Programación Paralela y Concurrente (CI-0117) - Uso de pthread

Esteban Rodríguez Betancourt

2024-03

Introducción

En esta práctica vamos a usar la biblioteca `pthread` para crear programas que hagan uso de múltiples hilos de ejecución. La biblioteca `pthread` es una implementación de POSIX Threads, que es un estándar para la creación y manejo de hilos de ejecución en sistemas operativos tipo UNIX. Pueden realizar esta práctica en cualquier sistema operativo que soporte `pthread`, como Linux, macOS o Windows (usando WSL).

1. Creación de un hilo

Los hilos se distinguen de los procesos en que comparten el espacio de direcciones del proceso que los creó. ¡No son copias! Esto significa que los hilos pueden acceder a las mismas variables globales y a las mismas variables locales de la función que los creó. Para crear un hilo en C, se utiliza la función `pthread_create`. A continuación se muestra un ejemplo de cómo crear un hilo que imprime un mensaje en la consola.

```c
// pthread01.c
#include <stdio.h>
#include <pthread.h>

void *print_message(char *message) {
    printf("%s\n", message);
    pthread_exit(NULL);
}

int main(void) {
    pthread_t thread;
    int argument = 42;

    if (pthread_create(&thread, NULL, print_message, &argument) != 0) {
        perror("Error creating thread\n");
        return 1;
    }

    pthread_join(thread, NULL);

    return 0;
}
```

Para compilar el programa anterior, se debe incluir la biblioteca `pthread` al momento de compilar. En sistemas UNIX, se debe agregar la opción `-pthread` al comando `gcc`.

```
gcc -pthread -o pthread01 pthread01.c
```

Corran el programa y reporten el resultado.

2. Los hilos comparten memoria

En la práctica de `fork` vimos que esta llamada hace una copia de la memoria del programa, y que no es posible afectar la memoria de los otros procesos (a menos que se use memoria compartida). En el caso de los hilos, los hilos comparten la memoria del proceso que los creó. Consideren el siguiente programa:

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

#define TOTAL_THREADS 16

int global = 0;

void *increment(void *arg) {
    int readGlobal;
    for (int i = 0; i < 1000; i++) {
        readGlobal = global;
        readGlobal++;
        global = readGlobal;
    }
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[TOTAL_THREADS];

    printf("Global before: %d\n", global);

    for (int i = 0; i < TOTAL_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, increment, NULL) != 0) {
            perror("Error creating thread\n");
            return 1;
        }
    }

    for (int i = 0; i < TOTAL_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Global after: %d\n", global);

    return 0;
}
```

Compilen el programa anterior y ejecútenlo varias veces. ¿Qué observan? ¿Por qué ocurre esto?

2.2 Errores de sincronización

Es evidente que el programa anterior tiene un problema de sincronización. Al ser un programa pequeño el problema es evidente, pero en programas más grandes, los problemas de sincronización pueden ser difíciles de encontrar. Para detectar estos problemas podemos usar herramientas como `helgrind` o `ThreadSanitizer`.

Helgrind

`Helgrind` es una herramienta que detecta errores de sincronización en programas que usan `pthread`. Para usar `helgrind`, se debe compilar el programa con la opción `-g` para incluir información de depuración y luego ejecutar el programa con `valgrind` y la opción `--tool=helgrind`.

```
gcc -pthread -g -o pthread02 pthread02.c
valgrind --tool=helgrind ./pthread02
```

Ejecuten el programa anterior con `helgrind` y reporten los resultados.

> **NOTA**: `Helgrind` solamente está disponible en Linux. Si están usando macOS y quieren usar `helgrind`, pueden usar Multipass para crear una máquina virtual de Ubuntu y ejecutar el programa en ella.

ThreadSanitizer

`ThreadSanitizer` es una herramienta que detecta errores de sincronización en programas que usan `pthread`. Para usar `ThreadSanitizer`, se debe compilar el programa con la opción `-fsanitize=thread` y luego ejecutar el programa.

```
gcc -pthread -fsanitize=thread -o pthread02gcc pthread02.c
./pthread02
```

También pueden compilar con Clang.

```
clang -pthread -fsanitize=thread -o pthread02clang pthread02.c
./pthread02
```

> **NOTA**: Si están en macOS, es posible que gcc en realidad sea un alias de Clang. En ese caso, no tiene sentido realizar la comparación.

Ejecuten el programa anterior con `ThreadSanitizer` y reporten los errores detectados. ¿GCC reporta errores diferentes que la versión compilada con Clang?

> **NOTA**: `ThreadSanitizer` añade instrumentación al programa para detectar errores de sincronización. Por lo tanto, el programa puede ser de 5 a 15 veces más lento que el programa original y usar de 5 a 10 veces más memoria. Por lo tanto, NO se recomienda usar `ThreadSanitizer` en programas grandes o en producción.

3 Atomicidad de las operaciones

El problema anterior se debe a que estamos realizando el incremento de una forma *no atómica*. Se dice que una operación es atómica si se ejecuta en un solo paso, sin posibilidad de interrupción. En el ejercicio anterior, la operación de incremento no es atómica, ya que se descompone en tres pasos: leer el valor de `global`, incrementar el valor leído y escribir el valor incrementado en `global`. Si un hilo lee el valor de `global`, y antes de incrementarlo, otro hilo lee el valor de `global`, ambos hilos incrementarán el valor leído y escribirán el valor incrementado en `global`, perdiendo uno de los incrementos.

Copien el programa anterior a un archivo llamado `pthread03.c` y modifíquenlo para que `increment` sea así:

```
void *increment(void *arg) {
    for (int i = 0; i < 1000; i++) {
        global++;
    }
    pthread_exit(NULL);
}
```

Compilen y ejecuten el programa varias veces. Reporten ¿Qué observan? ¿Por qué ocurre esto?

3.2 Instrucciones atómicas

Algunas operaciones son muy comunes y se realizan con frecuencia en programas concurrentes. Para mejorar el rendimiento en estos casos, los procesadores modernos proveen instrucciones atómicas para realizar estas operaciones. En el caso de C, existe la biblioteca `stdatomic.h` que provee funciones para realizar operaciones atómicas. El equivalente en C++ es la biblioteca `atomic`.

Copien el programa anterior a un archivo llamado `pthread03atomic.c` y modifíquenlo para añadir la biblioteca `stdatomic.h` y cambiar la variable `global` a un tipo `atomic_int`. El programa debe quedar así:

```
//...
#include <stdatomic.h>

atomic_int global = 0;

void *increment(void *arg) {
    for (int i = 0; i < 1000; i++) {
        global++;
    }
    pthread_exit(NULL);
}
//...
```

Compilen y ejecuten el programa varias veces. Reporten ¿Qué observan? ¿Se solucionó el problema?

Sincronización de hilos con mutex

Desafortunadamente, no todas las operaciones pueden ser realizadas nativamente de forma atómica. Sin embargo, es posible usar mecanismos de sincronización para evitar que los hilos accedan a la misma variable al mismo tiempo. Uno de estos mecanismos es el *mutex* (mutual exclusion). Un mutex es un objeto que permite a un hilo bloquear el acceso a una sección crítica de código, de forma que solo un hilo pueda ejecutar esa sección a la vez.

> **IMPORTANTE**: Siempre que se use un mutex, se debe asegurar que se libera el mutex después de usarlo. De lo contrario, se puede causar un *deadlock* (bloqueo) en el programa.

A continuación se muestra un ejemplo de cómo usar un mutex para sincronizar el acceso a la variable `global`:

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

#define TOTAL_THREADS 16

int global = 0;
pthread_mutex_t mutex;

void *increment(void *arg) {
    for (int i = 0; i < 1000; i++) {
        // printf("Thread %ld\n", pthread_self());

        // Inicia la sección crítica
        pthread_mutex_lock(&mutex);

        // Realizamos las operaciones que NO pueden ser realizadas
        // de forma concurrente.
        global++;

        // Finaliza la sección crítica
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[TOTAL_THREADS];

    pthread_mutex_init(&mutex, NULL);

    printf("Global before: %d\n", global);
```

```
    for (int i = 0; i < TOTAL_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, increment, NULL) != 0) {
            perror("Error creating thread\n");
            return 1;
        }
    }

    for (int i = 0; i < TOTAL_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    printf("Global after: %d\n", global);

    return 0;
}
```

Compilen y ejecuten el programa varias veces. Reporten los resultados. Descomenten la línea `printf("Thread %ld\n", pthread_self());` y ejecuten el programa varias veces. ¿Qué observan?

NOTA: La función `pthread_self()` retorna el identificador del hilo que la llama.

4. Paso de argumentos a un hilo

Hasta ahora, hemos pasado `NULL` como argumento a los hilos. Sin embargo, es posible pasar argumentos a los hilos. Para ello, se debe crear una estructura que contenga los argumentos que se desean pasar al hilo. A continuación se muestra un ejemplo de cómo pasar argumentos a un hilo:

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

struct arguments {
    int a;
    int b;
};

void *sum(void *arg) {
    struct arguments *args = (struct arguments *) arg;
    int result = args->a + args->b;
    printf("Result: %d\n", result);
    pthread_exit(NULL);
}

int main(void) {
    pthread_t thread;
    struct arguments args = { 42, 23 };

    if (pthread_create(&thread, NULL, sum, &args) != 0) {
        perror("Error creating thread\n");
        return 1;
    }

    pthread_join(thread, NULL);

    return 0;
}
```

Compilen y ejecuten el programa anterior. Reporten los resultados.

4.2 Paso de argumentos a múltiples hilos

En el ejercicio anterior, pasamos argumentos a un solo hilo. Sin embargo, es posible pasar argumentos a múltiples hilos. Para ello, deberá crear diferentes estructuras con los argumentos que se desean pasar a cada hilo. Existen múltiples estrategias para hacerlo:

- Crear un arreglo de estructuras, donde cada estructura contiene los argumentos para un hilo.
  - Ventaja: Es fácil de implementar.
  - Desventaja: Sirve para una cantidad limitada de hilos.
- Crear una instancia de la estructura por cada hilo.
  - Ventaja: Se pueden pasar argumentos diferentes a cada hilo.
  - Desventaja: Debe liberar la memoria de cada estructura después de usarla.

Implementen una de las estrategias anteriores y modifiquen el programa anterior para que se creen 16 hilos, cada uno con argumentos diferentes. Compilen y ejecuten el programa varias veces. Reporten los resultados.

Corran su programa con `valgrind` para verificar que no hay fugas de memoria. Reporten qué problemas encontraron en su implementación y qué hicieron para solucionarlos.