Due:

June 24, 2016, **before 11:59 pm.**

Notes

- Please follow both the "Programming Standards" and "Assignment Guidelines" for all work you submit. Programming standards 1-25 are in effect.
- Hand-in will be via the UMLearn Dropbox facility. Make sure you leave enough time before the deadline to ensure your hand-in works properly.
- Official test data will be provided several days before the assignment due date. Until then, work with your own test data.
- All assignments in this course carry an equal weight.

Assignment 2: Discrete Event Simulation

Description

To begin working in C++, you will write a program that will use a discrete-event simulation to simulate the operation of a multiprogramming Operating System. **I am assuming you have never done any work with event-driven simulations, and so I have put some background on this in an appendix posted to the website. This background will also be covered in class and will be posted to the course notes.** The operating system will manage "Processes", which enter the system, and require bursts of CPU time (our machine has a single CPU) and bursts of I/O time (our machine has only one I/O "device").

You should be making use of all the facilities for doing OO in C++ that we have covered, including appropriate use of abstract classes and methods, and safe dynamic casting. While I am not expecting you to completely follow OCCF standards, **you must write destructors to properly free any dynamically allocated memory** that you use. You don't need to worry about memory allocated in linked structures that will exist until the end of the program, but anything that is unlinked or goes out of scope during the course of the program's run must be properly disposed of. **You do not need to separately compile this program, but you can if you want to.** You may also use the *make* utility if you are familiar with it, but again, there's nothing forcing you to do that here.

Details

A data file is used to drive the simulation. The data file will be ordered by time, so you must have only **one arrival event in the event list at any time**. Each arrival must cause the next arrival event to be read. Each Process will get an ID number (start at 1) when it arrives, and each new Process will have an ID one higher than that before it. You must use a C++ static variable to keep track of the next Process number to be generated.

There are seven events that can happen to any Process:

An **Arrival** occurs when the Process is submitted. If no Process is currently executing on the CPU, you should schedule a **StartCPU** event; otherwise the Process is entered into a

queue of Processes waiting for their turn to execute on the CPU. (Use a strict FCFS queuing discipline.)

A **StartCPU** event causes the Process to be scheduled to execute on the CPU. Our OS supports *timesharing*: each Process may use the CPU for a maximum time quantum of 4 time units. Note that a real OS does not know beforehand if a Process will exhaust its time quantum, but in our simulation model, we know the length of a CPU burst, so we know whether the process will timeout (you should schedule a **Timeout** event) or complete the burst (you should schedule a **CompleteCPU** event).

A **CompleteCPU** event occurs when the Process completes a CPU burst. The Process will either have more bursts to process (the next one will be an I/O burst), or it will have finished its processing, and an **ExitEvent** should be scheduled. Be sure to check if there are Processes waiting their turn to execute, and schedule the first one on the queue to start execution.

A **Timeout** event occurs when the process exhausts its time quantum. The Process goes to the back of the queue and wait for another turn, to continue executing the CPU burst. Again, check the queue of waiting processes.

A **StartIO** event causes the Process to start an I/O operation. I/O operations are not time-shared; a Process gets to complete its entire I/O burst once it starts.

A **CompleteIO** event occurs when a Process completes an I/O operation. Be sure to check if there are Processes waiting to start an I/O operation. As with the **CompleteCPU** event, the Process will either have finished its processing, or have more bursts to be scheduled.

An **Exit** event occurs once the Process has completed all its CPU and I/O bursts and leaves the system. Final statistics are gathered about the execution of this Process.

You will need to maintain a list of future events in order by time. During the simulation process, at any point where the time unit is the same for two events, the Process that arrived earlier should be handled first. This means that as the simulation goes on you will be maintaining a list of pending events that is ordered by primarily by time, but within time, ordered by Process number. There is one exception to this rule: If an Arrival and a Timeout event occur at the same time, the Arrival event always is placed first.

Use a *command-line argument* to accept the name of the data file. Your program should then open that file and perform the simulation. This method will allow the markers to easily run your program on several different data files whose names you do not know ahead of time. Your program should write to standard (console) output, not to an output file (again this makes things easy on the markers because they can just read output in a terminal window).

Data File

The data file contains information on processes and their CPU and I/O requirements. Each process is on one line of the file. Each line consists of a series of integers that describe (in order):

- The arrival time of the process (that is, when it is submitted): a positive integer.

- A series of CPU burst times (positive integers) and I/O burst times (negative integers): this list will contain at least one CPU burst time. After the initial CPU time, the list may contain alternating I/O and CPU burst times. There is no fixed length for this list.

For example:

2 8 -20 4 -10 2

Describes a process arriving at time 2, which requires 3 CPU bursts and 2 I/O bursts.

The dataset descriptions above should be enough for you to generate your own test data to start with. Official test data will be provided several days before the assignment is due.

Data Structures

Your data structures must be your own, and you cannot use the C++ standard template library: you must make generic data structures (a Queue and an ordered list/priority queue) using your own linked structures and making use of C++'s object-orientation features. **In particular, you should have a polymorphic hierarchy of data items to go into generic data structures, and a polymorphic hierarchy of Events.**

Output

Your program should produce output that indicates the sequence of events processed and when they occurred in order by time. At the end of the simulation you will produce a summary table. Some sample output is shown below.

```
Simulation begins...
Time   2: Process  1 arrives in system: CPU is free (process begins execution)
Time   2: Process  1 begins CPU burst
Time   5: Process  2 arrives in system: CPU is busy (process will be queued)
Time   6: Process  3 arrives in system: CPU is busy (process will be queued)
Time   6: Process  1 exhausts its time quantum (requires 4 units more)
Time   6: Process  2 begins CPU burst
Time   8: Process  2 completes CPU burst
Time   8: Process  3 begins CPU burst
Time   8: Process  2 begins I/O burst

… etc.

...All Processes complete.  Final Summary:

Process    Arrival         CPU      I/O       Exit       Wait
      #       Time        Time     Time       Time       Time
-------------------------------------------------------------
      1          2          14       30         42          6
      2          5           2       22         35          6

… etc.

End of Processing
```

Hand-in

Submit all your source code for all classes. Each class should be declared as public in its own file. You should also submit two text documents:

- The output for the official test data.
- A short English description of how the interaction between objects in your project. This should answer questions like "Which class is initially called?", "Which method(s) process the input?", and which objects deal with tasks in the code. This description should be at most one page of text.

Submit all files on UMLearn.