Brian Quinn

Multicore Programming

Professor Mitchell

1 March 2017

Homework 2

1. The number of threads technically available to run depends on the number of cores, threads per core and cores per socket. Given your total possible threads based on the computing power, it is not necessarily a good idea to use all of them because all parallel threads are delegated space on the stack. If two many parallel threads are run, each gets a smaller delegated part of stack memory and could result in corrupted memory on the threads. Therefore, either number of cores or memory space will ultimately limit the number of available cores to run.

2. Peterson's algorithm works because each thread will set a corresponding flag to indicate it is active, one thread will be randomly chosen as the victim, and a conditional AND statement will send that victim into a busy wait until the thread finishes. The flags set are non-conflicting and assert that they are indeed seeking to use the lock (iteratively numbered thread ids are assumed), however, the victim variable is a global that will be conflicting among the two threads. Ultimately the variable can only hold one variable and thus, the later thread to set it will overwrite the earlier thread's victim status. Once the non-victim finishes it will set it's corresponding flag to false allowing the other thread to jump out of the busy wait.

Peterson’s algorithm needs to be modified for more than two threads because there is no condition on determining the next thread to run.

3. The reason for multiple levels of cache memory is that despite increasing latency among the cache, retrieving information from main memory is very costly and very slow. The latency examples used in class were that if L1 cache memory access was proportionately altered to take a single minute, the corresponding L2 cache memory would take 30 minutes, with the main memory ultimately taking 6 months. Cache memory is often a trade off as well for how often recent data will be used, larger latency times, and the necessary hardware. As such, cache memory usually only goes three levels deep, however, certain circumstances may provide more or less.

4.
```
pthread_mutex_t s_mutex;
static int sum_stat_a = 0;
static int sum_stat_b = 0;
int aggreagateStats(int stat_a, int stat_b) {
   pthread_mutex_lock(&s_mutex);
   sum_stat_a += stat_a;
   sum_stat_b += stat_b;
   int retval = sum_stat_a + sum_stat_b;
   pthread_mutex_unlock(stats_mutex);
   return retval;
}
void init(void) {
   pthread_mutex_init(&s_mutex, NULL);
}
```

5.
```
pthread_mutex_t a_mutex;
pthread_mutex_t b_mutex;
static int sum_stat_a = 0;
static int sum_stat_b = 0;
```

```
int aggreagateStats(int stat_a, int stat_b) {
   int retval = 0;
   pthread_mutex_lock(&a_mutex);
   sum_stat_a += stat_a;
   retval += sum_stat_a;
   pthread_mutex_unlock(a_mutex);
   pthread_mutex_lock(&b_mutex);
   sum_stat_b += stat_b;
   retval += sum_stat_b;
   pthread_mutex_unlock(b_mutex);
   return retval;
}
void init(void) {
   pthread_mutex_init(&a_mutex, NULL);
   pthread_mutex_init(&b_mutex, NULL);
}
```

6. When a thread is created resources are delegated to the thread and typically are not released until the thread terminated via an exit call. However, if the thread creates another thread via a fork call, it will wait for the other thread to terminate before it terminates itself. The join() function gets the return value of the internally created thread and subsequently terminates the outer thread. However, if there is no necessity to retrieve the return value of the internal thread, either the inner or outer thread can call detach() to detach the two threads to independent functions. Therefore, the original thread will exit and restore the resources demanded regardless of the state of the internally created thread.