Indice

Indice

Mejorar la eficiencia
- [TAREA] Usa archivos mapeados en memoria.

Bloqueo de archivos
- [TAREA] Bloquear el archivo al leer y escribir.

Hilos
- Enlazar la librería de hilos del sistema
- Operaciones con hilos
- Hilos y excepciones
- [TAREA] Implementa un prototipo multihilo
- Cancelación de hilos
- [TAREA] Coordinación entre hilos
- Cancelar hilos bloqueados
- [TAREA] Implementa la cancelación de hilos boqueados

Otras formas de terminar
- Operaciones con señales
- Manejo de señales en procesos multihilo
- [TAREA] Maneja las señales del sistema

Mejorar la eficiencia

Ambos programas necesitan dos llamadas al sistema para transferir y almacenar cada archivo. Por ejemplo, NetcpSend tiene que llamar a read() para leer una porción del archivo desde el disco y copiarla en la memoria. Después llama a sendto() para transferir los datos de la memoria al adaptador de red y enviarlos al otro proceso. Esto se lo repite NetcpSend una y otra vez hasta que el archivo completo ha sido enviado.

Podemos optimizar esta tarea eliminando una de las dos llamadas al sistema mediante el mapeo del archivo en la memoria. Si mapeamos el archivo completo, eliminamos la necesidad de llamar a read(). Si usáramos socket SOCK_STREAM (TCP), además, podríamos transferir todo el archivo en una sola llamada a sendto() —o en unas pocas, si el archivo es realmente inmenso—. Pero usamos socket SOCK_DGRAM, de cada vez solo podemos enviar algo menos de 64KiB —el tamaño máximo del paquete son 64KiB, pero hay dejar espacio para las cabeceras UDP, que ocupan 8 bytes— por lo que tendremos que hacer varias llamadas a sendto() para enviar todo el archivo mapeado.

En los apuntes de teoría se explica el concepto de archivos mapeados en memoria, cómo se usan y se enlazan ejemplos donde se puede ver cómo mapear y desmapear archivos.

[TAREA] Usa archivos mapeados en memoria.

Modifica tu prototipo de NetcpSend y NetcpReceive para que:

1. **Abran el archivo** en el modo adecuado para obtener un descriptor de archivo. Ojo, hasta el momento NetcpReceive mostraba el archivo por pantalla. Ahora tendrá que guardarlo en el sistema de archivos.
2. **Mapee el archivo** en la memoria.
3. **Usen sendto() y recvfrom() directamente sobre la memoria mapeada** para transferir su contenido o recibir el contenido y que se escriba directamente en ella. Es importante tener en cuenta que de cada vez solo se pueden enviar 64KiB - 8 bytes, como máximo.
4. Al terminar o en caso de error, **des-mapeen el archivo y cierren el descriptor**.

Como hemos hecho hasta ahora, **es necesario detectar las condiciones de error y lanzar la excepción adecuada** para notificar el error al resto del programa. Por eso **te recomendamos tener una clase File que siga el patrón RAII** para implementar esta funcionalidad:

1. Que cree el archivo y lo mapee desde el constructor.
2. Que tenga métodos para acceder al puntero y tamaño de la zona mapeada —o si trabajas con C++20 (GCC 10 en el Centro de Cálculo) que devuelva un objeto std::span—.
3. Que desmapee y cierre el archivo en el destructor.

En el caso de NetcpReceive, esta tarea presenta el reto adicional de que:

- **Tenemos que darle un nombre al archivo a guardar.**
- **El archivo debe tener el tamaño adecuado antes de mapearlo**. Sin embargo, el tamaño de un archivo nuevo es 0 y no sabemos cuantos bytes enviará NetcpSend hasta que los hemos recibido.

Por eso, **debes modificar NetcpSend para que envíe un mensaje con el nombre y el tamaño del archivo antes de comenzar a enviar el archivo en sí. NetcpReceive debe:**

1. **Leer ese mensaje.**
2. **Crear el archivo con el nombre indicado.**
3. **Cambiar el tamaño del archivo abierto usando ftruncate() con el tamaño indicado.**
4. Mapear el archivo.

Si has implementado la clase File que comentamos antes, el constructor debe recibir el tamaño del archivo y así todas estas operaciones pueden ocurrir en el constructor. La clase File serviría para gestionar archivos mapeados en memoria.

Si no te aclaras en cómo construir los mensajes para enviar ambos tipos de información: nombre + tamaño y luego contenido; puedes hacer un protocolo basado en texto. Es decir, primero enviar un mensaje con la cadena “nombre”\n”tamaño” y en los siguientes mensajes el contenido del archivo.

Bloqueo de archivos

¿Qué ocurre si otro proceso modifica el archivo mientras lo estamos leyendo para enviar o escribiendo al recibirlo? Lo más probable es que el archivo se corrompa al haber hilos de distintos procesos que escriben en él al mismo tiempo.

Para evitarlo, **el sistema operativo permite que un proceso bloquee el acceso de otros procesos a un archivo**. Simplemente tenemos que utilizar alguna de las llamadas al sistema disponibles con ese propósito: fcntl(), lockf() o flock().

