Indice

Ejecutar otros programas

Al final estás haciendo un programa de envío de archivos en la asignatura de sistemas operativos. ¿Cómo no los vas a poder usar para enviar la salida de un comando y escribir su resultado en un archivo en el servidor? Sin duda hace falta implementar la posibilidad de poder ejecutar comandos desde el Netcp de forma similar a esta:

```
/run prueba.txt ls -la
```

y que su salida sea enviada al fichero prueba.txt en el servidor.

Lo único es que para hacerlo primero tendremos que aprender unas cuantas cosas.

fork() y exec()

Como ya hemos comentado múltiples veces, para lanzar otro programa desde nuestro proceso la clave está en usar conjuntamente las llamadas al sistema fork() y exec(). La primera crea un nuevo proceso hijo que es una copia de nuestro proceso —su proceso padre— y el segundo sustituye el programa en el ejecución por el contenido en el ejecutable indicado.

Como hemos hablado tanto de estas llamadas no nos vamos a entretener demasiado en cómo se usan sino que simplemente comentaremos algunos detalles que pueden ser útiles:

1. **Ejemplos de cómo se usan fork() y exec()** hay miles en Internet pero recuerda que dispones del que usamos en las clases de teoría.
2. **Para que un proceso padre sepa si un proceso hijo ha terminado, se dispone de las llamadas al sistema wait() y waitpid()**:
    a. **Ambas hacen que se suspenda la ejecución del proceso padre hasta que un hijo termine**.
    b. **Ambas informan del motivo por el que el hijo terminó** —normalmente, mediante un 'return' o un exit(), o por una señal— y el código del estado de salida —el número especificado en el 'return' o el exit() con el que se hizo terminar el proceso—.
    c. El sistema operativo mantiene la información sobre el estado de terminación de un proceso en el PCB[1]. Por lo tanto no puede destruir el PCB de un proceso finalizado y liberar sus recursos hasta que su proceso padre pregunta por su estado. Mientras eso no ocurra se dice que los procesos están en estado 'zombi'. Por lo tanto, **un proceso padre siempre debe preguntar por el estado de terminación de sus procesos hijos, mediante mediante wait() o waitpid(), para evitar la acumulación de procesos zombis en el sistema**.
3. **Realmente exec() no es una llamada al sistema sino una familia de llamadas al sistema** de nombre exec*(). Debemos escoger una u otra según lo que más nos convenga:
    a. Las llamadas exec*() **con 'l' permiten especificar los argumentos del comando uno detrás de otro** en la misma llamada al sistema.

[1] El PCB o bloque de control de proceso contiene toda la información que el sistema operativo maneja sobre el proceso en cuestión.

---

   b. Las llamadas exec*() **con 'v' permiten especificar los argumentos del comando con array de char***, como el argv de la función main() de nuestro programa.
   c. Las llamadas exec*() **con 'p' buscan el comando en la variable de entorno PATH si sólo indicamos su nombre sin especificar la ruta**.
   d. Las llamadas exec*() **sin 'p' necesitan que se le indique la ruta hasta el ejecutable en el primer argumento de la llamada al sistema**.
   e. Las llamadas exec*() **con 'e' permiten indicar las variables de entorno para el nuevo proceso**. En caso contrario estas variables se heredan de padre a hijo.
4. **El primer argumento de línea de comandos de cualquier programa es el nombre del propio programa**. Hay que tenerlo en cuenta al preparar los argumentos para exec().
5. Si exec() tiene éxito, ahí termina la ejecución de nuestro programa en el proceso hijo, sustituido por el programa en el ejecutable indicado. Pero ojo, porque puede fallar y en ese caso la ejecución de nuestro programa continúa en la siguiente línea del programa después de la llamada a exec(). **¿Qué vas a hacer si intentas ejecutar un comando que no existe y exec() falla?**

Comunicando procesos mediante tuberías

Si desde nuestro proceso lanzamos otro programa, este imprimirá sus resultados por la salida estándar, en la misma terminal que el nuestro. ¿Qué podemos hacer hacer si queremos capturar esa salida para procesarla de alguna manera, por ejemplo, para enviarla al servidor?

