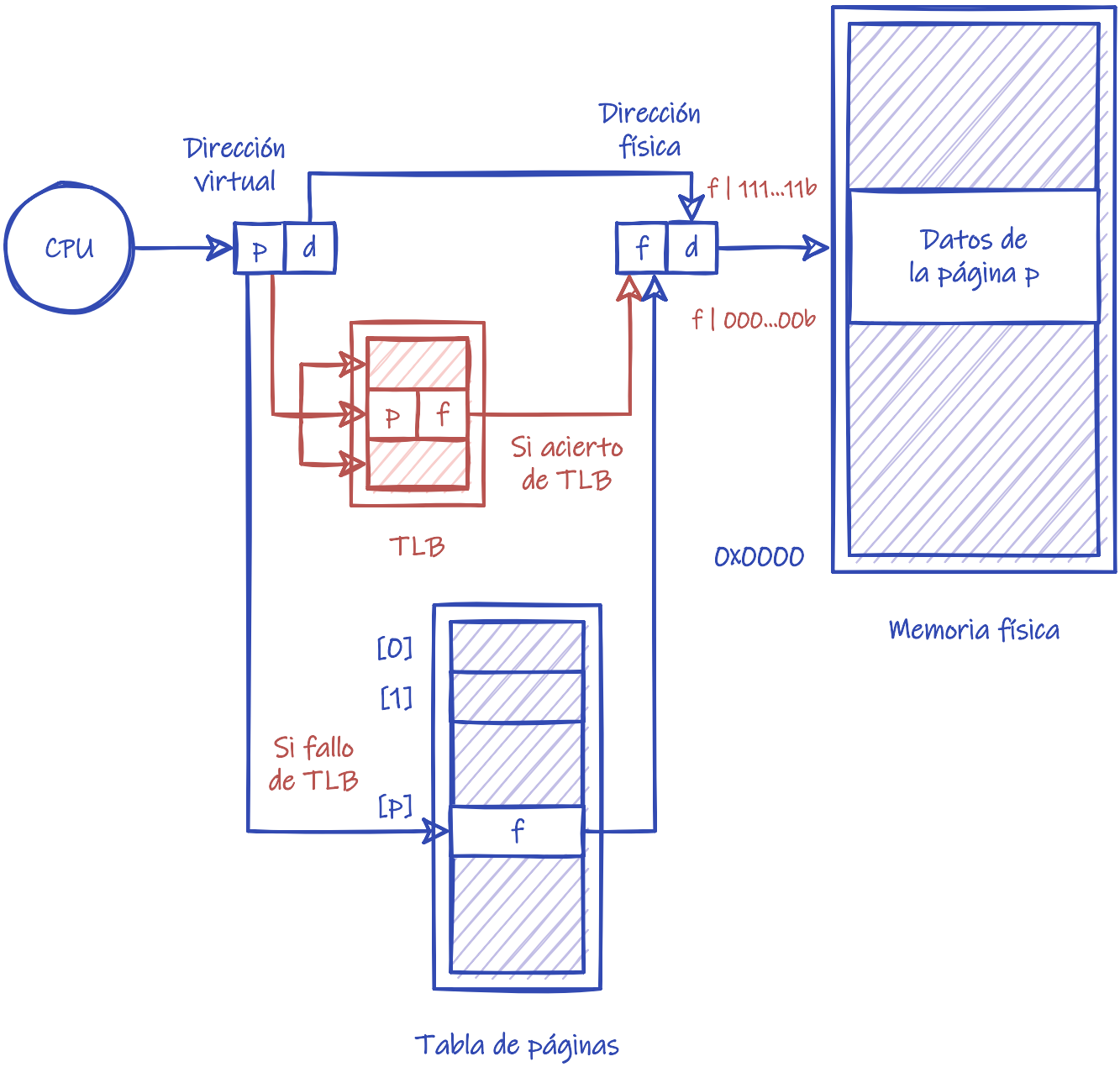


Figura 53. Soporte del hardware para la paginación con TLB.

1. La **TLB** contiene unas pocas entradas de la **tabla de páginas**.
2. Cuando la CPU genera una dirección virtual, el **número de página** es entregado a la **TLB**. La **TLB** utiliza los números de páginas como **clave**, por lo que si hay alguna coincidencia, devolverá la entrada correspondiente de la **tabla de páginas**.
3. Si hay coincidencia —o **acierto de TLB**— el **número de marco** es extraído de la entrada

devuelta por la **TLB** y es utilizado para generar la dirección física. Todo este proceso puede requerir un 10% más de tiempo que si no se hiciera la traducción de las direcciones.

4. Si no hay coincidencia —o **fallo de TLB**— es necesario acceder a la **tabla de páginas** para obtener la entrada correspondiente directamente de ella. Indudablemente, este acceso puede beneficiarse de la existencia de diferentes niveles de caché en el acceso a la memoria principal.
5. En este último caso, la entrada recuperada debe ser añadida a la **TLB**, por lo que sí está llena, se debe seleccionar una para ser sustituida. Los algoritmos de reemplazo utilizados van, desde elegir una aleatoriamente, hasta el **LRU** (*Least Recently Used*).

Borrado de la TLB en el cambio de contexto

Una cuestión importante es qué ocurre con las **TLB** cuando el sistema operativo realiza un **cambio de contexto**.

En general, es necesario que el asignador realice un borrado de la **TLB**. De lo contrario, el nuevo proceso podría utilizar las entradas de la **tabla de páginas** del viejo proceso, que estuvieran almacenadas en la **TLB**. Sin embargo, un proceso no tiene por qué utilizar todas las entradas de la **TLB**, por lo que sería más interesante, no tener que borrar las entradas de procesos anteriores, mientras no sean necesarias, por si estos vuelven a ser ejecutados en la CPU.

El borrado se puede evitar si cada entrada de la **TLB** tiene un **ASID** (*Address-Space Identification*), que no es más que un identificador único para cada proceso. En este tipo de **TLB**, en la **clave** se buscan pares **(número de página, ASID)**, donde el primero proviene de la dirección virtual y el segundo es el **ASID** del proceso actual. De esta forma, si el **número de página** coincide, pero no el **ASID**, se produce un fallo de la **TLB**. Esto obliga a acceder a la **tabla de páginas** en memoria para recuperar la entrada, evitando que se lea por error la entrada de un proceso anterior.

Esta característica está presente en los procesadores DEC Alpha, MIPS y Sun UltraSPARC. Entre 2005 y 2006 también comenzó a ser incluida en algunos procesadores de la familia x86, a través de las extensiones de virtualización Intel VT y AMD Pacifica.

Tiempos de acceso a la memoria

El rendimiento de un sistema con paginación, está relacionado con el concepto de **tiempo de acceso efectivo** a la memoria T_{em}, que intenta estimar el tiempo que realmente se tarda en acceder a la memoria, teniendo en cuenta mecanismos del sistema operativo como el método de paginación o la existencia de **TLB**.

En muchos sistemas informáticos, el **tiempo de acceso** a la memoria física T_m es de unos pocos nanosegundos. Por lo tanto, en el método de básico de paginación el **tiempo de acceso efectivo** es el doble del **tiempo de acceso** a la memoria:

$$T_{em} = 2T_m$$

Obviamente, en métodos de paginación donde hagan falta más accesos para obtener finalmente el **número de marco**, el **tiempo de acceso efectivo** será mayor.

Supongamos que tenemos un sistema con **TLB** y que conocemos la probabilidad p de que la entrada que consultamos esté en la **TLB**. Entonces, el **tiempo de acceso efectivo** se podría calcular como la

probabilidad (1 - p) de que la entrada no esté en la **TLB**, por el **tiempo de acceso** necesario, en ese caso $2T_m$, más la probabilidad de que la entrada sí esté en la **TLB** p, por el **tiempo de acceso** T_m, porque solo hace falta acceder una vez a la memoria:

$$T_{em} = (1-p)2T_m + pT_m = (2-p)T_m$$

Como se puede observar, cuanto más se aproxima a 1 la probabilidad p de que la entrada esté en la **TLB**, más cerca está T_{em} de T_m.

Para mejorar esta probabilidad:

- Las **TLB** permiten marcar algunas entradas como insustituibles. Esto normalmente se hace con las entradas de las **páginas** del código y los datos del núcleo, ya que son páginas que se utilizan con muchísima frecuencia.
- Si la MMU soporta páginas de mayor tamaño que el estándar, se utilizan para alojar el código y los datos del núcleo. De esta forma se minimiza el número de entradas de la **TLB** que utilizan, con el fin de disponer de más entradas libres para los procesos en ejecución.

En la familia x86 el tamaño de página estándar es de 4 KiB, pero también se puede disponer de páginas de 4 MiB. En x86-64 las páginas de gran tamaño son de 2 MiB, aunque algunos modelos también soportan páginas de 1 GiB.

16.3. Protección de la memoria

La protección de las páginas se consigue mediante unos bits que indican las operaciones que se pueden realizar sobre ellas. Normalmente, estos bits son almacenados en cada una de las entradas de la **tabla de páginas**.

16.3.1. Bits de protección

Los **bits de protección** pueden ser:

- **Solo lectura**.
- **Lectura — Escritura**. En algunos sistemas hay un bit específico para este permiso, mientras que en otros se utilizan bits separados, como: **lectura**, **escritura** y **ejecución**; que se pueden combinar libremente.
- **Solo ejecución**. Que no existen en todas las plataformas. Por ejemplo, la familia x86 careció de esta característica hasta que AMD la incluyó en su arquitectura x86-64, lo que obligó a Intel a incluirla en las versiones más modernas de Pentium IV. El bit —que para ser exacto indica **no ejecución**— fue introducido para evitar cierto tipo de ataques de seguridad.

Durante la traducción de las direcciones, la MMU comprueba que el tipo de acceso sea válido. Si no lo es, se genera una excepción de violación de protección de memoria, dado que el acceso en un modo no autorizado se considera una instrucción privilegiada. Normalmente, el sistema operativo responde a dicha excepción terminando el proceso que la generó.

16.3.2. Bit de válido

Además de los **bits de protección** comentados, se suele añadir a cada entrada un **bit de válido**:

- Cuando una **página es válida**, la página existe en el espacio de direcciones virtual del proceso. Es decir, que la página se puede utilizar. Otro término comúnmente utilizado, es que la página es **legal**.
- Cuando la **página es inválida**, la página no existe en el espacio de direcciones virtual del proceso. Es decir, que la página no se puede utilizar. El término alternativo utilizado es que la página es **ilegal**.

Al igual que con los **bits de protección**, los intentos de acceso a una página ilegal generan una excepción.

El sistema operativo puede utilizar este bit para permitir o denegar cualquier tipo de acceso a una **página**. Generalmente, porque no se le ha asignado un **marco** de memoria física, ya que esa página no está siendo utilizada por el proceso.

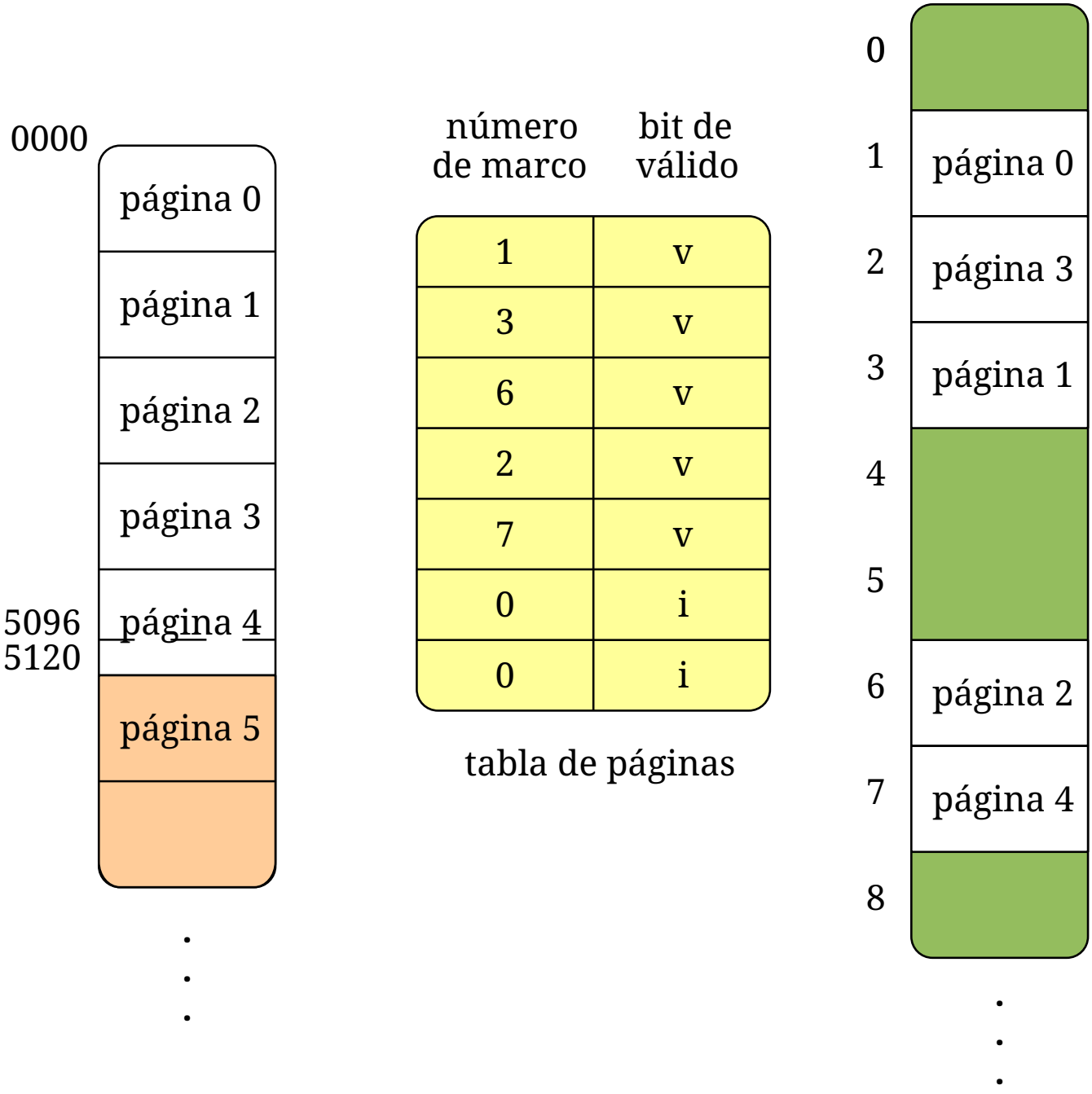


Figura 54. Bit de válido en la tabla de páginas.

Por ejemplo, en la Figura 54, vemos el espacio de direcciones virtual y la **tabla de páginas** de un proceso de 5096 bytes en un sistema con **páginas** de 1 KiB. Puesto que el proceso no ocupa todo el espacio de direcciones, solo las direcciones de la 0 a la 5119 son válidas. En dicho ejemplo, podemos apreciar varios fenómenos:

- Debido a la **fragmentación interna**, las direcciones de la 5097 a la 5119 son válidas, aunque el proceso solo ocupe hasta la 5096. Es decir, se está asignando al proceso una porción de memoria que no necesita.
- Solo las **páginas** con datos y código del proceso son válidas. Mientras que todas las **páginas** con direcciones por encima de la 5119 están marcadas como ilegales.

En general, los procesos solo necesitan una porción muy pequeña de su espacio de direcciones virtual. Por ejemplo, en un sistema de 32 bits, muy pocos procesos necesitan los 3 GiB disponibles como máximo para cada proceso —el 1 GiB restante suele estar ocupado por el núcleo del

sistema—. Utilizando el **bit de válido**, el sistema operativo no tiene que asignar **marcos** a **páginas** no utilizadas por el proceso, ahorrando mucha memoria.

En el Apartado 16.1.3, vimos que el tamaño de la **tabla de páginas** se puede calcular como el número máximo de páginas del espacio de direcciones virtual multiplicado por el tamaño de cada entrada de la tabla. Así, en un sistema de 32 bits con páginas de 4 KiB y 4 bytes por entrada, se necesitan 4 MiB de memoria para almacenar la **tabla de páginas**. Como un proceso suele ocupar muy poco de su espacio de direcciones virtual, suele ser un desperdicio de memoria crear y almacenar una **tabla de páginas** completa, con una entrada para cada **página** del espacio de direcciones.

Para evitarlo, en algunas CPU existe el registro **PTLR** (*Page-Table Length Register*) que se utiliza para indicar el tamaño actual de la **tabla de páginas**. Este valor es comparado por la MMU, durante la traducción de las direcciones virtuales, con el **número de página** de cada dirección virtual, de manera que las **páginas** con entradas más allá de la última almacenada en la tabla son consideradas ilegales.

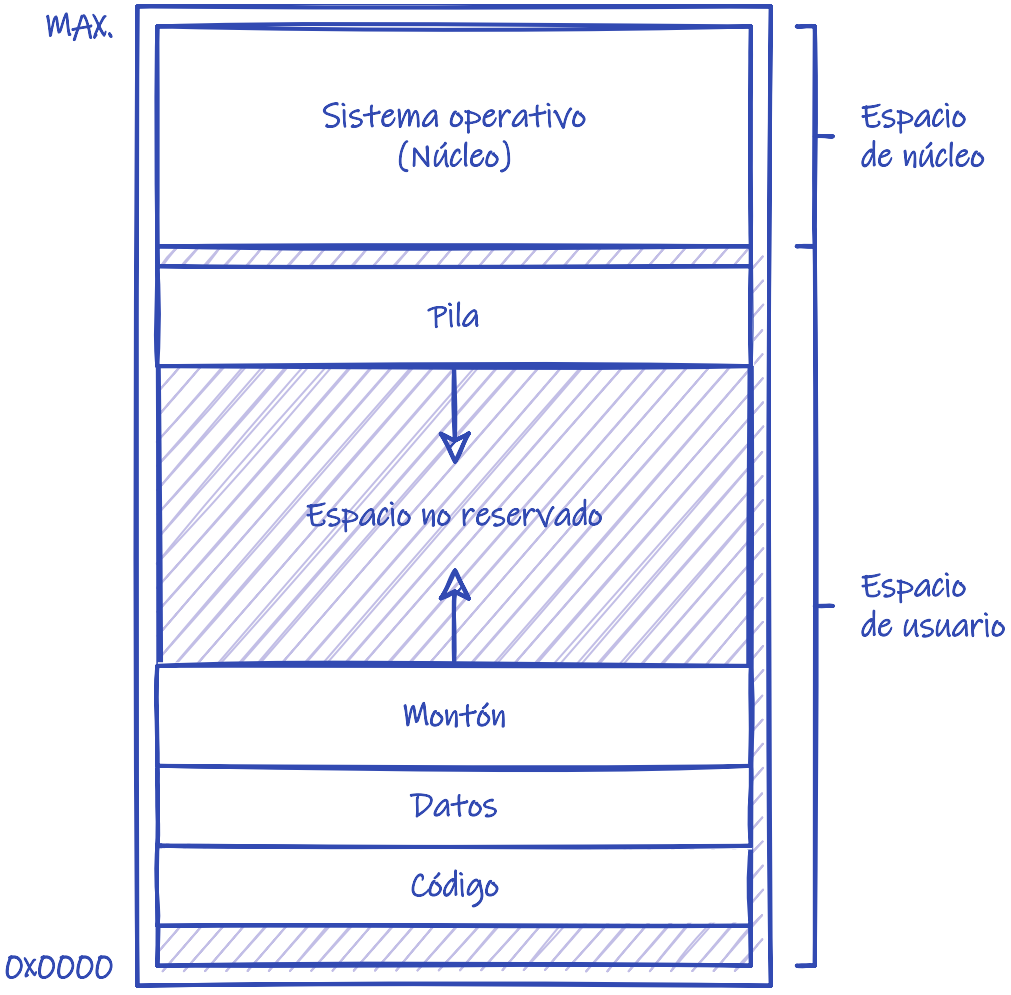


Figura 55. Anatomía de un proceso en memoria.

En realidad, el registro **PTLR** no es de mucha utilidad en los sistemas operativos modernos porque, tal y como vimos en el Apartado 9.1, lo más común es que los procesos tengan un espacio de direcciones virtual disperso como el de la Figura 55. En ella, podemos observar, como el sistema operativo ubica los diferentes componentes del proceso de una forma particular dentro del espacio de direcciones virtual. Este esquema permite que tanto el **montón** —a través del mecanismo de asignación dinámica de memoria— como la **pila** puedan extenderse —según las necesidades de

memoria que tenga el proceso— sobre la región de memoria no ocupada. Esa región también puede ser parcialmente ocupada por **librerías de enlace dinámico** o regiones de **memoria compartida**, si son necesarias durante la ejecución del proceso.

En cualquier caso, las **páginas** de la región no ocupada, forman parte del espacio de direcciones virtual, pero no necesitan tener asignado ningún **marco** de memoria física, en tanto en cuanto el proceso no las vaya a utilizar. La falta de **marco** es indicada por el sistema operativo utilizando el **bit de válido** para denegar el acceso.

16.4. Páginas compartidas

Una de las ventajas importantes de la paginación, es la posibilidad de compartir **páginas** entre procesos. Para conseguir esto, basta con que las **páginas compartidas** de los distintos procesos tengan asignadas un mismo **marco**. Esto permite, por ejemplo, que los procesos de un mismo programa puedan compartir las **páginas** de código o los datos de solo lectura con el fin de ahorrar memoria. También permite compartir las **páginas** de código de una librería compartida enlazada en diferentes procesos.

Compartir **páginas** no solo permite ahorrar memoria, pues en los sistemas operativos modernos, la comunicación entre procesos mediante memoria compartida (véase el Capítulo 11), se implementa mediante **páginas compartidas**.

16.5. Paginación jerárquica

Al método básico de paginación, se lo conoce como **tabla de páginas lineal**. Sin embargo, las CPU comúnmente, utilizan otras técnicas a la hora de estructurar la **tabla de páginas**. Una de las más comunes es la **paginación jerárquica**, utilizada en los procesadores de la familia x86 y en ARM, entre otros.

La mayor parte de los sistemas modernos soportan el uso de espacios de direcciones de gran tamaño. Por ejemplo, supongamos un sistema con un espacio de direcciones virtual de 32 bits:

tamaño del espacio de direcciones	$= 2^{32} = 4$ GiB
tamaño de página	$= 2^{12} = 4$ KiB
número de páginas	$= 2^{32} / 2^{12} = 2^{32-12} = 2^{20} = 1\,048\,576$ entradas

Es decir, que si el tamaño de cada entrada fuera de 4 bytes, la **tabla de páginas** de un proceso podría ocupar hasta 4 MiB; que debe ser alojada en una región continúa del espacio de direcciones físico, por lo que podría darse el caso de que en algún momento no hubiera un hueco contiguo lo suficientemente grande. Una forma de resolver este problema es partir la **tabla de páginas**, de manera que no sea necesario asignarle memoria de forma contigua.

16.5.1. Paginación jerárquica de dos niveles

La **paginación jerárquica** se basa en la idea de que un vector de gran tamaño puede ser mapeado en uno más pequeño, que a su vez, puede ser mapeado en un vector de menor tamaño.

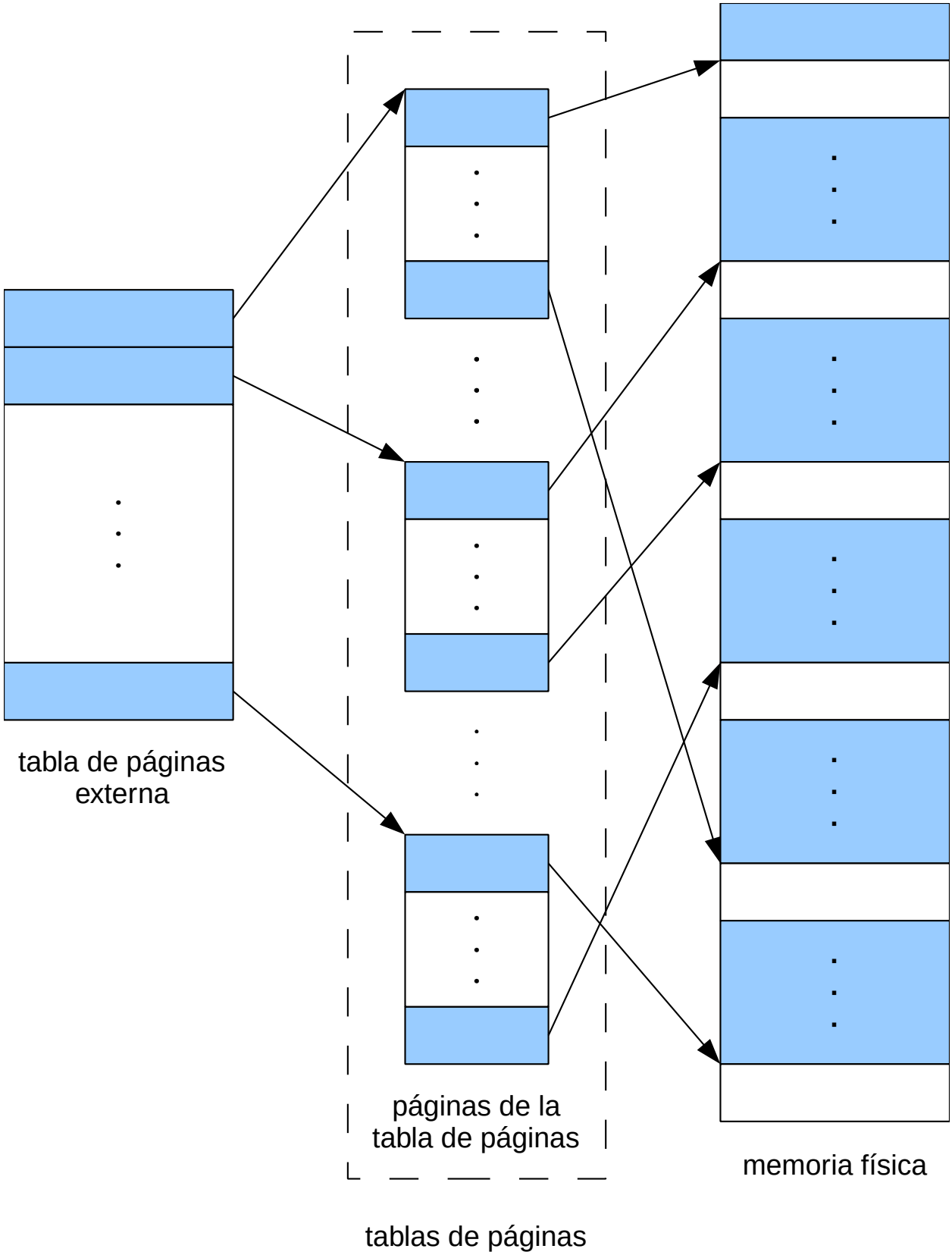


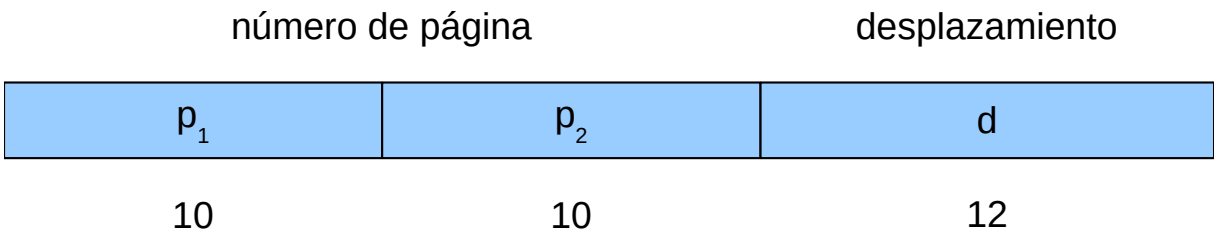
Figura 56. Esquema de paginación jerárquica de dos niveles.

Por ejemplo, si asumimos el caso anterior de un sistema con un espacio de direcciones de 32 bits y un tamaño de página de 4 KiB, entonces podemos dividir la tabla de páginas de 1 048 576 entradas —4 MiB si cada entrada necesita 4 bytes— en 1024 porciones, cada una de las cuales cabría en un **marco** de 4 KiB.

Estos **marcos**, a su vez, pueden ser mapeados por 1024 entradas con las direcciones físicas de cada **marco**. Si organizamos estas 1024 entradas en un vector lineal, obtendremos una **tabla de páginas externa** de 4 KiB (véase la Figura 56).

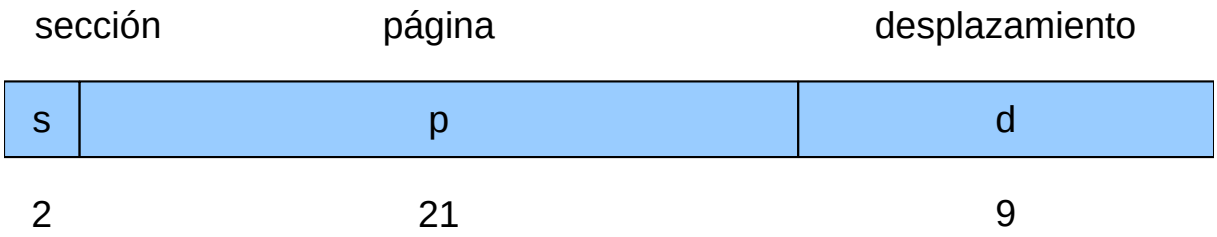
Dado que 4 KiB es una cantidad de memoria muy pequeña, muchos sistemas operativos mantienen la **tabla de páginas externa** en la memoria mientras el proceso se está ejecutando. Sin embargo, ahora la **tabla de páginas** está dividida en **marcos**, que no tienen por qué ser asignados de forma contigua en la memoria. Incluso podrían ser intercambiados al disco, en caso de necesitar memoria libre.

Para tener dos niveles de 1024 entradas, solo es necesario dividir el **número de página** *p* de la dirección virtual —que tenía 2^{20} bits— en dos **números de página** de 2^{10} bits cada uno:



Este es el método utilizado por la familia de procesadores x86.

Otra variación de la **paginación jerárquica de dos niveles** es la utilizada por VAX. Estos sistemas utilizaban una arquitectura de 32 bits con un tamaño de página de 512 bytes. Las direcciones virtuales eran divididas de la siguiente manera:



El espacio de direcciones de un proceso estaba dividido en 3 secciones. Los 2 bits de orden más alto *s* de las direcciones virtuales se utilizaban para indicar la **sección**. Cada **sección** estaba dividida en **páginas** de 512 bytes, por lo que los siguientes 21 bits de las direcciones virtuales *p* eran utilizadas para seleccionar la **página** concreta.

Dividiendo el espacio de direcciones de esta manera, el sistema operativo podía mantener secciones sin utilizar mientras no fueran necesarias. Esto era importante, puesto que la **tabla de páginas** de una **sección** tenía un tamaño de 8 MiB.

16.5.2. Paginación jerárquica de N niveles

En general, en la **paginación jerárquica** de *n* niveles el **número de página** *p* de cada dirección virtual es dividido en *n* números: { p_1, p_2, p_3, ..., p_N}, donde:

1. p_1 se utiliza para indexar la **tabla de páginas externa** —también llamada **directorio de páginas** o **tabla de páginas de nivel 0**— cuya dirección conoce la CPU mediante el **PTBR**. La entrada obtenida de esta manera, contiene la dirección en la memoria física de una porción de la **tabla de páginas** en el siguiente nivel —el nivel 1—.
2. p_2 se utiliza para indexar la **tabla de páginas de nivel 1**. E, igualmente, la entrada así obtenida contiene la dirección en la memoria física de una porción de la **tabla de páginas** en el siguiente nivel —el nivel 2—.
3. El proceso continúa hasta que p_N, se utiliza para indexar la **tabla de páginas de nivel N-1**, con la que se obtiene el **número de marco** que es utilizado, finalmente, para generar la dirección física al combinarlo con el desplazamiento d de la dirección virtual.

Como se puede ver, resolver una dirección virtual necesita tantos accesos a la memoria como niveles hay en la jerarquía.

Debido a que la traducción funciona desde las **tablas de páginas** de nivel superior —nivel 0— hacia las de nivel inferior —nivel N - 1— a esta estructura también se la conoce como **tabla de páginas directa**.

Los procesadores DEC Alpha y MIPS utilizan una variante denominada **tabla de páginas virtualizada**. En este esquema, el último nivel de la **paginación jerárquica**, aunque esté en marcos separados en el espacio de direcciones físico, se mapea de manera continua en el espacio de direcciones virtual del proceso.

Así, si la consulta a la **TLB** falla, se indexa la **tabla de páginas** directamente con direcciones virtuales usando el **número de página** completo. Esta consulta conlleva la traducción de la dirección virtual de la entrada indexada, que puede estar en la **TLB**. Si no es así, se pasa a recorrer la **tabla de páginas** desde el nivel 0 y usando direcciones físicas.

