Jake Tremblay

Danny Ha

Brian Richey

Programming Assignment 1 Report

Submitted Files:

● **PCB.h:** A header file that declares the class for PCB objects

● **PCB.cpp:** A source file that contains the functions for the PCB classes

● **ReadyQueue.h:** A header file that declares the class for ReadyQueue objects

● **ReadyQueue.cpp:** A source file that contains the functions for the ReadyQueue class

● **Main.cpp:** A source file that contains the code for test 1 and test 2

● **MaxHeap.h** A header file that contains the functions for MaxHeap

● **MaxHeap.cpp** Source file for MaxHeap that contains all functions for MaxHeap sorting

● **Makefile:** Allows us to compile the above files using the make command and creates executable prog1, also creates executables (.o) for all of the source files

How to Compile and Run the Program:

To compile the program, use command: make prog1

To run the program, use command: ./prog1

Results and runtime of test 2:

Our program was able to successfully complete test 1 and 2 and all of our classes are functioning. To get the average runtime of the loop of 1,000,000 iterations in test 2, we ran the program 5 times and took the average.

Time 1	Time 2	Time 3	Time 4	Time 5
0.106127s	0.107593s	0.106770s	0.106887s	0.108242s

Avg Time of 0.1071238 seconds for all instances of the program.

Features Implemented

● **PCB** -

Each PCB consists of the following attributes: ID number, priority number, and a process state. PCB contains a basic constructor and a parameterized constructor that allows for specific creation of processes.

● **MaxHeap** -

The MaxHeap class is what makes our program run quickly. MaxHeap implements the ReadyQueue such that the highest priority process will always be at the "front" of the heap (or front of the queue rather). A MaxHeap object will have a size attribute that reports how large ReadyQueue is, as well as an internal vector of PCB addresses. Within MaxHeap, we have functions siftUp and siftDown that will take allow us to insert/delete PCBs from the correct locations. These methods have been adapted from previous CS311 HeapSort code to allow for comparison of PCB priority rather than just integers.

● **ReadyQueue** -

Our ReadyQueue class is essentially a just a heap that is stored in a
max-heap order. All the functions of ReadyQueue will manipulate our max heap of processes. The ReadyQueue constructor will create a new MaxHeap object that can be added to, removed from, or displayed using the ReadyQueue functions.

Additional Features:

- Sift down/up (HeapSort helper functions): Allow us to swap around PCB to correct locations inside the ReadyQueue, as discussed above.

- Pragma: Something new we all hadn't seen before, the pragma once keyword inserted at the beginning of header files tells the compiler that the header will only be included in one destination file, avoiding need for include-guards (https://stackoverflow.com/questions/1263521/what-is-pragma-used-for)

Design and Implementation choices:

We chose to implement the ReadyQueue as a max-heap specifically because the assignment notes that process priority is stored such that higher number means higher priority. Drawing on knowledge gained in past courses (CS 311), we know that a max-heap is able to deliver the highest priority in O(1) time because of the nature of max-heap. To simplify the creation of PCB objects, we included an extra constructor that allows us to specify the fields of PCB at creation. Within the MaxHeap, we decided to use a vector of PCB addresses for a few reasons: 1- the vector class allows us to utilize the pop_back() function and 2- we need to have the addresses of the processes in order to effectively swap them during siftUp and siftDown. The pop_back() Function from vector is very useful when we have to do an extract function on the max-heap because we have to move the last element into the front of the list, then remove the tail (which is

made easier with pop_back()).

Time/Space complexity: inserting: **O(n log n)** removing: **O(log n)**

Case 1(we flip coin and insert 1m times) [worst case] In this case, we must call our siftUp function from a deeper level than normal. So as the level of our heap gets further and further out, we end up with **O(n)** calls to siftUp, which is **O(log n)** = **O(n log n)**

Case 2(roughly equal) [average case] In this case, the most likely of the three, we end up with a mostly normalized distribution of inserts and deletes. In this case we end up with an average time of **O(log n)**

Case 3(we flip coin and remove 1m times) [best case] In this case, we get a better performance than inserting, because siftDown only needs to call on all internal nodes of the heap. Leaf nodes need not be sifted down as they are leaf nodes at the lowest level of the heap. In this case we end up with **O(log n)** calls to siftDown.

Lessons Learned/Re-Learned:

One issue we ran into early on was not taking time to draw out the issue on paper. It happens far too often, that someone will end up programming themselves into confusion. It is always best to draw out your plan and stick to it.

References:

Most importantly, we used some of our previously written code from CS 311 to help implement the MaxHeap functions. We also needed to look up a few things, specifically the reference noted earlier about what pragma once does in c++. Also we used a stackoverflow article linked from the professor to do the timing of the test2.

Misc. (extra things done, future improvements etc):

In the future, we could improve the program by researching more about optimization of heap memory space and compiler optimization. I believe we could also refactor the files such that the MaxHeap files could be removed and instead we could implement the swap, siftUp, and siftDown

functions within ReadyQueue.