Synchronization, Part 6: Implementing a barrier

Vadiml1024 edited this page on Feb 22 · 7 revisions

How do I wait for N threads to reach a certain point before continuing onto the next step?

Suppose we wanted to perform a large multi-threaded calculation that has two stages but we dont want to advance to the second stage until the first stage is completed.

```
double data[256][8192]

1 Threads do first calculation (use and change values in data)

2 Barrier! Wait for all threads to finish first calculation before continuing

3 Threads do second calculation (use and change values in data)
```

The thread function has four main parts-

```
void *calc(void *arg) {
  /* Do my part of the first calculation */
  /* Am I the last thread to finish? If so wake up all the other threads! */
  /* Otherwise wait until the other threads has finished part one */
  /* Do my part of the second calculation */
}
```

Our main thread will create the 16 threads and we will divide each calculation into 16 separate pieces. Each thread will be given a unique value (0,1,2,..15), so it can work on its own block. Since a (void) *type can hold small integers, we will pass the value of i by casting it to a void*.

```
#define N (16)
double data[256][8192] ;
int main() {
    pthread_t ids[N];
    for(int i = 0; i < N; i++)
        pthread_create(&ids[i], NULL, calc, (void *) i);
```

Note, we will never dereference this pointer value as an actual memory location - we will just cast it straight back to an integer:

```
void *calc(void *ptr) {
// Thread 0 will work on rows 0..15, thread 1 on rows 16..31
  int x, y, start = N * (int) ptr;
  int end = start + N;
  for(x = start; x < end; x++) for (y = 0; y < 8192; y++) { /* do calc #1 */ }
```

After calculation 1 completes we need to wait for the slower threads (unless we are the last thread!). So keep track of the number of threads that have arrived at our barrier aka 'checkpoint':

```
// Global:
int remain = N;


// After calc #1 code:
remain--; // We finished
if (remain ==0) {/*I'm last!  -  Time for everyone to wake up! */ }
else {
  while (remain != 0) { /* spin spin spin*/ }
}
```

However the above code has a race condition (two threads might try to decrement `remain`) and the loop is a busy loop. We can do better! Let's use a condition variable and then we will use a broadcast/signal functions to wake up the sleeping threads.

A reminder, that a condition variable is similar to a house! Threads go there to sleep (`pthread_cond_wait`). You can choose to wake up one thread (`pthread_cond_signal`) or all of them (`pthread_cond_broadcast`). If there are no threads waiting inside these calls have no effect.

A condition variable version is usually very similar to a busy loop incorrect solution. We will need to add a mutex and condition global variables and to initialize them in `main` ...

```
//global variables
pthread_mutex_t m;
pthread_cond_t cv;

main() {
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&cv, NULL);
```

We will use the mutex to ensure that only one thread modifies `remain` at a time. The last arriving thread needs to wake up all sleeping threads - so we will need `pthread_cond_broadcast(&cv)`

```
pthread_mutex_lock(&m);
remain--;
if (remain ==0) { pthread_cond_broadcast(&cv); }
else {
  while(remain != 0) { pthread_cond_wait(&cv, &m) }
}
pthread_mutex_unlock(&m);
```

When a thread enters `pthread_cond_wait` it releases the mutex and sleeps. At some point in the future it will be awoken. Once we bring a thread back from its sleep, before returning it must wait until it can lock the mutex. Notice that even if a sleeping thread wakes up early, it will check the while loop condition and re-enter wait if necessary.