Warmup #2

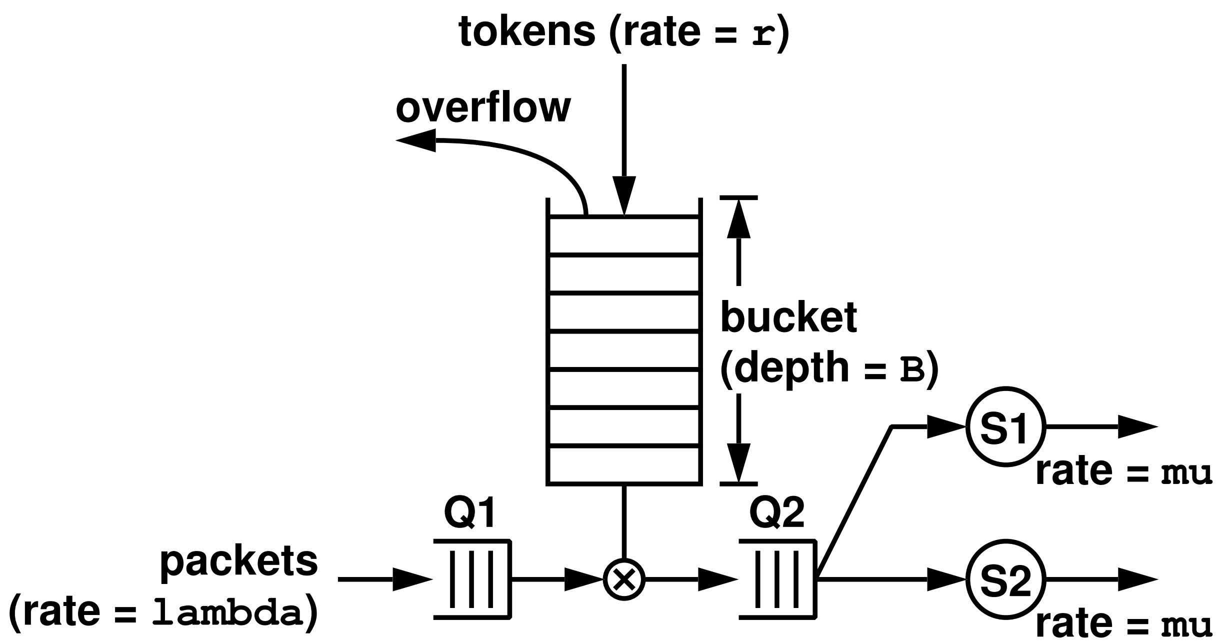
Bill Cheng

http://merlot.usc.edu/cs402-f18

Multi-threading Exercise

- Make sure you are familiar with the *pthreads* library
  - Ch 2 of textbook - threads, signals
    - additional resource is a book by Nichols, Buttlar, and Farrell *"Pthreads Programming"*, O'Rielly & Associates, 1996
  - you must learn how to use pthreads *mutex* and *condition variables* correctly
    - `pthread_mutex_lock()/pthread_mutex_unlock()`
    - `pthread_cond_wait()/pthread_cond_signal()/ pthread_cond_broadcast()`
  - you must learn how to handle UNIX *signals* (<Cntrl+C>)
    - `pthread_sigmask()/sigwait()`
    - `pthread_kill()/pthread_cancel()`
  - you need to learn how to perform *cancellation* in pthreads
    - `pthread_setcancelstate()`
    - `pthread_setcanceltype()`
    - `pthread_testcancel()`

