Terminology

A queueing system has

- *servers*: these perform the service
- *buffer* : We define "buffer" as the place where requests **wait** before their service starts.

Please note this buffer has no relationship to the data structure you use for storing requests in your queueing system. You may choose to define a structure called buffer to hold both waiting requests and requests in service. OR you may define a structure called buffer that holds only waiting requests, and hold requests in service somewhere else.
Your implementation is completely orthogonal to the above definition. **Please do not confuse the two.**
Events: An event is the 'thing that happens'. It is the thing that changes the state of the system. Do not confuse it with the actual entities in the system. A server is an actual entity in the system. A request is an actual entity in the system. A request does not "happen". An arrival of a request happens. The arrival is an event. A server is an actual entity in the system. A server does not "happen". A service finishing its current service is what "happens" (which in our system is equivalent to a request departure event).

The buffer is a queue of requests. It is like the waiting line in front of a railway ticket counter. An event list is a priority queue of future events - i.e. events that *have not yet "happened"* . An event list does not exist in a real system - it only exists in the simulator of a system. But a waiting line does exist in the actual system. **Do not confuse these two "queues".**

More Definitions:
90%ile of any random variable X. It is that value X_{90} such that:

$P[X <= X_{90}] = 0.9$

Now, how do we define 90%ile for a set of N values? Note that in a sorted list of 100 values, X[0], X[1],...X[99], the value at X[89] is the 90 %ile, because 90 elements - that is 90% of elements in this case have values less or equal to X[89]. That is, the "90th smallest" element is the 90 percentile.
If N=200, the 180th smallest element would be the 90 %ile. And so on.

So let us define the value we want, that is, the 90 %ile as the $(\text{ceiling}(0.9*N))^{th}$ smallest element

So, say we had 25 values. Counting ordered positions from 1 onwards, then we want the ceiling $(0.9*25)^{th}$ position element. That is we want the element at position 23. That is if it was an array X, we want X[22]. Then 92% elements are <= X[22] and 8% are >= than X[22].

Problem Statement

Problem Statement may undergo changes as it still evolving. Please be alert to updates and check back frequently.
Write a simulator for a multiple server queuing system, with servers that change speed dynamically (for power saving).

Inputs (taken from stdin):

-Number of servers (s)
-Maximum number of requests that are waiting in the system not including those in service (w). (New requests arriving when number of requests in the system (waiting+being served) is greater than w+s are dropped).
-Min speed, Max speed that the server can operate at (in MIPS - million instructions per second)
-step size S (in mips)
- probe interval P (in seconds). The job queue size is checked every P seconds and speed change action may be taken (see below)
-Job Queue size thresholds: Jlow, JHigh. At a probe, whenever job queue length (waiting requests only) is seen above threshold JHigh, speed is increased by S (upto max). Whenever job queue length is seen below threshold Jlow, speed is decreased by S (upto min).
-A "trace" of request arrivals with the following information:
request_arrival_time request_processing_size_in_million_instructions
- "-1" to terminate above trace

Trace Outputs:

- A request trace file (request.log) with the following format:
Request _# arrival_time processing_start_time processing_end_time waiting_time service_time
- A server trace file (server.log) with the following format
probe_interval_# job_q_size_at_probe speed_before_probe speed_after_probe
-An event trace written to the screen: where a line of output is generated just after an event has been removed from the event list as the next event to be processed. At this point, the following line should be written out:
sim_time event_being_processed next_waiting_req total_num_waiting_in_buffer next_event_type

Here, event_being_processed will be : ARRIVAL/DEPARTURE/PROBE
next_req_in_buffer should be shown as: (A1,S1) where (A1, S1) is the first request in the buffer queue with arrival time A1 and service requirement of S1.
next_event_type will also be ARRIVAL/DEPARTURE/PROBE

Summarized outputs (write to screen):

Number of requests arrived (including dropped):
Number of requests serviced:
Number of requests dropped:
Average waiting time:
Average service time:
90%ile of waiting time:
Maximum waiting time:
Max queue length reached: (where "queue length" is requests in wait queue only. Not including in service)
Average operating speed of the server:
Let Si, i=1,2,3...k be the operating speeds of the server

Let Ti be the time for which configured operating speed of the server was Si.
Then average = sum_i {Si * Ti} / sum_i {T_i}
(here \sum_i {T_i} will be the total simulation time}
90%ile of speed:

Bonus Points for:
Simulating request timeouts (sample input for this TBD)
Average Server Utilization
Average Job queue length

Sample Input
Sample Output (Format only, no values are given)
Sample code upload TBD

Submission requirements

1. Submit one version of the simulator that **only adds the multiple servers feature.** You must use STL priority queue for event list and STL queue for buffer. Input/Output file can be the same - just skip over the unnecessary inputs (use only the first speed as the speed of the server), and do not print out the speed related metrics
2. Submit a second version which implements everything as specified.
    a. In this version, along with the code you must submit a presentation (in pdf format) which outlines the class design, the data structures used, and the design decisions taken for efficiency. Detailed instructions regarding this will be posted later.

