Array of Struct Variables, Searching: Process Control Block List

The purpose of this assignment is to validate certain skills you must have on entering CS 1704. If you have much difficulty in completing this assignment, you should carefully consider whether you are ready for this course. The remaining projects will steadily become much harder.

This project requires implementing a vastly simplified simulation of a process-scheduling system as might be used in the kernel of a multi-processing operating system. Logically a process is represented by a relatively simple data record, called a process control block (PCB), which encapsulates some of the essential attributes of the process. Assuming the underlying hardware contains only one CPU, then only one process can actually be executing at any given time. To achieve the illusion of multi-tasking, most modern operating systems keep the PCBs for existing processes in some sort of data structure and cycles the processes, giving each the CPU for short periods of time. The act of moving the running process to a waiting state, and selecting another process to run is called a context switch.

In this project, you will use a simple array to hold the PCBs, which will simply be variables of an appropriate `struct` type. Your system will keep track of which process is currently in simulated execution, and which processes are waiting for their turn. You will support making context switches, and also killing an existing process. All of these actions, as well as the creation of a new process, will be triggered by commands read from a script file.

PCBs for processes that are eligible to run, but not in the run state, will be stored in an array of dimension 10. We will refer to this as the eligible queue (although it is not, strictly speaking, a queue). Newly-created processes will be added at the first available cell of the array, if there is one. On a context switch, the process whose PCB is at the front of the array will be moved to the run state (removed from the array), and the PCB for the previously running process, if any, will be appended to the array. If the array is empty, no context switch occurs.

Note that you will not be simulating the actual execution of the processes that are being scheduled, just the scheduling itself. Note also that you are explicitly forbidden to write any classes for this project. That will come later. Note also that you will not be allocating any data dynamically in this project; i.e., you are forbidden to invoke `new` or `malloc`.

On startup, your program will initialize the necessary data structures then read commands from a script file and log the results of processing those commands to an output file.

Script file:

This project will involve only one input file, containing the commands that are to be processed. The script file, named `"PCBScript.txt"`, contains a sequence of commands, one per line. Each command is to be processed as soon as it is read, so you will not need to read and store all of the commands. There will be a number of commands, described below:

`start``<tab><process name><tab><priority><newline>`

Create a PCB for the process and append it to the eligible queue. If the eligible queue is full, then no PCB is stored and the process is simply lost. The system assigns a unique process identifier (PID); the PID is just a nonnegative integer, corresponding to the sequence in which the processes are created, starting with zero.

`kill``<tab><PID><newline>`

Search for a PCB storing the given PID. If one is found, remove the PCB. Note the PCB may be in the run state or on the eligible queue; deletion may leave either of those empty. (There's no automatic context switch.)

`switch``<tab><newline>`

As described above, if the eligible queue is empty this has no effect. Otherwise, the first PCB in the eligible queue is moved to the run state and the running process is moved to the end of the eligible queue.

`ps``<tab>[` **`all`** `|` **`PID`** `]<newline>`

If the parameter is `all`, log all the existing processes, showing the process name, PID, and current state (RUNNING or ELIGIBLE), and the process priority. The running process, if any, should be logged first, followed by the processes in the eligible queue, in stored order. If the parameter is `PID`, locate the PCB for the specified process and log it as described above.

`sort by<tab>[ name | priority ]<newline>`

Sort the list of eligible processes using the specified PCB field as the sort key. Sorting by name should place the PCBs in ascending order by name; sorting by priority should place them in descending order by priority.

This particular command is somewhat artificial; it is unlikely that a real OS scheduler would support something like this; it is, however, useful for validating certain skills.

`exit<tab><newline>`

Immediately stop reading the script and exit the program.

Legend:

`process name`	a character string, containing no tabs, with no more than 20 characters
`priority`	a non-negative integer value; higher values indicate greater priority
`PID`	a non-negative integer value identifying a process, assigned upon process creation

As a general rule, every command you process should result in a logged message, either confirming the command was carried out or indicating that an error occurred. There is no limit on the number of commands that may be given. Your program should be designed to stop when an input failure is reached. Here is a sample command file:

```
; PCB List Script file
;
; Load some processes:
start     P01    1
ps  0
start     P02    3
start     P03    1
;
; Check the status of the jobs:
ps  all
;
; Sort and check again:
sort by   priority
ps  all
;
; Put a job into the run state:
switch
;
; Verify:
ps  all
;
; Do a context switch and verify:
switch
ps  all
;
; Again:
switch
ps  all
;
; Add another process:
start     P04    2
ps  3
;
; Kill two and verify:
kill      2
kill      0
ps  all
;
; Shut down:
exit
```

The script file is guaranteed to conform to the specified syntax. However, it is always a good idea to design for invalid entries, such as an invalid sort field specifier. Note it is entirely possible that there may be logically invalid data, such as a PID for a non-existent process. Deal with it. Sensibly. Additional sample script files will be posted on the course website.

Log file:

Your program will write all of its output to a log file, named "`PCBLog.txt`". Note that it is important to conform to the log file formatting specified in this document.

The log file will begin with some identification lines specifying the name of the programmer and the assignment. Then, for each scripted command, the log file will echo the complete command and then the resulting output. As always, be careful to match the fixed text shown below (e.g., "`ELIGIBLE`").

The logged output for each command must be delimited for easy reading, as shown. Each echoed command must have a descriptive label, as shown.

