Indice

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 hayas hecho todo bien.

Eso es debido a que el proceso termina cuando aún hay hilos en ejecución. A fin de cuentas, el hilo principal que lee de teclado termina correctamente ¿pero qué ocurre con el otro hilo?. No le hemos dicho de ninguna forma que debe morir. Cuando el proceso vaya a terminar, seguramente estará bloqueado en read() o en send_to().

A veces necesitamos crear hilos para hacer algo durante toda la vida del programa, por lo que debemos hacer que terminen justo antes de terminar el proceso, para evitar el problema comentado anteriormente. Lamentablemente **std::thread no proporciona ninguna forma estándar de indicarle a un hilo que debe morir**, por lo que tendremos que usar nuestros propios medios.

Cancelación en la librería de hilos

A nivel de librería del sistema se suele poder soportar alguno de los siguientes tipos de cancelación:

- **Cancelación asíncrona**, donde un hilo puede terminar inmediatamente, en cualquier momento la ejecución de otro. En general el resultado de terminar un hilo de esta manera no está determinado, en el sentido que puede tener cualquier efecto. Pueden quedarse recursos reservados sin posibilidad de liberarlos —hasta la finalización del proceso— o las estructuras de datos compartidas podrían quedar inconsistentes si un hilo las estaba manipulado en el momento de su terminación
- **Cancelación en diferido**, donde la terminación sólo puede ocurrir en puntos concretos, denominados puntos de cancelación. Como los puntos de cancelación están documentados —pueden ocurrir dentro de ciertas llamadas al sistema— se supone que el programador debe tenerlos en cuenta para evitar los problemas de la cancelación asíncrona.

La cancelación en diferido fue diseñada para lenguajes de programación sin excepciones ni desenrollado de la pila, como C, que suele ser el lenguaje de la librería del sistema. Pero en C++ y otros lenguajes similares, antes de que el hilo termine, debería asegurarse la destrucción de todas las variables locales creadas por el camino hasta que la ejecución llegó al punto de cancelación.

Cancelación coordinada

El estándar C++11 iba a incluir un mecanismo de terminación de hilos denominado cancelación coordinada, pero finalmente esto no ocurrió porque no hubo consenso. Este mecanismo es el mismo que utilizan Java y .NET y ya se incluye en la librería de hilos boost.thread, que es muy utilizada en todo tipo de proyectos de C++.

De haberse incluido, terminar un hilo sería tan sencillo como llamar al método std::thread::request_cancellation()

```
my_thread.request_cancellation();
```

que señala al hilo “my_thread” que debe terminar. Sin embargo eso no ocurre inmediatamente sino cuando el hilo alcanza unos lugares especiales llamados puntos de cancelación. Obviamente la idea era que muchas de las funciones y llamadas que pueden bloquear el hilo por largos periodos de tiempo sean puntos de cancelación. Y por si no usamos ninguna de esas funciones, hubiéramos podido poner los nuestros propios llamando a:

```
std::this_thread::cancellation_point();
```

en el código del hilo. En la cancelación coordinada, cuando la ejecución de un hilo al que se le ha pedido terminar alcanza un punto de cancelación se lanza una excepción especial —especial porque no se utiliza para indicar un error—. Esta recorre las funciones invocadas en orden inverso, hasta llegar a la función principal del hilo, donde este termina. En el proceso se destruyen las variables locales creadas durante la ejecución del hilo —llamando obviamente a su destructor—.

Cancelación coordinada con POSIX Threads

La librería de hilos de los sistemas POSIX —llamada POSIX Threads— admite tanto la cancelación asíncrona como la diferida. Sin embargo **la implementación GNU de POSIX Threads se desvía del estándar al soporta también la cancelación coordinada cuando se usa en programas escritos en C++**. Podemos aprovecharnos de eso tanto en sistemas Linux como en otros sistemas operativos cuando se usa un compilador basado en GCC —como es el caso de MinGW en Windows—.

En POSIX Threads la terminación de un hilo se solicita invocando `pthread_cancel()`. Esa función espera recibir un manejador con el que POSIX Threads identifica al hilo. Para obtener dicho manejador a partir de un objeto std::thread se puede utilizar su método native_handle(). Por tanto, en un programa escrito en C++ para un sistema operativo POSIX se podría cancelar el hilo “my_thread” de la siguiente manera:

```
#include <pthread.h>

...

pthread_cancel(my_thread.native_handle());
```

Lamentablemente, **hacer lo anterior es una muy mala idea en la mayor parte de los sistemas operativos**, porque POSIX Threads no sabe nada de C++, ni de sus objetos, ni de la necesidad de llamar siempre al destructor de estos para liberar de forma segura los recursos que reservan. Simplemente el hilo se evaporará del proceso sin más, según las reglas de la cancelación en diferido.

