Warmup #2
(Part 1)

Bill Cheng

http://merlot.usc.edu/william/usc/

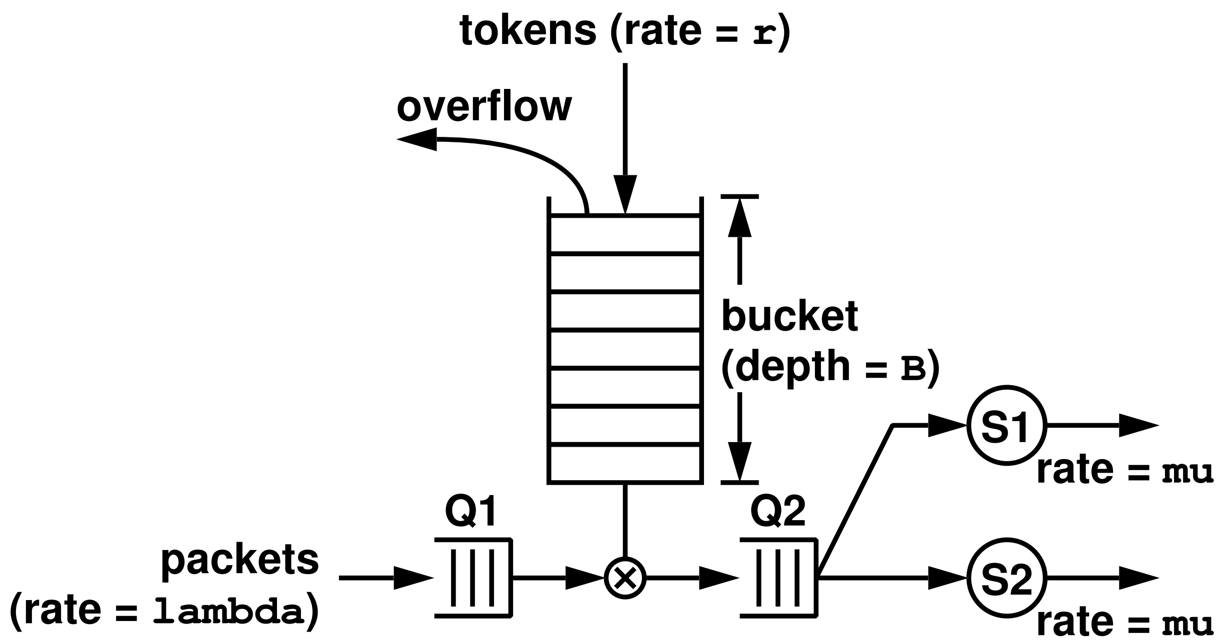
Multi-threading Exercise

- 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_broadcast()`
      - do *not* use `pthread_cond_signal()` for warmup2

next week

  - you must learn how to handle UNIX *signals* (<Ctrl+C>)
    - `sigprocmask()/sigwait()`
    - `pthread_cancel()`
  - you may want to learn how to disable/enable *cancellation* in pthreads
    - `pthread_setcancelstate()`

Token Bucket Filter



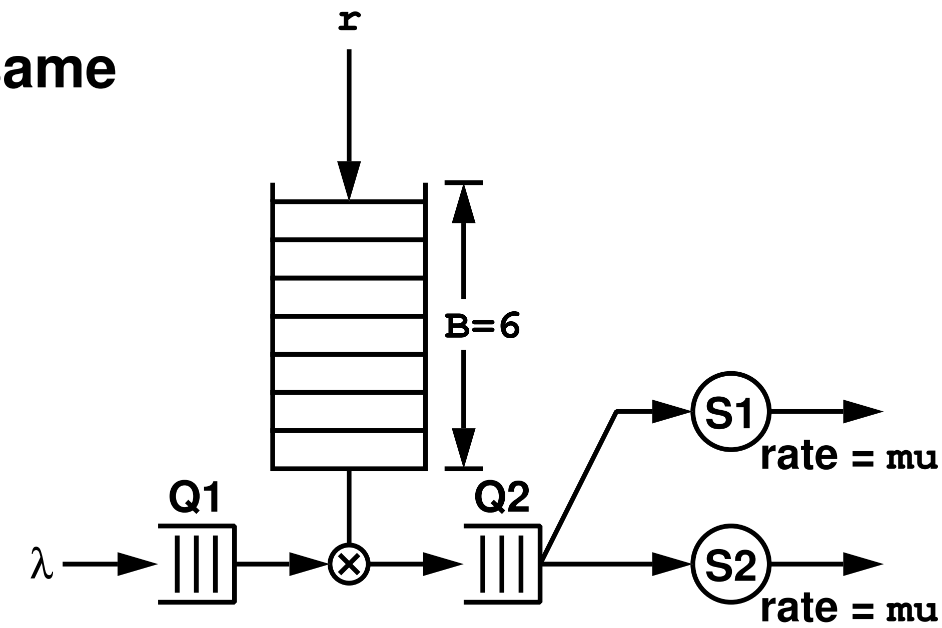
- **Ex:**
  - **traffic controller/shaper**
- **Your job is to implement 4 *cooperating child threads* to move the packets along by following rules described in the spec**
  - **the main thread creates these threads, join with them, then print statistics**

