Sesión 5: Programación con sockets

Concurrencia en el servidor

José Luis Díaz

Curso 2011–2012

En esta sesión abordaremos el problema de la concurrencia en el servidor, es decir, cómo programar servidores capaces de mantener varias conexiones TCP abiertas simultáneamente y con capacidad para responder a todos los clientes conectados a ellas.

Para el mejor desarrollo de la práctica resulta conveniente conocer someramente la forma en que Unix maneja los procesos, cómo listarlos usando el comando `ps`, cómo enviarles señales usando el comando `kill`, y qué efecto tienen estas señales sobre los procesos y cómo cambiar ese efecto. Estos aspectos se recogen en el anexo 4, pues no forman parte de la programación de las comunicaciones. No obstante se recomienda al alumno, especialmente si no está familiarizado con estas cuestiones, que realice también los experimentos sugeridos en dicho apéndice.

1. Experimento inicial

Toma el `echo-tcp-basico.c` cuyo código ya vimos en la sesión 3, compílalo y ejecútalo. Desde otra terminal conéctate con ese servidor, usando `telnet` y comprueba cómo te devuelve el eco de lo que le envías.

Ahora, sin cerrar el `telnet` anterior, abre una terminal más y trata de conectarte con el mismo servidor. Verás que `telnet` te indica que la conexión tiene éxito, pero sin embargo no recibes eco de nada de lo que escribes. Lo que ha ocurrido es que, cuando `telnet` ha llamado a `connect()`, pasándole una IP y número de puerto, el protocolo TCP ha comprobado si el socket al que pretende conectarse existe, y si tiene sitio para un cliente más. Siendo así, TCP da por completada la conexión. El cliente `telnet` retorna de la llamada a `connect()`, pero en realidad, ya que en el extremo del servidor éste no está ejecutando `accept()` la conexión no está del todo completa. El socket de escucha tiene una cola en la que queda almacenada esta petición pendiente. Tan pronto como el servidor llame a `accept()`, la conexión quedará completa.

El caso es que, hasta entonces, todo lo que tecleamos en `telnet`, se envía por esa conexión incompleta, pero no se recibirá en el otro extremo hasta que el servidor haya hecho `accept` y después `read`.

Puedes comprobar que, tan pronto como sales de `telnet` en el primer cliente, el segundo comienza a recibir el eco de lo que había enviado.

Vamos a modificar el servidor para que dos o más clientes puedan estar conectados y recibiendo eco simultáneamente.

2. Creación previa de procesos

Modifica el servidor para que use la técnica de creación previa de procesos. Tras crear el socket de escucha y asignarle una dirección, crea dos procesos hijos usando `fork()`. Tras el `fork()` los tres procesos (padre y dos hijos) hacen exactamente lo mismo: entran en un bucle infinito en que llaman a `accept()` para recibir un cliente, y después entran en otro bucle en que leen datos y los devuelven hasta detectar el cierre del socket de datos.

Modifica el código también para que, cuando `accept` retorne, además de imprimir "Se ha conectado un cliente", imprima "Soy el proceso XXXXX", donde XXXXX es el `PID` del proceso en cuestión, el cual se averigua llamando a la función `getpid()`, que retorna un entero. Puedes imprimir este PID también cada vez que el servidor reciba un texto del cliente.

Ejecuta el servidor y comprueba con `ps` cómo hay tres copias del proceso, y las tres durmiendo. Cuando desde otra terminal, usando `telnet` te conectes al servidor, uno de estos tres procesos despertará para completar su `accept`. Puedes tener hasta tres clientes simultáneos "hablando" con el servidor. Cada uno hablará con una copia del proceso, como podrás comprobar por los `PID` que el servidor muestra en pantalla. ¿Qué crees que pasará si se intenta conectar un cuarto cliente?

3. Creación de procesos conforme llegan clientes

La otra posibilidad es invocar a `fork()` cada vez que un cliente es aceptado. El nuevo proceso así creado entra en un bucle en el que recibe datos del cliente y se los envía de nuevo, hasta detectar el cierre del socket, y en ese momento finaliza su ejecución con `exit()`. Mientras tanto el proceso original cierra el socket de datos y vuelve al `accept()` a esperar por otro cliente.

En esta implementación hay que tener cuidado con los procesos *zombies*, puesto que los procesos hijo mueren antes que el padre (véase el anexo 4.2.1).

