
Indice

Manejo de errores

Supongo que mirando los ejemplos de la semana pasada y de los apuntes te habrás dado cuenta que siempre comprobamos el valor devuelto por socket(), bind(), sendto() y recvfrom() para detectar condiciones de error.

```
#include <iostream>
#include <cerrno>           // para errno
#include <cstring>          // para std::strerror()
#include <sys/socket.h>     // para socket()

...

int fd = socket(domain, type, protocol);
if (fd < 0) {
      std::cerr << "no se pudo crear el socket: " <<
            std::strerror(errno) << '\n';
      return 3;  // Error. Termina siempre con un valor > 0
}
```

Y lo mismo hay que hacer con open(), read() y write.

Detección de errores

Al programar es conveniente que siempre comprobemos las condiciones de error de cualquier función o llamada al sistema para, si fuera necesario, terminar nuestro programa de forma ordenada. Para saber cómo nos informa una función de cualquier error es aconsejable echarle un vistazo a su documentación.

Si lo hacemos descubriremos que la mayor parte de las llamadas al sistema lo hacen siempre de la misma manera:

- **Las funciones o llamadas al sistema devuelven un valor menor de 0 en caso de error.**
- **El código que nos permite conocer la causa del error lo dejan almacenado en la variable global 'errno'**.
    - Esta es una variable entera y se declara en el archivo de cabecera <cerrno>.
    - En el mismo archivo se declara una macro para cada posible valor de 'errno'. Sin embargo, no todos los errores son posibles en todas las funciones. En la documentación de cada llamada se indican qué errores son posibles. Por ejemplo, mira la página de socket().
- **Con std::strerror() se puede obtener una cadena descriptiva para cualquier código de error**. Esta función se declara en <cstring>.

Mensajes de error

En realidad en nuestros ejemplos no estamos generando mensajes de error siguiendo la convención más común. Para comprobarlo, hagamos una prueba:

```
$ ls /nodir
ls: no se puede acceder a /nodir: No existe el archivo o el directorio
```

o dos:

```
$ /bin/ls /nodir
/bin/ls: no se puede acceder a /nodir: No existe el archivo o el directorio
```

A ahora veamos de dónde sale cada parte del error de 'ls':

- **"No existe el archivo o el directorio" es lo que devuelve std::strerror() cuando errno es EEXIST**. Es decir, cuando una llamada al sistema descubre que el archivo o directorio indicado no existe.
- **"no se puede acceder a /nodir" es la parte del mensaje con la que contribuye el propio comando 'ls'**. A fin de cuentas con lo que devuelve std::strerror() estamos seguros que el problema está en un archivo o directorio que no existe, pero ¿qué archivo o directorio?. El comando debe aportar esta información para que sepamos lo que estaba intentando hacer.
- **'ls' es el nombre del programa**. A bote pronto parece una tontería indicar que el error es de "ls" cuando está claro que el usuario sabe qué comando ha ejecutado. Pero cuando un el error lo imprime un programa invocado desde un script de BASH o desde otro programa, ayuda mucho saber quién se está quejando y por qué.

En resumen, un mensaje de error adecuado podría ser así:

```
std::cerr << "ViewNet: no se pudo crear el socket: " <<
     std::strerror(errno) << '\n';
```

Propagación de errores

Por norma, los errores deben manejarse de forma lo más local posible. Es decir, lo más cerca posible al punto donde son detectados. Por ejemplo, si el constructor de nuestra clase Socket recibe un error al usar la llamada al sistema socket(), lo mejor es que el mismo evite llamar a bind() y que muestre un mensaje de error.

Pero incluso así, lo suyo es que quien intentó crear un Socket() sea informado de que la operación no pudo realizarse, por sí debe hacer algo como consecuencia. Es decir, los errores debe propagarse hacia la función invocadora.

Recordemos que algunos errores deben hacer que el programa termine, pero los programas en C++ deben ser terminados volviendo de main(). Luego **no parece buena idea ejecutar exit() allí donde ocurre un error**, sino notificar de lo que ha pasado hacia arriba —propagando el error de funciones invocadas a invocadoras— hasta que la ejecución vuelva a main() y se pueda terminar el programa.

