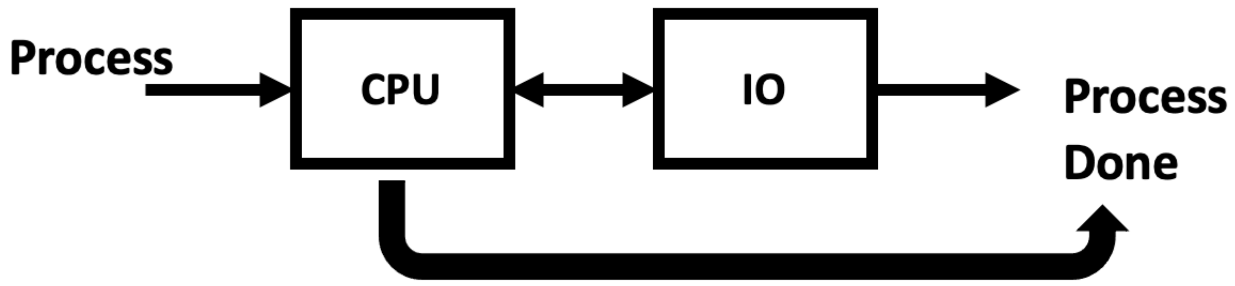
COMP 2150 - Winter 2021 - Assignment 2

Due March 11 at 11:59 PM

In this assignment, you will write a program that will use a discrete-event simulation for the operation of a multiprogramming Operating System. The operating system will manage *Processes*, which enter the system, and require *bursts* of time – that is, the processes require certain resources for a particular length of time. In our model, processes need bursts of CPU time (our machine has a single CPU) and bursts of I/O time (our machine has only one I/O "device"). The requirements of the processes will also switch back and forth between the two resources – a process will need bursts of CPU time, then I/O time, then perhaps more CPU time, etc.

That is, a Process will arrive at the CPU first, and then move back and forth between the CPU and the I/O device as needed. Whenever computation is complete, the process finishes, as depicted below. However, while on the CPU, the process may time out – the process may use up its pre-allocated amount of time on the CPU, and another process may start. Thus, each process may have to wait for CPU time if it can't complete the time it needs on the CPU at once.



Your solution to this assignment will be in C++. You should use, as appropriate, the OO tools we have developed in the course (see the section on OO Programming below). There are also specific C++ requirements, including using a makefile, using command line arguments and separate compilation (see the section on C++ specifics).

Details

A data file is used to drive the simulation.

The first line of the input file contains a single integer. This is the maximum time that a process can use the CPU at once before it times out (see Events below for more details). Note that there is no maximum time that a process can use the I/O device (as noted below in the StartIO event description).

The data file will be **ordered by time**, so you must have only one arrival event in the event list at any time. Do **not** read the entire file at once. 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.

Each process is on one line of the file (after the first line). Each line consists of a series of integers that describe (in order):

- First, the arrival time of the process (that is, when it is submitted): a positive integer.
- After the arrival time, 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, and every process starts with a CPU burst. After the initial CPU time, the list may contain alternating I/O and CPU burst times. There is no maximum length for this list, but the information for each process will be contained on one line. The I/O bursts are listed as negative numbers for identification, but they represent positive values in reality.

For example, a line with this data

```
2 8 -20 4 -10 2
```

describes a process arriving at time 2, which requires 3 CPU bursts (length 8, 4 and 2) and 2 I/O bursts (length 20 and 10). A full example input file could therefore look like this:

```
3
2 8 -20 4 -10 2
5 2 -22
6 6 -12 4
```

Official test data will be provided before the assignment is due.

Events

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 first-come first-served queue, provided to you.)

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 amount of time units (the *time quantum*, given in the input file on the first line). 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 Exit event should be scheduled. Be sure to check if there are Processes waiting their turn to execute on the CPU and schedule the first one in the queue to start execution.

A **Timeout** event occurs when the process exhausts its time quantum on the CPU. The Process goes to the back of the queue and wait for another turn on the CPU. Again, check the queue of waiting processes to see if another process needs to start execution on the CPU.

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. Thus, you should schedule a CompleteIO event.

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, and schedule one if appropriate. As with the CompleteCPU event, the Process that completed its I/O operation will either have finished its processing, or have another burst 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. (Note that you may need to store all Processes in your simulation for the duration of the program to ensure that you are able to print the summaries for each of them.)

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 to the Operating System 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 time. However, if two pending events occur at the same time, then they are ordered by Process number (earlier arrivals get lower process numbers). There is **one exception to this rule**: If an Arrival and a Timeout event occur at the same time, the Arrival event always is handled first, regardless of the process numbers of each.

As in the provided code, 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 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).

Simulation and Provided Code

The basic simulation will follow the logic presented in the video on umlearn. However, your simulation will be more detailed. In particular, you will likely observe that your simulation needs to maintain more than one queue.

You are provided with some basic code:

- a main file (A2main.cpp) that starts the simulation.
- the header file for the simulation class that defines the methods that the simulation (Simulation.h).
- the header file for the general event class (Event.h) and partial implementation (Event.cpp). You will use this to define the subclasses of Event.
- an implementation of a Node, ListItem, and Queue classes.

