Noah Legault, Andrew Donate, Devin St John

Data Structures 4415 Section 1

Mini Project 1 Part C Report

The task for Part C is to create a Scheduler by using a “Priority Queue” data structure. The way it works is that 2 inputs “n” and “m” are typed in, with “n” being the amount of threads available and “m” being the amount of jobs that must be processed. Each thread can only process 1 job at a time, and the amount of time it takes to process the job is given by the number value of the job. For example, the input:

2 5
1 2 3 4 5

Gives us 2 threads with 5 jobs, being 1, 2, 3, 4 and 5. The output shows us the ID of the thread and the time it has picked up a new job. The output for our example would be:

0 0
1 0
0 1
1 2
0 4

Once all the instructions were understood as well as the basic model of the schedular provided by the template, all that needed to be done was to implement a priority queue into the program. A priority queue is much like a normal queue that you can push and pop values, except instead of a first-in first-out, the highest value automatically rises to the front of the queue. For example, inserting the sequence (8, 1, 10, 2) into a priority queue gives us (10, 8, 2, 1).

At first, there is an issue when using this as a priority scheduler. When determining which thread should receive the next job, we need to figure out which thread has the least amount of jobs loaded. A simple modification to the declaration of a priority queue changes it to prioritize the smallest element in the list, as shown below.

```
priority_queue<
Worker,
vector<Worker>,
std::greater<Worker>
```

Next, a custom object had to be created in order to keep track of which thread has which amount of jobs loaded. This is where the “Worker” class comes in, a class with 2 members “I” and “next_free_time” as well as an operator overloading function for less than.

```
bool operator>(const Worker &w2) const {
  // if multiple threads are open then the one with the lower ID will take it
  if (this->next_free_time == w2.next_free_time) {
    return this->i > w2.i;
  }
  return (this->next_free_time > w2.next_free_time);
}
```

Now our priority queue is set up and our queue also knows how to prioritize our threads. At this point, the thread that has the lowest amount of jobs scheduled is pushed to the top to make job processing more efficient.

Finally, we must implement a way to use this priority queue and to add jobs to it. The code below demonstrates this.

```
while (!pq.empty()) {

  // get highest priority from the queue
  Worker temp = wq.top();
  wq.pop();

  // add information to output
  assigned_workers_[next_job] = temp.i;
  start_times_[next_job++] = temp.next_free_time;

  // update the next free time
  temp.next_free_time += pq.front();
  pq.pop();
  wq.push(temp);
}
```

We take the thread at the top of the priority queue (referred to as temp) and remove it. We then record the ID of the temp and the job amount. Then, we add to it the job at the front of the input list and reinsert it back to the priority queue. It is necessary to remove it and reinsert it because an element in a priority queue can not be updated directly.

Once all of this was finished, the program was ready to go. Most of the template was kept as the framework for the project, and here are some results for examples in the instructions: