Unidad 2

Comunicación entre procesos

Jesús Torres

Vamos a desarrollar un sistema de backup distribuido en dos programas que se comunican mediante tuberías con nombre (FIFO) y señales, aplicando los conocimientos de manipulación de archivos de la Unidad 1 y aprendiendo técnicas de comunicación entre procesos.

Contenidos

1. Actividad 2

1.1. Objetivo

Vamos a crear un sistema de backup compuesto por dos programas que se comunican entre sí:

- `backup-server`: Un servidor que se ejecuta en segundo plano y espera solicitudes de backup. Cuando recibe una solicitud, copia el archivo indicado al directorio de backups.
- `backup`: Un cliente que solicita al servidor que haga una copia de seguridad de un archivo específico.

La comunicación entre ambos programas se realizará mediante:

- **Tubería con nombre (FIFO)**: Para enviar la ruta del archivo a copiar desde el cliente al servidor.
- **Señales**: Para notificar al servidor que hay una nueva solicitud pendiente.

Ambos programas deben cumplir con los siguientes requisitos:

- Utilizarán exclusivamente las llamadas al sistema `open()`, `read()`, `write()`, `close()`, `mkfifo()`, `unlink()`, `kill()`, `sigwaitinfo()`, `sigprocmask()` y similares, tal y como se explica en los apéndices.
- Reutilizarán la función `copy_file()` desarrollada anteriormente para realizar las copias de los archivos.
- Antes de finalizar, los programas cerrarán todos los descriptores de archivo abiertos, liberarán correctamente cualquier otro recurso reservado y finalizarán con el código `EXIT_SUCCESS` (`0`).

1.2. Arquitectura del sistema

Tal y como se ilustra en la Figura 1, el sistema funciona de la siguiente manera:

1. El servidor `backup-server` se inicia y crea una tubería con nombre (FIFO) en un directorio de trabajo específico.
2. El servidor también crea un archivo con su PID (identificador de proceso) para que los clientes puedan encontrarlo.
3. El servidor queda bloqueado esperando recibir la señal `SIGUSR1`.
4. Cuando un cliente `backup` quiere hacer una copia de seguridad:
   - Escribe la ruta del archivo en la FIFO.
   - Envía la señal `SIGUSR1` al servidor usando su PID.
5. El servidor despierta al recibir la señal, lee la ruta de la FIFO y copia el archivo al directorio de backups.
6. El servidor vuelve a quedar bloqueado esperando la siguiente señal.

```
shape: sequence_diagram
backup: "backup\n(cliente)"
fifo: "FIFO\n(backup.fifo)"
server: "backup-server\n(servidor)"

loop: {
  server."sigwaitinfo() bloqueado\nesperando SIGUSR1"

  backup -> fifo: "1. Escribe ruta del archivo"
  backup -> server: "2. Envía señal SIGUSR1"

  server."3. sigwaitinfo() despierta"

  server -> fifo: "4. Lee ruta desde FIFO"

  server."5. copy_file()\ncopia el archivo"
  server."Vuelve a sigwaitinfo()"
}
```

Figura 1: Flujo de comunicación entre cliente y servidor de backup

1.3. Configuración del sistema

1.3.1. Variable de entorno

Ambos programas utilizan la variable de entorno `BACKUP_WORK_DIR` para conocer el directorio donde se encuentran los archivos de comunicación:

- `backup.fifo`: La tubería con nombre para enviar las rutas de los archivos.
- `backup-server.pid`: El archivo que contiene el PID del servidor.

Esta variable debe estar definida y apuntar a un directorio existente antes de ejecutar cualquiera de los dos programas.

Para leer el valor de una variable de entorno se utiliza la función `getenv()`. Se aconseja crear una función para leer variables de entorno que devuelva un `std::string` en lugar de un puntero `char*`, para evitar problemas si la variable no está definida:

```
std::string get_environment_variable(const std::string& name)
{
    char* value = getenv(name.c_str());
    if (value)
    {
        return std::string(value);
    }
    else
```

```
    {
        return std::string();
    }
}
```

Acontinuación se muestra un ejemplo de la configuración necesaria para ejecutar los programas:

```
$ export BACKUP_WORK_DIR=/tmp/backup-work
$ mkdir -p /tmp/backup-work
```

1.4. Programa backup-server

1.4.1. Línea de comandos y funcionamiento básico

El programa `backup-server` acepta los siguientes argumentos de línea de comandos:

```
backup-server [DIRECTORIO_DESTINO]
```

Donde:

- `DIRECTORIO_DESTINO` (opcional): Ruta del directorio donde se guardarán las copias de seguridad. Si no se indica, se utilizará el directorio actual de trabajo.

El servidor realiza las siguientes acciones al iniciarse:

1. Valida que la variable `BACKUP_WORK_DIR` esté definida.
2. Verifica que el directorio de destino existe y es accesible.
3. Comprueba que no haya otro servidor ejecutándose.
4. Crea la tubería con nombre (FIFO).
5. Escribe su PID en el archivo `backup-server.pid`.
6. Configura el manejo de señales para bloquear `SIGUSR1`.
7. Abre la FIFO para lectura.
8. Muestra un mensaje indicando que está listo.
9. Entra en un bucle infinito esperando señales.

Ejemplo de ejecución:

```
$ backup-server /tmp/backups
backup-server: esperando solicitudes de backup en /tmp/backups
```

O usando el directorio actual:

```
$ cd /tmp/backups001
$ backup-server
backup-server: esperando solicitudes de backup en /tmp/backups001
```

1.4.2. Inicialización del servidor

1.4.2.1. Verificar que no haya otro servidor ejecutándose

Antes de iniciarse, el servidor debe comprobar si ya existe otro servidor ejecutándose. Para ello, debe intentar leer el archivo `backup-server.pid` del directorio de trabajo. Si el archivo existe:

1. Leer el PID contenido en el archivo.
2. Usar `kill()` con la señal `0` para verificar si el proceso existe (ver la Sección B.7).
3. Si el proceso existe, mostrar un error y terminar.
4. Si el proceso no existe, mostrar una advertencia y continuar (el archivo es de un servidor anterior que terminó inesperadamente).

1.4.2.2. Crear la tubería con nombre

Para crear la FIFO se utiliza la función `mkfifo()`, descrita en la Sección A.1.

Si la FIFO ya existe –por ejemplo, de una ejecución anterior– se debe eliminar primero usando `unlink()` y luego crearla de nuevo.

La FIFO debe crearse con permisos `0666` (lectura y escritura para todos los usuarios).

1.4.2.3. Escribir el archivo PID

El servidor debe escribir su PID en el archivo `backup-server.pid` del directorio de trabajo.

Para obtener el PID del proceso actual se utiliza `getpid()`.

El archivo debe crearse con permisos `0644` y contener el PID como una cadena de texto seguida de un salto de línea.