La respuesta más obvia ya la deberíamos conocer. Sólo tenemos que comunicar ambos programas. Que el programa que lanzamos por petición del usuario no imprima sus resultados por pantalla sino que los envié a nuestro programa de Netcp, para que a su vez este los mande al servidor.

En realidad, ya somos capaces de comunicar los programas de Netcp mediante socket, así que esto debería ser muy parecido. Sin embargo en este caso **no vamos a usar sockets sino tuberías por tres motivos fundamentales**:

- Así aprendemos a utilizar otra tecnología de comunicación entre procesos diferente a los sockets.
- **Las tuberías son más fáciles de utilizar cuando queremos conectar un proceso padre y un proceso hijo**. Para conectar un proceso padre con un proceso hijo no necesitamos comunicaciones en red. Las tuberías no saben nada de ese tema, así que no tenemos que preocuparnos ni de direcciones IP ni de puertos ni cuestiones similares.
- Existe un problema adicional del que no hemos hablado. **No podemos modificar el programa de cada comando que el usuario quiera ejecutar para que no imprima por pantalla sino que envíe los resultados** a nuestro programa. Esa dificultad se puede superar usando tuberías y un pequeño truco que veremos posteriormente.

Las tuberías se crean con la llamada al sistema pipe():

```
#include <unistd.h>

...

int pipe(int pipefd[2]);
```

Lo único que necesita es un array vacío de dos enteros que **a la vuelta tendrá el descriptor de archivo que representa la salida de la nueva tubería en pipefd[0] y el que representa la entrada en pipefd[1]**. Es decir:

- El proceso hijo, que ejecuta el comando, podría usar la llamada al sistema write() sobre el descriptor almacenado en pipefd[1] para enviar datos a la tubería.
- El proceso padre, que es nuestro Netcp, podría usar la llamada al sistema read() sobre el descriptor almacenado en pipefd[0] para recibir datos inyectados por el hijo en la tubería.

¿Cómo obtienen acceso padre e hijo a la misma tubería? Parece evidente que cada uno no puede llamar a pipe() por su cuenta porque entonces cada uno crearía una tubería diferente, no conectadas entre sí. **La tubería la crea el proceso padre** y al crear un proceso con fork() el hijo hereda los descriptores de archivo abiertos por el padre, obteniendo acceso a la misma tubería.

En los ejemplos que usamos en las clases de teoría puedes ver cómo se utiliza fork(), pipe(), read() y write() para comunicar un proceso padre con un proceso hijo.

Capturando la salida de un programa

Lo que hemos comentado hasta ahora está muy bien pero no termina de resolver nuestro problema. No parece realista pensar que podemos modificar todos los comandos que un usuario pueda querer ejecutar desde NetCp, para que hagan un write() en la tubería compartida en lugar de imprimir sus resultados por pantalla.

Necesitamos otra solución. Necesitamos **engañar al proceso hijo para que todo lo que intente imprimir el programa por pantalla sea enviado realmente a la entrada de la tubería**.

Redirección de la E/S estándar

Ya hemos explicado anteriormente que en general **todo proceso tiene 3 descriptores de archivos predefinidos abiertos**. Estos “archivos especiales” **se utilizan para acceder a la entrada estándar** —entrada de teclado— **la salida estándar** —salida por pantalla—**y la salida de error** —salida por pantalla destinada a errores. Por lo general **estos descriptores de archivo son 0, 1 y 2 respectivamente**. Si bien por portabilidad entre sistemas y legibilidad del código se recomienda utilizar las macros STDIN_FILENO, STDOUT_FILENO, y STDERR_FILENO respectivamente, definidas en el archivo de cabecera 'unistd.h'.

No importa el lenguaje de programación o la librería que use el programa que se ejecuta en el proceso, si se quiere leer del teclado o imprimir en la pantalla de la terminal, al final tendrá haber una llamada el sistema read() o write() sobre alguno de estos descriptores. Por lo tanto se podría interceptar la entrada o la salida de cualquier proceso si se pudiera hacer que los descriptores de archivo 0, 1 o 2 no apunten a los dispositivos de la terminal, sino a cualquier otro recurso que se pueda usar con un descriptor de archivo: un archivo, una tubería, un socket, etc.

