
Taller 1: Hilos (*Threads*)

Fecha de asignación:	10 de marzo
Grupo:	1 persona

Fecha de entrega:	15 de marzo
Profesor:	Luis Barboza Artavia

Estudiante: Shakime Richards Sparks

Carné: 2018170667

2. Investigación

1. Investigue posibles métodos (bibliotecas, apis, etc) para el uso de hilos bajo el sistema operativo GNU Linux. (Por ejemplo *pthread*).

En GNU Linux existen diferentes métodos para el uso de hilos que permiten la ejecución concurrente de tareas en un mismo proceso. A continuación, se presentan algunos de los métodos más comunes:

Según [1], **POSIX Threads**, es una API para hilos en sistemas tipo Unix basada en estándares para C/C++. Permite generar un nuevo flujo de proceso concurrente proporcionando funciones para crear, destruir, sincronizar y comunicar hilos en un proceso.

C++11 Threads [2] es uno de los métodos más utilizados. A partir de la versión C++11, el estándar del lenguaje incluye soporte nativo para hilos mediante la clase `std::thread`. Esta clase proporciona métodos para crear, unir y gestionar hilos en un proceso. En cada aplicación C++ hay un hilo principal predeterminado, el `main()`. Sin embargo, se pueden crear hilos adicionales creando objetos de la clase `std::thread`, cada uno de los objetos `std::thread` puede estar asociado con un hilo.

GNU Parallel es una herramienta de línea de comandos que permite la ejecución paralela de scripts, comandos de shell o de tareas en sistemas GNU Linux. Se utiliza para distribuir tareas entre múltiples procesadores o núcleos en un equipo, es software libre, fue escrito por Ole Tange en Perl y está disponible bajo los términos de GPLv3. [3]

Otra biblioteca de C++ que proporciona una API para trabajar con hilos es **Boost.Thread**, conjuntamente de las funciones básicas para utilizar hilos en un proceso, también brinda características avanzadas como variables condicionales, mutex y semáforos. La primera

versión fue originalmente escrita y diseñada por William E. Kempf. La segunda versión, de Anthony Williams, es una reescritura importante que sigue de cerca la propuesta presentada al comité de estándares C++. Finalmente, una tercera versión escrita por Vicente J. Botet, se adaptó para que coincida con la reconocida biblioteca Thread C++11. [4]

Hay muchos otros métodos, y cada uno de estos métodos tiene sus propias características y ventajas, es necesario elegir el método apropiado para cada situación de desarrollo paralelo.

2. ¿Qué es el concepto de *mutex* en multiprogramación y qué busca hacer?

Según [5], en multiprogramación, un mutex (mutual exclusión) es un mecanismo de sincronización utilizado para evitar que múltiples procesos o hilos accedan simultáneamente a un recurso compartido, puede ser una variable compartida en memoria, un archivo e incluso una sección crítica de código.

El objetivo del mutex es garantizar que solo uno de los hilos tenga acceso exclusivo a los recursos, ya sea para leer, escribir o modificar datos o para ejecutar cierto código de aplicación, en un momento determinado. Mientras, los demás hilos deben esperar hasta que el recurso quede libre. Esto se logra a través de la implementación de una sección crítica protegida por el mutex, que permite que los procesos o hilos obtengan el derecho de acceso exclusivo a ese recurso.

Por lo general funciona así, "cuando se inicia un programa, primero crea un mutex para un recurso específico al solicitarlo al sistema, y el sistema le devuelve un nombre o ID. Cualquier hilo que necesite el recurso debe usar el mutex para bloquear el recurso de otros hilos mientras lo usa. Si el mutex ya está bloqueado, el sistema usualmente pone en cola al hilo que necesita el recurso y luego le pasa el control cuando la mutex está desbloqueado. Luego nuevamente, el mutex será bloqueado por el nuevo hilo proveniente de la cola" [5].

3. ¿Qué sucede cuando dos hilos quieren utilizar el mismo recurso? ¿Cómo se manejan estos casos?

Cuando dos o más intentan acceder al mismo recurso simultáneamente, se produce una situación conocida **race condition**. Este entorno ocurre cuando los resultados de la ejecución del programa dependen del orden en que los hilos acceden al recurso compartido [7].