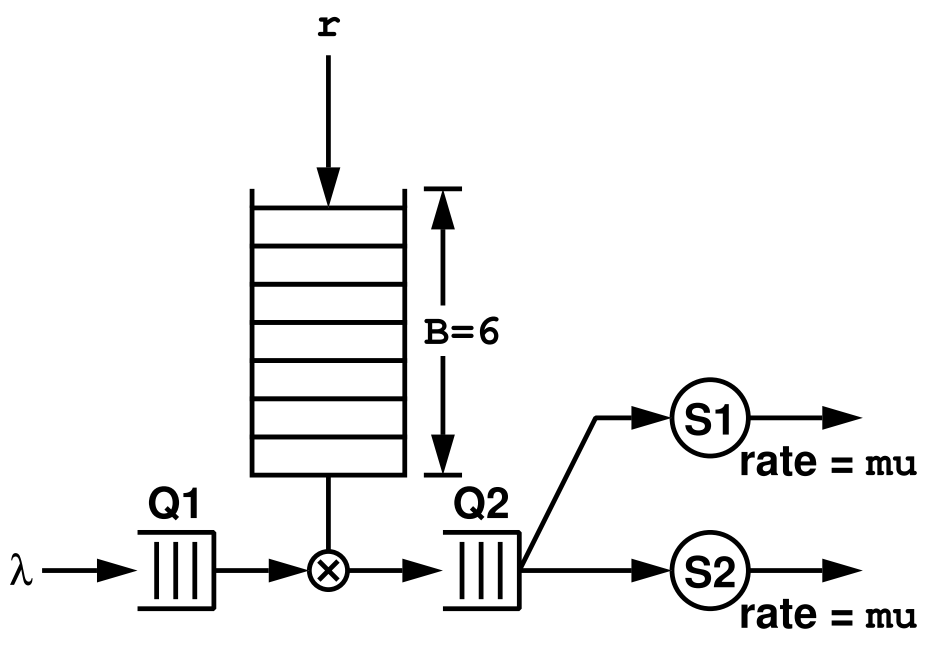
Token Bucket Filter

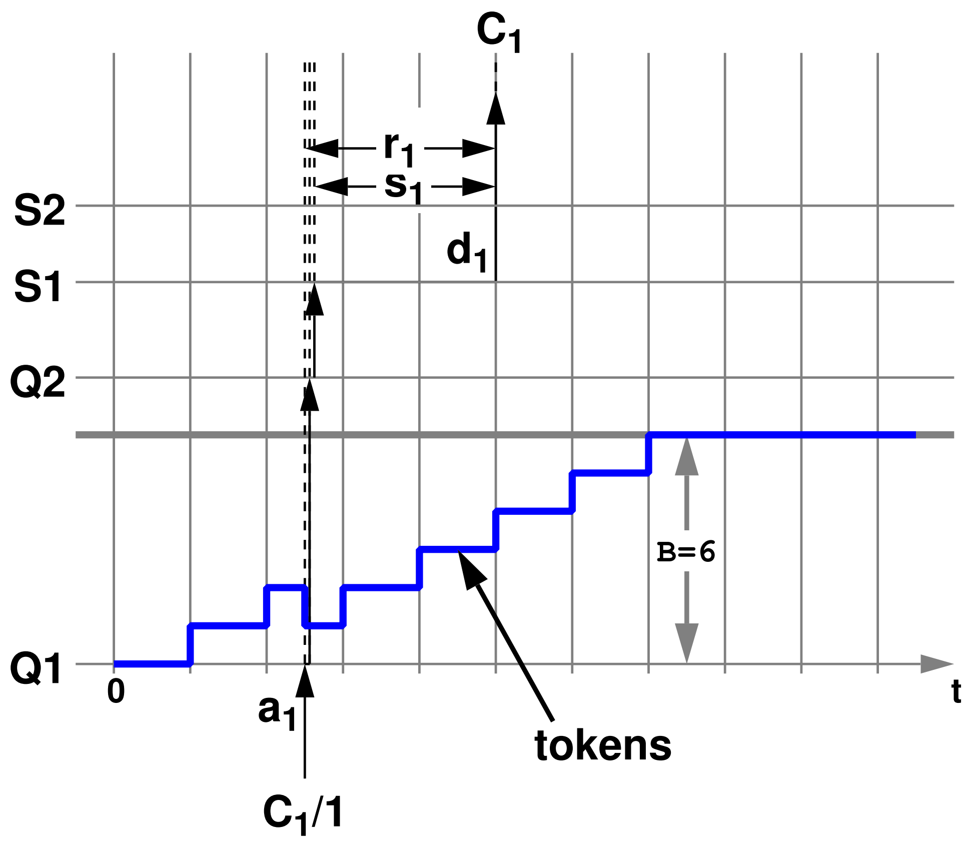


- **Ex:**
  - ticket scalper?!
  - traffic controller/shaper

Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response (system) time
- q_i : queueing/waiting time

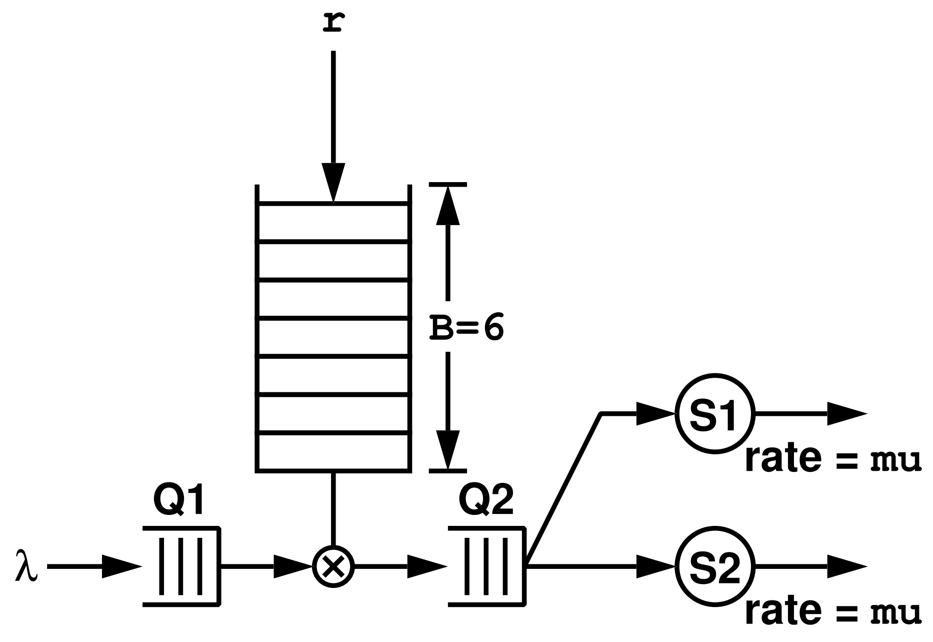


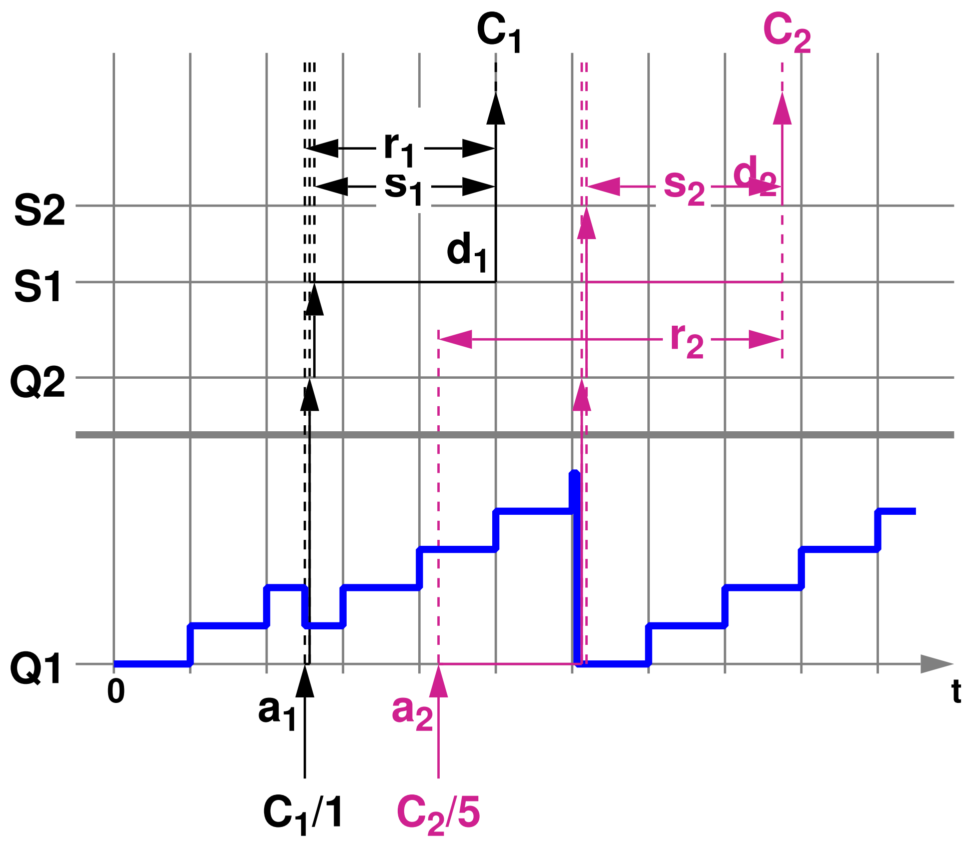


- $r_1 = d_1 - a_1$

Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response (system) time
- q_i : queueing/waiting time

