SESIÓN N° 08

SINCRONIZACIÓN DE HILOS

I
OBJETIVOS

- Conocer y aplicar los conceptos de mecanismos de sincronización para hilos.
- Elaborar programas que sincronicen la ejecución de varios hilos

II
TEMAS A TRATAR

- Funciones para la sincronización de hilos
- Variables de condición

III
MARCO TEORICO

La concurrencia es la propiedad de los sistemas que permiten que múltiples procesos o hilos sean ejecutados al mismo tiempo, y que potencialmente puedan interactuar entre sí.

Trabajar con hilos es trabajar nativamente con programas concurrentes, uno de los mayores problemas con los que nos podemos encontrar, y que es tácito en la concurrencia, es el acceso a variables y/o estructuras compartidas o globales, es decir usar variables que son modificadas por otros hilos.

Existen regiones críticas (RC) que son parte de código susceptible de verse afectada por la concurrencia, por lo que para solucionar el problema los hilos POSIX nos ofrecen los semáforos binarios o semáforos mutex (o simplemente mutexs).

Un **semáforo binario** es una estructura de datos que puede tener dos estados: abierto o cerrado. Cuando el semáforo está abierto, al primer hilo que pide un bloqueo se le asigna ese bloqueo y no se deja pasar a nadie más por el semáforo. Mientras que, si el semáforo está cerrado, porque algún hilo ya tiene el bloqueo, el hilo que lo pidió parará su ejecución hasta que no sea liberado el mencionado bloqueo. Solo puede haber un solo hilo teniendo el bloqueo del semáforo, mientras que puede haber más de un hilo esperando para entrar en la RC situados en la cola de espera del semáforo. Es decir, los hilos se excluyen mutuamente (de ahí el nombre de mutex) el uno al otro para entrar.

Funciones para la sincronización de hilos

Las principales funciones para el uso de mutex son las siguientes (figura 8.1):

Función	Descripción

`pthread_mutex_init`	Inicializa un mutex con los atributos especificados
`pthread_mutex_destroy`	Destruye el mutex especificado
`pthread_mutex_lock`	Permite solicitar acceso al mutex, el hilo se bloquea hasta su obtención
`pthread_mutex_trylock`	Permite solicitar acceso al mutex, el hilo retorna inmediatamente. El valor retornado indica si otro hilo lo tiene.
`pthread_mutex_unlock`	Permute liberar un mutex.

Figura 8.1. Funciones para el uso de mutex

Inicialización de mutex

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

- **mutex**: es un puntero a un parámetro del tipo *`pthread_mutex_t`*, que es el tipo de datos que usa la librería *`pthreads`* para controlar los mutex.

- **attr**: es un puntero a una estructura del tipo *`pthread_mutexattr_t`* y sirve para definir qué tipo de mutex queremos: normal, recursivo o *errorcheck*. Si este valor es NULL (recomendado), la librería le asignará un valor por defecto.

La función devuelve 0 si se pudo crear el mutex o -1 si hubo algún error. Esta función debe ser llamada antes de usar cualquiera de las funciones que trabajan con mutex.

Alternativamente podemos invocar:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

lo cual inicializa el mutex con los atributos por omisión. Es equivalente a invocar:

```
pthread_mutex_t mylock;
pthread_mutex_init(&mylock, NULL);
```

Petición de bloqueo

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

- **mutex:** es un puntero al mutex sobre el cual queremos pedir el bloqueo o sobre el que nos bloquearemos en caso de que ya haya alguien dentro de la RC.

Esta función pide el bloqueo para entrar en una RC. Si queremos implementar una RC, todos los hilos tendrán que pedir el bloqueo sobre el mismo semáforo. Como resultado, devuelve 0 si no hubo error, o diferente de 0 si lo hubo.

Liberación de bloqueo

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

- **mutex**: es el semáforo donde tenemos el bloqueo y queremos liberarlo.

Esta función libera el bloqueo que tuviéramos sobre un semáforo. Retorna 0 como resultado si no hubo error o diferente de 0 si lo hubo.

Destrucción de mutex

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

mutex: el mutex que queremos destruir.

Esta función le dice a la librería que el mutex que le estamos indicando no lo vamos a usar más, y que puede liberar toda la memoria ocupada en sus estructuras internas por ese mutex. La función devuelve 0 si no hubo error, o distinto de 0 si lo hubo.

Variables de condición:

Las variables de condición son otro mecanismo de sincronización entre hilos. Las variables de condición usadas en conjunto con mutex permiten a un hilo esperar por la ocurrencia de una condición arbitraria. La espera es libre de carreras críticas, para conseguir despertar cuando una condición de interés pueda cambiar.

En principio podríamos pensar que, teniendo posesión de un mutex podríamos consultar por el cambio de la variable que esperamos. Una vez hecha la consulta deberíamos liberar el mutex. El problema es que estamos obligados a hacer un loop esperando por el cambio de la variable (espera activa) lo cual ocupa mucha CPU.