Un ejemplo de este contexto [8]:

```
for ( int i = 0; i < 10000000; i++ )  
{  
    x = x + 1;  
}
```

Si se tienen a la vez 5 hilos ejecutando este código, el valor de x no terminaría siendo 50,000,000. Más bien, variaría con cada ejecución.

Cada hilo debe 1) recuperar el valor de x , 2) sumar 1 a este valor y 3) almacenar el nuevo valor en x .

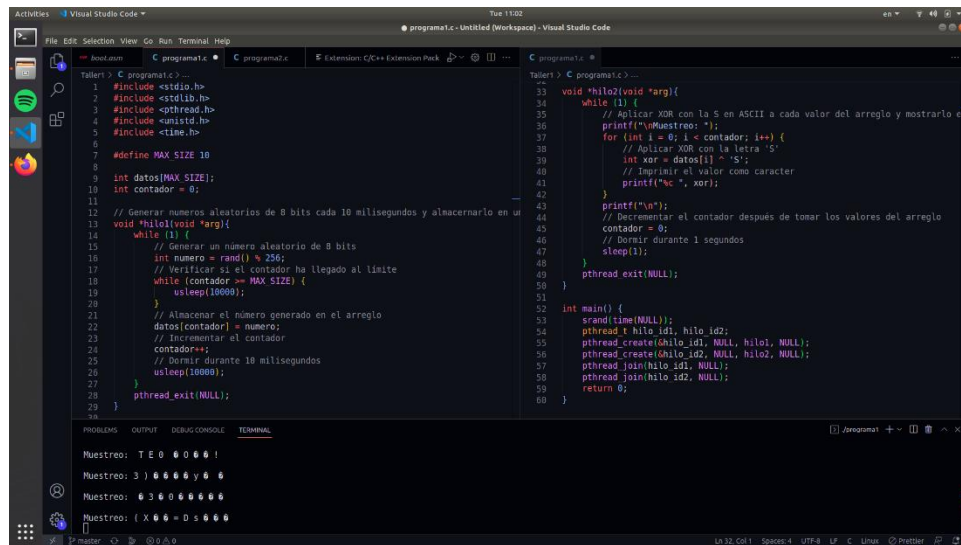
Cualquier hilo puede estar en cualquier etapa del proceso en cualquier momento y pueden interferir entre sí al procesar recursos compartidos. En el tiempo entre la lectura de x y la reescritura de x , otro hilo puede cambiar el estado de x .

Como se observa, en una situación de race condition los hilos pueden producir resultados inesperados, pues no se cumple con certeza el mismo orden de ejecución cada vez que se ejecuta el programa. Asimismo, existe la posibilidad de que suceda un **deadlock**, esto ocurre cuando dos hilos bloquean cada uno una variable distinta y luego intentan bloquear la variable que ha sido bloqueada por el otro hilo; como consecuencia ambos hilos esperan a que la variable sea liberada y no se termina de ejecutar el programa, pues ambos hilos permanecen bloqueados [7].

Como se menciona en [8], para evitar estos casos se utilizan mecanismos de sincronización, como los mutex, semáforos y monitores, que garantizan que solo un hilo tenga acceso al recurso compartido en un momento dado, lo que asegura la exclusión mutua. Los semáforos funcionan como contadores que controlan el acceso a un recurso, mientras que los monitores son objetos que contienen métodos que se ejecutan específicamente para evitar problemas de sincronización.

Desarrollo

1. Diseñe una aplicación en C que interactúe con 2 hilos. El primer hilo simulará el proceso de muestreo, por lo que debe generar números aleatorios de 8 bits (entre 0 y 255) y almacenarlo en un arreglo cada 10 milisegundos. Simultáneamente, el segundo hilo debe tomar los datos uno a uno, aplicar una operación *XOR* a una constante definida por el estudiante y mostrar el dato como caracter (char) en la terminal.