Existen algunos procesadores que utilizan más de dos niveles. Por ejemplo, los procesadores x86-64 utilizan un esquema de 4 niveles de paginación. Cada **página** es de 4 KiB —como en el resto de la familia x86— pero como cada entrada en la **tabla de páginas** es de 8 bytes —con el fin de poder almacenar direcciones de 64 bits— en cada una caben 512 entradas, por lo que los **números de página** de cada nivel necesitan 9 bits. Eso significa que de las direcciones virtuales se utilizan actualmente 48 bits —resultado de multiplicar 4 niveles por 9 bits cada uno más 12 bits de desplazamiento— aunque el límite de la arquitectura para las direcciones virtuales sea de 64 bits.

Chapter 17. Memoria virtual

Tiempo de lectura: 1 hora y 5 minutos

La **memoria virtual** es una técnica que permite la ejecución de procesos sin que estos tengan que ser cargados completamente en la memoria.

Los programas suelen tener partes de código que rara vez son ejecutadas. Por ejemplo, las funciones para manejar condiciones de error que, aunque útiles, generalmente nunca son invocadas. También es frecuente que se reserve más memoria para datos de lo que realmente es necesario. Por ejemplo, muchos programadores tienen la costumbre de hacer cosas tales como declarar un *array* de 65536 elementos, cuando realmente solo necesitan 255. Teniendo todo esto en cuenta, y con el fin de mejorar el aprovechamiento de la memoria, parece que sería interesante no tener que cargar todas las porciones de los procesos y que, aún así, pudieran ejecutarse. Eso es exactamente lo que proporciona la **memoria virtual**.

La habilidad de ejecutar un proceso cargado parcialmente en memoria proporciona algunos beneficios importantes:

- Un programa nunca más estaría limitado por la cantidad de memoria disponible.

  Es decir, los desarrolladores pueden escribir programas considerando que disponen de un espacio de direcciones virtual extremadamente grande, sin considerar la cantidad de memoria realmente disponible. No debemos olvidar que sin memoria virtual, para que un proceso pueda ser ejecutado, debe estar completamente cargado en la memoria.

- Puesto que cada programa ocupa menos memoria, más programas se pueden ejecutar al mismo tiempo; con el correspondiente incremento en el uso de la CPU y en el rendimiento del sistema, pero sin efectos negativos en el tiempo de respuesta y en el de ejecución.

El concepto de **memoria virtual** no debe confundirse con el de **espacio de direcciones virtual**. Sin embargo están relacionados, puesto que el que exista separación entre la memoria física y la manera en la que los procesos perciben la memoria es un requisito para poder implementar la memoria virtual.

17.1. Paginación bajo demanda

La **paginación bajo demanda** es la técnica con la que frecuentemente se implementa la **memoria virtual** en los sistemas con paginación.

En la **paginación bajo demanda** las páginas individuales, en las que se dividen los espacios de direcciones virtuales de los diferentes procesos, pueden ser sacadas de la memoria de manera temporal y copiadas a un almacenamiento de respaldo, para posteriormente volver a ser traídas a la memoria cuando son necesitadas por su proceso. A este proceso de guardado y recuperación de las páginas sobre el almacenamiento de respaldo se lo denomina **intercambio** o ***swapping*** y es llevado a cabo por un componente del sistema operativo denominado el **paginador**.

Para que se puedan cargar las páginas cuando son necesitadas por su proceso, hace falta que el

paginador sepa cuándo lo son. Eso requiere que el hardware proporcione algún tipo de soporte, por ejemplo, incorporando un **bit de válido** a la entrada de cada página en la tabla de páginas, que se utiliza de la siguiente manera:

- Cuando el **bit de válido** está a 1 la página es legal y está en la memoria. Es decir, la página existe en el espacio de direcciones virtual del proceso y tiene asignado un marco de memoria física.
- Cuando el **bit de válido** está a 0, pueden ocurrir varias cosas:
  - La página es legal, pero está almacenada en disco y no en la memoria.
  - La página no es legal. Es decir, no existe en el espacio de direcciones virtual del proceso.

    Esto puede ser debido a que la página esté en un hueco del espacio de direcciones —en una región que no está siendo utilizada— por lo que el sistema operativo no le ha asignado espacio de almacenamiento ni en disco ni en la memoria.

Si un proceso accede a una página legal, no ocurre nada y la instrucción se ejecuta con normalidad. Pero si accede a una página marcada como inválida:

1. Al intentar acceder a la página, la MMU comprueba el bit de válido y genera una excepción de **fallo de página** al estar marcada como inválida. Dicha excepción es capturada por el sistema operativo.
2. El sistema operativo comprueba en una tabla interna si la página es legal o no. Es decir, si la página realmente no pertenece al espacio de direcciones virtual del proceso o si pertenece, pero está almacenada en el disco. Esta tabla interna suele almacenarse en el PCB del proceso como parte de la información de gestión de la memoria.
3. Si la página es ilegal, el proceso ha cometido un error y debe ser terminado. En sistemas POSIX, por ejemplo, el sistema envía al proceso una señal de **violación de segmento** que lo obliga a terminar.
4. Si la página es legal, se carga desde el disco:
   a. El núcleo busca un marco de memoria libre que, por ejemplo, se puede escoger de la lista de marcos libres del sistema.
   b. Se solicita una operación de disco para leer la página deseada en el marco asignado.

      Puesto que no resulta eficiente mantener la CPU ocupada mientras la página es recuperada desde el disco, el sistema debe solicitar la lectura de la página y poner al proceso en estado **esperando**.

   c. Cuando la lectura del disco haya terminado, se modifica la tabla interna, antes mencionada, y la tabla de páginas para indicar que la página está en la memoria.
   d. Reiniciar la instrucción que fue interrumpida por la excepción. Generalmente esto se hace colocando el proceso nuevamente en la **cola de preparados** y dejando que el **asignador** lo reinicie cuando sea escogido por el **planificador** de la CPU.

Un caso extremo de la paginación bajo demanda es la **paginación bajo demanda pura**. En ella la ejecución de un proceso se inicia sin cargar ninguna página en la memoria. Cuando el sistema operativo sitúa el contador de programa en la primera instrucción del proceso —que es una página

no residente en memoria— se genera inmediatamente un **fallo de página**. La página es cargada en la memoria —tal y como hemos descrito anteriormente— y el proceso continúa ejecutándose, fallando cuando sea necesario con cada página que necesite y no esté cargada. Las principales ventajas de la **paginación bajo demanda pura** son:

- Nunca se trae desde el disco una página que no sea necesaria.
- El inicio de la ejecución de un proceso es mucho más rápido que si se cargara todo el proceso desde el principio.

17.1.1. Requerimientos de la paginación bajo demanda

Los requerimientos hardware para que un sistema operativo pueda soportar la **paginación bajo demanda** son:

- Tabla de páginas con habilidad para marcar entradas inválidas, ya sea utilizando un bit específico o con valores especiales en los bits de protección.
- Disponibilidad de un dispositivo de almacenamiento secundario.

  En este dispositivo se guardan las páginas que no están presentes en la memoria principal. Normalmente se trata de un disco conocido como **dispositivo de intercambio**, mientras que la sección de disco utilizada concretamente para dicho propósito se conoce como **espacio de intercambio** o ***swap***.

- Posibilidad de reiniciar cualquier instrucción después de un fallo de página.

  En la mayor parte de los casos esta funcionalidad es sencilla de conseguir. Sin embargo, la mayor dificultad proviene de las instrucciones que pueden modificar diferentes posiciones de la memoria, como aquellas pensadas para mover bloques de bytes o palabras. En el caso de que el bloque de origen o de destino atraviese un borde de página, la instrucción sería interrumpida cuando la operación solo haya sido realizada parcialmente. Si además ambos bloques se superpusieran, no se podría reiniciar la instrucción completa. Las posibles soluciones a este problema deben ser implementadas en la CPU.

17.1.2. Rendimiento de la paginación bajo demanda

Indudablemente, el rendimiento de un sistema con **paginación bajo demanda** se ve afectado por el **número de fallos de páginas**. En el peor de los casos, en cada instrucción un proceso puede intentar acceder a una página distinta, empeorando notablemente el rendimiento. Sin embargo, esto no ocurre, puesto que los programas tienden a tener localidad de referencia (véase el Apartado 17.6).

Tiempo de acceso efectivo

Como con la **paginación**, el rendimiento de un sistema con **paginación bajo demanda** también está relacionado con el concepto de **tiempo de acceso efectivo** a la memoria, que intenta estimar el tiempo que realmente se tarda en acceder a la memoria teniendo en cuenta mecanismos del sistema operativo.

Supongamos que conocemos la probabilidad p_{fp} de que ocurra un fallo de página. El **tiempo de**

acceso efectivo se podría calcular como la probabilidad de que no ocurra un fallo de página (1 - p_{fp}) por el **tiempo de acceso** a la memoria T_m, más la probabilidad de que ocurra un fallo de página p por el tiempo necesario para gestionar cada fallo de página —o **tiempo de fallo de página**— T_{fp}:

$$T_{em} = (1 - p_{fp})T_m + p_{fp}T_{fp}$$

Por tanto, para calcular el **tiempo de acceso efectivo** T_{em} necesitamos estimar el **tiempo de fallo de página** T_{fp}, que se consume fundamentalmente en:

- Servir la excepción de fallo de página.

  Esto incluye capturar la interrupción, salvar los registros y el estado del proceso, determinar que la interrupción es debida a una excepción de **fallo de página**, comprobar si la página es legal y determinar la localización de la misma en el disco. Aproximadamente, en realizar esta tarea el sistema puede tardar de 1 a 100 µs.

- Leer la página en un marco libre. En esta tarea se puede tardar alrededor de 8 ms, pero este tiempo puede ser mucho mayor si el dispositivo está ocupado y se debe esperar a que se realicen otras operaciones.
- Reiniciar el proceso. Si incluimos el tiempo de espera en la cola de preparados, se puede tardar entre 1 y 100 µs.

Como se puede apreciar, la mayor parte del **tiempo de fallo de página** es debido al tiempo requerido para acceder al **dispositivo de intercambio**.

Para ilustrar el cálculo del **tiempo de acceso efectivo** a la memoria, solo vamos a considerar el tiempo requerido para acceder al **dispositivo de intercambio**, ignorando las otras tareas a realizar durante el **fallo de página**, ya que comparativamente consumen mucho menos tiempo. Vamos suponer que el **tiempo de acceso** a la memoria T_m es de 200 ns y que la probabilidad p_{fp} es muy pequeña —es decir, p << 1—:

$$\begin{aligned} T_{em} &= (1 - p_{fp}) \cdot 200\,\text{ns} + p_{fp} \cdot 8\,\text{ms}. \\ &= (1 - p_{fp}) \cdot 200\,\text{ns} + p_{fp} \cdot 8\,000\,000\,\text{ms}. \\ &\approx 200\,\text{ns} + 7\,999\,800\,\text{ms}. \cdot p \end{aligned}$$

Como se puede apreciar el **tiempo de acceso efectivo** es proporcional a la **tasa de fallos de página** τ_{fp}:

$$T_{em} \approx T_m + \tau_{fp}$$

donde:

$$\tau_{fp} = p_{fp}T_{fp}$$

Por ejemplo, si un proceso causa un fallo de página en uno de cada 1000 accesos —es decir, p_{fp} = 0.001— el **tiempo de acceso efectivo** es de 8.2 ms, por lo que el rendimiento del sistema es 40 veces inferior debido a la **paginación bajo demanda**. Por tanto, es necesario mantener la **tasa de fallos de página** τ_{fp} lo más baja posible para mantener un rendimiento adecuado.

Por simplicidad, por el momento hemos considerado T_m como el tiempo de acceso a la memoria

física. Sin embargo, realmente estamos en un sistema que utiliza paginación —incluso con varios niveles— y que puede tener una TLB para mejorar su rendimiento. Entonces, si queremos ser más precisos, podemos considerar T_m como el **tiempo de acceso efectivo** que vimos en el Apartado 16.2.3.3 para el método básico de paginación. Por lo que sustituyéndolo en la expresión anterior:

$$T_{em} \approx (2 - p_{tlb})T_m + tau_{fp}$$

donde p_{tlb} es la probabilidad de que una entrada esté en la TLB y T_m ahora sí es el **tiempo de acceso** a la memoria física.

Manejo y uso del espacio de intercambio

Otro aspecto fundamental que afecta al rendimiento de la **paginación bajo demanda** es el uso del espacio de intercambio.

Cuando un proceso genera un **fallo de página**, el sistema operativo debe recuperar la página de allí donde esté almacenada. Si esto ocurre al principio de la ejecución, ese lugar seguramente será el archivo que contiene la imagen binaria del programa, pues es donde se encuentran las páginas en su estado inicial. Sin embargo, el acceso al espacio de intercambio es mucho más eficiente que el acceso a un sistema de archivos, incluso aunque el primero esté almacenado dentro de un archivo de gran tamaño. Esto es debido a que los datos se organizan en bloques contiguos de gran tamaño, se evitan las búsquedas de archivos y las indirecciones en la asignación de espacio. Por ello debemos plantearnos qué hacer con las imágenes de los programas que van a ser ejecutados.

- Se puede mejorar el rendimiento copiando en el espacio de intercambio la imagen completa de los programas durante el inicio del proceso, para después realizar la **paginación bajo demanda** sobre dicha copia.
- Otra alternativa es cargar las páginas desde el archivo que contiene la imagen cuando son usadas por primera vez, pero siendo escritas en el espacio de intercambio cuando dichas páginas tienen que ser reemplazadas. Esta aproximación garantiza que solo las páginas necesarias son leídas desde el sistema de archivos, reduciendo el uso de espacio de intercambio, mientras que las siguientes operaciones de intercambio se hacen sobre dicho espacio.
- También se puede suponer que el código de los procesos no puede cambiar. Esto permite utilizar el archivo de la imagen binaria para recargar las páginas de código, lo que también evita escribirlas cuando son sustituidas. Sin embargo, el espacio de intercambio se sigue utilizando para las páginas que no están directamente asociadas a un archivo, como la pila o el montón de los procesos.

  Este último método parece conseguir un buen compromiso entre el tamaño del espacio de intercambio y el rendimiento. Por eso se utiliza en la mayor parte de los sistemas operativos modernos.

17.2. Copy-on-write

El ***copy-on-write*** o **copia durante la escritura** permite la creación rápida de nuevos procesos, minimizando la cantidad de páginas que deben ser asignadas a estos.

Para entenderlo, es importante recordar que la llamada al sistema fork() crea un proceso hijo cuyo

espacio de direcciones es un duplicado del espacio de direcciones del padre. Indudablemente, esto significa que durante la llamada es necesario asignar suficientes marcos de memoria física como para alojar las páginas del nuevo proceso hijo. El ***copy-on-write*** minimiza de la siguiente manera el número de marcos que deben ser asignadas al nuevo proceso:

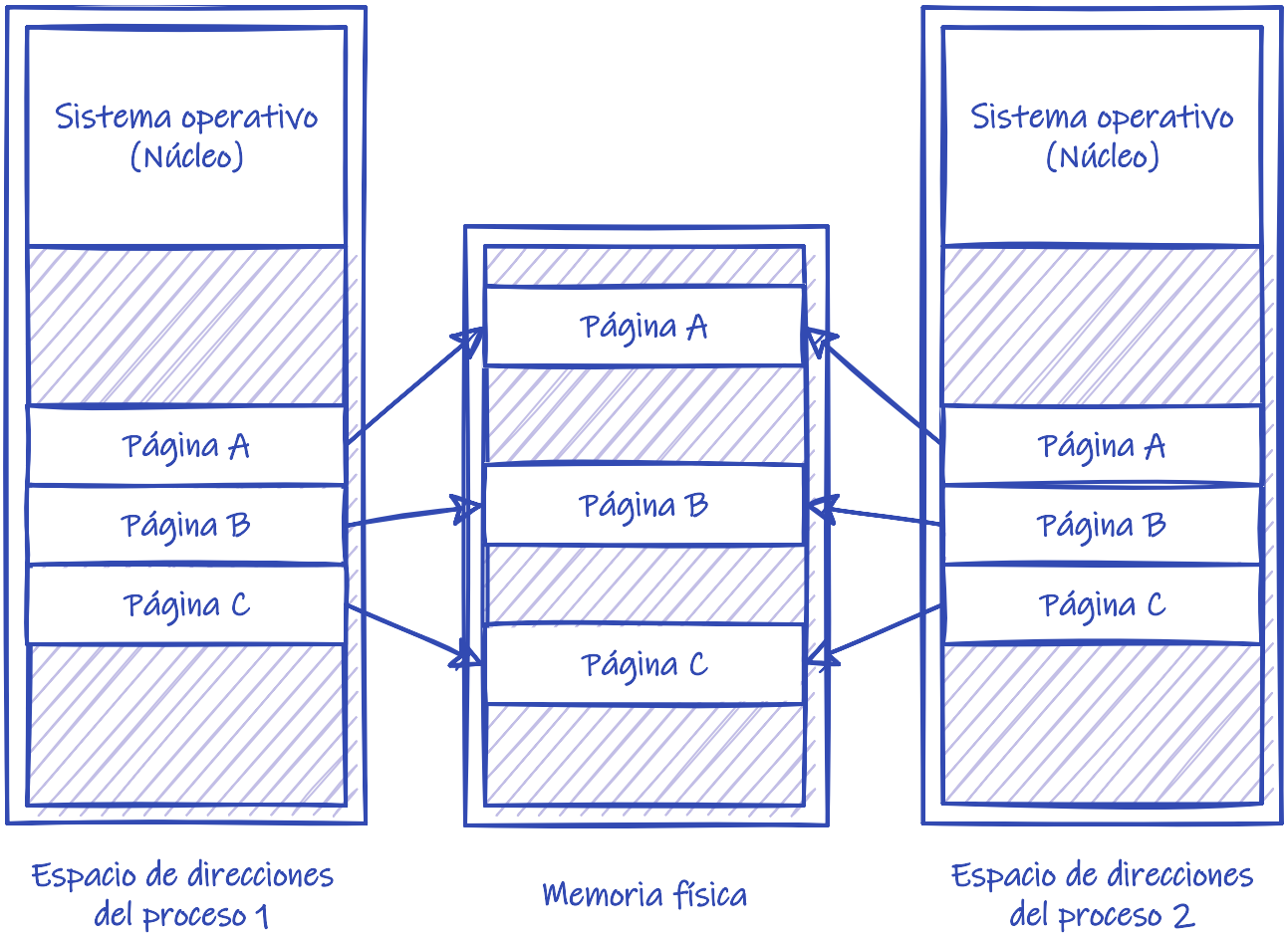


Figura 57. Copy-on-write antes de que el proceso 1 modifique la página A.

1. Cuando la llamada al sistema fork() crea el nuevo proceso lo hace de forma que este comparta todas sus páginas con las del padre (véase la Figura 57).

   Sin el ***copy-on-write*** el fork() tendría que asignar marcos de memoria física al hijo, para a continuación copiar las páginas del padre en ellos. Sin embargo, con el ***copy-on-write*** padre e hijo mapean sus páginas en los mismos marcos, evitando tener que asignar memoria libre.

2. Las páginas compartidas se marcan como ***copy-on-write***.

   Para ello se pueden marcar todas las páginas como de **solo lectura** en la tabla de páginas de ambos procesos y utilizar una tabla interna alojada en el PCB para indicar cuáles son realmente de solo lectura y cuáles están en ***copy-on-write***. Es importante destacar que realmente solo las páginas que pueden ser modificadas se marcan como ***copy-on-write***. Las páginas que no pueden ser modificadas —por ejemplo, las que contienen el código ejecutable del programa— simplemente pueden ser compartidas como de solo lectura por los procesos, como hemos comentado anteriormente.

3. Si algún proceso intenta escribir en una página ***copy-on-write***, la MMU genera una excepción para notificar el suceso al sistema operativo. Siguiendo lo indicado en el punto anterior, la

excepción se originaría porque la página está marcada como de solo lectura, por lo que el sistema operativo comprobará si se trata de un acceso a una página ***copy-on-write*** o a un intento real de escribir en una página de solo lectura. Para ello, el sistema solo tiene que mirar la tabla interna almacenada en el PCB. Si se ha intentado escribir en una página de solo lectura, el proceso ha cometido un error y generalmente será terminado.

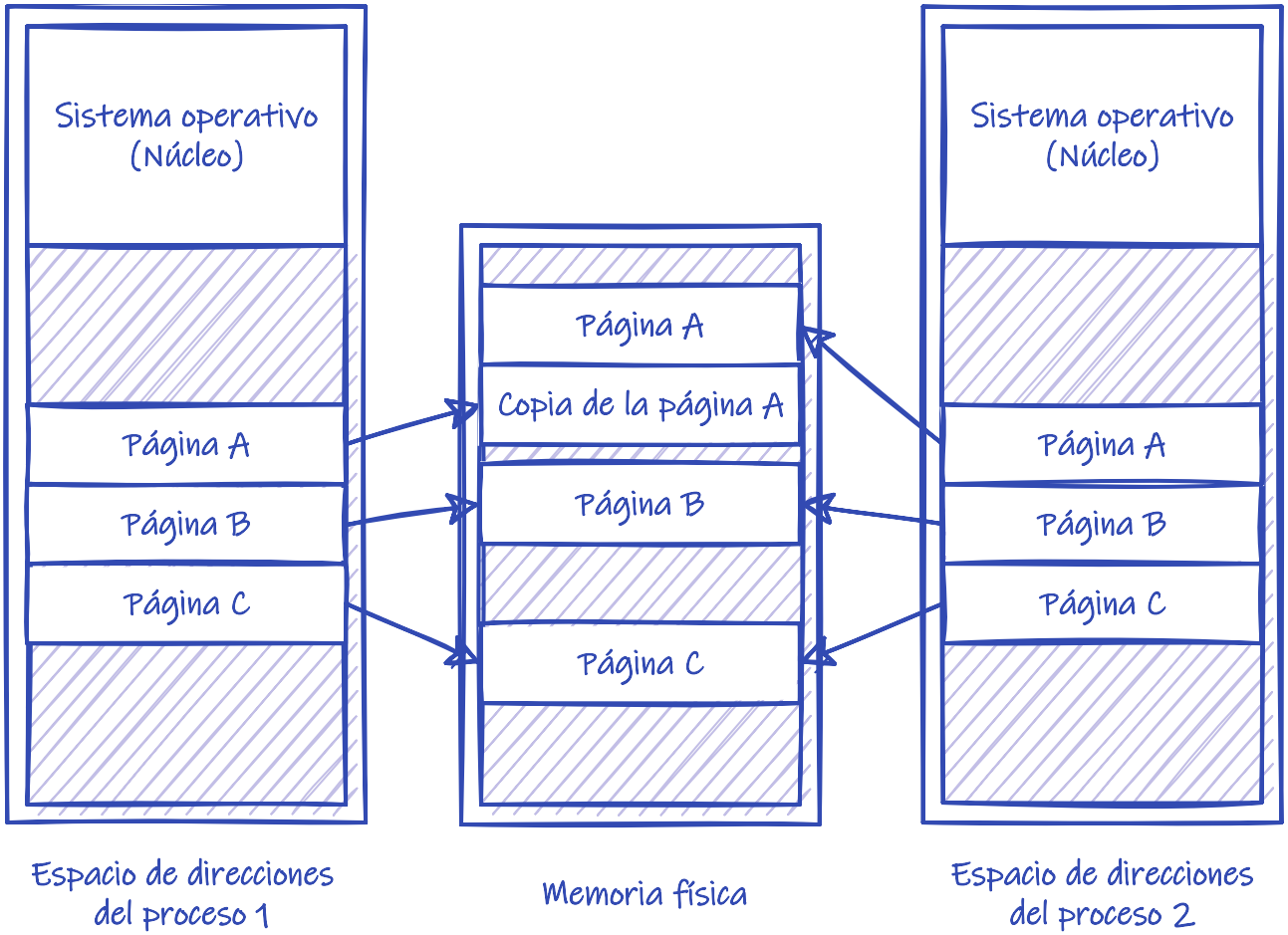


Figura 58. Copy-on-write después de que el proceso 1 modifique la página A.

1. Si el sistema detecta una escritura a una página de ***copy-on-write*** solo tiene que copiarla en un marco libre y mapearlo en el espacio de direcciones del proceso (véase la Figura 58).

   Para esto se sustituye la página compartida por otra que contiene una copia, pero que ya no está compartida. Obviamente, la nueva página debe ser marcada como de escritura para que en el futuro pueda ser modificada por el proceso.

2. La página original marcada como ***copy-on-write*** puede ser marcada como de escritura y no como ***copy-on-write***, pero solo si no va a seguir siendo compartida. Esto es así porque una página marcada como ***copy-on-write*** puede estar siendo compartida con otros procesos.
3. El sistema operativo puede reiniciar el proceso. A partir de ahora, este puede escribir en la página sin afectar al resto de los procesos. Sin embargo, puede seguir compartiendo otras páginas en ***copy-on-write***.

El ***copy-on-write*** permite ahorrar memoria y tiempo en la creación de los procesos, puesto que solo se copian las páginas que son modificadas por estos. Por eso se trata de una técnica común en múltiples sistemas operativos, como por ejemplo los sistemas POSIX modernos y Microsoft Windows.

El ***copy-on-write*** es especialmente interesante si a continuación se va a utilizar la llamada al sistema exec() puesto que si es así, copiar el espacio de direcciones completo es una pérdida de tiempo.

17.3. Archivos mapeados en memoria

Los **archivos mapeados en memoria** permiten acceder a un archivo como parte del espacio de direcciones virtuales de un proceso. Algunas de las características de esta técnica son:

- Cuando una región del espacio de direcciones queda marcada para ser mapeada sobre una región de un archivo, se utiliza una estrategia similar a la comentada para el método básico de la paginación bajo demanda. La diferencia es que las páginas son cargadas desde dicho archivo y no desde el espacio de intercambio. Es decir, en un primer acceso a una página mapeada se produce un **fallo de página** que es resuelto por el sistema operativo leyendo una porción del archivo en el marco asignado a la página.
- Esto significa que la lectura y escritura del archivo se realiza a través de lecturas y escrituras en la memoria, lo que simplifica el acceso y elimina el costo adicional de las llamadas al sistema: read(), write(), etc.
- Las escrituras en disco se suelen realizar de forma asíncrona. Es decir, los datos no se escriben inmediatamente en disco cuando se modifica la página, sino que el sistema operativo comprueba periódicamente las páginas modificadas y las escribe en disco.
- Los marcos utilizados en el mapeo pueden ser compartidos, lo que permite compartir los datos de los archivos. Además, se puede incluir soporte de ***copy-on-write***, lo que permite a los procesos compartir buena parte de un archivo en modo de solo lectura, pero disponiendo de sus propias copias de aquellas páginas que modifiquen.

  Indudablemente, para que los procesos puedan compartir datos es necesario que exista algún tipo de coordinación (véase el Capítulo 13).

17.3.1. Ejemplo de mapeo de archivos

Tanto en los sistemas POSIX como en Windows API el archivo a mapear hay que abrirlo con la llamada al sistema destinada a ello. Por ejemplo, en sistemas POSIX:

```
int fd = open( "foo.txt", O_RDONLY );
```

En la API POSIX con esto es suficiente para usar la llamada al sistema mmap() y mapear el archivo en memoria:

```
int length = lseek( fd, 0, SEEK_END ); ②
void* p = mmap(
    NULL,
    length,                ②
    PROT_READ,             ④
    MAP_SHARED,
    fd,                    ①
```

```
    0                   ③
);
```

① Descriptor del archivo abierto con open().

② La longitud de la región del archivo a mapear. Si queremos mapear todo el archivo, podemos usar lseek() para conocer su longitud.

③ Si no se mapea todo el archivo, se puede indicar la posición del archivo donde comenzar el mapeo. Esta posición debe ser múltiplo del tamaño de página.

④ Permisos de la memoria mapeada. Deben coincidir con el modo con el que se abrió el archivo.

Una vez mapeado el archivo, si no se van a crear más mapeos, se puede cerrar el descriptor de archivo con close(). Y la región de memoria mapeada se puede liberar, al terminar, con munmap().

En mapped-files.c se puede ver un ejemplo de un programa que cuenta el número de líneas, palabra y caracteres de un archivo. Para acceder al archivo, primero lo mapea en memoria, para así poder acceder a su contenido sin tener que leerlo usando read(). Una vez ha terminado, libera la memoria mapeada.

El ejemplo en mapped-files.cpp es identico al de mapped-files.c, pero desarrollado en C++. Usa la clase definida en memory_map.hpp para abstraer la gestión del mapeo del archivo, por lo que en la implementación de sus métodos, obviamente, se utilizan las mismas llamadas al sistema que en mapped-files.c. Este ejemplo es algo más complejo porque también muestra como manejar el puntero a la memoria mapeada de forma que sea liberada automáticamente cuando ya no es necesaria, siguiendo las pautas recomendadas en C++.

Con Windows API el proceso requiere un paso más. Primero hay que usar el manejador del archivo abierto para crear un **objeto de mapeo de archivo** con CreateFileMapping(). Después, el manejador devuelto por CreateFileMapping() es usado con MapViewOfFile() para mapear el archivo en la memoria del proceso.

Para liberar la memoria mapeada, se utiliza UnmapViewOfFile(). Mientras que para cerrar el manejador del **objeto de mapeo de archivo** se usa CloseHandle(), como es habitual.

17.3.2. Mapeo de archivos en el núcleo

Algunos sistemas operativos ofrecen el servicio de mapeo de archivos en la memoria solo a través de una llamada al sistema concreta, permitiendo utilizar las llamadas estándar —read(), write(), etc.— para hacer uso de la E/S tradicional. Sin embargo, muchos sistemas modernos utilizan el mapeo en la memoria independientemente de que se pidan o no.

Por ejemplo, en los sistemas POSIX, si un proceso utiliza llamada al sistema mmap() es porque explícitamente pide que el archivo sea mapeado en memoria. Por tanto, el núcleo mapea el archivo en el espacio de direcciones del proceso.

Pero en Linux, Microsoft Windows y otros sistemas operativos modernos, adicionalmente, cuando un archivo es abierto con la llamada al sistema estándar —como **open**— el archivo es mapeado en el espacio de direcciones del núcleo y las llamadas **read** y **write** son traducidas en accesos a la memoria en dicha región. No importa como sea abierto el archivo. Estos sistemas tratan toda la E/S

a archivos como mapeada en memoria, permitiendo que el acceso a los mismos, tenga lugar a través del eficiente componente de gestión de la memoria.

17.4. Reemplazo de página

Hasta el momento hemos considerado que disponemos de memoria física suficiente para atender cualquier fallo de página, pero ¿qué ocurre cuando no quedan marcos libres?. En ese caso el código que da servicio a la excepción de **fallo de página** debe escoger alguna página, intercambiarla con el disco y utilizar el marco de la misma para cargar la nueva página. Es decir, debemos modificar la función que ejecuta los pasos descritos en el Apartado 17.1 de la siguiente manera:

4. Si la página es legal, debe ser cargada desde el disco.
    a. Buscar la localización de la página en disco.
    b. El núcleo debe buscar un marco de memoria libre que, por ejemplo, se puede escoger de la lista de marcos libres del sistema.
        i. Si hay uno, usarlo.
        ii. Si no hay, usar un **algoritmo de reemplazo de página** para seleccionar una víctima.
        iii. Escribir la víctima en el disco y cambiar las **tablas de páginas y de marcos libres** de acuerdo a la nueva situación. Para evitar mantener la CPU ocupada, el sistema debe solicitar la escritura de la página y poner al proceso en estado de **esperando**.
    c. Se solicita una operación de disco para leer la página deseada en el marco asignado. Para evitar mantener la CPU ocupada, el sistema debe solicitar la escritura de la página y poner al proceso en estado de **esperando**.
    d. Cuando la lectura del disco haya terminado, se debe modificar la tabla interna de páginas válidas, y la tabla de páginas para indicar que la página está en la memoria.
    e. Reiniciar la instrucción que fue interrumpida por la excepción.

Es importante destacar que en caso de **reemplazo** se necesita realizar dos accesos al disco. Esto se puede evitar utilizando un **bit de modificado** asociado a cada entrada en la **tabla de páginas**:

- Este bit es puesto a 1 por el hardware cuando se escribe en la página.
- Se puede evitar escribir en disco aquellas páginas que tienen este bit a 0 cuando son seleccionadas para reemplazo, siempre que el contenido de la página no haya sido sobrescrito por otra en el espacio de intercambio.

En general, para implementar la paginación bajo demanda necesitamos:

- Un **algoritmo de asignación de marcos**, que se encarga de asignar los marcos a los procesos.
- Un **algoritmo de reemplazo de página** para seleccionar qué página reemplazamos cuando no hay marcos suficientes.

Obviamente, estos algoritmos deben ser escogidos de forma que mantengan la **tasa de fallos de página** lo más baja posible para perjudicar en lo mínimo el rendimiento del sistema.

17.4.1. Algoritmos de reemplazo de páginas

Hay muchos **algoritmos de reemplazo de página**. La cuestión es cómo seleccionar uno en particular, sabiendo que debe tener la menor tasa posible de **fallos de página**.

Trazas de referencias

Podemos evaluar el algoritmo utilizando una secuencia de referencias a memoria y calculando el número de **fallos de página**. A dicha secuencia de referencias se la denomina **trazas de referencias**.

