PARTE II
PROGRAMACION CON THREADS EN C

II.1 INTRODUCCION

Una librería o paquete de threads permite escribir programas con varios puntos simultáneos de ejecución, sincronizados a través de memoria compartida. Sin embargo la programación con hilos introduce nuevas dificultades. La programación concurrente tiene técnicas y problemas que no ocurren en la programación secuencial. Algunos problemas son sencillos (por ejemplo el ínter bloqueo) pero algunos otros aparecen como penalizaciones al rendimiento de la aplicación.

Un thread es un concepto sencillo: *un simple flujo de control secuencial*. Con un único thread existe en cualquier instante un único punto de ejecución. El programador no necesita aprender nada nuevo para programar un único thread. Sin embargo, cuando se tienen múltiples hilos en un programa significa que en cualquier instante el programa tiene múltiples puntos de ejecución, uno en cada uno de sus threads. El programador decide cuando y donde crear múltiples threads, ayudándose de una librería o paquete en tiempo de ejecución.

En un lenguaje de alto nivel, las variables globales son compartidas por todos los threads del programa, esto es, los hilos leen y escriben en las mismas posiciones de memoria. El programador es el responsable de emplear los mecanismos de sincronización de la librería de hilos proporcionada por el lenguaje para garantizar que la memoria compartida se acceda de forma correcta por los hilos.Las facilidades proporcionadas por la librería de hilos que se utilice son conocidas como *primitivas ligeras*, lo que significa que las primitivas de:

- ☐ Creación
- ☐ Mantenimiento
- ☐ Sincronización y
- ☐ Destrucción

Son suficientemente económicas en esfuerzo para las necesidades del programador.

II.2. EL ESTANDAR POSIX THREADS

Uno de los problemas que se tenían al utilizar múltiples threads de ejecución es que hasta hace poco no existía un estándar para ello. La extensión POSIX.1c se aprobóen Junio de 1995. Con la adopción de un estándar POSIX para los hilos, se están haciendo más comunes las aplicaciones con Threads.

El estándar POSIX Threads significa técnicamente el API Thread especificado por el estándar formal internacional POSIX 1003.1c-1995. POSIX significa: Portable Operating System Interface. Todas las fuentes que empleen POSIX 1003.1c, POSIX.1c o simplemente Pthreads, deben incluir el archivo de encabezado pthread.h con la directiva:

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

Ya que pthread es una librería POSIX, se podrán portar los programas hechos con ella a cualquier sistema operativo POSIX que soporte threads. Por tanto, para crear programas que hagan uso de la librería *pthread.h* necesitamos en primer lugar la librería
en sí. Esta viene en la mayoría de las distribuciones de LINUX y si no es así, se puede bajar de la red.

Una vez que tenemos la librería instalada, deberemos compilar el programa y ligarlo con dicha librería en base al compilador que se utilice. La librería de hilos POSIX.1c debe ser la última librería especificada en la línea de comandos del compilador:

CC . . . –lpthread

Por ejemplo, la forma más usual de hacer lo anterior, si estamos usando un compilador GNU como *gcc*, es con el comando:

```
gcc prog_con_hilos.c –o prog_con_hilos_ejecutable -lpthread
```

En POSIX.1c todos los hilos de un proceso comparten las siguientes características:

- ☐ El espacio de direcciones
- ☐ El ID del proceso
- ☐ EL ID del proceso padre
- ☐ El ID del proceso líder del grupo
- ☐ Los identificadores de usuario
- ☐ Los identificadores de grupo
- ☐ El directorio de trabajo raíz y actual
- ☐ La máscara de creación de archivos
- ☐ La tabla de descriptores de archivos
- ☐ El timer del proceso

Por otro lado, cada hilo tiene la siguiente información específica:

- ☐ Un identificador de hilo único
- ☐ La política de planificación y la prioridad
- ☐ Una variable errno por hilo
- ☐ Datos específicos por hilo
- ☐ Gestores de cancelación por hilo
- ☐ Máscara de señales por hilo

Las operaciones llevadas a cabo sobre un hilo son:

- ☐ Creación y destrucción
- ☐ Sincronización entre hilos
- ☐ Posibilidad de disponer para cada thread memoria local propia

