Project 05: Processor Priority Queue

Goals:

With this project, we are demonstrating compounding our data types. We are going to model a processor handling processes as they come into the structure, putting them at a proper priority, and executing them as they are told. We will use both a list and a priority queue to handle this. We will also use lower level data types to build up a more complicated managed data structure.

Project Description:

For this project, You will be implementing the functions necessary for both the List and the Priority Queue for this structure. You are given the Process Object, the main executable, and then the completed Headers for ProcessList and ProcessQueue. You are also given a lot of comparison operators that allow you to compare Process and ProcessList by their Priority level.

The idea here, is you have a variety of priority levels that a process can be. From there you can have multiple processes in that level. We will use a Priority Queue to handle the process level, and a regular queue style list to handle our list of processes. There is a twist though!

In a previous lesson you have been introduced to std::vector<T> as a storage container. This is a managed dynamic array data structure within the standard template library. Its nice, but for this example, we will use the std::deque<T> in its place. Both our Priority Queue and List will use this as the underlying data structure. Deque has many of the same operations that vector had, and the full list can be found here: https://en.cppreference.com/w/cpp/container/deque
Pay particular attention to the following:

front()
back()
pop_front()
pop_back()
push_front(T&)
push_back(T&)

The std::deque has an operator=(std::deque&) operator, meaning you can assign one vector from another (this will be handy for a swap). Please see the video on the std::deque for more information.

For this project you are implementing two sets of files:
From PriorityList:

- SwapList(). This will swap the data and priority of the calling list (this) and the other list.

- Pop(). This will grab the last item in the deque, remove it from the deque, and return it.
- Push(). This will push this item to the front of the list.

The point of the priority list is to maintain the idea of FIFO (First In First Out) that a queue mantains.

From the ProcessQueue:

- PushProcess(). This will handle multiple cases. 1) If the priority does not exist in the queue as is, it needs to create a new list to be appended to. If this happens the Priority Queue needs to be rebalanced as is done with Heaps. 2) if it does exist, it needs to add this item to the list appropriately. In both cases, this is where the process will gain a PID, and is initialized. The PID will be incremented every time this function is called.
- ExecuteNextProcess() this will grab the next process available. This will be from the "highest priority" item in the queue. This is grabbed from the appropriate item from the list. After this process is grabbed, if the list is empty, this list needs to be removed from the queue and the queue needs to be re-balanced as needed. After all of this is done, you will call the process.ExecuteProcess(std::cout). You will pass it std::cout.
- ReheapUp and ReheapDown. These two functions function almost EXACTLY like they do in a regular heap / priority queue. There are a few changes to watch for.
- IsEmpty() this should be a pretty quick one to figure out, you need to let the caller know there is nothing in the std::deque.

All operator overloads, the Process object, and a few other functions in PriorityList are implemented for you.

C++ 11

This is another project that is using C++11. From here, we have a few items that have been added:

1) You will see the word "default" in front of all the constructors. This means that all variables are initialized to a value in the header, and as such, the constructor does not need to be implemented.

2) Default initialization of variables. This occurs in the header file. If you look at PriorityList.hpp, the PID is initialized to 0 when this object is created.

3) enum class. This is a way to scope the enumeration. Instead of saying enumVal = Low, this forces you to write enumVal = Priority::Low; For more information see my lecture on this.

Compiling:

You will be able to compile this with a make command.

```
-bash-4.2$ make
```

You can then run the executable by running the following:

```
-bash-4.2$ ./Project05 NameOfInputFile
```

Much like all previous examples, you can run a "Make clean" to remove all object code and make sure your compile is in a fresh state.

Debugging:

Your program must be fully commented (including variable and parameter declarations, function descriptions, descriptions of logical blocks of code, etc.) in order to receive debugging assistance from the instructor and teaching assistants. See Canvas for an example of an acceptable commenting style.

Submission:

Make sure to follow the checklist below to verify you get full credit on your submission. You MUST implement all of these classes and each of these files will need to be submitted through Canvas.

Please submit the following to canvas:

- ProcessQueue.cpp
- PriorityList.cpp

DO NOT MODIFY MAIN.CPP
DO NOT MODIFY ANY HEADER FILES TO MAKE YOUR CODE WORK
DO NOT MODIFY PROCESS.CPP