Sistemas Operativos
Práctica 3

Ing. Andrés Bustamante
afbustamanteg@unal.edu.co
Ingeniería de Sistemas
Facultad de Ingeniería
Universidad de la Amazonia

2009

1. Objetivo

El objetivo de la práctica es que el estudiante aprenda el manejo de hilos y procesos en UNIX y Linux. Particularmente se espera que comprenda las rutinas que se utilizan para su creación y gestión.

2. Metodología

Mediante el uso del lenguaje de programación C, el estudiante creará una pequeña aplicación que permita crear hilos para realizar determinadas funciones. De igual manera, para la parte de procesos se creará un programa que permita la creación de un proceso pesado. Para tal fin se le instruirá en las funciones básicas para dichos manejos. De igual manera se recomienda consultar las páginas del manual de Linux (man) para solucionar dudas más específicas.

3. Marco teórico

3.1. Programación con hilos

Los sistemas operativos basados en UNIX y en el estándar POSIX de IEEE, como por ejemplo Linux, disponen y aceptan una biblioteca multiplataforma para la manipulación de hilos en lenguaje C. Esta biblioteca se conoce como `pthread`. Para trabajar con hilos en programas en lenguaje C entonces hay que incluir un nuevo encabezado con la biblioteca pthread.h así:

```
#include <pthread.h>
```

Esta biblioteca permite el uso de funciones como las que se presentan en la figura 3.1.

Por ejemplo, para crear un nuevo hilo, la función que crea e inicia la ejecución de un nuevo hilo (lo pone en la cola del planificador) tiene esta estructura:

```
int pthread_create(pthread_t *nuevo_hilo, const pthread_attr_t *atributos,
void *<NOMBRE_FUNCION> (void *), void *arg)
```

El parámetro `nuevo_hilo` apunta al identificador del hilo que se crea. Este identificador se utiliza en la declaración previa del hilo del tipo `pthread_t`. Los atributos del hilo (p.ej. tamaño de pila o política de planificación) se encapsulan en el objeto al que apunta `atributos`, normalmente este parámetro es `NULL`. El tercer parámetro `<NOMBRE_FUNCION>` se refiere al nombre de la función que ejecutará el hilo.

Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figura 1: Algunas llamadas o funciones de la biblioteca pthread en C

Cuando la función devuelva el control implícitamente, el hilo finalizará. Si la función necesita un parámetro (el único), se especifica con `arg` (o `NULL` si no lo tiene).

Veamos un ejemplo:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

void h1()
{
  int i = 0;
  int aleatorio;
  while (i < 10)
  {
    printf("%s%d\n", "Hilo 1, iteracion ", i + 1);
    i++;
    aleatorio = rand();
    sleep(1 + (aleatorio % 5));
  }
}

void h2()
{
  int i = 0;
  int aleatorio;
  while (i < 10)
  {
    printf("%s%d\n", "Hilo 2, iteracion ", i + 1);
    i++;
    aleatorio = rand();
    sleep(1 + (aleatorio % 5));
  }
}
```

```
int main(int argc, char *argv[])
{
  srand((unsigned)time(NULL));
  setbuf(stdout, NULL);
  int error1, error2;
  pthread_t tp1;
  pthread_t tp2;
  error1 = pthread_create (&tp1, NULL, (void *)h1, NULL);

  if (error1) printf("\n", error1);

  error2 = pthread_create (&tp2, NULL, (void *)h2, NULL);

  if (error2) printf("\n", error2);

  pthread_join (tp1, NULL);
  pthread_join (tp2, NULL);

  printf("%s\n", "Ambos hilos finalizaron");
  return (0);
}
```

Nota: Para compilar un programa en C que utiliza la biblioteca de hilos, ponemos:

