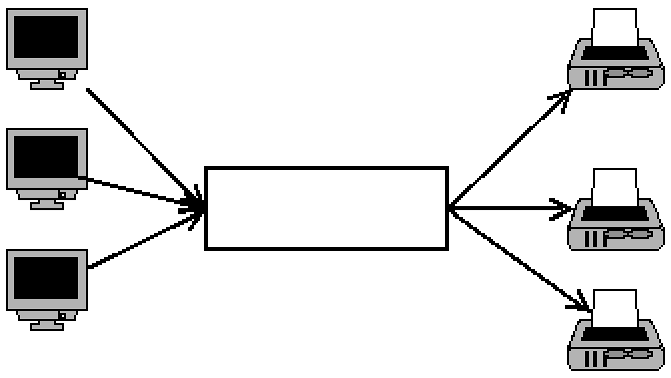
Event-driven simulator (C++11 allowed)

For this assignment you have to create a simple event driven simulator of a print queue.



Print queue

Print queue is a buffer between the clients and printers. Clients, instead of submitting theirs jobs directly to a specific printer, send them to the print queue. Print queue will keep jobs in an internal queue until there is an available printer, at this point job will be assigned to a printer.

Note: compare this approach to a case when client chooses a specific printer – since clients do not synchronize, they may all choose the same printer, which will cause big delays.

When clients submits a job, print queue will check if there is an available printer in the system, if there is no such printer, the job is placed into a queue ordered by the job priority (LOW,MEDIUM,HIGH) – where higher priority places job closer to the top of the queue so that it will be executed earlier. Jobs with the same priority a ordered by their submission timestamps – if job 1 was submitted before job2, then job 1 will be assigned to a printer before job2 (given job and job 2 have the same priority). In the case there is an available printer, the job is assigned to that printer without been put into a queue (notice that if there is an available printer, then the job queue HAS to be empty, the opposite is not true).

To know which printer is available print queue requires printers to report back, which is similar to a producer/consumer architecture, where printers (consumers) ask for the next job right after they are done with the previous one. When printer reports back, printer queue will check if there are any jobs in the queue. In the case print queue is NOT empty, the top-most job is popped out of the queue and assigned to the printer. If the queue is empty printer becomes idle (available).

Command pattern

We'll implement this logic using Command pattern, which is also called “delayed callback”. A short description – delayed callback is applicable when client knows exactly WHO to call and with WHAT arguments, but doesn't know WHEN.
In C++ it's implemented as follows:
there is an abstract TimedCommand class which declares a virtual Execute method.
client subclasses (derives) from the TimedCommand a ConcreteCommand, which has a pointer to an object (of ANY TYPE) and a pointer to a method (ANY method, ANY signature). Execute method is implemented to call the above method on the above object.
When client decides that a delayed callback is required, s/he will create a ConcreteCommand object, initializes pointers to object and method and passes it to an entity that knows WHEN to call.
And basically that's it!!!

See my example Announce and Repeater, understand how they work. You may compile it using “make example”.

Notice that the entity that calls Execute needs to know TimedCommand only (it doesn't need to know about neither ConcreteCommand, nor the type of the object pointer, nor method pointer).

The Command pattern lets Simulator pass the requests (events) of unspecified objects (driver, class Announce, and class Repeater) by turning the request itself into an object. This object can be stored and passed around like other objects. The key to this pattern is an abstract TimedCommand class, which declares an interface for executing operations. In the simplest form this interface includes an abstract Execute operation. Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing Execute to invoke the request. The receiver has the knowledge required to carry out the request.

Event-driven simulation

Now about overall design of the program – you will implement an **event-driven simulation**. Event-driven simulation is the one that only simulates time points when something interesting is happening. In other words it only active when there are changes in the system.

The following description is based on
http://www.cs.utoronto.ca/~heap/270F02/node54.html
If events aren't guaranteed to occur at regular intervals, and we don't have a good bound on the time step (it shouldn't be so small as to make the simulation run too long, nor so large as to make the number of events unmanageable – that is when something happens at time 0.15, while timestep is 0.1, now we'll see that event has happened only at time 0.2 and we have to deal with changes that the event had caused in the past 0.05 seconds), then it's more appropriate to use an event-driven simulation. A typical example might be simulating a lineup at a bank, where customers don't arrive at regular time intervals, and may be deterred by a long lineup (or a printer queue!).

This approach uses a list of events that occur at various time, and handles them in order of increasing time. Handling an event MAY alter the list of later events (doesn't happen in print queue). The simulation makes time ``jump" to the time of the next event.

How do we stop? Again, we can stop when time reaches a certain point, or when the system reaches a certain state, usually when there are no event in the event queue. Here is a generic event-driven algorithm:

1. Initialize system state
2. Initialize event list
3. While (simulation not finished)
    1. Collect statistics from current state
    2. Remove first event from list, handle it
    3. Set time to the time of this event.

How is the list of events managed? It should be ordered by increasing time. We don't generate all the events in the list at the beginning (this would be analogous to knowing the entire sequence of states of the simulation at the outset). Instead we initialize the simulation with certain events, with their associated times (kickstart the system). Certain events may cause more events events, which are inserted at the appropriate place in the event list.

Print queue events are

1) new job arrives – these events are sent by the driver

2) job completed - this event will be created by the print queue when the job is assigned to a printer. Print queue will estimate the completion time as <job size> / <printer speed> and will insert an event that will be fired at NOW() + <job size> / <printer speed> (see Repeater class)

To submit:

`print_queue.h`, `print_queue.cpp` – no implementations in the header file, except the `less<>` which I provided.