Las trazas pueden ser obtenidas aleatoriamente u obteniéndolas a partir de las que hace un proceso en un sistema real. Indudablemente, esta última alternativa puede proporcionar miles de millones de referencias. Para reducir el número de datos podemos hacer dos cosas:

- De cada referencia solo necesitamos considerar el número de página.
- Si tenemos una referencia a la página *p*, cualquier referencia inmediatamente posterior a dicha página *p* nunca provocará un fallo de página, por lo que podemos ignorarlas. Esto no tendría por qué ser cierto y si consideramos que hay varios procesos en el sistema y unos pueden expropiar a los otros. Pero por simplicidad, ese no es nuestro caso.

Por ejemplo, si obtenemos la siguiente traza de un proceso particular:

```
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610,
0102, 0103, 0104, 0101, 0609, 0102, 0105
```

con páginas de 100 bytes, podemos reducirla a:

```
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
```

Indudablemente, para determinar el número de **fallos de página** también debemos conocer el número de marcos disponibles para el proceso.

Reemplazo FIFO

El **algoritmo de reemplazo de páginas FIFO** es el más sencillo. Funciona de la siguiente manera:

- Se asocia a cada página el instante de tiempo en que fue cargada por última vez.
- Cuando hay que hacer reemplazo se selecciona como víctima a la página más antigua.

Realmente no es estrictamente necesario almacenar el instante de tiempo en que una página fue cargada. En su lugar, las páginas en memoria pueden ser almacenadas en una cola FIFO, puesto que así se conserva su orden de llegada en el tiempo, que es lo que nos interesa. Esta misma cola FIFO es utilizada en algunos algoritmos que veremos posteriormente.

Vamos a ilustrar este algoritmo con un ejemplo, donde supondremos que tenemos 3 marcos de memoria física:

7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
7	7	7	2					2	2			1	1			1	7	7	7
	0	0	0					4	4			4	3			3	3	0	0
		1	1					1	0			0	0			2	2	2	1
F	**F**	**F**	**F**					**F**	**F**			**F**	**F**			**F**	**F**	**F**	**F**
																			13 fallos de página

Como se puede observar, las primeras 3 referencias generan **fallos de página** porque se supone que los marcos no están asignados a ninguna página. A partir de ahí se utiliza el **algoritmo de reemplazo FIFO** para seleccionar una página cuyo marco es utilizado para cargar la página requerida por el proceso.

El **algoritmo FIFO** no siempre tiene un buen rendimiento:

- Puesto que utiliza el orden en el tiempo, puede reemplazar tanto páginas que no están siendo utilizadas como páginas usadas frecuentemente. Sin embargo, aunque esto pase, todo seguirá funcionando correctamente, aunque aumentará la tasa de **fallos de página** enlenteciendo el sistema.
- No siempre que aumenta la cantidad de memoria disponible mejora el rendimiento.

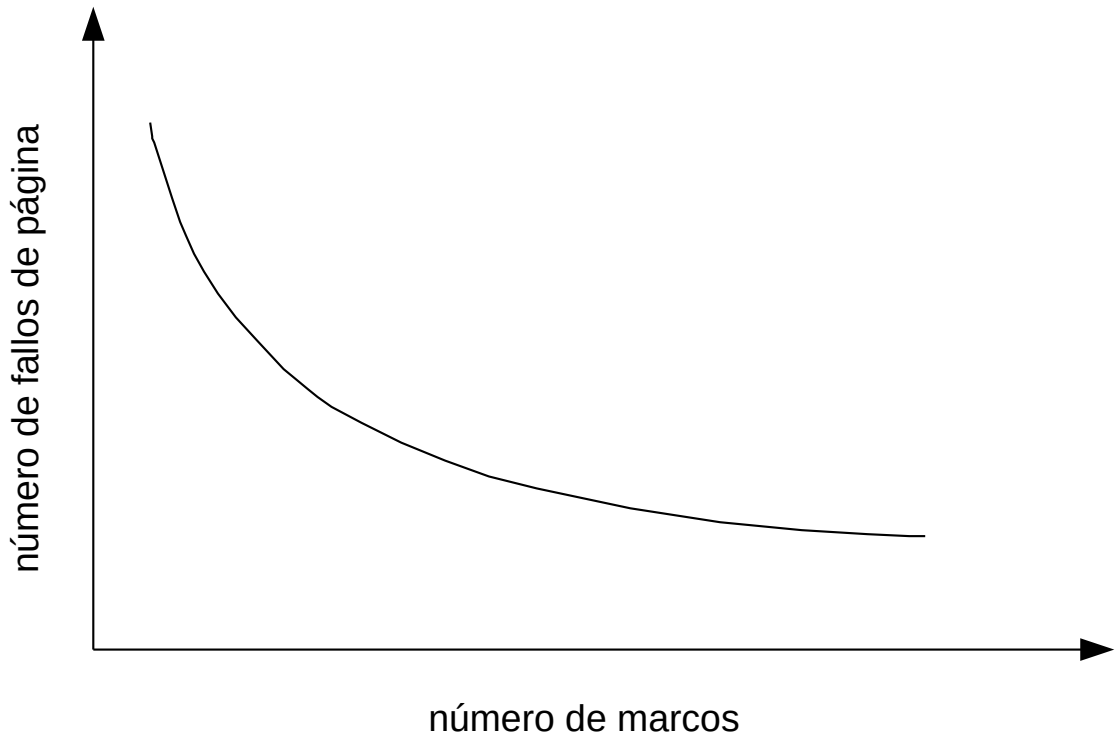


Figura 59. Fallos de página frente a número de marcos.

Es de esperar que si el número de marcos disponibles aumenta, el número de **fallos de página** disminuya (véase la Figura 59). Sin embargo, con el **algoritmo FIFO** el número de **fallos de página** puede aumentar cuando el número de marcos disponibles se incrementa. Es lo que se conoce como la **anormalidad de Belady**.

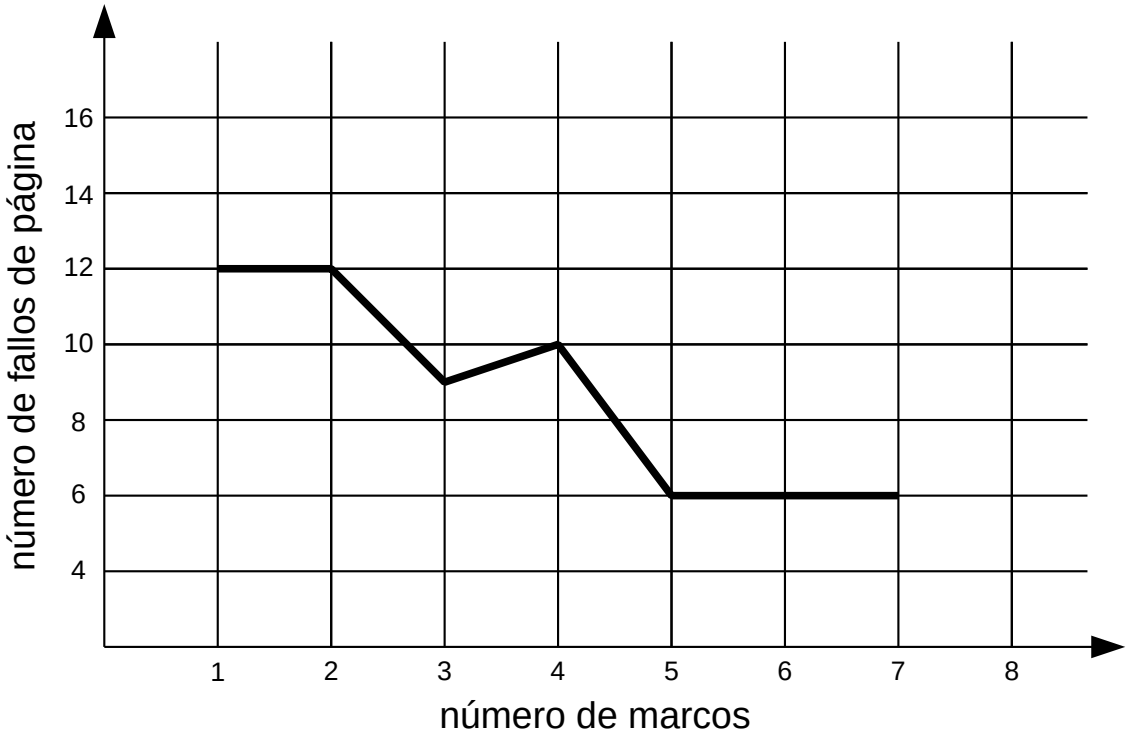


Figura 60. Fallos de página con el algoritmo de reemplazo FIFO.

Para ilustrarlo consideraremos la siguiente traza de referencias:

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

La Figura 60 muestra la curva de **fallos de página** frente el número de marcos. Como se puede apreciar, el número de **fallos de página** con cuatro marcos es superior al número de fallos con tres marcos, aunque era de esperar que el número de fallos de páginas decrementara al aumentar el número de marcos.

Reemplazo óptimo

Como resultado del descubrimiento de la **anormalidad de Belady** se comenzó a buscar un algoritmo de reemplazo de página óptimo. Es decir:

- Que tuviera la **tasa de fallo de página** más baja posible.
- Que nunca sufra de la **anormalidad de Belady**.

Ese algoritmo existe y consiste en reemplazar la página que no va a ser utilizada en el mayor periodo de tiempo.

7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
7	7	7	2					2				2	3			2	2		
	0	0	0					0				0	0			0	0		
		1	1					4				1	1			1	7		
F	**F**	**F**	**F**					**F**				**F**	**F**			**F**	**F**		

7	0	1	2	0	2	0	2	4	0	2	0	1	3	0	1	2	7	0	1
9 fallos de página																			

Desafortunadamente, el **algoritmo de reemplazo óptimo** es difícil de implementar, puesto que necesita saber las páginas que serán referenciadas en el futuro. Por lo que solo se usa en estudios comparativos, con el fin de saber cuánto se aproxima al óptimo un algoritmo de reemplazo determinado.

Reemplazo LRU

El **algoritmo de reemplazo de página LRU** (*Least Recently Used*) es una aproximación del óptimo. La hipótesis es que si una página no ha sido usada durante un gran periodo de tiempo, entonces probablemente tampoco será utilizada en el futuro; por lo que reemplazando la página que hace más tiempo que no se usa, nos estaríamos aproximando al algoritmo óptimo. Vamos a ilustrarlo con un ejemplo:

7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
7	7	7	2					2				2	3			2	2	2	1
	0	0	0					0				0	0			0	7	7	7
		1	1					4				1	1			1	1	0	0
F	**F**	**F**	**F**					**F**				**F**	**F**			**F**	**F**	**F**	**F**
11 fallos de página																			

Este algoritmo se considera bastante eficiente cuando se aplica al reemplazo de páginas, por lo que es utilizado con frecuencia.

Un detalle significativo es que necesita asociar cada página con el momento en que fue utilizada por última vez. Aunque se pueden utilizar interrupciones para implementarlo por software —monitorizando cada acceso a la memoria— esta solución es muy ineficiente, puesto que se necesitaría actualizar algunos datos en la memoria en cada referencia a una página. Por ello el reemplazo LRU es inconcebible sin apoyo del hardware.

Son posibles dos implementaciones:

- Utilizando contadores:

  1. La CPU debe tener un reloj lógico o contador que se incrementa con cada referencia a la memoria.
  2. A cada página se añade un campo de **instante de uso**, que se almacena en la entrada de la tabla de páginas correspondiente.
  3. Cuando una página es referenciada, el valor del contador de la CPU se almacena en el campo de **instante de uso** de dicha página.

  Esta implementación requiere que se haga una escritura en la memoria con cada referencia. Además de una búsqueda por toda la tabla de páginas para localizar la página LRU.

- Utilizando una pila:

    1. Se utiliza una pila de números de página.
    2. Si se referencia una página, se quita el número correspondiente de la mitad de la pila y se inserta arriba. Debido a esto, lo mejor es implementarla como una lista doblemente enlazada.

- La actualización de la pila —que debe realizarse en cada referencia— tiene mayor coste que en la implementación anterior, pues puede ser necesario cambiar hasta 6 punteros. Sin embargo, no es necesario realizar ninguna búsqueda para seleccionar la víctima del reemplazo, puesto que esta se puede extraer directamente del final de la pila.
- Es una estrategia ideal para ser implementada en software o microcódigo.

El **algoritmo reemplazo LRU** pertenece a una clase denominada **algoritmos de pila**, que nunca se ven afectados por la **anormalidad de Belady**. En estos algoritmos, las páginas en memoria en un instante dado para N marcos son un subconjunto de las que podría haber con $N + 1$ marcos. Concretamente, en el **algoritmo LRU** las páginas en los marcos son las N páginas más referenciadas recientemente. Si el número de marcos aumentase, estas N páginas seguirán estando en memoria, pues siguen siendo las referenciadas más recientemente.

Reemplazo LRU aproximado

Pocos sistemas tienen soporte para utilizar el **algoritmo de reemplazo de página LRU**. Incluso en algunos casos no hay soporte de ningún tipo, por lo que no queda más remedio que utilizar el reemplazo FIFO.

Sin embargo, muchos proporcionan algo de ayuda en la forma de un **bit de referencia**:

- Cada entrada de la tabla de páginas tiene un **bit de referencia**.
- El hardware pone a 1 el **bit de referencia** cada vez que se referencia a una página.

Con el **bit de referencia** no podemos saber con exactitud el instante, pero sí que una página ha sido referenciada recientemente. Utilizándolo, podemos implementar diversas aproximaciones al algoritmo LRU.

Reemplazo NRU

En el **algoritmo de reemplazo de página NRU** se utiliza el **bit de referencia** de la siguiente manera:

1. En intervalos regulares de tiempo, todos los **bits de referencia** son puestos a cero por el sistema operativo.
2. Según los procesos se van ejecutando, el **bit de referencia** asociado a cada página se pone a 1 por el hardware, al ser referenciadas.
3. Cuando haya que escoger una página para ser reemplazada, se intenta seleccionar una que no haya sido referenciada —es decir, con el bit de referencia a 0—.
4. En caso de que haya varias alternativas entre las que elegir se puede utilizar el **algoritmo de reemplazo de página FIFO** o escoger un aleatoriamente.

Vamos a ilustrarlo con un ejemplo:

- Utilizaremos 3 marcos de página.
- Indicaremos el valor del bit de referencia con un superíndice junto al número de página.

No debemos olvidar que en caso de que ocurra un fallo de página, después de la carga de la página se reinicia la instrucción que generó dicho fallo. Por lo tanto, habrá un acceso que pondrá el **bit de referencia a 1**.

- Marcaremos con una flecha el instante de tiempo en el que todos los bits de referencia se ponen a cero.

En un sistema operativo real los bits son desplazados en intervalos fijos de tiempo. Pero esto no tiene que coincidir con una cantidad fija de referencias en la traza, puesto que hemos eliminado las referencias consecutivas a una misma página.

- En caso de coincidencia, utilizaremos el **reemplazo FIFO** para seleccionar la víctima.

				↓			↓			↓				↓			↓		
7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
7^1	7^1	7^1	2^1	2^0	2^1		2^1	2^1	2^1	2^1	2^1	2^1	3^1	3^0	3^0	2^1	2^0	2^0	2^0
	0^1	0^1	0^1	0^1	0^1		0^0	4^1	4^1	4^0	4^0	1^1	1^1	1^0	1^1	1^1	7^1	7^1	7^1
		1^1	1^1	1^0	1^0		1^0	1^0	0^1	0^0	0^1	0^1	0^1	0^1	0^1	0^1	0^0	0^1	1^1
F	**F**	**F**	**F**					**F**	**F**			**F**	**F**			**F**	**F**		**F**
																			11 fallos de página

Se puede mejorar el algoritmo anterior considerando tanto el **bit de referencia** como el **bit de modificado**, para clasificar las páginas en distintas categorías y escoger una página en la mejor categoría. Este es el tipo más común de **algoritmo de reemplazo de página NRU**, pero lo veremos en el Apartado 17.4.1.5.4 bajo el nombre de **Algoritmo de la segunda oportunidad mejorado**.

Con bits de referencia adicionales

Podemos obtener información adicional sobre el orden en que se realizan las referencias, guardando los **bits de referencia** en intervalos periódicos de tiempo. El **reemplazo LRU aproximado con bits de referencia adicionales**, o de **envejecimiento**, consiste en lo siguiente:

1. Aparte del **bit de referencia**, cada página tiene un conjunto de **bits de referencia adicionales** —por ejemplo, 8 bits— que se guardan en alguna tabla interna que mantiene el sistema operativo.
2. A intervalos regulares —por ejemplo, cada 100 ms— el sistema operativo desplaza los **bits de referencia adicionales** de cada página; insertando el **bit de referencia** de la página en el bit de orden más alto y descartando el bit de orden más bajo.
3. Cuando haya que escoger una página para ser reemplazada se escoge la que tenga el valor más pequeño en el conjunto de **bits de referencia adicionales**. Esto es así, puesto que dicho

conjunto contiene la historia de referencias de la página. Por ejemplo, {1 1 0 0 0 1 0 0} es más reciente que {0 1 1 1 0 1 1 1}.

4. Se puede utilizar el **algoritmo de reemplazo de página FIFO** o escoger un aleatoriamente, si dos páginas tienen el mismo valor.

Vamos a ilustrarlo con un ejemplo:

- Utilizaremos 3 marcos de página.
- Utilizaremos 2 bits adicionales.
- Indicaremos el valor del **bit de referencia** y de los **bits adicionales** con un superíndice junto al número de página.
- Marcaremos con una flecha el instante de tiempo en el que los bits son desplazados por el sistema operativo.
- En caso de coincidencia, utilizaremos el **reemplazo FIFO** para seleccionar la víctima.

				↓			↓			↓				↓			↓		
7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
$7^{1\|00}$	$7^{1\|00}$	$7^{1\|00}$	$2^{1\|00}$	$2^{0\|10}$	$2^{1\|10}$	$2^{1\|10}$	$2^{1\|11}$	$2^{1\|11}$	$2^{1\|11}$	$2^{1\|11}$	$2^{1\|11}$	$2^{1\|11}$	$2^{1\|11}$	$2^{0\|11}$	$2^{0\|11}$	$2^{1\|11}$	$2^{0\|11}$	$2^{0\|11}$	$2^{0\|11}$
	$0^{1\|00}$	$0^{1\|00}$	$0^{1\|00}$	$0^{1\|10}$	$0^{1\|10}$	$0^{1\|10}$	$0^{0\|11}$	$0^{0\|11}$	$0^{1\|11}$	$0^{0\|11}$	$0^{1\|11}$	$0^{1\|11}$	$0^{1\|11}$	$0^{1\|11}$	$0^{1\|11}$	$0^{1\|11}$	$0^{0\|11}$	$0^{1\|11}$	$0^{1\|11}$
		$1^{1\|00}$	$1^{1\|00}$	$1^{0\|10}$	$1^{0\|10}$	$1^{0\|10}$	$1^{0\|01}$	$4^{1\|00}$	$4^{1\|00}$	$4^{0\|10}$	$4^{0\|10}$	$1^{1\|00}$	$3^{1\|00}$	$3^{0\|10}$	$1^{1\|00}$	$1^{1\|00}$	$7^{1\|00}$	$7^{1\|00}$	$1^{1\|00}$
F	**F**	**F**	**F**					**F**				**F**	**F**		**F**		**F**		**F**
																			10 fallos de página

El número de **bits adicionales** puede variar de una implementación a otra, pero en cualquier caso debe ser seleccionado para realizar la actualización lo más rápidamente posible, teniendo en cuenta las características del hardware. En un caso extremo, el número de **bits de referencia adicionales** podría ser cero, dejando solo el **bit de referencia**. A este algoritmo se lo conoce como el **algoritmo de la segunda oportunidad**.

Algoritmo de la segunda oportunidad

El **algoritmo de reemplazo de la segunda oportunidad** o del **reloj** es un **algoritmo de reemplazo de página FIFO**, pero donde una página es seleccionada considerando el **bit de referencia**:

1. Cuando es necesario seleccionar una víctima para reemplazo se extrae una página de la cola FIFO. Esta cola contiene todas las páginas con marcos asignados y en el orden en que fueron cargadas, como ocurre con el **algoritmo FIFO de reemplazo**.
2. Si el **bit de referencia** está a 0, se utiliza esta página para reemplazo.
3. Si el **bit de referencia** está a 1:
    a. Se pone el **bit de referencia** a 0 y se vuelve a insertar la página en el final de la cola.
    b. Se extrae la siguiente página del principio de la cola y se vuelve al punto 2.

De este esquema podemos destacar algunos aspectos:

- Una página a la que se le da la segunda oportunidad, no será reemplazada hasta que no se le dé la segunda oportunidad a todas las demás; siempre que no sea referenciada antes y el **bit de referencia** se vuelva a poner a 1.
- En el peor de los casos, cuando todas las páginas tienen sus bits a uno, degenera en un reemplazo FIFO.

Vamos a ilustrarlo con un ejemplo:

- Utilizaremos 3 marcos de página.
- Indicaremos el valor del **bit de referencia** con un superíndice junto al número de página en el marco.
- Indicaremos el principio de la cola con una flecha junto al número de página.

7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
7^1	7^1	$\rightarrow 7^1$	2^1	2^0	2^1		2^1	$\rightarrow 2^1$	$\rightarrow 2^1$	$\rightarrow 2^1$	$\rightarrow 2^1$	1^1	1^1	$\rightarrow 1^1$	$\rightarrow 1^1$	2^1	2^1	2^1	$\rightarrow 2^0$
$\rightarrow$	0^1	0^1	$\rightarrow 0^0$	$\rightarrow 0^1$	$\rightarrow 0^1$		$\rightarrow 0^1$	0^0	0^1	0^1	0^1	$\rightarrow 0^0$	3^1	3^1	3^1	$\rightarrow 3^0$	7^1	7^1	7^0
	$\rightarrow$	1^1	1^0	1^0	1^0		1^0	4^1	4^1	4^1	4^1	4^0	$\rightarrow 4^0$	0^1	0^1	0^1	$\rightarrow 0^1$	$\rightarrow 0^1$	1^1
F	**F**	**F**	**F**					**F**	**F**			**F**	**F**			**F**	**F**		**F**
																			11 fallos de página

Algoritmo de la segunda oportunidad mejorado

Se puede mejorar el **algoritmo de la segunda oportunidad** considerando tanto el **bit de referencia** como el **bit de modificado**. Algunos autores denominan a este algoritmo como **algoritmo de la segunda oportunidad mejorado**, mientras otro lo llaman **NRU**, ya que es una versión mejorada del algoritmo del Apartado 17.4.1.5.1.

Con esos dos bits, el sistema operativo clasifica las páginas en una de las siguientes cuatro clases:

1. (0,0) ni recientemente usado ni modificado. Las páginas de esta clase son las mejores para ser reemplazadas.
2. (0,1) no usado recientemente pero modificado. No es una buena elección, puesto que hay que escribir primero la página al disco antes del reemplazo.
3. (1,0) recientemente usado pero no modificado. Probablemente será usada de nuevo en un corto espacio de tiempo.
4. (1,1) usada y modificada. Será utilizada pronto y la página tendría que ser escrita a disco para ser reemplazada.

Cuando el reemplazo de página es invocado:

- Se examina la clase a la que pertenece cada página y se reemplaza una página en la clase de menor importancia que no esté vacía. Indudablemente, deberemos examinar la lista varias veces antes de encontrar la página que debe ser reemplazada.
- A intervalos regulares los **bits de referencia** de todas las páginas son puestos a cero por el sistema operativo.

En un sistema real, el sistema operativo escribe las páginas modificadas en el almacenamiento secundario cuando está desocupado y luego pone el **bit de modificado** a 0.

Vamos a ilustrarlo con un ejemplo:

- Utilizaremos 3 marcos de página.
- El que la referencia a la memoria sea para lectura **R** o escritura **W** vendrá indicado junto al número de página en la traza.
- Marcaremos con una flecha el instante de tiempo en el que todos los bits de referencia se ponen a cero.
- Indicaremos el valor del **bit de referencia** y del **bit de modificado** con un superíndice junto al número de página en el marco.
- Indicaremos el principio de la cola con una flecha junto al número de página. Sirve para mantener un orden en las páginas, de tal forma que si hay varios candidatos de la misma categoría, se escoja el primero encontrado. Si, por ejemplo, la elección en caso de varios candidatos es aleatoria, no hace falta ese puntero.

				□			□			□				□			□		
7r	**0r**	**1w**	**2r**	**0r**	**2r**	**0w**	**2r**	**4w**	**0w**	**2r**	**0r**	**1r**	**3w**	**0r**	**1r**	**2w**	**7w**	**0r**	**1r**
7^{10}	7^{10}	$\rightarrow_0 7^1$	2^{10}	2^{00}	2^{10}	2^{10}	2^{10}	2^{10}	$\rightarrow_0 2^1$	$\rightarrow_0 2^1$	$\rightarrow_0 2^1$	2^{10}	3^{11}	3^{01}	3^{01}	2^{11}	2^{01}	2^{01}	1^{10}
→	0^{10}	0^{10}	$\rightarrow_0 0^1$	$\rightarrow_0 0^1$	$\rightarrow_0 0^1$	$\rightarrow_1 0^1$	$\rightarrow_1 0^0$	4^{11}	4^{11}	4^{01}	4^{01}	1^{10}	$\rightarrow_0 1^1$	$\rightarrow_0 1^0$	$\rightarrow_0 1^1$	$\rightarrow_0 1^1$	7^{11}	7^{11}	$\rightarrow_1 7^1$
	→	1^{11}	1^{11}	1^{01}	1^{01}	1^{01}	1^{01}	$\rightarrow_1 1^0$	0^{11}	0^{01}	0^{11}	$\rightarrow_1 0^1$	0^{11}	0^{11}	0^{11}	0^{11}	$\rightarrow_1 0^0$	$\rightarrow_1 0^1$	0^{11}
F	**F**	**F**	**F**					**F**	**F**			**F**	**F**			**F**	**F**		**F**
11 fallos de página																			

Reemplazo basado en contador

Otros algoritmos utilizan un contador del número de referencias que son realizadas a cada página. En esos casos, la elección de la víctima se puede realizar utilizando dos esquemas: la que tiene el valor mayor o el menor. Ninguno de los dos es muy común, ya que su implementación es costosa y no son una buena aproximación del óptimo.

Por lo general, la actualización del contador no la realiza la CPU, ya que leer la entrada de la página, incrementar el contador y volver a guardar la entrada, tiene un coste importante. En su lugar:

1. El contador de cada página se guarda en una tabla interna.
2. Periódicamente el sistema operativo examina el **bit de referencia** de cada página y si está a 1, incrementa el contador de la página correspondiente.

El mayor problema es que permite hacer el seguimiento de la frecuencia con la que se usan las páginas, pero no tiene en cuenta el periodo durante el que se usan. Por ejemplo, los procesos

durante su inicialización pueden usar intensamente ciertas páginas y después no necesitarlas más. Debido a que esas páginas han sido utilizadas intensamente, tiene un contador de referencias con un valor muy alto, por lo que son mantenidas en memoria aunque no vaya a ser utilizadas.

La solución es utilizar el **algoritmo LRU aproximado con bits de referencia adicionales** (véase Apartado 17.4.1.5.2) porque tiene un coste muy similar a este y prioriza las usadas más recientemente sobre las que fueron usadas con mucha frecuencia en el pasado.

Reemplazo LFU

En el **algoritmo de reemplazo de página LFU** (*Least Frequently Used*) o **NFU** (*Not Frequently Used*) se escoge la página con el contador más bajo. Esto es así, puesto que suponemos que las páginas menos referenciadas son las que no se están utilizando de forma más activa.

Vamos a ilustrarlo con un ejemplo:

- Indicaremos el valor de los contadores con un superíndice junto al número de página.
- En caso de coincidencia, utilizaremos el reemplazo **FIFO** para seleccionar la víctima.

7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
7^1	7^1	7^1	2^1	2^0	2^2	2^2	2^3	2^3	2^3	2^4	2^4	2^4	2^4	2^4	2^4	2^5	2^5	2^5	2^5
	0^1	0^1	0^1	0^2	0^2	0^3	0^3	0^3	0^4	0^4	0^5	0^5	0^5	0^6	0^6	0^6	0^6	0^7	0^7
		1^1	1^1	1^1	1^1	1^1	1^1	4^1	4^1	4^1	0^1	1^1	3^1	3^1	1^1	1^1	7^1	7^1	1^1
F	**F**	**F**	**F**					**F**			**F**	**F**	**F**		**F**	**F**	**F**		**F**
12 fallos de página																			

Reemplazo MFU

En el **algoritmo de reemplazo de página MFU** (*Most Frequently Used*), se escoge la página con el contador más alto. Se basa en el argumento de que la página con el contador más pequeño probablemente acaba de ser traída, por lo que aún no ha sido utilizada.

Vamos a ilustrarlo con un ejemplo:

- Indicaremos el valor de los contadores con un superíndice junto al número de página.
- En caso de coincidencia, utilizaremos el reemplazo **FIFO** para seleccionar la víctima.

7	**0**	**1**	**2**	**0**	**2**	**0**	**2**	**4**	**0**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**7**	**0**	**1**
7^1	7^1	7^1	2^1	2^1	2^2	2^2	2^3	2^3	0^1	0^1	0^2	1^1	1^1	1^1	1^2	2^1	2^1	2^1	2^1
	0^1	0^1	0^1	0^2	0^2	0^3	0^3	4^1	4^1	4^1	4^1	4^1	3^1	3^1	3^1	3^1	7^1	7^1	7^1
		1^1	1^1	1^1	1^1	1^1	1^1	1^1	1^1	2^1	2^1	2^1	2^1	0^1	0^1	0^1	0^1	0^2	1^1
F	**F**	**F**	**F**					**F**	**F**	**F**		**F**	**F**	**F**		**F**	**F**		**F**
13 fallos de página																			

Por extraña que parezca esta política, suele ser más eficiente que el **LRU** cuando se utiliza en las

aplicaciones de almacenamiento de datos, porque algunas las páginas se utilizan intensamente durante breves periodos de tiempo, pero están un tiempo sin utilizarse.

17.4.2. Algoritmos de buffering de páginas

Existen otros procedimientos que pueden ser utilizados, junto con alguno de los **algoritmos de reemplazo** comentados, con el objetivo de mejorar su eficiencia. Estos procedimientos se agrupan dentro de lo que se denomina **algoritmos de *buffering* de páginas**.

- Se puede mantener una lista de marcos libres. Cuando se produce un **fallo de página** se escoge un marco de la lista y se carga la página, al tiempo que se selecciona una página como víctima y se copia al disco.

  Esto permite que el proceso se reinicie lo antes posible, sin esperar a que la página reemplazada sea escrita en el disco. Posteriormente, cuando la escritura finalice, el marco es incluido en la lista de marcos libres.

- Recordar qué página estuvo en cada marco antes de que este pasara a la lista de marcos libres, sería una mejora de lo anterior. De esta forma las páginas podrían ser recuperadas directamente desde la lista, si fallara alguna antes de que su marco sea utilizado por otra página.

  Esto permite reducir los efectos de que el **algoritmo de reemplazo** escoja una víctima equivocada.

- Se puede mantener una lista de páginas modificadas y escribirlas cuando el dispositivo del espacio de intercambio no esté ocupado.

  Este esquema aumenta la probabilidad de que una página no esté marcada como modificada —con el **bit de modificado**— cuando sea seleccionada por el algoritmo de reemplazo, evitando tener que hacer en ese momento la escritura en disco.

17.4.3. Reemplazo local frente a global

Cuando un proceso necesita un marco, el algoritmo de reemplazo puede, tanto extraerlo de cualquier proceso, como ser obligado a considerar solo aquellas páginas que pertenecen al proceso que generó el fallo. Eso permite clasificar los algoritmos de reemplazo en dos categorías:

- En el **reemplazo local** solo se pueden escoger marcos de entre los asignados al proceso. Por tanto:
  - El número de marcos asignados a un proceso no cambia porque ocurran **fallos de página**.
  - El mayor inconveniente es que un proceso no puede hacer disponible a otros procesos los marcos de memoria que menos utiliza.
- En el **reemplazo global** se pueden escoger marcos de entre todos los del sistema, independientemente de que estén asignados a otro proceso o no. Por tanto:
  - El número de marcos asignados a un proceso puede aumentar si durante los **fallos de página** se seleccionan marcos de otros procesos.
  - El mayor inconveniente es que los procesos no pueden controlar su **tasa de fallos de**

página, puesto que esta depende del comportamiento de los otros procesos, pudiendo afectar a su tiempo de ejecución de forma significativa.

Generalmente, el **reemplazo global** proporciona mayor rendimiento, por lo que es el método más utilizado.

17.5. Asignación de marcos de página

La cuestión que queda por resolver es cómo repartir los marcos de memoria física libre entre los diferentes procesos, con el fin de cubrir las necesidades de reemplazo de cada uno de ellos. Posibles soluciones a esto serían: repartir la memoria por igual entre todos los procesos o hacerlo en proporción a la cantidad de memoria virtual que utilizan.