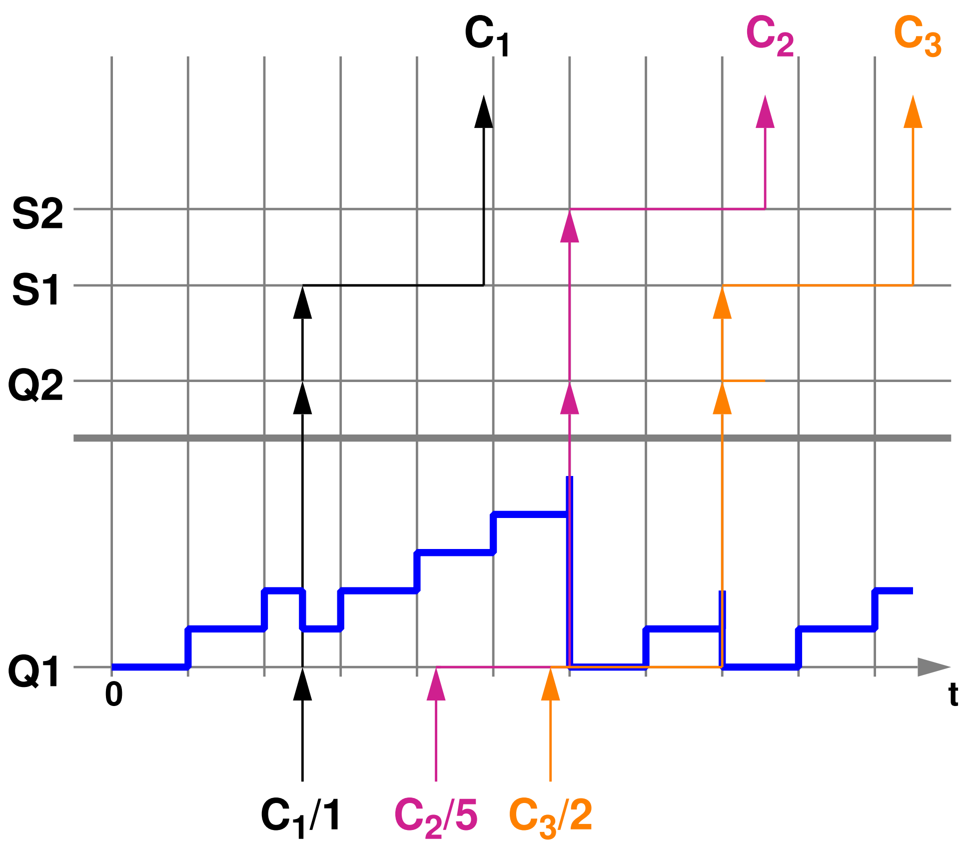
We Are *Not* Doing Event-driven Simulation

- An *event queue* is a sorted list of events according to timestamps; smallest timestamp at the head of queue
  - event has zero duration (events can happen at the same time)
- *Object oriented:* every object has a "next event" (what and when it will do next), 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)
- The simulator never "sleeps"; it tries to run as fast as it can to finish the simulation as quickly as possible

We Are *Not* Doing Event-driven Simulation

- Multiple event can happen at the same time in an *event-driven simulation*
  - we will *not* be doing that!



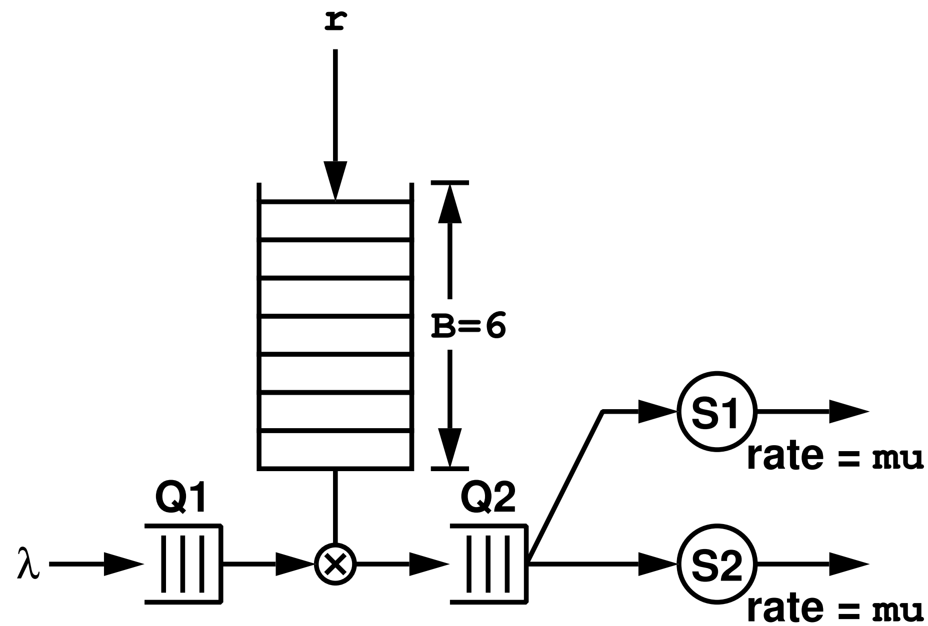


- $r_3 = d_3 - a_3$

"Time Driven" Simulation

- We will use the words "simulation" and "emulation" interchangeably
- No "event queue"
  - every *active object* is implemented as a *thread*
  - threads interacting with one another through the use of *shared variables*
    - how else can threads "talk" to each other?!
- It takes time to execute simulation code
  - the time it takes to do all that is part of the simulation
  - to simulation the passing of time, call `usleep()`
    - e.g., if doing something takes `x` usec, call `usleep(x)`
    - Ubuntu does not run a "realtime" OS, it's "best effort"
    - `usleep(x)` will return more than `x` usec later
      - and sometimes, *a lot more* than `x` usec later
      - you need to decide if the extra delay is reasonable or it's due to a bug in your code