Sin embargo la implementación GNU de la librería POSIX Threads se desvía del estándar POSIX en programas en C++, pues desde los puntos de cancelación lanza una excepción especial para terminar el hilo. La excepción que lanza la implementación GNU de POSIX Threads para la cancelación

coordinada es abi::__forced_unwind. Pero debemos tener presente que **este no es un mecanismo estándar de C++**, así que no funciona con otros compiladores. **La única forma correcta de terminar un hilo en C++ estándar es retornar manualmente de la función del hilo**.

Ojo, porque interceptar abi::__forced_unwind da problemas. Es la única excepción que si la interceptamos estamos obligados a relanzarla desde el **catch** para dejar que el hilo termine correctamente. Y claro, si tenemos el **catch** así desde la primera parte de la práctica, estamos interceptando todas las señales, incluida abi::__forced_unwind:

```
    } catch (...) {
        eptr = std::current_exception();
    }
```

Por suerte, como abi::__forced_unwind no es estándar, no hereda de la clase base std::exception de la que si heredan el resto de excepciones. Así que podemos interceptar todas las excepciones excepto abi::__forced_unwind —y otras que no hereden de std::exception— de la siguiente manera:

```
    } catch (const std::exception& e) {
        eptr = std::current_exception();
    }
```

Con POSIX Threads podemos incluir nuestros propios puntos de cancelación en un hilo, si lo consideramos necesario, usando:

```
#include <pthread.h>

...

pthread_testcancel();
```

[TAREA] Implementa request_cancellation()

Como hay que cancelar los hilos del programa antes de terminar el proceso, te recomendamos que implementes la función:

```
void request_cancellation(std::thread& thread);
```

de tal forma que haga terminar el hilo indicado con “thread” usando pthread_cancel() del API POSIX Threads, como hemos explicado anteriormente. **No olvides manejar los errores de la función pthread_cancel(), como hemos hecho con otras**. Las funciones de POSIX Threads devuelven el código de error directamente, en lugar de usando la variable global ‘errno’.

Cualquiera de los dos hilos puede terminar el cualquier momento:

- El hilo de Socket::send_to() puede terminar por una excepción —un error—.
- El hilo que lee de teclado puede terminar por una excepción, o por que el usuario teclee quit.

El hilo principal debe coordinar la terminación:

- Debe esperar a la finalización de cualquiera de los dos hilos anteriores.
- Si eso ocurre, debe hacer que termine el otro utilizando request_cancellation().

Pero **¿cómo puede hacer el hilo principal un std::thread::join() a dos hilos al mismo tiempo para esperar por ambos?** No puede.

Una posibilidad es que los hilos indiquen que van a terminar utilizando una variable común compartida que el hilo principal comprueba constantemente.

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

std::atomic<bool>& quit(false);

void my_function1(/* mas argumentos… */, std::exception_ptr& eptr)
{
      try {

      ...

      } catch (...) {
            eptr = std::current_exception();
      }

      // Se acabó. Este hilo se va para casa...
      quit = true;
}

void my_function2(/* mas argumentos… */, std::exception_ptr& eptr)
{
      try {

      ...

      } catch (...) {
            eptr = std::current_exception();
      }

      // Se acabó. Este hilo se va para casa...
      quit = true;
}

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

```
	std::exception_ptr eptr1 {};
	std::thread my_thread1(&my_function1, std::ref(eptr1));

	std::exception_ptr eptr2 {};
	std::thread my_thread2(&my_function2, std::ref(eptr2));


	...

	// Hacer otras cosas...

	...

	// Esperar a que un hilo termine...
	while (!quit);

	// Mátalos a todos!!!
	request_cancelation(my_thread1);
	request_cancelation(my_thread2);

	// ...y espera a que mueran antes de terminar.
	my_thread1.join()
	my_thread2.join()

	// Si algún hilo terminó con una excepción, relanzarla aquí.
	if (eptr1) {
		std::rethrow_exception(eptr1);
	}

	if (eptr2) {
		std::rethrow_exception(eptr2);
	}

	return 0;
}
```

Observa que el tipo de "quit" es std::atomic<bool> —o std::atomic_bool— **para asegurar que el acceso a la variable es atómico**, de tal forma que mientras el hilo principal comprueba el valor de la variable "quit" ningún otro hilo está cambiando su valor.

[TAREA] Usa request_cancellation()

Usa request_cancellation() como hemos explicado para que la aplicación termine correctamente, cancelando todos los hilos antes de terminar.

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 la más común entre las aplicaciones de consola. Es decir, que pulsen Ctrl+C.

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.

Envío de señales

El sistema operativo no es el único que puede enviar una señal a un proceso cuando se dan las condiciones adecuadas. **Un proceso puede enviar un señal a otro proceso utilizando la llamada al sistema kill()**.