Haz las modificaciones necesarias en el servidor para que se comporte como se ha descrito, y llama al resultado `echo-fork.c`. Ejecútalo y comprueba desde otra terminal (usando `ps`) que inicialmente sólo hay una copia del proceso. Desde una tercera terminal, conéctate con el servidor mediante `telnet` y verifica desde la segunda terminal que ha aparecido un proceso nuevo. Puedes probar a seguir lanzando terminales con sesiones `telnet` y comprobar que todas reciben eco, a la vez que constatas cómo van apareciendo los nuevos procesos. Cuando las sesiones `telnet` van cerrando, los procesos hijo deberían desaparecer (si quedan zombies, has olvidado ignorar la señal).

4. Anexo: conceptos de programación en Unix

4.1. Las señales

En Unix es posible notificar a un proceso que esté en ejecución de que algo ha ocurrido enviándole una señal. Realizaremos algunos experimentos para comprender esta idea.

Comencemos por compilar y ejecutar el programa `espera.c` (el listado se muestra en el apéndice 5.1, pero se suministra el fuente en clase de prácticas). Este programa simplemente espera a que el usuario pulse una tecla, y seguidamente muestra por pantalla la tecla pulsada. Repite este comportamiento una y otra vez. Al ejecutarlo veremos algo como:

```
$ espera
Escribe algo: Hola
Has escrito: 'Hola'
Escribe algo:
```

Abre otra terminal y conéctate a la misma máquina en que está ejecutándose este programa. Si desde esta nueva terminal usas el comando `ps`, con las opciones adecuadas podrás ver el programa `espera` entre los procesos en ejecución. En particular, usaremos las opciones `uxf`. La opción `u` indica que queremos ver la información en formato "de usuario" (afecta a qué información mostrará para cada proceso). La opción `x` significa que queremos ver también los procesos que no estén conectados con ninguna terminal. Finalmente la opción `f` indica que queremos que nos muestre los nombres de los programas formando un "árbol" que permita ver fácilmente qué procesos son los padres y cuáles los hijos. El resultado será algo como esto:

```
$ ps uxf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
arriba   31966  0.0  0.0   9088  2868 ?        S    16:50   0:00 sshd: arriba@pts/0
arriba   31967  0.0  0.0   2844  1552 pts/0    Ss   16:50   0:00  \_ -bash
arriba   31990  0.0  0.0   2308   688 pts/0    R+   16:51   0:00      \_ ps uxf
arriba   31634  0.0  0.0   9088  2868 ?        S    16:38   0:00 sshd: arriba@pts/15
arriba   31635  0.0  0.0   2852  1608 pts/15   Ss   16:38   0:00  \_ -bash
arriba   31957  0.0  0.0   1360   300 pts/15   S+   16:49   0:00      \_ ./espera
```

La salida de este comando nos da mucha información (aún podría dar más si en lugar de la opción `u` usamos `l`). En particular nos interesa la columna titulada `PID`, que nos dice el número que identifica cada proceso, y la columna `COMMAND` que nos da el nombre del proceso y un rudimentario dibujo que indica qué proceso es el padre. También puede resultar interesante la columna `TTY` que indica el nombre de la terminal asociada con ese proceso (se trata de la terminal desde la cual fue lanzado y sobre la que realizará su entrada/salida). Así vemos que hay un par de procesos llamados `sshd`, que no tienen terminal asociada (son los servidores que he arrancado por cada conexión *ssh* que tengo activada), estos procesos han creado cada uno de ellos un hijo llamado `bash`, que es el *interfaz de comandos* que usamos para navegar por las carpetas, listarlas, crear ficheros, compilarlos o ejecutarlos. Vemos que uno de los `bash` ha creado un hijo llamado *espera*, mientras que otro ha creado un hijo llamado `ps uxf`. Es evidente que se trata de los comandos que he lanzado desde cada una de las terminales.

La columna STAT me da el estado de cada proceso. Veo que la mayoría de ellos están en un estado S, que significa "*sleeping*" (el proceso está esperando a que algo ocurra, por ejemplo bash está esperando a que su hijo "espera" finalice, mientras que espera está esperando a que el usuario introduzca texto. Sólo uno de los procesos está en estado R (*running*).