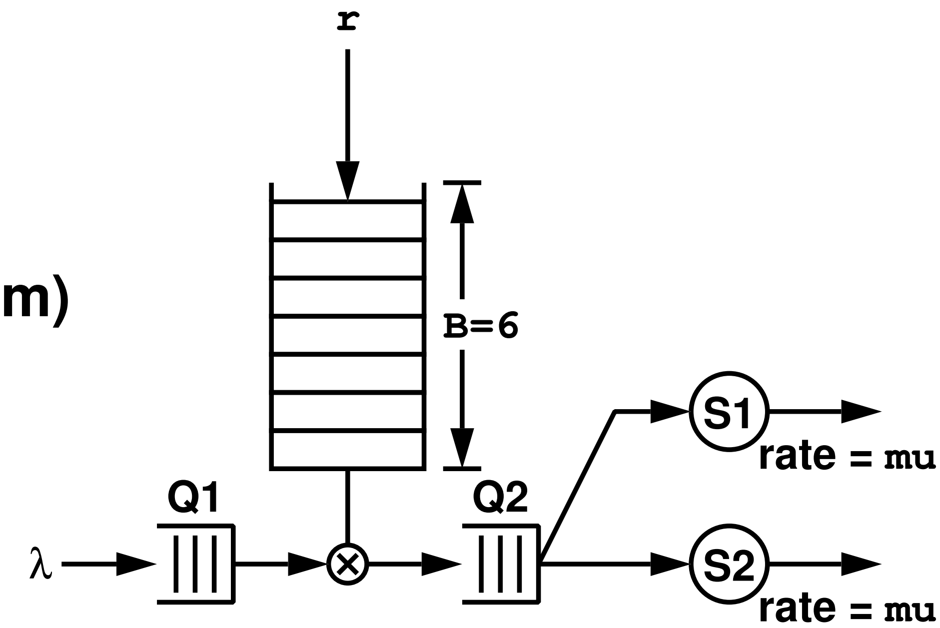
"Time Driven" Simulation

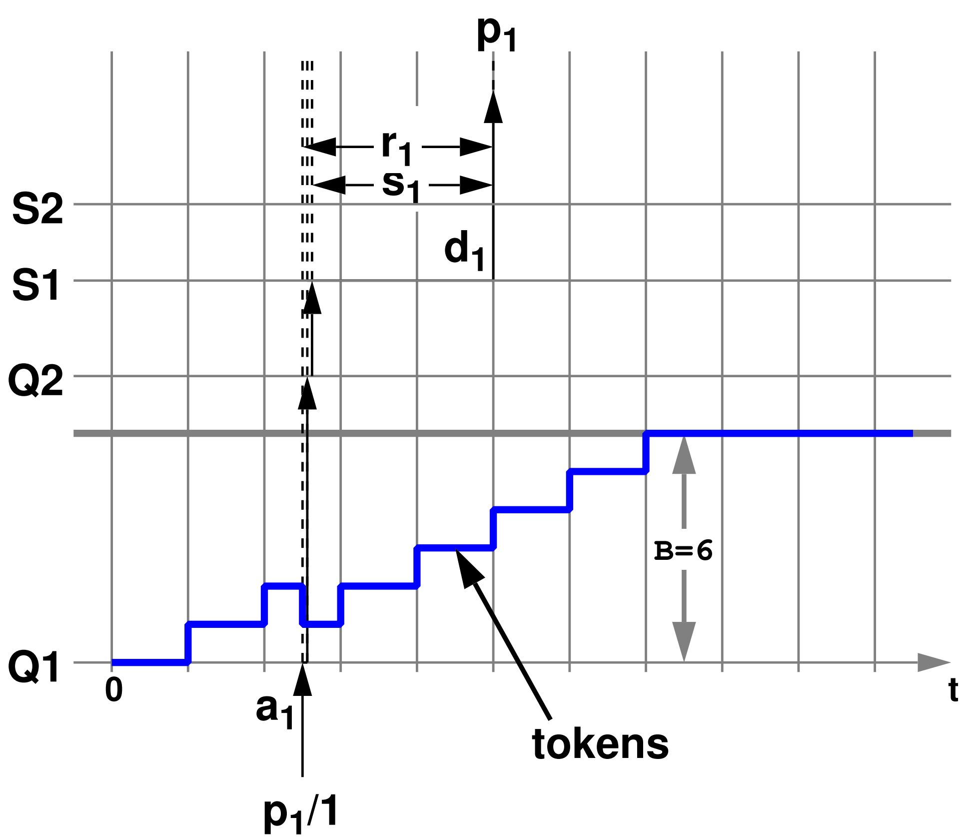


- Let your machine decide which thread to run next
  - results can never be reproducible exactly
  - debugging can be more challenging
- Compete for resources (such as Q1, Q2, and anything shared), must use a *single mutex*
- *No busy-waiting*
  - must use a *single CV*

Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response time (time in system)
- q_i^1, q_i^2 : queueing/waiting time