Una variable de condición es una variable del tipo *`pthread_cond_t`* y se utiliza con las funciones apropiadas para la espera y posterior continuación del proceso. El mecanismo de variable de condición permite que los hilos suspendan su ejecución y renuncien al procesador hasta que se cumpla alguna condición.

Las funciones utilizadas con las variables de condición son:

Iniciar un variable de condición

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

- **cond** : variable de condición
- **attr** : especifica los atributos con los que se creara inicialmente la variable condicional, en caso de que este argumento sea NULL, se tomarán los atributos por defecto.

Al igual que mutex, podemos iniciar su valor con la constante `PTHREAD_COND_INITIALIZER`.

Destruir una variable de tipo condición

```
int pthread_cond_destroy(pthread_cond_t *cond, pthread_condattr_t *attr);
```

Suspender un proceso en espera de la señal en la variable condición

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

Suspende al proceso ligero hasta que otro proceso ejecute una operación *`c_signal`* sobre la variable condicional pasada como primer argumento. De forma atómica se libera el

mutex pasado como argumento. Cuando el proceso se despierta volverá a competir por el mutex.

Desbloquear un proceso por la emisión de una operación signal

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

Desbloquea a un proceso suspendido en la variable condicional pasada como argumento a esta función.

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

Esta función desbloquea a todos los procesos ligeros suspendidos en una variable condicional.

IV
ACTIVIDADES

01. El siguiente programa crea dos hilos, los cuales se esperan uno al otro para acceder a los datos de un arreglo de enteros compartido. Uno de los hilos pone todos los ítems del arreglo con el valor 0, luego los rellena todos con el valor 1, después con 2 y así sucesivamente. El otro hilo verifica que todos los valores sean iguales al de la posición 0, de lo contrario muestra un error.

    La versión del programa sin mecanismo de sincronización sería el siguiente, compile y ejecútelo. Observe los resultados.

```
1   #include <pthread.h>
2   void *funcionThread(void *parametro)

3   #define TAMANO_BUFFER 1000
4   int bufer[TAMANO_BUFFER];

5   int main(){
6      pthread_t idHilo;
7      int error;
8      int contador;
9      int i;

10     error=pthread_create(&idHilo,NULL,funcionThread,NULL);
11     if (error!=0){
12          perror("No puedo crear thread..");
13          exit(-1);
14     }

15     while(1){
16          for(i=0;i<TAMANO_BUFFER;i++)
17              buffer[i]=contador;
18          contador++;
19     }
20     return 0;
21  }

22  void *funcionThread(void *parametro) {
```

```
23    int i;
24    elementoDist=0;
25    while(1){
26         for(i=0;i<TAMANO_BUFFER;i++){
27             if (buffer[0]!=buffer[i]){
28                 elementoDist=1;
29                 break;
30             }
31         }
32         if (elementoDist)
33             printf("Hijo: Error. Elementos distintos en el
34                                 buffer\n");
35         else
36             printf("Hijo: Correcto\n");
37    }
38 }
```

Si no hay ningún tipo de sincronización en cuanto al acceso a la estructura compartida (arreglo), es fácil notar que el hilo que escribe no haya terminado y empiece a leer el otro hilo, dando errores ya que no todos los valores del arreglo son iguales.

Para resolver esto, se utilizarán los mutex de forma tal, que mientras que un hilo este accediendo a la variable compartida (arreglo), el otro no pueda acceder. Para esto realice las siguientes modificaciones al programa anterior. Compile, ejecute y observe los resultados.

Colocar entre las líneas 4 y 5 lo siguiente para declarar un mutex:

```
pthread_mutex_t mutexBuffer;
```

entre las líneas 9 y 10, creamos un mutex:

```
pthread_mutex_init(&mutexBuffer,NULL);
```

entre las líneas 15 y 16, bloqueamos el acceso al arreglo en el hilo principal:

```
pthread_mutex_lock(&mutexBuffer);
```

Una vez modificados todos los elementos del arreglo a un único valor, después de la línea 17 se levanta el mutex para permitir que otros hilos puedan acceder al arreglo.

```
pthread_mutex_unlock(&mutexBuffer);
```

El otro hilo debe de verificar que todos tengan el mismo valor, así que antes de línea 26 se debe de bajar el semáforo, y después de la 36 volver a subirlo. Agregue las instrucciones en estas líneas.

Ejecute el programa después de las modificaciones y observe los resultados.

02. El siguiente programa tiene dos hilos que comparten una variable `count` la misma que es modificada por ambos hilos. Para garantizar la exclusión mutua en el acceso a

`count` se utiliza un mutex asociado a una variable de condición que permita suspender a los hilos cuando sea necesario.

Ejecute el programa y analice los resultados.

