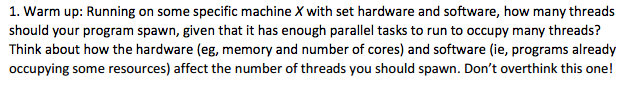
James Maciej Krywin

Multicore Programming

Homework 2



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.

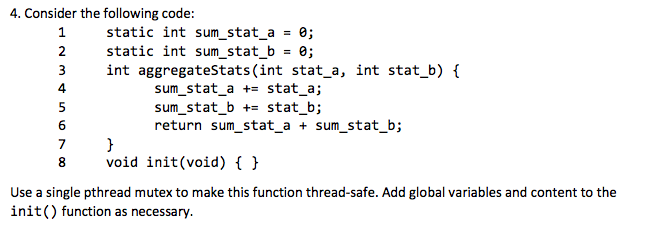
Macintosh HD:Users:james:Desktop:Screen Shot 2017-03-01 at 2.14.39 PM.png

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).

Macintosh HD:Users:james:Desktop:Screen Shot 2017-03-01 at 2.14.44 PM.png

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.



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;

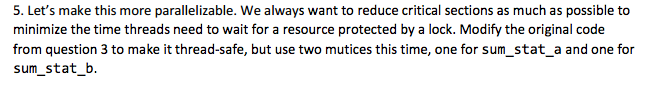
int temp\_b = sum\_stat\_b;

**pthread\_mutex\_unlock(&mut);**

return temp\_a + temp\_b;

}

void init(void);



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 stat\_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);

Macintosh HD:Users:james:Desktop:Screen Shot 2017-03-01 at 2.15.12 PM.png

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.