PTHREADS

Selene Barrios Cornejo

1 BARRERAS

Esta implementación desperdicia ciclos de CPU cuando los subprocesos estén en el bucle de espera ocupada y, si ejecutamos el programa con más subprocesos que núcleos, es posible encontrar que el rendimiento del programa se degrada seriamente.

```
void *Thread_work(void* rank) {
#  ifdef DEBUG
   long my_rank = (long) rank;
#  endif
   int i;

   for (i = 0; i < BARRIER_COUNT; i++) {
      pthread_mutex_lock(&barrier_mutex);
      barrier_thread_counts[i]++;
      pthread_mutex_unlock(&barrier_mutex
          );
      while (barrier_thread_counts[i] <
          thread_count);
#     ifdef DEBUG
      if (my_rank == 0) {
         printf("All threads entered
             barrier %d\n", i);
         fflush(stdout);
      }
#     endif
   }

   return NULL;
}  /* Thread_work */
```

1.1 Semáforos

Tenemos un contador que se usa para determinar cuántos subprocesos han entrado en la barrera. Hay 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 a 1 por lo que el primer hilo en alcanzar la barrera podrá continuar más allá de la llamada a *sem_wait*.

```
void *Thread_work(void* rank) {
#  ifdef DEBUG
   long my_rank = (long) rank;
#  endif
   int i, j;

   for (i = 0; i < BARRIER_COUNT; i++) {
      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_sems[i]);
      } else {
         counter++;
         sem_post(&count_sem);
         sem_wait(&barrier_sems[i]);
      }
#     ifdef DEBUG
      if (my_rank == 0) {
         printf("All threads completed
             barrier %d\n", i);
         fflush(stdout);
      }
#     endif
   }

   return NULL;
}  /* Thread_work *
```

Sin embargo la reutilización de *barrier_sem* dará como resultado una condición de carrera.

• *Barrios Cornejo Selene*
sbarrios@unsa.edu.pe,
Universidad Nacional de San Agustín.

1.2 Variables de condición

Una variable de condición es un objeto de datos que permite que un subproceso suspenda la ejecución hasta que se produzca un determinado evento o condición, siempre está asociada con un mutex. Sin embargo, algunos subprocesos pueden avanzar y cambiar la condición, y si cada subproceso está comprobando la condición, un subproceso que se despertó más tarde puede encontrar que la condición ya no se satisface y volver a dormir.

```
void *Thread_work(void* rank) {
#  ifdef DEBUG
   long my_rank = (long) rank;
#  endif
   int i;

   for (i = 0; i < BARRIER_COUNT; i++) {
      pthread_mutex_lock(&barrier_mutex);
      barrier_thread_count++;
      if (barrier_thread_count ==
          thread_count) {
         barrier_thread_count = 0;
#        ifdef DEBUG
         printf("Thread %ld > Signalling
             other threads in barrier %d\n
             ",
                my_rank, i);
         fflush(stdout);
#        endif
         pthread_cond_broadcast(&
             ok_to_proceed);
      } else {
         while (pthread_cond_wait(&
             ok_to_proceed,
                   &barrier_mutex) != 0);
#        ifdef DEBUG
         printf("Thread %ld > Awakened in
              barrier %d\n", my_rank, i);
         fflush(stdout);
#        endif
      }
      pthread_mutex_unlock(&barrier_mutex
          );
#     ifdef DEBUG
      if (my_rank == 0) {
         printf("All threads completed
             barrier %d\n", i);
         fflush(stdout);
      }
#     endif
   }
   return NULL;
}  /* Thread_work */
```

2 LISTA ENLAZADA MULTITHREADING

Varios subprocesos pueden leer simultáneamente una ubicación de memoria sin conflictos, es decir que varios subprocesos pueden ejecutar Member simultáneamente.

```
int  Member(int value) {
   struct list_node_s *temp, *old_temp;

   pthread_mutex_lock(&head_mutex);
   temp = head;
   if (temp != NULL) pthread_mutex_lock
       (&(temp->mutex));
   pthread_mutex_unlock(&head_mutex);
   while (temp != NULL && temp->data <
       value) {
      if (temp->next != NULL)
         pthread_mutex_lock(&(temp->next
             ->mutex));
      old_temp = temp;
      temp = temp->next;
      pthread_mutex_unlock(&(old_temp->
          mutex));
   }

   if (temp == NULL || temp->data > value
       ) {
#     ifdef DEBUG
      printf("%d is not in the list\n",
          value);
#     endif
      if (temp != NULL)
         pthread_mutex_unlock(&(temp->
             mutex));
      return 0;
   } else { /* temp != NULL && temp->data
        <= value */
#     ifdef DEBUG
      printf("%d is in the list\n", value
          );
#     endif
      pthread_mutex_unlock(&(temp->mutex)
          );
      return 1;
   }
}  /* Member */
```

Sin embargo, podemos tener problemas si intentamos ejecutar simultáneamente otra operación mientras estamos ejecutando una inserción o una eliminación.