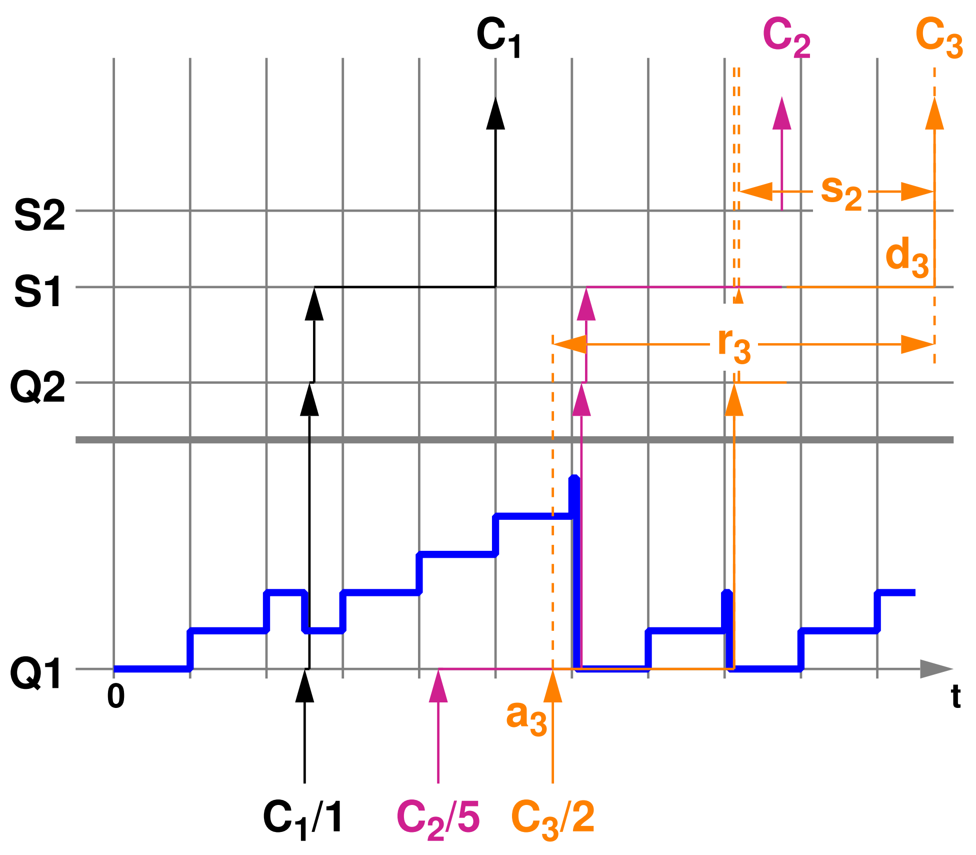


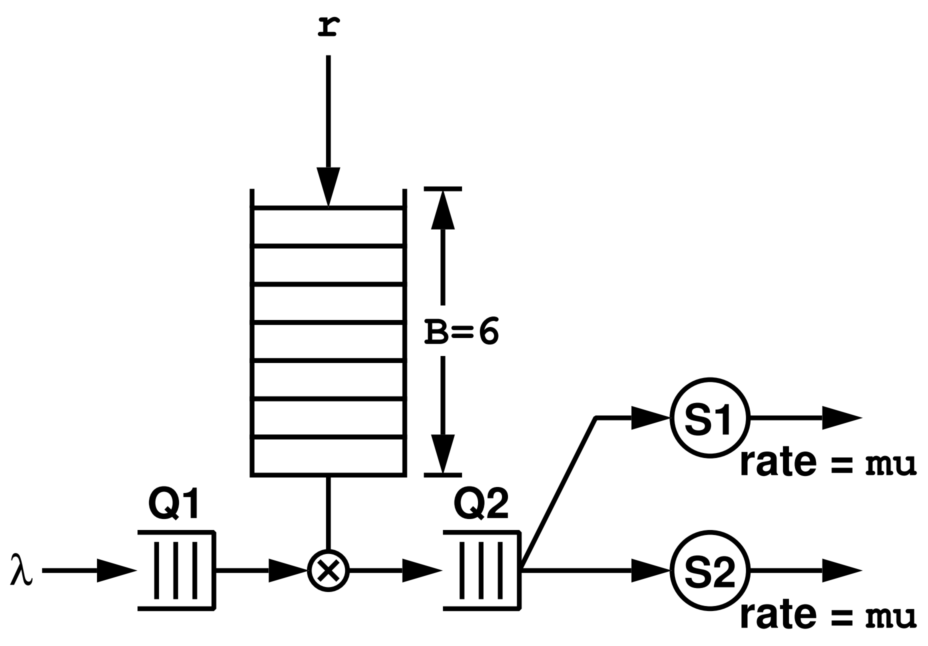


- $r_2 = d_2 - a_2$

Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response (system) time
- q_i : queueing/waiting time