Por ejemplo, para utilizar lockf() para bloquear el acceso al archivo abierto con el descriptor almacenado en 'fd', haríamos lo siguiente:

```
#include <unistd.h>

lockf(fd, F_LOCK, 0)  // Bloquear el acceso al archivo

...                   // Aquí va el código que manipula el archivo

lockf(fd, F_ULOCK, 0) // Desbloquear el acceso al archivo
```

De tal forma que solo un proceso al mismo tiempo podrá ejecutar el código en la sección crítica entre los dos lockf(). El resto de los procesos que lo intenten al mismo tiempo serán suspendidos en el primer lockf().

En la sección bloqueos de archivo de los apuntes se comentan algunas características de lockf() que debemos tener presentes. También se enlaza un ejemplo dónde usa para crear un archivo con el PID de un proceso.

[TAREA] Bloquear el archivo al leer y escribir.

Modifica el código que gestionar el mapeo de los archivos para bloquear el archivo tras abrirlo —antes de hacer ninguna otra cosa más con él— y lo desbloquee antes de cerrarlo con close(). Si has creado una clase File para seguir el patrón RAII, como te hemos aconsejado, el bloqueo ocurrirá en el constructor y el desbloqueo en el destructor.

Hilos

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

1. **Procesar los comandos que puede indicar el usuario**, dado que la aplicación será interactiva.
2. **Leer el fichero y enviarlo** con sendto().
3. **Recibir un archivo** recvfrom() y guardarlo.

De esta manera NetcpSend y NetcpReceive se integran único programa: Netcp.

Los comandos inicialmente admitidos pueden ser:

- **Receive [directorio]** activar el modo de recepción. Con en el modo de recepción activado, el programa puede recibir archivos de otro Netcp y guardarlos en **'directorio'**, que debe ser creado si no existe. Cada vez que un archivo termine de recibirse, se indicará que la descarga ha finalizado y su ruta, mostrando un mensaje en la salida estándar.
- **Send [nombre_archivo]** iniciará el envío del archivo indicado. Cuando el archivo termine de enviarse, se indicará con un mensaje en la salida estándar.
- **Abort**, abortará el envío. Si el comando es **Abort receive**, se desactiva el modo de recepción.
- **Pause**, pausará el envío.
- **Resume**, continuará el envío.
- **Quit,** termina la ejecución del proceso de forma ordenada, cerrando los archivos, sockets y el resto de recursos utilizados.

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 llegan 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.

Si un hilo crear 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, 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 al hilo principal 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 (...) {
		eptr = std::current_exception();
	}

