Procesos e Hilos en C

6 de febrero de 2012

En esta sesión vamos a escribir programas en lenguaje C que utilicen hilos y procesos para comparar el rendimiento del sistema ante la gestión de unos y otros. En concreto, mediremos el tiempo medio de creación de hilos y el de creación de procesos y se compararán. Para ello se va a utilizar el compilador de C de GNU **gcc** que está instalado en los equipos del laboratorio.

1. Creación de procesos en Linux

Para poder iniciar en una máquina GNU/Linux, procesos adicionales al que se está ejecutando, se emplea el comando `fork` cuya signatura es la siguiente:

```
pid_t fork(void);
```

Su definición se encuentra en los archivos de cabecera **sys/types.h** y **unistd.h**. En **unistd.h** está definida la función y en **types.h** el tipo `pid_t` que no es más que un identificador de proceso linux PID.

La ejecución de `fork` crea un proceso hijo que se diferencia de su creador únicamente por su PID y por su PPID (Parent PID, identificador del proceso padre del proceso actual).

Tras la creación del proceso hijo ambos procesos son ejecutados concurrentemente, es decir, tenemos dos procesos ejecutando el código del programa. La diferencia entre los dos se haya en lo que devuelve la llamada al `fork`. En cada uno de esos procesos este valor es diferente, para el proceso padre fork devuelve el PID del proceso recién creado y para el proceso recién creado `fork` devuelve 0, además el PPID del proceso recién creado es el PID del proceso padre.

Para conocer el PID y el PPID de un proceso se pueden emplear las funciones `getpid` y `getppid` respectivamente. Nuevamente estas funciones están definidas en el archivo de cabecera unistd.h con la siguiente signatura:

```
pid_t getpid (void);
pid_t getppid (void);
```

El listado1 contiene un ejemplo de uso de **fork**:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  pid_t pid;

  pid = fork();
  if(pid==-1) {
    printf("Fallo en fork\n");
    return -1;
  } else
  if (!pid) {
    printf("Proceso hijo: PID %d\n", getpid());
  } else {
    printf("Proceso padre: PID %d\n", getpid());
  }
  return 0;
}
```

Listado 1: Ejemplo de creación de procesos en c

Ejercicio 1 *Escribe el programa anterior y comprueba su funcionamiento.*

2. pthread.lib: La librería de hilos POSIX de Linux

Para escribir programas multihilo en C podemos utilizar la librería pthread que implementa el standard POSIX (Portable Operating System Interface). Para ello en nuestro programa debemos incluir algunas cabeceras y a la hora de compilar es necesario linkar el programa con la librería. Veamos con un ejemplo lo que estamos diciendo. Hagamos un programa que escriba "Hola Mundo" utilizando para ello dos hilos distintos, uno para la palabra Hola y otro para la palabra Mundo. Además, hagamos que cada palabra se escriba muy lentamente para ver bien el efecto.

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

void *hola(void *arg) {
  char *msg = "Hola";
  int i;

  for ( i = 0 ; i < strlen(msg) ; i++ ) {
    printf("%c", msg[i]);
    fflush(stdout);
    usleep(1000000);
  }
  return NULL;
}


void *mundo(void *arg) {
  char *msg = " mundo";
  int i;

  for ( i = 0 ; i < strlen(msg) ; i++ ) {
    printf("%c", msg[i]);
    fflush(stdout);
    usleep(1000000);
  }
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t h1;
  pthread_t h2;

  pthread_create(&h1, NULL, hola, NULL);
  pthread_create(&h2, NULL, mundo, NULL);
  printf("Fin\n");
}
```

Listado 2: Ejemplo de creación de hilos en c

como se puede ver en el listado 2, hemos escrito una función para cada tarea que queremos realizar en un hilo. El estandard POSIX impone que toda función que vaya a ejecutarse en un hilo debe tener un parámetro de entrada de tipo puntero a void (void *) y tiene que devolver también un puntero a void. Las funciones imprimen un mensaje cada una, pero lo hacen letra a letra. Además, tras cada impresión se bloquean durante un tiempo usando la función `usleep(ms)`, la cual detiene el hilo que la ejecuta durante `ms` microsegundos.

En el programa principal tenemos dos variables de tipo `pthread_t` que van a almacenar el identificador de cada uno de los dos hilos que vamos a crear. El identificador de un hilo es necesario guardarlo ya que, una vez que un hilo comienza a funcionar, la única forma de controlarlo es a través de su identificador. El tipo `pthread_t` está definido en la cabecera **pthread.h**, por eso es necesario incluirla al principio del programa.

El lanzamiento o creación de los hilos se realiza con la función `pthread_create` que también está definida en **pthread.h**. Esta función crea un hilo, inicia la ejecución de la función que se le pasa como tercer argumento dentro de dicho hilo y guarda el identificador del hilo en la variable que se le pasa como primer argumento. Por lo tanto, el primer argumento será la dirección (&) de la variable de tipo `pthread_t` que queramos que guarde el identificador del hilo creado. El tercer argumento será el nombre de la función que queramos que se ejecute dentro de dicho hilo.

