Kenny Liu, Jesus Mendoza

Dr. Xiaoyu Zhang

CS 433 – Operating Systems

11 September 2023

Assignment 1 Report

Submitted Files:

- pcb.h
- pcbtable.h
- pcbtable.cpp
- readyqueue.h
- readyqueue.cpp

How to Compile/Run the Program:

To compile, command used: make

To run, command used:

- ./test1 && ./test2
- g++ test1.cpp (or test2.cpp) pcbtable.cpp readyqueue.cpp ⇒ followed by ./a.out
    - Unfortunately, this method was time consuming, so then the command to use most of the time would be ./test1 && ./test2

Results and runtime of test 2:

Our program ran both test 1 and 2 with all the supported files in complete fashion. Nevertheless, we ran the program 5 times to get an exact average time to calculate the following table below.

Time 1	Time 2	Time 3	Time 4	Time 5
0.130639	0.093292	0.140748	0.054367	0.120954

With this in mind, the average execution time is expected to be 0.0971266seconds.

Features Implemented (via. class):

1. PCB - includes an enum class to represent the following process states in the system, which includes an ID, priority, and state.
    a. Constructor sets at this pointer of the following variables in the public PCB class.
    b. Each variable has its own get() to return the ‘this’ pointer of the variable.
    c. Both priority and state will have a set() that will have a ‘this’ pointer assigned to the variable.
    d. display() will output the following process states that corresponds to the following cases (will be used when its on the readyqueue.cpp)
2. PCBTable - This will use a vector array of all PCB elements inside of the system, which have the variables, size (checks how many PCBs are in the table) and processes (vector array.)
    a. Constructor includes the vector array that has the resize function to check the sizes available inside the table (array pointer.)
    b. Destructor includes a for loop to go through the size of the vector array, which

will delete the elements inside the array and will set the array to NULL once all the elements have been deleted. Clear the array after the loop and set the size to 0.

c. getPCB() will return the following element in a given index.

d. addPCB() will use the insert function of the vector stl class which will iterate the fronts and ends at index PCB, then iterate the size.

e. operator[](unsigned int index) will overload of the operator [] that returns the PCB at index

3. ReadyQueue - Queue of PCB's that are in the READY state to be scheduled to run, which will be a priority queue of the process with the highest priority to be selected next. Variables consist of the PCB* array and count.

a.ReadyQueue() is the constructor where it creates a new ReadyQueue object. In this we store our capacity, size and PCB's named heap because we are using the heap sorting algorithm

b. ReadyQueue(const ReadyQueue& object) is a copy constructor that copies one existing ReadyQueue object to the new ReadyQueue object.

c. ReadyQueue operator=(const ReadyQueue& object) is another copy constructor but lets us overload the equal operator so that way we can use it for swap() function. Also we have to make sure we are passing by reference and not by value.

d. swap(PCB* &heap1, PCB* &heap2) is our swap function when we have to swap our PCB objects. We also want to make sure that we are passing by reference to not change the address of our heap and focusing on our data inside our object.

d. addPCB() will add a PCB to the queue by first checking the queue being full. If not, then store the new pcbPtr and put it at heap[size1] and to make sure we update the state of the PCB to READY. Then we have to percolateUP(size1). And finally increase our size1 by one because our size1 object is the object holding the amount of elements in our array of vectors.

e. percolateUp(int index) is a function to make sure the top or first index of the array is the highest priority. So first we need to know the fatherIndex of the given index. Then check if the index is valid by making sure its not less than or equal to 0. Then we check if the priority at the given heap[index] is > than heap[fatherindex] priority, then we swap because the old index is the true father/bigger priority than the old fatherindex. And we recursively call percolateUP with the old fatherindex.

f. removePCB() will remove a queue object PCB in the array. It first creates a new PCB retval to return that pcb. It then checks if the size1 is > 0 because we want to make sure that we have pcb's in our function. If there are elements inside the queue, we set the variable retval to the root(heap[0]). Note: we have to change the state first to running and then store it in retval. We check if we have one element then we change that to nullptr and size1 to 0 and return the retval. If it's not one but more pcb's, then we put the bottom pcb to the top by heap[size1 - 1] and decrement the size1 by one. We then call percolateDown(0) to make sure we sort the top with the highest priority. If we have no elements at all, return 0.

g. percolateDown(int index) is to make sure our tree have the highest priority on

top and the bottom be the lowest priority pcb's. We need to find left and right child of the parent and then compare the priority while making sure that left and the right child index is valid. We swap our two heaps of old index != indexbigger(indexbigger = leftchild or rightchild.). We then recursively called percolate down again.

h. size() will return the length of the array.

i. displayAll() will set the following respective statements when the queue is empty or not.

Additional:

- Vector PCB* pcb used to store PCBs to include within the system for pcbtable.h

Choice of Data Structure:

After trying out multiple different types of data structures, we ended up deciding to use a binary heap for this program because of its efficiency, predictable behavior, and ease of implementation. Our binary heap ensures that the PCB with the highest priority is always at the top of the heap. This allows for efficient insertion and removal of the highest-priority element. Another reason why this data structure is efficient is because it offers O(log n) time complexity for both insertion and removal of elements. It also maintains a balanced structure which efficiently maintains our memory storage. The implementation of this data structure was straightforward as it consists of simple operations like percolating up and down which we learned from CS311. The binary heap provides a predictable behavior because we know that the item with the highest-priority would always be the root. This predictability is essential as it ensures that the processes with higher priority are always scheduled first.

Time Complexity of Implementation:

Insertion (best, worst): O(1) , O(Log N)

Deletion (best, worst): O(1) , O(Log N)

Lessons Learned/Re-Learned:

In the beginning, we were overwhelmed by the instructions and were unsure where to start. Eventually, we started to understand how we wanted to implement our program and what kind of data structures we would like to use. We even utilized the office hours to see if we were headed towards the right direction and asked for clarification on some topics which were beneficial. Personally, we thought the hardest part of this assignment was debugging. It was difficult to trace my errors at first, especially the segmentation faults, but after testing our code with gradescope, we had an easier time understanding where the errors were occurring and why. In conclusion, it felt satisfying to be able to trace the bugs and fix them.

References/Resources:

In our readyqueue.cpp, we were able to reference some of the code we had written in CS 311 to implement Heap. Aside from that, we utilized old CS 311 notes and many online sources to help solidify our understanding of topics such as max heap, vectors and pointers, and much more.