Sistemas Operativos Multimedia

Práctica 1.1

Creación de procesos

Uno de los objetivos de los sistemas operativos es hacer que el computador sea más fácil de utilizar. Para ello el sistema operativo ofrece un conjunto de operaciones que actúan a diferentes niveles (procesos, archivos, comunicación, etc.). Estas operaciones se pueden clasificar en dos grupos:

- Llamadas al sistema que proporcionan la interfaz a nivel del programador. Son operaciones básicas que facilita el sistema operativo para realizar programas. Por ejemplo, crear un proceso, abrir un archivo, crear un tubo, etc.

- Órdenes o comandos de shell que proporcionan la interfaz a nivel de usuario. Normalmente son programas más elaborados construidos a partir de las llamadas al sistema anteriores. Por ejemplo, copiar un archivo, crear un directorio, imprimir un archivo, etc.

Los sistemas operativos multiproceso permiten ejecutar varios procesos simultáneamente. Para ello deben proporcionar un conjunto de llamadas al sistema que no sólo puedan crear y eliminar procesos, sino también cambiar su imagen, esto es, cambiar su código, memoria, etc.

De la misma forma que el núcleo del sistema operativo trabaja mediante interrupciones para sincronizar-comunicar, a nivel de procesos, el sistema ofrece un mecanismo análogo basado en interrupciones software para trabajar con los procesos a nivel de usuario: las señales.

En esta práctica estudiaremos algunas de las llamadas al sistema relacionadas con la creación y terminación de procesos, cambio de imagen de un proceso y gestión de señales (enviar, programar, esperar, etc.). Hemos incorporado al final de la práctica un anexo que recoge en una tabla las llamadas al sistema Unix. Asimismo, hemos añadido otro anexo que describe sucintamente cómo se compila y ejecuta un programa en lenguaje C que es el que se emplea para desarrollar las prácticas a nivel de programador.

CREACIÓN DE PROCESOS

LLAMADA FORK

```
int fork(void)
```

En Unix, un proceso es creado mediante la llamada del sistema *fork*. El proceso que realiza la llamada se denomina proceso padre (*parent process*) y el proceso creado a partir de la llamada se denomina proceso hijo (*child process*). La sintaxis de la llamada efectuada desde el proceso padre es: `pid=fork();` La llamada *fork* se realiza una sola vez, pero retorna dos valores (correspondientes a los procesos padre e hijo). Las acciones implicadas por la petición de un *fork* son realizadas por el núcleo (*kernel*) del sistema operativo. Tales acciones son las siguientes:

1. asignación de un hueco en la tabla de procesos para el nuevo proceso (hijo).
2. asignación de un identificador único (PID) al proceso hijo.
3. copia de la imagen del proceso padre (con excepción de la memoria compartida).
4. asignación al proceso hijo del estado "preparado para ejecución".
5. dos valores de retorno de la función: al proceso padre se le entrega el PID del proceso hijo, y al proceso hijo se le entrega el valor cero.

A la vuelta de la llamada *fork*, los dos procesos poseen copias iguales de sus contextos a nivel usuario y sólo difieren en el valor del PID devuelto. La indicada relación de operaciones del *fork* se ejecuta en modo núcleo en el proceso padre. Al concluir, el planificador de procesos pondrá en ejecución un proceso que puede ser:

- el mismo proceso padre: vuelta al modo usuario y al punto en el que quedó al hacer la llamada *fork*.
- el proceso hijo: éste comenzará a ejecutarse en el mismo punto que el proceso padre, es decir, a la vuelta de la llamada *fork*.
- cualquier otro proceso que estuviese "preparado para ejecución" (en tal caso, los procesos padre e hijo permanecen en el estado "preparado para ejecución").

Las dos primeras situaciones se distinguen gracias al parámetro devuelto por *fork* (cero identifica al proceso hijo, no cero al proceso padre).

La ejecución del siguiente programa `creaproc.c` y la posterior petición de información sobre procesos en ejecución permitirá comprobar el funcionamiento de la llamada *fork*.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>


int main()
{
    pid_t pid;

    pid = fork();
    switch (pid)
    {
        case -1:
            printf ("No he podido crear el proceso hijo \n");
            break;
        case 0:
            printf ("Soy el hijo, mi PID es %d y mi PPID es %d \n",
                     getpid(), getppid());
            sleep (20);
            break;
        default:
            printf ("Soy el padre, mi PID es %d y el PID de mi hijo es %d \n",
                     getpid(), pid);
            sleep (30);
    }
    printf ("Final de ejecución de %d \n", getpid());
    exit (0);
}
```

Para ejecutar el programa en modo background, y así poder verificar su ejecución con la orden `ps`, se debe añadir al nombre del programa el carácter & (ampersand):

