Concurrencia en *Threads*

Índice de contenidos

1.- Objetivos

- Utilizar mecanismos de gestión de la concurrencia (Semáforos y variables de condición).
- Escribir programas que den lugar a la ejecución de múltiples *threads* de manera concurrente gestionando de manera ordenada su interacción.

2.- Mecanismos de sincronización con *threads* en *LINUX*

Exclusión mutua. Semáforos (*mutex*)

El *mutex* es un semáforo binario con dos operaciones atómicas:

- *lock()*: Intenta bloquear un *mutex*. Si está bloqueado el hilo se suspende hasta que otro hilo realice una operación *unlock()*.
- *unlock()*: Desbloquea el *mutex*. Si existen hilos bloqueados se desbloquea a uno de ellos.

Para utilizar los mutex, primero se deben crear mediante la función *pthread_mutex_init* :

```
int pthread_mutex_init(pthread_mutex_t * mutex,
      const pthread_mutexattr_t *attr);
```

con los siguientes argumentos:

- `mutex`: *mutex* que se inicializa.
- `attr`: Parámetros de inicialización. Para inicializar con parámetros por defecto utilizamos *NULL,* en ese caso el *mutex* se inicializa desbloqueado.

pthread_mutex_init retorna *0* si no se ha producido un error. Si se produce un error retorna el código de error.

Para eliminar un *mutex* se utiliza la función *pthread_mutex_destroy* :

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

pthread_mutex_destroy retorna *0*, si no se ha producido un error. Si se produce un error retorna el código de error .

Para bloquear un *mutex* se utiliza la función *pthread_mutex_lock* :

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

pthread_mutex_lock retorna *0*, si no se ha producido un error. Si se produce un error retorna el código de error .

Para desbloquear un *mutex* se invoca a la función *pthread_mutex_unlock* :

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

pthread_mutex_unlock retorna *0*, si no se ha producido un error. Si se produce un error retorna el código de error .

El siguiente código muestra como realizar una exclusión mutua mediante semáforos:

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

void *Hilo1(void *arg) {
  pthread_mutex_lock(arg);
  printf("Comienza un hilo\n");
  sleep(5);
  printf("Finaliza un hilo\n");
  pthread_mutex_unlock(arg);
  pthread_exit(NULL);
} /* Fin de Hilo */

int main() {
  pthread_t t1,t2;
  pthread_mutex_t misemaforo;
  // Creamos un semaforo
  if (pthread_mutex_init(&misemaforo, NULL)!=0) exit(-1);
  pthread_create(&t1, NULL, Hilo1, &misemaforo);
  pthread_create(&t2, NULL, Hilo1, &misemaforo);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  if (pthread_mutex_destroy(&misemaforo)!=0) exit(-1);
  return 0;
}
```

El siguiente código resuelve (parcialmente) la cuestión 5 de la práctica anterior utilizando semáforos:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#define N 5.0

pthread_mutex_t mutex;
int i[5]={1,2,3,4,5};

void *hilo(void *ptr) {
  int x;
  pthread_mutex_lock(&mutex);
  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: Tiempo ejecución %d\n", *(int *)ptr, x);
```

```
  pthread_mutex_unlock(&mutex);
}

int main() {
  pthread_t t1, t2,t3,t4,t5;
  pthread_mutex_init(&mutex, NULL);
  pthread_create(&t1, NULL, hilo, (void *)&i[0]);
  pthread_create(&t2, NULL, hilo, (void *)&i[1]);
  pthread_create(&t3, NULL, hilo, (void *)&i[2]);
  pthread_create(&t4, NULL, hilo, (void *)&i[3]);
  pthread_create(&t5, NULL, hilo, (void *)&i[4]);
  pthread_exit(NULL);
  return 0;
}
```

Este código no resuelve totalmente la cuestión 5, ya que pedía que los hilos se sincronizasen en orden. Es decir, el primer hilo que tendría que entrar en la exclusión mutua sería el t1, el segundo t2, etc. En el código anterior, si bien garantizamos la exclusión mutua, dejamos en manos del *mutex* el orden de entrada a esta zona crítica.

Variables de condición

Una variable de condición es un objeto de sincronización que permite bloquear a un hilo hasta que otro decide reactivarlo. Una variable de condición siempre está asociada a un *mutex*.

