Informe Laboratorio Pthreads

Renzo Augusto Cayllahue Ccora
Universidad Nacional de San Agustin de Arequipa

1. Introducción

En esta primera parte se ejecuta distintos programas para poder comparar los diferencias tiempos.

2. EJERCICIOS

2.1. Implementar y comparar las técnicas de sincronización Busy-waiting y Mutex, obtener una tabla similar a la Tabla 4.1 del libro.

Debido a que la arquitectura se basa en un sistema de memoria compartida se tiene que limitar el acceso a la sección critica ya que existe el peligro de que todos los hilos puedan manipular el mismo dato en memoria, por tal razón existen las barreras Busy wait y Mutex que controlan esto.

2.1.1. Calcular el valor de PI

Para poder comprobar los tiempos tomamos este algoritmo para calcular el valor de Pi, tanto con Busy-waiting como con mutex.

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right).$$

Figura 1. Computación del valor de **pi**

Al momento de analizar el problema, nos damos cuenta que encontramos variables dependientes al momento de hacer el cálculo de pi, encontrando una sección crítica la cual abordaremos utilizando dos métodos, busy-waiting y mutex.

Busy-waiting

Al momento de abordar nuestro problema primero debemos identificar la sección crítica, para luego tener en cuenta donde colocar el bucle busy-waiting.

En el primer caso (Busy-wait 1), la sección crítica se encuentra dentro del bucle principal, de la siguiente manera.

```
98   void* Thread_sum(void* rank) {
99      long my_rank = (long) rank;
100     double factor;
101     long long i;
102     long long my_n = n/thread_count;
103     long long my_first_i = my_n*my_rank;
104     long long my_last_i = my_first_i + my_n;
105
106     if (my_first_i % 2 == 0)
107        factor = 1.0;
108     else
109        factor = -1.0;
110
111     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
112        while (flag != my_rank);
113        sum += factor/(2*i+1);
114        flag = (flag+1) % thread_count;
115     }
116
117     return NULL;
118  }  /* Thread_sum */
119
```

En los resultados de la ejecución podemos observar que el tiempo serial es diferente al tiempo que toma calcular el valor de pi utilizando un solo thread, esto se debe a que el optimizador del compilador se encuentra activado al momento de la ejecución, lo cual causa una disminución considerable al momento de usar una sola hebra.

En el caso de Busy-wait 2 el desempeño es mejor, y esto se debe a que la sección crítica se sacó del bucle principal, de manera que hubo conflictos tanto en caché como en memoria principal; esto sin dejar de controlar el trabajo de los threads.

```
98   void* Thread_sum(void* rank) {
99      long my_rank = (long) rank;
100     double factor, my_sum = 0.0;
101     long long i;
102     long long my_n = n/thread_count;
103     long long my_first_i = my_n*my_rank;
104     long long my_last_i = my_first_i + my_n;
105
106     if (my_first_i % 2 == 0)
107        factor = 1.0;
108     else
109        factor = -1.0;
110
111     for (i = my_first_i; i < my_last_i; i++, factor = -factor)
112        my_sum += factor/(2*i+1);
113
114     while (flag != my_rank);
115     sum += my_sum;
116     flag = (flag+1) % thread_count;
117
118     return NULL;
119  }  /* Thread_sum */
120
```

Mutex: Protegemos la sección critica con el uso de mutex, de la siguiente manera.

```
98   void* Thread_sum(void* rank) {
99      long my_rank = (long) rank;
100     double factor, my_sum = 0.0;
101     long long i;
102     long long my_n = n/thread_count;
103     long long my_first_i = my_n*my_rank;
104     long long my_last_i = my_first_i + my_n;
105
106     if (my_first_i % 2 == 0)
107        factor = 1.0;
108     else
109        factor = -1.0;
110
111     for (i = my_first_i; i < my_last_i; i++, factor = -factor)
112        my_sum += factor/(2*i+1);
113
114     while (flag != my_rank);
115     sum += my_sum;
116     flag = (flag+1) % thread_count;
117
118     return NULL;
119  }  /* Thread_sum */
120
```

Ejecución

```
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.309761e-01 seconds
   Math library estimate of pi    = 3.141592653589793
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ./pth_pi_busy2 8 100000000
With n = 100000000 terms,
   Multi-threaded estimate of pi  = 3.141592643589880
   Elapsed time = 6.761909e-02 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.321279e-01 seconds
   Math library estimate of pi    = 3.141592653589793
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ./pth_pi_busy2 16 100000000
With n = 100000000 terms,
   Multi-threaded estimate of pi  = 3.141592643589896
   Elapsed time = 1.143210e-01 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.309759e-01 seconds
   Math library estimate of pi    = 3.141592653589793
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ./pth_pi_busy2 32 100000000
With n = 100000000 terms,
   Multi-threaded estimate of pi  = 3.141592643589664
   Elapsed time = 1.381111e-01 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.282829e-01 seconds
   Math library estimate of pi    = 3.141592653589793
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ^C
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ./pth_pi_busy2 64 100000000
With n = 100000000 terms,
   Multi-threaded estimate of pi  = 3.141592643589874
   Elapsed time = 5.885229e-01 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.272091e-01 seconds
   Math library estimate of pi    = 3.141592653589793
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ./pth_pi_busy2 64 100000000
With n = 100000000 terms,
   Multi-threaded estimate of pi  = 3.141592643589874
   Elapsed time = 3.749290e-01 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.292640e-01 seconds
   Math library estimate of pi    = 3.141592653589793
```