```
$ gcc –o creaproc creaproc.c
$ creaproc &
Soy el hijo, mi PID es 944 y mi PPID es 943
Final de ejecución de 944
Soy el padre, mi PID es 943 y el PID de mi hijo es 944
Final de ejecución de 943
$ ps
  PID TTY          TIME CMD
  893 pts/1    00:00:00 bash
  943 pts/1    00:00:00 creaproc
  944 pts/1    00:00:00 creaproc
  946 pts/1    00:00:00 ps
$
```

Dos procesos vinculados por una llamada *fork* poseen zonas de datos propias, de uso privado (no compartidas). La ejecución del siguiente programa, en el cual se asigna distinto valor a una misma variable según se trate de la ejecución del proceso padre o del hijo, permitirá comprobar tal característica de la llamada *fork*.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main ( ) {
     int i,j;
     pid_t pid;

     pid = fork( );
     switch (pid) {
     case -1:
          printf ("\nNo he podido crear el proceso hijo");
          break;
     case 0:
          i = 0;
          printf ("\nSoy el hijo, mi PID es %d y mi variable i
(inicialmente a %d) es par", getpid(), i);
          for ( j = 0; j < 5; j ++ ) {
               i ++;
               i ++;
               printf ("\nSoy el hijo, mi variable i  es %d", i);
               };
         break;
     default:
         i = 1;
         printf ("\nSoy el padre, mi PID es %d y mi variable i
(inicialmente a %d) es impar", getpid(), i);
         for ( j = 0; j < 5; j ++ ) {
              i ++;
              i ++;
              printf ("\nSoy el padre, mi variable i  es %d", i);
              };
     };
     printf ("\nFinal de ejecucion de %d \n", getpid());
     exit (0);
}
```

En este programa, el proceso padre visualiza los sucesivos valores impares que toma su variable i privada, mientras el proceso hijo visualiza los sucesivos valores pares que toma su variable i privada y diferente a la del proceso padre.

```
$ gcc –o creaproc1 creaproc1.c
$ creaproc1 &

Soy el padre, mi PID es 956 y mi variable i (inicialmente a 1) es impar
Soy el padre, mi variable i  es 3
Soy el padre, mi variable i  es 5
Soy el padre, mi variable i  es 7
Soy el padre, mi variable i  es 9
Soy el padre, mi variable i  es 11
Final de ejecucion de 956

Soy el hijo, mi PID es 957 y mi variable i (inicialmente a 0) es par
Soy el hijo, mi variable i  es 2
Soy el hijo, mi variable i  es 4
Soy el hijo, mi variable i  es 6
Soy el hijo, mi variable i  es 8
Soy el hijo, mi variable i  es 10
Final de ejecucion de 957
$
```

Las variables descriptores, asociadas a archivos abiertos en el momento de la llamada *fork*, son compartidas por los dos procesos existentes a la vuelta del *fork*. Es decir, los descriptores de archivos en uso por el proceso padre son heredados por el proceso hijo generado.

Un proceso no sabe a priori su nombre y ni el de su padre (aunque lo tenga en sus estructuras internas). Para obtener el identificador de un proceso o el de su padre hay que acudir a las llamadas getpid() y getppid().

LLAMADA GETPID

int getpid(void)

La llamada `getpid()` devuelve el número de identificación de proceso (PID) del proceso actual. Devuelve –1 en caso de error y el PID del proceso en caso de éxito.

LLAMADA GETPPID

int getppid(void)

La llamada `getppid()` devuelve el número de identificación de proceso (PID) del proceso padre al proceso actual (el PID del proceso que creó al actual). Devuelve –1 en caso de error y el PID del padre en caso de éxito.

TERMINACIÓN DE PROCESOS

LLAMADA EXIT

void exit(int estado)

La llamada `exit()` hace finalizar un proceso con estado `estado`. Este estado es un valor entero que se retorna al padre del proceso finalizado. Para que el padre puede leer este valor, deberá realizar una llamada `wait()`.

Si un proceso finaliza sin ejecutar la llamada `exit()`, por ejemplo, al realizar un retorno de la función `main()`, su estado de finalización será indefinido.

El sistema tiene una variable de entorno, denominada `?`, donde puede recogerse el estado de finalización de la ejecución de la última orden. Para ver el estado de finalización del último hijo de shell (última orden) hay que introducir

```
$ echo $?
...
```

A lo que el sistema contestará con 0, 2, o cualquier estado posible.

LLAMADA WAIT

*int wait(int *estado)*

int wait(NULL)

La llamada `wait()` hace que el proceso actual quede detenido en espera hasta que muere alguno de sus hijos o recibe una señal.

