CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Mutlithreading and **ThreadPool**s

- In the hypothetical scenario where the myth cluster was 16,000 machines instead of just 16, our final **myth-buster** implementation would have created 16,000 threads over the course of its execution.
- The decision to batch create 8, 16, or 32 threads—any small multiple of the number of cores— and wait for them to finish before creating another batch is all the more crucial. We certainly shouldn't create 16,000 threads at once, even if the OS allows it.
  - Our most recent version of our myth-buster used semaphores to limit the number of threads to be at most 8 at any one time.
  - It doesn't, however, reduce the number of threads ultimately created. 16,000 threads would come and go, even if they come and go in waves.
  - The cost of setting up a new thread is actually quite large. Instead of allocating a new thread each time, we might try using a **fixed** number of threads—a **pool** of constantly recycled threads—to maximize parallelism without overwhelming the OS. Each thread in the pool can be used to execute many things in sequence, knowing that many other threads in the pool are doing the same.

Mutlithreading and **ThreadPools**

- One idea is to rely on a multithreading abstraction that's so common that many other programming languages just provide it. That abstraction is the **ThreadPool** class, which for us exports the following **public** interface:

```
class ThreadPool {
public:
   ThreadPool(size_t numThreads);
   void schedule(const std::function<void(void)>& thunk);
   void wait();
   ~ThreadPool();
};
```

- Here is a client program of our **ThreadPool** to illustrate what happens:

```
int main(int argc, char *argv[]) {
  ThreadPool pool(4);
  for (size_t id = 0; id < 10; id++) {
    pool.schedule([id] {
      cout << oslock << "Thread (ID: " << id << ") has started." << endl << osunlock;
      size_t sleepTime = (id % 3) * 10;
      sleep_for(sleepTime);
      cout << oslock << "Thread (ID: " << id << ") has finished." << endl << osunlock;
    });
  }
  pool.wait();
  cout << "All done!" << endl;
  return 0;
}
```

Mutlithreading and **ThreadPools**

- The output of the test program from the previous slide certainly looks to be using the four threads in the pool to ultimately execute the ten schedule functions.
- The win here is that we only allocate resources for four threads, which collaborate to execute all scheduled functions in a FIFO manner until the thread pool is empty.
- The thread routines are constrained to be zero argument functions with no return value. Such function are often called **thunks**.
- We're using the **ThreadPool** here, and you'll be implementing it for your next assignment.

```
myth64~$ ./tptest
Thread (ID: 3) has started.
Thread (ID: 2) has started.
Thread (ID: 1) has started.
Thread (ID: 0) has started.
Thread (ID: 3) has finished.
Thread (ID: 4) has started.
Thread (ID: 0) has finished.
Thread (ID: 5) has started.
Thread (ID: 1) has finished.
Thread (ID: 6) has started.
Thread (ID: 4) has finished.
Thread (ID: 7) has started.
Thread (ID: 6) has finished.
Thread (ID: 8) has started.
Thread (ID: 2) has finished.
Thread (ID: 9) has started.
Thread (ID: 9) has finished.
Thread (ID: 5) has finished.
Thread (ID: 7) has finished.
Thread (ID: 8) has finished.
All done!
myth64~$
```

Mutlithreading and **ThreadPools**

- Here is a new and improved myth-buster-pooled that uses a ThreadPool of size 8—that's the number of CPUs—to poll all 16 myths at once.

```
static void countCS110Processes(int num, const unordered_set<string>& sunetIDs,
                                map<int, int>& processCountMap, mutex& processCountMapLock) {
  int numProcesses = getNumProcesses(num, sunetIDs);
    if (numProcesses >= 0) {
        processCountMapLock.lock();
        processCountMap[num] = numProcesses;
        processCountMapLock.unlock();
        cout << "myth" << num << " has this many CS110-student processes: "
             << numProcesses << endl;
    }
}

static const int kMinMythMachine = 51;
static const int kMaxMythMachine = 66;
static const int kMaxNumThreads = 8;
static void compileCS110ProcessCountMap(const unordered_set<string> sunetIDs,
                                        map<int, int>& processCountMap) {
  ThreadPool pool(kMaxNumThreads);
  mutex processCountMapLock;
  for (int num = kMinMythMachine; num <= kMaxMythMachine; num++) {
    pool.schedule([num, &sunetIDs, &processCountMap, &processCountMapLock]() {
      countCS110Processes(num, sunetIDs, processCountMap, processCountMapLock);
    });
  }
  pool.wait();
}
```