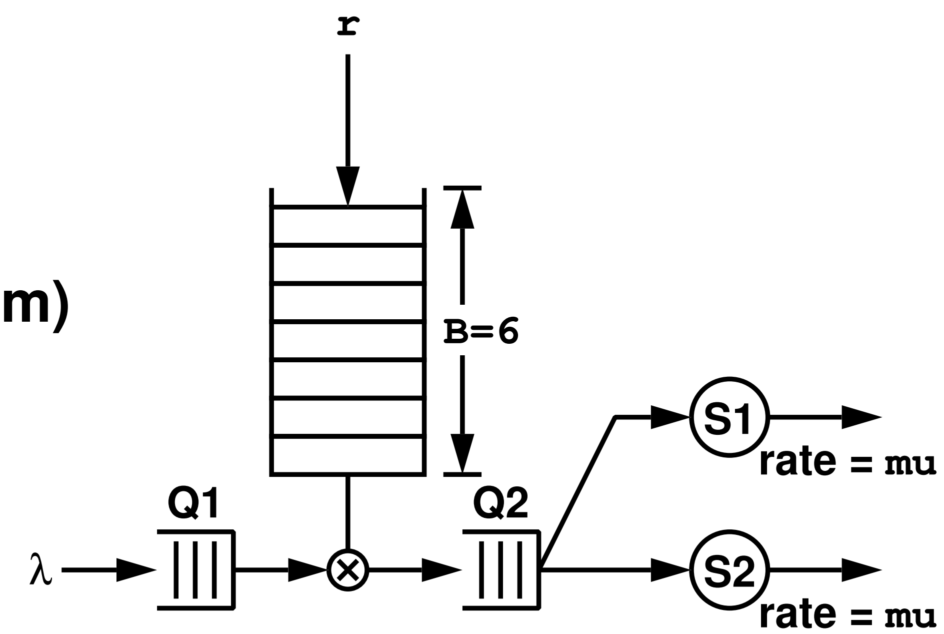


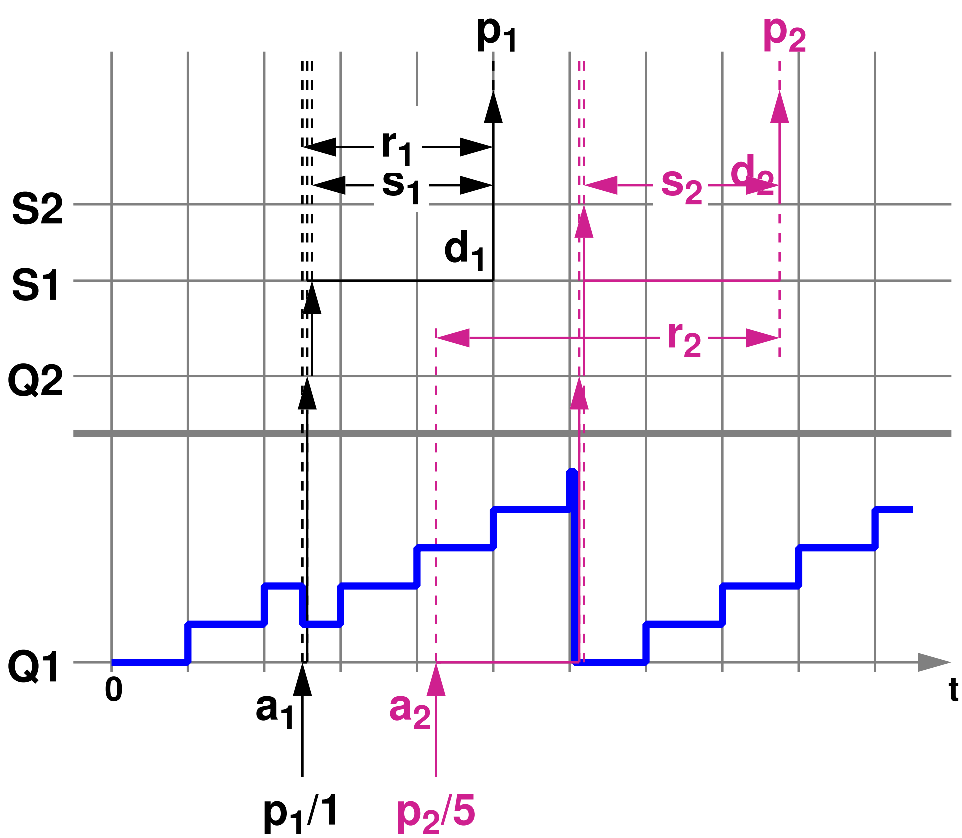


- $r_1 = d_1 - a_1$

Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response time (time in system)
- q_i^1, q_i^2 : queueing/waiting time