`wait()` devuelve –1 en caso de error y el PID del hijo que muere en caso contrario.

Un proceso puede finalizar por dos causas: una terminación explícita mediante la ejecución de una llamada `exit()`, o bien porque otro proceso lo mató enviándole una señal.

El parámetro `estado` es un entero en el que `wait()` escribe información dividida en dos partes:

- Si el hijo termina con una llamada `exit(estado)`, en los 7 bits menos significativos `(estado & 0x7F)` escribe 0 y en los 8 bits más significativos `((estado>>8) & 0xFF)` escribe el valor `estado`.
- Si el hijo termina matado por una señal, en los 7 bits menos significativos `(estado & 0x7F)` escribe el número de señal.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main ( ) {
   int estado, numero;

   switch(fork()) {
      case -1 : /* ERROR */
                perror("Error en fork");
                exit(1);
      case 0  : /* HIJO */
                numero = 13;
                printf("Soy el hijo y muero con %d...\n", numero);
                sleep(20);
                exit(numero);
      default : /* PADRE */
                wait(&estado);
                printf("Soy el padre. ");
                if ((estado & 0x7F) != 0) {
                   printf("Mi hijo ha muerto con una señal.\n");
                }
                else {
                   printf("Mi hijo ha muerto con exit(%d).\n",
                          (estado>>8) & 0xFF);
                }
                exit(0);
    }
}
```

Si compilamos y ejecutamos obtenemos lo siguiente:

```
$ gcc -o creaproc3 creaproc3.c
$ creaproc3
Soy el hijo y muero con 13...
Soy el padre. Mi hijo ha muerto con exit(13).
$ creaproc3 &
[1] 1006
Soy el hijo y muero con 13...
$ ps
UID        PID  PPID  C STIME TTY          TIME CMD
alu        893   891  0 18:06 pts/1    00:00:00 /bin/bash
alu       1006   893  0 18:34 pts/1    00:00:00 creaproc3
alu       1007  1006  0 18:34 pts/1    00:00:00 creaproc3
alu       1008   893  0 18:34 pts/1    00:00:00 ps -f
$ kill -9 1007
Soy el padre. Mi hijo ha muerto con una señal.
$
```

ANEXO I

TABLA DE LLAMADAS AL SISTEMA

Llamada	Categoría	Descripción
`alarm()`	Comunicación	Programa una alarma con un plazo de un cierto número de segundos.
`kill()`	Comunicación	Envía una señal a un proceso.
`pause()`	Comunicación	Detiene la ejecución indefinidamente (hasta la llegada de una señal).
`pipe()`	Comunicación	Crea una tubería y proporciona sus descriptores.
`signal()`	Comunicación	Programa un manejador para una señal.
`chdir()`	Archivos	Cambia el directorio por defecto (directorio actual).
`chmod()`	Archivos	Cambia los permisos de un archivo.
`chown()`	Archivos	Cambia el propietario de un archivo.
`close()`	Archivos	Cierra un archivo.
`creat()`	Archivos	Crea un archivo y lo abre para escritura.
`dup()`	Archivos	Duplica un descriptor de archivo en la siguiente entrada libre de la tabla de archivos.
`dup2()`	Archivos	Duplica un descriptor de archivo en la entrada de la tabla de archivos que se indique.
`fstat()`	Archivos	Obtiene información sobre un archivo abierto: tamaño, fecha, permisos, etc.
`link()`	Archivos	Crea un enlace (duro).
`lseek()`	Archivos	Sitúa el puntero de lectura/escritura en cualquier posición de un archivo.
`mkdir()`	Archivos	Crea un directorio.
`mknod()`	Archivos	Crea un dispositivo especial.
`mount()`	Archivos	Monta un dispositivo en un directorio.
`open()`	Archivos	Abre un archivo y proporciona un descriptor.
`read()`	Archivos	Lee datos de un archivo.
`rmdir()`	Archivos	Elimina un directorio vacío.
`stat()`	Archivos	Obtiene información sobre un archivo: tamaño, fecha, permisos, etc.
`symlink()`	Archivos	Crea un enlace simbólico.
`umount()`	Archivos	Desmonta un dispositivo.
`unlink()`	Archivos	Elimina un enlace duro de un archivo (lo borra del disco).
`utime()`	Archivos	Modifica la fecha y hora de acceso y modificación de un archivo.
`write()`	Archivos	Escribe datos de un archivo.
`getgid()`	Otras	Proporciona el identificador de grupo del usuario actual (GID).
`getuid()`	Otras	Proporciona el identificador del usuario actual (UID).
`setgid()`	Otras	Cambia el identificador de grupo del usuario actual (GID).
`setuid()`	Otras	Cambia el identificador del usuario actual (UID).
`time()`	Otras	Proporciona el número de segundos desde el 1 de enero de 1970 a las 0 horas.
`times()`	Otras	Proporciona el tiempo consumido en la ejecución del proceso actual.
`exec`*`XX`*`()`	Procesos	Cambia el programa actual por el programa en disco indicado (ejecuta un programa).
`exit()`	Procesos	Termina el proceso actual indicando un estado de finalización.
`fork()`	Procesos	Crea un proceso hijo (desdobla el proceso actual).
`getpid()`	Procesos	Proporciona el PID del proceso actual.

Llamada	Categoría	Descripción
`getppid()`	Procesos	Proporciona el PID del proceso padre.
`nanosleep()`	Procesos	Detiene la ejecución en espera pasiva durante un número de nanosegundos.
`sleep()`	Procesos	Detiene la ejecución en espera pasiva durante un número de segundos.
`usleep()`	Procesos	Detiene la ejecución en espera pasiva durante un número de microsegundos.
`wait()`	Procesos	Espera por la muerte de un hijo (cualquiera) y obtiene su estado de finalización.

ANEXO II

UTILIZACIÓN DEL COMPILADOR DE C

El compilador de C en los sistemas UNIX es un programa llamado cc (`gcc` en Linux) que funciona como un comando más, sin editor ni entorno. Los archivos fuente y las opciones de compilación deben pasarse como parámetro. Los errores encontrados se imprimen por la salida de errores.

El programa `cc` realiza todos los pasos de compilación en C: preproceso, compilación a objeto y enlace. Por ello es necesario incluir en la llamada al compilador todos los archivos fuente que intervienen en la compilación, así como las bibliotecas necesarias.

Habitualmente se construyen archivos Makefile para facilitar la tarea de compilación. Los archivos Makefile son archivos de texto que contienen una descripción formal de los pasos necesarios para recompilar un sistema complejo, de manera que, con introducir el comando `Make`, el compilador es invocado con los parámetros adecuados.

El funcionamiento básico del compilador consiste en invocar el comando cc indicando únicamente el nombre del archivo fuente:

```
$ gcc fuente.c
```

Este comando realiza la compilación del archivo `fuente.c` y produce un programa ejecutable llamado `a.out`. Este es el nombre del programa ejecutable por defecto. Para que el ejecutable generado tenga otro nombre podemos utilizar la opción `-o` como en el siguiente ejemplo:

```
$ gcc –o salida fuente.c
```

Ahora el ejecutable creado se llamará `salida`. En los ejemplos anteriores se han utilizado las bibliotecas del sistema para enlazar. Si son necesarias otras bibliotecas, podemos utilizar la opción `-l` para indicar que se incluyan, como en el siguiente ejemplo:

```
$ gcc –o salida fuente.c -lm
```

En este caso `-lm` indica que se debe enlazar con la biblioteca matemática estándar. Las principales bibliotecas estándar se recogen en la siguiente tabla:

Nombre	Uso
`c`	Biblioteca estándar del C. No hace falta incluirla explícitamente
`m`	Biblioteca estándar matemática
`curses`	Contiene rutinas de visualización con ventanas, etc.
`l`	Biblioteca para uso del *lex* y el *yacc*

Cuando el programa fuente esté compuesto por más de un archivo, deberá especificarse la lista completa de archivos que lo componen. En esta lista pueden aparecer archivos fuentes en C, ensamblador y archivos objeto. Por ejemplo:

```
$ gcc –o programa principal.c modulo1.c modulo2.c utilidades.o
```

Esta instrucción compilará los archivos fuente en C `principal.c`, `modulo1.c` y `modulo2.c` junto con el código objeto de `utilidades.o` y generará un ejecutable llamado `programa`. Es muy importante especificar correctamente la extensión de cada archivo, ya que el compilador interpretará su contenido a través de la extensión. Por ejemplo, para compilar un archivo en C, obligatoriamente la extensión debe ser `.c`. Se suelen utilizar las extensiones de la siguiente tabla para nombrar los diferentes tipos de archivos:

Extensión	Tipo
`.c`	archivo fuente en C
`.h`	archivo cabecera fuente en C (para los `#include`)
`.s`	archivo fuente en ensamblador
`.o`	archivo objeto
`.a`	archivo de biblioteca

REFERENCIAS

BIBLIOGRAFÍA

- F. M. Márquez: UNIX. Programación avanzada, Rama, 2004.
- K. A. Robbins y S. Robbins: UNIX. Programación práctica, Prentice may, 2000.