Este sistema es tan versátil que para hacerlo el sistema operativo nos ofrece 3 llamadas al sistema: dup(), dup2() y dup3(). Las tres sirven para duplicar un descriptor de archivo en otro. Por ejemplo, una

redirección de la entrada estándar —descriptor 0 o STDIN_FILENO— desde un archivo de nombre 'entrada.txt' se haría así:

```
int fd = open("entrada.txt", O_RDONLY);
if (fd < 0) {
     ...        // Aquí manejo de errores por no abrir el archivo
}

int result = dup2(fd, STDIN_FILENO)
if (result < 0) {
     ...        // Aquí manejo de errores por no duplicar el descriptor
}

close(fd);
```

Veamos lo que hemos hecho en detalle:

1. **Usamos la llamada al sistema open() para abrir el archivo 'entrada.txt'**, obteniendo un descriptor de archivo que se almacena en la variable 'fd'. El archivo se abre en modo de sólo lectura —O_RDONLY— porque se va a conectar a la entradas estándar que sólo se usa para leer, por lo general, datos del teclado.
2. **dup2() hace la redirección de la entrada estándar al archivo 'entrada.txt'**. Cierra el descriptor de archivo indicado en el segundo argumento —STDIN_FILENO, el de la entrada estándar— y después copia en ese mismo descriptor los detalles del recurso abierto en el descriptor de archivo del primer argumento —'fd', el descriptor del archivo 'entrada.txt' abierto previamente—.
3. **Cerramos el descriptor de archivo que ya no necesitamos**. Si ninguna de las llamadas al sistema falla, después de invocar a dup2() se puede leer el contenido del archivo 'entrada.txt' tanto a través del descriptor STDIN_FILENO como del descriptor almacenado en ‘fd’. Es decir, tenemos el mismo archivo abierto dos veces. Por lo que si no vamos a usar nunca más el descriptor ‘fd’, podemos cerrarlo.

Una vez hecho esto todas las operación sobre la entrada estándar —por ejemplo, std::cin, std::scanf(), std::getline(), std::gets(), std::getchar(), etc.— realmente leerán 'entrada.txt' sin saberlo el programa que se ejecuta.

Utilizar una tubería para capturar la entrada o la salida de un proceso es muy similar. Solo hay que utilizar los descriptores de archivo devueltos por pipe(). Por ejemplo, en el problema concreto de capturar la salida estándar de los comandos ejecutados por nuestro programa de Netcp, **sólo tenemos que duplicar el descriptor en pipefd[1]** —la entrada de la tubería— **en el descriptor STDOUT_FILENO** —la salida estándar— **del proceso hijo** y después ya podemos usar exec() para lanzar el nuevo programa en el proceso.

Leer la tubería desde el proceso padre

La solución al problema que nos hemos planteado requiere de tres pasos:

1. Crear una tubería que conecte al proceso padre con el proceso hijo.

2. Engañar al hijo para que cuando imprima en su salida estándar realmente esté inyectado contenido a la entrada de la tubería.
3. Que el proceso padre lea de la salida de la tubería los contenidos enviados por el hijo.

De esos tres pasos solo nos queda hablar del último pero por suerte hay muy poco que contar. Ya hemos visto que las tuberías, desde el punto de vista de los procesos, son como archivos. Se accede a ellas mediante descriptores de archivo, así que **se lee de ella con read(), como se lee de cualquier archivo abierto**. Lo único que debemos tener en cuenta es que las tuberías son un canal de comunicación. Los datos solo se pueden leer si se han transmitido desde el otro extremo, por lo que **generalmente hay que hacer varias lecturas para capturar toda la salida del programa.**

Consideraciones adicionales

Hay dos cuestiones adicionales que debemos tener en cuenta:

1. Por lo general se lee de la tubería hasta que ya no se puede más —es decir, cuando la llamada al sistema devuelve un "fin de archivo"—. Obviamente cuando eso ocurre es porque el proceso en el otro extremo ha terminado. Pero pese a saber esto, **no debemos olvidarnos de llamar a wait() o waitpid() para no dejar procesos zombis detrás**.
2. **Es importante evitar dejar abiertos descriptores que no necesitamos**. Por ejemplo, tener abierto en el hijo el descriptor de salida de la tubería cuando sólo lo va a usar el padre para leer de ella. O dejar el viejo descriptor de entrada abierto al duplicar, cuando realmente el proceso sólo va a usar el descriptor de la salida estándar.