Las operaciones atómicas que realiza son:

- Esperar una condición (*wait*) :
  - Esta condición se debe realizar con el *mutex* cerrado (*lock*).
  - El hilo se suspende hasta que otro señaliza la condición y el *mutex* asociado se desbloquea.
- Señalizar una condición (*signal*) :
  - Se señaliza esta condición.
  - Si no existen hilos suspendido por esta operación no tiene ninguna consecuencia.
  - Si existe un hilo, o más, suspendido/s, se activa/n y pasa a competir por el *mutex* asociado.

Para esperar una condición se utiliza la función *pthread_cond_wait* :

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

con los siguientes argumentos:

- **`cond`**: Condición.
- **`mutex`**: *mutex* asociado a esta señal.

pthread_cond_wait retorna *0* si no se ha producido ningún error.

Para señalizar una condición se utiliza la función *pthread_cond_signal* :

```
int pthread_cond_signal(pthread_cond_t *cond);
```

con los siguientes argumentos:

- **`cond`**: Condición.

pthread_cond_*signal* retorna 0 si no se ha producido ningún error.

El contenido de la variable de condición es opaco para el programador. Es decir, las variables declaradas *pthread_cond_t* sólo se utilizan para realizar operaciones *wait* y *signal*, el contenido de estas variables carecen de significado para el programador .

Antes de utilizar un *mutex* este se debe inicializar mediante la función *pthread_cond_init*:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

con los siguientes argumentos:

- **`cond`**: Variable de condición que se va a inicializar.
- **`attr`**: Atributos de la variable de condición. Usaremos *NULL* para las opciones por defecto.

pthread_cond_*init* retorna 0 si no se ha producido ningún error.

Una vez que no se vaya a utilizar más una variable de condición, se destruiye para liberar recursos mediante la función *pthread_cond_destroy*:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

pthread_cond_destroy retorna 0 si no se ha producido ningún error.

Con las variables de condición podemos resolver la cuestión 5 de la práctica anterior de manera que se especifique el orden de ejecución de los hilos:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#define N 5.0

pthread_mutex_t mutex;
int i[5]={1,2,3,4,5};
pthread_cond_t condiciones[6];
/*
Cada condición expresa la finalización de un hilo
condiciones[0] fin del hilo main
condiciones[1] fin del hilo t1
condiciones[2] fin del hilo t2
condiciones[3] fin del hilo t3
condiciones[4] fin del hilo t4
condiciones[5] fin del hilo t5
*/
pthread_cond_t condicion; // han entrado los cinco threads
int M=0; // Numero de threads que han entrado

void *hilo(void *ptr) {
  int x;
  pthread_mutex_lock(&mutex);
```

```
  // Enviar señal al thread main cuando los 5 threads estén en el semáforo
  if ((++M)>=5) pthread_cond_signal(&condicion);

  // Comprobamos si ha finalizado su hilo precedente
  pthread_cond_wait(&condiciones[*(int *)ptr-1], &mutex);

  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: Tiempo ejecucion %d\n", *(int *)ptr, x);

  // enviamos señal de finalización del hilo
  pthread_cond_signal(&condiciones[*(int *)ptr]);

  pthread_mutex_unlock(&mutex);
}

int main() {
  pthread_t t1, t2,t3,t4,t5;
  pthread_attr_t tattr;
  pthread_mutex_init(&mutex, NULL);
  int x;

  // inicializamos las condiciones de espera de los hilos
  for(x=0; x<=5; x++)
    if (pthread_cond_init(&condiciones[x], NULL)!=0) exit(-1);

  // inicializamos la condicion de espera del hilo principal
  if (pthread_cond_init(&condicion, NULL)!=0) exit(-1);

  pthread_mutex_lock(&mutex);
  pthread_create(&t1, NULL, hilo, (void *)&i[0]);
  pthread_create(&t2, NULL, hilo, (void *)&i[1]);
  pthread_create(&t3, NULL, hilo, (void *)&i[2]);
  pthread_create(&t4, NULL, hilo, (void *)&i[3]);
  pthread_create(&t5, NULL, hilo, (void *)&i[4]);

  // Esperamos a que los 5 threads queden detenidos en el semáforo
  pthread_cond_wait(&condicion, &mutex);
  printf("Van a comenzar los hilos\n");
  //Enviamos la señal de finalización del hilo principal
  pthread_cond_signal(&condiciones[0]);
  pthread_mutex_unlock(&mutex);
  pthread_exit(NULL);
  return 0;
}
```