Por ejemplo, así haríamos para enviar la señal SIGUSR1 al proceso 1234:

```
#include <sys/types.h>
#include <csignal>

...

kill(1234, SIGUSR1);
```

En caso de querer enviar una señal desde la shell o desde un script, el sistema operativo provee **el comando kill, que es un programa cuya única función es usar la llama al sistema kill() para enviar la señal que le indiquemos** al proceso especificado:

```
yo@mihost ~ $ kill 1234 -INT
```

Manejo de señales

Mediante la función signal() de la librería de sistema de los sistemas operativos que siguen el estándar POSIX —como Linux, Mac OS X y otros UNIX— **podemos configurar qué queremos que ocurra cuando llega una señal a un proceso**. Esta función también tiene una versión llamada std::signal() en la librería estándar de C++, que es la adecuada cuando se utiliza este lenguaje.

El problema es que algunos aspectos del funcionamiento de estas funciones no están estandarizados. Es decir, son específicos de cada implementación, por lo que pueden comportarse de manera diferente en distintos sistemas operativos. Por eso el estándar POSIX introdujo la función alternativa sigaction(), que ofrece un control mucho más explícito sobre el comportamiento esperado para que en todos los sistemas operativos el manejo de señales funcione igual.

En este proyecto utilizaremos std::signal() porque es más sencilla de entender que sigaction(). Si bien debemos tener presente que la primera es una función obsoleta. En cualquier proyecto real deberíamos utilizar siempre sigaction().

Esta es la definición de std::signal():

```
#include <csignal>

...

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

donde “sighandler_t” es un tipo que indica cómo deben ser declaradas —técnicamente se denomina signatura o firma— las funciones encargadas de reaccionar ante una señal del sistema. Por ejemplo así:

```
void signal_handler(int signum)
{
      ...
}
```

Tal y como indica el estándar de C++, el significado de los parámetros de la función std::signal() es el siguiente:

- **signum**, es la señal de la que queremos interceptar.
- **hander**, es el manejador para la señal indicada en ‘signum’ y puede ser:
    - **SIG_IGN**, si queremos que la señal sea ignorada.
    - **SIG_DFL**, si queremos que se use el manejador por defecto, que por lo general hace que el proceso termine.
    - **Puntero a la función que será invocada cuando una señal del tipo “signum” llegue al proceso** —como nuestra ‘signal_handler()’ del ejemplo anterior.

En este último caso, cuando llegue al proceso la señal indicada en “signum” la ejecución del código es interrumpida temporalmente para ejecutar la función manejadora de señal indicada en “handler”. Algunos detalles de este proceso son los que dependen de la implementación de la librería y por lo que se recomienda usar sigaction() en lugar de std::signal()[1].

Por ejemplo, así podríamos interceptar la señal SIGINT —la que recibe el proceso cuando el usuario pulsa Ctrl+C:

```
#include <csignal>

void int_signal_handler(int signum)
{
	if (signum == SIGINT) {
		write(1, "¡Señal SIGINT interceptada!\n");
	}
}

...

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

	...

	std::signal(SIGINT, &int_signal_handler);

	...

	return 0;
}
```

Bloqueo de señales

En general **cualquier hilo del proceso puede ser interrumpido y usado para ejecutar el manejador de una señal que llega al proceso**:

- Si un señal se deriva de la ejecución de alguna instrucción de la CPU —como SIGSEGV o SIGFPE— el sistema destina la señal al hilo que ejecutó la instrucción que la causó.
- Pero las señales dirigidas al proceso —como todas aquellas debidas a eventos externos del estilo de SIGINT, SIGTERM o SIGHUP— pueden ser entregadas a cualquiera de los hilos que no hayan bloqueado la señal explícitamente.

Esto último sobre que **los hilos pueden bloquear las señales que no le interesan** es el detalle importante del asunto. De hecho es una **práctica recomendada que las señales dirigidas al proceso se bloqueen en todos menos en uno de los hilos del proceso** —el principal, por ejemplo— para

---

[1] En el artículo “All about Linux signals” se describe en detalle cómo funcionan las señales en Linux y cómo utilizar adecuadamente las funciones del estándar POSIX para manejarlas.

tener la seguridad de que será ese el único que ejecute el manejador de señal correspondiente, evitando así problemas de concurrencia al acceder a variables globales y recursos compartidos

Bloque de un conjunto de señales

La función **pthread_sigmask() del estándar POSIX es la responsable del bloqueo** de un conjunto de señales en el hilo que la invoca:

```
#include <signal.h>

...

pthread_sigmask(int how, const sigset_t* set, sigset_t* oldset);
```

Veamos algunos detalles:

- El argumento “how” puede valer:
  - **SIG_BLOCK**, si queremos añadir las señales indicadas en ‘set’ al conjunto de señales bloqueadas actualmente.
  - **SIG_UNBLOCK**, si queremos desbloquear las señales indicadas en ‘set’ de entre las que estaban bloqueadas previamente.
  - **SIG_SETMASK**, si queremos que sólo queden bloqueadas aquellas señales incluidas en ‘set’, mientras que el resto no lo estarán.
- La función necesita dos conjuntos de señales porque el primero —set— contiene el conjunto de señales que se van a bloquear o desbloquear y el segundo —oldset— contendrá a la vuelta el conjunto de señales que estaban bloqueadas antes de llamar a la función. Por lo tanto no es necesario inicializar ‘oldset’. Incluso puede valer “nullptr” si no estamos interesados en la información que nos proporciona.

Conjuntos de señales

Como hemos visto pthread_sigmask() necesita que le indiquemos conjuntos de señales o sigset_t:

```
sigset_t set;
```

que se puede inicializar vacío —sin incluir ninguna señal en el conjunto—:

```
sigemptyset(&set);
```

o lleno —incluyendo todas las señales soportadas—

```
sigfillset(&set);
```

Además en cualquier momento podemos añadir una señal al conjunto:

```
sigaddset(&set, SIGINT);
```

o quitarla:

```
sigdelset(&set, SIGINT);
```

Una vez tenemos el conjunto de señales adecuado podemos bloquearlas para el actual utilizando pthread_sigmask().

```
void my_function1(/* mas argumentos… */, std::exception_ptr& eptr)
{
      // Primero bloqueamos la señal SIGINT y SIGTERM para este hilo.
      sigset_t set;
      sigaddset(&set, SIGINT);
      sigaddset(&set, SIGTERM);

      pthread_sigmask(SIG_BLOCK, &set, nullptr);

      try {

      ...

      } catch (...) {
            eptr = std::current_exception();
      }

      // Se acabó. Este hilo se va para casa...
      quit = true;
}
```

Seguridad frente a señales

Las señales pueden interrumpir un hilo en cualquier momento. Así que como ocurre con las variables y recursos globales compartidos entre varios hilos, **debemos tener mucho cuidado en lo que hacemos desde la función manejadora de señales**, puesto que no hay garantías del estado en el que están los recursos compartidos.

Tal es así que **el estándar POSIX tiene una lista muy pequeña de funciones de la librería que se pueden llamar de forma segura** desde el manejador de señales.

Por eso lo mejor es adoptar una solución similar a la que empleamos anteriormente para indicar a los hilos que deben terminar. Es decir, indicar al programa que debe terminar a través de una bandera atómica global. Modificando el ejemplo de la terminación de hilos para que también soporte señales, quedaría algo así.

```
#include <atomic>
#include <csignal>

std::atomic<bool> quit(false);

void int_signal_handler(int signum)
```

```
{
	quit = true;
}

int protected_main(int argc, char* argv[])
{
	// Cuando llegue SIGINT, invocar int_signal_handler
	std::signal(SIGINT, &int_signal_handler);

	...

	// Esperar a que un hilo termine...
	while (!quit);

	// Mátalos a todos!!!
	request_cancelation(my_thread1);
	request_cancelation(my_thread2);

	...