PROGRAMA 1. Creacion de Hilos

```
/* Creamos MAX_THREAD threads que sacan por pantalla una cadena y su
identificador. Una vez que terminan su ejecución devuelven como resultado su
identificador */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS       10

pthread_t tabla_thr[MAX_THREADS]; //tabla con los identificadores de los threads

typedef struct // tipo de datos y tabla con los parámetros
{
int id;
char *cadena;
} thr_param_t;

thr_param_t param[MAX_THREADS];

//tenemos que crear una tabla para los parámetros porque los pasamos
//por referencia. Así, si solo tuviéramos una variable para
        los parámetros al modificar esta modificaríamos todas las que habíamos
//pasado anteriormente porque los threads no se quedan con el valor
//sino con la direccion

// Esta es la funcion que ejecutan los threads

 void * funcion_thr(void *p)
{
thr_param_t *datos;
datos= (thr_param_t *)p;
printf("%s %d\n", datos->cadena, datos->id);
pthread_exit(&(datos->id)); //una vez terminamos, devolvemos el  valor
}

int main(void)
{
int i, *res;
void * r;

printf("Creando threads...\n");
for (i=0; i<MAX_THREADS; i++)
{
param[i].cadena="Hola, soy el thread";
param[i].id= i;

pthread_create(&tabla_thr[i],NULL,funcion_thr,(void*)&param[i]);
}
```

```
printf("Threads creados. Esperando que terminen...\n");
for (i=0; i<MAX_THREADS; i++)
{
pthread_join(tabla_thr[i],&r);
res=(int *)r;
printf("El thread %d devolvio el valor %d\n", i, *res);
}

printf("Todos los threads finalizados. Adios!\n");
system("PAUSE");
return 0;
}
```

Responde las siguientes preguntas:

a)¿ Cual es la salida de este programa ?
b) ¿ Su salida es siempre la misma?

Todo este procedimiento se remite a tres pasos:

1. Crear el (los) thread(s)
2. Esperar a que terminen
3. Recoger y procesar el (los) resultado(s)

A esto se le llama PARALELISMO ESTRUCTURADO.

PROGRAMA 2. Realizaremos la aplicación más típica de todo lenguaje de programación, el programa HOLA MUNDO en su versión multi-hilo. La aplicación imprimirá el mensaje “HOLA MUNDO” en la salida estándar (stdout). El código para la versión secuencial (no multi-hilo) de Hola Mundo es:

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

void imprimir_mensaje( void *puntero );

int main() {
char *mensaje1 = "Hola";
char *mensaje2 = "Mundo"; imprimir_mensaje((void *)mensaje1);
imprimir_mensaje((void *)mensaje2); printf("\n");
system("PAUSE");
exit(0);
return 1;
}

void imprimir_mensaje( void *puntero )
{
char *mensaje;
mensaje = (char *) puntero;
printf("%s ", mensaje);
}
```

Para la versión multihilo necesitamos dos variables de hilo y una función de comienzo para los nuevos hilos que será llamada cuando comiencen su ejecución. También necesitamos alguna forma de especificar que cada hilo debe imprimir un mensaje diferente. Una aproximación es dividir el trabajo en cadenas de caracteres distintas y proporcionar a cada hilo una cadena diferente como parámetro. Examinemos el programa 3:

PROGRAMA 3. El programa crea el primer hilo llamando a pthread_create( ) y pasando "Hola" como argumento inicial. El segundo hilo es creado con "Mundo" como argumento. Cuando el primer hilo inicia la ejecución, comienza en la función imprimir_mensaje( ) con el argumento "Hola". Imprime "Hola" y finaliza la función. Un hilo termina su ejecución cuando finaliza su función inicial. Así pues, el primer hilo termina después de imprimir el mensaje "Hola". Cuando el 2do. Hilo se ejecuta, imprime "Mundo" y al igual que el anterior hilo, finaliza.

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

void * imprimir_mensaje( void *puntero );

int main()
{
pthread_t hilo1, hilo2; char *mensaje1 = "Hola"; char *mensaje2 = "Mundo";

pthread_create(&hilo1,NULL,imprimir_mensaje,(void*)
mensaje1);
pthread_create(&hilo2,NULL,imprimir_mensaje,(void*)
mensaje2); printf("\n\n"); system("PAUSE"); exit(0);
return 1;
}

void * imprimir_mensaje( void *puntero )
{
char *mensaje;
mensaje = (char *) puntero; printf("%s ", mensaje);
return puntero;
}
```

Aunque este programa parece correcto, posee errores que debemos considerar como defectos considerables:

1. El defecto más importante es que los hilos pueden ejecutarse concurrentemente. No existe entonces garantía de que el primer hilo ejecute la función *printf()* antes que el segundo hilo. Por tanto, podríamos ver el mensaje *"Mundo Hola"* en vez de *"Hola Mundo"*.

