Programación de aplicaciones — Parte III

Sistemas Operativos 2023/2024

Vamos a desarrollar una herramienta para copiar archivos entre equipos conectados a Internet. En este caso, añadiremos las características necesarias para poder capturar la salida de un comando y enviarla a otro `netcp`, así como para recibir un archivo y usarlo como entrada del programa que le indiquemos.

Contenidos

1. Introducción

1.1. Objetivo

Adicionalmente, a lo que se ha implementado en partes anteriores, la nueva versión de `netcp` soportará indicar a través de los últimos argumentos de la línea de comandos un nombre de comando y sus argumentos. Para que el programa no interprete el nombre del comando como un nombre de archivo, se utilizará la opción `-c` para activar este modo de funcionamiento:

```
$ ./netcp -c ls -a /etc
```

El programa ejecutará el comando indicado, capturará su salida estándar y la enviará al otro `netcp`, usando el protocolo de comunicación que hemos desarrollado en las partes anteriores.

Cuando se usa la opción `-c` también se podrá utilizar `-1`, que indicará que se quiere capturar y enviar la salida estándar del comando, que es el comportamiento por defecto. Mientras que si se indica la opción `-2`, solo se capturará del comando la salida de error:

```
$ ./netcp -2 -c ls -a /etc
```

Sí se indican ambas opciones, se capturarán y enviarán tanto la salida estándar como la salida de error.

```
$ ./netcp -1 -2 -c ls -a /etc
```

En el modo `-l` también se podrá indicar un nombre de comando y sus argumentos usando la opción `-c`. En ese caso, el comando indicado recibirá por su entrada estándar los datos recibidos por `netcp`.

```
$ ./netcp -l -c wc -l
```

En todos los casos anteriores, si el programa `netcp` recibe las señales `SIGINT`, `SIGTERM`, `SIGHUP` o `SIGQUIT`, matará al proceso hijo donde se ejecuta el comando indicado –si aún no ha finalizado– antes de terminar.

A modo de resumen, estos son los argumentos de línea de comandos que debe soportar la versión definitiva de `netcp`:

```
netcp -h
netcp ARCHIVO
netcp [-1] [-2] -c COMANDO [ARG...]
netcp -l ARCHIVO
netcp -l -c COMANDO [ARG...]
```

2. Implementación

En el modo de funcionamiento normal, abrimos el archivo indicado con `open()`, para obtener un descriptor de archivos, y usamos nuestras funciones `read_file()` o `write_file()`, para leer o escribir los datos del archivo a través de dicho descriptor.

Para el modo `-c` implementaremos la clase `subprocess`, que se encargará de ejecutar el comando indicado y darnos un descriptor, con el que también usaremos `read_file()` o `write_file()`, pero para leer o escribir la salida o la entrada estándar del comando, respectivamente:

```
class subprocess
{
public:
```

```
    enum class stdio
    {
        in,
        out,
        err,
        outerr
    };

    subprocess(
        const std::vector<std::string>& args,                    ①
        subprocess::stdio redirected_io                          ②
    );

    bool is_alive();                                             ③

    std::error_code exec();                                      ④
    std::error_code wait();                                      ⑤
    std::error_code kill();                                      ⑥

    pid_t pid();                                                 ⑦

    int stdin_fd();                                              ⑧
    int stdout_fd();                                             ⑨
};
```