	return 0;
}
```

Obviamente es necesario comprobar el valor de la bandera en aquellos puntos del programa desde donde se pueda iniciar la finalización del mismo.

[TAREA] Maneja las señales del sistema

Ha llegado el momento de que hagas que tu programa maneje la señales correctamente:

1. Debes interceptar al menos SIGINT, SIGTERM y SIGHUP.
2. 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.

Además debes de preocuparte de que sólo uno de los hilos del programa se haga cargo de manejar estas señales. Obviamente una muy buena opción es usar el hilo principal pero puedes hacerlo como prefieras.

NetCp multiarchivo

En esta actividad haremos que nuestro prototipo mejore añadiendo funcionalidades extra.

Argumentos de la línea de comandos

¿Te imaginas que cada programa tuvieran ventanas de colores y formas diferentes? ¿qué algunos programas llamaran al menú Editar de forma diferente y que dentro no todos tuvieran las clásicas operaciones de Seleccionar, Copiar, Pegar, etc? ¿Si algunos programas usasen Ctrl+V para copiar en lugar de Ctrl+C?

Que nuestro entorno de trabajo sea consistente es fundamentalmente para que nos sintamos cómodos y seamos productivos. Y esto se aplica tanto a las aplicaciones gráficas como de la consola. Por eso, al decidir cómo los usuarios deben pasar las opciones de línea de comandos a nuestro programa, es conveniente que veamos como lo hacen programa similares.

Si por ejemplo la inmensa mayoría usan:

- -h para mostrar la ayuda,
- -r para operar de forma recursiva,
- -f para forzar,
- -q para no mostrar mensajes (quiet),
- -v para mostrar la versión o para dar más detalles durante la ejecución del programa (verbose),
- -d para mostrar mensajes de depuración (debug) o para convertir el programa en un demonio,

¿por qué no hacerlo igual en nuestro programa? Sin duda los usuarios lo agradecerán.

En todo caso el estándar POSIX recomienda respetar una serie de convenciones respecto al manejo de la línea de comandos:

- Un argumento es una opción si empieza '-'.
- Múltiples opciones pueden encadenarse detrás de un '-' siempre que ninguna de ellas lleve ningún argumento. Por ejemplo, '-abc' es equivalente a '-a -b -c'.
- Cada opción es un único caracter alfanumérico. Por ejemplo, '-a', '-p' o -'6'.
- Algunas opciones pueden requerir un argumento. Por ejemplo, la opción '-o' del comando g++ requiere un argumento —el nombre del archivo de salida— que se pone a continuación de la opción: 'g++ -o holamundo'.
- Una opción y su argumento pueden aparecer separados o no. Es decir, el espacio que los separa es opcional. Por ejemplo, 'g++ -o holamundo' es equivalente a 'g++ -oholamundo'.
- En la línea de comandos se espera generalmente que las opciones vayan antes que otros argumentos que no sean opciones. Por ejemplo 'g++ -o holamundo holamundo.cpp', donde holamundo.cpp es un argumento que obviamente no es una opción porque no empieza por '-'.
- El argumento '--' se usa para indicar que ninguno de los siguientes argumentos debe ser interpretado como una opción aunque empiece por '-'. Por ejemplo 'cat -- -p' mostraría el contenido del archivo '-p' en lugar de interpretarlo como una opción de cat.
- Un argumento que es un guión sin más caracteres es interpretado como un argumento no opción. Por convención se suele usar en el lugar de un nombre de archivo para indicar que se use como tal la entrada o la salida estándar. Por ejemplo, 'ps | vi -' permite editar la salida del comando 'ps' que llega a 'vi' por su entrada estándar, a través de una tubería.
- Las opciones pueden aparecer en cualquier orden y múltiples veces. La interpretación exacta de esto queda en manos del programa.

El proyecto GNU añade a estas opciones la posibilidad de tener opciones largas, que son mucho más sencillas de recordar y entender para los usuarios:

- Las opciones largas consisten en '--' seguido por el nombre dicha opción. Sólo se pueden usar caracteres alfanuméricos o guiones.
- Por lo general consisten en entre una y tres palabras separadas por guiones. Por ejemplo '--verbose', '--ignore-case' o '--file-type'.
- Las opciones largas se pueden abreviar siempre que la abreviación sea única y no haya conflicto con otras opciomes. Por ejemplo, 'ls --directo' es equivalente a 'ls --directory'.
- Para especificar un argumento para una opción larga se escribe '--name=value'.

Getopt

Procesar la línea de comandos para contemplar todas estas convenciones no es sencillo. Por eso el estándar POSIX incluye la función getopt()[2] para que lo haga por nosotros.

Veamos cómo usarla con un ejemplo:

```
#include <iostream>        // para std::cerr
#include <string>          // para std::string()
#include <unistd.h>        // para getopt(), optarg, optind, …
#include <vector>

// Estructura que usaremos para guardar los argumentos de línea de comandos
// que han sido indicados por el usuario.
struct CommandLineArguments
{
      bool show_help = false;
      bool server_mode = false;
      unsigned short conn_port = 0;
      std::vector<std::string> other_arguments;

      CommandLineArguments(int argc, char* argv[]);
};

CommandLineArguments::CommandLineArguments(int argc, char* argv[])
{
      // Mira getopt(argc, argv, "hsp:01") mas abajo, en el while:
      //
      //    "hsp:01" indica que nuestro programa acepta las opciones
      //    "-h", "-s", "-p", "-0" y "-1".
      //
      //    El "p:" en la cadena indica que la opción "-p" admite un
      //    argumento de la forma "-p argumento"
```

[2] En la Wikipedia también hay un artículo muy detallado sobre getopt() en los distintos lenguajes de programación.

```
//    En cada iteración la variable "c" contiene la letra de la
//    opción encontrada. Si vale -1, es que ya no hay más
//    opciones en argv.
int c;
while ( (c = getopt(argc, argv, "hsp:01")) != -1)
{
      // Recuerda que "c" contiene la letra de la opción.
      switch (c) {
      case '0':
      case '1':
            std::cerr << "opción " << c << std::endl;
            break;
      case 'h':
            std::cerr << "opción h\n";
            show_help = true;
            break;
      case 's':
            std::cerr << "opción s\n";
            server_mode = true;
            break;
      case 'p':
            // Esta opción recibe un argumento.
            // getopt() lo guarda en la variable global "optarg"
            std::cerr << "opción p con valor " << optarg << std::endl;
            conn_port = std::atoi(optarg);
            break;
      case '?':
            // c == '?' cuando la opción no está en la lista
            // getopt() se encarga de mostrar el mensaje de error.
            throw std::invalid_argument("Argumento de línea de comandos
      "
                                        "desconocido");
      default:
            // Si "c" vale cualquier otra cosa, algo fue mal con
            // getopt(). Esto no debería pasar nunca.
            throw std::runtime_error("Error procesando la línea de "
                                     "comandos");
      }
}

// Llegados aquí hemos procesado todas las opciones.
// Es decir, todos los argumentos del tipo -<letra>.
//
// Por ejemplo, en el comando "ls -lt -R /bin/sh" son opciones
// 'l', 't' y 'R'. Mientras que '/bin/sh' es un argumento para el
// programa pero no es una opción.