Sin embargo, intuitivamente parece interesante intentar estimar de alguna manera el **mínimo número de marcos** que realmente necesita cada proceso. Así, si a cada proceso se le proporciona el número mínimo de marcos necesario, el sistema podría disponer de memoria libre para aumentar el número de procesos —aumentando el uso de la CPU— o para dedicarla a otras funciones —como es el caso de los búferes y las cachés de E/S—.

El **mínimo número de marcos** viene establecido por diversos factores:

- Cuando ocurre un fallo de página, la instrucción que la ha provocado, debe ser reiniciada después de cargar la página en un marco libre. Por lo tanto, un proceso debe disponer de suficientes marcos como para guardar todas las páginas a las que una única instrucción pueda acceder pues, de lo contrario, el proceso nunca podría ser reiniciado al fallar permanentemente en alguno de los accesos a memoria de la instrucción. Obviamente, este límite viene establecido por la arquitectura de la máquina.
- Todo proceso tiene una cierta cantidad de páginas que en cada instante son utilizadas frecuentemente. Si el proceso no dispone de suficientes marcos como para alojar dichas páginas, generará fallos de página con demasiada frecuencia. Esto afecta negativamente al rendimiento del sistema, por lo que es conveniente que el sistema asigne al número de marcos necesario para que eso no ocurra.

En general, si se va reduciendo el número de marcos asignados a un proceso, mucho antes de haber alcanzado el mínimo establecido por la arquitectura, el proceso dejará de ser útil debido a la elevada **tasa de fallos de página**, que será mayor cuanto menos marcos tenga asignados. Cuando eso ocurre se dice que el proceso está **hiperpaginando**.

17.6. Hiperpaginación

Se dice que un proceso sufre de **hiperpaginación** cuando gasta más tiempo paginando que ejecutándose.

17.6.1. Causas de la hiperpaginación

En los primeros sistemas multiprogramados que implementaron la paginación bajo demanda, era posible que se diera el siguiente caso:

1. El sistema operativo monitorizaba el uso de la CPU. Si el uso de la misma era bajo, se cargaban nuevos procesos desde la **cola de entrada** para aumentar el número de procesos ejecutándose al mismo tiempo —también llamado **grado de multiprogramación** en esos sistemas—.
2. Si un proceso necesitaba demasiada memoria, le podía quitar los marcos a otro, puesto que se utilizaba un **algoritmo de reemplazo global**. Esto podía ocasionar que aumentara la **tasa de fallos de página** del proceso que perdía los marcos.
3. Al aumentar los **fallos de página** el uso de la CPU decrecía, por lo que el sistema operativo cargaba más procesos para aumentar el **grado de multiprogramación** y con ello el uso de la CPU.
4. Esto reducía la cantidad de memoria disponible para cada proceso, lo que aumentaba la **tasa de fallos de páginas**, que nuevamente reducía el uso de la CPU.
5. Este mecanismo iteraba hasta reducir considerablemente el rendimiento del sistema.

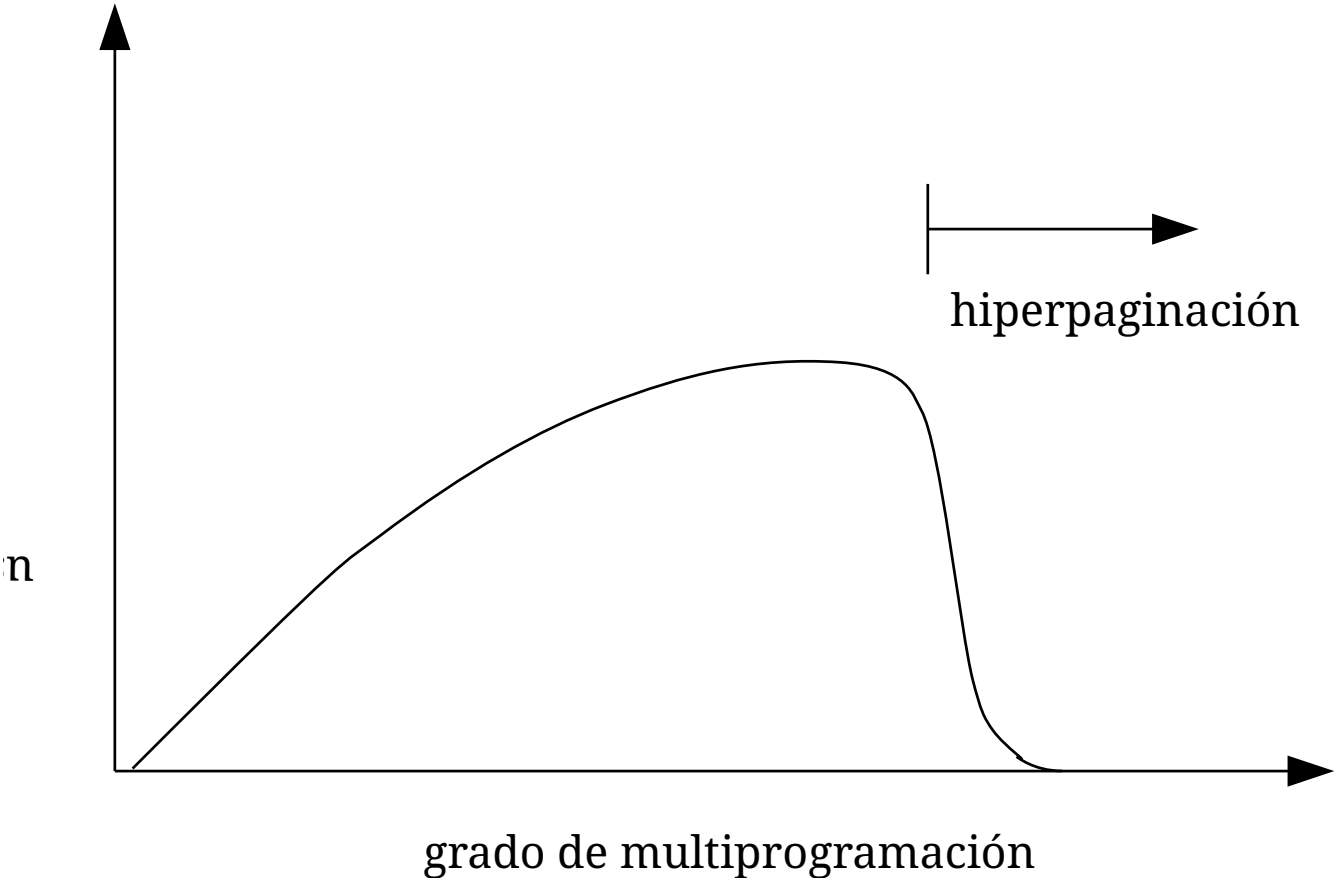


Figura 61. Hiperpaginación en sistemas multiprogramados.

El fenómeno comentado se ilustra en la Figura 61, donde se muestra el uso de la CPU frente al número de procesos cargados en el sistema. Cuando esto último aumenta, el uso de la CPU aumenta hasta alcanzar un máximo. Si el **grado de multiprogramación** supera dicho punto, el sistema comienza a **hiperpaginar**, por lo que el uso de la CPU disminuye bruscamente. Por lo tanto, si el sistema está **hiperpaginando**, es necesario reducir el **grado de multiprogramación** con el objetivo de liberar memoria.

En los sistemas operativos modernos ocurre algo parecido a lo descrito para los sistemas multiprogramados, aunque sin el efecto en cadena ocasionado por el intento del planificador de largo plazo de maximizar el uso de la CPU, ya que estos sistemas carecen de dicho planificador. Sea como fuere, en ambos casos, los procesos **hiperpaginan** si no se les asigna un número suficiente de

marcos, haciendo que sea imposible utilizarlos.

17.6.2. Soluciones a la hiperpaginación

Para el problema de la **hiperpaginación** existen diversas soluciones:

- Utilizar un algoritmo de reemplazo local, pues de esta manera un proceso que **hiperpagina** no puede afectar a otro.

  Sin embargo, esto no es cierto del todo. El uso intensivo del **dispositivo de intercambio** podría afectar al rendimiento del sistema, al aumentar el **tiempo de acceso efectivo** al disco.

- Proporcionar a un proceso tantos marcos como le hagan falta. Como ya hemos comentado en diversas ocasiones, para evitar la **hiperpaginación** es necesario asignar al proceso al menos un número mínimo de marcos, que a priori no es conocido. Una de las estrategias que pretenden estimar dicho número es el **modelo de conjunto de trabajo**.

17.6.3. Modelo del conjunto de trabajo

Para entender el modelo de conjunto de trabajo es necesario comenzar definiendo el **modelo de localidad**. El **modelo de localidad** establece que:

- Una **localidad** es un conjunto de páginas que se utilizan juntas.
- Cuando un proceso se ejecuta, se va moviendo de una **localidad** a otra.

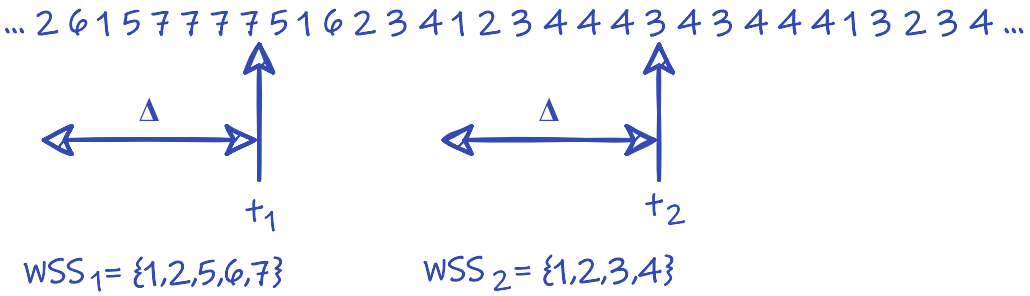
Por ejemplo, cuando se invoca una función se define una nueva **localidad**. En esta **localidad** las referencias a la memoria se realizan: al código de la función, a las variables locales de la misma y a algunas variables globales del programa.

Supongamos que proporcionamos a un proceso suficientes marcos como para alojar toda su **localidad** en un momento dado. Entonces, el proceso generará **fallos de página** hasta que todas las páginas de su **localidad** estén cargadas. Después de eso no volverá a fallar hasta que no cambie a una nueva **localidad**. Sin embargo, si damos al proceso menos marcos de los que necesita su **localidad**, este **hiperpaginará**.

El **modelo de conjunto de trabajo** es una estrategia que permite obtener una aproximación de la **localidad** del programa y consiste en lo siguiente:

- Definir el parámetro Δ como el tamaño de la **ventana del conjunto de trabajo**.
- En un instante dado, el conjunto de páginas presente en las Δ últimas referencias a la memoria se consideran el **conjunto de trabajo**.
- Por lo tanto, el **conjunto de trabajo** es una aproximación de **localidad** del programa.

Por ejemplo, dada la siguiente lista de referencias a páginas en la memoria:



si Δ = 10 referencias a la memoria, entonces el conjunto de trabajo en t_1 es {1, 2, 5, 6, 7}. Mientras que en t_2 el conjunto de trabajo es {1, 2, 3, 4}.

Obviamente, la precisión del **conjunto de trabajo** como aproximación de la **localidad** del programa depende del parámetro Δ. Por ejemplo:

- Si Δ es muy pequeña, el **conjunto de trabajo** no cubriría toda la **localidad**.
- Si Δ es muy grande, el **conjunto de trabajo** se superpondría a varias **localidades**.

17.6.4. Uso del conjunto del trabajo para evitar la hiperpaginación

El uso del conjunto de trabajo es bastante sencillo:

1. Los diseñadores del sistema seleccionan Δ.
2. El sistema operativo monitoriza el **conjunto de trabajo** de cada proceso y le asigna tantos marcos como páginas haya en el **conjunto de trabajo**.
3. Si sobran suficientes marcos otro proceso puede ser iniciado —en el caso de los sistemas multiprogramados— o se puede destinar la memoria libre a otros usos.
4. Si el **tamaño del conjunto de trabajo** total *WSS* crece y excede el número de marcos disponibles, el sistema podría seleccionar un proceso para ser suspendido. Este podrá volver a ser reiniciado más tarde.

Donde el **tamaño del conjunto de trabajo** *WSS* es la suma del **tamaño de los conjuntos de trabajo** WSS_i para cada proceso i:

$$WSS = \sum WSS_i$$

y representa la demanda total de marcos. Por eso, si *WSS* es mayor que el número de marcos disponibles, habrá **hiperpaginación**.

El sencillo algoritmo anterior permite evitar la **hiperpaginación**. Sin embargo, el problema está en cómo mover la **ventana del conjunto de trabajo** en cada referencia, con el fin de volver a calcular el **conjunto de trabajo**. Una posible aproximación sería utilizar un temporizador que periódicamente invocase a una función encargada de examinar el **bit de referencia** de las páginas en la ventana Δ. Es de suponer que las páginas con el **bit de referencia** a 1 forman parte de la **localidad** del programa y por tanto serán el **conjunto de trabajo** a lo largo del siguiente periodo.

17.7. Otras consideraciones

Ya hemos comentado que las principales decisiones que deben ser tomadas en el diseño de un sistema con **paginación bajo demanda** son la elección del **algoritmo de reemplazo** y la de la **asignación de marcos de página**. Sin embargo, hay otras consideraciones que deben ser tenidas en cuenta.

17.7.1. Prepaginado

El **prepaginado** es una técnica que consiste en cargar múltiples páginas junto con la página demandada en cada **fallo de página**.

Esas otras páginas se escogen especulativamente bajo la hipótesis de que van a ser necesitadas por el proceso en un corto espacio de tiempo. De manera que si la predicción es acertada, la **tasa de fallos de página** se reduce significativamente. Esta técnica puede ser utiliza, por ejemplo, en las siguientes situaciones:

- En la **paginación bajo demanda pura**, el sistema sabe de antemano que cuando se inicia un proceso siempre fallan las primeras páginas de código, por lo que son buenas candidatas para el **prepaginado**.
- En el acceso secuencial a **archivos mapeados en memoria**.

  El sistema puede determinar que el acceso va a ser de tipo secuencial, tanto mediante el uso de técnicas heurísticas como mediante las opciones indicadas por el proceso en la llamada al sistema con la que se abrió el archivo.

  En cualquier caso, si el sistema determina que el acceso al archivo es secuencial, en cada **fallo de página** puede cargar tanto la página demanda como las siguientes, en previsión de que vayan a ser utilizadas por el proceso.

En general, el único inconveniente del **prepaginado** es que debe ser ajustado para que el coste del mismo sea inferior al de servir los **fallos de página**.

17.7.2. Aplicaciones en modo RAW

Algunas aplicaciones, cuando acceden sus datos a través de los mecanismos de memoria virtual del sistema operativo, ofrecen peor rendimiento del que conseguirían si este mecanismo no existiera.

El ejemplo típico, son los gestores de bases de datos, que conocen sus necesidades de memoria y disco mejor que cualquier sistema operativo de propósito general, por lo que salen beneficiadas si implementan sus propios algoritmos de gestión de la memoria y de *buffering* de E/S.

Por eso muchos sistemas operativos modernos permiten que los programas que lo soliciten puedan acceder a los discos en **modo RAW**. En el **modo RAW** no hay sistema de archivos, ni paginación bajo demanda, ni bloqueo de archivos, ni prepaginación, ni muchos otros servicios del sistema operativo; por lo que dichas aplicaciones deben implementar sus propios algoritmos de almacenamiento y gestión de la memoria.

Sin embargo, hay que valorar muy bien las necesidades del programa antes de optar por este modo.

La mayor parte de las aplicaciones siempre funcionan mejor utilizando los servicios convencionales ofrecidos por el sistema operativo.

17.7.3. Tamaño de las páginas

Como ya comentamos al estudiar el método básico de paginación (véase el Capítulo 16), una decisión de diseño importante es escoger el tamaño adecuado para las páginas:

- **Con páginas grandes**:
  - Se consiguen menos **fallos de páginas**. Por ejemplo, en un caso extremo, un proceso de 100 KiB solo podría generar un **fallo de página** si cada página es de 100 KiB, pero puede generar 102400 fallos si cada página es de 1 byte.
  - Se consiguen tablas de páginas más pequeñas.
  - La E/S para acceder al contenido de cada página requiere menos tiempo.

    En general el tiempo de transferencia es proporcional a la cantidad de información transferida, lo que debería beneficiar a los sistemas con páginas de pequeño tamaño. Sin embargo, la latencia y el tiempo requerido para posicionar la cabeza lectora de los discos es muy superior al tiempo de transferencias de datos, por lo que es más eficiente tener menos transferencias de mayor tamaño —como cuando se usan páginas grandes— que más transferencias de menor tamaño —como cuando se usan páginas pequeñas—.
- **Con páginas pequeñas**:
  - Se consigue tener menos **fragmentación interna** y, por tanto, un mejor aprovechamiento de la memoria.
  - Teóricamente, se obtiene una mejor resolución para asignar y transferir al disco solo la memoria que realmente necesitamos. Esto a la larga debería redundar en menos memoria asignada y menos operaciones de E/S.

En la actualidad, el tamaño de página más común es de 4 KiB en sistemas de 32 bits y 8 KiB en los de 64 bits, ya que son adecuados para la mayor parte de las aplicaciones. Sin embargo, muchos sistemas modernos soportan el uso simultáneo de múltiples tamaños de página. Esto permite que la mayor parte de las aplicaciones utilicen el tamaño estándar, mientras las que hacen un uso intensivo de la memoria —como es el caso de los gestores de bases de datos— puedan utilizar páginas de mayor tamaño.

17.7.4. Efecto de la estructura de los programas

Los programas creados considerando la **localidad de referencia** pueden mejorar su rendimiento en los sistemas con **paginación bajo demanda**.

Vamos a ilustrarlo con el siguiente ejemplo de un programa que inicializa a 0 un *array* de 128 por 128 elementos.

```
char data[][] = new char[128][128];

for (int j = 0; j < 128; ++j)
```

```
  for (int i = 0; i < 128; ++i)
    data[i][j] = 0;
```

Un *array* como el indicado es almacenado en filas:

```
data[0][0], data[0][1], ..., data[0][127]
data[1][0], data[1][1], ..., data[127][127]
```

De manera que si suponemos que el tamaño de cada página es de 128 bytes, en el mejor de los casos cada fila estará almacenada en una página. Por lo tanto:

- Si el sistema le asigna 128 marcos o más, el proceso solo generará 128 fallos de página.
- Si el sistema operativo le asigna un solo marco, el proceso tendrá 16 384 fallos, aproximadamente.

Sin embargo, el ejemplo sería diferente si el bucle interno del programa recorriera las columnas del *array* y no las filas:

Pues se podrían a 0 primero todos los bytes de una misma página antes de empezar con la siguiente. Esto reduciría el número de **fallos de página** a 128, aunque el sistema operativo solo asigne un marco al proceso.

Por lo tanto se puede concluir que:

- La selección cuidadosa de las estructuras de datos y de programación pueden mejorar la **localidad**, reduciendo la **tasa de fallos de páginas** y el tamaño del **conjunto de trabajo**. Por ejemplo, las estructuras de datos tipo pila tienen buena **localidad**, puesto que el acceso siempre se realiza en lo alto de las mismas. Sin embargo, las tablas de dispersión, obviamente, están diseñadas para dispersar las referencias, lo que produce una mala **localidad**.
- La elección del lenguaje de programación también puede tener efecto. En los lenguajes como C y C++ se utilizan punteros con frecuencia, lo que aleatoriza el acceso a la memoria, empeorando la **localidad de referencia**. Además, algunos estudios indican que los lenguajes orientados a objetos tienden a tener peor **localidad de referencia** que los que no lo son.
- El compilador y el cargador también pueden tener un efecto importante:
  - Separando el código de los datos para permitir que las páginas de código puedan ser de solo lectura. Esto es interesante porque las páginas no modificadas no tienen que ser escritas antes de ser reemplazadas.
  - El compilador puede colocar las funciones que se llaman entre sí en la misma página.
  - El cargador puede situar las funciones en la memoria de tal forma que en lo posible no crucen los bordes de las páginas.

17.7.5. Interbloqueo de E/S

Supongamos que un proceso solicita una operación de E/S sobre el contenido de alguna de las páginas de su espacio de direcciones y que, antes de que la operación sea realizada, la página es

reemplazada mientras el proceso está esperando. En ese caso, la operación de E/S se podría acabar realizando sobre una página que pertenece a un proceso diferente. Para evitarlo existen diversas soluciones:

- Se puede utilizar la memoria del núcleo como búfer en las operaciones de E/S.

  En una escritura, esto obliga a la llamada al sistema a copiar los datos desde las páginas del proceso a la memoria del núcleo, antes de solicitar la operación de E/S. Mientras que en las operaciones de lectura sería justo al contrario.

- Cada página puede tener un **bit de bloqueo** que se utiliza para indicar qué páginas no pueden ser seleccionadas para reemplazo.

Además los **bits de bloqueo** se pueden utilizar en otras muchas situaciones:

- Bloquear las páginas del núcleo para evitar que sean reemplazadas.
- Bloquear las páginas que acaban de ser cargadas.

  Esto evita que un **fallo de página** en un proceso de mayor prioridad pueda reclamar el marco antes de que el proceso para el que se cargó la página originalmente sea reiniciado, desperdiciando el trabajo de cargarla y provocando un nuevo **fallo de página**.

  Para implementarlo, se puede poner el **bit de bloqueo** a 1 cuando la página se carga, volviéndolo después a poner a 0 cuando el proceso es planificado por primera vez, tras el **fallo de página** que provocó la carga de la página.

- En los sistemas con **tiempo real flexible**, se suele permitir que las tareas de tiempo real informen de cuáles son las páginas más importantes, con el fin de que sean bloqueadas para evitar que puedan ser reemplazadas.

  Para evitar riesgos, el sistema suele considerar estas solicitudes como «consejos de bloqueo». De esta manera el sistema es libre de descartar dichos consejos si el conjunto de marcos libres llega a ser demasiado pequeño o si un proceso pide bloquear demasiadas páginas.

17.8. Interfaz de gestión de la memoria

Gracias a la abstracción de las técnicas de memoria virtual —como la paginación bajo demanda— desde el punto de vista de los procesos, en cualquier sistema moderno prácticamente solo hace falta una llamada al sistema para gestionar su espacio de direcciones virtual.

En los sistemas POSIX esta llamada es mmap() y en Windows API es VirtualAlloc(), que se usan junto a sus opuestas munmap() y VirtualFree(), respectivamente.

Ambas funciones permiten:

- Reservar una porción del espacio de direcciones virtual del proceso.

  La llamada solo hace la reserva de un rango de direcciones para que pueda ser utilizado por el proceso —es decir, que las páginas en ese rango sean válidas— siendo el componente de paginación bajo demanda, el responsable de asignar la memoria física que lo respalda, cuando

el proceso acceda a esas direcciones.

- Establecer permisos —lectura, escritura y ejecución— opciones de compartición entre procesos, bloqueo de páginas en la memoria física, páginas de gran tamaño y otras opciones, en la región de memoria virtual a reservar.

Además, en los sistemas POSIX, mmap() se utiliza también para mapear archivos en regiones del espacio de direcciones virtual. Mientras que en Windows API para esa función se utilizan llamadas diferentes, como hemos visto.

Ambas funciones ofrecen una buena cantidad de funcionalidades, pero operan a muy bajo nivel. Por eso en ambas la página es la unidad mínima en la gestión de la memoria. Es decir, las regiones reservadas del espacio de direcciones virtual, siempre deben comenzar en un borde de página y su tamaño debe ser múltiplo del tamaño de página.

El problema es cómo compatibilizar eso, con las necesidades reales de los programas, que durante su ejecución necesitan reservar y liberar constantemente memoria para pequeños elementos, como: *arrays*, cadenas de texto, estructuras u objetos. Para esos casos, utilizar directamente mmap() o VirtualAlloc() no es una solución, puesto que la fragmentación interna conlleva un importante derroche de recursos.

17.8.1. Anatomía del espacio de direcciones virtual del proceso

Los procesos pueden utilizar diversas ubicaciones dentro de su espacio de direcciones virtual para almacenar los datos que necesitan para su ejecución (véase la Figura 62):

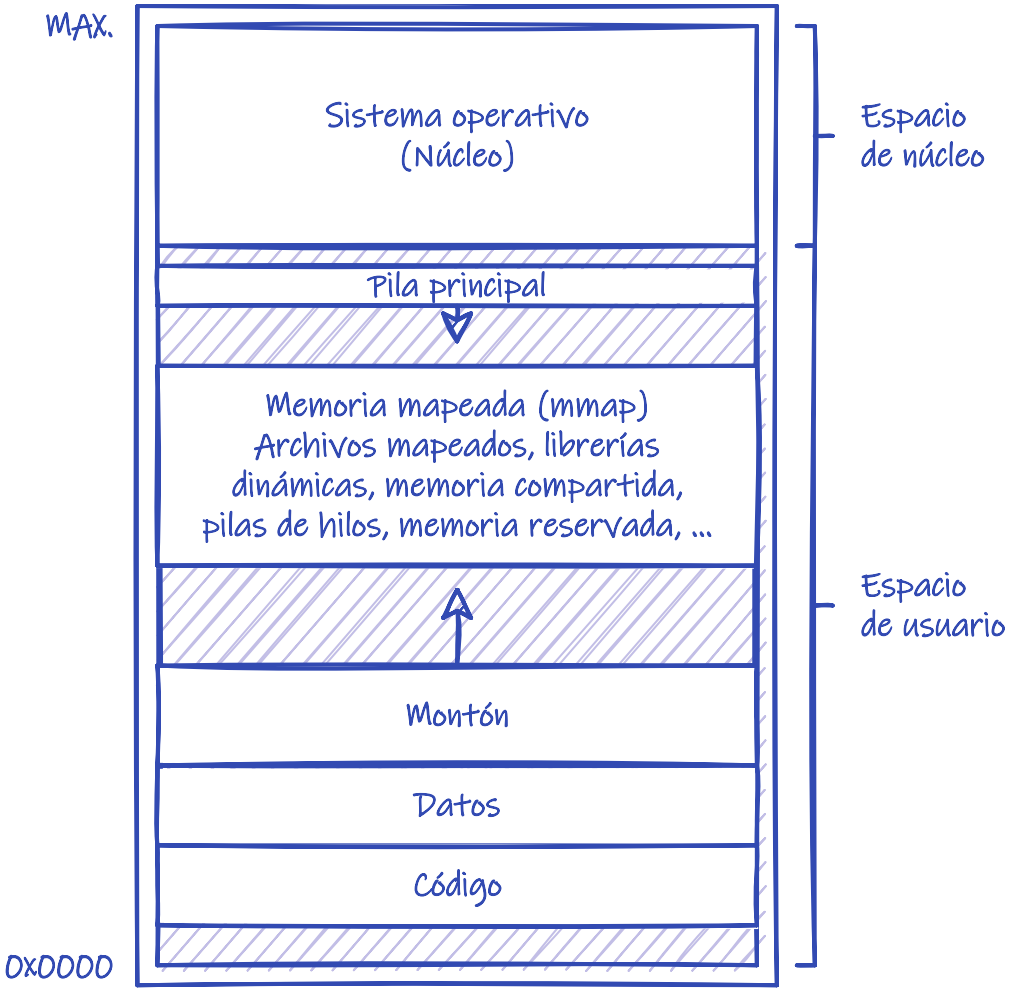


Figura 62. Anatomía de un proceso en memoria.

- Las variables y constantes globales se almacenan en el **segmento de datos**, que tiene tamaño fijo, ya que las dimensiones de estas variables se conocen de antemano en tiempo de compilación, al igual que ocurre con el código del programa.
- Las variables locales y los argumentos de las funciones se almacenan en la **pila**, junto con las direcciones de retorno para volver de las funciones.

  Esta es la ubicación ideal para ellos, puesto que al retornar de una función, la pila se restablece al estado previo al que tenía cuando se invocó dicha función, haciendo que las variables locales y argumentos desaparezcan automáticamente.

- Las variables reservadas dinámicamente —por ejemplo, usando malloc()/free() en C o new /delete en C++ o Java— se almacenan en el **montón**, que no es más una región contigua de memoria ubicada inmediatamente después del **segmento de datos** del proceso.
- En la región entre el **montón** y la **pila** se ubican los **archivos mapeados en memoria**, las regiones de **memoria compartida**, las **librerías de enlace dinámico**, las **pilas** de cada hilo —en procesos multihilo— y, en general, la memoria reservada con funciones como mmap() y VirtualAlloc().

Cada lenguaje de programación, debe proporcionar —a través de su **librería estándar**— un mecanismo en espacio de usuario, adecuado para la gestión en tiempo de ejecución de la memoria del **montón** del proceso. Para eso, cada lenguaje puede utilizar su propia implementación de dicho mecanismo o bien recurrir a la proporcionada por la **librería del sistema**.

Por ejemplo, en los sistemas POSIX, la **librería del sistema** proporciona su propia implementación, accesible a través de las funciones malloc() y free(), que es utilizada directamente por los programas escritos en C. Esta implementación hace uso de mmap(), pero ofrece mayor control sobre la cantidad de memoria que podemos reservar, como veremos en el Apartado 17.8.2.

Otros lenguajes de programación tienen otras interfaces para gestionar la memoria, pero utilizan internamente las funciones malloc() y free() de la **librería del sistema**. Sin embargo, este no es el caso ni de C++ ni de Java ni de algunos otros lenguajes. En C++, los operadores new y delete utilizan sus propios algoritmos de gestión de la memoria del **montón**, más optimizados que malloc() y free() para la creación y destrucción de objetos de cualquier tamaño de manera eficiente.

En Windows API ocurre algo similar. La **librería del sistema** proporciona su propia gestión de la memoria del **montón**, que es accesible para cualquier programa a través de las funciones HeapAlloc() y HeapFree(), y que se implementa sobre VirtualAlloc(). La **librería estándar** de C utiliza, a su vez, esas funciones para implementar malloc() y free(). Lo mismo ocurre en otros lenguajes, aunque no en todos, ya que algunos optan por implementar algoritmos más eficientes para sus casos de uso directamente sobre VirtualAlloc().

17.8.2. Gestión de la memoria del montón

Para ilustrar cómo se puede gestionar la memoria del **montón** utilizaremos como ejemplo el mecanismo empleado por la **librería del sistema** de los sistemas POSIX —accesible a través de las funciones malloc() y free(). Sin embargo, es importante tener en cuenta que esta tarea se realiza de manera muy similar en las implementaciones de otros sistemas operativos y lenguajes de programación.

El funcionamiento básico de malloc() sigue las siguientes reglas:

1. Cuando la memoria solicitada supera cierto umbral —128 KiB en sistemas GNU/Linux— es reservada directamente mediante la llamada al sistema mmap(). Eso significa que las peticiones de gran tamaño realmente no consumen espacio del **montón**, si no que se reservan del hueco entre el **montón** y la **pila**.
2. Cuando un proceso hace una petición de memoria dinámica espera que el espacio ofrecido sea continuo en el espacio de direcciones virtual, por lo que la memoria del **montón** se gestiona usando un algoritmo de **asignación contigua de memoria** (véase Apartado 15.5) y, puesto que las peticiones pueden ser de tamaño variable, se utiliza con un esquema de **particionado dinámico**.

   Es decir, que para las peticiones que no entran en el caso anterior, se busca en la tabla de huecos libres y ocupados del **montón** uno lo suficientemente como grande para atender la petición. Se asigna el espacio solicitado y el resto sigue marcado como hueco libre.

   La estrategia más común de búsqueda es el **mejor ajuste**, utilizando algún tipo de estructura de datos que mantenga los huecos libres ordenados por tamaño, para encontrar el de tamaño adecuado rápidamente.

3. Si no hay suficiente memoria libre contigua como para atender la petición, se utiliza la llamada al sistema mmap(), para extender el tamaño del **montón** reservando una nueva región

separada —a veces llamada **arena**— y comenzar a repartirla.

En aplicaciones pequeñas, algunas implementaciones intentan ampliar primero el espacio libre utilizando la llamada al sistema brk(), que sirve para extender el **montón** sobre la región adyacente no asignada del espacio de direcciones virtual del proceso. Este es el caso de la implementación estándar de malloc() en GNU/Linux.

La llamada al sistema brk() ha sido eliminada del estándar POSIX, pero algunos sistemas la mantienen por compatibilidad hacia atrás, dado que era la forma en la que tradicionalmente se ampliaba la memoria del **montón** en los primeros UNIX. En macOS esta llamada se emula con una región de 4 MiB reservada con **mmap** la primera vez que se utiliza.

17.8.3. Fragmentación

La estrategia comentada sufre de **fragmentación interna**. En las peticiones grandes, mmap() reserva en múltiplos de tamaño de página, por lo que siempre se puede perder cierta cantidad, aunque pequeña en comparación al tamaño de la región reservada. En las peticiones pequeñas, la memoria se asigna en múltiplos de una unidad mínima —por ejemplo, 16 o 32 bytes— por lo que también se puede perder cierta cantidad de memoria.

Además sufre de **fragmentación externa**, porque después de que el proceso lleva un tiempo en ejecución, liberando y reservando memoria, el espacio puede comenzar a quedar fraccionado en un gran número de pequeños huecos, obligando a la librería a buscar más espacio para el **montón**, aunque en suma haya suficiente espacio en los huecos libres.