A través de ps he averiguado que el PID del proceso espera vale 31957 (evidentemente el número puede ser diferente en tu caso). Conociendo su PID puedo enviarle una señal, usando el comando kill[1]. Vamos a enviar la señal SIGHUP, que significa que la conexión con la terminal se ha cortado (literalmente significa que el módem ha colgado). Tras el envío de esta señal podemos ver que el programa ya no está en ejecución:

```
$ kill -HUP 31957
$ ps uxf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
arriba   31966  0.0  0.0   9088  2868 ?        S    16:50   0:00 sshd: arriba@pts/0
arriba   31967  0.0  0.0   2844  1560 pts/0    Ss   16:50   0:00  \_ -bash
arriba   32127  0.0  0.0   2308   684 pts/0    R+   17:02   0:00      \_ ps uxf
arriba   31634  0.0  0.0   9088  2868 ?        S    16:38   0:00 sshd: arriba@pts/15
arriba   31635  0.0  0.0   3032  1736 pts/15   Ss+  16:38   0:00  \_ -bash
```

Además veremos que en la otra terminal ha aparecido el mensaje "Colgar (hangup)". Esta señal, como vemos, causa que el proceso que la recibe termine su ejecución. Otras señales que podemos enviarle para causar que finalice son SIGQUIT, SIGTERM, SIGINT ó SIGKILL. Si experimentas lanzando de nuevo el proceso desde una terminal y enviándole señales desde la otra verás que en todos estos casos el proceso termina su ejecución, si bien el mensaje que imprime en pantalla es diferente cada vez.

Ante la señal SIGINT no imprime nada. Esta señal es especial, porque no necesitamos ir a otra terminal para enviarla. Desde la misma terminal en la que el proceso se está ejecutando podemos enviarle esta señal si pulsamos Ctrl-C. Es por esto que solemos decir que Ctrl-C sirve para terminar la aplicación, pero la afirmación no es del todo cierta. Para lo que sirve es para enviarle la señal SIGINT, la cual por defecto termina la aplicación, pero podría ser manejada de otro modo.

4.1.1. Manejando las señales

Al arrancar un proceso, automáticamente tiene predefinido un curso de acción para cada señal que pueda llegarle. Por defecto, la mayoría de las señales causan que el proceso termine, salvo unas pocas entre las que se encuentra SIGCHLD. Si queremos que, en lugar de terminar, el proceso haga otra cosa al recibir la señal, podemos asignarle un manejador a la señal en cuestión, mediante la función signal(). Esta función recibe el nombre de la señal y el nombre de una función. La función será ejecutada cuando la señal llegue, sin importar lo que estuviera haciendo el proceso en ese momento. Es necesario que la función en cuestión sea declarada con un parámetro de tipo entero (a través del cual recibirá el número de señal que causó su activación) y valor retornado void.

Por ejemplo, el siguiente código causa que al recibir SIGINT, en lugar de terminar, se ejecute la función DetectadaINT():

```
...
void DetectadaINT(int senial)
{
  printf("Detectada la señal INT!\n");
  exit();
}
main()
{
 ...
 signal(SIGINT, DetectadaINT);
 ...
}
```

Prueba a añadir este código a espera.c. Después ejecútalo y pulsa Ctrl-C ¿qué crees que ocurrirá? Esto suele utilizarse para hacer que un servidor cierre sus sockets antes de terminar, cuando se pulse Ctrl-C.

¿Qué crees que ocurrirá si quitamos la llamada a exit() que hay en la función DetectadaINT()? Pruébalo. Pulsa Ctrl-C dos veces. Si has tenido éxito, será imposible terminar la aplicación pulsando Ctrl-C. Sin embargo

[1] A pesar de su nombre, el comando no es para matar procesos, sino para enviarles señales. El nombre se debe a que, si no especificamos otra cosa, la señal que enviará será SIGTERM, que sirve para pedirle al proceso que muera.

aún es posible terminarla si, desde otra terminal, usamos el comando `kill` una vez que averiguamos su `PID`. El comando `kill`, si no se le especifica qué señal enviar sino tan sólo el `PID` del proceso, enviará la señal `SIGTERM`. Con esta información no es difícil modificar el programa para que tampoco pueda ser terminado con `kill`.

Como ves, es posible que una aplicación cambie su comportamiento ante ciertas señales. No obstante, hay una señal que nunca puede reprogramarse. Se trata de la señal `SIGKILL`, que siempre causa la terminación del proceso, sin que éste pueda evitarlo. Es por esto que, si un proceso se "resiste a morir" con el comando `kill`, siempre se puede matar con el comando `kill -KILL`, o también `kill -9` (ya que cada señal tiene asignado un número, y `SIGKILL` es la número 9).

4.2. Creación de procesos

Una llamada a `fork()` causa que el proceso sea duplicado y ambas copias del mismo continúen su ejecución en el mismo punto (la instrucción siguiente a `fork()`), y con los mismos valores en todas sus variables, locales o globales. Si antes del `fork()` el proceso había abierto ficheros o sockets, el proceso hijo también tendrá acceso a éstos, a través de los descriptores que recibe del padre. Si ambos procesos van a escribir en el mismo fichero deben coordinarse para evitar hacerlo a la vez[2].