// En este punto la variable global "optind" contiene el índice del
```

```
      // argumento de la línea de comandos que no es una opción.
      // Si optind == argc, es que ya no quedan más argumentos en argv
      // para procesar.
      if (optind < argc) {
            std::cerr << "-- empezamos con los argumentos no opción --\n";
            for (; optind < argc; ++optind) {
                  std::cerr << "argv[" << optind << "]: " <<
                        argv[optind] << '\n';
                  other_arguments.push_back(argv[optind]);
            }
      }
}

int main(int argc, char* argv[])
{
      CommandLineArguments arguments(argc, argv);
}
```

Para probar como funciona solo queda compilar y ejecutar el programa. Por ejemplo, si lo ejecutamos así:

```
./a.out -s -p 8000 -0 ruta/a/archivo
```

el programa mostrará la siguiente salida:

```
opción s
opción p con valor 8000
opción 48
-- empezamos con los argumentos no opción --
argv[5]: ruta/a/archivo
```

Aunque no está soportado por el estándar POSIX, **la implementación GNU de getopt() admite que una opción sea seguida de un par de ':' para indicar que dicha opción admite un argumento pero que ponerlo es opcional**. Por ejemplo, con la cadena de opciones "hsp::01" la opción "-p" puede aparecer sola "-p" o como "-p argumento".

Si se desea dar soporte tanto a opciones largas como cortas, GNU ofrece la función getopt_long():

- Parsing Long Options with getopt_long — The GNU C Library
- Example of Parsing Long Options with getopt_long — The GNU C Library
- Example 2 (using GNU extension getopt_long) — Wikipedia

[TAREA] Implementa el modo servidor

Modifica tu prototipo de Netcp añadiendo opciones de la línea de comandos para soportar los siguientes casos de uso:

---

1. `Netcp -h`
   muestra la ayuda del programa y termina.
2. `Netcp -s -p 8000`
   Inicia Netcp en **modo recibir** escuchado en el puerto 8000 —o el que sea que se haya indicado con la opción '-p'—
3. `Netcp -c 10.2.1.8 -p 8000`
   Inicia Netcp en **modo cliente** que se conecta al servidor 10.2.1.8 en su puerto 8000 —según lo indicado por las opciones '-c' y '-p'—. Localmente el cliente debe usar un socket con un puerto cualquiera asignado por el sistema operativo.

NOTA: Lo mejor es procesar la línea de comandos desde protected_main(), dejando main() solo para el manejo de excepciones.

En el modo cliente el programa se debe seguir comportando como hasta ahora, con la diferencia de que la dirección IP y el puerto del socket en el otro extremo de la conexión ahora se indica a través de la línea de comandos.

Para que sea un sistema multi-archivo, se permitirá el envío de múltiples archivos simultáneamente, con lo que se creará un thread por cada archivo que se envíe. El thread desaparecerá cuando se termine el envío. Cada thread leerá un archivo y lo enviará al destino indicado.

Necesitamos controlar el número de threads creados, con lo que se sugiere utilizar una estructura tipo set de pairs, que asocie el identificador de archivo con el que estamos trabajando, con el thread que lo gestiona, al crear un thread debemos añadir una entrada en el conjunto, y cuando la thread termine su trabajo, eliminar la entrada:

```
std::set<std::pair<int, std::thread *>> active_threads;
```

Esta estructura nos servirá también para terminar de manera controlada todos los threads activos a través de request_cancellation.

El modo servidor también es muy parecido, sólo que su dirección y puerto vendrá fijada por parámetros. También debe almacenar las direcciones y puertos desde los que está recibiendo archivos y enviar confirmación de que se ha recibido correctamente.

Cambiaremos la funcionalidad del modo servidor, y en vez de mostrar los ficheros por pantalla los almacenará en un fichero que creará para almacenar los datos recibidos. Se permitirá la recepción de múltiples archivos simultáneamente.

Para poder implementar esto, necesitamos una estructura para enviar no solo los datos, sino identificar qué fichero y datos estamos enviando y con que descriptor de archivo lo estamos manejando. Podríamos utilizar un struct similar al que se indica a continuación, donde se incluye además de los datos a enviar, información del nombre del archivo que se está enviando, qué tipo de paquete de envío y el identificador que se utiliza en el emisor para referirnos a este archivo.