Evaluation Criteria

This assignment will be judged very strongly for software design. Design & code will be judged for

- Efficiency - in space and time (choice of data structures and algorithms. Other optimizations.).
    - Specific requirement: The implementation of event list should be as a heap with “locator” capability. That is, there must be a way to directly access a specific item and change its priority value (timestamp, in this case). This change may change relative locations of other entries in the heap - your implementation must take care of this issue. “Search” based data structures are semantically incorrect and will face marks penalties.
- Maintainability, extendability, elegance
    - File organization, modularity, encapsulation
    - Will design choices not “scale” with added simulation features?
    - Scalability will be judged both on coding ease vs time/space scalability
- Readability of code
    - Comments, variable names, indentation
- Any other features you wish to “show-off”

Note that no one design is perfect and there are trade-offs. Your presentation should describe the thinking that went into your choice (E.g. “I chose this design because it is efficient, although less maintainable/extendable”, or vice-versa)

Some Do's and Don'ts About Simulation

The basic simulation design should **always** be as follows:

simulation main:

initialize simulation {
  set time=0,
  schedule initial events
    (that means create Event objects, timestamp them, add any necessary information about which entities in the actual system these events might be about.
     Then throw the event into the event list)
}

while (eventlist not empty) {

    remove the nearest event (pop from priority queue, key being event timestamp, priority queue is required because events are not added into the queue in the order that they may
    happen. You may add an event e1 scheduled at time t+s, before you add an event e2 scheduled at time t.)
    advance simulation clock to this event time
    handle event based on event type (i.e. call event handler).
        (This will usually be a switch that picks the event handler based on the event type. For advanced simulations, event handlers will need arguments. This couldbe information that was retrieved from the event object itself. Mainly, simulations of complex systems need information about **which** object of the system is the event about. E.g. in multiple server simulation, if you get a departure
        event, you need to know which server the departure is from. Or which request departing - server ID could be in the request)

}

event handlers should have logic like this:

    update system state - whatever needs to be done to the state, do it.
    Based on new system state, new events may get created.
        Create them, add them into eventlist.
    Collect statistics.

- Do not distribute events in multiple data structures. Consolidate events into **one** event list
- Think about the best way of attaching information to an event, that helps you process it.

ADDED PORTION STARTS HERE:

Important point to note; it is OKAY TO USE WEISS TEXTBOOK CODE AS-IS TO IMPLEMENT THE BASIC HEAP

DESIGN SUGGESTION FOR DYNAMIC CPU SPEED FEATURE

(This is basically the "indexedHeap" implementation of Sedgewick, proposed for the exact reason we want it).

The "locatorHeap" class will have the following public methods

- idType insert(objectT o): here idType can be int or long
- deleteMin as usual
- changeKey(idType id, newKey): this will change the key value of the object whose ID in the heap was "id" to newKey
- remove(idType id): this will remove the object with this id (not needed unless you implement request timeout during service - then you have to *remove* it's departure event)

Private methods:

- increaseKey(position p, delta): increase key of object at index p in the heap vector by delta.
- decreaseKey(position p, delta): decrease k of object at index p in the heap vector by delta.
- percolate Up/Down etc

Data members of locatorHeap:

- Apart from the basic heap vector, there is an additional vector of size NMAX (call it "positions". This keeps the position of the object whose ID is "i" at the index i.
- That is if position[100]=18, that means that object (here it will be event), of ID 100 is at index 18 in the heap vector. So changekey(100, 4.3567) will result in a call to either decreaseKey(18) or increaseKey(18), depending on the change. This is an O(1) access to the element.

Note that insert, deleteMin, percolate Up/Down will all need to manage updates to the position vector for various elements (any element whose position in the heap changes, should have its position vector updated). Thus, the ID of the element should be stored WITH each element in the heap. It would seem that only O(log n) elements would have their positions changed (those affected by the percolate up/down. This should be a straightforward change in the code - carefully find the points in the percolate code that are moving elements in the heap code.).

How will the simulation use this? For any event that is potentially "reschedulable" (word credit: Guna) you should keep the ID that now the locatorHeap will return, with the concerned entity. So e.g., the request should have an Event ID member, which will correspond to the departure event ID, when its service begins. Every time there is a probe and the departure changes, you just call "changeKey" on this ID.

- But NMAX is the limit on ID range and array of size NMAX is statically allocated!! This is a severe limitation. Perhaps IDs can be reused? (Once an object with a certain ID is removed, that ID can be reused)

<u>One idea extension (this is mine, so it is tentative and may have flaws)</u>

Instead of a raw positions array, we should use a hashtable to keep the (ID, position) pair. The hash will be on the key ID, and we call a find on that ID, and get the "position", and use it for the incr/decrkey operation.

This way we just use the STL unordered_multimap again, and let it manage the table size etc. The ID value will be limited not by table size, but the max value of the ID type (e.g. max of long int). With every insert, we keep incrementing the ID...if we reach max, we can even query the hashtable in O(1) and see if wraparound is possible.

Assertion Checks to do with your output

Acknowledgements: Many students already did these with their own code and with reference code and this helped remove ref code bugs. (Esp Kandarp, and one post by Rahul Singhal)

This has been posted on piazza, but put here for your convenience:
Some "assertions" that you should check (and our scripts will, in your evaluation):

- Number of requests arrived = number serviced + number dropped
- Average operating speeds should be between min and max
- Max queue length reached <= size of wait buffer
- 90%ile of speed is between min and max
- Sim time clock should always be advancing
- Number of lines (entries) in request.log (you can find by running "wc request.log") should be = number of requests serviced
- Number of requests serviced/sim time is <= (num of servers/avg_service_time)

Some trends ((all else being the same of course):

- As speed increases, number of requests serviced should increase, waiting time should decrease
- As buffer size decreases, number of requests dropped should increase, but waiting time should decrease
- As number of servers increase, number of requests serviced should increase, waiting time should decrease
- Avg Service Time should decrease with increasing average speed