You can modify the files as you see fit (with one exception, below), but most will require minimal modifications: for instance, you should be able to complete the assignment without modifying the Queue, ListItem and Node files. You will likely need to modify the Simulation.h by adding new fields and methods to it. You should aim to reuse code from the Node and ListItem classes for the Priority Queue that you need to implement.

You **cannot modify** how A2main.cpp reads input from the file. This must be kept the same so that the markers can be sure all programs operate in the same way.

Object Oriented Programming

Your assignment should use OO programming. In particular:

1. You should use good OO practice in general, including proper information hiding. Fields should not be public.
2. You should also not release data structures that are private. For instance, the Queues and Priority Queues contained in your simulation should be modified only by public methods in the Simulation class – **never** provide a getQueue() method that releases the entire data structure to the user of the Simulation class, for instance.
3. You should have a hierarchy of data items to go into any data structures, and a polymorphic hierarchy of Events. (That is, you will have a hierarchy of Events, corresponding to the types of Events described above. Each must implement the handleEvent method. Your handling of events through the handleEvent method must be polymorphic.)
4. You should have code reuse as much as possible. If you have duplicated code for the same task, you will lose marks.
5. You should have as little non-OO code as possible. You should not add any non-OO code to the A2main.cpp file, or elsewhere in your project.

Hint: One challenge of this assignment (and separate compilation) will be allowing the Events to know information from the larger simulation and possibly modify the state of the simulation. For instance, some Events cause other Events to be read from the file. One way to achieve this is by allowing all Events to have a pointer to other classes, including the class that created it.

C++ Requirements

1. You must read the names of the file from a command line argument, as is done in the provided main.
2. You must use separate compilation for all classes. You should provide minimal includes and forward references as much as possible.
3. You must write a makefile to build your project. The command "make" with no arguments should build the project. (i.e., the default rule should produce the executable.) The command “make test” should build the executable for testing (see below).

Data Structures

The data structures you write must be your own, and you cannot use the C++ standard template library (STL): you must make data structures (in particular, a priority queue) using your own linked structures and making use of C++ OO features. If you don't write your own data structures, you will lose marks. You should not use arrays in the assignment code for **any** tasks (arrays are ok for unit testing).

For this assignment, you do not need to concern yourself with data structure efficiency. Your Priority Queue should maintain items in sorted order, but you do not need to implement any more efficient operations (e.g., a heap).

When dealing with hierarchies and data structures, you must use safe casting practices.

Unit Testing

Construct a set of unit tests for your code. To do unit testing in C++, you should:

1. Use Catch2 for a simple test framework. Download Catch2 as a single header file from the course website. You must use a C++11 compiler (`clang++ --std=c++11` on aviary) for catch2.
2. Place the catch.hpp in the same directory as your code.
3. Create a new file for the tests.
4. Preprocessor commands: add these commands (in this order) to the test file:
   ```
   #define CATCH_CONFIG_MAIN
   #include "catch.hpp"
   ```
5. Use the TEST_CASE and REQUIRE to write test cases. (You can see a tutorial for catch2 here). Each test must have an assertion (REQUIRE) in it.

Your tests should focus on the data structures you create for your project. You should include at least five meaningful for your data structure(s). You should **not** test the Queue class. You will be graded on your tests, so write useful tests. The markers will be running your tests, as well, so ensure that they pass. If tests do not pass or your code does not compile, you will lose a substantial number of marks.

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 that shows the arrival time, exit time and total wait time (CPU waiting plus I/O waiting) for each process, in order by process ID. Some sample output is shown below (**as an example format only – may not correspond to an actual execution of the program)**.

```
Simulation begins...
Time     2: Process   1 arrives in system: CPU is free (process begins execution).
Time     2: Process   1 begins CPU burst (will time out; needs 8 units total).
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 times out (needs 4 units more).
Time     6: Process   2 begins CPU burst (will complete all 2 remaining units).
Time     8: Process   2 completes CPU burst. Queueing for I/O.
Time     8: Process   2 begins I/O burst of length 22.
Time     8: Process   3 begins CPU burst (will time out; needs 6 units total).
Time     8: Process   2 begins I/O burst

[lots of lines deleted]

...All Processes complete.  Final Summary:

  Process   Arrival      Exit      Wait
        #      Time      Time      Time
---------------------------------------------
        1         2        74        28

[ print one line for each process, in order by process ID. ]
```

The format is not strict, but you should make your output as easy to read as possible.

Hand-in

Submit all your source code for all classes, including all provided code (modified or not) and the catch.hpp file. Use separate compilation and have each class in its own .h and .cpp file. Your main function should also remain in a separate .cpp file. Also include your makefile. Make sure that for all assignment code you follow the programming standards given on the website.

You MUST submit all of your files in a zip file. Additionally, you **MUST** follow these rules:

- Include a README.TXT file (in the zip file) that describes exactly how to compile and run your code from the command line. (It is understood that the makefile and command line arguments will be the same for everyone, but in case of problems, the markers will be reading this file). If your code does not compile directly, you will lose marks. Code will be run on a standard linux environment (aviary) using the makefile you provide.
- You should also submit a text document (in the zip file) giving the output on the official test data. (Official test data will not be released until just before the due date.)

The easier it is for your assignment to mark, the more marks you are likely to get. Do yourself a favour.

Submit your zip file on umlearn.