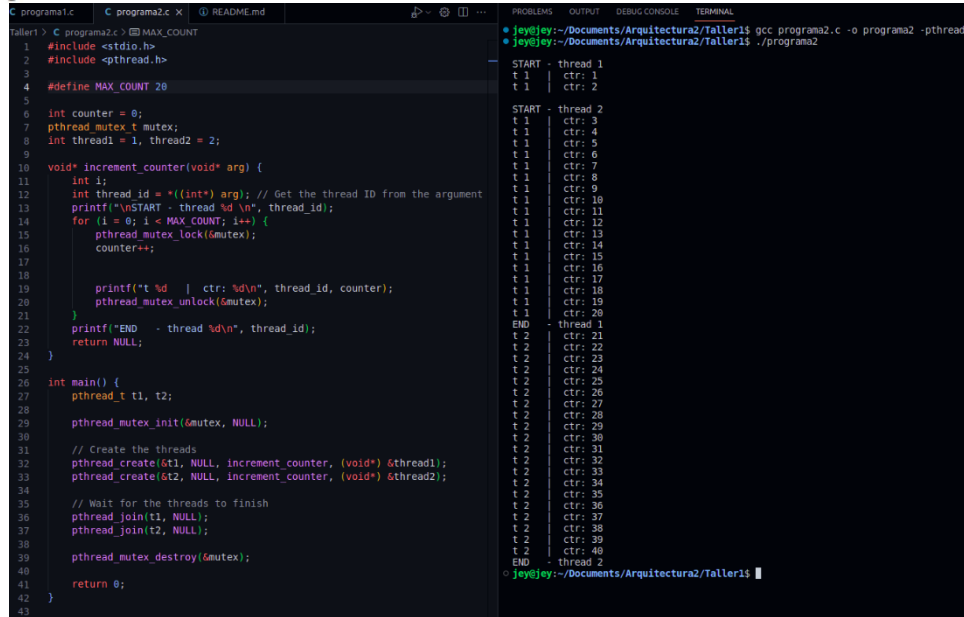
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5 #include <time.h>
6
7 #define MAX_SIZE 10
8
9 int datos[MAX_SIZE];
10 int contador = 0;
11
12 // Generar numeros aleatorios de 8 bits cada 10 milisegundos y almacenarlo en un arreglo
13 void hilo1(void *arg){
14     while (1) {
15         // Generar un numero aleatorio de 8 bits
16         int numero = rand() % 256;
17         // Verificar si el contador ha llegado al limite
18         while (contador >= MAX_SIZE) {
19             usleep(10000);
20         }
21         // Almacenar el numero generado en el arreglo
22         datos[contador] = numero;
23         // Incrementar el contador
24         contador++;
25         // Dormir durante 10 milisegundos
26         usleep(10000);
27     }
28     pthread_exit(NULL);
29 }
30
31 void hilo2(void *arg){
32     while (1) {
33         // Aplicar XOR con la 5 en ASCII a cada valor del arreglo y mostrarlo en terminal
34         printf("Muestreo: ");
35         for (int i = 0; i < contador; i++) {
36             // Aplicar XOR con la letra 'S'
37             int xor = datos[i] ^ 'S';
38             // Imprimir el valor como caracter
39             printf("%c ", xor);
40         }
41         printf("\n");
42         // Decrementar el contador despues de tomar los valores del arreglo
43         contador = 0;
44         // Dormir durante 1 segundos
45         sleep(1);
46     }
47     pthread_exit(NULL);
48 }
49
50 int main() {
51     srand(time(NULL));
52     pthread_t hilo_id1, hilo_id2;
53     pthread_create(&hilo_id1, NULL, hilo1, NULL);
54     pthread_create(&hilo_id2, NULL, hilo2, NULL);
55     pthread_join(hilo_id1, NULL);
56     pthread_join(hilo_id2, NULL);
57     return 0;
58 }
```

Terminal output:

```
Muestreo: T E 0 0 0 0 0 0 0 0 0 0
Muestreo: 3 ) 0 0 0 0 0 0 0 0 0 0
Muestreo: 0 3 0 0 0 0 0 0 0 0
Muestreo: ( X 0 0 0 0 0 0 0 0 0 0
```

Algunos de los códigos ASCII resultantes no son reconocidos por la terminal, especialmente si se trata de valores que no corresponden a caracteres imprimibles. Para cambiar a numérico basta con cambiar 'S' por un 83 (en la línea 39) y el "%c" por un %d (en la línea 41).

2. Diseña una aplicación en C que interactúe con 2 hilos. Ambos hilos van a ejecutar una misma función la cual hace escritura sobre un contador compartido. Como ambos hilos van a utilizar un elemento compartido, este elemento se debe bloquear, por medio de mutex. Muestre los *prints* de inicio y fin de la ejecución de cada hilo. Detalle lo sucedido en este problema.