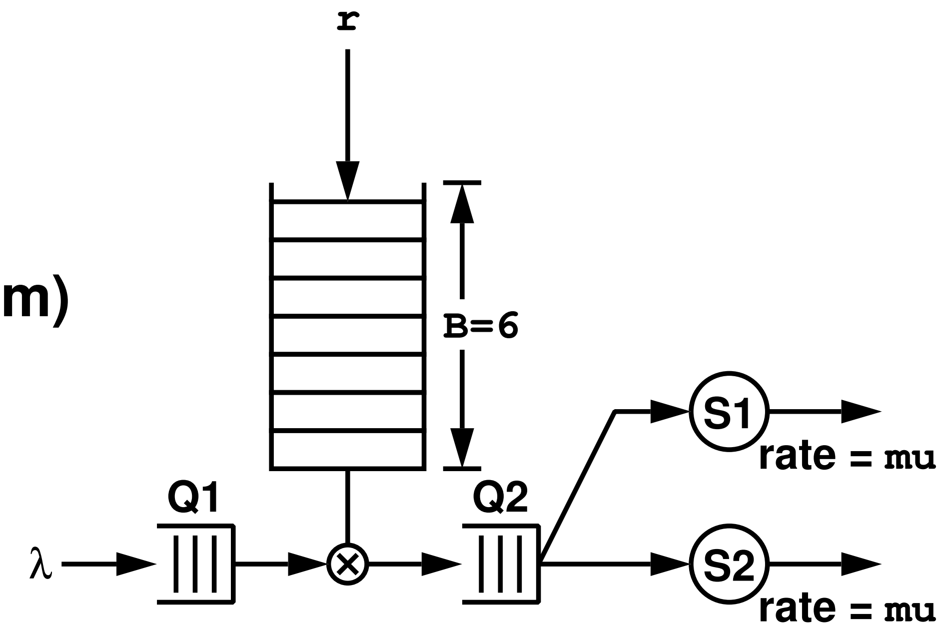


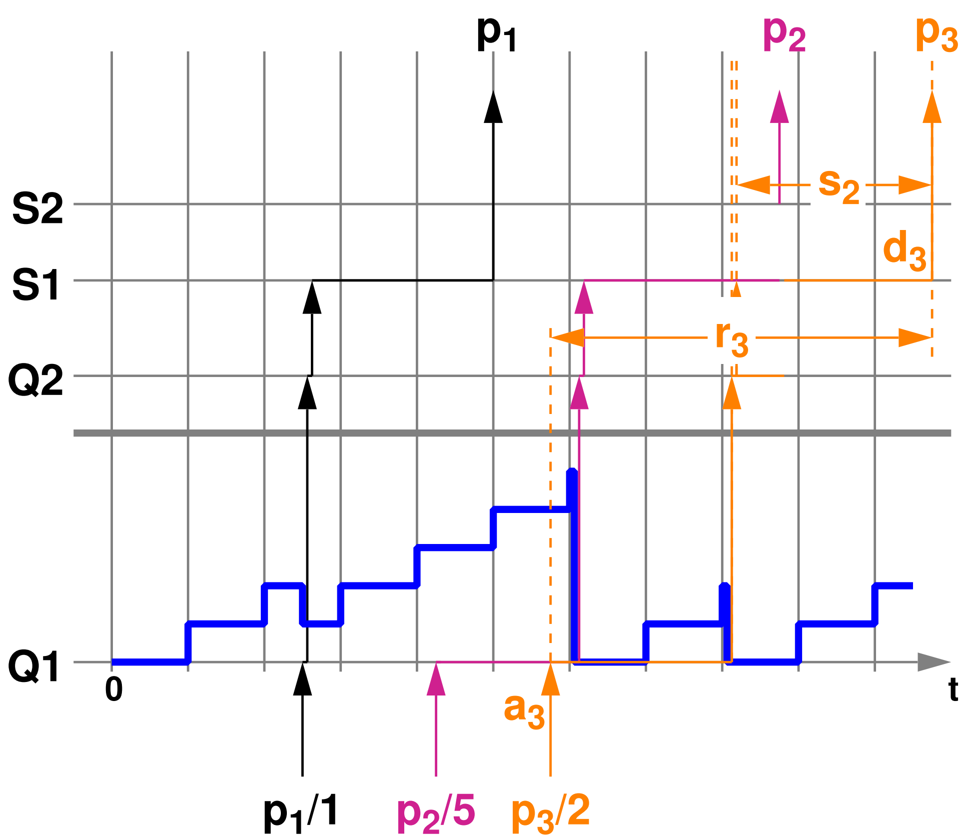


- $r_2 = d_2 - a_2$

Arrivals & Departures

- a_i : arrival time
- d_i : departure time
- s_i : service time
- r_i : response time (time in system)
- q_i^1, q_i^2 : queueing/waiting time

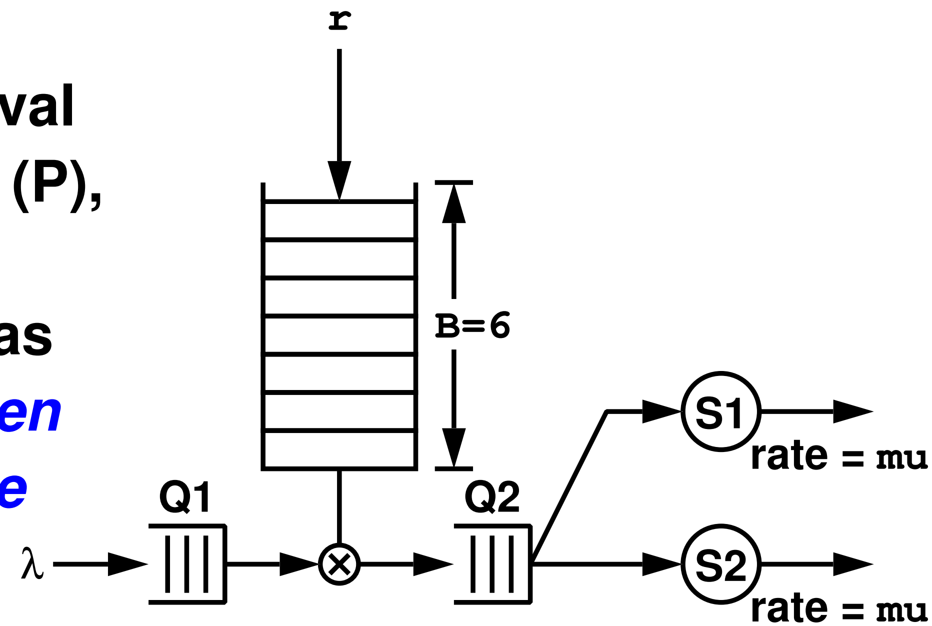




- $r_3 = d_3 - a_3$

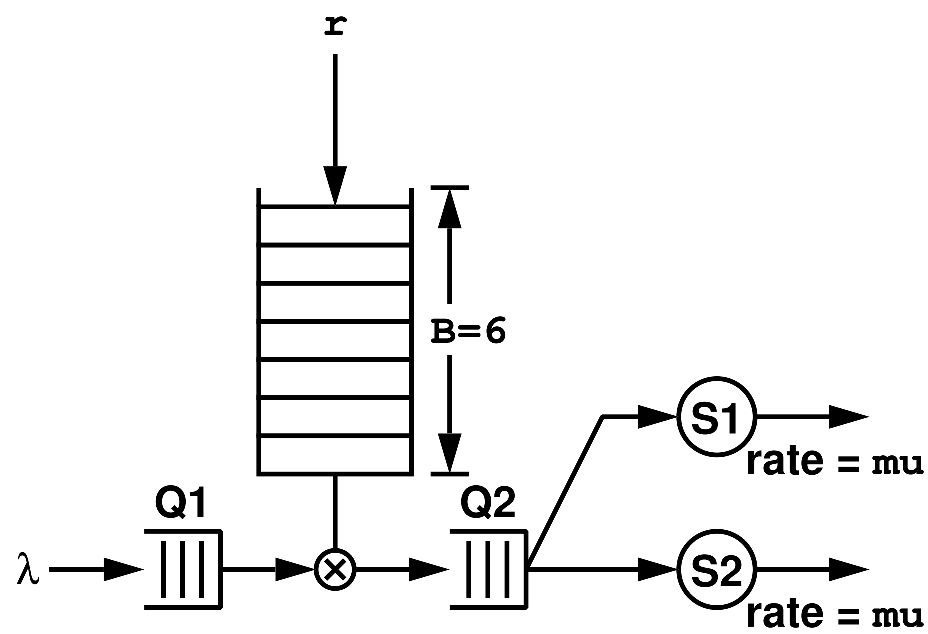
Simulation/Emulation

- Two simulation modes
  1) *Deterministic:* fixed inter-arrival time ($1/\lambda$), token requirement (P), and service time (1/mu)
  2) *Trace-driven:* every packet has its own *inter-arrival time*, *token requirement*, and *service time* (a line in a "`tsfile`")



