Fundamentos de Software de Comunicaciones
E.T.S.I. Telecomunicación

Prácticas de procesos (fork y exec*)

A lo largo de este guión se estudiarán los distintos mecanismos de multiproceso disponibles en Unix. El concepto de multiprocesamiento introduce un nuevo elemento, la **concurrencia**, y nuevos retos para la construcción de software de comunicaciones.

Por concurrencia entendemos la característica por la que muchos procesos se ejecutan de forma simultánea. Lógicamente, si el número de procesos en ejecución es superior al número de CPUs disponibles en el ordenador, se produce competencia entre los mismos, puesto que no todos se están ejecutando al mismo tiempo. Aparentemente, para el usuario, todas las tareas se ejecutan a la vez. Así, es responsabilidad del sistema operativo planificar los procesos para que se aproveche al máximo la capacidad de proceso y los recursos hardware y de memoria de la máquina.

Los entornos concurrentes introducen gran flexibilidad para los programadores. Por un lado, permiten elaborar estrategias de cooperación entre procesos, siendo posible alcanzar un objetivo en menor tiempo al dividirlo en tareas paralelas. Por otro lado, se optimiza el manejo de recursos limitados, como los dispositivos de entrada y salida. Este último punto es importante en comunicaciones: supongamos un proceso que está esperando la llegada de datos a través de la tarjeta de red. Precisamente porque no está haciendo nada más que esperar dichos datos, el denominado *planificador* del S.O. decide **bloquearlo**, de forma que lo saca de la CPU y reanuda la ejecución de otro proceso hasta la recepción de los datos esperados. En el momento en que haya datos listos desde la red, el S.O. volverá a activar al proceso para que los reciba y continúe su ejecución.

Esta alternancia entre procesos que entran a ejecutarse en la CPU provoca múltiples situaciones de interacción entre ellos, sobre todo si comparten algún tipo de recurso (como ficheros abiertos). No obstante, esto plantea problemas:

- Dos procesos no pueden manipular un recurso compartido a la vez, puesto se podría dar un problema de inconsistencia en los datos. Hay que asegurar que sólo un proceso está accediendo a un recurso compartido en cualquier momento de la ejecución. Las zonas de código que hay que proteger para que el acceso sea único se llaman de exclusión mútua o **secciones críticas** del proceso. El sistema operativo proporciona los denominados *semáforos* para eso.
- Dos procesos deben sincronizarse para cooperar: es el caso en que un proceso no pueda avanzar hasta que otro termine una tarea previa. Para esto también se emplean los semáforos.
- ¿Cómo se comparten datos entre dos o más procesos distintos? Recordemos que cada proceso tiene su propia tabla de descriptores, su pila y sus páginas de memoria independientes. El sistema operativo proporciona técnicas como las tuberías de datos, que son canales para intercambiar bytes entre procesos.

1. Creación de procesos desde un programa: exec y fork()

En Unix, existen dos formas de crear procesos desde un programa. Ambas formas son muy utilizadas en código de comunicaciones, por lo que se introducirán aquí para utilizarlas durante el resto del curso.

Funciones de la familia exec()

Existen diversas variantes de esta función, dependiendo de los argumentos que se utilicen. Su característica principal consiste en que `exec` ejecuta el proceso que se le pasa como parámetro. En ese momento, el proceso original que llama a la función es completamente sustituido por el nuevo proceso, que comienza su ejecución desde el principio.

Ejemplo con una función de la familia, denominada execv():

```
#include <unistd.h>
...
int ret;
char *cmd[] = { "ls", "-l", (char *)0 };

ret = execv ("/bin/ls", cmd);
/* si execv() ha tenido exito este codigo YA NO SE EJECUTA (este
proceso es reemplazado por el nuevo) */
if (ret == -1){
      /* pero si ha fallado la sustitucion de este proceso por otro,
se sigue por aqui */
}
```

La función execv() tiene dos argumentos:

- el primero es la cadena con el comando/programa ejecutable. Nótese que en el ejemplo se utiliza la ruta absoluta al comando ls (esta ruta se puede consultar a su vez utilizando este comando de consola: which ls)
- el segundo es un array de cadenas (o vector de cadenas, de ahí la v en la que termina execv) que incluye la información que va a recibir el nuevo comando a ejecutar como segundo argumento en su main. Es decir, las cadenas de cmd[] en el ejemplo son las que se van a recibir en el argv[] del main(int argc, char *argv[]) propio del nuevo comando. Nótese cómo la última casilla es cero, y la primera es el nombre del comando (en este caso solo el nombre sin la ruta)

Llamada al sistema fork(): procesos padre e hijo

Esta llamada al sistema crea un nuevo proceso hijo, que es una copia idéntica del padre que se está ejecutando:

```
pid_hijo = fork();
```

Es la única llamada al sistema que devuelve dos valores:

- Si estamos en el hijo, fork devuelve cero
- Si estamos en el padre, fork devuelve un valor mayor que cero (el pid del hijo)

La diferencia principal con las funciones `exec` es que aquí el proceso padre se sigue ejecutando (no es sustituido por el hijo). Ambos siguen ejecutándose simultáneamente a partir de la salida de la llamada a `fork()`. Por tanto, su valor de retorno sirve para distinguir código que únicamente ha de ser ejecutado en el padre o sólo en el hijo.