1.4.2.4. Configurar el manejo de señales

El servidor debe configurar el manejo de señales para dos propósitos diferentes que requieren estrategias distintas:

- **Manejo síncrono de `SIGUSR1`:** El servidor utiliza esta señal como notificación de que hay una nueva solicitud de backup pendiente en la FIFO. Para procesarla de forma controlada dentro del bucle principal, se bloquea `SIGUSR1` (ver la Sección B.4) de la siguiente manera:

  1. Crear un conjunto de señales vacío con `sigemptyset()`.

2. Añadir `SIGUSR1` al conjunto con `sigaddset()`.
3. Bloquear las señales en este conjunto con `sigprocmask()` usando `SIG_BLOCK`.

Después, en el bucle principal, el servidor esperará la llegada de `SIGUSR1` usando `sigwaitinfo()` (ver la Sección B.5).

> **Importante**
>
> Es fundamental bloquear la señal antes de usar `sigwaitinfo()`. Si no se bloquea, la señal se entregaría de forma asíncrona al proceso y el comportamiento sería impredecible, provocando seguramente la terminación inmediata del proceso.

- **Manejo asíncrono de señales de terminación:** El servidor debe responder apropiadamente a señales que solicitan su terminación (`SIGTERM`, `SIGHUP`, `SIGQUIT`, `SIGINT`), pero necesita hacerlo de forma segura. Para ello, se deben instalar manejadores (ver la Sección B.3) que:

  1. Usen `write()` para mostrar un mensaje indicando que se ha recibido la señal.

     ```
     backup-server: señal de terminación recibida, cerrando...
     ```

  2. Establecer a `true` la variable global `quit_requested` que, como veremos posteriormente, se comprobará en el bucle principal para salir de forma ordenada, según el patrón descrito en la Sección B.3.3.

1.4.2.5. Abrir la FIFO para lectura

Una vez creada la FIFO, el servidor debe abrirla para lectura usando `open()` con el flag `O_RDONLY`, tal y como se describe en la Unidad 1.

> **Nota**
>
> La llamada a `open()` con `O_RDONLY` se bloqueará hasta que algún proceso abra la FIFO para escritura. En nuestro caso, esto ocurrirá cuando el primer cliente ejecute `backup`.

El descriptor de archivo resultante se utilizará posteriormente para leer las rutas de los archivos que se deben copiar. No se debe cerrar hasta que el servidor vaya a terminar.

1.4.3. Bucle principal

Una vez completada la inicialización, el servidor entra en un bucle infinito donde:

1. Espera la llegada de una señal usando `sigwaitinfo()`, descrito en la Sección B.5.
2. Cuando recibe `SIGUSR1`, lee la ruta del archivo desde la FIFO.
3. Extrae el nombre del archivo de la ruta.
4. Construye la ruta de destino combinando el directorio de backups con el nombre del archivo.

5. Llama a `copy_file()` para copiar el archivo.
6. Muestra un mensaje indicando el resultado de la operación.
7. Vuelve al paso 1, mientras la variable `quit_requested` siga siendo falsa (ver la Sección 1.4.4).

El pseudocódigo del bucle sería el siguiente:

```
while not quit_requested:
    sigwaitinfo(signal_set, out_info)

    ruta_origen = read_path_from_fifo(fifo_fd)                    ①
    if has_error(ruta_origen):
        raise error("Error al leer ruta")

    nombre_archivo = get_filename(ruta_origen)
    ruta_destino = directorio_backups + "/" + nombre_archivo

    resultado = copy_file(ruta_origen, ruta_destino)

    if has_error(resultado):
        print("error al hacer backup de {ruta_origen}: {resultado.error}")
    else:
        print("backup completado: {ruta_origen} -> {ruta_destino}")
```

① Recuerda que la FIFO fue abierta para lectura durante la inicialización del servidor. Se supone que el descriptor de archivo se guardó en `fifo_fd`.

En este pseudocódigo se indican algunas de las funciones que se deben implementar y cuándo se deben llamar. Cuando se indica `raise error("...")`, significa que el programa terminará tal y como se describe en Sección 1.4.5, mostrando un mensaje de error por la salida de error y retornando desde `main()` con un código distinto de `EXIT_SUCCESS`.

1.4.4. Terminación del servidor

El servidor debe limpiar correctamente todos los recursos que ha creado antes de terminar, independientemente de si termina de forma normal o por un error.

Cuando el servidor termina, debe:

- Cerrar el descriptor de archivo de la FIFO
- Eliminar el archivo FIFO del sistema de archivos
- Eliminar el archivo PID
- Liberar cualquier otro recurso reservado previamente
- Terminar con el código de salida apropiado

La terminación normal se produce cuando llega una señal de terminación externa, como `SIGTERM`, `SIGHUP`, `SIGQUIT` o `SIGINT` (`Ctrl-C`). Como comentamos en la Sección 1.4.2.4, el manejador de estas señales debe establecer a `true` la variable global `quit_requested`, haciendo que el bucle principal termine y se proceda con la limpieza .

1.4.5. Manejo de errores

El servidor debe manejar las siguientes situaciones de error:

- **Durante la inicialización:**
  - ⚠ `BACKUP_WORK_DIR` no está definida.
  - ⚠ El directorio de trabajo no existe.
  - ⚠ El directorio de destino no existe o no es accesible.
  - ⚠ Ya hay otro servidor ejecutándose.
  - ⚠ No se puede crear la FIFO.
  - ⚠ No se puede escribir el archivo PID.
  - ⚠ Error al configurar las señales.
  - ⚠ Error al abrir la FIFO para lectura.
- **Durante la ejecución del bucle:**
  - ⚠ Error al leer de la FIFO.
  - La ruta leída está vacía.
  - El archivo origen no existe o no es accesible.
  - Error al copiar el archivo (sin espacio, permisos, etc.).

En todos los casos se deben **mostrar mensajes de error descriptivos por la salida de error**.

Los casos marcados con ⚠ debe provocar la terminación inmediata de la ejecución del servidor, saliendo por `main()` con un código de error distinto de `EXIT_SUCCESS`, como se explicó en la Unidad 1. Recuerda que **antes de salir es necesario liberar todos los recursos** (descriptores, memoria, terminar y esperar por procesos hijos en ejecución, etc.).

En el resto de situaciones de error, el servidor debe continuar ejecutándose y procesar las siguientes solicitudes.

1.5. Programa backup

1.5.1. Línea de comandos y funcionamiento básico

El programa `backup` acepta los siguientes argumentos de línea de comandos:

```
backup ARCHIVO
```

Donde:

- `ARCHIVO`: Ruta del archivo del que se quiere hacer una copia de seguridad.