- if you think about it carefully, there is really no difference between these two modes
  - write your code for the trace-driven mode
  - if running in deterministic mode, instead of reading a line from the "`tsfile`" to create a packet, just create a packet using information stored in global variables

Simulation/Emulation



- You will need to implement 4 *cooperating child threads*
  - *packet* arrival thread
  - *token* depositing thread
  - two *server* threads
  - these threads work together to simulate the operation of this token bucket filter
    - threads work together using *shared variables*

Simulation/Emulation

- Very high level pseudo-code for the *packet/token thread*:

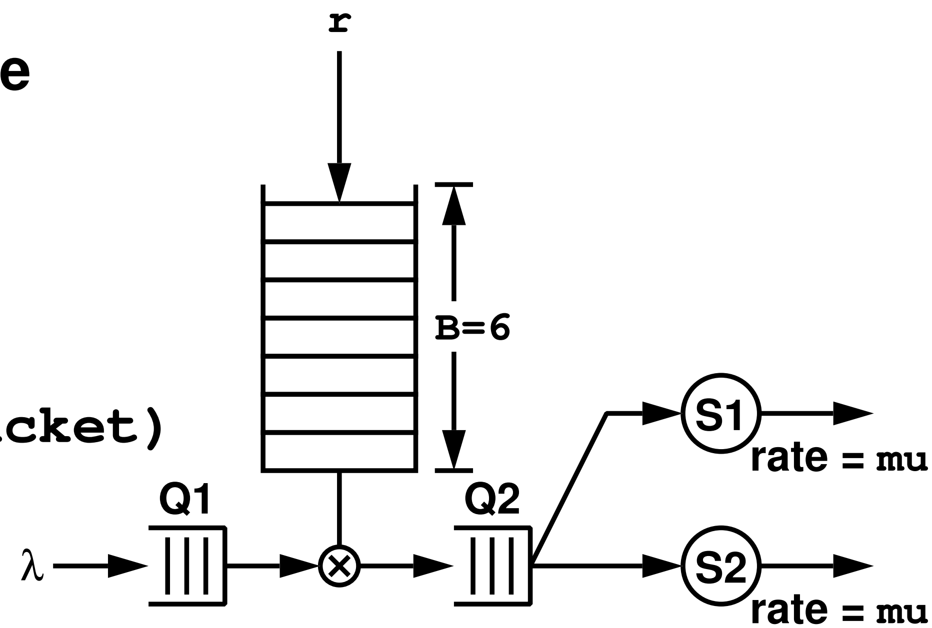
```
for (;;) {
  sleep
  generate a packet/token
  add packet/token to token bucket filter
}
```

  - where must you lock and unlock mutex?

- Very high level pseudo-code for the *server thread*:

```
for (;;) {
  wait for packet in Q2
  remove packet from Q2
  sleep (to transmit packet)
}
```

  - where must you lock and unlock mutex?

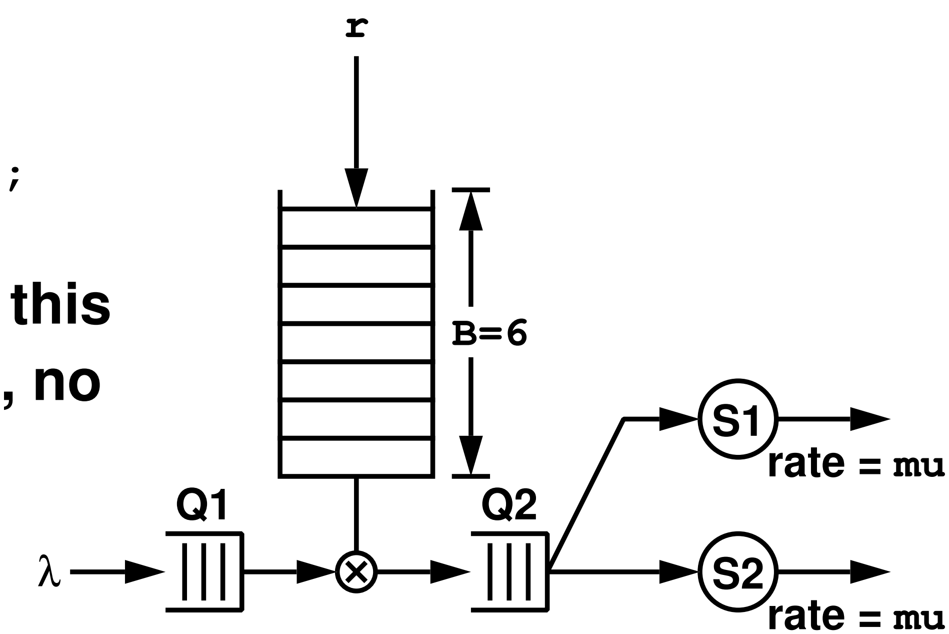


Simulation/Emulation

Packet thread pseudo-code (incomplete):

```
for (;;) {
  /* read a line from tsfile if in trace mode */
  get inter_arrival_time, tokens_needed, and service_time;
  /* calculate sleep time from inter_arrival_time */
  usleep(...);
  packet = NewPacket(tokens_needed, service_time, ...);
  pthread_mutex_lock(&mutex);
  Q1.enqueue(packet);
  ... /* other stuff */
  pthread_cond_broadcast(&cv);
  pthread_mutex_unlock(&mutex);
}
```

- must self-terminate as soon as this thread is no longer needed (i.e., no need to generate packets)
- must not call `pthread_cond_signal()`