```
// Estructura de los mensajes
struct Message {
```

```
   int identificador;        // Identificador para el cliente
   std::array<char, 25> nombreArchivo;  // Nombre del archivo en el
cliente
   int tipoMensaje;  /// Tipo de mensaje que se está enviando
                     // Primer mensaje
                     // Mensaje intermedio
                     // Ultimo mensaje
                     // Mensaje único


     std::array<char, 1024> text;    // Igual que "char text[1024]"
                                     // pero recomendado así en C++

     ...                             // Otros campos del mensaje

};
```

Necesitamos almacenar la información de quien nos envía el archivo, el nombre del archivo y el identificador de archivo. Para esto lo más sencillo es que el servidor:

1. Tenga una list, vector, map o set de direcciones de los clientes de los que está recibiendo archivos. Un set de tuplas de <dirección IP en 32 bits, número de puerto, nombre de archivo, identificador de archivo,file descriptor del fichero abierto> creado en el servidor es una buena opción para controlar los envíos.

   ```
   std::set<std::tuple<uint32_t, in_port_t, std::string, int>>
   receiving_data;
   ```

2. Cuando reciba un mensaje consulte en el tipo de mensaje para ver si se trata de un nuevo envio, o es un envío en proceso.
   - Si es un nuevo envio, se debe crear un archivo con el nombre indicado en el mensaje, y añadir un elemento al set, se realizará un write de los datos en este archivo.
   - Si el envio es de un archivo que ya existe, hay que buscar en el set el identificador de archivo para escribir los datos en él utilizando el identificador de archivo ya creado.
   - Si es el último mensaje, con el último trozo de archivo, se escribirán los datos en el fichero, se cerrará el descriptor de archivo y se eliminará del set la tupla afectada.

[OPCIONAL] Puedes intentar soportar argumentos de línea de comandos en versión larga: --help, --client, --server y --port.

Por último, **no te olvides de manejar los posibles errores adecuadamente.** Es decir, comprobar condiciones de error y lanzado e interceptando las excepciones correspondientes.

Variables de entorno

Otra forma de pasar opciones, configuraciones e información sobre el sistema operativo a las aplicaciones es a través de las variables de entorno.

Estas variables pueden ser configuradas por un proceso por ejemplo la shell, a través de sus archivos de configuración ~/.bash_profile, ~/.bashrc, /etc/profile y /etc/bash.bashrc— son heredadas por sus hijos, que a su vez pueden cambiar sus valores o crear variables nuevas.

Ejecutando 'env' en la terminal podemos hacernos una idea de todas las variables de entorno que estarán a disposición de nuestro programa cuando lo ejecutemos.

```
$ printenv
LANGUAGE=
LOGNAME=jesus
PATH=/usr/local/bin:/usr/local/sbin:/usr/sbin:/usr/bin:/sbin:/bin
USER=jesus
LANG=es_ES.UTF-8
TMPDIR=/tmp/user/1000
DISPLAY=:0
PWD=/home/yo/
TEMP=/tmp/user/1000
TMP=/tmp/user/1000
PAGER=less
TERM=xterm
EDITOR=vim
DESKTOP_SESSION=plasma
VISUAL=vim
XDG_SESSION_DESKTOP=KDE
LESS=-F -g -i -M -R -S -w -X -z-4
HOME=/home/jesus
TEMPDIR=/tmp/user/1000
SHELL=/bin/bash
```

Para acceder a estas variables desde un programa en C o C++ la librería estándar ofrece dos funciones:

- getenv() para obtener de una variable de entorno.
- setenv() para cambiar el valor de un variable de entorno o crear una nueva.

Por ejemplo:

```
#include <iostream>
#include <cerrno>     // para errno
```

```
#include <cstring>    // para std::strerror()
#include <cstdlib>    // para std::getenv() y std::setenv()

int main()
{
      // Si getenv() devuelve nullptr, es que la variable
      // especificada no existe.
      const char* path = std::getenv("PATH");
      if (path != nullptr)
            std::cout << "Tu PATH es: " << path << '\n';

      // El tercer argumento de setenv() indica que la variable debe
      // modificarse en caso de que ya exista. Si valiera false,
      // setenv() sólo sería capaz de crear variables nuevas.
      int result = setenv("HOME", "/", true);
      if ( result < 0 )
            std::cerr << "setenv error: " << std::strerror(errno) << '\n';
      else
            std::cout << "El nuevo directorio personal del usuario es: /\n";
}
```

[TAREA] Soporta nombres de usuario

Sabiendo esto, coge tu versión de NetCp y modifícala para añadir la opción '-u username' de tal forma que:

1. Se envíe el nombre del usuario especificado con la opción '-u' en cada mensaje enviado al servidor. Se puede añadir un campo nuevo al mensaje para soportar esta opción que indique el nombre del usuario.
2. Si el usuario no especifica la opción '-u' en la línea de comandos, el programa debe usar por defecto el valor de la variable de entorno USER.
3. El servidor cuando intente crear el nuevo archivo, consultará el nombre de usuario a ver si este usuario existe en el sistema actual, para ello podemos utilizar la llamada al sistema, getpwnam() que devuelve una estructura struct passwd donde estan los datos de ese usuario. Si no existe el usuario devolverá un error, con lo que crearemos el fichero con el usuario por defecto:

   ```
   #include <sys/types.h>
   #include <unistd.h>
   #include <pwd.h>

    struct passwd *getpwnam(const char *name);

   struct passwd {
            char   *pw_name;      /* username */
            char   *pw_passwd;    /* user password */
            uid_t   pw_uid;       /* user ID */
   ```