El cliente realiza las siguientes acciones:

1. Valida que la variable `BACKUP_WORK_DIR` esté definida.
2. Verifica que el archivo existe y es un archivo regular.
3. Lee el PID del servidor desde el archivo `backup-server.pid`.
4. Verifica que el servidor está ejecutándose.
5. Bloquea la señal `SIGPIPE`.
6. Abre la FIFO para escritura.
7. Convierte la ruta del archivo a ruta absoluta.
8. Escribe la ruta en la FIFO.
9. Envía la señal `SIGUSR1` al servidor.
10. Cierra la FIFO.
11. Muestra un mensaje de confirmación.

Este sería un ejemplo de la ejecucion normal del programa:

```
$ backup /home/user/important.txt
backup: solicitud enviada para /home/user/important.txt
```

```
$ backup ./document.pdf
backup: solicitud enviada para /home/user/Desktop/document.pdf
```

1.5.2. Flujo de operación

1.5.2.1. Validar el archivo

El programa debe verificar que:

- El archivo indicado existe.
- Es un archivo regular (no un directorio, enlace simbólico, etc.).

Para ello, se puede usar `stat()` y la macro `S_ISREG()`, como se vio en la Unidad 1.

1.5.2.2. Leer el PID del servidor

El cliente debe leer el contenido del archivo `backup-server.pid` para obtener el PID del servidor. Como el archivo contiene el PID como cadena de texto, se debe convertir a un número entero para usarlo como `pid_t`.

Después de leer el PID, el cliente debe verificar que el proceso existe usando `kill(pid, 0)`. Si `kill()` retorna `-1` con `errno` igual a `ESRCH`, significa que el proceso no existe.

1.5.2.3. Bloquear la señal SIGPIPE

Aunque hemos comprobado que el servidor está ejecutándose, un error podría hacer que el servidor termine mientras el cliente envía la ruta por la FIFO. En ese caso, la llamada a `write()` fallaría con `EPIPE` y el cliente recibiría la señal `SIGPIPE`, terminando inesperadamente.

Para evitar este problema, el cliente debe bloquear `SIGPIPE` antes de abrir la FIFO para escritura, usando `sigprocmask()` tal y como se explica en la Sección B.4.

1.5.2.4. Abrir la FIFO para escritura

El cliente debe abrir la FIFO usando `open()` con el flag `O_WRONLY`, tal y como se vio en la Unidad 1.

> **Nota**
>
> La llamada a `open()` con `O_WRONLY` se bloqueará hasta que el servidor tenga la FIFO abierta para lectura. Esto garantiza que el servidor está listo para recibir solicitudes.

1.5.2.5. Escribir la ruta en la FIFO

Antes de escribir la ruta en la FIFO, el cliente debe convertirla a una ruta absoluta si es relativa. Para ello, implementaremos la función `get_absolute_path()`, que utiliza `realpath()` para realizar la conversión.

La ruta absoluta se escribe en la FIFO seguida de un carácter de salto de línea (`\n`) usando `write()`, como se explicó en la Unidad 1.

1.5.2.6. Enviar la señal al servidor

Una vez escrita la ruta en la FIFO, el cliente envía la señal `SIGUSR1` al servidor usando `kill()`, como se describe en la Sección B.6.

1.5.3. Manejo de errores

El cliente debe manejar las siguientes situaciones de error:

- `BACKUP_WORK_DIR` no está definida.
- El archivo indicado no existe.
- El archivo no es un archivo regular.
- El archivo `backup-server.pid` no existe (servidor no ejecutándose).
- Error al leer el archivo PID.
- El contenido del archivo PID no es un número válido.
- El servidor no está ejecutándose (proceso no existe).
- Error al abrir la FIFO para escritura.

- Error al convertir la ruta a absoluta.
- Error al escribir en la FIFO.
- Error al enviar la señal.

En todos los casos, el programa debe mostrar un mensaje descriptivo por la salida de error y terminar con un código de salida distinto de `EXIT_SUCCESS`.

1.6. Implementación

A continuación se describen las funciones principales que deben implementarse para ambos programas.

1.6.1. Funciones comunes

Estas funciones son útiles para ambos programas:

1.6.1.1. Gestión de rutas

- ```
  std::string get_work_dir_path();
  ```

  Lee la variable de entorno `BACKUP_WORK_DIR` y retorna su valor. Si la variable no está definida, retorna una cadena vacía.

- ```
  std::string get_fifo_path();
  ```

  Retorna la ruta completa de la FIFO combinando el directorio de trabajo con el nombre `backup.fifo`.

- ```
  std::string get_pid_file_path();
  ```

  Retorna la ruta completa del archivo PID combinando el directorio de trabajo con el nombre `backup-server.pid`.

- ```
  std::expected<std::string, std::system_error> get_absolute_path(
      const std::string& path);
  ```

  Convierte una ruta (relativa o absoluta) a una ruta absoluta usando `realpath()`. Retorna la ruta absoluta o un error si la ruta no existe o no es accesible.

  > **!** Importante
  >
  > La forma recomendada de utilizar `realpath()` es pasando `nullptr` como segundo argumento, para que la función reserve internamente el buffer necesario. En ese caso, debemos llamar a `free()` con el puntero devuelto para liberar el buffer

> reservado por `realpath()`, tras copiar su contenido al `std::string` que se va a retornar.

El uso de `std::expected` fue explicado en la Unidad 1.

1.6.1.2. Utilidades de archivo

- ```
  bool file_exists(const std::string& path);
  ```

  Verifica si un archivo existe usando `access()` con el flag `F_OK`.

- ```
  bool is_regular_file(const std::string& path);
  ```

  Verifica si un archivo es un archivo regular usando `stat()` y la macro `S_ISREG()`, como se explicó anteriormente en la Unidad 1.

- ```
  bool is_directory(const std::string& path);
  ```

  Verifica si un archivo es un directorio usando `stat()` y la macro `S_ISDIR()`. Esta función ya fue implementada en la Unidad 1.

- ```
  std::string get_current_dir();
  ```

  Obtiene el directorio actual de trabajo usando `getcwd()`.

- ```
  std::string get_filename(const std::string& path);
  ```

  Extrae el nombre del archivo de una ruta completa. Esta función ya fue implementada anteriormente.

1.6.2. Funciones de backup-server

- ```
  std::expected<pid_t, std::system_error> read_server_pid(
      const std::string& pid_file_path);
  ```

  Lee el PID del servidor desde el archivo indicado, pero recuerda que solo puedes usar las llamadas al sistema `open()`, `read()` y `close()`. Retorna el PID o un error si no se puede leer el archivo o el contenido no es válido.

- ```
  bool is_server_running(pid_t pid);
  ```

  Verifica si el proceso con el PID indicado está ejecutándose usando `kill(pid, 0)`. Retorna `true` si el proceso existe, `false` en caso contrario.
  ```

- ```
  std::expected<void, std::system_error> create_fifo(
      const std::string& fifo_path);
  ```

  Crea una FIFO en la ruta indicada con permisos `0666`. Si la FIFO ya existe, la elimina primero con `unlink()` y luego la crea. El uso de `mkfifo()` se describe en la Sección A.1.