Estilo C

En C la solución tradicional es la misma que hemos visto usar a las llamadas al sistema. Es decir, **devolver un valor de retorno concreto para indicar el error —por ejemplo un false, 0 o un valor**

negativo— y dejar que se use "errno" allí donde sea necesario si se quiere conocer el motivo del error. Si una función falla, porque alguna de las que ella llama devolvió un error, lo que debe hacer es propagarlo hacia arriba, devolviendo a su vez el valor que indica que hubo un error.

Si necesitamos crear nuestros propios tipos de errores porque con los soportados con "errno" no es suficiente, la mejor solución es crear nuestra propia variable estilo-errno donde guardar el último código de error ocurrido, de entre la serie de errores que nos inventemos.

En los programas fifo-client.c y fifo-server.c podemos observar un ejemplo de esta forma de manejar los errores.

Excepciones de C++

Por fortuna C++ nos proporciona excepciones, que es un mecanismo muchísimo más potente que el tradicional de C.

Una excepción no es más que un objeto que contiene información sobre un error ocurrido durante la ejecución de nuestro programa. Cuando ocurre un error, se crea el objeto excepción adecuado con la información necesaria sobre el error y se "lanza" para notificar al resto del programa lo que ha ocurrido. Algunos de estas excepciones son lanzadas por las clases y funciones de las librerías que utilizamos, mientras que otros los podemos lanzar nosotros desde nuestro propio código.

Lanzar un objeto excepción desde una función es muy similar a devolverlo con "return", **solo que para lanzar excepciones se utiliza la palabra clave "throw"**. Veamos un ejemplo:

```
#include <stdexcept>
#include <iostream>

class MyClass
{
public:

	void MyFunc(char c)
	{
		if(c > 128) {
			throw std::invalid_argument("Argumento demasiado grande.");
		}
	}
};

int main()
{
	MyClass object;
	try
	{
		object.MyFunc(256);   // Causa una excepción
		std::cout << "En caso de excepción esto no se verá nunca.\n";
	}
```

```
	catch(std::invalid_argument& e)
	{
		std::cerr << e.what() << '\n';
		return 1;  // Error. Terminar con un valor diferente y > 0
	}

	return 0;  // Exito.
}
```

La palabra clave "throw" se comporta como un "return". La diferencia entre ambas es que:

- **return <valor>**, hace que la función termine y la ejecución vuelva a la función invocadora, retornando <valor>.
- **throw <valor>**, también retorna a la función invocadora llevando <valor> pero de ahí salta directamente —sin ejecutar ninguna sentencia más— a la función invocadora de esa y de ahí a la invocadora de esa y así sucesivamente hasta salir de main(); si nadie para el proceso, como veremos más adelante. **Si la excepción sale de main() sin haber sido detenida antes, el programa termina inesperadamente llamando a std::terminate()**.

Como ya sabemos, al salir de una función con “return” **todas las variables y objetos locales son destruidos y sus recursos son liberados**. Lo mismo ocurre con “throw” para cada una de las funciones de las que retorna en su recorrido hacia main(). Eso asegura que al lanzarse una excepción los recursos de los objetos que no se van a usar más son liberados correctamente, si nos hemos preocupado de implementar destructores correspondientes.

Si una parte del código necesita hacer algo especial cuando una función invocada falla con una excepción, lo que debe hacer es interceptarla. Ese “algo especial” puede ser cualquier cosa que se nos ocurra. Incluido mostrar al usuario un mensaje de error adecuado. Una vez manejado el error se puede continuar con la ejecución del programa, a partir del punto de intercepción, o relanzar la excepción para notificar el problema a funciones superiores.