```
$ cc -lpthread programa.c -o programa
```

3.2. Programación con procesos

El sistema operativo Linux dispone de un conjunto de llamadas al sistema que definen una poderosa interfaz para la programación de aplicaciones (API) que involucren múltiples procesos; abriendo las puertas a la programación concurrente. Esta interfaz suministra herramientas al desarrollador de software, tanto para la creación, sincronización y comunicación de nuevos procesos, como la capacidad de ejecutar nuevos programas.

Entre los aspectos más destacados de la gestión de procesos en UNIX/Linux se encuentra la forma en que estos se crean y cómo se ejecutan nuevos programas. Aunque más adelante se mostrarán las correspondientes llamadas al sistema, en esta práctica es conveniente presentar una visión inicial conjunta, que permita entender mejor la forma en que estas llamadas se utilizan.

Cuando el kernel crea un nuevo proceso, proceso hijo, hace una copia (clonación) del proceso que realiza la llamada al sistema `fork` (proceso padre). Así, salvo el PID y el PPID, los dos procesos serán inicialmente idénticos. El proceso hijo obtiene una copia de los recursos del padre (hereda su entorno).

Sin embargo, para ejecutar un nuevo programa, uno de los procesos ha de realizar otra llamada al sistema: `exec`, que reinicia (recubre) sus segmentos de datos de usuario e instrucciones a partir de un programa en disco. En este caso no aparece ningún proceso nuevo.

Cuando un proceso termina (muere), el sistema operativo lo elimina recuperando sus recursos para que puedan ser usados por otros procesos.

3.2.1. Creación de procesos

Sintaxis: `pid_t fork (void)`

`fork` crea un nuevo proceso, pero no inicia un nuevo programa. Los segmentos de datos de usuario, de sistema y el segmento de instrucciones del nuevo proceso (hijo) son copias casi exactas del proceso que realizó la llamada (padre).

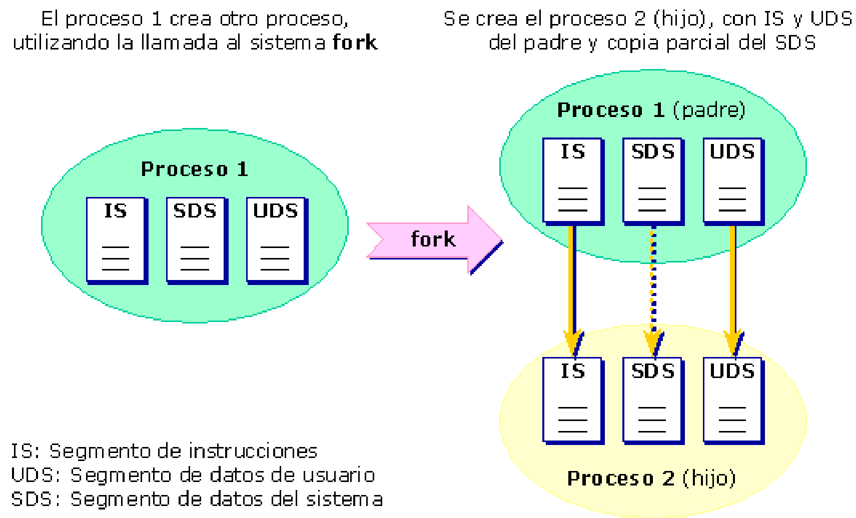

Figura 2: Esquema de la creación de procesos en UNIX y Linux: Creación de procesos hijos

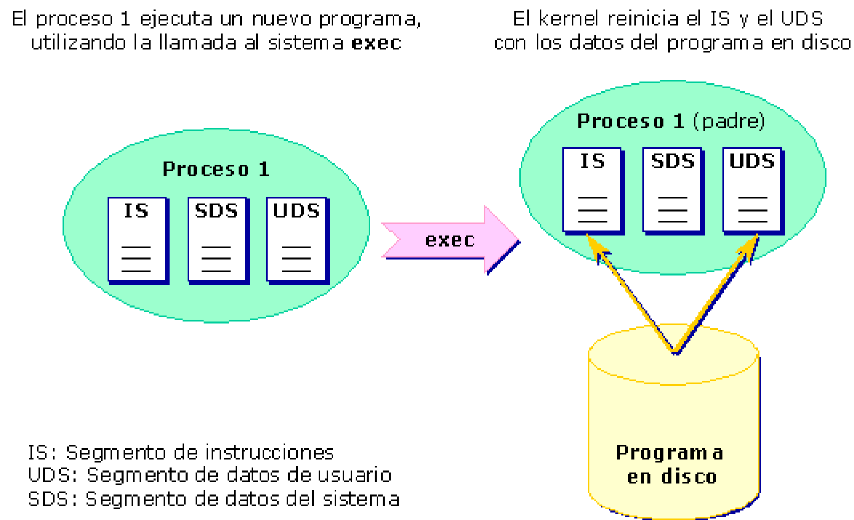

Figura 3: Esquema de la creación de procesos en UNIX y Linux: Independencia de ejecución del hijo

El valor de retorno de `fork` es:

Proceso hijo:	0
Proceso padre:	PID del proceso hijo
Si fork falla:	-1

El proceso hijo hereda la mayoría de los atributos del proceso padre, ya que se copian de su segmento de datos del sistema. Sólo los siguientes atributos difieren entre padre e hijo:

- **PID:** Identificador del proceso
- **Archivos:** El hijo obtiene una copia de la tabla de descriptores de archivo del proceso padre, con lo que comparten el puntero de un archivo. Si uno de los procesos cambia ese puntero (realiza una operación de E/S), la siguiente operación de E/S realizada por el otro proceso se hará a partir de la modificación realizada por el primer proceso. Sin embargo, al existir dos tablas de descriptores de archivos si un proceso cierra su descriptor, el otro no se ve afectado.

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

main (int argc, char *argv[])
{
  int pidHijo;
  printf ("Ejemplo de fork. Este proceso va a crear otro proceso\n");

  if (pidHijo=fork()) // Código ejecutado por el padre
    printf ("Proceso PADRE: He creado un proceso con PID %i\n", pidHijo);
  else  // Código ejecutado por el hijo
    printf ("Proceso HIJO: Mi PID es %i\n", pidHijo);
  // Esta línea es ejecutada por los dos procesos */
  printf ("Fin del proceso cuyo pidHijo vale %i\n", pidHijo);
}
```