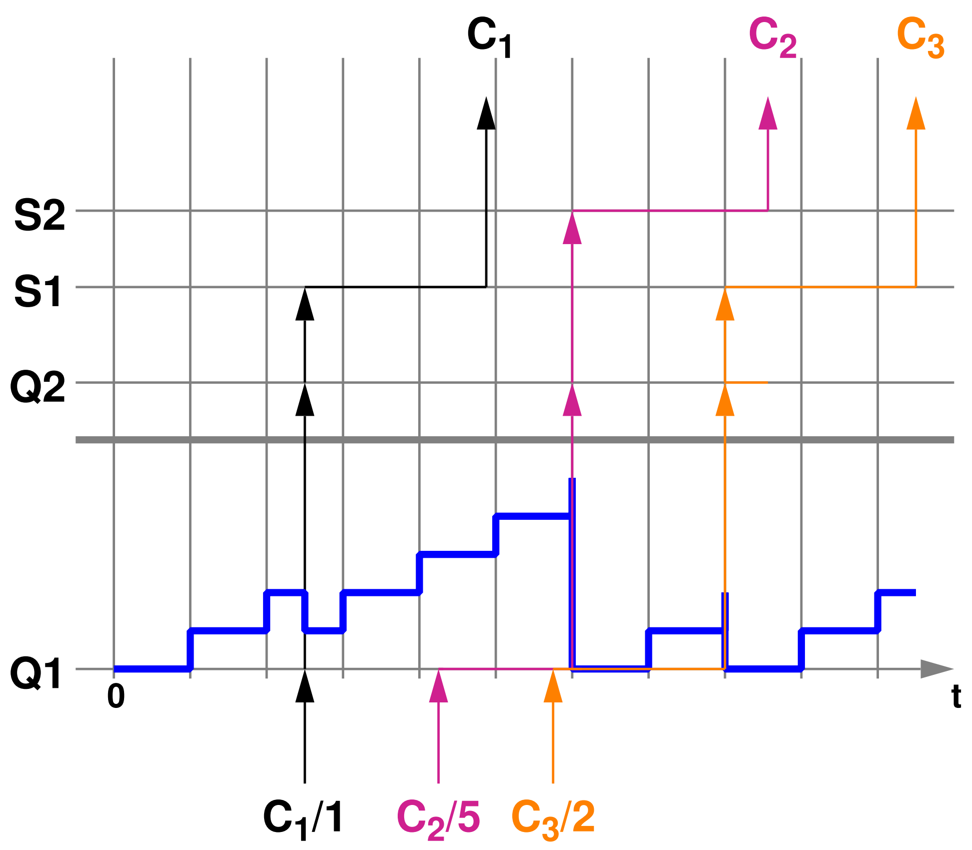


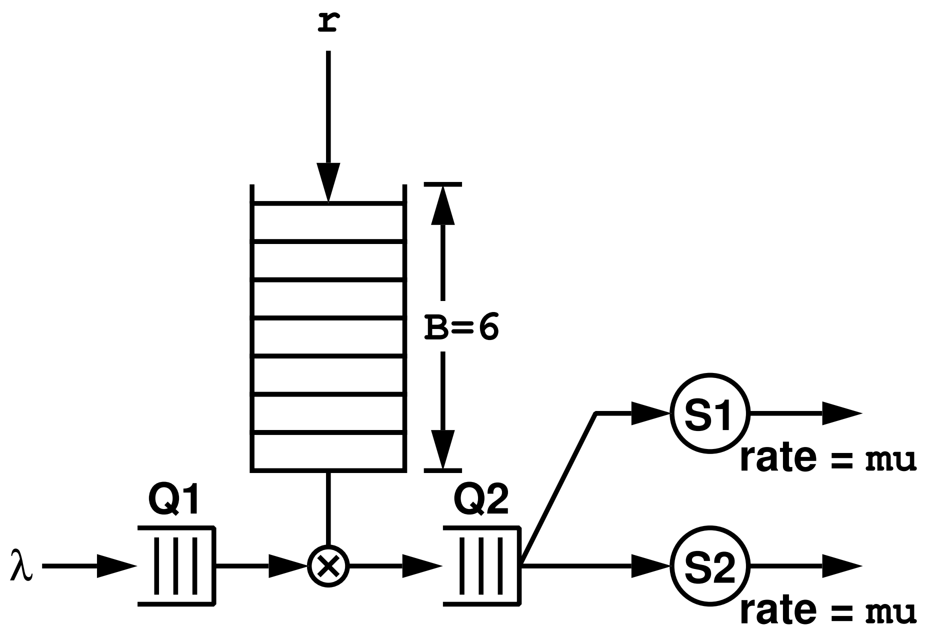
- $r_3 = d_3 - a_3$

Event Driven Simulation

- An *event queue* is a sorted list of events according to timestamps; smallest timestamp at the head of queue
- *Object oriented:* every object has a "next event" (what it will do next if there is no interference), this event is inserted into the event queue
- Execution: remove an event from the head of queue, "execute" the event (notify the corresponding object so it can insert the next event)
- Insert into the event queue according to timestamp of a new event; insertion may cause additional events to be deleted or inserted
- Potentially repeatable runs (if the same seed is used to initialize random number generator)

Event Driven Simulation (Cont...)





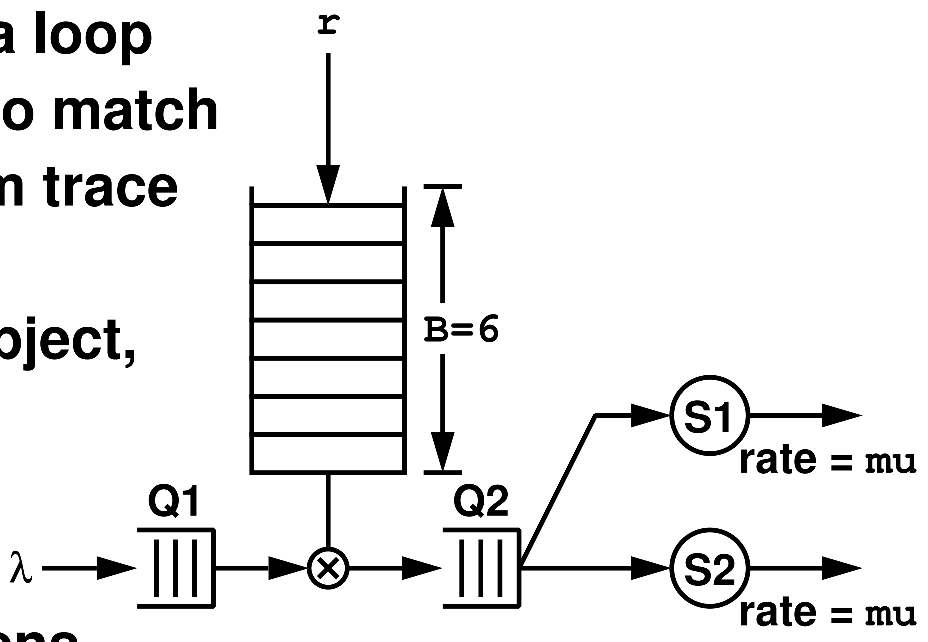
▬ $r_3 = d_3 - a_3$

Time Driven Simulation

- Every active object is a thread
- To execute a job for x msec, the thread sleeps for x msec
  - nunki.usc.edu does not run a realtime OS
  - it may not get woken up more than x msec later, and sometimes, *a lot more* than x msec later
    - you need to decide if the extra delay is reasonable or it is due to a bug in your code
- Let your machine decide which thread to run next (irreproducible results)
- Compete for resources (such as Q1), must use mutex

Time Driven Simulation (Cont...)

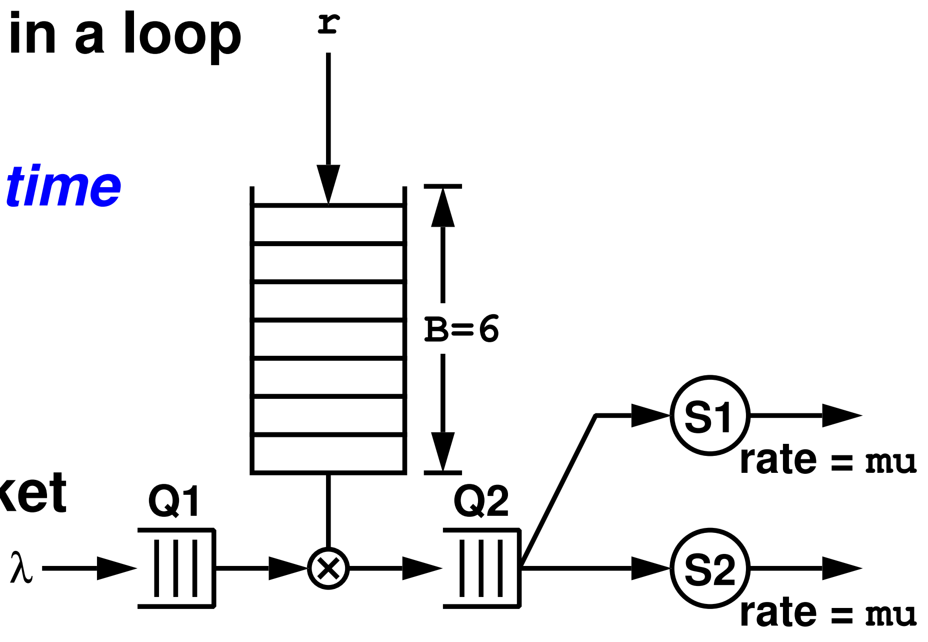
- You will need to implement 4 threads (or 1 main thread and 4 child threads)
  - the *packet arrival thread* sits in a loop
    - sleeps for an interval, trying to match a given *inter-arrival time* (from trace or deterministic)
    - wakes up, creates a packet object, locks mutex
    - enqueues the packet to Q1
    - moves the first packet in Q1 to Q2 if there are enough tokens
    - if Q2 was empty before, need to *signal* or *broadcast* a *queue-not-empty condition*
    - unlocks mutex
    - goes back to sleep for the "right" amount