- ```
  std::expected<void, std::system_error> write_pid_file(
      const std::string& pid_file_path);
  ```

  Escribe el PID del proceso actual en el archivo indicado. Crea el archivo con permisos `0644` usando `open()` con los flags `O_WRONLY | O_CREAT | O_TRUNC` y escribe el PID como cadena de texto seguida de un salto de línea, usando `write()`.

- ```
  std::expected<void, std::system_error> setup_signal_handler();
  ```

  Configura el manejo de señales:

  Para el **manejo síncrono con `sigwaitinfo()`**:

  1. Crea un conjunto de señales vacío con `sigemptyset()`.
  2. Añade `SIGUSR1` al conjunto con `sigaddset()`.
  3. Bloquea las señales del conjunto con `sigprocmask()` usando `SIG_BLOCK`.

  - Estas funciones se describen en la Sección B.4.

  Para el **manejo asíncrono de terminación**:

  - Instala un manejador para las señales `SIGTERM`, `SIGHUP`, `SIGQUIT` y `SIGINT` usando `sigaction()`. El uso de `sigaction()` se describe en la Sección B.3.
  - El código del manejador debe usar `write()` para mostrar un mensaje indicando que se ha recibido la señal. Por ejemplo:

    ```
    backup-server: señal de terminación recibida, cerrando...
    ```

    y luego establecer la variable global `quit_requested` a `true` para que el bucle principal termine (ver la Sección B.3.3). Esta variable debe definirse como `std::atomic<bool> quit_requested{false};` para evitar problemas de concurrencia.

- ```
  std::expected<std::string, std::system_error> read_path_from_fifo(
      int fifo_fd);
  ```

  Lee una línea completa (hasta encontrar `\n`) del descriptor de archivo de la FIFO. Retorna la ruta leída (sin el `\n` final) o un error si no se puede leer.

  Para implementar esta función, se debe leer carácter a carácter usando `read()` hasta encontrar `\n` o hasta alcanzar un máximo de `PATH_MAX` caracteres.

> **Nota**
>
> `PATH_MAX` es una constante definida en `<limits.h>` que indica la longitud máxima de cualquier ruta en el sistema.

- ```
  void run_server(int fifo_fd, const std::string& backup_dir);
  ```

  Implementa el bucle principal del servidor:

  1. Crea un conjunto de señales con `SIGUSR1`.
  2. En un bucle infinito:
     - Llama a `sigwaitinfo()` para esperar la señal.
     - Lee la ruta desde la FIFO.
     - Extrae el nombre del archivo.
     - Construye la ruta de destino.
     - Llama a `copy_file()`.
     - Muestra el resultado.

  El uso de `sigwaitinfo()` se describe en la Sección B.5.

1.6.3. Funciones de backup

- ```
  bool check_args(int argc, char* argv[]);
  ```

  Verifica que se haya indicado exactamente un argumento (además del nombre del programa).

- ```
  std::expected<void, std::system_error> check_work_dir_exists(
      const std::string& work_dir);
  ```

  Verifica que el directorio de trabajo indicado existe y es accesible. Usa `stat()` para verificar que la ruta existe y es un directorio.

- ```
  std::expected<int, std::system_error> open_fifo_write(
      const std::string& fifo_path);
  ```

  Abre la FIFO para escritura usando `open()` con el flag `O_WRONLY`. Retorna el descriptor de archivo o un error.

- ```
  std::expected<void, std::system_error> write_path_to_fifo(
      int fifo_fd, const std::string& file_path);
  ```

  Escribe la ruta del archivo en la FIFO seguida de un salto de línea. La escritura debe hacerse usando `write()`, asegurándose de escribir todos los bytes.
  ```

1.7. Comprobación

1.7.1. Preparación del entorno

Antes de probar el sistema, debemos configurar el entorno. Primero, definimos la variable de entorno `BACKUP_WORK_DIR` y creamos el directorio de trabajo en el que se guardarán los archivos copiados:

```
$ export BACKUP_WORK_DIR=/tmp/backup-work
$ mkdir -p /tmp/backup-work
$ mkdir -p /tmp/backups
```

Luego, creamos algunos archivos de prueba:

```
$ echo "Contenido importante" > /tmp/test1.txt
$ echo "Otro archivo" > /tmp/test2.txt
$ dd if=/dev/urandom of=/tmp/binary.dat bs=1M count=10 iflag=fullblock
```

1.7.2. Ejecución del servidor

En una terminal, iniciamos el servidor:

```
$ backup-server /tmp/backups
backup-server: esperando solicitudes de backup en /tmp/backups
```

El servidor quedará bloqueado esperando señales. Dejamos esta terminal abierta para ver los mensajes del servidor.

1.7.3. Solicitar backups

En otra terminal (con la misma configuración de `BACKUP_WORK_DIR`), ejecutamos el cliente:

```
$ backup /tmp/test1.txt
backup: solicitud enviada para /tmp/test1.txt
```

En la terminal del servidor deberíamos ver:

```
backup-server: backup completado: /tmp/test1.txt -> /tmp/backups/test1.txt
```

Luego podemos solicitar más backups:

```
$ backup /tmp/test2.txt
backup: solicitud enviada para /tmp/test2.txt
$ backup /tmp/binary.dat
backup: solicitud enviada para /tmp/binary.dat
```

1.7.4. Verificar los backups

Comprobamos que los archivos se han copiado correctamente:

```
$ ls -l /tmp/backups
total 8
-rw-r--r-- 1 user user 20 nov  3 10:30 test1.txt
-rw-r--r-- 1 user user 13 nov  3 10:31 test2.txt
-rw-r--r-- 1 user user 10M nov  3 10:32 binary.dat

$ cmp /tmp/test1.txt /tmp/backups/test1.txt
$ cmp /tmp/test2.txt /tmp/backups/test2.txt
$ cmp /tmp/binary.dat /tmp/backups/binary.dat
```

Si `cmp` no muestra nada y retorna 0, los archivos son idénticos.

1.7.5. Terminar el servidor

Para terminar el servidor, podemos presionar `Ctrl-C` en su terminal o, desde otra terminal, ejecutar el comando `kill` usando su PID:

```
$ kill $(cat /tmp/backup-work/backup-server.pid)
```

En ambos casos deberíamos ver en la terminal del servidor como este muestra el siguiente mensaje y termina correctamente:

```
backup-server: señal de terminación recibida, cerrando...
```

Esto debe ocurrir inmediatamente, incluso si el servidor está ocupado copiando un archivo de gran tamaño.

