Sistemas Operativos 2021-02

Código: ICI 2341

Tema: Procesos, Hilos (Threads) y Semáforos en C

Sandra Cano sandra.cano@pucv.cl

1. C en Linux (Instalación, Compilación y Editores)

1. **Compilación** gcc programa.c -o programa
2. **Ejecución** ./programa
3. **Editores:** Sublime-Text, Visual Studio Code.
4. **Compilación con hilos**: gcc programa.c -o programa -lpthread

2. Creación de procesos en Linux

Para poder iniciar en una máquina Linux, procesos adicionales al que se está ejecutando se hace uso del comando **fork** cuya sintaxis es la siguiente:

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

Su definición se encuentra en las librerías **sys/types.h** y **unistd.h**. En **unistd.h** está definida la función y en **types.h** el tipo *pid_t*, el cual es un identificador del proceso Linux PID. Al ser ejecutado *fork*, se 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).

Cuando se crea el proceso hijo, ambos procesos son ejecutados **concurrentemente**, es decir, tenemos dos procesos ejecutando el código del programa. La diferencia entre los dos está en lo que devuelve *fork*. En cada uno de los procesos el valor es diferentes, 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. Estas funciones están definidas en la librería **unistd.h** con la siguiente forma:

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

Veamos un ejemplo de creación de proceso :

```
# 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;
 }
```

3. pthread.h: librería de hilos POSIX de Linux

Qué es POSIX? : Es una norma escrita por la IEEE, que define la interfaz estándar del SO y el entorno, incluyendo un intérprete de comandos. El término fue sugerido por Richard Stallman en la década de 1980, en respuesta a la demando del IEEE. POSIX= Portable Operating System Interface for uni-X.

Para escribir programas multihilo en C se puede hacer uso de la librería **pthread** que implementa el standard POSIX. Para ello se debe incluir la librería que son necesarios para hacer uso de las funciones respectivas. A continuación se

presenta un pequeño código, donde se imprime el mensaje **Hola Mundo** utilizando dos hilos distintos, uno para la palabra **Hola** y otro para la palabra **Mundo**.

Para crear un hilo se realiza con la función **pthread_create** que también est'a 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**, el cual queremos que guarde el identificador del hilo creado. El tercer argumento será el nombre de la función o la función que queremos que se ejecute dentro del hilo.

El segundo parámetro se puede utilizar para especificar atributos del hilo, pero si usamos NULL el hilo se crea con los parámetros normales.

Como se observa en el programa principal (main), se tienen dos variables de tipo **pthread_t** que van almacenar el identificador de cada uno de los dos hilos que se crean. 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 por la librería **pthread.h**.

```
# 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 ");
}
```

Tenga en cuenta que cuando compile y ejecuta el programa, debe linkarlo con la librería pthread por lo que hay que añadir a la llamada a gcc el parámetro ***-lpthread***, se añade al final.

Al ejecutar el programa no se va llegar a ver nada en la consola. La razón se debe que los hilos que se crean en el programa principal terminan automáticamente cuando el programa principal termina. Si en el programa principal solo lanzamos los dos hilos y después escribimos "Fin", lo más probable es que los hilos no lleguen a ejecutarse completamente o no lleguen ni a terminar de arrancar antes de que el programa principal escriba y termine.

Las funciones como fflush sirve para forzar el volcado de la información al dispositivo de salida. Sin embargo, fflush (stdout) se usa en el procesamiendo de mensajes de multiproceso y programación de red. La otra función llamada usleep se encarga de suspender la ejecución durante un intervalo de varios microsegundos (ms).

Por lo que, 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_t**. Para comprobarlo se añade las siguientes líneas al programa antes de la sentencia printf ("FIN") en el programa principal.

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

Como se puede ver cuando se ejecuta, **pthread_join** tiene 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. Como se va a devolver nada, entonces no es necesario utilizarlo. La librería que trabaja esta función es **unistd**

4. 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 a la función en el hilo que se ejecuta debe ser de tipo puntero a void, ya que es así como tiene que ser declarada.