Las excepciones se interceptan con un manejador de excepciones:

1. **Consisten en un bloque “try” donde se incluye el código que puede generar las excepciones que se quiere interceptar**. Tengamos en cuenta que el bloque interpretará tanto las excepciones generadas directamente por el código incluido en el bloque, como las generadas por las funciones invocadas por este o funciones invocadas por funciones invocadas por el código y así sucesivamente. En resumen, **el recorrido ascendente de una excepción lanzada con “throw” se detendrá al encontrar un bloque “try”**.
2. Después del bloque **try** se incluyen varios manejadores mediante la palabra clave **catch** sequida de una referencia o una variable entre paréntesis. Los bloques **catch** están asociados al bloque **try** anterior, de tal forma que si se lanza una excepción dentro de **try** la ejecución saltará al código del bloque **catch** cuyo objeto tenga un tipo que encaje con el del objeto usado en el **throw** para desencadenar la excepción.

Por ejemplo, si intentamos reservar una cadena de caracteres std::string de 10GB:

```
#include <iostream>
```

```
#include <string>

int main(int argc, char* argv[])
{
	try {
		std::string text(10000000000, 'x');    // Causa excepción
		std::cout << "Esta línea no se va a ejecutar nunca\n";
	}
	catch(std::bad_alloc& e) {
		std::cout << "Memoria insuficiente\n";
		return 1;  // Error. Terminar con un valor diferente y > 0
	}

	return 0;
}
```

y no tenemos memoria suficiente, **el constructor de std::string fallará con una excepción std::bad_alloc**, que será **capturada por el primer "catch" para mostrar el mensaje "Memoria insuficiente"**:

- Se escogerá el primer "catch" porque tiene el tipo que mejor encaja con el objeto de la excepción, que también es de tipo std::bad_alloc.
- Antes de ejecutar el código en el bloque "catch" se le asigna a "e" el objeto de la excepción —especificado en el "throw" que lanzó la excepción, dentro del constructor de std::string— que contiene información sobre el error.
  En un bloque "catch" se puede usar tanto "std::bad_alloc e" como "std::bad_alloc& e" pero hemos preferido usar una referencia para evitar que se genere una copia del objeto al asignarlo a "e"; de forma similar a como ocurre al pasar objetos a funciones.
- Una vez ejecutado el bloque "catch" la ejecución continúa en "return 0", sin ejecutar ningún otro bloque "catch", como ocurre en el caso de los bloques "if-else".

Si no existe ningún bloque "catch" adecuado para la excepción, esta continuará su recorrido hacia arriba hasta encontrar una función con un manejador donde encaje o hasta salir de main() y terminar el programa llamando a std::terminate().

Errores del sistema y excepciones

Como hemos comentado, podemos crear nuestras propias clases de excepción si tenemos necesidades particulares. Sin embargo, el estándar de C++ predefine unas pocas excepciones que podemos utilizar según nuestras necesidades y que se declaran en el archivo de cabecera <exception>. Una de ellas es `std::system_error`, que se utilizan para notificar errores del sistema.

Como mínimo necesita que se le indique un código de error del sistema —el valor de 'errno'— y opcionalmente una cadena descriptiva del error:

```
throw std::system_error(errno, std::system_category());
throw std::system_error(errno, std::system_category(),
	"no se pudo crear el socket");
```

Es decir, que el código para crear un socket podría quedar tal que así:

```
#include <sys/socket.h>
#include <system_error>
#include <cerrno>

...

int fd = socket(AF_INET, SOCK_DGRAM,0);
if (fd < 0)
      throw std::system_error(errno, std::system_category(),
            "no se pudo crear el socket");
```

No poder crear un socket no tiene porque ser un error crítico para una aplicación cualquiera. Pero si eso pasa en una herramienta como nuestro talk ¿qué otra cosa podemos hacer? Por lo tanto podemos interceptar la excepción en main(), mostrar un mensaje de error descriptivo y terminar el programa.

```
#include <iostream>
#include <system_error>

...

int main(int argc, char* argv[])
{
      try {

            ...

            Socket socket("127.0.0.1", 0)

            ...
      }
      catch(std::bad_alloc& e) {
            std::cerr << "mytalk" << ": memoria insuficiente\n";
            return 1;
      }
      catch(std::system_error& e) {
            std::cerr << "mitalk" << ": " << e.what() << '\n';
            return 2;
      }

      return 0;
}
```

std::system_error tiene dos métodos interesantes con información sobre el error:

- **code(), que devuelve el código de error especificado al crear el objeto**. En nuestro ejemplo el valor de 'errno'.
- **what(), que devuelve un mensaje descriptivo del error**. std::system_error interamente compone este mensaje con la cadena "no se pudo crear el socket", especificada al crear el objeto, seguida del texto descriptivo devuelto por std::strerror(errno).

