Práctica 7. Threads

Índice de contenidos

1.- Objetivos

- Utilizar las llamadas al sistema *UNIX* relacionadas con la identificación, creación y terminación de *threads*.
- Utilizar las llamadas al sistema *UNIX* para terminar threads y para conocer el estado en el que finalizan otros *threads*.
- Escribir programas que den lugar a la ejecución de múltiples *threads* de manera concurrente.

2.- Definición de *thread*

Un *thread* (hilo) es flujo de ejecución que pertenece a un proceso, es decir, los hilos se ejecutan de forma concurrente y comparten el mismo espacio de direcciones. Un hilo y un proceso son por tanto conceptos similares, indicados para resolver situaciones en las que un programa deba realizar varias tareas de manera simultánea (por ejemplo: mostrar una ventana y realizar un cálculo).

Tanto procesos, como hilos podrían resolver este tipo situaciones, sin embargo el uso de hilos implica una menor carga para la maquina ya que la creación, sincronización y comunicación de hilos implica menos consumo de recursos (*CPU*, *memoria*).

Cada hilo compartirá con el resto de hilos del proceso:

- *Espacio de memoria*: Lo que implica que ejecuta el mismo código, en la misma zona de memoria. Esta es una característica de especial importancia ya que cualquier puntero hará referencia a la misma posición de memoria.
- *Variables globales* : La modificación de una variable global en un hilo, la modifica para el resto de los hilos.
- *Ficheros abiertos*
- *Procesos hijos*
- *Señales y Temporizadores*

Por otra parte, cada hilo tiene:

- *Sus propios registros (SP, PC, ...)* : Los registros son independientes entre cada uno de los hilos.
- *Su propia pila* : Las variables locales se almacenan en la pila, por tanto los hilos tendrán variables locales independientes. A este respecto hay que tener en aunque la pila esta separada, sigue perteneciendo al mismo espacio de direcciones de todo el proceso.
- *Su estado*

Cada proceso podrá tener uno o más hilos de ejecución. Cada hilo de ejecución tendrá su propio

identificador (*tid*) que sólo será valido para los hilos del mismo proceso. A diferencia de los identificadores de proceso *pid* que son únicos para todos los procesos.

Planificación de *threads*. Tipos de *threads*

Cada hilo dispone de su propia política de planificación. Dependiendo del tipo de hilo, la planificación será gestionada por el sistema operativo, o por el planificador de hilos del proceso:

- *Hilos de Usuario*: El sistema operativo no tiene constancia de este hilo, será planificado dentro del proceso.
- *Hilos de Núcleo* : El sistema operativo es el encargado de planificar el hilo.

Dependiendo de la forma de hacer corresponder los hilos de usuario en los hilos del núcleo, existen tres tipos diferentes:

- *Muchos-a-uno (Many-to-one)*: Varios hilos de usuario se implementan como un solo hilo de núcleo.
  - Ejemplo: Biblioteca *light weighted processes* del sistema operativo *SunOS*.
- *Uno-a-uno (One-o-one)*: Cada hilo de usuario tiene correspondencia con un hilo del núcleo.
  - Ejemplo: *Linux* y *Windows NT* .
- *Muchos-a-muchos (Many-to-many)*: Varios hilos de usuario tienen correspondencia con varios hilos de núcleo.
  - Ejemplo: *Tru64* (antiguo *Digital UNIX*) .

3.- Biblioteca *pthreads.h*.

El uso de hilos en Linux se realiza por medio de interfaz POSIX *pthreads*. Este interfaz esta descrito en la biblioteca *pthreads.h* de C.

Creación y finalización de threads

Para crear un nuevo hilo se invoca la función *pthread_create()* :

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
                    void *(*start)(void *),void* arg);
```

con los siguientes argumentos:

- `thread` : Esta función retorna el identificador del hilo (*tid*) en la variable.
- `attr` : Si indica las características de hilo que se va a crear (*NULL* para utilizar los atributos por defecto).
- `start` : Un puntero a la función que ejecutara el hilo .
- `arg` : Argumentos de la función.

Para finalizar un hilo se invoca a la función *pthread_exit()*:

```
void pthread_exit(void *retval);
```

con los siguientes argumentos:

- **`retval`**: Valor de retorno.

Todas las funciones de hilos necesitan incluir la biblioteca *pthread.h*. Es importante resaltar que cuando se crea un nuevo hilo, no se duplica toda la información del proceso, como sucede cuando se invoca la función *fork()*. Es decir, toda la información que esta almacenada el *PCB* (*Process Control Block*) señala a la del proceso que creo el hilo.

El siguiente programa crea dos hilos, que comienzan utilizando la misma función, pero con argumentos diferentes[1]:

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

void *Hilo(void *arg) {
  printf ("%s\n",(char *)arg);
  pthread_exit (NULL);
}

int main() {
  pthread_t t1, t2;
  pthread_create (&t1, NULL, Hilo, "Ejecuta un hilo");
  pthread_create (&t2, NULL, Hilo, "Ejecuta otro hilo");
  printf("Fin del thread principal\n");
  return 0;
}
```