2. Por otro lado existe una llamada a *exit( )* realizada por el hilo padre en la función principal *main( )*. Si el hilo padre ejecuta la llamada a *exit( )* antes de que alguno de los hilos ejecute *printf( )*, la salida no se generará completamente. Esto es porque la función *exit( )* finaliza el proceso (libera la tarea) y termina todos los hilos. Cualquier hilo padre o hijo que realice la llamada a *exit( )* puede terminar con el resto de los hilos del proceso. Los hilos que deseen finalizar explícitamente, deben utilizar la función *pthread_exit( )*.

Así nuestro programa *"Hola Mundo"* tiene 2 condiciones de ejecución (race conditions) o condiciones problemáticas:

- ☐ La posibilidad de ejecución de la llamada a *exit( )*
- ☐ La posibilidad de qué hilo ejecutará la función *printf( )* primero.

Una forma de arreglar estas dos condiciones de manera artesanal es la siguiente, mostrada en el programa 4.

PROGRAMA 4. Como queremos que cada hilo finalice antes que el hilo padre, podemos insertar un retraso en el hilo padre para dar tiempo a los hilos hijos a que ejecuten printf(). Para asegurar que el primer hilo ejecute printf() antes que el segundo hilo, podríamos insertar un retraso antes de crear el segundo hilo. El código resultante es entonces:

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

void * imprimir_mensaje( void *puntero );

int main()
{
pthread_t hilo1, hilo2; char *mensaje1 = "Hola"; char *mensaje2 = "Mundo";
pthread_create(&hilo1,NULL,imprimir_mensaje,(void*)mensaje1); sleep(100);
pthread_create(&hilo2,NULL,imprimir_mensaje,(void*)mensaje2); sleep(100);
system("PAUSE");
exit(0);
return 1;
}

void * imprimir_mensaje( void *puntero )
{
char *mensaje;
mensaje = (char *) puntero; printf("%s ", mensaje); pthread_exit(0);
}
```

Este código parece funcionar bien, sin embargo no es seguro. Nunca es seguro confiar en los retrasos de tiempo. La condición problemática aquí es exactamente la misma que se tiene con una aplicación distribuida y un recurso compartido.
El recurso es la salida estándar y los elementos de la computación distribuida son los tres hilos. El hilo 1 debe usar la salida estándar antes que el hilo 2 y ambos deben realizar sus acciones antes de que el hilo padre realice la llamada a exit( ).

Además de nuestros intentos por sincronizar los hilos mediante retrasos, hemos cometido otro error. La función sleep( ), al igual que la función exit( ) es relativa a procesos. Cuando un hilo ejecuta sleep( ) el proceso entero duerme, es decir, todos los hilos duermen cuando el proceso duerme. Nos encontramos por tanto en la misma situación que teníamos sin las llamadas a sleep() y encima el programa tarda 200 milisegundos más en su ejecución.

PROGRAMA 5. La versión correcta de "HOLA MUNDO" empleando primitivas de sincronización que evitan las condiciones de ejecución no deseadas, se presenta en el siguiente código:

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

typedef struct param
{
char *mensaje;
pthread_t hilo;
} param_t;

void * imprimir_mensaje( void *puntero )
{
param_t *datos;

datos = (param_t *) puntero;

if (datos->hilo != 0)
pthread_join( datos->hilo, NULL );

printf("%s ", datos->mensaje);
pthread_exit( 0 );
}


int main()
{
pthread_t hilo1, hilo2;
param_t mensaje1 = {"Hola", 0};
param_t mensaje2 = {"Mundo", 0};

pthread_create(&hilo1,NULL,imprimir_mensaje,(void*)&mensaje1);
 mensaje2.hilo = hilo1;
pthread_create(&hilo2,NULL,imprimir_mensaje,(void*)&mensaje2);
```

```
pthread_join( hilo2, NULL );

system("PAUSE");
exit(0);
return 1;
}
```

En esta versión, el parámetro de la función imprimir_mensaje( ) es del tipo definido por el usuario param_t que es una estructura que nos permite pasar el mensaje
y el identificador de un hilo.

El hilo padre tras crear el hilo 1, obtiene su identificador y lo almacena como parámetro para el hilo 2. Ambos hilos ejecutan la función inicial imprimir_mensaje( ), pero la ejecución es distinta para cada hilo.

El primer hilo contendrá un 0 (cero) en el campo hilo de la estructura de datos con lo que no ejecutará la función pthread_join( ); ejecutará printf( ) y la función *pthread_exit( )*. El segundo hilo en cambio, contendrá el identificador del primer hilo en el campo *hilo* de la estructura de datos y ejecutará la función *pthread_join( )*, que lo que hace es esperar a la finalización del primer hilo. Después ejecuta la impresión con *printf( )* y después realiza la finalización con *pthread_exit( )*. El hilo padre, tras la creación de los 2 hilos se sincroniza con la finalización del hilo 2 mediante *pthread_join( )*.