Assignment 4 -- Process Scheduling (100 points)

In this assignment, we are simulating priority scheduling of processes on a single-processor system, without preemption. The idea is that when a process ends its CPU burst (or input burst or output burst), it is succeeded by the highest-priority process that is waiting.

For the sake of this assignment, we will make a few simplifying assumptions such as:

(a) We will assume the processes engage in CPU bursts, Input bursts and Output bursts and ignore any possible interrupts.

(b) We will assume that all processes are doing Input through the same device which can process one Input burst at a time.

(b) We will assume that all processes are doing Output through the same device which can process one Output burst at a time.

(c) We will assume the system starts out with no processes active. There may be processes ready to start at once.

(d) We are ignoring the context-switching time.

(e) The array called History (described below) is a convenience. (An alternative would be to generate the data dynamically using random numbers. Let's not do that; this is complicated enough.)

The events are governed by a timer which is simply counting clock ticks. (For our purposes, a clock tick might be one millisecond).

A description of the input file is provided below.

Write a program in C or C++ on the hopper system to accomplish this. In your program, you will need a struct or class to represent a **Process**; you will need a struct or class to implement a **queue**; you will need a struct or class to implement **priority queues**. The items stored in the queue and priority queues are pointers to processes. You may also need a struct for entries in the **History** array.

The program requires 4 queues (including priority queues): Entry, Ready, Input and Output. We will also have variables Active, IActive, and OActive, which are pointers to processes.

The Entry queue is an ordinary queue: First-In-First-Out. The other three are “priority queues”, that is, when we pop a process off the queue, it is the highest-priority process in the queue. If two processes have the same priority, the one which has been waiting longer should be next. Figure out a way to implement this.

<u>Constants</u>

You will need (at least) the following constants:

MAX_TIME: an integer = length of the whole run. Use the value 500.

AT_ONCE: an integer = maximum number of processes that can be in play at once (that is, not still in the Entry Queue or in the array). (This is called the degree of multiprogramming.) Use the value 5.

QUEUE_SIZE: an integer guaranteed larger than the maximum number of items any queue will ever hold. Use the value 20. (You might be able to get along without this one.)

ARRAY_SIZE: an integer = size of the History array to define in a process. Use the value 10.

HOW_OFTEN: an integer indicating how often to reprint the state of the system. Use the value 25.

<u>What is in the class/struct process?</u>

A process need to contain (at least) the following data:

ProcessName: the name of the process, a string.

Priority: a nonnegative integer (higher values: more important).

ProcessID: an integer, the ID number for the process. This is assigned by the system (i.e., your program). Use consecutive values such as 101, 102, 103, etc.

ArrivalTime: an integer indicating when the request for this process first arrived.

History: an array of pairs of the form (letter, value) as described below. You may assume there are no more than 12 such pairs.

Sub: a subscript into the array History.

CPUTimer: counts clock ticks for the process until it reaches the end of a CPU burst.

IOTimer: counts clock ticks for the process until it reaches the end of the I/O burst.

CPUTotal: accumulates the number of clock ticks the process spends as Active.

ITotal: accumulates the number of clock ticks the process spends as IActive.

OTotal: accumulates the number of clock ticks the process spends as OActive.

CPUCount: counts the CPU bursts for this process.

ICount: counts the Input bursts for this process.

OCount: counts the Output bursts for this process.

Input file

Each process is represented in the input file by two lines.

The first line contains (separated by white space):

--- a string = the name of this process

--- a nonnegative integer = the priority of this process

--- a nonnegative integer = the arrival time at which the request for
    this process was submitted

The second line contains a sequence of pairs of the form:

letter value (separated by spaces)

where letter = 'C' for a CPU burst, 'I' for an input burst or 'O' for an output burst, and value = the number of time units for this burst.
These describe the sequence of events for this process.

There are never two consecutive bursts of the same kind.

The end of the list of pairs is padded with pairs having letter = 'N' and value = 0.

You may assume there are no more than 20 processes in all. In the file, they are listed in increasing order by arrival time.

The last line has process name = "STOPHERE" and should not be included. It simply serves as a delimiter.

The file can be found here:

/home/turing/t90hch1/csci480/Assign4/data4.txt

If the input file is not accessible, please find the file at Blackboard and transfer it to your account at hopper/turing.

What should happen?

The structure of the program is that a process moves from the Entry queue to the Ready queue, eventually becomes Active, is moved to the

Input queue or the Output queue or back to the Ready queue as is appropriate, and eventually terminates. We will accumulate various statistics about each process and some global statistics about the run as a whole.

1. Read the input file. Create processes and store them in the Entry queue.

2. When processing starts, we have some initializations and then a loop.

3. We have a main loop. It starts with Timer = 0. Timer is incremented at the bottom of the loop, and the loop ends when Timer reaches MAX_TIME or there are no processes left (all queues empty, Active, IActive and OActive all = 0).