Veamos un ejemplo:

```
# 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 ");
```

```
}
```

También si se tiene más de un parámetro se puede hacer uso de una estructura (struct). Agrupando los distintos parámetros en una estructura y pasando un puntero a dicha estructura, la función tendrá acceso a todos los argumentos.

```
 #include <stdio.h >
 #include <stdlib.h >
 #include <string.h >
 #include <unistd.h >
#include <pthread.h >
 struct parametros {
       int id ;
      char *nombre;
      char *apellidos;
 };
 void *datos(void * arg){

      struct parametros *p;
      p=(struct parametros *) arg;
      printf("dato id=%d\n",p->id);
      printf("dato nombre=%s\n",p->nombre);

 }

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

       struct parametros p1;
       struct parametros p2;

       p1.id=1;
       p1.nombre="juanito";
       p1.apellidos="perez";


       p2.id=2;
       p2.nombre="juanita";
       p2.apellidos="perez";
```

```
        pthread_create(&h1, NULL, datos, (void *)&p1);
        pthread_create(&h2, NULL, datos, (void *)&p2);

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

        printf("FIN \n";


    }
```

5. Liberación de recursos

Los recursos asignados por el SO 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.

Veamos un ejemplo:

```
# 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) ;
        }
```

```
        pthread_exit(&msg);

        return NULL;
}

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 ");
}
```

6. Semáforos

Para trabajar con semáforos se debe seguir la siguiente secuencia:

1. Crear semáforo
2. Operaciones sobre el semáforo (wait /signal)
3. Destruir el semáforo.

Por lo que, se debe usar las siguientes operaciones:

1. Crear un semáforo: **int sem init(sem t *sem, int pshared, unsigned value);** donde
   a) **sem:** especifica el semáforo que se inicializará.
   b) **pshared**: este argumento especifica si el semáforo recién inicializado se comparte entre procesos o subprocesos. Un valor distinto de cero significa que el semáforo se comparte entre procesos y un valor de cero significa que se comparte entre subprocesos.
   c) **valor**: especifica el valor que se asignará al semáforo recién inicializado.
2. Operación wait: **int sem wait(sem t *sem);**
3. Operación signal **int sem post(sem t *sem);**

4. Destuir un semáforo **int sem destroy(sem t *sem);**

Para declarar un semáforo, el tipo de datos es **sem_t**

Veamos un ejemplo: debe considerarse la librería **semaphore**

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

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}


int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

En código se observa que se crean 2 subprocesos, uno 2 segundos después del

primero. Pero el primer hilo dormirá 4 segundos después de adquirir el bloqueo. Por lo tanto, el segundo hilo no ingresará inmediatamente despu'es que se llame, ingresará 4-2=2 segundos después que se llame.

7. Mutex: acceso controlado a objetos compartidos

Son funciones POSIX que incluyen mecanismos de exclusión mutua, mecanismo de señalización del cumplimiento de condiciones por parte de variables, y mecanismos de acceso de variables que se modifican en forma exclusiva, pero pueden ser leídas en forma compartida. Las funciones para el manejo de zonas de acceso exclusivo tienen el prefijo **pthread_mutex**

Un mutex es una variable especial que puede tener estado tomado (**locked**) o libre (**unlocked**). Es como una compuerta que permite el acceso controlado. Si un hilo tiene el mutex entonces se dice que es el dueño del mutex. Si ningún hilo lo tiene se dice que está libre (o unlocked). Cada mutex tiene una cola de hilos que están esperando para tomar el mutex. El uso de mutex es eficiente, pero debería ser usado sólo cuando su acceso es solicitado por corto tiempo.

1. **pthread_mutex_destroy:** destruye la variable (de tipo pthread_mutex_t) usada para manejo de exclusión mutua, o candado mutes.
2. **pthread_mutex_init:** permite dar las condiciones iniciales a un candado mutex
3. **pthread_mutex_lock:** permite solicitar acceso al mutex, el hilo se bloquea hasta su obtención
4. **pthread_mutex_trylock:**permite solicitar acceso al mutex, el hilo retorna inmediatamente. El valor retornado indica si otro hilo lo tiene.
5. **pthread_mutex_unlock:** permite liberar un mutex.