El segundo parámetro se puede utilizar para especificar atributos del hilo, pero si usamos NULL el hilo se crea con los parámetros normales. Si deséas más información sobre los atributos de los hilos pregunta al profesor. Más adelante veremos la utilidad del cuarto parámetro.

EJERCICIO 2 *Escribe el programa anterior, compílalo y ejecútalo. Ten en cuenta que al compilar has de linkar con la librería pthread por lo que hay que añadir a la llamada a gcc el parámetro* `-lpthread`*. Si el resultado no es el esperado, continúa leyendo el siguiente apartado.*

2.1. Esperando a la terminación de un hilo

Al ejecutar el programa anterior, es bastante improbable que se llegue a ver nada en la consola. La razón es que los hilos que se crean en el programa principal terminan automáticamente cuando el programa principal termina. Si en el programa principal símplemente lanzamos los dos hilos y después escribimos "Fin", lo más probable es que los hilos no lleguen a ejecutarse complétamente o incluso que no lleguen ni a terminar de arrancar antes de que el programa principal escriba y termine.

Evidentemente es necesario un mecanismo de sincronización que permita esperar a la terminación de un hilo. Este mecanismo es el que implementa la función `pthread_join` que también está definida en **pthread_h**. Para comprobarlo añadamos las siguientes líneas al programa anterior justo antes de la sentencia para escribir "Fin" en el programa principal.

```
pthread_join(h1, NULL);
pthread_join(h2, NULL);
```

como se puede ver, `pthread_join` espera como primer argumento el identificador del hilo que se quiere esperar. En el segundo argumento se puede especificar una variable donde queremos que se deje el valor devuelto por la función que se ejecuta en el hilo cuando termine. Normalmente no vamos a devolver nada así que no es necesario utilizarlo, pero si deseas conocer más sobre esto pregunta al profesor.

EJERCICIO 3 *Escribe las modificaciones comentadas en el programa y comprueba cómo la palabra Fin aparece tras la terminación de los dos hilos.*

2.2. Pasando parámetros a los hilos

Las funciones que se ejecutan en los hilos tienen acceso a las variables globales declaradas antes que la propia función. Pero si queremos pasar un parámetro concreto a la función que se ejecuta en un hilo dicho parámetro debe ser de tipo puntero a `void` ya que es así como tiene que estar declarada la función. Afortunadamente, en C una variable de tipo puntero a `void` puede guardar punteros a cualquier tipo y, utilizando el "casting", podemos hacer que el compilador lo acepte sin dar error.

Por lo tanto, para pasar un argumento de cualquier tipo a una función ejectuada en un hilo, hay que pasarle un puntero a dicha variable enmascarado como puntero a `void`. Dentro de la función, hay que hacer la operación contraria, es decir, como la función lo que recibe es un puntero a `void`, habrá que convertirlo a un puntero al tipo que queramos (el mismo que el pasado) para poder utilizarlo correctamente.

Ahora bien, dado que la función no se ejecuta directamente sino a través de la llamada a pthread_create, el argumento se pasa a la función a través del cuarto parámetro de pthread_create. Como ejemplo, veamos como hacer el programa anterior pero usando una sóla función que se lanza en dos hilos distintos parametrizada con la cadena a mostrar.

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

```
void *slowprintf(void *arg) {
  char *msg;
  int i;

  msg = (char *)arg;

  for ( i = 0 ; i < strlen(msg) ; i++ ) {
    printf("%c", msg[i]);
    fflush(stdout);
    usleep(1000000);
  }
}

int main(int argc, char *argv[]) {
  pthread_t h1;
  pthread_t h2;
  char *hola = "Hola ";
  char *mundo = "mundo";

  pthread_create(&h1, NULL, slowprintf, (void *)hola);
  pthread_create(&h2, NULL, slowprintf, (void *)mundo);
  pthread_join(h1, NULL);
  pthread_join(h2, NULL);
  printf("Fin\n");
}
```

Listado 3: Ejemplo de cómo pasar parámetros a los hilos en c

en el ejemplo del listado 3, las variables `hola` y `mundo` son los punteros a caracter que son pasados a la función `slowprintf` enmascarados como punteros a `void` con el casting `(void *)`.

Cuando se necesita pasar más de un parámetro a una función ejecutada en un hilo hay que recurrir a las estructuras (`struct`). Agrupando los distintos parámetros en una estructura y pasando un puntero a dicha estructura, la función tendrá acceso a todos los argumentos. Recordemos que para acceder a los campos de una estructura en C se utiliza el operador "." y que cuando lo que se tiene es un puntero a una estructura, se utiliza el operador "$->$". Por ejemplo, veamos cómo hacer una multiplicación de una matriz por un escalar dentro de un hilo.

