Programming Assignment 1 Report

Submitted Files:

- **PCB.h**: A header file that declares the class for PCB and PCBTable objects
- **PCB.cpp**: A source file that contains the functions for the PCB and PCBTable 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
- **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 well to the best of our knowledge. 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**
0.126959 seconds	0.125481 seconds	0.129454 seconds	0.130858 seconds	0.126191 seconds

Using this data, we determined the average execution time to be 0.1277886 seconds.

Features Implemented (by class):

- PCB - The first object type that we had to define was PCB.
  Each PCB consists of the following attributes: ID number, the last declared PCB's ID number (last_id), priority number, state, and a bool inQueue that tells us whether the PCB is in a ReadyQueue or not.

  The only function that PCB has is a constructor that initializes each of these attributes.

The static int last_id is also initialized in PCB.cpp and is used to allow each new PCB to have an ID equal to the most recently declared PCB's ID + 1 (unique IDs).

- PCBTable - Our PCBTable class is very simple and consists of an array (named table) of 20 PCBs.

  PCBTable has a single function that displays all of the PCBs in table, (displayAll).

- ReadyQueue - Our ReadyQueue class is essentially a priority queue implemented as a heap tree. To do this, we used an array of PCB pointers (named queue) so that the ReadyQueue can link back to the PCBs in the PCBTable by reference and adjust their contents.

  A ReadyQueue object has the following attributes: A PCB* array (queue, that we discussed above), and an integer (num_PCBs) to tell us how many PCBs are in the queue at a given time.

  The ReadyQueue functions are mainly used to maintain the structure of the heap, with some added basic functions. As far as the basic functions go, we have a default constructor that initializes attributes, we have function (size) that returns num_PCBs, and a function that displays the contents of the queue (displayQueue). Our functions to insert (insertProc) and remove (removeHighestProc) are structured to accommodate the heap tree. Functions getLeft (returns the index of a PCB's left child), getRight (returns the index of a PCB's right child), and getParent (returns the index of a PCB's parent), smallerChild (returns the index of the least-valued child of a PCB), and heapify (restructures the heap after removing the highest-prioritized PCB) are implemented for the sole purpose of maintaining the heap.

Additional Features:

- Global variable MAX_SIZE to allow an adjustable size for ReadyQueue
- Vector notInQueue (in main) to store all PCBs not currently in the ReadyQueue

***Design and Implementation choices*:**

At first, we chose to implement the PCBTable as a hash-table and the ReadyQueue as a circular array. We thought that PCBTable would be more efficient this way because we did not fully understand what functions it did and did not need to perform. We figured that we would organize the PCBs by priority in the PCBTable and then insert them into the ReadyQueue in-order. But, then we realized that it would be a lot more efficient for our tests to be able to just index into PCBTable by ID. It was ReadyQueue who needed to use the priorities rather than the PCBTable. After lots of errors and time wasted, we understood that PCBTable did not need all of the capabilities that we gave it and using a simple array would be much faster.

So, we changed our PCBTable to be an array and made ReadyQueue a heap tree because now we knew we needed a priority queue, and, to our knowledge, using a heap was the fastest way to implement that (see: references).

***Lessons Learned/Re-Learned*:**

From this assignment, we learned that we need to ask more specific questions about the assignment before diving into the assignment recklessly. We could have avoided all of our design issues if we had started with pseudo-code and walked through the necessary tests. The major design faults that we originally had were a result of us not understanding the assignment itself. We also re-learned a lot of (algorithms and data structures) that we learned from previous CS classes. Before this assignment, we had forgotten that a heap tree even existed among many other things.

***References*:**

Most noticeably, we used some of our previously written code from CS 311 to re-introduce ourselves to C++. This is how we figured out to use the heap as the priority queue, as we had used one in 311 and further research made it seem like our best option. We also used an online source (stack overflow) to figure out how to implement the gettimeofday function in order to record the execution time of our program, as we had a hard time learning about it from the manual.

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

In the future, we can improve this program by adding more test cases to ensure it works in a greater variety of situations. One thing we noticed throughout the coding process is that we didn’t use all of the different states, and consideration of all of the states is what led us to use a hash-table for PCBTable originally. If we were to add a terminated state, we might have to rethink our current design and its efficiency. Once we have to start removing from the PCBTable and we can no longer index into it by id, the array design becomes much less efficient. We could also stand to go through our code and do some more optimizing.