Veamos como la creación e iniciación de mutex

```
#include <pthread.h>
int  pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t * attr);
```

Alternativamente se puede 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);
```

Destrucción de un mutex

```
#include <pthread.h>
int  pthread_mutex_destroy(pthread_mutex_t * mutex);
```

Solicitud y liberación de un mutex

```
  #include <pthread.h>
int  pthread_mutex_lock(pthread_mutex_t * mutex);
int  pthread_mutex_trylock(pthread_mutex_t * mutex);
int  pthread_mutex_unlock(pthread_mutex_t * mutex);
```

Con pthread_mutex_trylock el hilo siempre retorna, si la función es exitosa, se retorna 0 como en los otros casos; si no se retornará EBUSY indicando que otro hilo tiene el mutex.
Veamos un ejemplo para proteger la zona crítica, se debe usar:

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mylock);
/* Sección crítica */
pthread_mutex_unlock(&mylock);
```

Ejemplo de dos funciones una (1) **Sin Mutex** y (2) **Con Mutex**

(1) Sin Mutex

```
int counter=0;
void functionC(){
   counter++;
}
```

(2) Con Mutex

```
/* Note scope of variable and mutex are the same */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter=0;

/* Function C */
void functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

Veamos un ejemplo donde se crean dos hilos, los cuales realizan una suma del arreglo respetando la región critica para no suma la misma casilla dos veces (Autor del código : Aquiles Lazaro)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 2
#define MAX 100000
//Variables globales
int arreglo[MAX];
int total=0;
int indice=0;
//Crea la variable mutex (o candado) que puede tener el valor lock (bloqueado) y unlock
pthread_mutex_t m1;
pthread_mutex_t m2;
//Funcion que ejecutan los hilos creados
void *PrintHello(void *threadid){
    int *id_ptr, taskid;
    int miIndice;
    int parcial=0;
    //Para saver el numero del hilo
id_ptr = (int *) threadid;
taskid = *id_ptr;
//Ciclo para sumar todos los elementos del arreglo
while(indice < MAX){
/*Para la exclusion mutua el hilo se detiene si otro esta dentro de la region critica*/
```

```
pthread_mutex_lock(&m1);
miIndice = indice;
indice++;
//Libera la region critica para que otro hilo entre
pthread_mutex_unlock(&m1);
parcial = parcial + arreglo[miIndice];
}
printf("Hilo %d termine el ciclo, mi parcial es: %d\n",taskid, parcial);
//Region critica para no sobrescribir el total
pthread_mutex_lock(&m2);
total = total + parcial;
pthread_mutex_unlock(&m2);
//Termina el proceso del hilo
    pthread_exit(NULL);
}

int main(){

pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t;
//Se inician las variables candado
pthread_mutex_init (&m1, NULL);
pthread_mutex_init (&m2, NULL);
//Llenado del arrreglo
for(t=0; t<MAX; t++){
arreglo[t]=1;
}
arreglo[MAX]=3;
//Creacion de los hilos
for(t=0;t<NUM_THREADS;t++) {
   taskids[t] = (int *) malloc(sizeof(int));
   *taskids[t] = t;
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
}
//Para esperar los hilos creados
for(t=0;t<NUM_THREADS;t++){
  pthread_join(threads[t],NULL);
}
printf("el total es: %d\n", total);
```

```
}
```

8. Problema

Ana y Pepe abrieron una cuenta en la tienda de la esquina, para hacer las compras diarias. Ellos van haciendo las compras cuando necesitan, pero como son muy independientes, no se avisan del último saldo disponible. Como al dueño de la tienda lo estafaron un par de veces, no deja que sus clientes se pasen del saldo que tienen en la cuenta. Pepe se propone hacer un pequeño sistema que permita administrar el dinero de la cuenta que tienen en el almacén.

1. Cree hilos que permita recrear el problema.
2. Utilice el mecanismo de sincronización más apropiado para resolver este problema.