```
    gid_t   pw_gid;       /* group ID */
    char   *pw_gecos;     /* user information */
    char   *pw_dir;       /* home directory */
    char   *pw_shell;     /* shell program */
};
```

4. Solamente root puede cambiar los propietarios del archivo, con lo que consultaremos si somos root con la llamada al sistema getuid(), si nos devuelve 0, sabremos que ejecutamos como root y podemos cambiar el propietario del archivo.

   ```
   #include <unistd.h>
   #include <sys/types.h>

   uid_t getuid(void);
   ```

5. Si somos root podemos cambiar el propietario del archivo, para eso, una vez recibido por completo el fichero, utilizamos la llamada chown para cambiar el propietario.

   ```
   #include <sys/types.h>
   #include <unistd.h>


   int chown(const char *path, uid_t owner, gid_t group);
   ```

Ojo con los recursos compartidos

En este punto nuestro programa puede tener un pequeño error. Lo que ocurre es que son tan bajas las probabilidades de que nos de problemas que seguramente no nos hayamos dado cuenta.

Al crear el código que se encarga de enviar los datos al servidor hemos añadido una estructura de datos en la que se guardan todas las threads que se están ejecutando actualmente.

Independientemente de cómo hemos creado la estructura, todos los hilos pueden acceder de forma concurrente —es decir, al mismo tiempo— a la estructura de datos.

Si todos los acceso fueran para consultar la estructura de datos, esto no sería un problema. Pero **desde el momento en que uno de los hilos quiera hacer una modificación pueden aparecer problemas**. Tengamos en cuenta que nada garantiza que estos cambios no vayan a hacerse al mismo tiempo que el otro hilo recorre la estructura para leerla, encontrando inconsistencias inesperadas.

Por ejemplo se puede estar creando simultáneamente un hilo a la vez que otro está terminando y eliminando su entrada de la estructura.

Para resolver este problema necesitamos usar mecanismos de sincronización que nos aseguren que la inserción es ejecutada completamente por uno de los hilos antes de que pueda entrar el otro.

Mutex

Como ya deberíamos saber la solución está en usar semáforos binarios o mutex y obviamente la librería estándar de C++ trae unas cuantas versiones con diferentes características. De todo ellas, como simplemente necesitamos controlar el acceso de forma exclusiva a nuestra función, con usar std::mutex será suficiente.

Sólo tenemos que **declarar el mutex de forma que pueda ser accedido de forma compartida desde lo vayamos a usar**, bloquearlo antes de entrar en la porción de código a proteger —la sección crítica— y desbloquearlo al salir de dicha porción de código:

```
std::mutex mutex;
mutex.lock()

...

mutex.unlock()
```

Sin duda no podría ser más sencillo ¿o sí?.

Salvaguardas de bloqueo

El problema de esta aproximación es que cuantos más puntos de salida tiene el código que quieres proteger más posibilidades de que te olvides de liberar el mutex antes de salir de la sección crítica. Por muy sencillo que sea el código es muy fácil que nos olvidemos de desbloquear el mutex antes de retornar. Si eso pasa, ningún otro hilo podrá volver a entrar en la función. Ni siquiera el hilo que lo dejó bloqueado por error.

Para evitar accidentes la librería estándar de C++ trae algunas salvaguardas, como por ejemplo std::lock_guard, std::unique_lock y similares. Básicamente la idea es crear localmente un objeto de salvaguarda usando nuestro mutex, que es bloqueado automáticamente en ese momento. Cuando salimos del alcance de la función, este objeto es destruido y con ello se libera el mutex:

```
class MiClase
{
	public:
		...

		void mi_función_segura();

		...

	private:
		std::mutex mutex_;
```

```
            ...
};

void MiClase::mi_función_segura()
{
      // El mutex es miembro de la clase. Así que es compartido por todos
      // los hilos que llamen a este método de un mismo objeto MiClase.
      std::lock_guard<std::mutex> lock(mutex_);

      // Sección crítica.
      // Sólo un hilo a la vez puede llegar a este punto.

      ...

      // El mutex será liberado automáticamente al salir del método
}
```

Así evitamos que el mutex pueda quedar bloqueado por error.

Si no queremos proteger un método o una función completa sino sólo una porción de ella, lo único que tenemos que hacer es usar la salvaguarda dentro de un “alcance anónimo” que rodee la sección crítica:

```
void MiClase::mi_función_segura()
{
      // Aquí el código que no hay que proteger

      ...

      {     // Aquí comienza el alcance anónimo
            std::lock_guard<std::mutex> lock(mutex_);

            // Sección crítica.

            ...

            // El mutex será liberado automáticamente al salir del alcance
      }

      // Aquí más código que no hay que proteger

      ...
}
```

[TAREA] Evita condiciones de carrera

¿Recuerdas el contenedor donde se guardan los threads y los identificadores de fichero? Bien, pues usa lo aprendido para evitar condiciones de carrera cuando varios hilos accedan a él al mismo tiempo para manipularlo.