James Maciej Krywin
Multicore Programming

Homework 2

1. Warm up: Running on some specific machine *X* with set hardware and software, how many threads should your program spawn, given that it has enough parallel tasks to run to occupy many threads? Think about how the hardware (eg, memory and number of cores) and software (ie, programs already occupying some resources) affect the number of threads you should spawn. Don't overthink this one!

In an ideal environment where there would be no interference with other processes, the program should spawn the exact number of threads as there is cores on the processor. This approach ensures that each core is fully and independently utilized. Another factor to keep in mind is the I/O wait time. Thus, if the processes depend on a lot of writes and reads to disk, the program should spawn more threads. This will give the processor additional work while it waits for the disk to complete the slow write/read requests for the previous thread.

2. Explain (in your own words!) how Peterson's Algorithm works. Why does it need to be re-written for more than 2 threads?

This algorithm uses a self defined binary mutex that allows the threads to know whether they can access the critical zone or not. This algorithm uses a 'bool' and an 'int' to keep track of the thread, or the 'victim', waiting for the unlock. Therefore, the 'victim' needs to wait until the bool flag for the other thread is set to false before it can proceed into the critical zone.

Peterson's algorithm was originally written only for 2 threads by a system of binary assumptions. Thus, the addition of any additional threads will not work because the binary assumptions break down i.e. (if one thread is in the critical zone, then other is not and vice versa → this cannot apply for 3).

3. If multiple on-die caches (eg, L2 and L3 cache) are higher-latency than an L1 cache, why do we have them anyway? (Hint: how big and slow is main memory? How big and slow is the L1 cache?)

Cache was implemented in order to reduce I/O time for a processor. L1 cache is relatively small and fast compared to main memory. These qualities allow the processor to save time by storing frequently used values in it. Last level caches are shared by the cores because they yield better results with caching. Despite them being larger and slower than the L1 cache, they are still significantly faster than retrieving values from the main memory.

4. Consider the following code:

```
1     static int sum_stat_a = 0;
2     static int sum_stat_b = 0;
3     int aggregateStats(int stat_a, int stat_b) {
4           sum_stat_a += stat_a;
5           sum_stat_b += stat_b;
6           return sum_stat_a + sum_stat_b;
7     }
8     void init(void) { }
```

Use a single pthread mutex to make this function thread-safe. Add global variables and content to the `init()` function as necessary.

```
static int sum_stat_a = 0;
static int sum_stat_b = 0;
pthread_mutex_t mut = PTHREAD_MUTEX_INIT(&mut_a, NULL);

int aggregateStats(int stat_a, int stat_b) {
        pthread_mutex_lock(&mut);
        sum_stat_a += stat_a;
        sum_stat_b += stat_b;
        int temp_a = sum_stat_a;
        int temp_b = sum_stat_b;
        pthread_mutex_unlock(&mut);
        return temp_a + temp_b;
}

void init(void);
```

5. Let's make this more parallelizable. We always want to reduce critical sections as much as possible to minimize the time threads need to wait for a resource protected by a lock. Modify the original code from question 3 to make it thread-safe, but use two mutices this time, one for `sum_stat_a` and one for `sum_stat_b`.

```
static int sum_stat_a = 0;
static int sum_stat_b = 0;
pthread_mutex_t mut_a; = PTHREAD_MUTEX_INIT(&mut_a, NULL);
pthread_mutex_t mut_b; = PTHREAD_MUTEX_INIT(&mut_b, NULL);


int aggregateStatA(int stat_a) {
        pthread_mutex_lock(&mut_a);
        sum_stat_a += stat_a;
        int temp_a = stat_a;
        pthread_mutex_unlock(&mut_a);
        return temp_a;
}

int aggregateStatB(int stat_b) {
        pthread_mutex_lock(&mut_b);
        sum_stat_b += stat_b;
        int temp_b = stat_b
        pthread_mutex_unlock(&mut_b);
        return temp_b;
}


int aggregateStats(int stat_a, int stat_b) {
        return aggregateStatA + aggregateStatB;
}

void init(void);
```

6. What's the difference between `join()`ing and `detach()`ing a pthread? When would we want to use each technique?

Join() and detach() are pthread methods that tell the program how to behave with its spawned threads. When Join() is called, it will block/wait until the thread finishes, it then allows the C++ thread object to be destroyed. When Detach() is called the method separats the thread from the C++ thread object. Thus, the join() method can no longer be called to ensure that the thread has completed and another solution will be needed to verify the threads completion. One example of this is in my Lab1 where I had the TS-Queue listening for the threads to enqueue their values. It kept track of the number of values inserted until it reached the number of spawned threads.