Ejecución:

```
Ejemplo de fork. Este proceso va a crear otro proceso
Proceso HIJO: Mi PID es 0
Proceso PADRE: He creado un proceso con PID 23654
Fin del proceso cuyo pidHijo vale 23654
Fin del proceso cuyo pidHijo vale 0
```

3.2.2. Terminación de procesos

exit: Sintaxis: `void exit (int status)`

`exit` finaliza al proceso que lo llamó, con un código de estado igual al byte inferior del parámetro `status`. Todos los descriptores de archivo abiertos son cerrados y sus buffers sincronizados. Si hay procesos hijo cuando el padre ejecuta un `exit`, el PPID de los hijos se cambia a 1 (proceso `init`). Es la única llamada al sistema que nunca retorna.

El valor del parámetro `status` se utiliza para comunicar al proceso padre la forma en que el proceso hijo termina. Por convenio, este valor suele ser 0 si el proceso termina correctamente y cualquier otro valor en caso de terminación anormal. El proceso padre puede obtener este valor a través de la llamada al sistema `wait`.

wait: Sintaxis: `pid_t wait (int *statusp)`

Si hay varios procesos hijos, `wait` espera hasta que uno de ellos termine. No es posible especificar por qué hijo se espera. `wait` retorna el PID del hijo que termina (o -1 si no se crearon hijos o si ya no hay hijos por los que esperar) y almacena el código del estado de finalización del proceso hijo (parámetro status en su llamada al sistema exit) en la dirección apuntada por el parámetro `statusp`.

Un proceso puede terminar en un momento en el que su padre no le esté esperando. Como el kernel debe asegurar que el padre pueda esperar por cada proceso, los procesos hijos por los que el padre no espera se convierten en procesos *zombies* (se descartan su segmentos, pero siguen ocupando una entrada en la tabla de procesos del kernel). Cuando el padre realiza una llamada `wait`, el proceso hijo es eliminado de la tabla de procesos. No es obligatorio que todo proceso padre espere a sus hijos.

3.2.3. Ejecución de nuevos procesos

La llamada al sistema `exec` permite reemplazar los segmentos de instrucciones y de datos de usuario por otros nuevos a partir de un archivo ejecutable en disco, con lo que se consigue que un proceso deje de ejecutar instrucciones de un programa y comience a ejecutar instrucciones de un nuevo programa. `exec` no crea ningún proceso nuevo.

Como el proceso continua activo, su segmento de datos del sistema apenas es perturbado, la mayoría de sus atributos permanecen inalterados. En particular, los descriptores de archivos abiertos permanecen abiertos después de un `exec`. Esto es, importante puesto que algunas funciones del lenguaje C (como `printf`) utilizan buffers internos para aumentar el rendimiento de la E/S; si un proceso realiza un `exec` y no se han volcado (sincronizado) antes los buffers internos, los datos de estos buffers se perderán. Por ello, es habitual cerrar los descriptores abiertos antes de realizar una llamada al sistema `exec`.

Hay seis formas de realizar una llamada al sistema `exec`:

```
execl (const char *path, const char *arg1, ...)
execlp (const char *file, const char *arg, ...)
execle (const char *path, const char *arg , ..., char * const envp[])
execv (const char *path, char *const argv[])
execvp (const char *file, char *const argv[])
execve (const char *filename, char *const argv [], char *const envp[])
```

El resultado de la llamada al sistema `exec` es un entero (-1), que sólo esta disponible si la llamada falla.

4. Práctica

1. Se deberá realizar un programa que contenga 4 hilos, cada uno de los cuales generará aleatoriamente 10 valores enteros entre 1 y 100. Entre cada generación deberá haber una espera aleatoria entre 1 y 3. El valor generado por cada hilo se deberá sumar en una variable global, de modo que al finalizar la ejecución de todos los hilos, esta variable contenga la suma total de los valores generados por los 4 hilos. Se debe tener en cuenta que cada hilo deberá informar del número que él genere mostrándolo en la pantalla. Tenga en cuenta que cada vez que un hilo muestre información en pantalla deberá identificarse.

2. Realice un programa que cree un nuevo proceso. El proceso padre generará y mostrará en pantalla los números pares hasta 100, mientras que el proceso hijo generará y mostrará en pantalla los números impares hasta 100. Al igual que en el programa de los hilos, cada vez que un proceso muestre información en pantalla deberá identificarse.

5. Entregables

1. Se debe entregar informe únicamente en formato PDF con la descripción del proceso realizado de manera detallada. Se recomienda realizar capturas de pantalla describiendo los **resultados** obtenidos. También se deben incluir conclusiones del aprendizaje de la práctica y bibliografía consultada (si es el caso).

2. También se debe entregar el archivo con el código fuente del programa elaborado en lenguaje C. Se recomienda realizar comentarios en el código para documentar el programa y colocar el nombre de los integrantes del grupo al principio del archivo de código fuente (como comentario).

Los entregables se deben enviar vía correo electrónico a la dirección electrónica del profesor.