For a start command, log that the process was added or that it was not, as shown in the sample log. For a kill command, log that the process was found and removed, or that it was not found, as shown. For a ps command, log the shown information for the relevant process(es). For a switch command, log which process has been moved to the run state, as shown. Finally, for the exit command, just log that the program is exiting, as shown.

Here is a sample log file:

```
Programmer: Bill McQuain
CS 1704:    Process Control Block List

--------------------------------------------------------------------------------
Command: start   P01    1
Process 0 added.
--------------------------------------------------------------------------------
Command: ps      0
P01                         0     ELIGIBLE       1
--------------------------------------------------------------------------------
Command: start   P02    3
Process 1 added.
--------------------------------------------------------------------------------
Command: start   P03    1
Process 2 added.
--------------------------------------------------------------------------------
Command: ps      all
P01                         0     ELIGIBLE       1
P02                         1     ELIGIBLE       3
P03                         2     ELIGIBLE       1
--------------------------------------------------------------------------------
Command: sort by priority
Sorting list of 3 processes by priority.
--------------------------------------------------------------------------------
Command: ps      all
P02                         1     ELIGIBLE       3
P01                         0     ELIGIBLE       1
P03                         2     ELIGIBLE       1
--------------------------------------------------------------------------------
Command: switch
Process 1 now in run mode.
--------------------------------------------------------------------------------
Command: ps      all
P02                         1     RUN       3
P01                         0     ELIGIBLE       1
P03                         2     ELIGIBLE       1
--------------------------------------------------------------------------------
```

```
Command: switch
Process 0 now in run mode.
---------------------------------------------------------------------------------
Command: ps      all
P01                         0     RUN       1
P03                         2     ELIGIBLE       1
P02                         1     ELIGIBLE       3
---------------------------------------------------------------------------------
Command: switch
Process 2 now in run mode.
---------------------------------------------------------------------------------
Command: ps      all
P03                         2     RUN       1
P02                         1     ELIGIBLE       3
P01                         0     ELIGIBLE       1
---------------------------------------------------------------------------------
Command: start   P04    2
Process 3 added.
---------------------------------------------------------------------------------
Command: ps      3
P04                         3     ELIGIBLE       2
---------------------------------------------------------------------------------
Command: kill    2
Process 2 removed.
---------------------------------------------------------------------------------
Command: kill    0
Process 0 removed.
---------------------------------------------------------------------------------
Command: ps      all
P02                         1     ELIGIBLE       3
P04                         3     ELIGIBLE       2
---------------------------------------------------------------------------------
Command: exit
Exiting script execution.
---------------------------------------------------------------------------------
```

Evaluation:

Do not waste submissions to the Curator in testing your program! There is no point in submitting your program until you have verified that it produces correct results on the sample data files that are provided. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, out of 100, based upon the runtime testing performed by the Curator System. We will also be evaluating your submission of this program for documentation style and a few good coding practices. This will result in a deduction (ideally zero) that will be applied to your score from the Curator to yield your final score for this project.

As stated in the course policies, the submission that receives the highest score will be evaluated. If two or more of your submissions are tied for the highest score, then the earliest of those submissions WILL be evaluated. Therefore, DO NOT make undocumented, incomplete submissions to the Curator and then complain that you didn't want those to be evaluated. I will grant NO exceptions.

Your implementation must meet the following requirements:

- Choose descriptive identifiers when you declare a variable or constant. Avoid choosing identifiers that are entirely lower-case.
- Use named constants instead of literal constants when the constant has logical significance.
- Use C++ streams for input and output, not C-style constructs.

- Use C++ `string` variables to hold character data, not C-style character pointers or arrays.
- You must make appropriate use of functions. ~~You should have a function that reads the vehicle data file and stores its contents into the data arrays.~~ You should have a function for servicing each of the commands. You should have a function that initializes the data arrays to sensible starting values. There are certainly other good candidates for functions. Aside from some testing functions, my solution uses ten functions besides `main()`.
- None of your functions can include more than 30 executable statements. An executable statement is one that causes something to happen. Comments, constant and variable declarations (even if they initialize) do not count as executable statements.
- When you pass an array to one of your functions, use pass by constant reference unless pass by reference is logically necessary. (Remember that for array parameters, pass by reference is the default.)

Reflecting the primary facet of this project, you must use a single array of `struct` variables to store the eligible queue of PCBs. (The use of parallel arrays is explicitly banned for this project.) You must initialize each of the cell of the `struct` array to some sensible default values. As always when arrays are involved, be careful about array indices and about the difference between the dimension and the usage of an array.

Read the *Programming Standards* page on the CS 1704 website for general guidelines. You should comment your code in some detail. In particular:

- You should have a header comment identifying yourself, and describing what the program does.
- Every constant and variable you declare should have a comment explaining its logical significance in the program.
- Every major block of code should have a comment describing its purpose.
- Every function implementation must have a header comment. The format of the header comment is described in the course notes, as well as on the *Programming Standards* page on the course website.
- Adopt a consistent indentation style and stick to it.

Understand that the list of requirements here is not a complete repetition of the *Programming Standards* page on the course website. It is possible that requirements listed there will be applied, even if they are not listed here.

Submitting your program:

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* and submission link can be found at: `http://www.cs.vt.edu/curator/`

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the course website in the header comment for your program.

Failure to include the pledge statement may result in a substantial grade penalty.