```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 #define MAX_COUNT 20
5
6 int counter = 0;
7 pthread_mutex_t mutex;
8 int thread1 = 1, thread2 = 2;
9
10 void* increment_counter(void* arg) {
11     int i;
12     int thread_id = *((int*) arg); // Get the thread ID from the argument
13     printf("\nSTART - thread %d\n", thread_id);
14     for (i = 0; i < MAX_COUNT; i++) {
15         pthread_mutex_lock(&mutex);
16         counter++;
17
18         printf("t %d | ctr: %d\n", thread_id, counter);
19         pthread_mutex_unlock(&mutex);
20     }
21     printf("END - thread %d\n", thread_id);
22     return NULL;
23 }
24
25
26 int main() {
27     pthread_t t1, t2;
28
29     pthread_mutex_init(&mutex, NULL);
30
31     // Create the threads
32     pthread_create(&t1, NULL, increment_counter, (void*) &thread1);
33     pthread_create(&t2, NULL, increment_counter, (void*) &thread2);
34
35     // Wait for the threads to finish
36     pthread_join(t1, NULL);
37     pthread_join(t2, NULL);
38
39     pthread_mutex_destroy(&mutex);
40
41     return 0;
42 }
43
```

```
START - thread 1
t 1 | ctr: 1
t 1 | ctr: 2
...
END - thread 1
START - thread 2
t 2 | ctr: 3
t 2 | ctr: 4
...
END - thread 2
```

Lo que sucede es que tenemos una variable compartida counter a la que dos hilos están accediendo simultáneamente y haciendo incrementos en ella. Para evitar problemas de concurrencia, utilizamos un mutex para bloquear el acceso a la variable compartida y garantizar que solo un hilo acceda a ella a la vez. Al emplear el mutex, los hilos se bloquean mutuamente y esperan a que el mutex esté disponible antes de acceder a la variable compartida. Esto garantiza que los incrementos se hagan de forma segura y que no se produzcan condiciones de carrera o errores de sincronización. Como se observa, el valor final es múltiplo de 20, puesto que la cuenta es de 20 en 20. En este caso son 2 hilos entonces 40, si fuesen 3 entonces 60, y así sucesivamente.

3. Referencias

- [1] G. Ippolito, "Linux Tutorial: POSIX Threads," *Cmu.edu*, 2020.
<https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>
- [2] "std::thread - cppreference.com," *en.cppreference.com*.
<https://en.cppreference.com/w/cpp/thread/thread>
- [3] "GNU Parallel - GNU Project - Free Software Foundation," www.gnu.org.
<https://www.gnu.org/software/parallel>
- [4] "Chapter 38. Thread 4.8.0 - 1.79.0," www.boost.org.
https://www.boost.org/doc/libs/1_79_0/doc/html/thread.html
- [5] "i5/OS: Mutexes and threads," www.ibm.com.
<https://www.ibm.com/docs/en/i/7.2?topic=threads-mutexes>
- [6] "What is mutex (mutual exclusion object)? - Definition from WhatIs.com," *SearchNetworking*. <https://www.techtarget.com/searchnetworking/definition/mutex>
- [7] HaiyingYu, "Race conditions and deadlocks - Visual Basic," *learn.microsoft.com*.
<https://learn.microsoft.com/en-us/troubleshoot/developer/visualstudio/visual-basic/language-compilers/race-conditions-deadlocks>
- [8] "multithreading - What is a race condition?," *Stack Overflow*.
<https://stackoverflow.com/questions/34510/what-is-a-race-condition>
- [9] "MECANISMOS PARA SINCRONIZACIÓN Semáforos."
https://www.cs.buap.mx/~mtovar/doc/PCP/Unidad3Semaforos_1.pdf