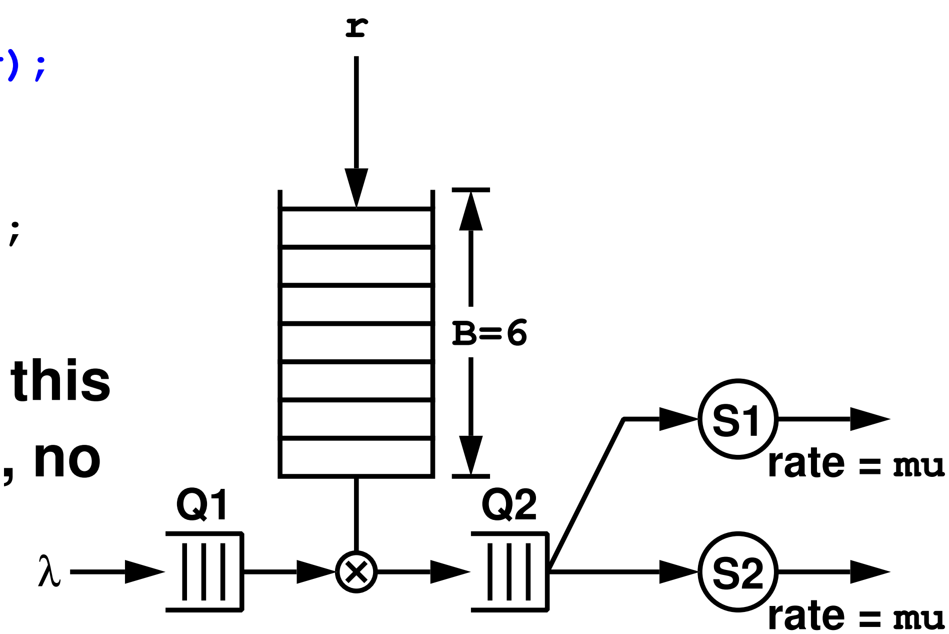


Simulation/Emulation

Token thread pseudo-code (incomplete):

```
for (;;) {
  /* calculate sleep time from inter-token arrival time */
  usleep(...);
  pthread_mutex_lock(&mutex);
  tokens++;
  if (first packet in Q1 can now be moved into Q2) {
    packet = Q1.dequeue();
    Q2.enqueue(packet);
    pthread_cond_broadcast(&cv);
    tokens = 0; /* why? */
  }
  pthread_mutex_unlock(&mutex);
}
```

- must self-terminate as soon as this thread is no longer needed (i.e., no need to generate tokens)
- must not call `pthread_cond_signal()`

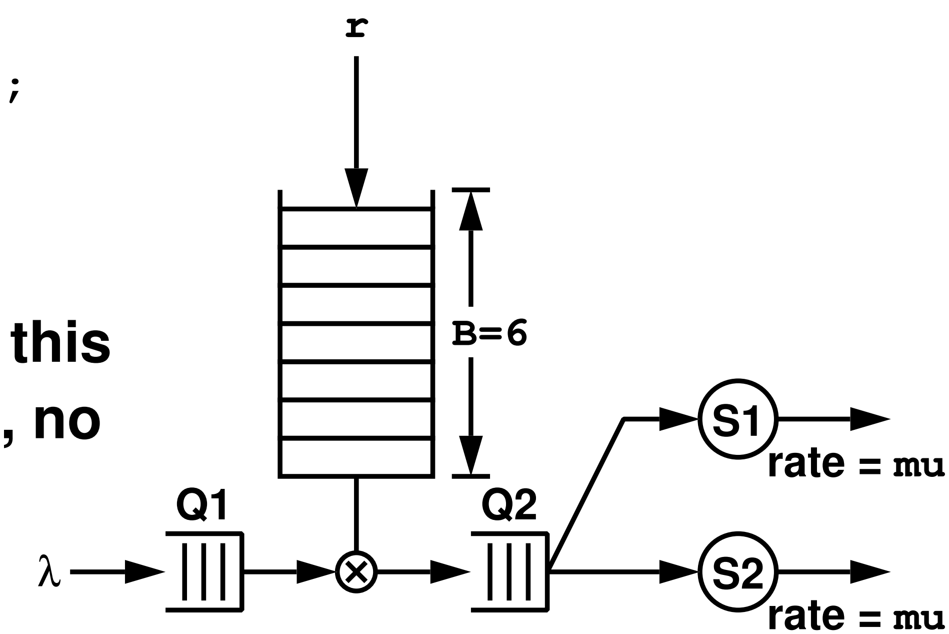


Simulation/Emulation

- Server threads pseudo-code (incomplete):
  - same first procedure for both server threads

```
for (;;) {
  /* wait for work */
  pthread_mutex_lock(&mutex);
  while (Q2.length() == 0 && !time_to_quit) {
    pthread_cond_wait(&cv, &mutex);
  }
  packet = Q2.dequeue();
  pthread_mutex_unlock(&mutex);
  /* work */
  usleep(packet.service_time);
}
```

  - must self-terminate as soon as this thread is no longer needed (i.e., no need to transmit packets)



Simulation/Emulation

- Many other requirements, for example:
  - must move a packet at the correct time
    - if a packet is eligible to be moved from Q1 to Q2, it must happen immediately
  - all threads must self-terminate when they are no longer needed
  - drop packets
    - if the token requirement for an arriving packet is too large (i.e., > B), must drop the packet
  - drop tokens
    - if an arriving token finds a full bucket, the token is dropped
  - and many more...
    - please read the spec yourself (don't get it from classmates)

Program Printout

- 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
...
```

from commandline or `tsfile`

Program Printout