2.0.1 *Implementar bloqueos de lectura y escritura*

El mutex protege la estructura de datos de bloqueo de lectura-escritura: siempre que un hilo llama a uno de las funciones , primero bloquea el mutex, y siempre que un hilo completa una de estas llamadas, desbloquea el mutex.

```
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   int i, val;
   double which_op;
   unsigned seed = my_rank + 1;
   int my_member=0, my_insert=0,
      my_delete=0;
   int ops_per_thread = total_ops/
      thread_count;

   for (i = 0; i < ops_per_thread; i++) {
      which_op = my_drand(&seed);
      val = my_rand(&seed) % MAX_KEY;
      if (which_op < search_percent) {
         pthread_mutex_lock(&mutex);
         Member(val);
         pthread_mutex_unlock(&mutex);
         my_member++;
      } else if (which_op <
          search_percent + insert_percent)
           {
         pthread_mutex_lock(&mutex);
         Insert(val);
         pthread_mutex_unlock(&mutex);
         my_insert++;
      } else { /* delete */
         pthread_mutex_lock(&mutex);
         Delete(val);
         pthread_mutex_unlock(&mutex);
         my_delete++;
      }
   }  /* for */

   pthread_mutex_lock(&count_mutex);
   member_total += my_member;
   insert_total += my_insert;
   delete_total += my_delete;
   pthread_mutex_unlock(&count_mutex);

   return NULL;
}  /* Thread_work */
```

thread_safety. Serializar el acceso a las líneas de entrada usando semáforos. Luego, después de que un hilo haya leído una sola línea de entrada, puede tokenizar la línea. Una forma de hacer esto es usar la función strtok

```
void *Tokenize(void* rank) {
   long my_rank = (long) rank;
   int count;
   int next = (my_rank + 1) %
      thread_count;
   char *fg_rv;
   char my_line[MAX];
   char *my_string;

   /* Force sequential reading of the
      input */
   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin);
   sem_post(&sems[next]);
   while (fg_rv != NULL) {
      printf("Thread %ld > my line = %s",
          my_rank, my_line);

      count = 0;
      my_string = strtok(my_line, " \t\n"
         );
      while ( my_string != NULL ) {
         count++;
         printf("Thread %ld > string %d =
             %s\n", my_rank, count,
            my_string);
         my_string = strtok(NULL, " \t\n"
            );
      }
      if (my_line != NULL) printf("Thread
          %ld > After tokenizing, my_line
          = %s\n",
        my_rank, my_line);

      sem_wait(&sems[my_rank]);
      fg_rv = fgets(my_line, MAX, stdin);
      sem_post(&sems[next]);
   }

   return NULL;
}  /* Tokenize */
```

3 THREAD SAFETY

Otro problema potencial que ocurre en la programación de memoria compartida:

4 RESULTADOS

Threads	Busy-Wait	Mutex	Semáforo
1	0	0.000290	0.000317
2	0	0.003019	0.002449
4	0.059340	0.011236	0.007140
8	1.366471	0.013808	0.020568
16	2.411756	0.061681	0.044779
32	3.851854	0.143638	0.054790
64	8.600837	0.166680	0.162702

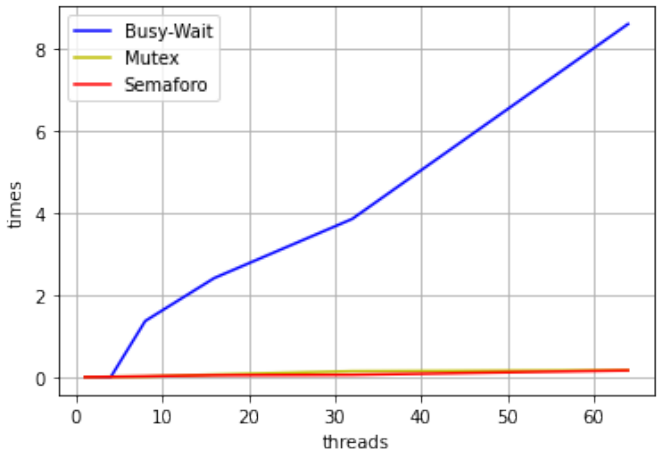

Figure 1. Tiempos de ejecución para 1, 2, 4, 8, 16, 32 y 64 subprocesos para busy waiting, mutex y semáforo

5 REPOSITORIO

El código se encuentra disponible en el siguiente repositorio de Github.