1.7.6. Casos de prueba adicionales

Es importante probar diferentes escenarios:

- **Intentar ejecutar dos servidores simultáneamente**:

```
$ backup-server /tmp/backups &
[1] 12345
backup-server: esperando solicitudes de backup en /tmp/backups

$ backup-server /tmp/backups
backup-server: error: ya hay un servidor ejecutándose (PID 12345)
```

- **Intentar hacer backup sin que el servidor esté ejecutándose**:

```
$ backup /tmp/test1.txt
backup: error: el servidor no está ejecutándose
```

- **Intentar hacer backup de un archivo que no existe**:

```
$ backup /tmp/noexiste.txt
backup: error: el archivo /tmp/noexiste.txt no existe
```

- **Intentar hacer backup de un directorio**:

```
$ backup /tmp
backup: error: /tmp no es un archivo regular
```

- **No definir la variable de entorno**:

```
$ unset BACKUP_WORK_DIR
$ backup /tmp/test1.txt
backup: error: BACKUP_WORK_DIR no está definida
```

A. Tuberías con nombre (FIFO)

Las tuberías son un mecanismo de comunicación entre procesos que permite el intercambio de datos utilizando la misma interfaz que se emplea para manipular archivos. Conceptualmente, cada tubería tiene dos extremos: uno para enviar datos mediante `write()` y otro para recibirlos mediante `read()`.

Existen dos tipos de tuberías:

- **Tuberías sin nombre** o **anónimas**: Se crean mediante `pipe()` y se utilizan para la comunicación entre procesos relacionados, como un proceso padre y sus hijos.
- **Tuberías con nombre** o **FIFO**: Se crean en el sistema de archivos mediante `mkfifo()` y se utilizan para la comunicación entre procesos no relacionados que se ejecutan de forma independiente.

A.1. Crear una tubería con nombre

Para crear una tubería con nombre se utiliza la función `mkfifo()`:

```
int mkfifo(const char *pathname, mode_t mode);
```

Donde:

`pathname` Ruta donde se creará el archivo especial tipo FIFO que representa la tubería.
`mode` Permisos de acceso del archivo FIFO (por ejemplo, `0666` para lectura y escritura para todos).

La función retorna:

- `0` si la tubería se crea correctamente.
- `-1` si ocurre un error, estableciendo `errno` con el código de error correspondiente.

Los códigos de error más comunes son:

- `EEXIST`: Ya existe un archivo con ese nombre.
- `EACCES`: No hay permisos para crear el archivo en el directorio indicado.
- `ENOENT`: Algún componente de la ruta no existe.

La función se puede utilizar de la siguiente manera:

```
int result = mkfifo("/tmp/myfifo", 0666);
if (result == -1)
{
    // Error al crear la FIFO...
}
```

> **Nota**
>
> Si la FIFO ya existe, `mkfifo()` fallará con el valor `EEXIST` en `errno`. En ese caso, se puede eliminar primero con `unlink()` y luego crearla de nuevo.

A.2. Abrir una tubería con nombre

Una vez creada la FIFO, se puede abrir para lectura o escritura usando `open()`, tal y como se explicó en la Unidad 1 al hablar de cómo trabajar con archivos. Para lectura se utiliza la bandera `O_RDONLY`:

```
int fd = open("/tmp/myfifo", O_RDONLY);
if (fd == -1)
{
    // Error al abrir la FIFO para lectura...
}
```

Mientras que para escritura se utiliza la bandera `O_WRONLY`:

```
int fd = open("/tmp/myfifo", O_WRONLY);
if (fd == -1)
{
    // Error al abrir la FIFO para escritura...
}
```

A.2.1. Comportamiento de bloqueo al abrir

Por defecto, la apertura de una FIFO es una operación bloqueante:

- `open()` con `O_RDONLY` se bloquea hasta que otro proceso abre la FIFO para escritura.
- `open()` con `O_WRONLY` se bloquea hasta que otro proceso abre la FIFO para lectura.

Este comportamiento garantiza que ambos extremos de la tubería están conectados antes de que se pueda realizar cualquier operación de E/S.

A.3. Leer y escribir en tuberías

Para leer y escribir en las tuberías se utilizan las funciones `read()` y `write()`, exactamente igual que con archivos regulares, tal y como se explicó en la Unidad 1.

Al igual que cuando se trabaja con archivos, es importante recordar que las funciones `read()` y `write()` pueden leer o escribir menos bytes de los solicitados, por lo que es recomendable utilizar bucles para asegurarse de que se procesan todos los datos necesarios.

A.3.1. Comportamiento de lectura bloqueante

La llamada a `read()` en una FIFO se bloquea si no hay datos disponibles. Cuando todos los descriptores de escritura de la FIFO se cierran y se han leído todos los datos, `read()` retorna `0`, indicando fin de archivo (EOF).

A.3.2. Escribir en una tubería rota

Si el proceso que lee de la tubería termina o cierra su extremo de lectura antes de que el escritor finalice, las llamadas a `write()` en el extremo de escritura fallarán con `EPIPE` y el proceso escritor recibirá la señal `SIGPIPE`, que terminará el proceso si la señal no es manejada o bloqueada (ver el Apéndice B).

Este comportamiento es útil en comandos con tuberías en la *shell*, donde un proceso puede escribir datos a otro proceso que los consume y si el proceso consumidor termina antes, el proceso productor también debe finalizar. Pero en nuestros programas, puede ser necesario manejar esta situación para evitar que el proceso termine inesperadamente.

A.4. Cerrar descriptores de FIFO

Los descriptores de archivo de las FIFO deben cerrarse cuando ya no se necesiten, usando `close()`:

```
close(fd);
```

Es importante cerrar los descriptores para:

- Liberar los recursos del sistema asociados.
- Permitir que las operaciones de lectura bloqueadas en el otro extremo detecten el fin de archivo.

A.5. Eliminar una tubería con nombre

Las tuberías con nombre se crean como archivos especiales en el sistema de archivos y persisten hasta que se eliminan explícitamente.

Para eliminar una FIFO se utiliza `unlink()`:

```
int unlink(const char *pathname);
```

La función retorna:

- `0` si el archivo se elimina correctamente.
- `-1` si ocurre un error, estableciendo `errno` con el código de error.

Por ejemplo, así borraríamos una FIFO llamada `/tmp/myfifo`:

```
int result = unlink("/tmp/myfifo");
if (result == -1)
{
    // Error al eliminar la FIFO...
}
```

> **Eliminación de FIFOs con descriptores abiertos**
>
> Si hay procesos con la FIFO abierta cuando se llama a `unlink()`, el archivo desaparece del sistema de archivos, pero los descriptores abiertos siguen siendo válidos, por lo que la comunicación puede continuar hasta que los procesos cierren sus descriptores.

B. Señales POSIX

Las señales son un mecanismo de comunicación entre procesos que permite notificar a un proceso que ha ocurrido un evento específico. En los sistemas POSIX, las señales se utilizan tanto para que el sistema operativo notifique a los programas ciertos errores y sucesos críticos, como para que los procesos se comuniquen entre sí.