La única diferencia entre las dos copias del proceso es su número identificador (`PID`), y el valor que la función `fork()` retorna a cada copia. A la copia original (*padre*) le retorna cuál es el `PID` de su copia (*hijo*), mientras que al hijo le retorna cero. El hijo puede averiguar cuál es el `PID` de su padre llamando a la función `getppid()`. Un proceso también puede averiguar su propio `PID` llamando a la función `getpid()`.

Por ejemplo, el código `ej-fork.c` llama a la función `fork()` varias veces. Cada proceso resultante escribe cuál es su `PID` y el de su padre. Tras esto, todos los procesos se quedan "dormidos" (bloqueados) durante tres minutos. Lee el código fuente (tienes una copia en el anexo 5.2) e intenta adivinar cuántos procesos tendremos dormidos una vez hayan arrancado todos, antes de hacer la prueba.

Si ejecutas el programa, verás que todos los mensajes de todos los procesos salen entremezclados en la pantalla, tras lo cual todos se quedan bloqueados. Desde otra terminal podemos usar el comando `ps uxf` para ver el árbol de procesos a que hemos dado lugar, así como los `PID` de todos ellos. También podemos comprobar que todos están en estado `S` (*sleeping*). Por ejemplo, esto es lo que me sale a mi:

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
arriba   31966  0.0  0.0   9088  2868 ?        S    Nov16   0:00 sshd: arriba@pts/0
arriba   31967  0.0  0.0   3040  1756 pts/0    Ss   Nov16   0:00  \_ -bash
arriba   19578  0.0  0.0   2308   692 pts/0    R+   17:13   0:00      \_ ps uxf
arriba   10208  0.0  0.0   9232  2880 ?        S    11:09   0:00 sshd: arriba@pts/15
arriba   10209  0.0  0.0   2852  1612 pts/15   Ss   11:09   0:00  \_ -bash
arriba   19574  0.0  0.0   1360   332 pts/15   S+   17:13   0:00      \_ ./ej-fork
arriba   19575  0.0  0.0   1360   332 pts/15   S+   17:13   0:00          \_ ./ej-fork
arriba   19576  0.0  0.0   1360   332 pts/15   S+   17:13   0:00          |   \_ ./ej-fork
arriba   19577  0.0  0.0   1360   332 pts/15   S+   17:13   0:00          \_ ./ej-fork
```

El carácter "+" que aparece junto al estado `S` indica que estos procesos están en "primer plano", es decir, conectados a la terminal para su entrada. Dicho de otro modo, lo que teclee el usuario en esa terminal irá dirigido a éstos procesos. En particular, si pulsamos `Ctrl-C`, la señal `SIGINT` será enviada a todos los procesos, por lo que todos morirán más o menos a la vez.

Otra posibilidad es poner el carácter `&` al final de la línea desde la que lanzamos un programa. En este caso el programa queda ejecutándose en "segundo plano", lo que siginfica dos cosas: por un lado que no podemos enviarle datos desde el teclado a ese programa (y por tanto tampoco interrumpirlo con `Ctrl-C`; y por otro lado que la terminal no queda "bloqueada" por ese proceso, sino que el shell nos da un *prompt* desde el que podemos seguir tecleando comandos. Es decir, si lanzo el programa poniendo:

```
$ ./ej-fork &
```

Entonces en esa misma terminal puedo seguir escribiendo comandos. No necesito abrir otra. Por otro lado, si ahora hago `ps`:

```
$ ps uxf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

[2]Para esta sincronización se pueden usar *semáforos* o *cerrojos*, pero estos métodos se salen de los objetivos del curso. El lector curioso puede consultar la página de manual de `flock()` o de `lockf()` para informarse sobre los cerrojos.

```
arriba   31966  0.0  0.0  9088 2868 ?        S    Nov16   0:00 sshd: arriba@pts/0
arriba   31967  0.0  0.0  3040 1756 pts/0    Ss+  Nov16   0:00  \_ -bash
arriba   10208  0.0  0.0  9232 2880 ?        S    11:09   0:00 sshd: arriba@pts/15
arriba   10209  0.0  0.0  2852 1612 pts/15   Ss+  11:09   0:00  \_ -bash
arriba   19650  0.0  0.0  1352  324 pts/15   S    17:17   0:00      \_ ./ej-fork
arriba   19651  0.0  0.0  1352  324 pts/15   S    17:17   0:00          \_ ./ej-fork
arriba   19652  0.0  0.0  1352  324 pts/15   S    17:17   0:00          |   \_ ./ej-fork
arriba   19653  0.0  0.0  1352  324 pts/15   S    17:17   0:00          \_ ./ej-fork
```

Vemos que los procesos están en ejecución (durmiendo) y que ya no tienen un + en su estado, pues ya no están en primer plano. En cambio es bash el que tiene el +, ya que éste es ahora el proceso que recibe mis comandos en esa terminal.