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

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_var   = PTHREAD_COND_INITIALIZER;

int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

void *functionCount1() {
    for(;;) {
       pthread_mutex_lock( &count_mutex );
       pthread_cond_wait( &condition_var, &count_mutex );
       count++;
       printf("FunctionCount1 - valor count: %d\n",count);
       pthread_mutex_unlock( &count_mutex );

       if(count >= COUNT_DONE) return(NULL);
    }
}

void *functionCount2() {
    for(;;) {
       pthread_mutex_lock( &count_mutex );

       if( count < COUNT_HALT1 || count > COUNT_HALT2 ) {
            pthread_cond_signal( &condition_var );
       }
       else {
            count++;
            printf("Function2Counter – valor count: %d\n",count);
       }

       pthread_mutex_unlock( &count_mutex );

       if(count >= COUNT_DONE) return(NULL);
   }
}

main() {
    pthread_t thread1, thread2;

    pthread_create( &thread1, NULL, &functionCount1, NULL);
    pthread_create( &thread2, NULL, &functionCount2, NULL);

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Final count: %d\n",count);

    exit(EXIT_SUCCESS);
}
```

03. El mutex no solo puede ser utilizado para problemas de secciones críticas como en los ejemplos anteriores, sino también, puede ser utilizado para resolver problemas de sincronización.

    El siguiente ejemplo nos muestre un programa que tiene tres hilos, cada uno de ellos imprime un carácter determinado.

    Ejecútelo y observe los resultados.

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
void *funcion01(void *arg) {
    int i;
    for (i=0; i<20; i++) {
        printf(“+”);
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}
void *funcion02(void *arg) {
    int i;
    for (i=0; i<20; i++) {
        printf(“o”);
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t mihilo01, mihilo02;
    int i;
    if (pthread_create(&mihilo01,NULL,funcion01,NULL) {
        printf(“Error creando el hilo.”);
        abort();
    }
    if (pthread_create(&mihilo02,NULL,funcion02,NULL) {
        printf(“Error creando el hilo.”);
        abort();
    }

    for (i=0; i<20; i++) {
        printf(“x”);
        fflush(stdout);
        sleep(3);
    }
    exit(0);
}
```

    Es obvio verificar que la secuencia resultante es alguna formada por caracteres “o”, ”x” y ”+” sin ningún orden determinado, tal como se muestra a continuación ya que los hilos no tienen ningún mecanismo de sincronización:

```
+oo+++oooxooooxooo++++...
```

Así que a continuación se tendrán que efectuar y agregar estos mecanismos de forma que los hilos se sincronicen de alguna forma.

V
EJERCICIOS

01. Modifique el ejemplo anterior (de los 3 hilos), de tal manera que la función ejecuta por cada hilo sea la misma función, a la cual se le envíe los parámetros necesarios de forma que la secuencia mostrada consista en:

```
xxxxxxxxxxxxxxxxxxx        (20 veces)
ooooooooooooooooooo        (19 veces)
++++++++++++++++++         (18 veces)
. . .                       ...
xxx                        (3 veces)
oo                         (2 veces)
+                          (1 vez)
```

02. Implementar usando hilos un programa que tenga un buffer acotado por un tamaño determinado (este tamaño se recibe como parámetro al momento de lanzar el proceso).

    El programa lanza 4 hilos: t_1; t_2; t_3; t_4. Cada hilo t_i escribe cada `i` segundos, un número `i` en la próxima posición libre ***p*** del buffer. Una vez escrito el elemento, el hilo t_i imprime por pantalla "`Escribí un i en la posición p del buffer`". Se debe cuidar que el acceso al buffer se realice de forma exclusiva.

    Cuando un hilo t_i descubre que el buffer está lleno, termina.

    El hilo original (el que lanzó todos los hilos) debe esperar a que terminen todos y luego terminar.

03. Implemente una solución al problema del productor-consumidor que utilice variables condicionales. Para esto:

    - El hilo principal es el consumidor, y un segundo hilo el productor.
    - El consumidor invoca a una función lee para escribir un valor del buffer.
    - El productor invoca a una función escribe para almacenar un valor en el buffer
    - La generación de datos la realiza la función rand(), y el consumo la función printf().
    - Las funciones de manejo del buffer realizan internamente la sincronización (en las funciones lee y escribe).
    - El buffer FIFO se debe implementar mediante un array circular

    Se sugiere utilizar las siguientes variables y objetos para la sincronización:

Mutex	mutex = 1	Proteje el arreglo FIFO.
Variables de condición	vacio	Indica que el arreglo tiene elementos que pueden ser retirados.
	lleno	Indica que el arreglo tiene elementos vacíos que pueden ser rellenados.
Variables compartidad	cont = 0	Cuenta de elementos ocupados.

VI
INFORME A PRESENTAR

01. Se debe entregar un informe de los ejercicios 2 y 3 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.

02. También se debe entregar los archivos con el código fuente de los programas elaborados en lenguaje C. Se recomienda realizar comentarios adecuados 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).