Tiempos de Ejecución y su comparación
Tomando un n = 10^8

Threads	Busy-wait	Mutex
1	0.25167	0.26872
2	0.14132	0.13507
4	0.09521	0.07702
8	0.06762	0.06743
16	0.11432	0.06688
32	0.1381	0.066704
64	0.58852	0.06439

El mutex excluye a todos los subthreads mientras ejecuta la seccion critica. Todos los threads que han llamado a pthread$_{m}utex_{l}ocksebloquearanensusllamadas, esperaranaqueelprimerthreadestehecho.Despuesdeque$

2.2. Basado en la sección 4.7, implementar un ejemplo de productor-consumidor. Explicar porque no se debe utilizar MUTEX.

EL problema del productor consumidor consiste en lo siguiente El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

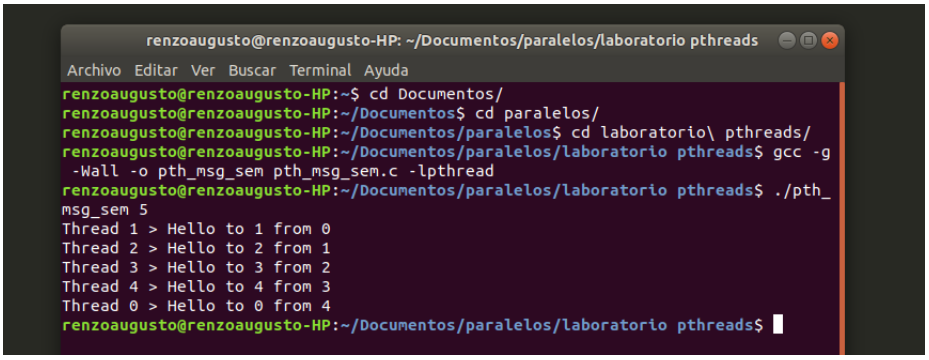
Una de las maneras de poder solucionar esto es usando mecanismos de comunicación de interprocesos, generalmente se usan semáforos. Una inadecuada implementación del problema puede terminar en un deadlock, donde ambos procesos queden en espera de ser despertados.

En nuestro caso se hiza un paso de mensajes con semaforos.

Una manera optima de resover el conflicto de sincronizacion es mediante semaforos

```c
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;
   sem_post(&semaphores[dest]);  /* "Unlock" the semaphore of dest */

   sem_wait(&semaphores[my_rank]);  /* Wait for our semaphore to be unlocked */
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

   return NULL;
}  /* Send_msg */

```

Ejecución con 5 hilos



USO DE MUTEX: No está del todo claro cómo los mutex pueden ser de ayuda aquí. Podríamos intentar llamar al desbloqueo de mutex de pthread para "notificar.[el] hilo con rango dest. Sin embargo, los mutexes se inicializan para ser desbloqueados, por lo que tendríamos que agregar una llamada antes de inicializar los mensajes [dest] para bloquear el mutex. Esto será un problema ya que no sabemos cuándo llegarán los hilos a las llamadas al bloqueo de mutex de pthread.
Para hacer esto un poco más claro, suponga que el hilo principal crea e inicializa una matriz de mutex, uno para cada hilo. Entonces, estamos intentando hacer algo como esto:

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

Ahora suponga que tenemos dos subprocesos, y el subproceso 0 se adelanta tanto al subproceso 1 que llega a la segunda llamada al bloqueo de mutex de pthread en la línea 7 antes de que el subproceso 1 llegue al primero en la línea 2. Luego, por supuesto, adquirirá el bloqueo y continúe con la declaración printf. Esto dará como resultado que el subproceso 0 elimine la referencia a un puntero nulo y se bloqueará.

POSIX también proporciona un medio algo diferente de controlar el acceso a secciones críticas: semáforos. Echemos un vistazo a ellos. Los semáforos se pueden considerar como un tipo especial de int sin signo, por lo que pueden tomar los valores 0, 1, 2,. . . . En la mayoría de los casos, solo

nos interesará usarlos cuando tomen los valores 0 y 1. Un semáforo que solo toma estos valores se llama semáforo binario. En términos muy generales, 0 corresponde a un mutex bloqueado y 1 corresponde a un mutex desbloqueado. Para usar un semáforo binario como mutex, lo inicializa en 1, es decir, está "desbloqueado". Antes de la sección crítica que desea proteger, realiza una llamada a la función sem wait. Un hilo que ejecuta sem wait se bloqueará si el semáforo es 0. Si el semáforo es distinto de cero, disminuirá el semáforo y continuará. Después de ejecutar el código en la sección crítica, un hilo llama a sem post, lo que incrementa el semáforo, y un hilo que espera en sem espera puede continuar.