Esto último punto es muy importante porque el sistema operativo solo informa en la salida de la tubería que se ha alcanzado el final de archivo cuando se cierran todos los descriptores de entrada abiertos —porque a partir de ese momento es imposible que lleguen más datos—. La notificación de final de archivo al proceso padre debe ocurrir cuando el proceso hijo muere, porque al terminar se cierra automáticamente el descriptor de su extremo de la tubería. Pero si quedase por algún lado otro descriptor abierto de entrada, el sistema operativo supondrá que puede ser usado para inyectar más datos en la tubería y no notifica de nada al padre. Esto seguramente dejaría bloqueado al proceso padre esperado por unos datos de la tubería que nunca llegarán.

[TAREA] Implementa la ejecución de programas

Modifica tu prototipo de Netcp para que que soporte la ejecución de cualquier comando usando la sintaxis:

/run prueba.txt comando arg1 arg2 ...

por ejemplo “/run prueba.txt ls -la“ o “/run hola.txt ps aux”, teniendo en cuenta que:

El programa debe capturar la salida estándar y la de error, y enviarla al archivo indicado. La salida de error la enviará como otro fichero con el mismo nombre pero con extensión .err.

Por último, **no te olvides de manejar los posibles errores de las llamadas al sistema adecuadamente**. Ten en cuenta que no todos los errores son irrecuperables. Es decir, **hay errores para los que la única solución no es terminar el programa**.

Histórico de mensajes

En modo servidor vamos a almacenar la información de los archivos que hemos recibido, a modo de resumen de comportamiento. Se podría hacer abriendo un nuevo archivo y guardando la información, pero esta vez lo vamos a hacer de una manera diferente mapeando el archivo en memoria con mmap().

Abrir el archivo a mapear

Lo primero antes de mapear un archivo en la memoria del proceso es abrirlo ya que mmap() necesitará un descriptor del archivo que queremos mapear. Ya conocemos de sobra como se hace esto.

Mapear el archivo en memoria

La llamada al sistema mmap() es una de las que admite más opciones:

```
#include <sys/mman.h>

...

void* mmap(void* addr, size_t length, int prot, int flags, int fd,
      off_t offset);
```

Los argumentos sencillos de entender son:

- **fd**, es el descriptor del archivo que queremos mapear.
- **length**, es el número de bytes del archivo que queremos mapear y, por tanto, es la longitud de la región de memoria que quedará reservada y mapeada al archivo. La llamada mapea páginas completas así que “length” debe ser múltiplo del tamaño de página de nuestro sistema —por defecto son de 4k en Linux pero puede obtenerse con sysconf(_SC_PAGE_SIZE)—.
- **offset**, la posición en bytes en el archivo donde queremos que comience el mapeo. Por ejemplo, si queremos que sea desde el principio debemos ponerlo a 0. También debe ser un número múltiplo del tamaño de página.

Mientras que los otros argumentos:

- **addr**, es una dirección en la memoria de proceso donde sugerimos al sistema operativo que haga el mapeo del archivo. Como por lo general no tenemos ninguna preferencia, se suele poner a NULL. Pero si se usa debe ser múltiplo del tamaño de página.
- **prot**, indican los permisos de protección de las páginas en la memoria. Obviamente si piensas leer de la memoria necesitarás permisos de lectura y si quieres escribir permisos de escritura.