- See skeleton code in warmup2 FAQ

Time Driven Simulation (Cont...)

- You will need to implement 4 threads (or 1 main thread and 4 child threads)
  - the *token depositing thread* sits in a loop
    - sleeps for an interval, trying to match a given *inter-arrival time* for tokens
    - wakes up, locks mutex, try to increment token count
    - check if it can move first packet from Q1 to Q2
    - if packet is added to Q2 and Q2 was empty before, *signal* or *broadcast* a *queue-not-empty condition*
    - unlocks mutex
    - goes back to sleep for the "right" amount



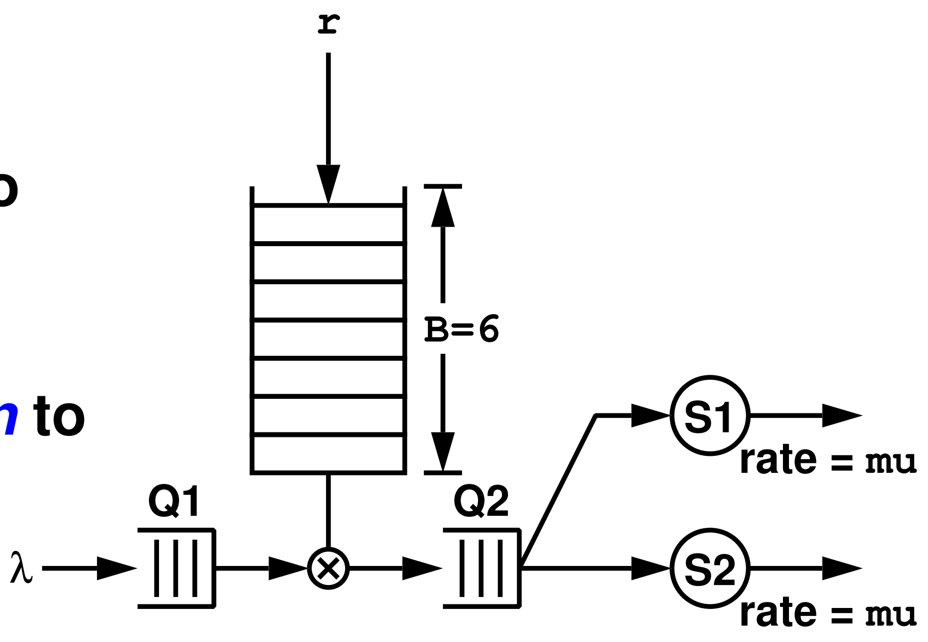
- See skeleton code in warmup2 FAQ

Time Driven Simulation (Cont...)

- You will need to implement 4 threads (or 1 main thread and 4 child threads)
  - each *server thread* sits in a loop

wait for work
    - lock mutex
    - stay in a loop to wait for Q2 to become non-empty
      - if empty, *wait* for the *queue-not-empty condition* to be *signaled/broadcasted*
    - when Q2 is not empty, dequeu a packet
    - unlock mutex

work
    - sleeps for an interval matching the *service time* of the packet
    - eject the packet from the system
    - loop



- See skeleton code in warmup2 FAQ

Time Driven Simulation (Cont...)

- Dropped packets
  - if the token requirement for an arriving packet is too large, drop the packet
- Dropped tokens
  - if an arriving token finds a full bucket, it is dropped
- Other requirements
  - please read the spec!

Program Output

- Program output must look like what's in the spec
  - you must *NOT* wait for emulation to end to print all these

