Barriers and condition variables

A **barrier**: synchronizing the threads by making sure that they all are at the same point in a program

Barriers have numerous applications

- Timing parts of a multithreaded system.
  - we'd like for all the threads to start the timed code at the same instant, and then report the time taken by the last thread to finish, i.e., the "slowest" thread. So we'd like to do something like this:

```
/* 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 this approach, we're sure that all of the threads will record my_start at approximately the same time
- Debugging
  - We can, of course, have each thread print a message indicating which point it's reached in the program, but it doesn't take long for the volume of the output to become overwhelming. Barriers provide an alternative:

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

Many implementations of Pthreads don't provide barriers, so if our code is to be portable, we need to develop our own implementation

- **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 busy-wait loop.

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

  - Of course, this implementation will have the same problems that our other busywait codes had: we'll waste CPU cycles when threads are in the busy-wait loop, and, if we run the program with more threads than cores, we may find that the performance of the program seriously degrades
  - It also suffers from errors that are prone to happen with the shared variable counter
- **Semaphores**

```
/* Shared variables */
int counter;        /* Initialize to 0 */
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);
   }
   . . .
}
```

Note:
- The count_sem semaphore is initialized to 1 (that is, "unlocked"), so the first thread to reach the barrier will be able to proceed past the call to sem_wait.
- Subsequent threads, however, will block until they can have exclusive access to the counter.
- Essentially coun_sem protects the counter itself and barrier_sem protects the threads from advancing from this portion of the code until everything is done

- As with the busy-wait barrier, we have a counter that we use to determine how many threads have entered the barrier. We use two semaphores: count_sem protects the counter, and barrier_sem is used to block threads that have entered the barrier.
- What the code is doing:
  "When a thread has exclusive access to the counter, it checks to see if counter < thread_count-1. If it is, the thread increments counter, relinquishes the lock (sem_post(&count_sem)), and blocks in sem_wait(&barrier_sem). On the other hand, if counter == thread_count-1, the thread is the last to enter the barrier, so it can reset counter to zero and "unlock" count_sem by calling sem_post(&count_sem). Now, it wants to notify all the other threads that they can proceed, so it executes sem_post(&barrier_sem) for each of the thread_count-1 threads that are blocked in sem_wait(&barrier_sem)."

- It should be clear that this implementation of a barrier is superior to the busy-wait barrier, since the threads don't need to consume CPU cycles when they're blocked in sem_wait.
- Can we reuse the data structures from the first barrier if we want to execute a second barrier?
  - The counter can be reused, since we were careful to reset it before releasing any of the threads from the barrier.
  - Also, count_sem can be reused, since it is reset to 1 before any threads can leave the barrier.
  - Barrier_sem cannot be reused caused by race condition

- **Condition variables**: is a data object that allows a thread to suspend execution until a certain event or condition occurs.
  - A somewhat better approach to creating a barrier in Pthreads is provided by condition variables
  - When the event or condition occurs another thread can signal the thread to "wake up."
  - A condition variable is always associated with a mutex.
  - Pseudocode:

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

  - Condition variables in Pthreads have type pthread_cond_t.
  - Functions
    - Will unblock one of the blocked threads:

```
int pthread_cond_signal(
    pthread_cond_t*  cond_var_p  /* in/out */);
```

- will unblock all of the blocked threads.

```
int pthread_cond_broadcast(
    pthread_cond_t* cond_var_p  /* in/out */);
```

  - This is one advantage of condition variables; recall that we needed a for loop calling sem_post to achieve similar functionality with semaphores.
- Does three things: mutex unlock, wait, mutex lock

```
int pthread_cond_wait(
      pthread_cond_t*    cond_var_p   /* in/out */,
      pthread_mutex_t*   mutex_p      /* in/out */);
```

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

- Example:

The following code implements a barrier with a condition variable:

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

- Things to look out for:

Note that it is possible that events other than the call to `pthread_cond_broadcast` can cause a suspended thread to unblock (see, for example, Butenhof [7], page 80). This is called a *spurious wake-up*. Hence, the call to `pthread_cond_wait` should usually be placed in a **while** loop. If the thread is unblocked by some event other than a call to `pthread_cond_signal` or `pthread_cond_broadcast`, then the return value of `pthread_cond_wait` will be nonzero, and the unblocked thread will call `pthread_cond_wait` again.

  - If a single thread is being awakened, it's also a good idea to check that the condition has, in fact, been satisfied before proceeding. In our example, if a single thread were being released from the barrier with a call to pthread_cond_signal, then that thread should verify that counter == 0 before proceeding.
  - Note that in order for our barrier to function correctly, it's essential that the call to pthread_cond_wait unlock the mutex.
  - Also note that the semantics of mutexes require that the mutex be relocked before we return from the call to pthread_cond_wait.

Like mutexes and semaphores, condition variables should be initialized and destroyed. In this case, the functions are

```
int pthread_cond_init(
      pthread_cond_t*              cond_p       /* out */,
      const pthread_condattr_t*    cond_attr_p  /* in  */);

int pthread_cond_destroy(
   pthread_cond_t*  cond_p  /* in/out */);
```

- We won't be using the second argument to pthread_cond_init—as with mutexes, the default the attributes are fine for our purposes—so we'll call it with second argument set to NULL
- `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
  - Same shit as pthread_cond_init

Condition variables are often quite useful whenever a thread needs to wait for something. When protected application state cannot be represented by an unsigned integer counter, condition variables may be preferable to semaphores.