Patrón protected_main()

Un patrón muy común para hacer el código más legible es separar el manejo de excepciones del resto del código de la función main(). Básicamente, consiste en extraer el código del bloque "try" para ponerlo en una función llamada protected_main().

```
int protected_main(int argc, char* argv[])
{
	...

	Socket socket("127.0.0.1", 0)

	...

	return 0;
}
```

Y luego llamar a esa función desde el bloque "try" de main():

```
int main(int argc, char* argv[])
{
	try {

		return protected_main(argc, argv)
	}
	catch(std::bad_alloc& e) {
		std::cerr << "mytalk" << ": memoria insuficiente\n";
		return 1;
	}
	catch(std::system_error& e) {
		std::cerr << "mitalk" << ": " << e.what() << '\n';
		return 2;
	}
}
```

Esto nos permite programar libremente en protected_main() las tareas y funcionalidades de nuestra aplicación, sabiendo que cualquier excepción será interceptada y tratada adecuadamente en main() antes de salir del programa.

Maneja bien los errores

Revisa tu programa.

- Asegúrate de comprobar todas condiciones de error posibles al solicitar servicios al sistema —en la clase Socket y en código que gestiona y lee el archivo abierto—. Recuerda que cualquier llamada al sistema puede fallar: socket(), bind(), sendto(), recvfrom(), open(), read(), write(), etc. Tendrás que verificar en todas ellas si hubo algún tipo de error y lanzar una excepción para notificar el error.
- Usa la excepción std::system_error para notificar errores en las llamadas al sistema.
- Intercepta las excepciones en main() e implementa protected_main().
- El manejo de la excepción debe acabar mostrar un mensaje informativo al usuario por la salida de error y termina correctamente el programa retornado por main(). El error mostrado debería ayudarte a saber dónde ocurrió y la causa.

Es importante que tengas presente esto en las futuras mejoras que hagas al programa a lo largo de la práctica. Gran parte del código de cualquier aplicación real es para el manejo de los errores del programa.

Hilos

El problema con los prototipos que tenemos hasta ahora es que queremos que hagan varias cosas al mismo tiempo:

1. **Esperar y leer de la entrada estándar el mensaje y enviarlo** con sendto().
2. **Esperar y recibir con recvfrom() un mensaje del otro extremo** y mostrarlo por pantalla.

En lugar de hacer primero (1) y después (2), como ocurre con el prototipo actual. Así, el programa podrá enviar y recibir mensajes en cualquier momento

Como ya sabemos, la forma más sencilla de hacer varias tareas a la vez en un mismo proceso es crear un hilo para cada una de ellas.

Enlazar la librería de hilos del sistema

Obviamente, la interfaz de hilos de C++ depende internamente de la librería de hilos del sistema. En sistemas como Linux esta librería es POSIX Threads y no se enlaza por defecto al compilar un programa. Tenemos que indicarlo explícitamente usando la opción '-pthread' al compilar:

```
$ g++ holamundo.cpp -o holamundo2 -pthread
```

Operaciones con hilos

En la sección operaciones con hilos de los apuntes se comenta como crear, esperar y terminar hilos. Aunque se centra mucho en la API de bajo nivel del POSIX Threads, en el ejemplo 5 se enlaza un ejemplo sencillo con std::thread, que es la clase que utilizamos en C++.

Hilos y excepciones

En un programa monohilo **cuando se lanza una excepción, C++ busca un manejador adecuado desenrollando la pila hasta llegar a main()**. En caso de **no encontrar ninguno, el programa termina inmediatamente a través de std::terminate()**.