Esto representa un reto para los desarrolladores de aplicaciones, que previsiblemente vayan a ejecutarse durante periodos muy largos de tiempo. En esos casos, es común optar por librerías externas, que implementen gestores de memoria que fragmenten menos la memoria, o soluciones basadas en alguna forma de referencias indirectas y recolección de basura, para ocasionalmente poder compactar la memoria del **montón**. Esto último, es lo que hace la máquina virtual de Java.

Parte V: Gestión del almacenamiento

Los **dispositivos de almacenamiento secundario** —como discos duros o memorias de estado sólido— organizan su espacio de almacenamiento en bloques. Es decir, a diferencia de la memoria principal, donde se pueden leer y modificar bytes individualmente, en estos dispositivos el acceso es en **bloques de bytes** —por ejemplo, en bloques de 512 bytes—.

Sin embargo, no resultaría muy ergonómico que los usuarios y los programas tuvieran que usar este espacio direccionando bloques directamente. Por eso el acceso y gestión del espacio de almacenamiento secundario se abstrae tras el concepto de **archivo**. Obviamente, para organizar un espacio diáfano de 0 a N bloques en archivos —con sus directorios y el resto de características a las que estamos acostumbrados— hacen falta ciertas estructuras de datos que se almacenan de forma persistente en el mismo espacio de almacenamiento. Es a esto a lo que denominamos **sistema de archivos**. Dedicaremos a ellos los siguientes apartados.

Chapter 18. Almacenamiento secundario

Tiempo de lectura: 10 minutos

18.1. Dispositivos de almacenamiento

Los ordenadores pueden almacenar información en diferentes soportes de almacenamiento —por ejemplo, en discos magnéticos, DVD o memorias de estado sólido—. Cada uno tiene propiedades físicas diferentes que pasamos a comentar brevemente a continuación.

18.1.1. Discos magnéticos

Los discos magnéticos son el tipo principal de almacenamiento secundario, generalmente en la forma de lo que se denominan discos duros.

Figura 63. Disco duro — Fuente: Wikipedia

Tal y como se puede apreciar en la Figura 63 cada unidad está compuesta por una serie de platos de forma circular recubiertos de material magnético. La información se almacena grabándola magnéticamente sobre los platos, para lo cual se utilizan unas cabezas de lectura que «flotan» tanto por encima como por debajo de cada plato.

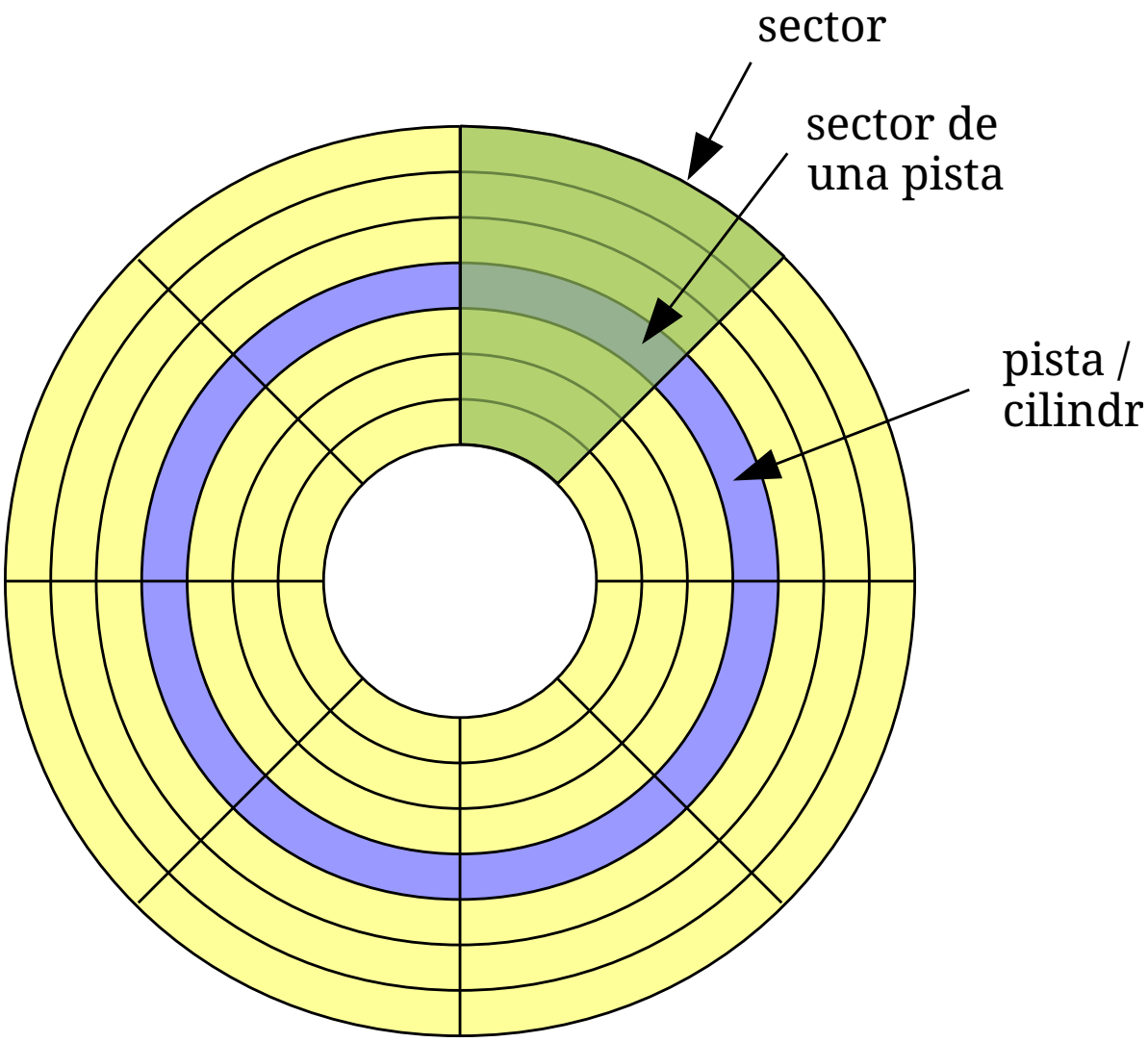


Figura 64. Estructura lógica de un disco magnético.

Desde el punto de vista lógico (véase la Figura 64) la superficie de cada plato está dividida en **pista** circulares, cada una de las cuales se subdivide en **sectores**. El conjunto de pistas formado por todas aquellas que están situadas en la misma posición en los distintos platos se denomina **cilindro**.

En estos dispositivos consume mucho más tiempo mover la cabeza de lectura hasta el sector de interés, que la lectura y transferencia de los datos almacenados a la memoria RAM. Por lo tanto, el tiempo de acceso aleatorio al disco es mucho mayor que el de acceso secuencial.

18.1.2. Discos ópticos

Los discos ópticos —CD, DVD, BluRay, o cualquier otro medio similar— consisten en un disco circular en el cual la información se almacena haciendo uso de surcos microscópicos, que se leen haciendo incidir un láser sobre una de las caras planas que lo componen.

En este tipo de discos la información se almacena siguiendo un recorrido continuo en espiral que cubre la superficie entera del disco, extendiéndose desde el interior hacia el exterior. Dado que el láser siempre debe desplazarse sobre la espiral, el acceso aleatorio a los datos es más lento que con otras tecnologías de disco.

18.1.3. Memorias de estado sólido

Una memoria de estado sólido —memoria USB o un SSD— es un dispositivo de almacenamiento que usa una memoria no volátil —como las memorias *flash*— para almacenar datos, en lugar de utilizar discos ópticos o magnéticos.

En este tipo de memorias la información se almacena como en un vector lineal de bytes, que se puede indexar aleatoriamente con la misma eficiencia con la que se accede secuencialmente —como ocurre con la memoria RAM—. Sin embargo algunos dispositivos, de cara al resto del sistema informático, emulan una interfaz y un modo de direccionamiento similar al utilizado por los discos magnéticos —es decir, usando pistas, sectores y cilindros— por temas de compatibilidad.

18.2. Archivos y sistemas de archivos

Teniendo en cuenta la gran diversidad de dispositivos de almacenamiento que existen, para que el sistema informático sea cómodo de utilizar, el sistema operativo proporciona una visión lógica uniforme de todos los sistemas de almacenamiento. Es decir, abstrae las propiedades físicas de los dispositivos de almacenamiento para definir una unidad de almacenamiento lógico que sea útil para los usuarios. Esta unidad es el **archivo**.

Un **archivo** o **fichero** es una colección de datos relacionados, identificados por un nombre, que es tratada por el sistema operativo como la unidad de información en el almacenamiento secundario. Desde la perspectiva de los usuarios, un archivo es la unidad más pequeña de almacenamiento. Es decir, los usuarios no pueden escribir datos en el almacenamiento secundario a menos que estos se encuentren dentro de un archivo.

El sistema operativo puede ofrecer esta abstracción gracias al **sistema de archivos**. Este proporciona los mecanismos para el almacenamiento de los datos y programas en archivos, tanto del propio sistema operativo como los de todos los usuarios del sistema informático.

Los sistemas de archivos están compuestos de dos partes claramente diferenciadas:

- **Una colección de archivos**, cada una de las cuales almacena una serie de datos relacionados.
- **Una colección de estructuras de metadatos**, que contienen información relativa a los archivos almacenados —nombre, ubicación en el disco, permisos, entre otros— y que se encarga de organizarlos; generalmente haciendo uso de una estructura de directorios.

18.3. Volúmenes de datos

Los dispositivos de almacenamiento comentados anteriormente pueden ser utilizados al 100% con un único sistema de archivos. Sin embargo, en ocasiones es interesante hacer divisiones con el objeto de disponer de múltiples sistemas de archivos en el mismo dispositivo. Cada una de esas divisiones es un **volumen**.

En otros casos interesa combinar divisiones o dispositivos de almacenamiento completos para crear espacios de mayor tamaño —también denominados **volúmenes**— cada una de las cuales puede albergar un único sistema de archivos. Así que en general, utilizaremos el término **volumen** para referirnos a un espacio de almacenamiento que alberga un sistema de archivos, tanto si ese

espacio es una pequeña parte del espacio completo del dispositivo, como si se trata de una estructura de mayor tamaño compuesta a partir de varios dispositivos.

A continuación comentaremos brevemente las tecnologías utilizadas con mayor frecuencia para construir estos volúmenes.

18.3.1. RAID

La tecnología **RAID** (*Redundant Array of Inexpensive Disks*) permite combinar varios discos duros para mejorar las prestaciones a través del paralelismo en el acceso o para mejorar la fiabilidad a través del almacenamiento de información redundante. En concreto se definen diversos **niveles RAID**, de entre los cuales los más comunes son:

- En un **conjunto RAID 0** se distribuyen los datos equitativamente en bloques de tamaño fijo entrelazados entre dos o más discos, sin incluir ningún tipo de información redundante. Esto permite leer y escribir más datos al mismo tiempo, ya que se pueden enviar en paralelo peticiones a los distintos discos. Sin embargo, la fiabilidad es inversamente proporcional al número de discos, ya que para que el conjunto falle basta con que lo haga cualquiera de ellos.
- En un **conjunto RAID 1** se crea una copia exacta —en espejo— de los datos en dos o más discos. El resultado es que, incluso con dos discos, se incrementa exponencialmente la fiabilidad respecto a tener uno solo, ya que para que el conjunto falle es necesario que lo hagan todos los discos. Adicionalmente, el rendimiento en las operaciones de lectura se incrementa linealmente con el número de copias, ya que los datos están disponibles en todos los discos al mismo tiempo, por lo que se pueden balancear las operaciones de lectura entre todos ellos.
- En un **conjunto RAID 5** se distribuyen los datos equitativamente en bloques de tamaño fijo entrelazados entre dos o más discos y se utiliza uno adicional para almacenar la información de paridad de los bloques de una misma división. En RAID se denomina división o *stripe* a la serie de bloques consecutivos, escogidos de cada uno de los discos del conjunto.

  El disco utilizado para almacenar el bloque de paridad cambia de forma escalonada de una división a la siguiente, de ahí que se diga que el bloque de paridad está distribuido. Algunos aspectos adicionales a tener en cuenta son que:

  - Cada vez que se escribe un bloque de datos se debe actualizar el bloque de paridad. Por lo tanto las escrituras en un conjunto RAID 5 son costosas en términos de operaciones de disco y tráfico.
  - Los bloques de paridad no se leen durante las lecturas de datos, ya que eso reduciría el rendimiento. Solo se hace en caso de que la lectura de un sector falle, puesto que el sector en la misma posición relativa dentro de cada uno de los otros bloques de datos de la división y en el bloque de paridad se pueden utilizar para reconstruir el sector erróneo.
  - En un conjunto RAID 5 el fallo de 2 discos provoca la pérdida completa de los datos. Esto significa que aunque se pueden añadir discos de manera ilimitada, eso no suele ocurrir, puesto que a más discos en el conjunto más probabilidad de que fallen dos de ellos.
- En un **conjunto RAID 6** se utiliza la misma estrategia que en RAID 5, pero en cada división hay dos bloques de paridad —en lugar de uno— en dos discos diferentes. Esto permite que fallen hasta dos discos sin perder los datos.

- En un conjunto con niveles anidados se combinan varios niveles RAID básicos como si fueran capas superpuestas. Ejemplos típicos son:
  - **RAID 0+1**, donde se hace un espejo de un conjunto RAID 0.
  - **RAID 1+0** o **RAID 10**, donde diversos conjuntos en espejo se combinan en un RAID 0, aumentando la capacidad total.
  - **RAID 50**, donde diversos conjuntos RAID 5 se combinan en un RAID 0, aumentando también la capacidad total.

La implementación de RAID es otra de las áreas donde existen diversas variantes:

- RAID puede implementarse en el hardware de la controladora de disco, de tal forma que solo los discos conectados a esta pueden formar parte de un conjunto RAID determinado. Esta solución es muy eficiente, especialmente cuando se utilizan niveles que requieren cálculo de la paridad, ya que se evita utilizar tiempo de CPU para ese trabajo. Sin embargo, estas controladoras son notablemente más caras que las que carecen de soporte para RAID.
- RAID puede implementarse dentro del sistema operativo en lo que se denomina el **software de gestión de volúmenes**. En este caso las soluciones RAID con paridad son bastante lentas, por lo que normalmente solo se soportan los niveles RAID 0, 1, 10 o 0+1. Algunas controladoras de disco modernas que dicen venir con soporte RAID realmente implementan esta tecnología en software, a nivel del controlador de dispositivo, mientras que en el hardware solo se implementan unas características de apoyo mínimas. En algunos entornos se denomina a este tipo de implementaciones *fakeRAID* o *hostRAID*.

Cada conjunto RAID se comporta como una unidad de almacenamiento independiente desde el punto de vista del resto del sistema, por lo que se puede utilizar entero para albergar un único sistema de archivos. Sin embargo, lo más común es dividirlo en regiones con el objeto de utilizar múltiples sistemas de archivos o combinarlo en estructuras de mayor tamaño, para lo cual se pueden utilizar alguna de las técnicas que veremos a continuación.

18.3.2. Particiones

Un disco, un conjunto RAID o cualquier otro dispositivo de almacenamiento se puede dividir en regiones para utilizar en cada una de ellas un sistema de archivos diferente. A esas regiones se las conoce comúnmente como **particiones**, **franjas** o **minidiscos**.

Según la plataforma, existen diversas maneras de implementar el soporte de particiones. Entre los sistemas de escritorio las tecnologías más difundidas y utilizadas son la **MBR** (*Master Boot Record*) y la **GPT** (*GUID Partition Table*). En ambas se almacena, en los primeros sectores del dispositivo de almacenamiento, una tabla con una entrada por partición donde se guardan las direcciones del primer y último sector de cada una de ellas en el dispositivo, así como otra información. Eso es todo lo que necesita el sistema operativo para determinar los límites de la región ocupada por cada sistema de archivos.

18.3.3. Volúmenes dinámicos

Según la tecnología que se utilice para particionar es posible encontrarse con una serie de restricciones comunes:

- Limitado número de particiones que puede contener un mismo dispositivo.
- Limitaciones o imposibilidad de redimensionar las particiones. Especialmente si el sistema operativo está en ejecución.
- La imposibilidad de crear particiones que hagan uso de regiones libres en diferentes dispositivos de almacenamiento.

Para resolverlo, algunos sistemas operativos incluyen un **software de gestión de volúmenes** que hace uso de tecnología propia para superar estas limitaciones. Estas herramientas generalmente permiten agrupar dispositivos completos, conjuntos RAID, particiones, etc. y sobre ellos construir los volúmenes que sean necesarios. Estos volúmenes pueden ser redimensionados —en ocasiones sin tener que detener la ejecución del sistema operativo— y en caso de que haga falta se pueden incluir dinámicamente nuevos dispositivos para incrementar el espacio disponible. Además, como ya hemos comentado, el software de gestión de volúmenes puede incluir alguna funcionalidad propia de conjuntos RAID, con el objeto de mejorar las prestaciones, a través del paralelismo en el acceso, o mejorar la fiabilidad, a través del almacenamiento de información redundante.

Chapter 19. Sistemas de archivos

Tiempo de lectura: 55 minutos

Como hemos comentado, cada volumen puede albergar un sistema de archivos. A continuación estudiaremos los elementos más comunes a la mayor parte de los sistemas de archivos actuales.

19.1. Estructura de un sistema de archivos

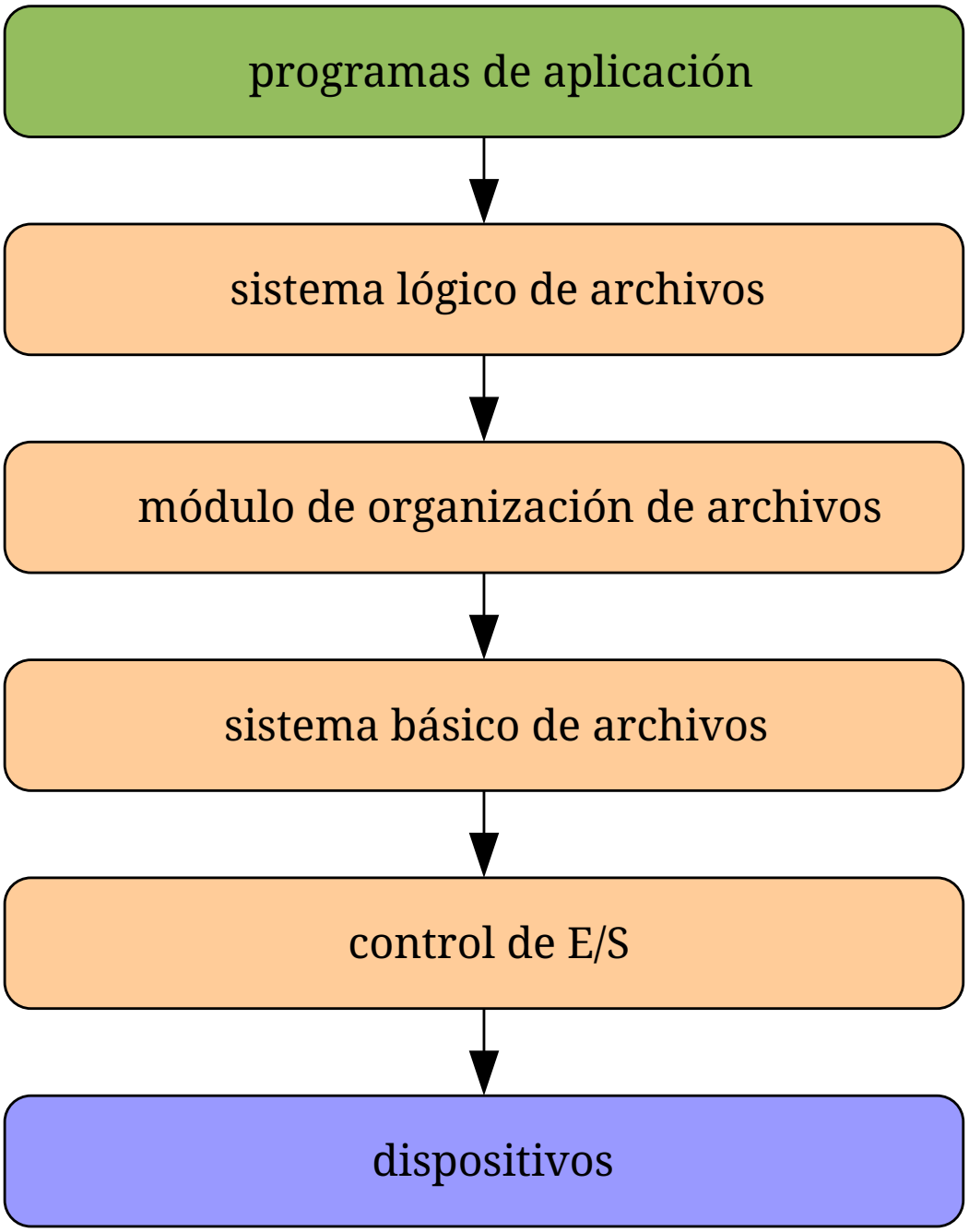


Figura 65. Estructura de un sistema de archivos.

Los sistemas de archivos son un componente complejo, por lo que suelen estar compuesto de varios niveles diferentes. En la Figura 65 se muestra un ejemplo típico de la estructura de un sistema de archivos diseñado en niveles. Cada nivel utiliza las funciones de los niveles inferiores y proporciona nuevas funciones a los niveles superiores.

19.1.1. Control de E/S

En el nivel más bajo, accediendo directamente a los dispositivos de almacenamiento, se encuentra el **control de E/S**.

Contiene los controladores de dispositivo encargados de transferir la información entre la memoria principal y el disco. Estos controladores, que generalmente son compartidos entre los distintos sistemas de archivos, transfieren los datos en unidades de **bloques** —en lugar de transferir un byte cada vez— para mejorar la eficiencia. Cada _*bloque* está formado por uno o más sectores.

Dependiendo de la unidad de disco, los sectores pueden tener tamaños de entre 32 bytes y 4096 bytes. Lo más común es que su tamaño sea de 512 bytes.

19.1.2. Sistema básico de archivos

El **sistema básico de archivos** se encarga de enviar comandos genéricos al controlador de dispositivo apropiado, con el fin de leer y escribir bloques físicos en el disco. Cada bloque físico se identifica mediante su dirección de disco numérica.

Por ejemplo, en dispositivos que usan direccionamiento de tipo cabeza-cilindro-sector (CHS), la dirección de un sector podría ser: unidad 1, cilindro 73, cabeza 2, sector 10. Mientras que en dispositivos que admiten direccionamiento LBA (*Logical Block Addressing*), la dirección de un sector podría ser: unidad 1, sector 4691123.

El **LBA** es un método común para especificar la localización de los sectores de un disco. Usa un esquema de direccionamiento lineal, donde cada sector es identificado con un número entero único. Antes de este método, se usaba el de cabeza-cilindro-sector (CHS), pero tenía la desventaja de hacer públicos los detalles físicos del dispositivo de almacenamiento.

19.1.3. Módulo de organización de archivos

El **módulo de organización de archivos** tiene conocimiento de los archivos y se encarga de traducir las direcciones lógicas de los bloques en los archivos —es decir, posición del bloque dentro del archivo, siendo 0 la dirección del primer bloque— en las direcciones físicas de bloque —por ejemplo, cilindro, cabeza y sector del bloque correspondiente en el dispositivo de almacenamiento— que serán enviadas al **sistema básico de archivos** para que realice las transferencias solicitadas.

Los bloques lógicos de cada archivo son numerados de 0 a *N*, pero los bloques físicos asignados a estos bloques lógicos no tienen por qué coincidir en los números de bloque. Por eso, el **módulo de organización de archivos** debe utilizar la ubicación del contenido del archivo en el disco y la información sobre los bloques físicos asignados, para traducir las direcciones lógicas en direcciones físicas.

Además, el módulo de organización, incluye el gestor de espacio libre, que controla los bloques no asignados y proporciona dichos bloques cuando el **módulo de organización de archivos** lo necesita, ya sea para crear un archivo nuevo o para extender uno existente.

19.1.4. Sistema lógico de archivos

El **sistema lógico de archivos** gestiona los **metadatos**. En los **metadatos** se incluye toda la estructura del sistema de archivos, excepto los propios datos de los archivos.

Entre dichos **metadatos** está la **estructura de directorios** y los **bloques de control de archivo**. Un **bloque de control de archivo** o **FCB** (*File Control Block*) contiene información acerca del archivo, incluyendo su propietario, los permisos y la ubicación del contenido del mismo.

Además, el **sistema lógico de archivos** también es responsable de las tareas de protección y seguridad.

Cada sistema operativo puede soportar uno o más sistemas de archivos para dispositivos de disco. Por ejemplo, en los sistemas UNIX se utiliza el «sistema de archivos UNIX» o UFS (*UNIX File System*), que está basado en el sistema FFS (*Fast File System*) de la Universidad de Berkeley. Microsoft Windows soporta los sistemas de archivo FAT (*File Allocation Table*), FAT32 y NTFS (*NT File System*). En Linux se soportan más de cuarenta sistemas de archivo, entre los que cabe destacar: la familia *extended filesystem* — ext2, ext3 y ext4— XFS y Btrfs.

Además, la mayoría de los sistemas operativos modernos soportan otros sistemas de archivo, como los utilizados en los soportes removibles. Por ejemplo el ISO 9660, utilizado por la mayor parte de los CD-ROM, o el UDF (*Universal Disk Format*), utilizado por los DVD-ROM y Blu-ray.

19.2. Estructuras de metadatos en disco

Para implementar un sistema de archivos se utilizan diversas estructuras de **metadatos** alojadas tanto en el disco como en la memoria. Estas estructuras varían dependiendo del sistema operativo y del sistema de archivos. Sin embargo, a continuación intentaremos describir brevemente las estructuras en disco más comunes.

19.2.1. Bloque de control de arranque

En todo sistema de archivos suele haber un **bloque de control de arranque** —también llamado **bloque de inicio** o **sector de arranque**— que suele ocupar el primer bloque de cada volumen y que contiene la información necesaria para iniciar un sistema operativo a partir de dicho volumen.

Este bloque puede estar vacío, si el volumen no contiene un sistema operativo.

19.2.2. Bloque de control de volumen

El **bloque de control de volumen** contiene todos los detalles acerca del volumen. Por ejemplo, el número máximo de bloques, el tamaño de los bloques, el número de bloques libres y punteros a los mismos; así como un contador de bloques de información **FCB** ocupados y punteros a estos.

A esta estructura se la denomina **superbloque**, en los sistemas de archivos de sistemas UNIX y Linux. Mientras que en NTFS esta información se almacena en la **tabla maestra de archivos** o **MFT** (*Master File Table*).

19.2.3. Bloque de control de archivo

Todo sistema de archivos tiene un **bloque de control de archivo** o **FCB** (*File Control Block*) por archivo, en que se almacenan numerosos detalles sobre cada uno de los archivos Por ejemplo, los permisos, el propietario, el tamaño y la ubicación de los bloques de datos, entre otros.

En términos generales, todos los **FCB** del sistema de archivos se almacenan en una tabla denominada **directorio de dispositivo** o **tabla de contenidos del volumen**.

En los sistemas de archivos de sistemas UNIX y Linux cada FCB se denomina **inodo** y se almacenan a continuación del **superbloque**. En NTFS esta información se almacena en la **MFT**, ya que cada entrada de dicha tabla es un **FCB**.

19.2.4. Estructura de directorios

Finalmente, por lo general los sistemas de archivos tienen una estructura de directorios, para organizar los archivos.

En los sistemas de archivos de sistemas UNIX y Linux, cada directorio es como un archivo especial que almacena los nombres de los archivos que contiene y los índices de los **inodos** de cada uno de ellos. En NTFS es similar, aunque la estructura de directorios completa se almacena en la propia **MFT**.

19.3. Estructuras de metadatos en memoria

La información almacenada en memoria se utiliza tanto para la gestión del sistema de archivos como para mejorar el rendimiento del mismo mediante mecanismos de caché.

Los datos se cargan en el momento de comenzar a utilizar el sistema de archivos —proceso denominado montaje— y se descartan cuando se va a dejar de hacer uso del mismo —es decir, en el desmontaje—.

Las estructuras existentes en la memoria pueden incluir las que a continuación se describen:

- Una **tabla de montaje** en memoria que contiene información acerca de cada volumen montado en el sistema.
- Una caché en memoria de la **estructura de directorios** que almacena la información relativa a los directorios a los que se ha accedido recientemente. Los directorios que actúan como puntos de montaje pueden contener un puntero a la entrada, en la tabla de montaje, del volumen montado en el directorio.
- La **tabla global de archivos abiertos** que contiene una copia del **FCB** de cada archivo abierto en el sistema, además de otras informaciones.
- La **tabla de archivos abiertos** de cada proceso. El **PCB** de cada proceso contiene una tabla donde se listan los archivos abiertos por el proceso.

La **tabla de archivos abiertos** contiene, para cada archivo, un puntero a la entrada correspondiente del mismo archivo en la tabla global de archivos abiertos, pero también guarda otras informaciones adicionales que son particulares de cada proceso. Por ejemplo, si el proceso lo

ha abierto para lectura o escritura o la posición del puntero que indica la siguiente posición a leer o escribir.

19.4. Montaje de sistemas de archivos

Un sistema de archivos debe **montarse** para que sus archivos sean accesibles a los procesos del sistema. El proceso de montaje incluye los siguientes pasos:

1. Al sistema operativo se le debe proporcionar el nombre o identificador del dispositivo y el punto de montaje. El **punto de montaje** es la ubicación dentro de la estructura de directorios —la ruta al directorio concreto— a la que queremos conectar el sistema de archivos. Después de que el proceso de montaje se haya completado, los archivos y directorios del sistema de archivos montado serán accesibles como descendientes del directorio del **punto de montaje**.
2. A continuación el sistema operativo verifica que el dispositivo contiene un sistema de archivos válido. Para ello lee el **bloque de control de volumen** y comprueba que tiene un formato válido.
3. Finalmente el sistema operativo registra en la **tabla de montaje** el tipo de sistema de archivos y el identificador del dispositivo montado. Después, almacena el índice de la entrada correspondiente en la **tabla de montaje** en la copia en memoria del **FCB** del directorio que hace de **punto de montaje**.

   Esto permite que pueda ser recorrida la estructura de directorios de distintos sistemas de archivos, pasando de uno a otro de forma transparente, según sea necesario.

En muchos sistemas operativos modernos, el montaje se ejecuta automáticamente cuando los dispositivos son detectados durante el arranque del sistema o cuando se conectan durante el funcionamiento del mismo —por ejemplo, cuando se inserta un medio en la unidad CD-ROM o se pincha una memoria flash en un puerto USB—. En algunos se permite, además, que el administrador del equipo ejecute operaciones de montaje manuales.

19.5. Archivos

Cada sistema de archivos almacena en disco una tabla donde cada entrada guarda un **bloque de control de archivo** o **FCB** (*File Control Block*) por archivo. Concretamente, en cada **FCB** se almacena diversa información acerca del archivo al que representa.

19.5.1. Atributos de archivos

La colección de atributos asociada a un archivo varía de un sistema operativo a otro, pero típicamente son los siguientes:

- **Nombre**. Nombre simbólico del archivo, que se mantiene en un formato legible por la conveniencia de los usuarios.
- **Identificador**. Identifica de forma unívoca el archivo dentro del sistema de archivos. Generalmente es el índice del **FCB** en la **tabla de contenidos del volumen**, donde se almacenan los **FCB**.
- **Tipo**. Es un atributo necesario en los sistemas que soportan diferentes tipos de archivos.

- **Ubicación**. Es un puntero a un dispositivo y a la ubicación de los bloques con los datos del archivo dentro del mismo.
- **Tamaño**. Indica el tamaño actual de archivo —en bytes, palabras o bloques— y, posiblemente, el tamaño máximo permitido.
- **Protección**. Información de control de acceso que determina quién puede leerlo, escribirlo, ejecutarlo, etc.
- **Fecha, hora e identificación del usuario**. Esta información puede mantenerse para los sucesos de creación, de última modificación y último uso del archivo. Puede resultar útil para la protección, seguridad y monitorización del uso del archivo.

Los atributos de los archivos se almacenan en las estructuras de **metadatos**.

Normalmente el nombre se almacena en la estructura de directorios, de tal manera que una entrada de directorio está compuesta del nombre de un archivo y del identificador de su **FCB**. Dicho identificador permite localizar el **FCB** en la **tabla de contenidos del volumen**, que contiene el resto de los atributos del archivo.

19.5.2. Operaciones con los archivos

Un archivo es un tipo abstracto de datos sobre el que pueden realizarse diversas operaciones. Concretamente el sistema operativo proporciona llamadas al sistema para: crear, abrir, escribir, leer, reposicionar el puntero de lectura/escritura, borrar y truncar o redimensionar archivos.

Generalmente el sistema mantiene un puntero de lectura/escritura que hace referencia a la ubicación dentro del archivo en la que debe tener lugar la siguiente operación. Este puntero se actualiza, avanzando cada vez que se realiza una nueva lectura/escritura.