Atributos de un *thread*

Cuando se crea un hilo mediante la función *pthread_create()* se pueden definir varios atributos referentes al nuevo hilo:

- Devolución de los recursos .
- Ámbito: Usuario o núcleo.
- Otros: Tamaño de la pila , ubicación de la pila, etc.

Los atributos se indicarán en el segundo argumento de *pthread_create()*. Antes de invocar esta función se debe inicializar los atributos con que deseamos que utilice este hilo (crear los atributos). Una vez que se haya creado el hilo se deben destruir los atributos (eliminar los atributos). Esto se realiza con las funciones *pthread_attr_init( )* y *pthread_attr_destroy( )*:

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

pthread_attr_init( ) inicializa los atributos (**`attr`**), para ser posteriormente utilizados en *pthread_create()*. Devuelve *0* si no se ha producido un error.

1 Para compilar utilizando la biblioteca pthread.h se le debe indicar al compilador que utilice esta biblioteca de la siguiente manera: **`gcc -o ejemplo1 ejemplo1.c -lpthread`**.

pthread_attr_destroy( ) destruye los atributos.

Devolución de recursos. Threads detached vs. threads joinable

Cuando un hilo finaliza, puede devolver los recursos que esta utilizando de dos formas:

- *Detached*: El hilo es autónomo, una vez que finaliza devuelve todos los recursos que esta utilizando (*tid*, *pila*, etc.) .
- *Joinable*: Cuando finaliza, retiene todos los recursos hasta que un hilo invoca la función *pthread_join()*.

Para inicializar la devolución de recursos se utilizara la función *pthread_attr_getdetachstate ()*:

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

con los argumentos:

- **`attr`**: Atributo a modificar.
- **`detachstate`**: Tipo de devolución de recursos:
  - `PTHREAD_CREATE_JOINABLE`
  - `PTHREAD_CREATE_DETACHED`

Al invocar la función *pthread_join()* el hilo que la invoca se queda detenido hasta que el hilo indicado finalice:

```
int pthread_join(pthread_t thread, void **retval);
```

donde:

- **`thread`**: Es el identificador del *thread* al que se espera.
- **`retval`**: Es el valor retornado por *pthread_exit ()*.

y donde *pthread_exit()* sería:

```
void pthread_exit(void *value_ptr);
```

donde:

- **`value_ptr`**: Es el valor que se recupera con *pthread_join()*.

El siguiente ejemplo muestra como la función *pthread_exit()* envía retorno a *pthread_join()*:

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

int return_value=10;

void *Hilo2(void *arg) {
  printf("%s\n", (char *)arg);
  sleep(10);
```

```
  pthread_exit(NULL);
}

void *Hilo1(void *arg) {
  printf ("%s\n", (char *)arg);
  pthread_exit((void *)return_value);
}

int main() {
  pthread_t t1, t2;
  pthread_attr_t attr;
  int ret1;
  // Inicializacion del attributo
  if (pthread_attr_init(&attr) != 0) {
    perror("Error en la creación de la estructura de los atributos.");
    exit(-1);
  }
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create (&t1, &attr, Hilo1, "Soy joinable");
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED );
  pthread_create (&t2, &attr, Hilo2, "Soy detached");
  pthread_join(t1, (void **)&ret1);
  printf("El thread joinable terminó con retval='%d'\n", ret1);
  return 0;
}
```

Ambito. Threads de usuario vs. Threads de núcleo

Los hilos se pueden ejecutar como hilos de usuario, donde será la propia biblioteca *pthread.h* la encargada de planificar los hilos. De esta manera, el sistema operativo desconoce la existencia de un nuevo hilo. Para el *kernel*, será simplemente un solo proceso.

O bien, se pueden ejecutar como hilos de núcleo. En este modo es el *kernel* del sistema operativo el que se encarga de planificar los hilos.

Los hilos de usuario son mas “ligeros” ya que cada vez que se conmuta de un hilo a otro no es necesario que intervenga el *kernel* del sistema operativo.

Para inicializar el ámbito de un hilo utilizará la función *pthread_attr_setscope ()*:

```
int pthread_attr_setscope (const pthread_attr_t *attr, int *scope);
```

con los argumentos:

- **`attr`**: Atributo a modificar:
- **`scope`**: Tipo de hilo:
  - *PTHREAD_SCOPE_SYSTEM* : Para hilos de núcleo.

- *PTHREAD_SCOPE_PROCESS* : Para hilos de usuario.

Las *threads* de *Linux* (*pthread.h*) sólo soportan los hilos de núcleo. Si intentamos poner ámbito de usuario en *Linux*, la función anterior nos devolverá un código de error. Sin embargo la norma *POSIX* lo permite.

Identificación de *threads*

Para obtener la identificación de hilo (*tid*) se dispone de la función *pthread_t pthread_self( )*:

```
pthread_t pthread_self(void);
```

El valor que retorna es únicamente valido dentro del mismo proceso. Este valor no coincide con el valor que muestra el comando *ps*[2]. Si lo que deseamos es mostrar el valor del identificador de thread que muestra el comando *ps* en *LINUX*, debemos utilizar la siguiente función:

```
pid_t gettid(void) {
  return syscall(__NR_gettid);
}
/* Requiere #include <sys/syscall.h>                        *
 * Esta función no esta implementada en ninguna biblioteca *
 * por lo que debemos codificarla en cada programa.        */
```

Lamentablemente esta función no es POSIX (no esta incluida en ninguna biblioteca), ya que hace una llamada al sistema operativo para obtener el valor SPID. Por tanto, si se desea hacer un programa portable POSIX no se debe utilizar.

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>

pid_t gettid(void) {
  return syscall(__NR_gettid);
}

void *pruebaSpid(void *ptr) {
  printf("Thread secundario: PID=%d, SPID=%d\n", getpid(), gettid());
  sleep(10);
  pthread_exit(NULL);
}
```

2 Con el comando *process status* (*ps*) se pueden listar los hilos de un proceso: **`ps -T`** . Este comando muestra el *pid* y el identificador de hilo *spid*. Se puede combinar con la opción **`-p`** para mostrar los hilos de un determinado proceso. Por ejemplo, **`ps -p 2345 -T`** , muestra el listado de todos los hilos del proceso *2345*.

```
int main() {
  pthread_t thread1;
  pthread_create(&thread1, NULL, pruebaSpid, NULL);
  printf("Thread principal: PID=%d, SPID=%d\n", getpid(), gettid());
  sleep(20);
  return 0;
}
```

Terminación de *threads*

Para obligar a un hilo a que termine se invoca la función *pthread_cancel()* pasando como argumento el *tid* del hilo que se desea cancelar :

```
int pthread_cancel(pthread_t thread);
```

Ejemplo:

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

void *Hilo (void *arg) {
  while(1) {
    printf ("%s\n", (char *)arg);
    sleep(1);
  }
}

int main() {
  pthread_t th1;
  pthread_create(&th1, NULL, Hilo,"Looped thread");
  printf("Espero 10 segundos antes de cancelar este thread.\n");
  sleep(10);
  pthread_cancel(th1);
  printf("Cancelado.\n");
  return 0;
}
```

4.- Desarrollo de la práctica

Espacio de direcciones en *threads* vs. espacio de direcciones en procesos

Examinemos el siguiente programa:

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

int a=0;
```

```
void *Hilo1(void *arg) {
  printf ("EL valor de a en el hilo es %d\n", a);
  a++;
  printf ("Nuevo valor de a en el hilo es %d\n", a);
  pthread_exit(NULL);
}

int main() {
  pthread_t t1;
  pthread_create(&t1, NULL, Hilo1, "Soy joinable\n");
  pthread_join(t1, NULL);
  printf("El valor después de ejecutar el thread es %d\n", a);
  return 0;
}
```

Este programa crea un *thread* que modifica la variable *a* y termina.

- ¿Qué valor muestra el *thread* principal?
- ¿Qué conclusión extrae sobre el espacio de direcciones que utilizan los *threads*?

El siguiente código es similar al anterior, pero utiliza procesos en lugar de *threads*:

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

int a=0;