En los programas mutihilo cada hilo tiene su propia pila, por lo que ocurre exactamente igual. **C++ busca un manejador adecuado a la excepción desarrollando la pila hasta llegar a la función principal del hilo**. Por el camino, va destruyendo los objetos locales creados en la pila durante la invocación de las funciones que llevaron al programa hasta el punto de fallo. Si durante este recorrido a la inversa no encuentra ningún manejador, el programa con todos sus hilos termina inmediatamente a través de std::terminate().

Como esto no es deseable, **la función principal de cada hilo es responsable de interceptar sus excepciones y, si el programa tiene que terminar, debe notificar a los demás hilos que se detengan** de forma que tengan posibilidad de hacerlo en condiciones seguras.

```
void my_function(/* argumentos… */)
{
	try {

	...

	catch(std::bad_alloc& e) {
		std::cerr << "error: memoria insuficiente\n";
	}

	catch(std::system_error& e) {
		std::cerr << "error: " << e.what() << '\n';
	}

	// En caso de excepción, el hilo saldrá del catch y terminará solo
	// al salir de la función por aquí.
}
```

Si un hilo crea otro hilo y queremos que las excepciones del segundo se propaguen al primero, para que sean gestionadas por los manejadores de excepción de este último, tendremos que hacerlo nosotros manualmente. Para eso podemos utilizar std::exception_ptr para pasar objetos excepción de un hilo a otro.

En el siguiente ejemplo, un hilo vital para el funcionamiento de la aplicación pasa sus excepciones std::bad_alloc& al hilo principal, porque las considera graves, y termina. El hilo principal, al detectar que la terminación fue por una excepción, la propaga como si fuera propia, haciendo que sea capturada por el manejador de excepciones en main(), que termina la aplicación.

```
#include <exception>
#include <thread>

void my_function(std::exception_ptr& eptr /*, más argumentos…, */)
{
	try {

	...

	} catch (std::bad_alloc&) {
		eptr = std::current_exception();
	}

	catch(std::system_error& e) {
		std::cerr << "error: " << e.what() << '\n';
	}

	// En caso de excepción, el hilo saldrá del catch y terminará solo
	// al salir de la función por aquí.
}
```

```
int protected_main(int argc, char* argv[])
{
	std::exception_ptr eptr {};
	std::thread my_thread(&my_function, std::ref(eptr) /*, más args...*/);

	...

	// Hacer otras cosas...

	...

	// Esperar a que el hilo termine...
	my_thread.join()

	// Si el hilo terminó con una excepción, relanzarla aquí.
	if (eptr) {
		std::rethrow_exception(eptr);
	}
}

int main(int argc, char* argv[])
{
	try {
		return protected_main(argc, argv);
	}
	catch(std::bad_alloc& e) {
		std::cerr << "ViewNet" << ": memoria insuficiente\n";
		return 1;
	}
	catch(std::system_error& e) {
		std::cerr << "ViewNet" << ": " << e.what() << '\n';
		return 2;
	}
}
```

Observa cómo hemos usado std::ref para indicar que queremos pasar 'eptr' por referencia. Esto es necesario porque **std::thread solo permite pasar o mover argumentos por valor** —al igual que también ocurre con std::vector y el resto de contenedores de la librería estándar de C++, que no permiten almacenar referencias—. Así que o usamos punteros —que son el valor de direcciones en la memoria— o usamos std::ref, **que crea un reference wrapper, un objeto** —un valor— **que internamente tiene la referencia a la variable 'eptr'**. Usamos esta opción para evitar el uso de punteros. Si quisiéramos pasar una referencia a un objeto de tipo 'const', tendríamos que usar std::cref en lugar de std::ref.

[TAREA] Implementa un prototipo multihilo

Con lo que sabes ahora resuelve los problemas de tu prototipo haciendo que ejecute en hilos separados sus tareas principales:

- **Tarea 1:** En uno de los hilos, leer de la entrada estándar el mensaje para después enviarlo con Socket::send_to().
- **Tarea 2:** En el otro hilo, recibir con Socket::receive_from() un mensaje del otro extremo y después mostrarlo por pantalla.