El hijo recibe una copia de memoria de la zona de datos y pila del padre, obteniendo así copias de los valores de las variables y de los ficheros abiertos previos al fork. Ojo, después de la llamada, ¡cada uno mantiene de forma independiente sus variables!

Ejemplo de código que usa fork(). NOTA: el código es común para un padre y un hijo, pero como se ha mencionado antes, habrá instrucciones que sólo debe ejecutar el padre y otras que sólo debe ejecutar el hijo (si no, no habría diferencia entre ellos):

```
main(){
      int pid_hijo;
      if( (pid_hijo = fork()) == -1){
            perror("No hay recursos!");
            exit(1);
      } else if (pid_hijo == 0) { /* Codigo del hijo */
      printf("Soy un hijo y debo terminar con exit!!\n");
            exit(0);
      } else {  /* Codigo del padre */
            printf("Soy un mal padre: no espero a mi hijo\n");
            exit(0);
      } /* else */} /* main */
```

Como regla, el hijo suele terminar su zona de código particular con exit():

```
exit(0); //valor cero siempre que termine correctamente
```

A su vez, el padre debe esperar la finalización de su hijo ya que, si no lo hace, el hijo queda como proceso ZOMBIE en el sistema operativo hasta que termine el padre.

```
...
      else {  /* Codigo del padre */

            printf("Soy un buen padre: espero a mi hijo");
            wait(0);
      } /* else */
```

Si el sistema operativo no detecta que el padre ha esperado al hijo mediante la llamada al sistema `wait()`, convierte al hijo en zombie, liberando sus recursos pero manteniendo su estado de terminación (esto hace que continúe en el sistema, ocupando un descriptor de procesos que podría asignarse a otro nuevo). Por tanto, esta situación hay que evitarla cuando el proceso padre vaya a ejecutarse más tiempo que cualquiera de sus hijos (ésta es la forma correcta de diseñar un programa) llamando siempre a `wait(0)`. Esta llamada al sistema hace que el proceso padre salga de la CPU en estado bloqueado, y no vuelva a ejecutarse hasta que alguno de sus hijos haya terminado. En ese momento el sistema operativo le lanza al padre la señal SIGCHLD (o SIGCLD).

En los Unix derivados de System V existe una manera de avisar al sistema operativo para que no genere zombies, ignorando directamente la señal SIGCLD (o SIGCHLD).

```
signal(SIGCLD,SIG_IGN);
```

Y, para el resto de versiones de Unix, se aconseja registrar un manejador para esta señal que incluya la llamada a wait(0) dentro:

```
void wait_hijos(int sig){
      wait(0);
      signal(SIGCLD,wait_hijos);
} /* wait_hijos */

int main(){
signal(SIGCLD,wait_hijos);
... //ahora se puede hacer el fork(), pero no antes del signal
} /* main */
```

Utilizar alguna de estas dos soluciones (o ignorar SIGCLD o bien manejarla e invocar al wait(0) en el manejador) nos permitirá no ejecutar directamente wait(0) dentro del código del padre, puesto que quedaría bloqueado e impediría ejecutar sus propias tareas, generar más hijos, etc., hasta que alguno de sus hijos lo desbloquease.

Ejercicios:

1. Implemente un programa que cree diez hijos usando *fork()*. El padre debe mantener la cuenta de cuántos hijos ha creado y cada hijo imprimirá su propio *pid* y el valor de número de hijos creado por su padre. El padre debe atender adecuadamente la muerte de sus hijos sin bloquearse en un wait.
    a. Haz que cada hijo imprima la información diez veces. ¿Los mensajes salen por orden de hijo creado? ¿Por qué? ¿Qué relación hay entre el *pid* del hijo y el valor del número de hijos que imprime ese hijo? ¿Cómo explicas esa relación?
    NOTA: utilice el esqueleto de código fork-esqueleto.c

2. Implemente un programa que cree 10 hijos. Si un hijo detecta que su pid es par, mostrará la fecha y hora del sistema. Si el pid del hijo es impar, éste mostrará el directorio actual en el que nos encontramos. Debes usar alguna función de la familia *exec*. ¿Por qué la función *perror* se usa justo después de la llamada a *exec*? ¿Qué nos indicaría si se ejecutara?
    NOTA: utilice el esqueleto de código execl-esqueleto.c

3. Implemente una función que emule a system(char *comando). Se llamará *fsc_system*, y tendrá el mismo argumento que la original. Nuestro fsc_system creará un hijo del proceso que mutará (usando alguna versión de exec-) para ejecutar una shell con el comando suministrado como argumento de entrada. Tras la creación del hijo, la función fsc_system se bloqueará en un wait() hasta que finalice la ejecución del comando en el hijo.
    NOTA: consulte la documentación de teoría disponible sobre exec-.

En los ejercicios se tendrá especial cuidado en:

- Controlar las señales y los errores en las llamadas al sistema