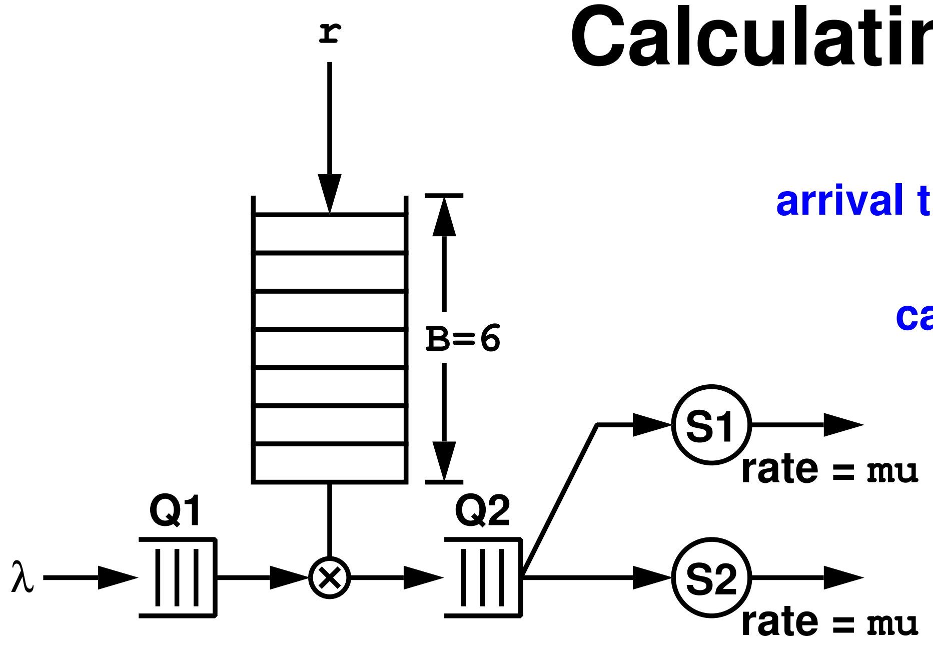
```
Emulation Parameters:
    number to arrive = 20
    lambda = 2              (if -t is not specified)
    mu = 0.35               (if -t is not specified)
    r = 4
    B = 10
    P = 3                   (if -t is not specified)
    tsfile = FILENAME           (if -t is specified)

00000000.000ms: emulation begins
00000251.726ms: token t1 arrives, token bucket now has 1 token
00000502.031ms: token t2 arrives, token bucket now has 2 tokens
00000503.112ms: p1 arrives, needs 3 tokens, inter-arrival time = 503.112ms
00000503.376ms: p1 enters Q1
00000751.148ms: token t3 arrives, token bucket now has 3 tokens
00000751.186ms: p1 leaves Q1, time in Q1 = 247.810ms, token bucket now has 0 token
00000752.716ms: p1 enters Q2
00000752.932ms: p1 leaves Q2, time in Q2 = 0.216ms
00000752.982ms: p1 begins service at S1, requesting 2850ms of service
00001004.271ms: p2 arrives, needs 3 tokens, inter-arrival time = 501.159ms
00001004.526ms: p2 enters Q1
00001007.615ms: token t4 arrives, token bucket now has 1 token
00001251.259ms: token t5 arrives, token bucket now has 2 tokens
00001505.986ms: p3 arrives, needs 3 tokens, inter-arrival time = 501.715ms
00001506.713ms: p3 enters Q1
00001507.552ms: token t6 arrives, token bucket now has 3 tokens
00001508.281ms: p2 leaves Q1, time in Q1 = 503.755ms, token bucket now has 0 token
00001508.761ms: p2 enters Q2
00001508.874ms: p2 leaves Q2, time in Q2 = 0.113ms
00001508.895ms: p2 begins service at S2, requesting 1900ms of service
...
```

Program Output

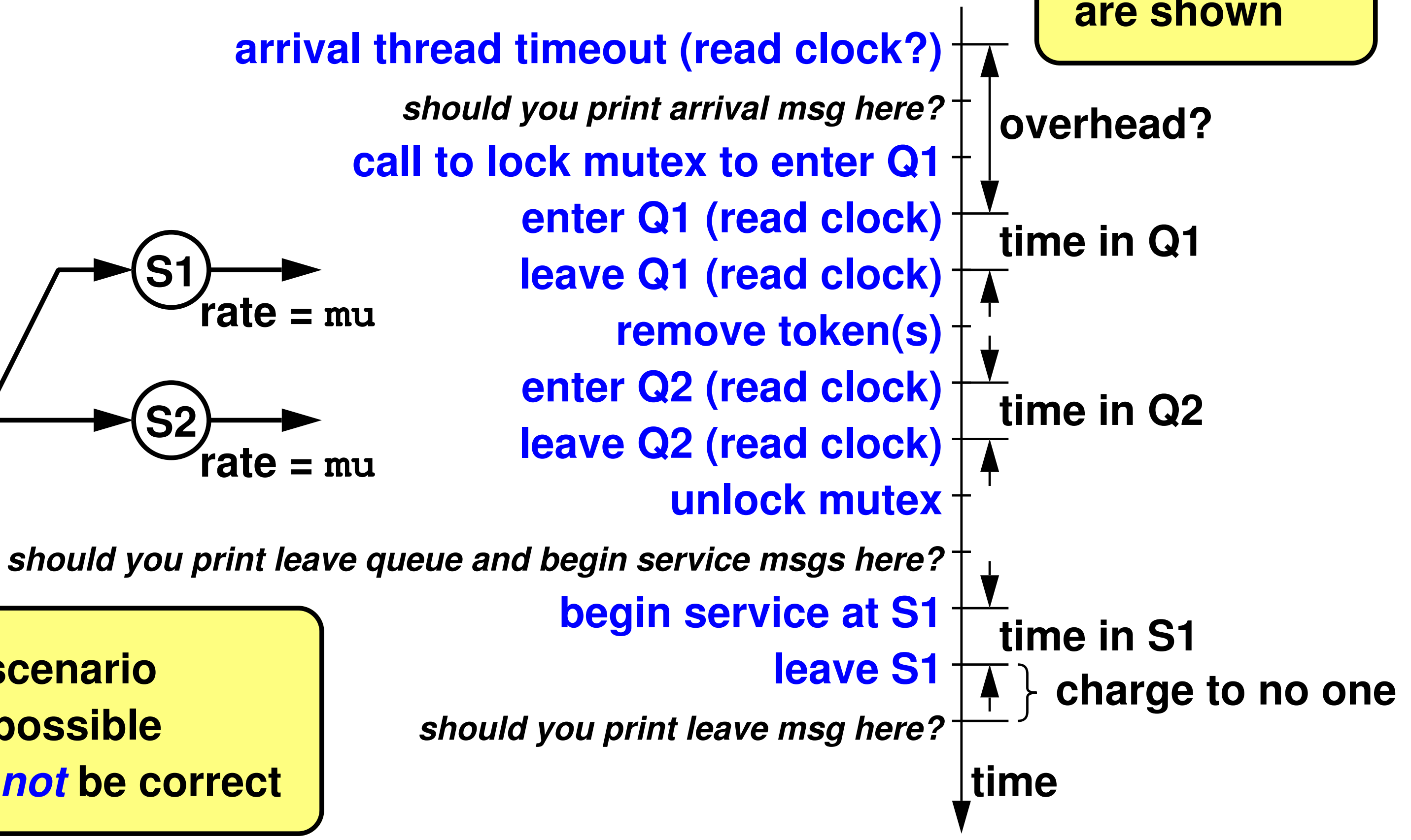
- Program output must look like what's in the spec
  - you must ***NOT*** wait for emulation to end to print all these

```
...
00003427.557ms: p2 departs from S2, service time = 1918.662ms, time in system = 2423.286ms
00003612.843ms: p1 departs from S1, service time = 2859.861ms, time in system = 3109.731ms
...
????????.???ms: p20 departs from S?, service time = ???.???ms, time in system =
???.???ms
????????.???ms: emulation ends

Statistics:

    average packet inter-arrival time = <real-value>
    average packet service time = <real-value>

    average number of packets in Q1 = <real-value>
    average number of packets in Q2 = <real-value>
    average number of packets at S1 = <real-value>
    average number of packets at S2 = <real-value>

    average time a packet spent in system = <real-value>
    standard deviation for time spent in system = <real-value>

    token drop probability = <real-value>
    packet drop probability = <real-value>
```