① `args` es la lista de argumentos que se le pasan al comando, incluido el nombre del comando como primer argumento.
② `redirected_io` indica qué entrada/salida del comando se redirigirá.
③ `is_alive()` devuelve `true` si el proceso hijo está ejecutándose o `false` si ya ha terminado o no ha empezado a ejecutarse.
④ `exec()` crean un proceso hijo y ejecuta el comando indicado en `args` con sus argumentos. No espera a que el proceso termine y devuelve un código de error si no se pudo ejecutar el comando.
⑤ `wait()` espera a que el proceso hijo termine. Devuelve un código de error si no se pudo esperar al proceso hijo.
⑥ `kill()` mata al proceso hijo. Devuelve un código de error si no se pudo matar al proceso hijo.
⑦ `pid()` devuelve el PID del proceso hijo o -1 si aún no se ha ejecutado el comando.
⑧ Devuelve el descriptor de archivos de la entrada estándar, si está siendo redirigida, o -1 si no se está redirigida. Este descriptor se puede utilizar con `write_file()` para escribir en la entrada estándar del comando.
⑨ Devuelve el descriptor de archivos de la salida estándar o salida de error, si alguna está siendo redirigida, o -1 si ninguna está redirigida. Este descriptor se puede utilizar con `read_file()` para leer la salida estándar o de error del comando.

Por ejemplo, con la clase implementada, para ejecutar el comando `ls -a /etc` y capturar su salida estándar, usaríamos la clase `subprocess` así:

```
subprocess ls( {"ls", "-a", "/etc"}, subprocess::stdio:out);

std::error_code error = ls.exec();
if (error)
{
    // ...
}

std::vector<uint8_t> buffer(1024);
error = read_file(ls.stdout_fd(), buffer);
```

> **!** Importante
>
> Es importante recordar que probablemente no se pueda leer toda la salida del comando de una vez, ya que el proceso hijo puede tardar en generarla. Por tanto, será necesario usar `read_file()` múltiples veces, hasta que devuelva un vector vacío, igual que cómo hacíamos para leer archivos.

2.1. Cómo saber si un hijo se está ejecutando

Antes de llamar a `subprocess::exec()` el método `subprocess::is_alive()` tiene que devolver `false`. Sin embargo, una vez el proceso hijo ha empezado a ejecutarse, `subprocess::is_alive()` tiene que devolver `true` hasta que el proceso hijo termine.

Para saber si un proceso hijo ha terminado, podemos usar la función de la librería del sistema `waitpid()` con la opción `WNOHANG`:

```
pid_t result = waitpid(pid, &status, WNOHANG);
```

Si `waitpid()` devuelve 0, es que el proceso hijo `pid` aún se está ejecutando.

2.2. Ejecución de comandos

La función `subprocess::exec()` tiene la responsabilidad de ejecutar el comando usando las funciones de la librería del sistema `fork()` y `exec()`.

En el tema 9 de los apuntes de la asignatura se explica en detalle cómo ejecutar un programa externo usando las funciones de la librería del sistema `fork()` y `exec()` y se enlazan algunos ejemplos de código.

2.2.1. Elección de la versión de exec()

Como comentamos en los apuntes de la asignatura, la `exec()` es una familia de funciones, con varias versiones que se diferencian en el tipo de argumentos que reciben.

Como los usuarios esperan poder ejecutar comandos así:

```
$ ./netcp -c ps -A
```

sín especificar la ruta para encontrar el ejecutable de `ps` –esperando que `netcp` lo localice y lo ejecute– necesitamos que la función `subprocess::exec()` use la versión de `exec()` que sabe buscar una ejecutable en los directorios de la variable de entorno `PATH`, cuando no se le indica la ruta del programa.

Además, los usuarios pueden ejecutar cualquier programa con cualquier número de argumentos, por lo que no sirven las versiones de `exec()` donde el número de argumentos queda preestablecido en el código, si no aquellas que reciben un array de argumentos, similar al que recibe `main()`.

2.2.2. Conversión de los argumentos

Una ver elegida la función `exec()`adecuada, el siguiente reto de `subprocess::exec()` es convertir `args` –la lista de argumentos `std::vector<std::string>`– en los argumentos que espera `exec()` –un array de cadenas de caracteres `char*[]`–.

Para esta conversión, la función puede hacer lo siguiente:

> Crear `argv` como `std::vector` de `const char*`
> **for** `arg` **in** `args` **do**
> Añadir `arg.c_str()` a `argv`
> **end for**
>
> Añadir `nullptr` a `argv`
> Ejecutar el programa con `exec(argv[0], argv.data())`

Es muy importante el paso de añadir `nullptr` al final de `argv`, ya que `exec()` espera que el array de argumentos que se le pase termine con un puntero nulo. De esta forma `exec()` sabe, al recorrer el array, que ha llegado al final y cuántos argumentos contiene.