Para desplazarse aleatoriamente por el archivo, el sistema operativo debe ofrecer una llamada al sistema que permita reposicionar el puntero allí donde interese.

Muchos sistemas también disponen de operaciones para consultar y modificar diversos atributos de un archivo, como la longitud o el propietario del mismo. Además, se suelen incluir llamadas para otras operaciones comunes, como añadir datos al final de un archivo o el renombrado de un archivo existente.

Tabla 14. Funciones de la API para manipular archivos.

	POSIX API	**Windows API**
Crear archivos	open() / openat()	CreateFile()
Abrir archivos	open() / openat()	OpenFile()
Cerrar archivos	close()	CloseHandle()
Borrar archivos	unlink()	DeleteFile()
Leer contenidos	read()	ReadFile()
Escribir contenido	write()	WriteFile()

	POSIX API	Windows API
Reposicionar puntero de lectura/escritura	lseek()	SetFilePointer()
Redimensionar archivos	ftruncate()	SetEndOfFile()
Consultar atributos	stat() / fstat()	GetFileAttributes()
Renombrar archivos	rename()	MoveFile()
Copiar archivos		MoveFile()
Mover archivos	rename() solo en el mismo sistema de archivos	CopyFile()

Estas operaciones primitivas pueden combinarse, a su vez, para realizar otras operaciones más complejas —por ejemplo, crear una copia de un archivo o moverlo a otro lugar de la estructura de directorios—.

19.5.3. Abrir archivos

La mayor parte de las operaciones comentadas implican realizar una búsqueda en el directorio para encontrar la entrada asociada con el archivo cuyo nombre se ha indicado. Para evitar realizar esta búsqueda una y otra vez, muchos sistemas requieren que el proceso haga una llamada al sistema **open**, antes de realizar cualquiera de estas operaciones por primera vez sobre un archivo.

En unos pocos sistemas, los archivos se abren automáticamente cuando un proceso solicita su primera operación sobre los mismos y se cierran cuando el proceso termina. Sin embargo, lo más común es que los procesos tengan que abrir los archivos explícitamente.

En concreto la operación **open**:

1. Busca en el directorio el nombre del archivo, hasta encontrar la entrada asociada y recupera el identificador del mismo.
2. Utiliza el identificador del archivo para recuperar el **FCB** correspondiente.
3. Crea una entrada para el archivo en la **tabla de archivos abiertos** donde se almacena la información del FCB.
4. Retorna al proceso devolviendo un identificador —en forma de puntero o de índice— a la nueva entrada en la tabla de archivos abiertos.

El nombre con el que se designa a esas entradas en la tabla de archivos abiertos varía de unos sistemas operativos a otros. En los sistemas POSIX se utiliza el término **descriptor de archivo** —o *file descriptor*— mientras que en los sistemas Microsoft Windows se prefiere el término **manejador de archivo** —o *file handler*—.

Después de utilizar la llamada al sistema **open**, cuando se desea solicitar una operación sobre un archivo, solo es necesario proporcionar el identificador devuelto, evitando así que haga falta realizar nuevamente la exploración del directorio para buscar el archivo.

En los sistemas operativos donde varios procesos pueden abrir un mismo archivo, se suelen utilizar dos niveles de tablas de archivos abiertos:

1. Una **tabla para cada proceso** —almacenada en el **PCB**— donde se indican todos los archivos que el proceso tiene abiertos.

   En dicha tabla se almacena toda la información referente al uso de cada archivo por parte de un proceso. Por ejemplo, se puede almacenar la posición actual utilizada por las operaciones de lectura y escritura o los derechos de acceso.

2. Una **tabla global para todo el sistema** donde se almacena toda la información independiente de los procesos, como la ubicación del archivo en el disco, las fechas de acceso y el tamaño del archivo.

Cuando un proceso invoca la llamada **open**, se añade una entrada en la **tabla de archivos abiertos del proceso**, que a su vez apunta a la entrada correspondiente dentro de la **tabla global del sistema**. Si el archivo no existe en esta última, también hay que crear una entrada en la tabla global del sistema haciendo uso de la información contenida en disco en el **FCB** correspondiente.

Es muy común, que la **tabla global** almacene un **contador de aperturas** para cada archivo, con el objetivo de indicar cuántos procesos lo mantienen abierto.

Cuando el archivo deja de ser utilizado activamente por el proceso, puede ser cerrado utilizando la llamada al sistema **close**. Entonces el **contador de aperturas** se decrementa, de forma que cuando alcance cero querrá decir que la entrada puede ser eliminada de la **tabla global de archivos abiertos**.

19.5.4. Tipos de archivo

Cuando se diseña un sistema operativo es necesario considerar si debe reconocer y soportar el concepto de **tipo de archivo**. Si el sistema operativo reconoce el **tipo de un archivo** puede operar con el mismo de formas razonables. Por ejemplo, el sistema puede impedir que un usuario intente imprimir los archivos que contienen programas en formato binario, pues el documento impreso sería ininteligible.

En los sistemas operativos más comunes las técnicas utilizadas para implementar los tipos de archivo son las siguientes:

- En MS-DOS y Microsoft Windows el tipo de archivo se incluye como parte del nombre del archivo. Es decir, el nombre se divide en dos partes: un nombre y una extensión, separadas por un punto.

  El sistema puede utilizar la extensión para conocer el tipo de archivo y el tipo de operaciones que se pueden realizar con el mismo.

- En macOS cada archivo tiene un atributo que almacena el tipo —por ejemplo, TEXT para los archivos de texto o APPL para las aplicaciones— y otro que contiene el nombre del programa que lo creó. Cuando el usuario hace clic con el ratón sobre el icono de un archivo, el programa que lo creó se ejecuta automáticamente y este abre el archivo.
- En los sistemas estilo UNIX se utiliza un **número mágico**, almacenado al principio de algunos

archivos, para indicar el tipo del mismo. No todos los archivos tienen números mágicos, por lo que se permite hacer sugerencias en forma de extensiones del nombre del archivo.

Sin embargo, en estos sistemas estas extensiones no son obligatorias ni el sistema depende de ellas. Su objetivo, fundamentalmente, es ayudar a los usuarios a determinar el tipo de contenido de un archivo, por lo que pueden ser utilizadas o ignoradas por cada aplicación concreta, en función de las preferencias de sus desarrolladores.

19.6. Estructura de directorios

Algunos sistemas de archivos pueden almacenar millones de archivos en terabytes de disco. Para gestionar todos esos datos necesitamos organizarlos de alguna manera, lo que generalmente implica el uso de directorios.

Un **directorio** puede considerarse una tabla de símbolos que traduce los nombre de los archivos en los identificadores que permiten recuperar sus correspondientes entradas en la **tabla de contenidos del volumen**, donde se almacenan los **FCB**.

A continuación vamos a estudiar los diversos esquemas para definir la estructura lógica del sistema de directorios.

19.6.1. Directorios de un nivel

En la estructura de directorios de un nivel todos los archivos están contenidos en un único directorio.

Esto presenta algunas limitaciones:

- Cuando el número de usuarios del sistema aumenta se hace más difícil que cada uno escoja nombres diferentes para sus archivos. Esto es necesario, puesto que todos los archivos se encuentran en el mismo directorio.
- Incluso en los sistemas operativos monousuario, puede ser difícil para un usuario mantener organizados sus datos a medida que se incrementa el número de archivos.

Este esquema fue utilizado por la primera versión del sistema operativo MS-DOS.

19.6.2. Directorio de dos niveles

En la estructura de directorios de dos niveles cada usuario tiene su propio **directorio de archivos de usuario** o **UFD** (*User File Directory*) que cuelga del **directorio maestro de archivos** o **MFD** (*Master File Directory*).

Cuando un usuario se conecta al sistema o inicia un trabajo, se explora el **MFD**. Esta es una tabla indexada por el nombre de los usuarios o por los números de cuenta, donde cada una de sus entradas apunta al **UFD** de dicho usuario. Puesto que cada **UFD** incluye solo los archivos del usuario al que pertenece, el sistema operativo puede confinar todas las operaciones que puede realizar un usuario sobre los archivos a su **UFD**.

Aunque esto resuelve el problema de la colisión de nombres entre diferentes usuarios, también

presenta algunas desventajas:

- La estructura descrita aísla a los usuarios, lo cual puede ser un problema cuando estos quieren compartir datos para cooperar en alguna tarea.

  La solución pasa por utilizar **nombres de ruta** para designar a un archivo de forma unívoca. Por ejemplo, si el usuario `usera` quiere acceder a su archivo `test`, simplemente debe referirse a él como `test`. Mientras que si quiere acceder al archivo `test` del usuario `userb`, debe utilizar un **nombre de ruta** como `/userb/test`, donde se indica el nombre del usuario y el nombre del archivo.

  En general, cada sistema operativo utiliza su propia sintaxis para nombrar los archivos contenidos en los directorios de otros usuarios.

- Puede ser difícil para un usuario mantener organizados sus datos a medida que se incrementa el número de archivos personales, incluso aunque tenga un directorio para él solo.

19.6.3. Directorios con estructura de árbol

La estructura de directorio de dos niveles puede generalizarse en la **estructura de directorios en árbol** de altura arbitraria. Esto permite que los usuarios puedan crear sus propios subdirectorios para organizar sus archivos de la forma más conveniente.

Cada sistema de archivos tiene un **directorio raíz** que puede contener tanto archivos como otros directorios. A su vez, cada directorio puede contener un conjunto de archivos y subdirectorios.

Normalmente, cada entrada de directorio incluye un bit donde se indica si dicha entrada apunta a un archivo o a un subdirectorio. Esto se hace así porque, generalmente, los directorios no son más que archivos con un formato interno especial; por lo que el sistema debe saber si la entrada apunta a un directorio para interpretar correctamente los datos del directorio.

Directorio de trabajo actual

Comúnmente, en el **PCB** de cada proceso se guarda cuál es su **directorio de trabajo** actual. De esta forma, cuando se hace referencia a un archivo en una llamada al sistema usando solo su nombre, se le busca en el **directorio de trabajo** del proceso.

Si se necesita un archivo que no se encuentra en el **directorio de trabajo** actual, entonces el usuario debe especificar un **nombre de ruta** desde el **directorio de trabajo**, o primero cambiar con una llamada al sistema el directorio de trabajo del proceso al directorio donde está almacenado el archivo.

Windows API permite indicar el **directorio de trabajo** de un nuevo proceso al crearlo, a través del argumento `lpCurrentDirectory` de CreateProcess(). Si este argumento vale NULL, el proceso hereda el **directorio de trabajo** del padre. En todo caso, cualquier proceso puede cambiar su **directorio de trabajo** actual llamando a SetCurrentDirectory().

En los sistemas POSIX, el proceso hijo creado con fork() hereda automáticamente el **directorio de trabajo** actual del proceso padre. Por lo que si necesitamos cambiar su **directorio de trabajo** —por ejemplo, antes de llamar a exec() para cambiar el **directorio de trabajo** para el nuevo programa—

el proceso puede usar chdir().

> Gracias a la herencia del **directorio de trabajo** es como los **intérpretes de comandos** indican a los comandos que ejecutan en qué directorio deben ejecutarse.
>
> Por ejemplo, en los sistemas POSIX el intérprete de comandos **Bash** se inicia usando el directorio personal del usuario como **directorio de trabajo**. El usuario puede cambiar el **directorio de trabajo** del proceso de la **Bash** usando el comando interno `cd`. Cuando se pide a la **Bash** que ejecute cualquier otro comando, esta crea un nuevo proceso hijo donde ejecutar el programa de dicho comando. Ese proceso hereda el **directorio de trabajo** actual de **Bash**.

Nombre de ruta

Los **nombres de ruta** es la forma en la que se indica la ubicación de un archivo o directorio en el árbol de directorios.

Los **nombres de ruta** pueden ser de dos tipos:

- Un **nombre de ruta absoluto** comienza en la raíz y va indicando los directorios que componen la ruta de forma descendente hasta llegar al archivo especificado.

  Por ejemplo, `/usr/share/doc`, `C:\Program Files\WindowsApps` o `\Windows\System32`

- Un **nombre de ruta relativo** define una ruta a partir del **directorio de trabajo actual**.

  Por ejemplo, `Imágenes/Enero/000001.jpg`, `Downloads\horario.zip` o `C:Desktop\Proyectos`.

Con una **estructura de directorios en árbol**, unos usuarios pueden acceder a los archivos de otros. Para eso, solo es necesario que se utilicen **nombres de ruta** para designar los archivos del otro usuario, o que se cambie el **directorio de trabajo actual**.

Este tipo de estructura de directorios es la utilizada por MS-DOS y por las distintas versiones de Microsoft Windows.

19.6.4. Directorios en grafo acíclico

La **estructura de directorio en grafo acíclico** es una generalización natural del esquema con **estructura en árbol**.

A diferencia de este último, la **estructura en grafo acíclico** permite que los mismos archivos y subdirectorios existan simultáneamente en distintos lugares de la estructura de directorios. Eso significa que para acceder a un archivo o directorio pueden existir diversos **nombres de ruta**.

Esto, por ejemplo, permite que los usuarios puedan compartir archivos de tal forma que los mismos archivos y directorios estén disponibles directamente desde el directorio personal de los diferentes usuarios.

Enlaces

Los archivos y subdirectorios compartidos pueden implementarse de diversas formas:

- Se puede crear una entrada de directorio especial denominada **enlace**. Un **enlace** es, generalmente, un archivo que contiene la ruta relativa o absoluta de otro archivo o subdirectorio. En los sistemas POSIX a estos se los conoce como **enlaces simbólicos**.
- También se puede duplicar toda la información de la entrada de directorio del archivo compartido en todos los directorios que también contienen dicho archivo.

  Así, mientras que los **enlaces simbólicos** son claramente diferentes de la entrada original de directorio, las entradas de directorio duplicadas hacen que la entrada original y la copia sean indistinguibles. En los sistemas POSIX, a este tipo de entradas duplicadas se las conoce como **enlaces duros**.

Como **enlaces simbólicos** almacenan una ruta, pueden apuntar a archivos o directorios en otros sistemas de archivos. Mientras que los **enlaces duros** solo pueden apuntar a archivos en el mismo sistema de archivos.

En los sistemas POSIX los **enlaces duros** se crean llamando a link() y los **enlaces simbólicos** con symlink().

En Microsoft Windows se soportan ambos tipos de enlaces desde la primera versión —Microsoft Windows NT 3.1— pero únicamente en el sistema de archivos NTFS y solo por compatibilidad con las aplicaciones POSIX. Windows API no tuvo funciones para crear **enlaces** hasta mucho después. Por eso su uso en Microsoft Windows no es tan común.

En Windows API los **enlaces duros** se crean con CreateHardLink() desde Windows 2000. Mientras que los **enlaces simbólicos** se crean con CreateSymbolicLink() desde Windows Vista.

Inconvenientes

Una estructura en grafo acíclico es más flexible que una estructura en árbol, pero no por eso está exenta de inconvenientes:

- Si estamos intentando recorrer el sistema de archivos completo —por ejemplo, para buscar un archivo o para copiarlos en un dispositivo para hacer copias de seguridad— debemos evitar acceder más de una vez a los archivos y subdirectorios enlazados. No olvidemos que en los sistemas con estructura en grafo acíclico, cada archivo puede tener múltiples nombres de ruta absoluta.

  Esto es más sencillo de resolver en el caso de los **enlaces simbólicos**, puesto que podemos evitar recorrerlos al ser claramente distinguibles de los archivos normales.

- Los diseñadores deben enfrentarse a la cuestión de cuándo liberar el espacio asignado a un archivo enlazado. Si lo hacemos cuando un usuario lo borra podríamos dejar **enlaces** que referencian a archivos que no existen.

Sobre esta última cuestión:

- El caso más sencillo de resolver es el de los **enlaces simbólicos**, ya que pueden ser borrados sin

que el archivo original se vea afectado, puesto que lo que se elimina es el **enlace** y no el archivo original.

- Si lo que se pretende borrar es la entrada de un archivo original que es apuntado desde un **enlace simbólico**, tampoco hay problema en hacerlo y liberar el espacio asignado al mismo, dejando que el enlace apunte a un archivo que no existe. Cuando se produzca un intento de acceder a los archivos a través del **enlace**, el sistema determinará que el archivo referenciado fue borrado y tratará el acceso al enlace de forma similar a cualquier otro acceso ilegal a un archivo que no existe.

Ciertamente, podríamos plantearnos la posibilidad de buscar todos los **enlaces** al archivo borrado y eliminarlos. Pero, a menos que el FCB de cada archivo guarde las rutas a los **enlaces** que le señalan, esta búsqueda podría ser muy costosa.

- Otra opción es almacenar en la entrada del archivo original un contador con el número de referencias al archivo. Cada vez que se elimina una referencia se decrementa el contador. Cuando el contador sea 0, sabremos que ha llegado el momento de liberar el espacio asignado.

  En los sistemas UNIX se utiliza esta técnica para saber cuándo liberar el contenido de archivos con **enlaces duros**.

Por último, no debemos olvidar que la estructura de directorios en grafo se conserva acíclica si se prohíbe que haya múltiples referencias a un mismo directorio. Ese es el motivo por el que en muchos sistemas POSIX no se permite, por defecto, que los **enlaces duros** hagan referencia a directorios. Sin embargo si se pueden utilizar **enlaces simbólicos** para este fin, puesto que al ser distinguibles del directorio original podemos evitar los ciclos, si mientras se explora se ignoran dichos enlaces.

19.6.5. Directorios en forma de grafo general

Uno de los principales problemas de la **estructura de directorios en grafo acíclico** es garantizar que no exista ningún ciclo. Esto es interesante, puesto que mientras sea así los algoritmos diseñados para recorrer el grafo y para determinar cuándo no existen más referencias a un archivo, son relativamente simples.

No olvidemos que:

- Es importante evitar encontrar cualquier archivo dos o más veces, tanto por razones de corrección como de rendimiento.
- En una **estructura de directorios en forma de grafo general** que use contadores de referencia para borrar archivos cuando no hay más referencias, puede que dicho contador no sea 0, aunque no haya más referencias al archivo.

Esto significa que generalmente se necesita algún mecanismo de recolección de basura para determinar con seguridad cuándo se ha borrado la última referencia. La recolección de basura implica recorrer todo el sistema de archivos y marcar todos aquellos elementos que sean accesibles. Después, en una segunda pasada, se elimina todo lo que no esté marcado. Por tanto, es evidente que la recolección de basura para un sistema de archivos basado en disco consume mucho tiempo, por lo que se utiliza en muy pocas ocasiones.

Es mucho más sencillo trabajar con **estructuras de directorio en grafo acíclico**. Para evitar que en un grafo aparezca un ciclo al añadir un nuevo **enlace**, se pueden utilizar diversos algoritmos. Sin embargo, puesto que también suelen ser muy costosos, lo más simple es ignorar todos los **enlaces simbólicos** en los casos en los que se recorre el árbol de directorios para realizar una tarea en la que es importante no entrar en un bucle —por ejemplo, al hacer una búsqueda—.

En el caso de los **enlaces duros** —donde se duplica entradas de directorio que no se pueden distinguir de la del archivo original y, por tanto, no se pueden ignorar— lo más sencillo es que el sistema operativo no permite crear múltiples referencias a un mismo directorio.

19.7. Compartición de archivos

Como ya hemos comentado, el que los usuarios puedan compartir archivos es algo muy deseable, pues permite que estos puedan colaborar en la realización de una tarea determinada. Sin embargo, al añadir esta característica, hay que tener en cuenta algunos aspectos que deben ser resueltos en el diseño del sistema operativo.

19.7.1. Múltiples usuarios y protección

Cuando un sistema operativo admite múltiples usuarios y utiliza una estructura de directorio que permite que estos compartan archivos, cobra gran importancia la protección de los datos. En este sentido, el sistema operativo debe adoptar un papel de mediador en lo que respecta a la compartición de los archivos.

Para implementar la compartición y los mecanismos de protección, el sistema debe soportar más atributos para cada archivo y directorio que los que necesita en un sistema monousuario. Aunque a lo largo de la historia se han adoptado diversos enfoques, la mayoría han evolucionado hasta utilizar los conceptos de **propietario** —o **usuario**— y **grupo** de un archivo:

- El propietario de un archivo es el usuario que puede cambiar los atributos y conceder el acceso. Se trata del usuario que dispone del mayor grado de control sobre el archivo.
- El grupo es un conjunto de usuarios que pueden compartir el acceso al archivo. El propietario del archivo es quien define qué operaciones pueden ser ejecutadas por los miembros del grupo.

Los identificadores del propietario y el grupo de un archivo se almacenan junto con los otros atributos en el FCB.

Cuando un usuario solicita realizar una operación sobre un archivo, se compara el identificador del usuario con el atributo del propietario para determinar si el solicitante es el propietario. Exactamente de la misma manera se puede proceder con los identificadores de grupo. El resultado de la comparación indica que permisos son aplicables. A continuación, el sistema aplica dichos permisos a la operación solicitada y la autoriza o deniega según sea el caso.

Existen diversas implementaciones del esquema utilizado para determinar los permisos aplicables a un usuario que pretende operar sobre un archivo concreto.

Lista de control de acceso

El esquema más general consiste en asociar a cada archivo o directorio una **lista de control de**

acceso o **ACL** (*Access-control list*) que especifique los nombres de usuario o grupos y los tipos de acceso para cada uno.

Cuando un usuario solicita acceder a un archivo concreto, el sistema operativo comprueba la **ACL** asociada a dicho archivo. Si el usuario, o alguno de sus grupos, está incluido en la lista para el tipo de acceso solicitado, se permite el acceso.

Esta técnica presenta diversas ventajas e inconvenientes:

- Se trata de la técnica más general, permitiendo la implementación de políticas de acceso muy complejas.
- Construir la lista puede ser una tarea tediosa. Por ejemplo, si queremos que varios usuarios puedan leer unos archivos determinados, es necesario enumerar todos los usuarios que disponen de ese acceso en las **ACL** de dichos archivos.
- El **FCB**, que hasta el momento tenía un tamaño fijo, ahora tendrá que ser de tamaño variable para almacenar la **ACL**, lo que requiere mecanismos más complejos de gestión del espacio.

La familia de sistemas operativos Microsoft Windows utiliza este tipo de **ACL**. Al crear un archivo nuevo, se puede indicar la **ACL** deseada a través del argumento `lpSecurityAttributes` de CreateFile().

También se puede consultar la **ACL** de cualquier objeto con permisos —incluidos los archivos— usando GetSecurityInfo() o GetNamedSecurityInfo() y modificarla usando SetSecurityInfo() o SetNamedSecurityInfo().

Las **ACL** y los **descriptores de seguridad** de Windows API son un tema complejo. Antes de manipularlos, es conveniente consultar «Security Descriptors» en Microsoft Docs.

Lista de control de acceso condensada

Para solucionar algunos de los problemas de las **ACL** muchos sistemas utilizan **listas de control de acceso condensadas**.

Para condensar la longitud de la lista de control de acceso, generalmente los sistemas clasifican a los usuarios en tres grupos: **propietario**, **grupo** y **otros**. Así solo es necesario un campo para cada clase de usuario, siendo cada campo una colección de bits, donde cada uno permite o deniega el tipo de acceso asociado al mismo.

Por ejemplo, en los sistemas POSIX se definen 3 campos —**propietario**, **grupo** y **otros**) de 3 bits cada uno: `rwx`, donde `r` controla el acceso de lectura, `w` controla el acceso de escritura y `x` controla la ejecución.

Las ACL condensadas son más sencillas de construir. Al mismo tiempo, por tener una longitud fija, es mucho más simple gestionar el espacio para el **FCB** donde se almacenan.

Permisos de archivo en sistemas POSIX

Los sistemas POSIX usan, por defecto, **listas de control de acceso condensadas**.

Al crear un archivo nuevo, se pueden indicar los permisos de forma numérica a través del argumento `mode` de open():

```
int fd = open(
    "foo.txt",
    O_CREAT | O_RDWR,
    0666 ① ②
);
```

① Indica los permisos del archivo en caso de crearlo. Se ignora si el archivo ya existe.

② Los bits a 1 del número especificado indican los permisos autorizados. Es muy común hacerlo en base octal, como en el ejemplo.

Otra llamada al sistema que permite indicar permisos es mkdir(), que se utiliza para crear directorios

En ambos casos, los permisos finalmente usados vienen determinados por la ***umask*** del proceso. Esta es una propiedad numérica de los procesos —heredada de padres a hijos— que indica qué permisos del argumento `mode` de open() y mkdir() se desactivan al crear un archivo o directorio. Por ejemplo, si el argumento `mode` es 0666 y ***umask*** es 0022, los permisos efectivos al crear el archivo serán 0644.

Un proceso puede cambiar su ***umask*** mediante la función umask() y los usuarios de **Bash**, las de su *shell* usando el comando del mismo nombre.

Los procesos pueden consultar fácilmente los permisos que tienen sobre un archivo usando access(). Y pueden cambiar su propietario y el grupo llamando a la función chown(). Si necesitan leer los permisos, el propietario y el resto de **metadatos** del archivo, pueden usar stat() o fstat().

Combinar ambos tipos de listas de control de acceso

Muchos sistemas POSIX también soportan un borrador de especificación llamado POSIX ACL, que describe una interfaz para usar las **ACL** más genéricas. Sistemas operativos como Linux, macOS o FreeBSD implementan ambos tipos de ACL.

Para más información sobre las listas de control de acceso POSIX, véase «acl(5) — Linux Manual»

Combinar ambos tipos de **ACL** ofrece lo mejor de ambos mundos, pero no es una solución que esté exenta de dificultades:

- Uno de los problemas es que los usuarios deben poder determinar cuando están activados los

permisos **ACL** más generales. En Linux, por ejemplo, se utiliza el símbolo + al listar los permisos de la ACL condensada para indicar dicha circunstancia. Esos permisos pueden ser gestionados utilizando los comandos setfacl y getfacl.

- Otra dificultad es la relativa a la asignación de precedencias cuando ambas **ACL** entran en conflicto. En general, se suele asignar a la **ACL** más prioridad que a la **ACL condensada**, pues la primera tiene una granularidad más fina y no se crea de forma predeterminada.

19.7.2. Semántica de coherencia

La **semántica de coherencia** especifica cuándo las modificaciones que un proceso realiza en los archivos serán observables por los otros procesos. Por tanto, es importante tenerla en cuenta cuando esperamos que varios procesos utilicen los mismos archivos al mismo tiempo.

A continuación vamos a comentar algunos ejemplos de tipos **semántica de coherencia**.

Semántica POSIX

Los sistemas de archivos de los sistemas operativos POSIX utilizan la siguiente **semántica de coherencia**:

- Las escrituras en un archivo abierto por parte de un proceso son visibles inmediatamente para los procesos que tengan abierto el mismo archivo.
- Existe un modo de compartición que permite a los procesos compartir el puntero de ubicación actual dentro del archivo. Así, el incremento de ese puntero por parte de un proceso afecta a todos los procesos que estén compartiendo el archivo.

En la semántica POSIX, cada archivo está asociado con una única imagen física con el contenido del archivo, a la que se accede en forma de recurso en **exclusión mutua**. Por ejemplo, un proceso que haga **read** sobre un archivo podría quedar en espera si al mismo tiempo otro proceso está ejecutando un **write**, hasta que este último termine.

La competición por acceder a esta imagen única provoca retrasos en los procesos debido a estos bloqueos.

Semántica de sesión

El sistema de archivos Andrew (AFS) es un sistema de archivos en red —o sistema de archivos distribuido— es decir, sirve para compartir archivos en una red de ordenadores y usarlos como si estuvieran almacenados localmente.

AFS es altamente escalable, existiendo despliegues con más de 25000 clientes. Para conseguirlo, cada equipo mantiene una copia local de los archivos abiertos. Las operaciones de lectura y escritura se realizan en esa copia. Cuando se cierra el archivo modificado, los cambios son enviados al servidor de archivos, para actualizar el archivo original.

Aunque es posible implementar la **semántica POSIX** —como hacen otros sistemas de archivos en red— esta no escala adecuadamente, porque implica mantener sincronizadas las copias locales de cada archivo. Es decir, asegurar que los nodos no pueden modificar sus copias locales simultáneamente y que los cambios se propaguen adecuadamente entre las copias, antes de

responder a cualquier operación de lectura solicitada por un proceso. Por eso el sistema de archivo AFS usa una semántica de coherencia diferente, denominada **semántica de sesión**.

Suponiendo que una **sesión de archivo** es el conjunto de operaciones entre las llamadas **open** y **close**, la **semántica de sesión** consisten en que:

- Las escrituras en un archivo abierto por parte de un proceso no son visibles inmediatamente para los otros usuarios que hayan abierto ese mismo archivo.
- Una vez que se cierra un archivo, los cambios realizados en él son visibles únicamente en las sesiones que comiencen posteriormente. Las sesiones ya abiertas sobre el archivo no reflejarán dichos cambios.

Esto significa que un archivo puede permanecer temporalmente asociado a distintas imágenes físicas de su contenido al mismo tiempo. Esto ocurre en el sistema de archivos AFS porque un mismo archivo tiene distintas copias locales temporales en los nodos que lo tienen abierto. Así se permite que múltiples nodos realicen accesos concurrentes, tanto de lectura como de escritura, en sus propias imágenes del archivo, evitando los retrasos.

A cambio hay que tener cuidado con el hecho de que un proceso puede estar leyendo datos obsoletos, sin saberlo. Si un proceso necesita acceder a los datos que escribe otro proceso, ambos deben sincronizarse explícitamente abriendo y cerrando el archivo.

Semántica de archivos compartidos inmutables

En esta semántica, cuando un archivo es declarado como compartido por su creador, ya no puede ser modificado.

Estos archivos inmutables cumplen dos propiedades clave: su nombre no puede reutilizarse y su contenido no puede ser modificado. Así podemos estar seguros de que el contenido de un archivo inmutable es fijo. Para escribir algo en uno de estos archivos, es necesario crear una copia con un nuevo nombre y hacer en ella los cambios.

Para optimizar la implementación de esta semántica se suele usar una técnica similar al **copy-on-write**. Con esta técnica, cuando se va a modificar un archivo inmutable, se genera una copia que tiene la misma asignación de bloques que el archivo original. Cada vez que se va a modificar la información de un bloque, se crea una copia de ese bloque, se aplican los cambios y se sustituye el identificador del bloque anterior por el del nuevo bloque en la asignación de bloques del archivo en el **FCB**. Así, las copias de archivos inmutables se hacen más rápido y se ahorra espacio, dado que de cada archivo solo se guardan los bloques modificados.

Además, cuando los sistemas operativos usan esta semántica, suelen tener una forma de crear automáticamente los nombres de las nuevas versiones de un archivo —por ejemplo, añadiendo un entero al nombre e incrementándolo en cada versión—.

La implementación de esta semántica en un sistema de archivos distribuido es muy simple, puesto que es muy sencillo hacer copias locales de los archivos. Al ser inmutables, no hace falta disponer de un mecanismo para sincronizar los cambios entre los nodos.

19.7.3. Bloqueos de archivo

Algunos sistemas operativos proporcionan funciones para bloquear un archivo abierto —o partes del mismo—. Esto permite que un proceso impida que otros procesos puedan acceder al archivo bloqueado.

Los bloqueos de archivo resultan útiles para encadenar varias operaciones de E/S sobre un archivo, teniendo la seguridad de que otros procesos no podrán hacer modificaciones en el mismo mientras tanto.

Los sistemas operativos pueden proporcionar diferentes tipos de bloqueos de archivo:

- Un **bloqueo compartido** es un tipo de bloqueo que puede ser adquirido —es decir, bloquear el archivo— al mismo tiempo por varios procesos.
- Un **bloqueo exclusivo** solo puede ser adquirido por un proceso cada vez. Si otro proceso intenta adquirir un **bloqueo exclusivo** sobre un archivo ya bloqueado, por cualquiera de los dos tipos de bloqueos, se suspende a la espera de que el bloqueo anterior sea liberado.

Algunos sistemas operativos solo proporcionan el **bloqueo exclusivo**. Sin embargo, en los que implementan ambos tipos de bloqueo, lo normal es que los procesos que pretenden acceder a un archivo compartido para solo lectura utilicen el **bloqueo compartido**, mientras que los que acceden para modificar el contenido utilicen el **bloqueo exclusivo**. Así, varios procesos pueden leer el archivo al mismo tiempo, pero si un proceso accede para escribir, ningún otro podrá acceder ni para leer ni para escribir.

Bloqueo obligatorio o sugerido

Además, los sistemas operativos pueden proporcionar dos tipos de mecanismos de bloqueo de archivos:

- Si el **bloqueo es obligatorio**, después de que un proceso adquiera un bloqueo exclusivo, el sistema operativo impedirá a todos los demás procesos que hagan cualquier operación sobre el archivo bloqueado.

  Esto ocurrirá incluso si los otros procesos no han sido programados para intentar adquirir el bloqueo. Por tanto, el sistema operativo es el encargado de garantizar que los bloqueos se cumplen, haciendo las comprobaciones pertinentes en las llamadas al sistema.

- Si el **bloqueo es sugerido**, el sistema operativo solo impedirá que accedan al archivo bloqueado aquellos procesos programados para adquirir el bloqueo explícitamente.