El hilo principal puede inicializar todos los semáforos a 0, es decir, "bloqueado", y luego cualquier hilo puede ejecutar una publicación sem en cualquiera de los semáforos y, de manera similar, cualquier hilo puede ejecutar sem espera en cualquiera de los semáforos.

2.3. Implementar y explicar las diferentes formas de barreras en PThreads mostradas en el libro

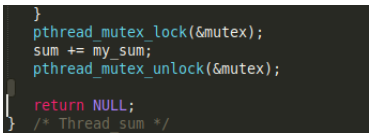
2.3.1. Barriers

Un barrier sincroniza los threads de tal manera que se asegura que todos se encuentran en un mismo punto de un programa. En otras palabras, ningún thread pasará la barrera hasta que los otros lo hayan alcanzado.

- **Barrier**: Solo es necesaria la inclusión de la librerıa pthread.h, es decir la librerıa Pthread ya implementa sus propias barreras del tipo de dato.
  Funciones:

  - pthread barrier init(b, NULL): Inicializa con la cantidad de hilos que se requieren sincronizar.
  - pthread barrier wait(b): Para esperar en una ba- rrera, una llamada de hilo.
  - pthread barrier destroy(b): Para destruir una ba- rrera.

- **Mutex y busy-waiting** :Utilizamos un contador compartido protegido por el mutex. Cuando el contador indica que cada hilo ha entrado en la sección critica, los hilos pueden abandonar la sección critica. Un mutex utilizamos cuando (hilo) desee ejecutar código que no deberıa ser ejecutado por ningún otro hilo al mismo tiempo. Mutex (abajo) ocurre en un hilo y mutex (arriba) debe pasar en el mismo hilo más adelante. Mutex: Mutex es para proteger el recurso compartido.



```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

- **Semáforos:** Use un semáforo cuando (hilo) quiera dormir hasta que otro hilo le indique que se despierte. El semáforo (abajo) ocurre en un hilo (productor) y el semáforo (arriba) (para el mismo semáforo) ocurre en otro hilo (consumidor). Semáforo: semáforo es enviar los hilos. Tenemos un contador que usamos para determinar cuántos hilos han entrado en la barrera. Usamos dos semáforos: count sem protege el contador, y barrier sem se usa para bloquear los hilos que han entrado en la barrera. El semáforo count sem se inicializa en 1 (desbloquead), por lo que el primer hilo que llegue a la barrera podrá continuar más allá de la llamada a sem wait.Los hilos posteriores se bloquearán hasta que puedan tener acceso exclusivo al contador. Funciones

  - sem init(sem, attr, val): Inicializa el semáforo sem al valor val con los atributos attr.
  - sem destroy (sem): Destruye el semáforo sem.
  - sem wait(sem): Si el valor del semáforo sem es positivo lo decrementa y retorna inmediatamente. En otro se bloquea hasta poder hacerlo
  - sem trywait(sem): Versión no bloqueante de sem wait(sem). En cualquier caso retorna inmediatamente.
  - sem post (sem): Incrementa el valor del semáforo sem. En caso de cambiar a un valor positivo des- bloquea a alguno de los llamadores bloqueados en sem wait(sem).

```
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier_sem;  /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
```

Semáforo Barrier

```
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ gcc -g -Wall -o pth_sem_bar pth_sem_bar.c -lpthread

renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ./pth_sem_bar 3
Elapsed time = 2.954960e-03 seconds
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$ ./pth_sem_bar 8
Elapsed time = 7.879019e-03 seconds
renzoaugusto@renzoaugusto-HP:~/Documentos/paralelos/laboratorio pthreads$
```

```
95  void *Thread_work(void* rank) {
96  #  ifdef DEBUG
97     long my_rank = (long) rank;
98  #  endif
99     int i, j;
100
101    for (i = 0; i < BARRIER_COUNT; i++) {
102       sem_wait(&count_sem);
103       if (counter == thread_count - 1) {
104          counter = 0;
105          sem_post(&count_sem);
106          for (j = 0; j < thread_count-1; j++)
107             sem_post(&barrier_sems[i]);
108       } else {
109          counter++;
110          sem_post(&count_sem);
111          sem_wait(&barrier_sems[i]);
112       }
113 #     ifdef DEBUG
114       if (my_rank == 0) {
115          printf("All threads completed barrier %d\n", i);
116          fflush(stdout);
117       }
118 #     endif
119    }
120
121    return NULL;
122 } /* Thread_work */
```

- **Variables de condición**: Una variable de condición es un objeto de datos que permite que un hilo suspenda la ejecución hasta que ocurra un determinado evento o condición. Cuando ocurre el evento o condición, otro hilo puede indicar al hilo que se (despierte) (wake up). Una variable de condición siempre está asociada con un mutex. Las variables de condición en Pthreads tienen el tipo pthread cond t:

  - Desbloquea uno de los threads bloqueados.

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

  - Desbloquea todos los threads bloqueados.

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

3. Enlace del código

https://github.com/rmzoccc/Compu-Paralela-y-distribuida