Mantén tu código organizado de forma reutilizable y modular, evitando mezclar conceptos. Como funciones principales de los hilos, crea funciones nuevas, fuera de la clase Socket. La clase Socket solo debe hacer una cosa: ser una sencilla interfaz C++ de las llamadas al sistema de sockets.

Recuerda que si necesitas pasar a un hilo referencias a un objeto, debes usar std::ref o std::cref.

El hilo principal debe esperar a que alguno de los dos hilos termine para iniciar la secuencia de terminación del programa.

- **Tarea 1:** Puede terminar normalmente porque el usuario ha escrito el comando “/quit” o puede terminar por una excepción en Socket::send_to().
- **Tarea 2:** Puede terminar por una excepción en Socket::receive_from()

Como el hilo principal debe detectar cuando alguno de los dos hilos termina, no puede dormirse en ninguno esperando con join(), porque entonces no detectaría la terminación del otro.

En su lugar se puede usar una variable de condición std::condition_variable en la que el hilo principal espera usando std::condition_variable::wait() y con la que las tareas notifican que van a terminar usando std::condition_variable::notifyAll().

Cuando el hilo principal despierta porque alguno de los hilos ha terminado, debe indicar a los hilos (a todos, porque realmente no sabe cuál ha terminado) que terminen, esperar por ellos usando el método join() de cada instancia de std::thread y terminar el programa.

La forma en la que el hilo principal puede cancelar a los hilos de las tareas, la veremos en el siguiente apartado.

Además, ten en cuenta que:

- **Deben manejarse correctamente los errores y las excepciones**, tanto en el programa principal como en los hilos, para que el programa termine siempre de forma controlada.
- **Cada tarea debe manejar sus excepciones, mostrando los errores al usuario y terminando su ejecución**. No es necesario propagar las excepciones al hilo principal para su manejo, pero el hilo principal si necesita saber cuando alguno de los dos hilos ha terminado para iniciar la terminación del programa. Para eso usamos la variable de condición.

- **La terminación del programa**, tanto si lo pide el usuario como si es por un error, **siempre debe ocurrir retornando de main().** Es decir, no se puede llamar a std::exit() en cualquier punto del programa.
- **El hilo principal debe asegurarse de que ambos hilos han terminado** antes de terminar el programa. Para eso **puede usar el método join() de cada objeto std::thread**. Si el proceso muestra el error "thread terminate called without an active exception", es porque ha finalizado sin haber detenido todos los hilos.

Cancelación de hilos

Es posible que tu versión actual del programa termine con este error:

```
terminate called without an active exception
Aborted
```

incluso si lo has hecho todo bien.

Eso es debido a que si el usuario escribe "/quit" mientras las tareas se están ejecutándose:

1. La **tarea 1** notifica que va a terminar mediante el uso de la variable de condición y termina.
2. El **hilo principal** despierta y termina el programa cuando aún hay hilos en ejecución.

A fin de cuentas, ¿qué ocurre con el hilo de la **tarea 2**?. Hemos comentado que debe terminarse, pero no hemos visto como hacerlo. Por eso el programa termina de forma anormal.

Lamentablemente, como vimos en cancelación de hilos en lenguajes de alto nivel, **std::thread no proporciona ninguna forma estándar de indicarle a un hilo que debe morir**, por lo que tendremos que usar nuestros propios medios.

Lo más común es pasar a los hilos una variable de tipo bool con la que señalarles cuándo deben terminar. El código de los hilos debe comprobar periódicamente el valor de dicha variable y, cuando valga **true**, terminar el hilo ordenadamente. Como esta variable va a ser leída y modificada desde hilos diferentes, debe ser atómica (véase instrucciones atómicas).

[TAREA] Coordinación entre hilos

Vamos a usar este mecanismo para implementar la cancelación de hilos. Necesitaremos una variable atómica que ambas tareas comprobarán periódicamente para ver si el hilo principal le está pidiendo que termine.