- **flags**, configura algunos aspectos del comportamiento de la llamada al sistema. De todos los posibles sólo nos interesan los del estándar POSIX —como verás en el manual, Linux soporta muchos más—:
    - **MAP_SHARED**, hace que el mapeo sea compartido con otros procesos que mapean el mismo archivo. Si eso ocurriera, los cambios de un proceso serían vistos por los otros procesos automáticamente, aunque dichos cambios no hayan sido guardados aún en el archivo.
    - **MAP_PRIVATE**, hace que el mapeo sea privado al proceso. Los cambios en la región de memoria mapeada no son visibles para otros procesos que hayan mapeado el mismo archivo y de hecho tampoco se guardan en el archivo.

Si la llamada tiene éxito **devuelve un puntero a una zona de la memoria del proceso donde automáticamente tenemos acceso a "length" bytes del archivo comenzado en la posición "offset" del mismo**.

La ventaja de esto es que podemos acceder a todo el contenido del archivo y modificarlo sin hacer más llamadas al sistema. Será el propio sistema operativo el que vaya trayendo las páginas según haga falta y el que las escriba cuando crea más conveniente si las modificamos.

Una vez creado el mapeo puedes cerrar el archivo con close() si no lo vas a usar más, puesto que eso no lo desmapeará de la memoria.

Desmapear de la memoria

La llamada al sistema munmap() **desactiva el mapeo para el rango de la memoria virtual indicado, que comienza donde apunta 'addr' y se extiende 'length' bytes**, haciendo que futuros accesos a esa región sean considerados como accesos inválidos:

```
#include <sys/mman.h>

...

int munmap(void* addr, size_t length)
```

Hay que tener en cuenta que terminar el proceso provoca el desmapeo automático de las regiones previamente mapeadas pero cerrar el descriptor de archivo no.

Bloqueo de archivos

La solución anterior funciona para controlar el acceso concurrente entre hilos de un mismo proceso. Así que no tendríamos problemas mientras no hayan más procesos que quieran abrir, mapear y manipular el archivo del historial. Pero **¿qué ocurre si el mismo usuario del sistema lanza varias copias del programa NetCp en modo servidor?** —como seguramente has estado haciendo hasta ahora para comprobar que el programa funciona—. Lo más probable es que el historial se corrompa al haber hilos de distintos procesos que escriben en él al mismo tiempo.