B.1. Características de las señales

Las señales tienen las siguientes características:

- **Comunicación directa**: Se envían usando el identificador del proceso de destino (PID).
- **Tamaño y formato fijos**: Las señales solo portan la información de que ha ocurrido un evento, identificado por un número de señal. No pueden transportar datos adicionales.
- **Asíncronas por defecto**: Pueden interrumpir la ejecución del proceso en cualquier momento.

Cada señal tiene un efecto particular por defecto en el proceso que la recibe. Por ejemplo, muchas señales, por defecto, terminan el proceso que las recibe. Sin embargo, los procesos pueden cambiar el comportamiento ante una señal mediante:

- **Manejadores de señal**: Funciones que se ejecutan cuando llega la señal.
- **Bloqueo de señales**: Impedir temporalmente que ciertas señales se entreguen al proceso.
- **Espera síncrona**: Usar `sigwait()`, `sigwaitinfo()` o `sigtimedwait()` para esperar señales de forma controlada.

B.2. Señales comunes

Algunas de las señales más comunes del estándar POSIX son:

`SIGALRM` Enviada cuando expira un temporizador configurado con `alarm()` o `setitimer()`.
`SIGHUP` Enviada cuando termina la sesión de terminal asociada al proceso. En servicios del sistema se usa convencionalmente para indicarles que deben reiniciarse o recargar su configuración.
`SIGKILL` Termina inmediatamente el proceso. Esta señal no puede ser bloqueada ni ser manejada.
`SIGINT` Enviada cuando el usuario pulsa Ctrl+C en la terminal. Por defecto termina el proceso.
`SIGPIPE` Enviada a un proceso cuando intenta escribir en una tubería o *socket* cuyo extremo de lectura ha sido cerrado. Por defecto termina el proceso.
`SIGSEGV` Enviada cuando un proceso intenta acceder a memoria no permitida (violación de segmento).
`SIGTERM` Solicita al proceso que termine de forma ordenada. Es la señal por defecto enviada por el comando `kill`. El sistema operativo envía esta señal a todos los procesos cuando se apaga el sistema.
`SIGUSR1` y `SIGUSR2` Señales reservadas para uso del usuario. No tienen un significado predefinido, por lo que los programadores pueden usarlas con el propósito que deseen.

Se puede consultar la lista completa de señales soportadas en Linux en la página del manual "signal(7)".

B.3. Manejo de señales

Por defecto, cuando una señal llega a un proceso, este interrumpe inmediatamente el código que está ejecutando para ejecutar la acción por defecto de la señal (que frecuentemente es terminar el proceso). Sin embargo, es posible configurar un **manejador de señal**: una función del programa de nuestro programa que será invocada por el sistema operativo cuando llegue una señal determinada.

B.3.1. Configurar manejadores con sigaction()

Para configurar un manejador de señal se utiliza la función `sigaction()`:

```
int sigaction(int signum, const struct sigaction *act,
    struct sigaction *oldact);
```

Donde:

`signum` Número de la señal para la que se configura el manejador.
`act` Puntero a una estructura `sigaction` que describe cómo tratar la señal.
`oldact` Puntero donde se guardará la configuración anterior. Si es `NULL`, se ignora.

La función retorna:

- `0` si tiene éxito.
- `-1` si ocurre un error, estableciendo `errno`.

La estructura `sigaction` tiene los siguientes campos principales:

`sa_handler` Puntero a la función de nuestro programa con la que queremos manejar la señal. Puede ser:

- Un puntero a una función con formato `void funct(int)`
- `SIG_DFL`: Para restaurar el comportamiento por defecto
- `SIG_IGN`: Para ignorar la señal

`sa_mask` Conjunto de señales a bloquear mientras se ejecuta la función manejadora (ver la Sección B.4.1). Esto evita que otras señales interrumpan la función manejadora mientras se está ejecutando.

`sa_flags` Opciones de configuración. Las más comunes son:

- `SA_RESTART`: Si la señal interrumpe una llamada al sistema, reanudarla automáticamente después del manejador.
- `SA_RESETHAND`: Restaurar el comportamiento por defecto después de ejecutar el manejador una vez.

Por ejemplo, para configurar un manejador para `SIGTERM` y `SIGINT`:

```
void signal_handler(int signum)                                          ①
{
    switch (signum)
    {
        case SIGTERM:
            write(STDOUT_FILENO, "Recibida señal SIGTERM\n", 24);        ②
            break;

        case SIGINT:
            write(STDOUT_FILENO, "Recibida señal SIGINT\n", 24);
            break;

        default:
            write(STDOUT_FILENO, "Recibida señal desconocida\n", 29);
            break;
    }
}

int main()
{
```

```
    struct sigaction sa;
    sa.sa_handler = signal_handler;                                    ③
    sigemptyset(&sa.sa_mask);                                          ④
    sa.sa_flags = 0;

    if (sigaction(SIGTERM, &sa, NULL) == -1)                           ⑤
    {
        // Error al configurar el manejador...
    }

    if (sigaction(SIGINT, &sa, NULL) == -1)
    {
        // Error al configurar el manejador...
    }

    // Código principal del programa...
    // En este ejemplo, simplemente se hace una espera de 5 minutos
    sleep(300);                                                        ⑥

    return EXIT_SUCCESS;
}
```

① Definir la función manejadora. Debe tener la firma `void handler(int signum)`.
② En el manejador, realizar las acciones necesarias al recibir la señal. Aquí se usa `write()` para mostrar un mensaje –en lugar de `std::cout`– porque las funciones de E/S estándar no son seguras para utilizarlas en manejadores de señales (ver la Sección B.3.2).
③ Configurar la estructura `sigaction` con el manejador y las opciones deseadas. En este ejemplo, se indica que la función que atenderá a las señales es `signal_handler()`.
④ Inicializar la máscara de señales (`sa_mask`) como vacía usando `sigemptyset()` (ver la Sección B.4.1). Tampoco se indican *flags* (`sa_flags`) adicionales.
⑤ Registrar el manejador para la señal `SIGTERM` y `SIGINT`.
⑥ El programa principal continúa su ejecución normal. Aquí simplemente se simula una espera larga con `sleep()` para poder enviar señales al proceso desde otra terminal. Durante este tiempo, si llega `SIGTERM` o `SIGINT`, se ejecutará la función `signal_handler()`.

B.3.2. Precauciones con manejadores de señales

Los manejadores de señal se ejecutan de forma asíncrona, interrumpiendo el flujo normal del programa. Esto puede causar problemas si el manejador:

- Modifica variables compartidas sin utilizar mecanismos de sincronización adecuados.
- Llama a funciones no reentrantes.
- Realiza operaciones que no son seguras en contextos de señales.

Por esta razón, se recomienda que los manejadores sean lo más simples posible y solo llamen a funciones marcadas como async-signal-safe. Igualmente, es aconsejable utilizar variables atómicas –como `std::atomic`– o mecanismos de sincronización adecuados para compartir datos entre el manejador y el resto del programa de forma segura.