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

#include <pthread.h>


struct parametros {
  int id;
  float escalar;
  float matriz[3][3];
};

void init(float m[3][3]) {
  int i;
  int j;

  for ( i = 0 ; i < 3 ; i++ ) {
    for ( j = 0 ; j < 3 ; j++ ) {
      m[i][j] = random()*100;
    }
  }
}

void *matrizporescalar(void *arg) {

  struct parametros *p;
  int i;
  int j;
```

```
  p = (struct parametros *)arg;

  for ( i = 0 ; i < 3 ; i++ ) {
    printf("Hilo %d multiplicando fila %d\n", p -> id, i);
    for ( j = 0 ; j < 3 ; j++ ) {
      p -> matriz[i][j] = p -> matriz[i][j] * p -> escalar;
      usleep(100000);
    }
  }
}

int main(int argc, char *argv[]) {
  pthread_t h1;
  pthread_t h2;

  struct parametros p1;
  struct parametros p2;

  p1.id = 1;
  p1.escalar = 5.0;
  init(p1.matriz);
  p2.id = 2;
  p2.escalar = 10.0;
  init(p2.matriz);

  pthread_create(&h1, NULL, matrizporescalar, (void *)&p1);
  pthread_create(&h2, NULL, matrizporescalar, (void *)&p2);
  pthread_join(h1, NULL);
  pthread_join(h2, NULL);
  printf("Fin\n");
}
```

Listado 4: Ejemplo de paso de varios parámetros a un hilo mediante una estructura

en el ejemplo anterior vemos como en primer lugar definimos la estructura `parametros` que contiene toda la información que se quiere pasar a las funciones que se ejecutan en los hilos. En concreto, un identificador de hilo de ejecución, que resultará útil para mostrar información de depuración por pantalla, una matriz de reales y un valor real por el cual multiplicar dicha matriz.

En el programa principal se usan dos variables del tipo `struct parametros` a cuyos campos se le asignan valores adecuados usando el operador ".". Después se crean los hilos pasando la misma función para ser ejecutada, pero distintos punteros a parámetros. Aquí hemos usado el operador & para obtener el puntero a la variable que se desea pasar como parámetro y además se usa el casting para enmascararlo.

Dentro de la función `matrizporescalar`, se necesita convertir el puntero a void recibido a un puntero al tipo de dato recibido, por eso se declara la variable local `p` y se hace el casting inverso. Para acceder a los campos de la variable apuntada por `p` usamos el operador "$->$".

EJERCICIO 4 *Escribe el programa anterior y comprueba su funcionamiento.*

2.3. Liberación de recursos

Los recursos asignados por el sistema operativo a un hilo son liberados cuando el hilo termina. La terminación del hilo se produce cuando la función que está ejecutando termina, cuando ejecuta un `return` o cuando ejecuta la función `pthread_exit`. Si la función no ejecuta ni return ni `pthread_exit`, se ejecuta automáticamente un `return 0`. Teniendo en cuenta que las funciones que se ejecutan en hilos deben devolver un puntero a `void`, en este caso, el valor devuelto es un puntero con valor NULL.

3. Comparando tiempos

El código del listado 5 muestra un proceso que crea 100 procesos hijos, midiendo el tiempo medio que tarda en la creación de cada uno de ellos.

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

int main() {
    struct timeval t0, t1;
    int i = 0;
    int id = -1;
    gettimeofday(&t0, NULL);
    for ( i = 0 ; i < 100 ; i++ ) {
        id = fork();
        if (id == 0) return 0;
    }
    if (id != 0) {
        gettimeofday(&t1, NULL);
        unsigned int ut1 = t1.tv_sec*1000000+t1.tv_usec;
        unsigned int ut0 = t0.tv_sec*1000000+t0.tv_usec;
        printf("%f\n", (ut1-ut0)/100.0); /* Tiempo medio en microsegundos */
    }
}
```

Listado 5: Código para medir tiempo de creación de procesos c

El código del listado 6 muestra un hilo que crea 100 hilos hijos, midiendo el tiempo medio que tarda en la creación de cada uno de ellos.

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

struct timeval t0, t1;
double media = 0.0;

void *hilo(void *arg) {
    gettimeofday(&t1, NULL);
    unsigned int ut1 = t1.tv_sec*1000000+t1.tv_usec;
    unsigned int ut0 = t0.tv_sec*1000000+t0.tv_usec;
    media += (ut1-ut0);
}

int main() {
    int i = 0;
    pthread_t h;
    for ( i = 0 ; i < 100 ; i++ ) {
        gettimeofday(&t0, NULL);
        pthread_create(&h, NULL, hilo, NULL);
        pthread_join(h, NULL);
    }
    printf("%f\n", (media/100.0)); /* Tiempo medio en microsegundos */
}
```

Listado 6: Ejemplo de creación de hilos en c

Ejercicio 5 *¿Con cuál de las dos opciones obtienes un mejor rendimiento?. Razona la respuesta*