  Para eso los programas deben invocar ciertas llamadas al sistema para adquirir el bloqueo y liberarlo, Pero el sistema operativo no impedirá el acceso al archivo a un proceso que lo abre y lo lee o escribe sin más.

Los sistemas operativos Microsoft Windows implementan un mecanismo de **bloqueo obligatorio**. En Windows API se puede indicar el modo de bloqueo al abrir el archivo con CreateFile() o se puede usar LockFile() para adquirir un bloqueo sobre una parte del contenido

En los sistemas UNIX y estilo UNIX, como regla general, no se bloquea un archivo al abrirlo. Existen

diferentes mecanismos de bloqueo, algunos de los cuales pueden ser **bloqueos obligatorios**, pero por defecto son **bloqueos sugeridos**.

Bloqueo de archivos en sistemas POSIX

En los sistemas POSIX el mecanismo más usado es fcntl(), que permite bloquear porciones del contenido de un archivo, tanto con **bloqueo exclusivo** como con **bloqueo compartido**. El bloqueo se asocia al **inodo** y al **PID** del proceso, por lo que:

- Si el archivo tiene varios nombres —por el uso de **enlaces duros**— el bloqueo sobre el archivo tiene efecto sin importar el nombre usado para abrirlo.
- Diferentes descriptores sobre el mismo archivo obtenidos llamando a open() varias veces en el mismo proceso, comparten los bloqueos adquiridos. Así que, usando este tipo de bloqueos es posible sincronizar distintos procesos pero no distintos hilos, ya que todos los hilos de un mismo proceso comparten la adquisición del bloqueo.

El estándar POSIX también soporta lockf(), que es como una versión simplificada de fcntl(). Aunque el estándar deja sin especificar cómo deben interactuar ambas llamadas, lo cierto es que es común que lockf() se implemente usando fcntl(). Esta función también crea bloqueos de porciones del contenido, asociados al **inodo** del archivo y el **PID**, pero solo soporta crear y liberar **bloqueos exclusivos**.

Finalmente, muchos sistemas UNIX y estilo UNIX soportan flock(). Esta función fue introducida en 4.2BSD, pero nunca fue incorporada al estándar POSIX. Admite tanto **bloqueos exclusivos** como **bloqueos compartidos** pero, a diferencia de las llamadas anteriores, el bloqueo afecta siempre al archivo completo y se asocia al descriptor de archivo.

Esto último significa que:

- Diferentes descriptores sobre el mismo archivo obtenidos llamando a open() varias veces, no comparten los bloqueos adquiridos. Así que pueden usarse para sincronizar incluso hilos de un mismo proceso.
- Diferentes descriptores de archivo obtenidos llamando a dup2() o a fork(), comparten los bloqueos adquiridos. Por lo que no pueden usarse para sincronizar hilos de un mismo proceso, pero permite que un proceso padre transfiera la adquisición del bloqueo a sus hijos.

Aparte de estos mecanismos, cada sistema operativo puede implementar algunas funcionalidades adicionales, no incluidas en el estándar POSIX. Por ejemplo, la llamada fcntl() de Linux permite un tipo de bloqueo con las ventajas de los bloqueos originales de fcntl() pero asociados a descriptores de archivo. Esto permite usarlos para sincronizar hilos de un mismo proceso y para que un proceso pueda transferir la adquisición del bloqueo a sus hijos, como ocurre con los bloqueos BSD de flock().

En filelock-server.c y filelock-client.cpp se puede ver un ejemplo similar al de capítulos anteriores, pero usando en esta ocasión bloqueo de archivos. El servidor filelock-server.c es un programa que muestra periódicamente la hora del sistema. Mientras que el cliente filelock-client.cpp, simplemente envía una señal `SIGTERM` al servidor cuando queremos que

termine. Para que el cliente conozca el PID del servidor —de entre todos los procesos en ejecución en el sistema— el servidor escribe su PID en un archivo en una ubicación conocida por ambos.

Como el cliente lee el archivo con una única operación read() y el servidor lo escribe con una única operación write(), no hace falta el uso de bloqueo de archivos para sincronizarlos. Gracias a la semántica de coherencia POSIX, el cliente no puede leer el archivo en medio de la escritura. Es decir, o ve el PID completo escrito por el servidor o no ve ninguno.

Pero si puede darse el caso de que se ejecuten varios servidores al mismo tiempo. Cada uno debe comprobar si archivo existe y, si es así, leer el PID que contiene y comprobar si hay un proceso con ese mismo PID. Si el archivo no existe o no encuentra un proceso con ese PID, debe entender que es el nuevo servidor y escribir su PID en el archivo, para que lo encuentre el cliente. En caso contrario, debe terminar.

Para evitar que varios servidores den todos esos pasos al mismo tiempo, acaben creyendo que son los únicos y sobrescriban el archivo varias veces, el acceso al archivo debe hacerse en **exclusión mutua**. Así lo van bloqueando de uno en uno y mientras no hace sus comprobaciones los demás esperan. Por eso filelock-server.c utiliza lockf() para bloquear el archivo, sincronizando el acceso de los servidores.

19.8. Coherencia

Como hemos comentado anteriormente, parte de los **metadatos** se almacena en la memoria principal para acelerar el acceso. Dicha información generalmente está más actualizada que la correspondiente en el disco, puesto que la información almacenada en la memoria no tiene por qué ser escrita inmediatamente después de una actualización.

Entonces ¿qué ocurriría si fallase el sistema? Pues que el contenido de la caché y de los búferes se perdería, y con ellos los cambios realizados en los directorios y archivos abiertos. Esto puede dejar el sistema de archivos en un estado incoherente, pues el estado real de algunos archivos no sería el que se describe en la estructura de **metadatos**.

19.8.1. Comprobación de coherencia

El **comprobador de coherencia** comprueba la estructura de **metadatos** y tratar de corregir todas las incoherencias que detecte.

Los algoritmos de asignación y de gestión del espacio de almacenamiento dictan los tipos de problemas que el comprobador puede tratar de detectar y también el grado de éxito que puede tener en esa tarea.

Por ejemplo, la pérdida de un **FCB**, cuando es este el que almacena la lista de bloques que contienen los datos del archivo, es desastrosa porque no hay forma de saber qué datos le pertenecen de entre todos los que hay en el disco. Por esta razón, UNIX almacena en caché las entradas de directorio para acelerar las lecturas, pero todas las escrituras de datos que provoquen algún cambio en la asignación de espacio o en algún otro tipo de metadato, se realizan síncronamente —antes de continuar ejecutando el proceso desde la llamada al sistema—.

Es decir, si se hace una escritura de datos que extiende el tamaño de un archivo; el cambio del **FCB** correspondiente, con el nuevo tamaño de archivo y la lista actualizada de las direcciones de los bloques que contienen —o van a contener— los datos del archivo, se escribe en disco antes de terminar la llamada al sistema y devolver el control al proceso que la invocó.

Sin embargo, no ocurre lo mismo con los datos que el proceso quería escribir en el archivo. El sistema operativo suele copiarlos a búferes internos en la memoria para escribirlos en disco más adelante, evitando interrumpir el proceso durante demasiado tiempo. Esto significa que en caso de fallo del sistema, el sistema de archivos puede estar en estado consistente pero haberse perdido los nuevos datos del archivo, porque no dio tiempo de escribirlos en el disco.

En Microsoft Windows el programa **comprobador de coherencia** se llama CHKDSK. Mientras que en sistemas POSIX se llama fsck.

19.8.2. Soft Updates

Para mejorar la eficiencia del sistema de archivos, sin comprometer la coherencia en caso de fallo, los distintos sabores de los sistemas UNIX BSD utilizan una técnica denominada **soft updates** en su implementación del sistema de archivos UFS.

Cuando se monta un sistema de archivos con la opción **soft updates**, el sistema operativo desactiva la escritura síncrona de los **metadatos**, que comentamos anteriormente, permitiendo que estos sean escritos cuando los algoritmos de gestión de la caché lo consideren necesario, pero se impone cierto orden en el que dichas operaciones de escritura deben ser realizadas.

Por ejemplo, cuando se van a escribir en el disco las modificaciones debidas a la creación de un nuevo archivo, el sistema se asegura de que primero se escribe el nuevo **FCB** —un *inodo*, en los sistemas UNIX BSD— y posteriormente se escribe el directorio con la nueva entrada de archivo con el identificador a dicho **FCB**.

Es sencillo darse cuenta de que haciéndolo al revés, si el sistema fallase antes de crear el **FCB**, acabaríamos con una entrada de directorio que apuntaría a un **FCB** inválido. Mientras que de esta manera el sistema de archivos permanecerá consistente aunque el sistema falle entre ambas operaciones.

19.8.3. Sistemas de archivos basados en registro

Otra solución al problema de la coherencia del sistema de archivos, consiste en aplicar técnicas de recuperación basadas en **registro** —o **journaling**— durante las actualizaciones de los **metadatos** del sistema de archivos.

Fundamentalmente, en los **sistemas de archivos basados en registro** cada conjunto de operaciones sobre los **metadatos**, necesario para realizar una tarea específica sobre el sistema de archivos, es una **transacción**. Por ejemplo, crear un archivo nuevo es una **transacción**, formada por el conjunto de operaciones sobre los **metadatos** necesarias para crearlo. También lo es añadir datos al final de un archivo existente, aunque en este caso la **transacción** está formada por las operaciones sobre los **metadatos** necesarias para extender el tamaño del archivo.

Las operaciones sobre los **metadatos** de una **transacción** se escriben secuencialmente en un

registro de operaciones que se usa de la siguiente manera:

1. Durante la llamada al sistema, la lista de operaciones sobre los **metadatos** necesarias para completar una **transacción** se escribe secuencial y síncronamente en el **registro**, antes de terminar la llamada al sistema. Cuando la lista de operaciones pendientes termina de ser escrita en el registro, se considera que las operaciones han sido **confirmadas** y la llamada al sistema puede volver al proceso, permitiendo que continúe con su ejecución.
2. Mientras tanto, el sistema operativo va ejecutando las operaciones indicadas en el **registro** sobre las estructuras reales del sistema de archivos. A medida que se realizan los cambios, se actualiza el **registro** para indicar las operaciones completadas.
3. Cuando todas las operaciones de una **transacción** se han ejecutado con éxito, dicha **transacción** se considera completada y se elimina del **registro**.

En el supuesto de que el sistema falle:

- Se comprueba el **registro** durante el montaje del sistema de archivos, antes de que pueda ser utilizado de nuevo.
- Todas las **transacciones confirmadas** que contenga el **registro** estarán a medias, por lo que se terminan de aplicar las **operaciones pendientes** antes de finalizar el proceso de montaje.
- Todos los cambios correspondientes a las **transacciones no confirmadas** que hubieran sido aplicados al sistema de archivos, son deshechos para preservar la coherencia. Las **transacciones no confirmadas** son aquellas no terminaron de ser escritas en el **registro** antes del fallo y, por tanto, cuya lista de operaciones no está completa.

Esta técnica es muy común en muchos sistemas operativos modernos. Por ejemplo, es utilizada en sistemas de archivos tales como: ext3, ext4, NTFS, XFS, HFS+, etc.

Operaciones sobre datos

Es importante recordar que lo se guarda en el **registro** son las operaciones sobre los **metadatos**, porque se trata de preservar la coherencia del sistema de archivos en caso de fallo del sistema. Sin embargo, esto quiere decir que se pueden perder lo que no son operaciones sobre los **metadatos**, como por ejemplo, las operaciones que modifican los datos de los archivos, con lo que lo estos pueden acabar almacenando datos corruptos o antiguos.

Para evitar en parte este problema, algunos sistemas de archivos fuerzan la escritura de los datos en el disco antes de **confirmar** la transacción en el **registro**.

Por ejemplo, al añadir datos al final de un archivo existente, se guardan en el registro las operaciones sobre los **metadatos** necesarias para extender el archivo sobre bloques nuevos. Pero esta **transacción** no se marca **confirmada**, hasta que los nuevos datos no se escriban en los nuevos bloques del archivo. Así, si el sistema falla antes de que se escriban los datos, como la operación no está **confirmada**, la extensión del archivo se deshace, quedando en su estado anterior, en lugar de extenderse sobre bloques que no se han terminado de escribir. Si el sistema falla después, como la operación está **confirmada**, el archivo termina de extenderse sobre los bloques actualizados con los datos.

El sistema de archivos ext4 tiene por defecto este comportamiento. Sin embargo, si la fiabilidad de

los datos es muy importante, se le puede indicar que considere las operaciones sobre los datos dentro de la **transacción**. De esta forma, las incluye en la lista de operaciones guardadas en el **registro**, las aplica posteriormente junto al resto de operaciones sobre los **metadatos** de la **transacción** y, en caso de fallo del sistema, puede decidir si deshacerlas o terminar de aplicarlas, para asegurar la consistencia de los datos en los archivos.

Rendimiento en el acceso a disco

Un efecto colateral de la utilización de un **registro** es la mejora del rendimiento en el acceso al sistema de archivo.

La razón de esta mejora, es que las costosas escrituras síncronas —es decir, antes de devolver el control al proceso— de los **metadatos** en lugares aleatorios del volumen se transforman en escrituras síncronas secuenciales —que son mucho más eficientes— en el **registro**. Mientras que todas las operaciones indicadas en el **registro** se aplican asíncronamente mediante escrituras aleatorias en las estructuras apropiadas, por lo que pueden ser ordenadas a conveniencia para maximizar el rendimiento.

Recordemos que en el registro puede haber operaciones de distintos procesos que afecten a regiones próximas del disco. Por lo que el resultado global es una significativa ganancia en la velocidad de las operaciones relativas a los **metadatos**, como por ejemplo la creación y borrado de archivos.

Journaling en XFS

El sistema de archivos XFS modifica ligeramente esta técnica, sustituyendo las escrituras síncronas necesarias para actualizar el **registro** por escrituras asíncronas. Es decir, el control se devuelve al proceso antes de terminar de escribir las operaciones en el **registro** y confirmar la transacción, por lo que el proceso puede seguir ejecutándose antes que en otros sistemas de archivos.

El uso de escrituras asíncronas en el **registro** es peligroso. Cualquier caída del sistema podría provocar la corrupción del **registro**, porque las escrituras asíncronas pueden ocurrir en cualquier orden. Esto, aparentemente, elimina cualquier ventaja de utilizar un **registro** de operaciones. Sin embargo, XFS impone cierto orden en las operaciones de escritura sobre el registro —de forma similar a como se hace con los **soft updates**— de tal manera que la coherencia del registro está asegurada.

El utilizar escrituras asíncronas en el **registro** —aunque sea preservando cierto orden— ofrece alguna mejora en el rendimiento, porque el **registro** deja de ser un cuello de botella para las operaciones que modifica los **metadatos** del sistema de archivos.

Como hemos comentado, en otros **sistemas de archivos basados en registro**, el registro es un recurso al que se accede en **exclusión mutua**. Es decir, las operaciones de una **transacción** se deben escribir antes de devolver el control al proceso y de permitir que otro proceso a la espera escriba las operaciones de su **transacción**. Estas esperas en el acceso al **registro** son las que XFS evita, permitiendo su modificación de manera asíncrona.

19.8.4. Sistemas de archivos basados en copia durante la escritura

Las técnicas anteriores son necesarias para preservar la coherencia porque la modificación de los metadatos se hace sobrescribiendo los datos que ya existen. Es decir, cuando se crea un nuevo archivo, el sistema busca un FCB libre, sobrescribe el bloque del dispositivo donde lo encuentra para almacenar el nuevo FCB, busca una entrada libre en el directorio y, nuevamente, sobrescribe el bloque del disco donde se almacena el directorio para incorporar la nueva entrada. Si algunos de estos cambios tienen lugar, pero otros no, el disco puede quedar en estado inconsistente.

*Los **sistemas de archivos basados en copia durante la escritura*** —o *copy-on-write— evitan cambiar los metadatos sobrescribiendo en el sitio*. En su lugar buscan un hueco libre, hacen en él una copia del bloque completo con los cambios y después modifican los metadatos del sistema de archivos que sirven para localizar el bloque modificado en su nueva ubicación. Estos cambios, a su vez, tampoco se hacen sobrescribiendo, sino que disparan la creación de copias modificadas de los bloques afectados, lo que nuevamente va seguido de cambios en los metadatos que ayudan a localizarlos. El proceso se repite hasta que se alcanza el *bloque de control de volumen* y se cambia, momento en el que toda la secuencia de cambios se consolida.

Los sistemas de archivos basados en copy-on-write suelen hacer uso intensivo de estructuras de datos basadas en árbol porque es muy sencillo mover un nodo de bloque, con un efecto mínimo en el resto de la estructura. Por ejemplo, al crear un archivo:

1. *Se busca un FCB libre, se lee el bloque que lo contiene en la memoria principal, se modifica y se escribe en un bloque libre*. El sistema de archivos debe tener alguna estructura de datos que permita encontrar el bloque que contiene un FCB a partir de su identificador. Por lo general, esta estructura es algún tipo de árbol. Así que se modifica el nodo del árbol que señala al bloque con el nuevo FCB para que conozca la nueva ubicación. Este cambio implica crear una copia del bloque de dicho nodo con el cambio, lo que a su vez significa modificar el nodo que señala a este. Y así sucesivamente hasta llegar a la raíz del árbol de FCB.
2. *Se busca una entrada libre en el directorio que va a contener al archivo y se modifica para añadir el nombre del archivo, el identificador de su FCB y otras propiedades*. Nuevamente, este cambio significa crear una copia, con los cambios descritos, del bloque que contiene la entrada y modificar el FCB del directorio para que contenga la nueva ubicación del bloque con el contenido del directorio. Como antes, este cambio en el FCB dispara copias y modificaciones por todo el árbol de FCB, hasta la raíz
3. Una vez la raíz del árbol ha sido copiada a una nueva ubicación con los cambios, se actualiza su nueva posición en el *bloque de control de volumen.*

Si el sistema falla antes de la modificación del *bloque de control de volumen,* durante el montaje del sistema de archivos no quedará ni rastro de ninguno de los cambios porque dicho bloque aún hace referencia a la antigua raíz del árbol de FCB y, a partir de ellas, a todos los nodos, bloques y FCB originales. Obviamente los sistemas que implementan este tipo de sistemas de archivo usan la memoria principal como caché con el objeto de combinar varias modificaciones sobre un mismo bloque antes de proceder a su escritura en disco, evitando desencadenar múltiples veces los cambios posteriores.

Los sistemas de archivos ZFS y Btrfs son los principales ejemplos de sistemas de archivos basados en *copy-on-write*. Esta solución no solo les permite tener las mismas propiedades que el uso de

registro en cuanto a la preservación de la coherencia —con la ventaja de evitar dos escrituras en disco, una en el registro y otra para el cambio propiamente dicho— sino que además facilita que puedan ofrecer características adicionales, como integrar en el propio sistema de archivos la gestión de volúmenes dinámicos (véase el Apartado 18.3.3) o la creación de copias instantáneas del volumen.

Chapter 20. Implementación de sistemas de archivos

Tiempo de lectura: 31 minutos

Como ya hemos comentado, un sistema de archivos suele estar compuesto de varios niveles diferentes. En la Figura 65 se muestra un ejemplo de la estructura de un sistema de archivos diseñado en niveles. Cada nivel utiliza las funciones de los niveles inferiores y proporciona nuevas funciones a los niveles superiores. El papel de cada uno de estos niveles fue descrito en el Apartado 19.1. Mientras que las estructuras de metadatos utilizadas, tanto en la memoria como en disco, fueron tratadas brevemente en el Apartado 19.2 y en el Apartado 19.3.

A continuación, vamos a profundizar aún más en las estructuras y operaciones utilizadas para implementar los sistemas de archivos

20.1. Implementación de directorios

Cada directorio suele contener una estructura de datos que relaciona el nombre de cada archivo que contiene con el identificador de su **FCB**. Dicho identificador permite localizar el FCB en la **tabla de contenidos del volumen**, que contiene el resto de los atributos del archivo.

En esta sección vamos a estudiar las formas más comunes de implementar la estructura de datos de un directorio.

20.1.1. Lista lineal

El método más simple para implementar un directorio consiste en utilizar una lista lineal o vector de nombres de archivos e identificadores del **FCB**.

Las acciones a realizar, para implementar cada una de las posibles operaciones sobre el directorio, serían:

- **Crear un archivo**. Primero se explora el directorio para estar seguros de que no haya ningún archivo con el mismo nombre. Después se añade una nueva entrada al final del directorio.
- **Borrar un archivo**. Primero se explora la lista en busca del archivo especificado y, una vez localizada, se libera la entrada correspondiente. Para reutilizar la entrada del directorio tenemos diversas alternativas:
  - Se puede marcar la entrada como no utilizada. Para eso se puede emplear un nombre especial o utilizar algún campo adicional —a parte de nombre de archivo e identificador del **FCB**— que se haya añadido a la entrada con ese propósito.
  - Insertar un puntero a la entrada en una lista de entradas libres, que se guarda dentro del mismo directorio.
  - Copiar la última entrada del directorio en la ubicación que ha quedado libre y reducir la longitud del directorio.

La principal desventaja de un directorio implementado como una lista lineal de entradas es que para localizar un archivo es necesario realizar una búsqueda lineal, lo cual puede resultar muy costoso en directorios con un número muy grande de archivos. Utilizando una lista ordenada se puede reducir el tiempo medio de búsqueda, pero complica los procesos de creación y borrado, pues puede que sea necesario mover cantidades importantes de información para mantener la lista ordenada.

También se puede utilizar una lista enlazada, tanto para reducir el tiempo necesario para borrar un archivo como para facilitar la tarea de mantener ordenada la lista.

Los sistemas de archivos FAT y FAT32 implementan los directorios utilizando una lista lineal, donde en cada entrada se almacena el nombre del archivo y el **FCB** del mismo. Al borrar un archivo, la entrada correspondiente se marca poniendo 0xE5 en el primer caracter del nombre del archivo.

Los sistemas de archivos ext2 y UFS también utilizan una lista lineal no ordenada, donde solo se almacena el nombre del archivo o subdirectorio y el identificador del **inodo** —el FCB, esos sistemas de archivo— correspondiente. En caso de borrar un archivo, el identificador del **inodo** se pone a 0.

20.1.2. Tabla de dispersión

En los directorios implementados con una tabla de dispersión también se almacenan las entradas de directorio en una lista lineal, pero al mismo tiempo se utiliza una tabla de dispersión para reducir enormemente el tiempo de búsqueda en el directorio. Para obtener la ubicación de dicho archivo dentro de la lista lineal, se usa un índice calculado con cierta función de dispersión a partir del nombre del archivo.

El único inconveniente es que debemos tratar la posible aparición de colisiones, que son aquellas situaciones en las que dos nombres de archivo dan lugar, al aplicarles la función de dispersión, la misma ubicación en la tabla. Esto se puede resolver utilizando una lista enlazada en cada entrada de la lista —cada entrada en la lista señalaría la ubicación de la siguiente entrada de la lista que tiene el mismo valor para la función de dispersión— a cambio de que las búsquedas sean un poco más lentas. En cualquier caso, este método será normalmente más rápido que una búsqueda lineal por todo el directorio.

20.1.3. Árbol B

Para mantener el directorio ordenado, algunos sistemas de archivos modernos utilizan estructuras de datos en árbol más sofisticadas, como por ejemplo árboles B.

Un caso concreto es el sistema de archivos NTFS, utilizado por Microsoft Windows. NTFS utiliza una estructura de datos denominada árbol B+ para almacenar el índice de los nombres de archivo contenidos en un directorio.

En la entrada en la **MFT** (*Master File Table*) de cada directorio se almacena un atributo denominado **raíz del índice**. Si el directorio es de pequeño tamaño, la **raíz del índice** contiene todas las entradas de archivos del directorio, pero para un directorio de gran tamaño, la **raíz del índice** solo puede almacenar unas pocas entradas de archivos del directorio. En ese caso la **raíz del índice** contiene el nivel superior del árbol B+. Es decir, cada una de esas entradas de archivos en la **raíz del índice** incluye también un puntero al bloque del disco que contiene un nodo del árbol con las

entradas con nombres alfabéticamente anteriores a ese. Si en dicho nodo tampoco caben todas las entradas, solo podrá contener algunas de ellas, por lo que cada una tendrá, a su vez, un puntero a un nuevo nodo del árbol; y así sucesivamente.

Las ventajas de los árboles B+ son:

- Eliminan el coste de reordenar las entradas del directorio.
- La longitud desde la raíz del árbol hasta un nodo hoja es la misma para todas los caminos por el árbol, por lo que el tiempo de búsqueda tiene una cota superior.

Implementación de directorios en XFS

El sistema de archivos XFS también utiliza un árbol B+, pero en este caso la implementación es un poco más compleja:

- Un directorio de pequeño tamaño almacena sus entradas como una lista lineal no ordenada dentro de su mismo **inodo** o FCB.
- Cuando el directorio no cabe en el **inodo** se le asigna un bloque propio, donde el directorio es implementado con una tabla de dispersión, tal y como hemos visto anteriormente.
- Cuando el tamaño del directorio excede el tamaño del bloque, la tabla de dispersión se extrae y se almacena en un bloque diferente. La lista lineal también se extrae, pero no tiene que ser almacenada en un único bloque, sino que puede estar repartida por distintos bloques a lo largo del disco.
- Finalmente, cuando la tabla de dispersión excede el tamaño de un bloque, dicha tabla se convierte en un árbol B+.

20.2. Métodos de asignación

El siguiente problema es cómo asignar el espacio disponible en el disco a los archivos almacenados, de forma que el espacio sea utilizado de forma eficiente y que se pueda acceder a los archivos de la forma más rápida posible.

Como la unidad mínima de asignación de espacio a un archivo es el bloque, la fragmentación interna suele ser un problema común a todos los métodos que veremos a continuación.

20.2.1. Asignación contigua

La **asignación contigua** requiere que cada archivo ocupe un conjunto contiguo de bloques en el disco. Esto es muy eficiente, puesto que el acceso a todos los datos de un archivo requiere un movimiento mínimo del cabezal del disco.

El problema de la **asignación contigua** puede verse como un caso concreto del problema de la asignación dinámica del almacenamiento (véase el Apartado 15.5). Es decir, que en un momento dado tendremos una petición de tamaño N que deberemos satisfacer con una lista de huecos libres de tamaño variable. Como estudiamos anteriormente, las estrategias más comunes son las del

primer ajuste y el **mejor ajuste**.

Fragmentación externa

La asignación contigua sufre el problema de la **fragmentación externa**. La solución sería utilizar alguna forma de **compactación** para unir los huecos libres, pero esto puede llevar mucho tiempo en discos duros de gran tamaño y en algunos sistemas esta tarea tiene que realizarse con el dispositivo desmontado. Por eso es conveniente evitar utilizar técnicas de compactación en los sistemas en producción.

Afortunadamente, la mayor parte de los sistemas operativos modernos que necesitan mecanismos de **desfragmentación** pueden realizar esta tarea sin detener el sistema, aunque la pérdida de rendimiento puede ser significativa.

Estimación del tamaño del archivo

En la asignación contigua es necesario determinar cuánto espacio necesita un archivo antes de asignárselo. El problema es que eso no siempre es posible.

Por ejemplo, si vamos a copiar un archivo, es indudable que conocemos de antemano cuánto espacio necesita la copia. Pero ¿qué ocurre cuando vamos a crear uno nuevo? Entonces al crear un archivo es necesario que el usuario haga una estimación del espacio que va a necesitar y se la indique al sistema.

¿Y si la estimación no es correcta o posteriormente queremos añadir nuevos datos al archivo? Entonces, lo más probable es que el espacio situado a ambos lados del archivo ya esté ocupado, si hemos utilizado la estrategia del **mejor ajuste**. Para resolver este problema existen dos estrategias:

- La primera, es terminar el programa de usuario, emitiendo un error. Entonces, el usuario deberá volver a crear el archivo indicando más espacio y volver a ejecutar el programa. Puesto que las ejecuciones repetidas pueden ser muy costosas, lo más común es que el usuario acabe sobreestimando el espacio, lo que dará como resultado un desperdicio considerable de espacio.
- La segunda, es buscar un hueco libre de mayor tamaño y copiar el contenido del archivo al nuevo espacio. Esto puede hacerse siempre que exista suficiente espacio, aunque puede consumir bastante tiempo.

Para minimizar estos problemas, se puede implementar un esquema de asignación contigua modificado, donde se asigna inicialmente un bloque contiguo de espacio al archivo y, posteriormente, si dicho espacio resulta no ser lo suficientemente grande, se añade otra área de espacio contiguo, denominado **extensión**.

La ubicación de las **extensiones** de un archivo se registran en el FCB, guardando la dirección del primer bloque de cada extensión que compone el archivo y el número de bloques que ocupa cada una.

Los sistemas de archivo XFS y ext4 utilizan **extensiones** para optimizar su funcionamiento. El motivo es que cuantos más bloques contiguos sean asignados a un archivo, menos reposicionamientos del cabezal del disco son necesarios para leerlos. En ext4 el espacio se asigna a los archivos en **extensiones** de hasta 128 MiB, compuestas por bloques, generalmente, de 4 KiB.

20.2.2. Asignación enlazada

En la **asignación enlazada** cada archivo es una lista enlazada de bloques de disco, pudiendo estos bloques estar dispersos por todo el disco:

- Cada entrada de directorio contiene un puntero al primer bloque. En ocasiones, la entrada también incluye un puntero al último, para facilitar añadir nuevos datos al final del archivo.
- Cada bloque contiene un puntero al bloque siguiente. Por ejemplo, si cada bloque tiene 512 bytes de tamaño y un puntero requiere 4 bytes, los bloques de disco tendrán un tamaño efectivo de 508 bytes.

Este mecanismo resuelve todos los problemas de la asignación contigua y además:

- **No hay fragmentación externa**, puesto que pueden utilizarse cualquier bloque libre para satisfacer una solicitud de espacio.
- **No es necesario declarar el espacio del archivo** en el momento de crearlo, pues el archivo podrá siempre podrá crecer mientras haya bloques libres.

Sin embargo, la asignación enlazada también tiene sus desventajas.

Eficiencia en accesos aleatorios

La **asignación enlazada** solo resulta eficaz para acceder a los archivos de acceso secuencial.

Si necesitamos ir directamente al bloque i-ésimo de un archivo, tendremos que comenzar desde el principio e ir leyendo cada bloque para obtener el puntero que nos indica el siguiente bloque. Es muy posible que esas lecturas deban ir precedidas de un reposicionamiento de los cabezales del disco.

Una solución parcial a esto puede ser guardar en la memoria una caché de las direcciones de los bloques de los archivos accedidos recientemente.

Eficiencia en el uso del espacio de almacenamiento

En la **asignación enlazada** se pierde cierta cantidad de espacio con los punteros.

Si, por ejemplo, un puntero ocupa 4 bytes y un bloque tienen un tamaño de 512 bytes, el 0.758% del espacio en disco será utilizado para los punteros, en lugar de para almacenar información útil.

La solución para este problema consiste en asignar los bloques en grupos —denominados **clústeres** —. Así, el primer bloque de cada **clúster** solo tendría que almacenar un puntero al siguiente **clúster**, lo que reduciría la cantidad de espacio desperdiciada en los punteros y mejoraría la eficiencia al reducir el número de reposicionamientos del cabezal del disco. Sin embargo, esto también incrementa el grado de **fragmentación interna** pues se pierde más espacio cuando un **clúster** está parcialmente lleno.

Fiabilidad

Teniendo en cuenta que los archivos están enlazados mediante punteros, parte de un archivo se puede corromper fácilmente con que solo uno de esos punteros se pierda o resulte dañado.

Asignación enlazada en FAT y FAT32

Los sistemas de archivos FAT y FAT32 utilizan una variante del mecanismo de asignación enlazada en la que se emplea una **tabla de asignación de archivo** o **FAT** (*File-Allocation Table*).

La FAT se almacena en una sección al principio del volumen. Contiene una entrada por cada **clúster** del disco y en cada una guarda el número del siguiente **clúster** del archivo. Es decir, lo que hace la FAT es agrupar en un solo lugar los punteros de la **asignación enlazada**.