Por otro lado, es posible que el tipo devuelto por `argv.data()` no coincida con el esperado por `exec()`. El tipo esperado por `exec()` es `char* const[]` –o el equivalente `char* const*`– mientras que el tipo devuelto por `argv.data()` es `const char**`. Esto se puede resolver haciendo un *typecast* explícito: `const_cast<char* const*>(argv.data())`.

2.3. Espera y procesos zombies

Al salir de `subprocess::exec()` con éxito, el proceso hijo se estará ejecutando mientras el proceso padre continuará con su ejecución.

Sea cual sea el motivo por el que el proceso padre quiera terminar, debe asegurarse de que el proceso hijo también termina y esperar a que lo haga llamando a `subprocess::wait()`, que a su vez debe usar la función de la librería del sistema `waitpid()`. Esto es necesario para evitar que el proceso hijo se quede como un proceso zombie.

2.4. Redirection de la E/S estándar

Para redirigir la E/S del comando, `subprocess::exec()` debe hacer lo siguiente:

Crear una tubería con `pipe()`
Crear el proceso hijo con `fork()`
if dentro del proceso hijo **then**
    Redirigir la E/S estándar a la tubería usando `dup2`
    Cerrar el extremo de la tubería que no se usa en el hijo
    Ejecutar el comando con `exec()`
else
    Cerrar el extremo de la tubería que no se usa en el padre
    Guardar el descriptor del extremo que si se va a usar
enf if

Este procedimiento también se ilustra en uno de los ejemplos de código de la asignatura.

Por ejemplo, en caso de que se redirija la entrada estándar, el proceso padre debe guardar el extremo de escritura de la tubería en el objeto `subprocess`, para que el resto del código puede acceder a él mediante el método `subprocess::stdin_fd()`. Mientras que en el proceso hijo se debe redirigir el extremo de lectura usando `dup2()` a su entrada estándar.

Si redirige la salida estándar, el proceso padre debe guardar el extremo de lectura de la tubería —para se accesible al resto del programa mediante el método `subprocess::stdout_fd()`— mientras que en el proceso hijo se debe redirigir el extremo de escritura a su salida estándar.

Si se quieren redirigir tanto la salida estándar como la salida de error, el proceso padre debe guardar el extremo de lectura de la tubería en la clase `subprocess`, mientras que en el proceso hijo se debe redirigir el extremo de escritura de la tubería a su salida estándar y a su salida de error, al mismo tiempo. Por tanto, ambas salidas se mezclarán en la tubería y se enviarán al proceso padre a través del mismo extremo de lectura, que estará disponible para el resto de código de `netcp` a través del método `subprocess::stdout_fd()`.

Tanto en el padre como el hijo, los descriptores que ya no son necesarios se deben cerrar con `close()`.

Qué E/S estándar se debe redirigir en `subprocess::exec()` depende del valor de `subprocess::stdio redirected_io` que se haya pasado al constructor de la clase `subprocess`. Si es `subprocess::stdio::in`, se redirigirá la entrada estándar. Si es `subprocess::stdio::out`, se redirigirá la salida estándar. Si es `subprocess::stdio::err`, se redirigirá la salida de error. Y si es `subprocess::stdio::outerr`, se redirigirán tanto la salida estándar como la salida de error.

Los extremos guardados en la clase `subprocess` en el padre, son los descriptores que devuelven los métodos `subprocess::stdin_fd()` y `subprocess::stdout_fd()`, según corresponda. Recordando, que si se pide un descriptor que no está siendo redirigiendo, el método debe devolver -1.

Por ejemplo, si en el constructor se indica `subprocess::stdio::out`, el método `subprocess::stdin_fd()` debe devolver -1, mientras que `subprocess::stdout_fd()` debe devolver el descriptor del extremo de lectura de la tubería.