Cuando el hilo principal vaya a terminar el programa porque la tarea 1 o 2 hayan terminado, debe usará esta variable atómica compartida para **pedirle a la otra tarea que termine**, esperar a que lo hagan —usando join()— y luego terminar el programa. **Esta espera con join() es importante, porque la terminación de los hilos puede llevar algo de tiempo**. Si el proceso terminase inmediatamente tras cambiar la variable, los hilos no tendrían tiempo de terminar adecuadamente.

Un buen nombre para esta variable puede ser **quit_app**.

Cancelar hilos bloqueados

El mecanismo de cancelación comentado tiene el problema de que algunos hilos pueden estar bloqueados en algunas llamadas al sistema. Por ejemplo, de la **tarea 2** seguramente estará dormido en recvfrom() esperando a que llegue un mensaje, por lo que no verá que `quit_app == true` hasta que no llegue ese mensaje. Si no llega nunca, como el hilo principal espera a que todos los hilos terminen, antes de terminar la aplicación, la aplicación quedará bloqueada para siempre. Y lo mismo ocurre con el hilo de la **tarea 1**, que seguramente esté dormido en getline().

¿Cómo podemos hacer que esos hilos aborten la llamada al sistema que los tiene bloqueados para que comprueben el valor de las variables de cancelación? Hay varias opciones. Una de ellas es enviar una señal al hilo, pues si está en una llamada al sistema esta se verá interrumpida. Como en el caso de un error, la llamada devolverá un valor negativo y podrá en la variable global errno el valor especial EINTR.

El problema de esta opción es que tendemos que modificar los métodos que pueden ser interrumpidos para que no consideren EINTR un error. Por ejemplo:

- Socket::receive_from() ahora debe reconocer cuando recvfrom() terminó por un error real —para lanzar la excepción `std::system_error` solo en ese caso— y cuando es por EINTR.
- Si la terminación es por EINTR, no tiene sentido lanzar la excepción. Lo correcto es indicar al llamador que la función terminó sin recibir un mensaje. Eso puede hacerse devolviendo un mensaje vacío.

Como vimos en señales enviadas por otros hilos, POSIX Threads ofrece la función pthread_kill() para enviar una señal a un hilo concreto. Esta función necesita el manejador `pthread_t` del hilo actual, que se puede obtener con el método std::thread::native_handle(), de los objetos std::thread.

```
pthread_kill(thread1_task2.native_handle(), SIGUSR1);
```

De entre todas las señales disponibles, SIGUSR1 y SIGUSR2 son las mejores opciones, porque son señales pensadas para que las use el programador para lo que necesite.

Sin embargo, necesitamos instalar un manejador de señales para la señal que vayamos a usar, porque por defecto estas señales terminan el proceso. Este manejador no tiene que hacer nada —aunque te recomendamos que use write() para mostrar un mensaje por pantalla que te ayude a comprobar que funciona correctamente—. Es decir, sirve con que sea una función vacía. Solo hace falta instalarlo para evitar la muerte del proceso.

En la sección señales en sistemas operativos POSIX de los apuntes, se puede ver un ejemplo de como instalar un manejador de señales —también tiene enlaces a ejemplos más complejos—. Las únicas diferencias son que:

- No queremos manejar SIGTERM, como se ve en el ejemplo, sino SIGUSR1.
- En `sigaction::sa_flags` no debemos poner SA_RESTART, porque esa opción tiene el efecto de evitar que las llamadas al sistema sean interrumpidas al llegar la señal al hilo. Es decir, lo opuesto al motivo por el que vamos a usar señales.

[TAREA] Implementa la cancelación de hilos boqueados

Implementa el mecanismo que hemos comentado para cancelar hilos que pueden estar bloqueados en esperas indefinidas. En nuestro caso, **tanto las tareas 1 como 2**.

Los manejadores de señal son un recurso global del proceso —es decir, si un hilo los manipula lo hace para todo el proceso— así que no es mala idea dejar la tarea de instalar el manejador de SIGUSR1 al hilo principal, antes de crear ningún otro hilo.