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

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

```
#include <unistd.h>

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

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

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

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

La llamada al sistema lockf() tiene algunas características que debemos tener presentes:

- **El tipo de bloqueo es sugerido**, al igual que con las otras dos llamadas al sistema comentadas. Eso significa que los procesos pueden cooperar usando lockf() para bloquear archivos y saber si un archivo está bloqueado por otro proceso. Pero eso no impide que otros procesos puedan ignorar el uso de lockf() para acceder al archivo y manipularlo libremente, incluso cuando está bloqueado.
- **Este tipo de bloqueo de archivos no se puede utilizar para controlar el acceso exclusivo entre hilos de un mismo proceso**, porque los bloqueos son compartidos por todos los hilos del proceso. Es decir, si un hilo de un proceso bloquea un archivo, el resto de los hilos del mismo proceso no se suspenderán si intentan bloquear el archivo porque se supone que ya tiene acceso todo el proceso. Para que el acceso sea realmente exclusivo es necesario utilizar un mutex —para el bloqueo de los hilos del mismo proceso— junto con la llamada al sistema lockf() —para el bloqueo de hilos de otros procesos—.
- **La llamada al sistema permite el bloqueo de secciones de bytes dentro del archivo**, en lugar de bloquear siempre el archivo completo.
- **El archivo se desbloquea si el proceso que lo tiene bloqueado cierra el archivo**.

Respecto a los argumentos de lockf():

1. **El primer argumento es el descriptor del archivo** abierto que se quiere bloquear.
2. **El segundo argumento es la operación** que se quiere realizar:
    a. **F_LOCK**, para bloquear de forma exclusiva. Es decir, si ya existe un bloqueo en manos de otro proceso, el hilo es suspendido a la espera de que pueda bloquearlo.
    b. **F_ULOCK**, para desbloquear.
    c. **F_TLOCK** es como F_LOCK. Sirve para bloquear de forma exclusiva pero devuelve un error si ya existe un bloqueo en manos de otro proceso.
    d. **F_TEST** comprueba si ya existe un bloqueo. Devuelve 0 si no hay bloqueo o si existe pero es de este mismo proceso.

---

3. **El tercer argumento es la longitud en bytes de la sección del archivo** que se quiere bloquear o desbloquear. La sección empieza en la ubicación actual del indicador de posición de las operaciones sobre el archivo —ver lseek()— y se extiende el número de bytes indicado en este argumento. **Si el argumento vale 0**, la sección de bytes se extiende desde la posición actual del indicador hasta el infinito, lo que **permite bloquear el archivo completo si dicho indicador de posición está en el byte 0 del archivo**.

[TAREA] Implementa el historial

Modifica tu prototipo de NetCp para que en el modo servidor mantenga un historial de los ficheros recibido.. Para ello utilizaremos un fichero de texto donde se guardará el nombre del archivo, la dirección IP de origen, el puerto de origen y el tamaño. Colocaremos un archivo en cada línea.

1. **El acceso al historial debe hacerse mapeando el archivo en la memoria**.
2. **Te recomendamos que crees una clase History que siga el patrón RAII** para implementar esta funcionalidad:
    a. Que cree el archivo y lo mapee desde el constructor.
    b. Que desmapee en el destructor.
    c. Que tenga un método add_file() para añadir el nuevo archivo recibido.
3. **El historial debe ser un registro circular de aproximadamente 1MB de las últimas conversaciones**:
    a. Recuerda que con ftruncate() puedes cambiar el tamaño del archivo y que debes darle el tamaño correcto antes de usar mmap() para mapearlo.
    b. Se debe guardar de forma ordenada los archivos recibidos.
    c. Al ejecutar, terminar y volver a ejecutar el programa **el historial debe mantenerse, añadiéndose a partir de donde se quedó al terminar el programa la última vez**. Una forma de hacer esto es truncar con ftruncate() el archivo, hasta dónde está ocupado, antes de cerrar el historial. De forma que al abrirlo de nuevo se puede consultar su tamaño para saber dónde continuar.
    d. Al llegar al final del archivo se debe comenzar a sobreescribir los archivos más antiguos comenzando por el principio del archivo.
4. **El historial debe ser un archivo almacenado como "~/.Netcp/<usuario>.log"** donde <usuario> es el nombre del usuario en el Netcp:
    a. Observa que el nombre del directorio lleva un "." para que permanezca oculto. Esto es muy común.
    b. Tanto el archivo como el directorio que contiene el historial tendrán que ser creados por el programa si no existen cuando el programa es ejecutado. Funciones como dirname(), opendir() o stat() —para intentar abrir un directorio con el objeto de determinar si existe— pueden ser muy útiles. Mientras que mkdir() te servirá para crear el directorio.
    c. **Evita el acceso concurrente de varios procesos al mismo archivo del historial**. El usar como nombre de archivo "<usuario>.log" evita que dos procesos Netcp con usuario diferente usen el mismo archivo. Pero no evita problemas si usan el mismo nombre. Usa el bloqueo de archivos para que solo el primero que mapee un archivo de historial pueda usarlo. Los que lleguen detrás deberán detectar el bloqueo y terminar el programa indicando que el nombre de usuario ya se está usando.
5. **Recuerda que el historial es para que lo lean los usuarios usando comandos de texto** como 'cat', 'less', 'vi' o cualquier editor. Pónselo fácil.. Si tienes dudas mira el contenido de archivos de "/var/log/ como "/var/log/syslog".

[OPCIONAL] Puedes intentar incluir en el historial la fecha y la hora en la que se escribió. Si te atreves con esta tarea, mira std::time().

Por último, **no te olvides de manejar los posibles errores de todas estas llamadas al sistema adecuadamente.** Nunca sabes con seguridad que una llamada al sistema no va a fallar nunca.

[OPCIONAL] Puedes intentar enviar en cada mensaje la fecha y la hora en la que se escribió, mostrarlas junto a cada mensaje en la terminal y guardarlas igualmente en el historial. Si te atreves con esta tarea, mira std::time().

Por último, **no te olvides de manejar los posibles errores de todas estas llamadas al sistema adecuadamente.** Nunca sabes con seguridad que una llamada al sistema no va a fallar nunca