4.2.1. Procesos zombie

Cuando un hijo termina por la razón que sea, debe devolverle al padre su *código de salida* (que es el valor que ha especificado en `exit()`, o bien cero si no ha llamado a `exit()`. El padre puede recuperar este valor llamando a la función `wait()`, pero si no lo hace, el hijo queda en un estado *zombie*. En este estado ya no se está ejecutando, ni ocupa memoria, pero el sistema operativo sigue manteniendo información sobre él en la tabla de procesos (en concreto, necesita mantener el código de salida, por si el padre lo pregunta). Con el comando `ps` estos procesos aparecen marcados con un estado `Z`.

Podemos hacer la prueba modificando el código anterior. Hazlo de tal forma que el padre de todos siga durmiendo sus tres minutos, pero los hijos en cambio no. Los hijos terminarán antes que el padre, pero ya que éste no recoge el *código de salida* de los hijos, éstos quedarán en estado *zombie*, lo que podrás comprobar con `ps`[3]:

```
arriba   10209  0.0  0.0  pts/15   Ss   11:09   0:00  \_ -bash
arriba   19934  0.0  0.0  pts/15   S+   17:27   0:00      \_ ./ej-fork-zombie
arriba   19935  0.0  0.0  pts/15   Z+   17:27   0:00          \_ [ej-fork-zombie] <defunct>
arriba   19937  0.0  0.0  pts/15   Z+   17:27   0:00          \_ [ej-fork-zombie] <defunct>
```

Observa que, cuando el proceso padre de los zombies haya muerto (tras tres minutos o antes si le enviamos una señal `SIGINT`), los zombies también desaparecerán.

4.2.2. Evitar los procesos zombie

Para evitar que se queden procesos zombies en el sistema, lo mejor es que el padre indique que no está interesado en recibir los *códigos de salida* de sus hijos. Para indicar esto basta añadir la siguiente línea cuando el padre arranque:

```
signal(SIGCHLD, SIG_IGN);
```

Esto asigna a la señal `SIGCHLD` el manejador `SIG_IGN`, que es un manejador especial que no hace nada con la señal, sino que simplemente la consume. Esto arregla el problema de los códigos de salida, porque estos son enviados al padre precisamente a través de esta señal. Al ignorarla de este modo, evitamos que estos códigos queden sin consumir.

Prueba a añadir esta modificación al código y verás como los hijos mueren y desaparecen mientras el padre queda durmiendo sus tres minutos.

5. Anexo: código fuente

5.1. espera.c

```
1  #include <stdio.h>
2  #include <string.h>
3  int main()
```

[3] ¿Por qué sólo quedan zombies dos procesos en lugar de los tres que hemos creado? O dicho de otro modo ¿por qué el proceso que fue creado a su vez desde un hijo no ha quedado zombie? La respuesta es que este proceso ha quedado además "huérfano", pues su padre está muerto (zombie). Hay siempre un proceso en el sistema (el proceso número 0, también llamado "init") que se ocupa de recoger los códigos de salida de los procesos huérfanos, por lo que éstos ya no quedan zombies.

```
{
  char txt[100];

  while(1) // Bucle infinito
    {
      printf("Escribe algo: ");
      fgets(txt, sizeof(txt), stdin);
      txt[strlen(txt)-1]=0;   // Quitarle el retorno de carro del final
      printf("Has escrito: '%s'\n", txt);
    }
}
```

5.2. ej-fork.c

```
// Este programa llama a fork varias veces para crear varios
// procesos. Cada proceso creado imprime su PID y el de su padre
// tras lo cual queda dormido tres minutos.
#include <unistd.h>
#include <stdio.h>

int main()
{
  int i;
  int hijo;

  printf("Soy el proceso padre\n");
  for (i=0; i<2; i++) {
    hijo=fork();
    if (hijo==0) printf("Proceso creado  : ");
    else         printf("Proceso original: ");
    printf("PID=%d, PPID=%d, fork retorna=%d\n",
           getpid(), getppid(), hijo);
  }
  // Ahora todos a dormir
  printf("El proceso %d va a dormir\n", getpid());
  sleep(180);
  printf("El proceso %d despierta y termina\n", getpid());
  return 0;
}
```

Índice