At this point, various things may happen in the loop:

If the number of processes in the cycle is less than AT_ONCE, look in the Entry queue. A process may move from the Entry queue to the Ready queue if its arrival time has been reached. You may initially push several processes onto the queue, but later it will normally be one at a time (after another process has terminated).

If we do not have an Active process, we need to look for one in the Ready queue.

If we still do not have an Active process, there is little for the CPU do on this iteration of the loop: idle time.

If we do have an Active process, what could happen to it?

--- It could come to the end of a CPU burst. Look in the History array; you should find an Input or Output burst. It moves to the Input queue or Output queue.

--- It could come to the end of a CPU burst which is followed by termination of the process. Look in the History array; you should find a terminator ("N" and 0).

If we do not have an IActive process, we need to look for one in the Input queue. (It might be empty: no IActive process for a while.)

If we have an IActive process, it may come to the end of its Input burst, in which case it is moved to the Ready queue.

If we do not have an OActive process, we need to look for one in the Output queue. (It might be empty: no OActive process for a while.)

If we do have an OActive process, it may come to the end of its Output burst, in which case it is moved to the Ready queue.

4. After the main loop, print summary data about the whole run, such as these (you may think of more):

--- What was the value of the Timer at the end?

--- How many processes terminated normally?

--- What was left in each queue at the end?

--- How much time was spent doing nothing (idle time)?

This description leaves out a few details, such as initializing the variables. You may in fact need a few more variables. You will need a Terminate() function and perhaps more.

As the program runs, print output about major events such as when a process enters the cycle from the Entry queue or when a process terminates. When a process terminates, print what we know about it, such as:

--- its name and priority
--- how many CPU bursts occurred
--- how many Input bursts occurred
--- how many Output bursts occurred
--- how much time was spent in the CPU
--- how much time was spent in Input
--- how much time was spent in Output
--- how much time was spent waiting in queue (other than Entry)

Whenever the timer is a multiple of HOWOFTEN, print out the state of the system:

--- the ID number of the Active process (if there is one)
--- the ID number of the IActive process (if there is one)
--- the ID number of the OActive process (if there is one)
--- the ID numbers of the contents of the Entry, Ready, Input and Output queues

You will need to invent the report format for all this data. Put some effort into it and make it easy to read. (For the queues, try printing the list of ID numbers all on one line.)

Notice that there may be occasions when all processes are doing Input or Output, the Ready queue is empty and we have idle time for the CPU. There may be times when the Entry queue is empty because some processes have not arrived yet. (We could theoretically have fewer than AT_ONCE processes in the cycle at such a moment.)

<u>Comments</u>

Copy the input file into your own directory.

You may use other variable names if you want, but they should be reasonable names reflecting their purposes.

Your program should be appropriately indented and well documented. You can find style guidelines on the web site the CSCI 241 course.

It may be easier to write the code if Active, IActive, OActive and the queues are global variables. You may want to make some other variables global as well.

You may need to invent functions for various purposes. Be sure to document them.

You may use the standard template libraries if you wish.

As you work on this, you may find it useful to print out a great deal more information as you go along. If you do so, please ensure that the extra information is not printed by your final executable file.

Here is a list of files recommended for you to implement for this assignment. You can give different file names. You can also design your own program structure.

Queue.h	– header file of the regular queue class/struct.
Queue.cc	– implementation file of above class/struct.
PQueue.h	– header file of the priority queue class/struct.
PQueue.cc	– implementation file of above class/struct.
Process.h	– header file of the process class/struct and the data type (call it event) of elements stored in the History array.
Process.cc	– implementation file of above class/struct. It may only contain the constructor.pri
Assign4.h	– some constants used in this program are stored here.
Assign4.cc	– driver program of this assignment. It can contain the implementation of the CPU scheduling.

If you decide to use STL queue and priority_queue classes, you do not need to implement above Queue.h/cc and PQueue.h/cc files.

You should have a makefile. The name of the executable file should be "Assign4".

When you are done, you need to submit your work on Blackboard. As in the other assignments, you should create a tar file containing the files involved. To do this, you need the "tar" utility.

Do the following (replacing "Znumber" with your own Z-ID):

(a) Create a subdirectory named Znumber_A4_dir.

(b) Copy the files into it (source code files, headers, input file and makefile).

(c) In the parent directory of Znumber_A4_dir, use this command:

```
tar -cvf Znumber_A4.tar Znumber_A4_dir
```

Use an FTP program to retrieve the tar file and then submit it on Blackboard. The TA will move it to hopper, extract the files and run your makefile, as in:

```
tar -xvf Znumber_A4.tar
cd Znumber_A4_dir
make
./Assign4
```

If your makefile does not run or your program does not compile and run, you will receive no credit.