Calculating Statistics



NOTE: not all activities are shown



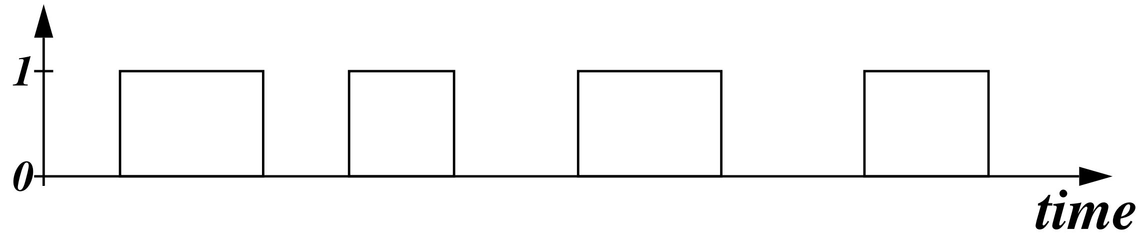
NOTE: this is just one scenario
- other scenarios are possible
- events on right may *not* be correct

- time between begin service and leave server is the amount of time in `select()` or `usleep()`

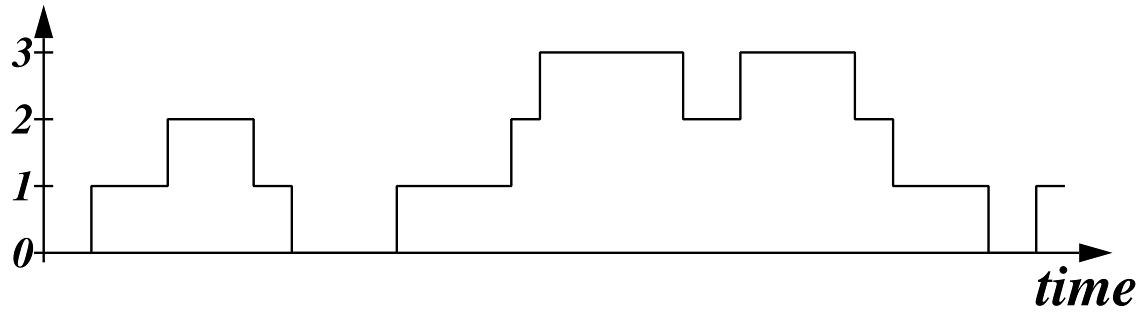
Some packets needs to be excluded from certain statistics
- e.g., if a packet is dropped, it should not participate in the time-in-system statistics

Mean and Standard Deviation

- Average time
  - for *n* samples, add up all the time and divide by *n*
- Average number of packets at a server
  - same a fraction of time the server is busy



- Average number of packets at Q1



- Standard deviation is the squareroot of variance
  - $Var[X] = E[X^2] - (E[X])^2$
    - must use *population variance* equation

SIGINT

- <Cntrl+C>
  - you need to *termiante* your emulation *gracefully* in case the user presses <Cntrl+C>
    - the OS will *delivery a SIGINT Unix signal to your process*
    - you need to *catch the signal*
    - you need to let all your threads know that it’s "time to quit"
    - you need to *wait* for all your threads to terminate
    - then you can print statistics and quit
- We will cover this in lecture next week
  - will discuss this at the beginning of the next discussion section