	// 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 << "mytalk" << ": memoria insuficiente\n";
		return 1;
	}
	catch(std::system_error& e) {
		std::cerr << "mytalk" << ": " << e.what() << '\n';
		return 2;
	}
	catch (...) {
		std::cout << "Error desconocido\n";
		return 99;
	}
}
```

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 tres principales tareas:

- **Tarea 1.** Leer de la entrada estándar los comandos que puede realizar el usuario, interpretarlos y realizar la acción correspondiente, tal y como indicamos al comienzo de la sección. Este hilo y el hilo principal son los únicos hilos que deben estar siempre en ejecución. Si el hilo de la **tarea 1** termina, la aplicación debe finalizar.
- **Tarea 2.** Si el usuario indica en la **tarea 1** el comando Send, se crea un hilo con una tarea para enviar el archivo. Este hilo termina cuando el archivo es enviado, indicándoselo al usuario con un mensaje por la salida estándar.
- **Tarea 3.** Si el usuario indica en la **tarea 1** el comando Receive, se crea un hilo para una tarea que recibe archivos. Esta tarea la detiene **tarea 1** cuando el usuario indica el comando **Abort receive**.

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. Mientras que la clase File debe encargarse de los archivos mapeados en memoria.

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

Además, ten en cuenta que:

- **Crear un directorio** (para el comando Receive) **se hace con la llamada al sistema mkdir().**
- **La terminación del programa**, tanto si lo pide el usuario como si es por un error, **siempre debe ocurrir retornando de main().**
- **El hilo principal debe esperar por el hilo interactivo de la tarea 1**. Cuando el usuario escribe “Quit” como comando u ocurre un error grave en el hilo de la **tarea 1**, este debe finalizar. Entonces, el hilo principal, que esperaba, debe continuar su ejecución para finalizar el programa.
- **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 hilo debe manejar sus excepciones, mostrando los errores al usuario y terminando su ejecución**. Solo los errores y excepciones en el hilo de la **tarea 1** (el hilo interactivo) y el hilo principal deben desencadenar la terminación del proceso; ya que si alguna de las otras tareas falla, el usuario puede reintentarlo usando el comando correspondiente o escribir Quit para terminar el proceso.

Si al terminar tu programa recibes el error “thread terminate called without an active exception”, no te preocupes. Pronto nos pondremos con eso.

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 2** o **3** están ejecutándose, el proceso termina cuando aún hay hilos en ejecución. A fin de cuentas, el hilo principal termina correctamente cuando termina el hilo de la **tarea 1** ¿pero qué ocurre con los otros hilos?. No le hemos dicho de ninguna forma que deben morir ni esperamos a que lo hagan. Por eso el programa termina de forma anormal.

De hecho, la **tarea 1** no tiene forma de controlar las **tareas 2** y **3**. No sabe cómo poner en pausa y abortar los hilos que envían y reciben archivos cuando el usuario lo solicita mediante un comando.

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 varios casos de uso que requieren que un hilo señale algo a otro hilo. Por eso seguramente necesitaremos varias variables atómicas que algunos hilos comprobarán periódicamente para ver si otro hilo le está pidiendo que haga algo.

1. **Cuando el hilo principal vaya a terminar el programa porque la tarea 1 haya terminado, debe** usar alguna variable atómica compartida para **pedirle a las tareas 2 y 3 que terminen**, esperar a que lo hagan y luego terminar el programa. **Esta espera 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**.
2. El hilo de la **tarea 1** necesita poder indicar al hilo de la **tarea 2** que aborte, pause o continúe el envío de archivos, cuando el usuario escriba **Abort**, **Pause** o **Resume**.
3. El hilo de la **tarea 2** necesita poder indicar al hilo de la **tarea 3** que detenga la recepción de archivos, cuando el usuario escribe el comando **Abort receive**.

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, el hilo de la **tarea 3** 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.

¿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 en 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 toda 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. Es solo 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 la 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, como el de la **tarea 3**.

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.

Otras formas de terminar

Aunque demos a los usuarios una forma correcta de hacer que termine nuestro programa —como escribir **Quit**— lo más probable es que los usuarios utilicen otra forma más común entre las aplicaciones de consola. Es decir, que pulsen Ctrl+C o incluso que cierren ventana de la terminal.

Cuando un usuario pulsa Ctrl+C la terminal envía la señal SIGINT —en este caso INT viene de interrupt— al proceso en primera plano para que termine. El problema es que, por defecto, si el programa no maneja la señal adecuadamente, el proceso termina inmediatamente; sin tiempo para detener los hilos, guardar los datos, ni finalizar tareas pendientes adecuadamente. Esto, por ejemplo, puede corromper los archivos de datos del programa.

SIGINT no es la única señal que existe que puede terminar un proceso. Existen muchísimas señales más y todas terminan el proceso de manera instantánea. Por lo general eso no está mal, ya que muchas de estas señales indican condiciones anómalas, detectadas por el sistema operativo, que deben abortar el programa. Pero al menos hay tres que debemos intentar que detengan el proceso de forma segura:

- **SIGINT**, enviada cuando el usuario pulsa Ctrl+C en la terminal.
- **SIGTERM**, enviada antes de apagar el sistema.
- **SIGHUP**, enviada cuando el usuario cierra la ventana de la terminal.

Operaciones con señales

En la sección señales en sistemas operativos POSIX de los apuntes vimos como enviar señales a otro proceso y manejar las señales que llegan al nuestro.

Manejo de señales en procesos multihilo

En que manejo de señales en procesos multihilo vimos cada hilo tiene una máscara con la que puede bloquear las señales que no le interesen. Esto nos ofrece una forma muy cómoda de manejar las señales en nuestra aplicación:

1. **Todos los hilos deben bloquear las señales SIGINT, SIGTERM y SIGHUP** usando pthread_sigmask() para no tener que preocuparnos de ellas. Si lo hacemos en el hilo principal, los hilos que creemos posteriormente heredarán la máscara con estas señales bloqueadas.
2. **Creamos un hilo adicional dedicado a gestionar las señales** para todo el proceso. Este hilo también tiene que tener bloqueadas las señales SIGINT, SIGTERM y SIGHUP porque vamos a recibirlas con sigwait().
3. **La única función de este hilo es dormir en la llamada al sistema sigwait()**, a la espera de que llegue alguna de estas señales al proceso. El hilo saldrá de esa función si llega alguna de

---

las señales mencionadas. En ese caso, debe indicar al hilo de la **tarea 1** que termine —con las variables atómicas y señales de las que hablamos anteriormente— para que el hilo principal, al detectarlo, inicie la terminación de la aplicación.

Algunas consideraciones adicionales al respecto:

- En este ejemplo se puede ver cómo se usa sigwait().
- Como el nuevo hilo no reserva recursos ni hace nada más que llamar a sigwait(), el hilo principal puede usar el método std::thread::detach() con él tras crearlo. Así no tenemos que preocuparnos por su terminación ni esperar por él con std::thread::join() antes de terminar la aplicación —de hecho, no podemos—. Simplemente esperamos que sea destruido automáticamente por el sistema operativo cuando termine el proceso.

[TAREA] Maneja las señales del sistema

Ha llegado el momento de que hagas que tu programa maneje la señales correctamente. **Debes manejar al menos las señales SIGINT, SIGTERM y SIGHUP** tal y como se ha descrito. Ante la llegada de esas señales el programa debe terminar de forma segura. Exactamente de la misma manera que si el usuario hubiera escrito Quit. Es decir, deteniendo los hilos, liberando recursos, etc.