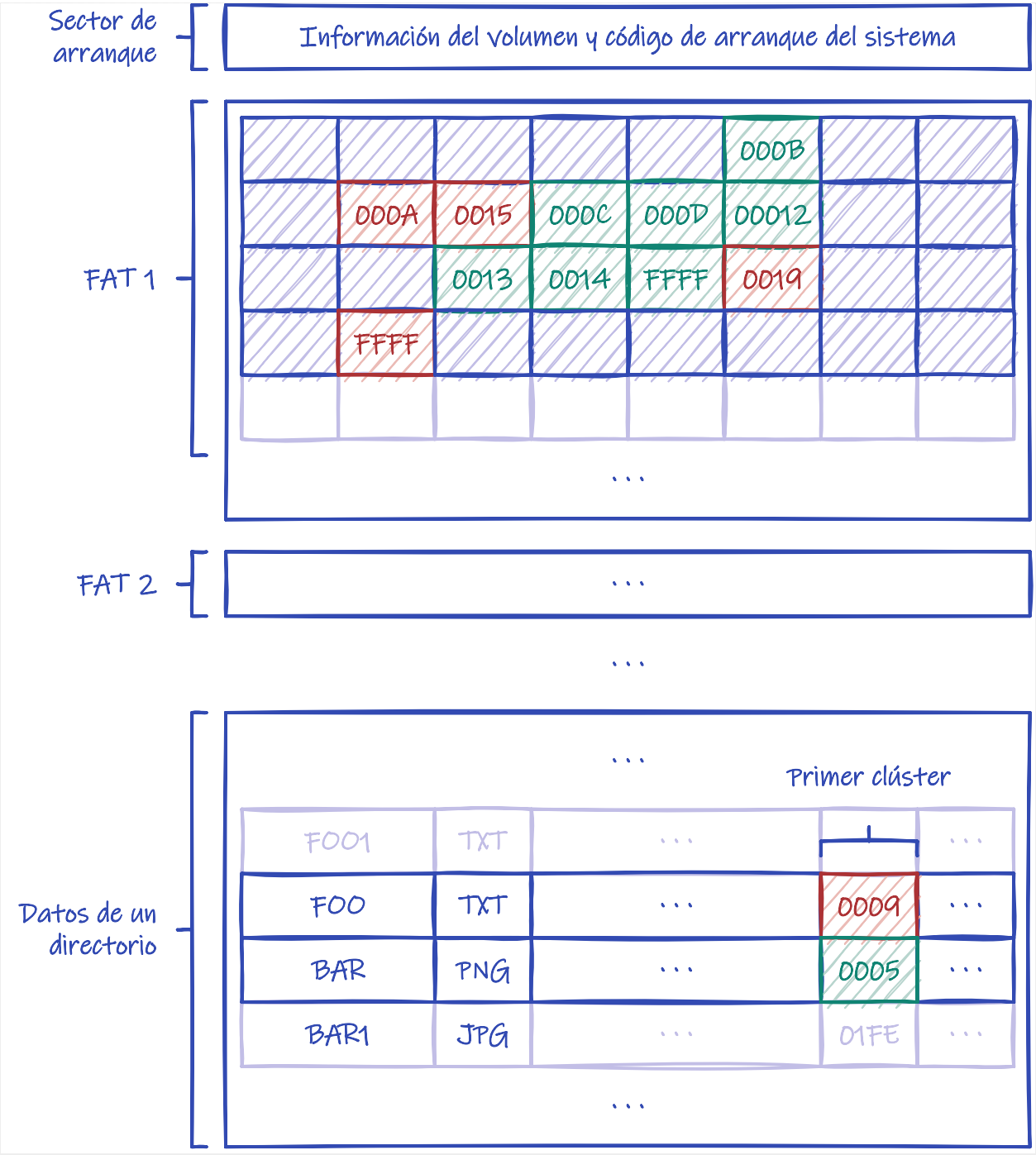


Figura 66. Asignación enlazada en el sistema de archivos FAT.

Eso significa que la FAT es una estructura crítica. Si se corrompe, puede provocar la pérdida de todo el sistema de archivos. Por eso, realmente se almacenan dos copias de la FAT al principio del volumen.

El indexar con clústeres —o grupos de bloques— sirve para ubicar cerca bloques contiguos, pero sobre todo es una decisión de diseño para permitir que los sistemas de archivos FAT puedan gestionar volúmenes más grandes. Por ejemplo, la FAT de MS-DOS 3.0 y posteriores usaba 16 bits para numerar los clústeres —por eso se llamaba FAT16—. Si FAT16 trabaja directamente con bloques y cada bloque tuviera 512 bytes, no podría gestionar volúmenes de más de 512×2^{16} bytes, es decir, 32 MiB. Sin embargo, trabajando con 64 bloques por clúster, se puede usar para gestionar volúmenes de hasta 2 GiB.

Aunque algunas versiones de Microsoft Windows NT llegaron a admitir 128 e, incluso, 512

bloques por clúster, con clústeres tan grandes la **fragmentación interna** es un problema.

Por eso —entre otras cosas— Microsoft introdujo FAT32, que utiliza 32 bits para numerar los clústeres. Eso implica poder gestionar más cantidad de clústeres, lo que permite gestionar volúmenes más grandes con clústeres más pequeños y, por tanto, con menor **fragmentación interna**. Por ejemplo, FAT32 podía gestionar volúmenes de hasta 2 TiB con 64 bloques por clúster.

Como se puede ver en la Figura 66, en los sistemas de archivos FAT cada entrada de directorio contiene, aparte del nombre del archivo y otros atributos, el número del primer clúster con datos del archivo. La entrada de la FAT indexada según ese número contiene el número del siguiente clúster del archivo. Iterando de esa manera, se puede conocer los números de todos los clústeres de un archivo.

El último bloque del archivo se indica con un valor especial en su entrada en la FAT. Mientras que los bloques no utilizados se indican con un valor igual a 0 en su entrada en la FAT.

El uso de la FAT puede provocar un número importante de reposicionamientos del cabezal de disco, debido a que siempre es necesario volver al principio del volumen para leer dicha tabla. Por eso, es muy habitual que el sistema operativo intente mantener una copia de la FAT en la memoria, a modo de caché.

Una de las ventajas de este esquema es que mejora el tiempo de los accesos aleatorios a los archivos —respecto a la asignación enlazada convencional— porque se puede conocer la ubicación de cualquier bloque a partir de la información en la FAT, sin tener que leer todos los bloques del archivo uno a uno.

20.2.3. Asignación indexada

El mecanismo de **asignación indexada** agrupa todos los punteros de la asignación enlazada en una única ubicación: el **bloque de índices**. Así se resuelve la falta de eficiencia de la asignación enlazada convencional —en ausencia de FAT— cuando se realizan accesos aleatorios.

En la **asignación indexada**:

- Cada archivo tiene su propio **bloque de índices**, que es un bloque del disco con una tabla con los números de los bloques del disco que contienen los datos del archivo.
- La entrada i-ésima del bloque de índice contiene la dirección del bloque i-ésimo del archivo.
- Cada entrada de directorio contiene la dirección del bloque de índices del archivo correspondiente.

Este mecanismo soporta el acceso aleatorio eficiente, además de no sufrir el problema de la fragmentación externa. Sin embargo, también tiene sus desventajas:

- Se pierde más espacio en los punteros que con el mecanismo de asignación enlazada, pues siempre hay que reservar un **bloque de índices** completo para cada archivo. Mientras que con la asignación enlazada, solo se pierde el espacio de los punteros que realmente es necesario utilizar.
- Al diseñar el sistema de archivos debemos determinar el tamaño del **bloque de índices**.

  Por el inconveniente anterior y puesto que cada archivo debe tener un bloque de índices, ese bloque debe ser lo más pequeño posible para no perder espacio. Pero si es demasiado pequeño, no podrá almacenar suficientes punteros para un archivo de gran tamaño.

Entre los mecanismos que pueden utilizarse para resolver este último problema están los siguientes:

- En el **esquema enlazado**, se enlazan los bloques de índices.

  Por ejemplo, se puede utilizar el último puntero del bloque de índices para apuntar al siguiente bloque de índices. Si dicho puntero tiene valor 0, entonces estamos en el último bloque de índices del archivo.

- En el **índice multinivel**, los punteros del bloque de índices no señalan a los bloques del archivo, sino a un conjunto de bloques de índices de segundo nivel y estos, a su vez señalan, a los bloques del archivo. Esta técnica puede ampliarse utilizando un tercer o cuarto nivel, dependiendo del tamaño máximo de archivo que se desee.
- En el **esquema combinado** las primeras entradas del bloque de índices apuntan directamente a los primeros bloques del archivo. Mientras que las siguientes entradas contiene punteros indirectos, que apunta a un conjunto de bloques de índices de segundo nivel. Después podría haber entradas que contienen punteros doblemente indirectos y luego entradas con punteros triplemente indirectos.

Para mejorar el rendimiento de los mecanismos de asignación indexados, es muy común que el sistema operativo intente mantener los bloques de índices en la memoria.

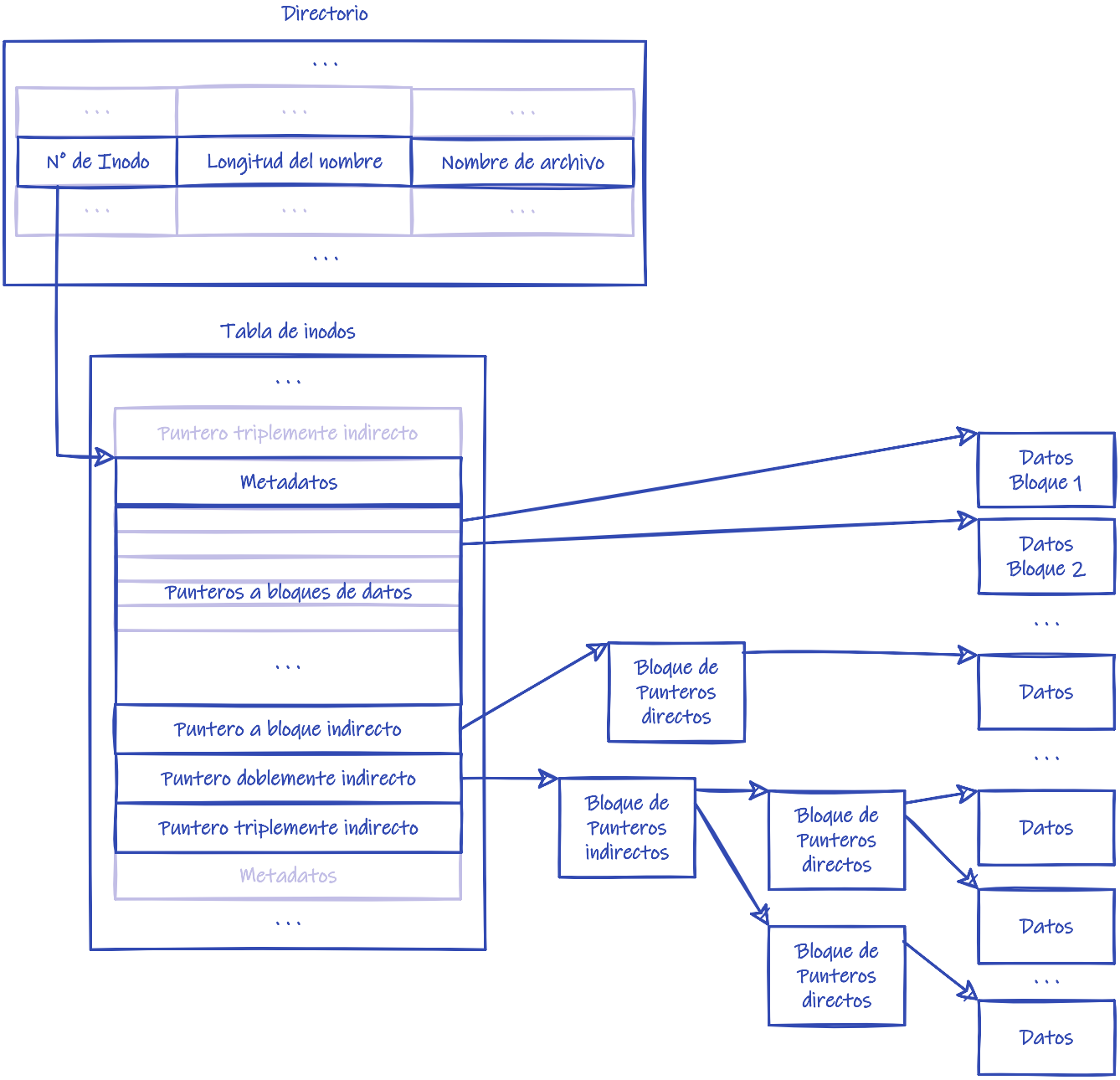


Figura 67. Asignación indexada combinada en el sistema de archivos ext2.

Los sistemas de archivos ext2 y ext3 utilizan el mecanismo de **asignación indexada** con **esquema combinado**. Concretamente el mecanismo en ext2 se implementa de la siguiente manera:

- El disco se divide en múltiples grupos de bloques de disco.
- En cada grupo, se utilizan los primeros bloques para almacenar una tabla de **inodos**. Estos **inodos** son los **FCB** de los archivos almacenados en el grupo.

  El resto de los bloques se utilizan para almacenar los datos de los archivos representados por los **inodos** del grupo.

- Dentro de cada **inodo** —entre otra información típica en un **FCB**— se almacenan los punteros a los bloques del archivo, en lugar de utilizar un bloque de índices aparte.
- Como se puede ver en la Figura 67, los primeros 12 punteros en el **inodo** son

directos, seguidos de un puntero indirecto, un puntero doblemente indirecto y uno triplemente indirecto. Esto permite almacenar hasta 2^{64} bytes de información en cada archivo.

20.3. Gestión del espacio libre

Puesto que el espacio en disco es limitado, necesitamos poder reutilizar el espacio de los archivos borrados. Para controlar el espacio libre en el disco, el sistema de archivos mantiene una **lista de espacio libre** que contiene todos los bloques de disco disponibles. Para crear un archivo, se explora la **lista de espacio libre** hasta obtener la cantidad de espacio requerida y se asigna ese espacio al nuevo archivo.

A continuación estudiaremos las diferentes maneras de implementar esa lista.

20.3.1. Vector de bits

La lista de espacio libre puede ser implementada como un **vector de bits** o **mapa de bits**, donde cada bloque es representado por un bit. Si el bloque está libre, el bit está a 1, mientras que si el bloque está asignado, el bit está a 0.

Este enfoque es relativamente sencillo y eficiente, puesto que muchos procesadores disponen de instrucciones para manipular bits, que pueden utilizarse para obtener rápidamente el primer bloque libre. Por ejemplo, la familia de procesadores x86, a partir del Intel 80386, tiene instrucciones que devuelven la posición del primer bit a 1 en el valor de un registro.

Sin embargo, leer el vector de bits, modificarlo y actualizarlo en disco en cada ocasión, es tremendamente ineficiente. Por tanto, usar vectores de bits es ineficiente, a menos que se mantenga el vector completo en la memoria principal, escribiéndose ocasionalmente en el disco.

Esto último puede ser imposible para los discos de gran tamaño, en función de la cantidad de memoria principal. Por ejemplo, un disco de 40 GiB con bloques de 512 bytes necesitará un mapa de bits de más de 10 MiB, lo que no es un gran requisito para un sistema moderno, pero sí lo era hace dos décadas.

El sistema de archivo NTFS y la familia *extended filesystem* —es decir, ext, ext2, ext3 y {ext4— utilizan mapas de bits, tanto para gestionar los bloques de datos libres como las entradas disponibles en la tabla de **inodos**.

20.3.2. Lista enlazada

Otra técnica consiste en enlazar todos los bloques de disco libres. Para eso se puede guardar un puntero al primer bloque libre en una ubicación especial del disco y que ese bloque contenga un puntero al siguiente bloque libre del disco. El segundo bloque contenga un puntero al tercer bloque libre, y así sucesivamente.

El inconveniente es que recorrer la lista no resulta eficiente, pues tenemos que leer cada bloque para conocer la dirección del siguiente bloque libre en disco. Sin embargo, debemos tener en cuenta que no es frecuente tener que recorrer la lista de espacio libre completa porque, por lo

general, basta con encontrar el primer bloque libre para asignar el espacio.

Los sistemas de archivos FAT incorporan el control de bloques libres dentro de la **tabla de asignación de archivos** guardando un 0 en las entradas de los **clústeres** libres, por lo que no se necesita ningún método adicional.

20.3.3. Agrupamiento

Una modificación de la técnica basada en la lista enlazada consiste en almacenar las direcciones de *N* bloques libres en el primer bloque libre. Los primeros *N — 1* de esos bloques estarían realmente libres, pero el último de ellos apuntaría a otro bloque con *N* bloques libres. Así podrían localizarse rápidamente las direcciones de un gran número de bloques libres, lo cual mejora la eficiencia respecto a la técnica de lista enlazada.

20.3.4. Recuento

Generalmente los bloques son asignados o liberados en bloques contiguos, especialmente si el espacio es asignado mediante asignación contigua o en **extensiones** o **clústeres**. Esto puede ser aprovechado para mantener una lista donde cada entrada almacena la dirección del primer bloque de un conjunto de bloques libres contiguo, así como el número de bloques del conjunto.

Por ejemplo, el sistema de archivos XFS utiliza un árbol B+ para almacenar las direcciones de las extensiones de bloques libres y mantenerlas ordenadas por el tamaño de la extensión a la que apuntan. Así el sistema operativo puede localizar rápidamente el espacio libre necesario para satisfacer una necesidad concreta.

20.4. Sistemas de archivos virtuales

En el Apartado 19.4 vimos cómo el sistema operativo **monta** sistemas de archivos de tal forma que aparenten estar integrados en una única estructura de directorios, permitiendo a los usuarios moverse de forma transparente entre distintos dispositivos y tipos de sistemas de archivos. Para hacerlo, un sistema operativo moderno debe ser capaz de soportar de manera eficiente distintos tipos de sistemas de archivos, ocultando sus diferencias de cara a los usuarios.

Un método para implementar múltiples tipos de sistemas de archivos consiste en escribir diferentes rutinas de acceso, manipulación y gestión —de los directorios y archivos— para cada uno de los tipos de sistema de archivo existentes. Sin embargo, en lugar de esta solución, la mayoría de los sistemas operativos utilizan técnicas de programación orientada a objetos para implementar diferentes tipos de sistemas de archivos detrás de una misma interfaz de programación. Es decir, se utilizan estructuras de datos y procedimientos comunes para separar las llamadas al sistema de los detalles de su implementación real, para cada uno de los sistemas de archivos.

La implementación de un sistema de archivos está compuesta de tres niveles fundamentales: la **interfaz del sistema de archivos**, el **sistema de archivos virtual** y, finalmente, la implementación real del sistema de archivos.

20.4.1. Interfaz del sistema de archivos

El primer nivel es la **interfaz del sistema de archivos**, a la que acceden los desarrolladores a través de las llamadas al sistema. En sistemas POSIX, estamos hablando de las llamadas open(), read(), write() y close(), entre otras. Y de los descriptores de archivos con los que se identifican los archivos abiertos.

Esta interfaz es la misma sea cual sea el sistema de archivos al que se esté intentando acceder.

20.4.2. Sistema de archivos virtual

El segundo nivel es la interfaz del **sistema de archivos virtual** o **VFS** (*Virtual File System*). Este nivel es utilizado por el anterior para atender las peticiones realizadas.

Describe operaciones genéricas sobre cualquier sistema de archivos y estructuras genéricas. Por ejemplo, el **FCB virtual**, que identifica de forma unívoca a cada archivo o directorio en uso en todo el sistema, dando acceso a sus metadatos.

En Linux el **FCB virtual** se denomina **vnodo**. El **vnodo** de un archivo lo identifica de forma unívoca en todo el sistema, incluso diferenciando archivos en sistemas de archivos diferentes. Mientras que el **inodo** es un detalle de la implementación real del sistema de archivos, por lo que solo es único dentro del mismo sistema de archivos.

Este nivel cumple con dos importantes funciones:

- Separa las operaciones genéricas sobre el sistema de archivos con respecto a su implementación.

  VFS define una interfaz muy clara y común para todos los sistemas de archivos. De esta interfaz existirán diversas implementaciones en el mismo sistema, una para cada sistema de archivos diferente.

- Proporcionar un mecanismo para acceder de forma coherente a los archivos a través de la red.

  Una implementación de **VFS** no tiene que estar limitada exclusivamente a ofrecer acceso a archivos en dispositivos conectados físicamente al sistema. Las operaciones de la interfaz **VFS** pueden implementarse de tal forma que usen un protocolo de acceso a algún servidor de archivos conectado a la red.

20.4.3. Implementación del sistema de archivos

El tercer nivel es donde se implementa cada tipo de sistema de archivos o los distintos protocolos de los servidores de archivos en la red. La interfaz **VFS** recurre a la implementación correspondiente para cada tipo de sistema de archivos, para satisfacer las solicitudes de los niveles superiores.

Así, por ejemplo, un read() puede implicar que se tenga que recuperar el **vnodo** del archivo involucrado desde la tabla de archivos abiertos, usando el descriptor de archivo indicado en la llamada al sistema. Después, se invoca la operación **VFS** `read()` sobre el **vnodo**, en la

implementación concreta de **VFS** según el tipo de sistema de archivos involucrado. Será esa implementación quien extraiga del **vnodo** la información necesaria —por ejemplo, el **inodo** real del archivo en el sistema de archivos— para llevar a cabo la operación indicada, según las especificidades del sistema de archivos.

20.5. Planificación de disco

Como ya hemos comentado, es responsabilidad del sistema operativo usar los recursos del hardware de forma eficiente. Eso incluye planificar los procesos en la CPU para conseguir el mínimo tiempo de espera que sea posible o aprovechar de la mejor forma la memoria principal disponible para atender la demanda de los distintos procesos al mismo tiempo; pero también, intentar obtener el menor tiempo de acceso y el mayor ancho de banda posible en el acceso a los discos.

20.5.1. Rendimiento del acceso a disco

En un disco duro magnético el **tiempo de acceso al disco** T^d viene determinado por el **tiempo de búsqueda** T^b y la **latencia rotacional** T^r:

$$T_d = T_b + T_r$$

El **tiempo de búsqueda** T^b es el tiempo que se tarda en mover el brazo del disco hasta el cilindro deseado. Mientras que la **latencia rotacional** T^r es el tiempo que hay que esperar para que el disco gira hasta que la cabeza llegue al sector deseado del cilindro. Por lo tanto, el **tiempo de acceso al disco** es menor cuando se realizan accesos consecutivos a sectores físicamente próximos que cuando están dispersos por todo el disco.

El **ancho de banda** o **tasa de transferencia** del disco es el total de bytes transferidos en una petición dividida por el tiempo total que transcurre desde la primera solicitud de servicio hasta el final de la última transferencia, con la que se termina de atender la petición. Al considerar todo el tiempo necesario para atender la petición, a más **tiempo de acceso al disco** menor es el **ancho de banda**.

En los dispositivos de almacenamiento basados en memorias de estado sólido (véase el Apartado 18.1.3) el tiempo de acceso viene determinado por las características de la memoria —entre otros factores— lo que hace que las diferencias entre accesos secuenciales y accesos aleatorios sean mucho menos significativas.

20.5.2. Cola de E/S al disco

Cuando se solicita una operación de E/S sobre el almacenamiento, si la controladora y la unidad de disco están desocupadas, el sistema operativo puede atender la petición sobre la marcha. Pero si están ocupadas, el sistema operativo almacena la solicitud en una cola de peticiones pendientes. Cuando se resuelve una solicitud, el sistema operativo escoge otra de la cola y se comunica con el hardware para programar la siguiente petición. La cuestión es ¿cuál es el orden adecuado para escoger la peticiones de E/S de la cola, si se quiere acceder al disco de la forma más eficaz posible?

20.5.3. Planificación FCFS

En la planificación **FCFS** (*First Come, First Served*) o **primero que llega, primero servido** la cola de E/S al disco es FIFO. Es decir, que las solicitudes se atienden en orden de llegada.

Es el algoritmo de planificación más simple y es equitativo, en sentido de que atiende a todos los procesos por igual. Pero, lamentablemente, no proporciona el servicio más rápido en discos duros magnéticos, donde interesa mover el brazo del disco lo menos posible.

En Linux el **FCFS** es actualmente denominado **NOOP**. Se suele utilizar en los discos basados en memorias de estado sólido, donde reordenar las solicitudes no proporciona una mejora significativa del rendimiento, o cuando se utilizan controladoras de disco inteligentes que pueden reordenar las solicitudes según su propio criterio.

20.5.4. Planificación SSTF

En la planificación **SSTF** (*Sortest Seek Time First*) o algoritmo de **tiempo de búsqueda más corto**, de toda cola se selecciona la solicitud con el menor **tiempo de búsqueda** desde la posición actual de la cabeza. Este algoritmo de planificación primero da servicio a las solicitudes cercanas a la posición actual de la cabeza, antes de alejarse para dar servicio a otras solicitudes, dado que el **tiempo de búsqueda** se incrementa a medida que lo hace el número de cilindros que es necesario recorrer para llegar a una posición dada.

Aun así, la solución no es óptima, dado que puede provocar inanición de algunas solicitudes, si van llegando constantemente nuevas solicitudes sobre regiones cercanas a donde está actualmente la cabeza del disco.

20.5.5. Planificación SCAN y C-SCAN

En la planificación **SCAN**, algoritmo de **exploración** o del **ascensor**, el brazo del disco comienza en un extremo del disco y se mueve hacia el otro, atendiendo solicitudes a medida que pasa por cada cilindro, hasta llegar al otro extremo del disco. En el otro extremo, la dirección de movimiento de la cabeza se invierte para recorrer el disco en sentido inverso, repitiendo el proceso.

Suponiendo que las solicitudes se distribuyen de forma uniforme a lo largo del disco, es de suponer que cuando se llega a un extremo —justo antes de volver— la cantidad de solicitudes en dicho extremo será notablemente menor que dónde comenzó el barrido que acaba de terminar. Entonces ¿por qué no volver a empezar desde ese extremo?

A la variante del **SCAN** que cuando llega a un extremo vuelve al inicio, para volver a barrer desde allí, sin atender ninguna solicitud por el camino, se la denomina **C-SCAN** —de *Circula SCAN*—. Con esta variante el tiempo que tiene que esperar una solicitud para ser atendida es más uniforme que con el algoritmo **SCAN** original.

20.5.6. Planificación LOOK y C-LOOK

En teoría los algoritmos **SCAN** y **C-SCAN** hacen que el brazo recorra los cilindros del primero al último, pero normalmente no se suelen implementar así.

Por lo general, cuando en el recorrido del brazo, tras atender una solicitud, se descubre que ya no hay más solicitudes siguiendo la misma dirección, el brazo invierte la dirección sin llegar hasta el extremo del disco.

A estas variantes de **SCAN** y **C-SCAN** se las denomina **(LOOK)** y **(C-LOOK)**, respectivamente.

Linux utilizó **C-LOOK**, bajo el nombre de ***elevator***, como planificador de E/S de disco hasta Linux 2.6.

20.5.7. Planificación N-Step-SCAN, N-Step-LOOK y F-SCAN

Los algoritmos **N-Step-SCAN** y **N-Step-LOOK** son variantes de los algoritmos **SCAN** y **LOOK**, respectivamente; donde se limita a *N* el número de solicitudes que se atenderán en cada barrido del brazo del disco. Estos algoritmos funcionan de la siguiente manera:

1. Se utiliza una cola con espacio para *N* solicitudes pendientes, que se van atendiendo mientras el brazo barre el disco.
2. Mientras tanto, todas las nuevas solicitudes se incorporan a una cola diferente.
3. Cuando el brazo termina el barrido y las *N* primeras solicitudes han sido atendidas, el planificador toma otras *N* solicitudes de la segunda cola y las introduce en la primera para repetir el proceso.

Si en lugar de copiar *N* peticiones de la segunda a la primera cola, se copian todas las solicitudes pendientes, el algoritmo se denomina **F-SCAN**.

Estos algoritmos previenen un problema denominado **rigidez del brazo** —o *arm stickiness*, en inglés— a diferencia de los algoritmos SSTF, SCAN, C-SCAN, LOOK y C-LOOK, que no lo hacen. El término **rigidez del brazo** hace referencia a cuando hay un flujo continuo de solicitudes para el mismo cilindro, esto hace que con los algoritmos vistos hasta ahora, el brazo no avance por los cilindros hasta llegar al otro extremo. Como **F-SCAN**, **N-Step-SCAN** y **N-Step-LOOK** separan las solicitudes en dos colas —haciendo que las nuevas tengan que esperar— el brazo siempre continúa su barrido hacia el extremo del disco.

20.5.8. Planificación CFQ

El planificador **CFQ** (*Completely Fair Queuing*) se diseñó para compartir de forma equitativa el **ancho de banda** entre todos los procesos que solicitan acceso al disco. Fue utilizado por defecto en muchas distribuciones de Linux hasta la versión 5.0 del núcleo y funciona de la siguiente manera:

- **CFQ** mantiene una cola de solicitudes para cada proceso y en ella inserta las solicitudes síncronas de E/S. Cada cola tiene una ventana de tiempo —o **cuanto**— para acceder al disco. La longitud del **cuanto** y el tamaño máximo de cada cola dependen de la prioridad de E/S que tenga el proceso.

Las solicitudes síncronas de E/S son aquellas que hacen que el proceso permanezca en estado **esperando** hasta que se resuelve la petición. Por ejemplo, las operaciones de lectura bloqueantes, como ocurre por defecto con read(). Mientras que las solicitudes asíncronas son las que permiten que el proceso

continúe su ejecución en la CPU mientras se atiende la petición. Como las escrituras con write().

- **CFQ** mantiene una cola de solicitudes por cada prioridad de E/S, donde se insertan las solicitudes asíncronas de todos los procesos. Una solicitud asíncrona se inserta en una cola u otra según la prioridad del proceso que la generó.
- Usando el algoritmo ***round-robin***, el planificador **CFQ** recorre las colas y extrae de ellas las solicitudes durante el tiempo marcado por el cuanto de cada cola. Las solicitudes extraídas se insertan en la cola de E/S al disco, donde se ordenan para minimizar el **tiempo de búsqueda**, antes de ser enviadas al dispositivo.

Actualmente, los planificadores **CFS** y **NOOP** se consideran obsoletos. En su lugar el planificador por defecto en Linux para SSD y discos duros es **mq-deadline** —una adaptación del planificador deadline, también obsoleto—. Mientras que para dispositivos NVM Express se utiliza **NONE**, que básicamente desactiva la planificación de la E/S de disco.

Otros planificadores disponibles actualmente son BFQ —diseñado para minimizar la latencia— y Kyber.

Bibliografía

Parte de los contenidos de este documento están basados en las siguientes fuentes:

- [Bavier2000] Bavier, A. (2000). Creating New CPU Schedulers with Virtual Time. *WIP Proceedings of 21st IEEE Real-Time Systems Symposium (RTSS 2000)*.
- [Chase2009] Chase, K. y Russinovich, M. (2009). *Processes, Threads, and Jobs in the Windows Operating System*. The Microsoft Press Store by Pearson. https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=7
- [Friedman1999] Friedman, M. B. (1999). Windows NT Page Replacement Policies. *Proceedings of 25th International Computer Measurement Group Conference* (pp. 234-244).
- [Ganger2000] Ganger, G. R., McKusick, M. K., Soules, C. A. N. y Patt, Y. N. (2000). Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18 (2), 127—153.
- [Gaydov2016] Gaydov, V. (2016). *File locking in Linux*. https://gavv.github.io/articles/file-locks/
- [Gorman2004] Gorman, M. (2004). *Understanding the Linux Virtual Memory Manager*. Prentice Hall.
- [Hailperin2006] Hailperin, M. (2006). *Operating Systems and Middleware: Supporting Controlled Interaction*. Course Technology.
- [Jacob1998] Jacob, B y Mudge, T. (1998). Virtual Memory: Issues of Implementation. *Computer*, 31, 33-43. http://dx.doi.org/10.1109/2.683005
- [Microsoft2005] Microsoft Corporation. (2005). *Kernel Enhancements for Microsoft Windows Vista and Windows Server Longhorn*. SlideShare. https://www.slideserve.com/iolani/kernel-enhancements-for-windows-server-longhorn.
- [Microsoft2003] Microsoft Corporation. (2003). *Kernel Enhancements for Windows XP*. Archivado en https://web.archive.org/web/20080307105611/http://www.microsoft.com/whdc/driver/kernel/xp_kernel.mspx
- [Microsoft2018] Microsoft Corporation. (2018). *Processes and Threads: Scheduling Priorities*. https://docs.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities
- [Silberschatz2004] Silberschatz, A., Galvin, P. y Gagne, G. (2004). *Operating System Concepts with Java*. (6º ed.) John Wiley & Sons Inc.
- [Silberschatz2005] Silberschatz, A., Galvin, P. y Gagne, G. (2006). *Fundamentos de Sistemas Operativos*. (7ª ed.) McGraw Hill Interamericana.
- [SGI2006] Silicon Graphics Inc. (2006). *XFS Filesystem Structure*. Archivado en https://web.archive.org/web/20210427160429/https://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure//tmp/en-US/html/index.html
- [Solomon2005] Solomon, D. A. y Russinovich, M. (2005). Unit OS4: Scheduling and Dispatch 4.4. Windows Thread. *Windows OS Internals*. https://slideplayer.com/slide/6972581/
- [Stallings2005] Stallings, W. (2005). *Sistemas operativos: Aspectos internos y principios de diseño*. (5ª ed.) Pearson.
- [Sun2006] Sun Microsystems, Inc. (2006). *ZFS On-Disk Specification (Draft)*. https://es.scribd.com/

document/217201456/Zfs-ondiskformat

- [Tanenbaum2014] Tanenbaum, A. y Bos, H. (2014). *Modern Operating Systems*. (4ª ed.) Prentice Hall Press.
- [Wikipedia-cmalloc] C dynamic memory allocation. En *Wikipedia*. https://en.wikipedia.org/wiki/C_dynamic_memory_allocation

Índice

R

S