int main()
 {
  pid_t hijo=fork();
  if (hijo==-1) {
    perror("Error en la llamada a fork()");
    exit(-1);
  } else if (hijo==0) {
    printf("El valor de a en el proceso hijo es %d\n", a);
    a++;
    printf("Nuevo valor de en el proceso hijo es %d\n", a);
  } else {
    hijo=wait(NULL);
    printf("El valor después de crear/terminar el proceso hijo es %d\n", a);
  }
  return 0;
}
```

Ahora es un proceso el que modifica una variable global:

- ¿Qué valor muestra el proceso hijo?.

- Una vez finalizado este proceso, ¿Qué valor muestra el proceso padre?
- ¿Qué conclusión extraemos sobre el espacio de direcciones que utilizan los procesos?

Terminación de procesos con *threads*

Analiza el siguiente código:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>

void *hiloColgado(void *ptr) {
  while(1) {
    printf("%s: PID=%d\n", (char *)ptr, getpid());
    sleep(1);
  }
}

void *hiloAsesino(void *ptr) {
   printf("%s: PID=%d. Voy a esperar 10 segundos y termino todo este lío.\n",
(char *)ptr, getpid());
  sleep(10);
  exit(0);
}

int main() {
  pthread_t thread1, thread2;
  char *message1 = "Hilo colgado";
  char *message2 = "Hilo asesino de proceso";
  pthread_create(&thread1, NULL, hiloColgado, (void*)message1);
  pthread_create(&thread2, NULL, hiloAsesino, (void*)message2);
  while (1){
    printf("Hilo principal: PID=%d\n", getpid());
    sleep(1);
  }
  return 0;
}
```

El único hilo que termina es el que invoca la función hiloAsesino, entonces, ¿Por qué terminan el resto de los hilos?

Terminación de hilos

Analiza el siguiente código:

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

```
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>

void *hilo(void *ptr) {
  int i, x=*(int*)ptr;
  for (i=0; i<x; i++) {
    printf("Voy a repetir %d y estoy en la iteración %d\n", x, i);
    sleep(1);
  }
}

int main() {
  pthread_t thread1, thread2;
  int n1=5, n2=10;
  pthread_create( &thread1, NULL, hilo, (void*)&n1);
  pthread_create( &thread2, NULL, hilo, (void*)&n2);
  return 0;
}
```

Los hilos del *thread1* y *thread2* no se ejecutan completamente. ¿Qué es lo que sucede?

Modifica la función *main* para que los hilos se ejecuten correctamente.

Sincronización de hilos

Analiza el siguiente código:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>

#define N 15.0

void *tareaHilo(void *ptr) {
  int x;
  printf("COMIENZO TAREA HILO %d\n", *(int *)ptr);
  srand ((int)pthread_self());
  x = 1+(int)(N*rand()/RAND_MAX+1.0); // X es un número aleatorio entre 1 y N

  sleep(x);
  printf("FIN TAREA HILO %d\n",*(int *)ptr);
}

int main() {
```

```
  pthread_t thread1;
  int hilo = 1;
  pthread_create(&thread1, NULL, tareaHilo, &hilo);
  sleep(1);
  hilo++;
  tareaHilo(&hilo);
  pthread_exit(NULL);
  return 0;
}
```

El resultado de ejecutar el programa es:

```
COMIENZO TAREA HILO 1
COMIENZO TAREA HILO 2
FIN TAREA HILO 2
FIN TAREA HILO 2
```

¿Como explicas que no aparezca FIN TAREA 1?

Cree un programa que sincronice los dos hilos utilizando *pthread_join()*, de tal manera que el programa primero realice la tarea del hilo 1 y después la del hilo 2 (hilo principal).

Por tanto, la salida del programa será:

```
COMIENZO TAREA HILO 1
FIN TAREA HILO 1
COMIENZO TAREA HILO 2
FIN TAREA HILO 2
```

Compara esta sincronización con la que realizaremos en el siguiente punto.

Sincronización de varios hilos

Analiza el siguiente código:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>

#define N 5.0

void *hilo(void *ptr) {
  int x;
  printf("COMIENZO TAREA HILO\n");
  srand((int)pthread_self());
  x=1+(int)(N*rand()/ RAND_MAX +1.0); // X es un número aleatorio entre 1 y N
```

```
  sleep(x);
  printf("FIN TAREA HILO : Tiempo ejecucion %d\n",x);
}

int main() {
  pthread_t t1, t2, t3, t4, t5;
  pthread_create(&t1, NULL, hilo, NULL);
  pthread_create(&t2, NULL, hilo, NULL);
  pthread_create(&t3, NULL, hilo, NULL);
  pthread_create(&t4, NULL, hilo, NULL);
  pthread_create(&t5, NULL, hilo, NULL);
  printf("Van a comenzar los hilos :\n");
  pthread_exit(NULL);
  return 0;
}
```

Todos los hilos están sin sincronizar, es decir, comienzan y terminan sin esperar unos a otros. Modifica el programa para queden sincronizados. Para hacerlo, NO puedes utilizar la función *pthread_join()* en la función *main()*. Todos los hilos serán ejecutados de forma concurrente, de tal manera que el hilo *t1*, espere a que finalice el hilo principal, el *t2* al *t1*, el *t3* al *t2*, etc. Por tanto el programa mostrará por pantalla:

```
Van a comenzar los hilos :
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 2
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 6
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 5
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 4
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 3
```