Una alternativa más segura es usar `sigwait()`, `sigwaitinfo()` o `sigtimedwait()`, para manejar señales de forma síncrona, como se explica en la Sección B.5.

B.3.3. Patrón típico de terminación segura

Generalmente, queremos que los generadores de señales disparen ciertas acciones en el programa –como la liberacion de recursos y terminar de forma ordenada– pero no podemos hacerlo directamente desde el manejador de señal por las limitaciones mencionadas anteriormente. En su lugar, un patrón común es que el manejador de señal simplemente establezca variables atómicas indicando que se ha solicitado la terminación del programa o cualquier otra acción que nos interese. El programa debe verificar estas variables periódicamente y realizar las acciones necesarias de forma segura cuando detecte que se ha solicitado alguna.

Por ejemplo, aquí tenemos un ejemplo de este patrón para manejar señales de terminación y salir de forma segura del programa:

```
std::atomic<bool> quit_requested{false};                                    ①

void signal_handler(int signum)
{
    char msg[] = "Señal de terminación recibida, cerrando...\n";
    write(STDOUT_FILENO, msg, sizeof(msg) - 1);

    quit_requested = true;                                                  ②
}

int main()
{
    struct sigaction sa = {                                                 ③
        .sa_handler = signal_handler,
    };

    sigaction(SIGTERM, &sa, NULL);
    sigaction(SIGINT, &sa, NULL);

    while (!quit_requested)                                                 ④
    {
        // Código principal del programa...
        // Aquí se puede hacer trabajo útil, dormir, etc.
    }
```

```
    return EXIT_SUCCESS;
}
```

① Definir una variable atómica `quit_requested` que indique si se ha solicitado la terminación del programa.
② El manejador de señal establecerá la variable `quit_requested` a `true` cuando llegue una señal de terminación.
③ Configurar el manejador de señal para `SIGTERM` y `SIGINT`.
④ En el bucle principal del programa, se verifica periódicamente la variable `quit_requested`. Cuando se detecta que es `true`, se sale del bucle y se procede a la limpieza y terminación ordenada del programa.

B.3.4. Señales y llamadas al sistema

Una cuestión común al trabajar con señales es cómo afectan a las llamadas al sistema que el proceso pueda estar ejecutando cuando llega una señal. Especialmente porque si el programa está bloqueado en una llamada al sistema (como `read()`, `write()`, `sleep()`, etc.)y llega una señal, puede que queramos que la llamada se interrumpa para que el programa pueda atender la señal. Por ejemplo, en el programa anterior, si el proceso está bloqueado en un `read()` y llega `SIGINT`, seguramente queremos que `read()` sea interrumpido para que el bucle principal pueda comprobar `quit_requested` y salir.

Por defecto, muchas llamadas al sistema se interrumpen cuando llega una señal. Solo en los casos en los que no queremos que esto ocurra, debemos usar el *flag* `SA_RESTART` al configurar el manejador de señal con `sigaction()`.

Las llamadas al sistema que son interrumpidas por una señal, terminan con error y establecen `errno` a `EINTR`. Por tanto, no debemos considerar esto como un error fatal, sino simplemente como una situación que debemos manejar adecuadamente, según las necesidades de nuestro programa. Si `errno` es `EINTR`, puede interesarnos ignorarlo y reintentar la llamada al sistema, o comprobar las variables establecidas por nuestros manejadores de señal –como `quit_requested`– para decidir que tenemos que hacer a continuación.

B.4. Bloquear señales

Un proceso puede bloquear temporalmente la entrega de ciertas señales. Cuando una señal está bloqueada, queda pendiente hasta que se desbloquea, momento en el que se entrega al proceso.

Para bloquear y desbloquear señales se utilizan la función `sigprocmask()` que necesita un conjunto de señales (`sigset_t`) con las señales a bloquear o desbloquear.

B.4.1. Conjuntos de señales

Un conjunto de señales es una estructura de datos que representa un grupo de señales. Se manipula mediante las siguientes funciones:

- ```
  int sigemptyset(sigset_t *set);
  ```

  Inicializa un conjunto de señales vacío (sin ninguna señal). Retorna `0` en éxito, `-1` en error.

- ```
  int sigfillset(sigset_t *set);
  ```

  Inicializa un conjunto con todas las señales. Retorna `0` en éxito, `-1` en error.

- ```
  int sigaddset(sigset_t *set, int signum);
  ```

  Añade la señal `signum` al conjunto `set`. Retorna `0` en éxito, `-1` en error.

- ```
  int sigdelset(sigset_t *set, int signum);
  ```

  Elimina la señal `signum` del conjunto `set`. Retorna `0` en éxito, `-1` en error.

- ```
  int sigismember(const sigset_t *set, int signum);
  ```

  Verifica si la señal `signum` está en el conjunto `set`. Retorna `1` si está, `0` si no está, `-1` en error.

Por ejemplo, para crear un conjunto de señales que contenga solo `SIGTERM`:

```
sigset_t signal_set;

sigemptyset(&signal_set);                                    ①
sigaddset(&signal_set, SIGTERM);                             ②
```

① Inicializar `signal_set` como un conjunto vacío (sin ninguna señal).
② Añadir la señal `SIGTERM` al conjunto `signal_set`.

B.4.2. Bloquear y desbloquear con sigprocmask()

La función `sigprocmask()` permite cambiar la máscara de señales bloqueadas del proceso:

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

Donde:

`how` Especifica cómo modificar la máscara de señales bloqueadas:
```

- `SIG_BLOCK`: Añade las señales de `set` a las ya bloqueadas.
- `SIG_UNBLOCK`: Elimina las señales de `set` de las bloqueadas.
- `SIG_SETMASK`: Reemplaza la máscara de bloqueadas con `set`.

`set` Puntero al conjunto de señales a usar según `how`. Si es `NULL`, no se modifica la máscara (útil solo para consultar).
`oldset` Puntero donde se guardará la máscara anterior. Si es `NULL`, no se guarda.

La función retorna:

- `0` si tiene éxito.
- `-1` si ocurre un error, estableciendo `errno`.

Por ejemplo, para bloquear las señales `SIGTERM` y `SIGINT`:

```
sigset_t signal_set;

sigemptyset(&signal_set);                                          ①
sigaddset(&signal_set, SIGTERM);                                   ②
sigaddset(&signal_set, SIGINT);

if (sigprocmask(SIG_BLOCK, &signal_set, NULL) == -1)               ③
{
    // Error al bloquear señales...
}

// Codigo del proceso que se ejecuta con SIGTERM y SIGINT bloqueadas...
```

① Inicializar `signal_set` como un conjunto vacío.
② Añadir las señales `SIGTERM` y `SIGINT` al conjunto `signal_set`.
③ Bloquear las señales en `signal_set` (es decir, `SIGTERM` y `SIGINT`).

Mientras que se pueden desbloquear todas las señales bloqueadas con:

```
sigset_t signal_set;

sigfillset(&signal_set);                                           ①

if (sigprocmask(SIG_UNBLOCK, &signal_set, NULL) == -1)             ②
{
   // Error al desbloquear señales...
}
```

① Inicializar `signal_set` con todas las señales.
② Desbloquear todas las señales en `signal_set`.

> **!** Importante
>
> Bloquear señales no las ignora ni las descarta. Las señales bloqueadas quedan pendientes y se entregarán cuando se desbloqueen. Sin embargo, debemos tener en cuenta que las señales estándares pendientes no se acumulan. Si la misma señal llega múltiples veces mientras está bloqueada, solo se entregará una vez cuando sea desbloqueada.

B.5. Esperar señales de forma síncrona

Como hemos comentado, las señales se entregan de forma asíncrona por defecto. Es decir, pueden interrumpir lo que esté ejecutando el proceso en cualquier momento. Sin embargo, es posible esperar señales de forma síncrona usando funciones como `sigwait()`, `sigwaitinfo()` o `sigtimedwait()`:

```
int sigwait(const sigset_t *set, int *sig);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
  const struct timespec *timeout);
```

Esto quiere decir, que en lugar de dejar que las señales interrumpan el flujo normal del programa, el proceso puede decidir cuándo quiere esperar y procesar las señales. Para ello, se utilizan estas funciones para consultar las señales pendientes y atenderlas de forma controlada, dentro del flujo normal de ejecución del programa.

Si al llamar alguna de estas funciones no hay ninguna señal del conjunto `set` pendiente, la función bloquea la ejecución del proceso hasta que llegue una señal incluida en dicho conjunto.

De las tres funciones mencionadas, vamos a centrarnos en `sigwaitinfo()`. Sus argumentos son:

`set` Conjunto de señales por las que se quieren esperar. Debe contener las señales que queremos recibir.

`info` Puntero a una estructura `siginfo_t` donde se devolverá información detallada sobre la señal recibida. Esto permite al proceso saber detalles como qué señal ha llegado, para poder actuar en consecuencia, o qué proceso la envió:

```
typedef struct siginfo {
    int      si_signo;    // Número de señal
    int      si_code;     // Código de la señal
    pid_t    si_pid;      // PID del proceso que envió la señal
    uid_t    si_uid;      // UID del proceso que envió la señal
    // Otros campos específicos de la señal...
} siginfo_t;
```

La función retorna:

- En caso de éxito, retorna el número de la señal recibida (un valor positivo).
- `-1`, si ocurre un error, estableciendo `errno` con el código de error correspondiente.

> **!** Importante
>
> Para usar `sigwait()`, `sigwaitinfo()` o `sigtimedwait()` correctamente, las señales por las que se quiere esperar **deben estar bloqueadas** antes de llamar a la función. Si no están bloqueadas, las señales se entregarían de forma asíncrona en lugar de ser capturadas por estas funciones.

El siguiente ejemplo muestra cómo usar `sigwaitinfo()` para esperar las señales `SIGTERM` y `SIGINT`:

```
sigset_t signal_set;
siginfo_t signal_info;

sigemptyset(&signal_set);                                          ①
sigaddset(&signal_set, SIGTERM);
sigaddset(&signal_set, SIGINT);

// 2. Bloquear las señales
if (sigprocmask(SIG_BLOCK, &signal_set, NULL) == -1)               ②
{
    // Error al bloquear señales...
}

// Aquí el proceso puede realizar otras tareas...

int result = sigwaitinfo(&signal_set, &signal_info);               ③
if (result == -1)
{
    // Error esperado señales...
}

if (signal_info.si_signo == SIGINT)                                ④
{
    std::println("Recibida señal SIGINT ({}) por el proceso {}",
      signal_info.si_signo, signal_info.si_pid);
}
```

① Crear un conjunto de señales que incluya las señales que se desean esperar. En este ejemplo: `SIGTERM` y `SIGINT`.
② Bloquear las señales del conjunto para que `sigwaitinfo()` pueda capturarlas.
③ Llamar a `sigwaitinfo()` para esperar una señal del conjunto bloqueado.

④ Comprobar qué señal se ha recibido y actuar en consecuencia. En este caso, se comprueba si es `SIGINT` y se muestra un mensaje con su número y el PID del proceso que la envió.

> **Nota**
>
> Usar las funcioones `sigwait` simplifica el código al evitar la complejidad de los manejadores de señales –donde habría que preocuparse por funciones reentrantes y condiciones de carrera– al permitir manejar las señales dentro del flujo normal del programa.

B.6. Enviar señales

Para enviar una señal a un proceso se utiliza la función `kill()`:

```
int kill(pid_t pid, int sig);
```

Donde:

`pid` Identificador del proceso (PID) al que enviar la señal. Valores especiales:

- `> 0`: Envía la señal al proceso con ese PID.
- `0`: Envía la señal a todos los procesos del mismo grupo que el proceso actual.
- `-1`: Envía la señal a todos los procesos (requiere privilegios).

`sig` Número de la señal a enviar. El valor especial `0` no envía ninguna señal, pero verifica si el proceso existe y si tenemos permisos para enviarle señales.

La función retorna:

- `0` si tiene éxito.
- `-1` si ocurre un error, estableciendo `errno`.

Los errores más comunes son:

- `ESRCH`: No existe ningún proceso con ese PID.
- `EPERM`: No hay permisos para enviar señales a ese proceso.

Por ejemplo, para enviar la señal `SIGHUP` a un proceso con PID conocido:

```
pid_t other_pid = 12345;                                    ①

// Enviar SIGHUP al servidor
if (kill(other_pid, SIGHUP) == -1)                          ②
{
    if (errno == ESRCH)
    {
        // Error porque el proceso no existe...
```

```
    }
    else if (errno == EPERM)
    {
        // Error de permisos para enviar la señal...
    }
    else
    {
        // Otro error...
    }
}
```

① Suponemos que conocemos el PID del proceso al que queremos enviar la señal.
② Llamar a `kill()` para enviar la señal `SIGHUP` al proceso con el PID especificado. Comprobar si hay errores y actuar en consecuencia.

B.7. Verificar si un proceso existe

Para verificar si un proceso con un PID determinado existe, se puede usar `kill()` con la señal `0`. El proceso no recibirá ninguna señal, pero la llamada a `kill()` nos indicará si el proceso no existe si retorna `-1` con `errno` igual a `ESRCH`.

```
pid_t other_pid = 12345;

if (kill(other_pid, 0) == -1)
{
    if (errno == ESRCH)
    {
        // El proceso con PID 12345 no existe
    }
}
else
{
    // El proceso con PID 12345 existe
}
```