Introduction to parallel computing

Shared Memory Programming with Pthreads (3)

Zhiao Shi (modifications by Will French)
Advanced Computing Center for Education & Research
Vanderbilt University

Last time

- Mutex lock

```
int pthread_mutex_init (pthread_mutex_t *mutex_lock,
    const pthread_mutexattr_t *lock_attr);

int pthread_mutex_lock (pthread_mutex_t *mutex_lock);

int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);

int pthread_mutex_destroy(pthread_mutex_t *mutex_lock);
```

BARRIERS AND CONDITION VARIABLES

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.

- No thread can cross the barrier until all the threads have reached it.

Using barriers to time the slowest thread

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;


my_elapsed = my_finish − my_start;

elapsed = Maximum of my_elapsed values;
```

Using barriers for debugging

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

Busy-waiting and a Mutex

```
/* 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);
   . . .
}
```

However, the Pthread library provides its own barrier functions…

Creating and Initializing a Barrier

- To initialize a barrier, use code similar to this (which sets the number of threads to 4):

```
pthread_barrier_t b; // declare with global scope
pthread_barrier_init(&b,NULL,4);
```

- The second argument specifies an attribute object for finer control; using NULL yields the default attributes.

- To wait at a barrier, a thread call:

```
pthread_barrier_wait(&b);
```

- To destroy a barrier:

```
pthread_barrier_destroy(&b);
```

Condition Variables

- Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

- With locks, the global variable would need to be examined at frequent intervals ("polled") within a critical section.

    - Very time-consuming and unproductive.

- Can be overcome by introducing so-called *condition variables*.

Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to “wake up.”

Condition Variables for Synchronization

- A condition variable is associated with the predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.

- A condition variable always has a mutex associated with it. A thread locks this mutex and tests the predicate defined on the shared variable.

Condition Variables for Synchronization

- If the predicate is not true, the thread waits on the condition variable associated with the predicate using the function `pthread_cond_wait`.
  - This also releases the lock on the mutex so that others can change the condition variable

- At a later time when another thread makes the predicate true, that thread calls `pthread_cond_signal` to unblock the waiting thread.
  - The signaled (waiting) thread now also has the lock on the mutex

Condition Variables for Synchronization

- Pthreads provides the following functions for condition variables:

```
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);

int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Condition variables: wait

- **`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`**

  - Blocks the calling thread, waiting on cond.
  - Unlock the mutex
  - Re-acquires the mutex when unblocked.

Condition variables: signal

- **`pthread_cond_signal(pthread_cond_t *cond)`**
  - Unblocks one thread waiting on cond.
  - The scheduler determines which thread to unblock.
  - If no thread waiting, then signal accomplishes nothing.

Condition variables: broadcast

- **`pthread_cond_broadcast(pthread_cond_t *cond)`**
  - Unblocks **all** threads waiting on cond.

Implementing a barrier with condition variables

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

Producer consumer program without condition variables

```c
/* Globals */
int data_avail = 0;
pthread_mutex_t data_mutex;
pthread_mutex_init(&data_mutex, NULL);

void *producer(void *)
{
   pthread_mutex_lock(&data_mutex);

   /* Produce data
      insert data into queue;
  */
   data_avail=1;

   pthread_mutex_unlock(&data_mutex);

}
```

```
void *consumer(void *)
{
   while( !data_avail );
            /* do nothing - keep looping!!*/

   pthread_mutex_lock(&data_mutex);

   // Extract data from queue;
   if (queue is empty)
     data_avail = 0;

   pthread_mutex_unlock(&data_mutex);

   consume_data();
}
```

Producer consumer program with condition variables

```
int data_avail = 0;
pthread_mutex_t data_mutex;
pthread_cond_t data_cond;
pthread_mutex_init(&data_mutex, NULL);
pthread_cond_init(&data_cond, NULL);

void *producer(void *) {
   pthread_mutex_lock(&data_mutex);

   //Produce data
   //Insert data into queue;
   data_avail = 1;

   pthread_cond_signal(&data_cond);
   pthread_mutex_unlock(&data_mutex);

}
```

```
void *consumer(void *)
{
   pthread_mutex_lock(&data_mutex);
   while( !data_avail ) {
         /* sleep on condition variable*/
         pthread_cond_wait(&data_cond, &data_mutex);
    }
   /* woken up */
   /* Extract data from queue; */
   if (queue_is_empty())
     data_avail = 0;

   pthread_mutex_unlock(&data_mutex);

   consume_data();
}
```

Producer-Consumer Using Condition Variables

- Why do the previous two slides use `while`-loops around the `pthread_cond_wait()`?
  - It seems that if we received the signal, then the while-condition must be false.

- If we had multiple producers or consumers, one of the other threads may have received the lock first and since invalidated the condition.

- It is also possible that the thread was woken up for other reasons (*e.g.,* an OS signal).