```
...
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>
```

- *Timestamps* in the left column must have *microsecond* resolution
  - *measured time interval* must have *microsecond* resolution
  - use "%.6g" in printf() for <real-value>
- A value anywhere in the right column must be the exact differences between two corresponding timestamps

Program Printout

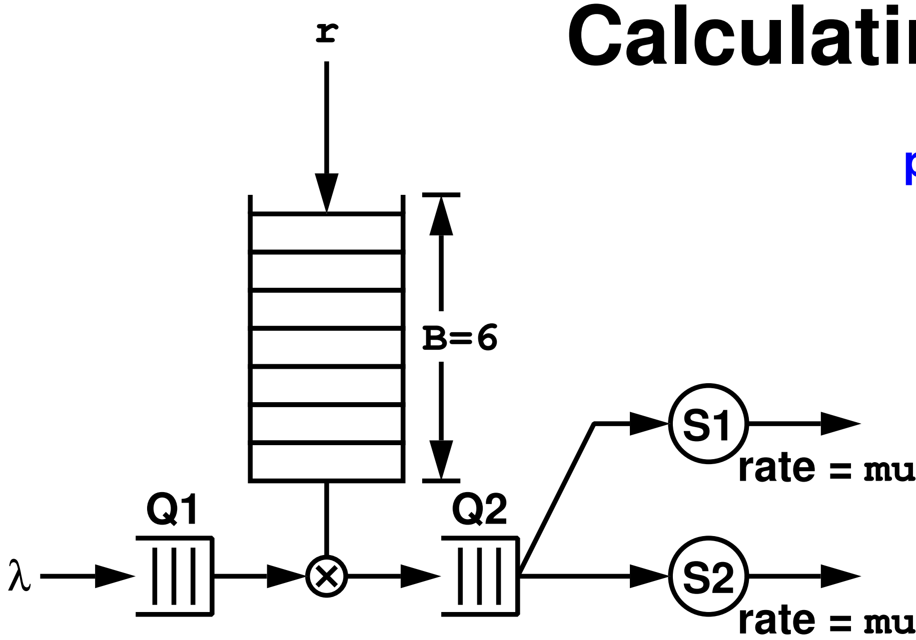
```
...
00003427.557ms: p2 departs from S2, service time = 1918.662ms, time in system = 2423.286ms
00003612.843ms: p1 departs from S1, service time = 2859.861ms, time in system = 3109.731ms
...
????????.???ms: p20 departs from S?, service time = ???.???ms, time in system = ???.???ms
????????.???ms: emulation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

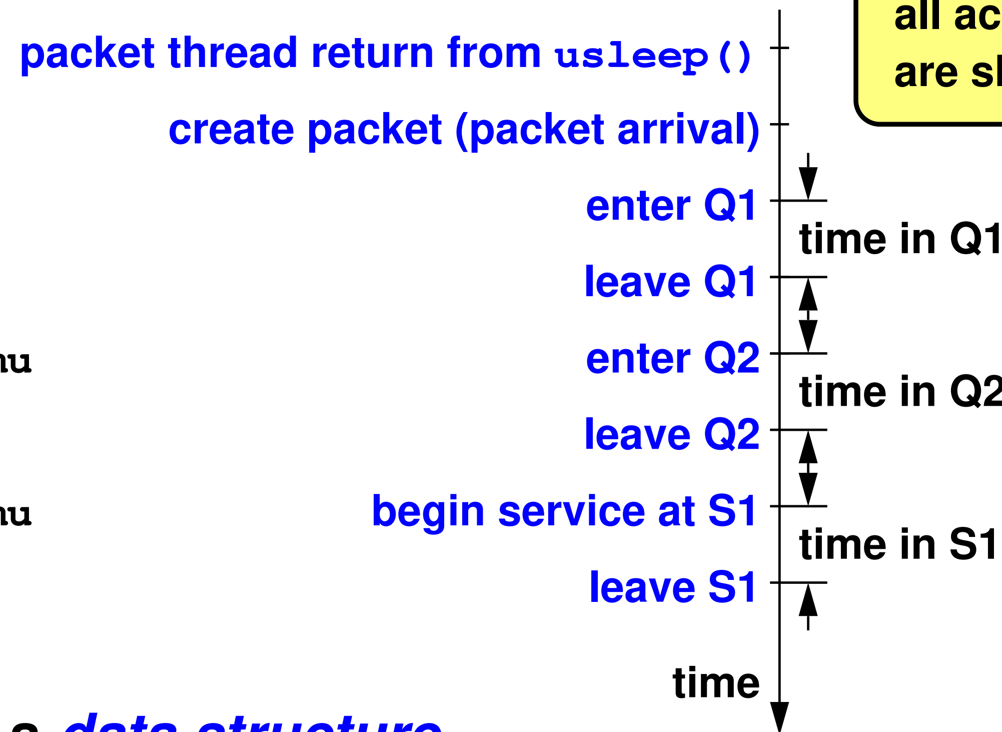
- **Ex: why must the service time for p1 be exactly 2859.861ms?**

  `3612.843ms - 752.982ms = 2859.861ms`

  - why so strict?
    - your *printout* must be *self-consistent*

Calculating Statistics



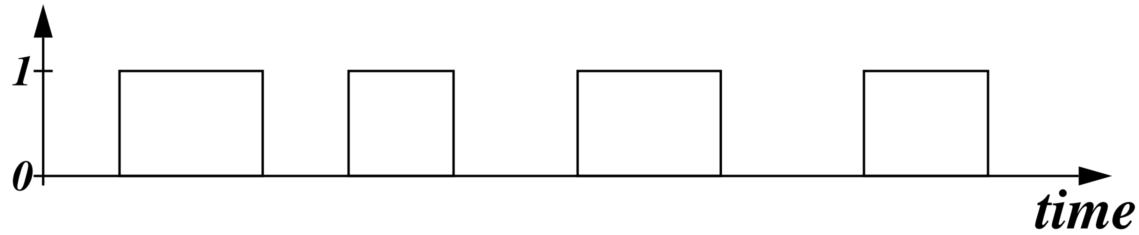


NOTE: not all activities are shown

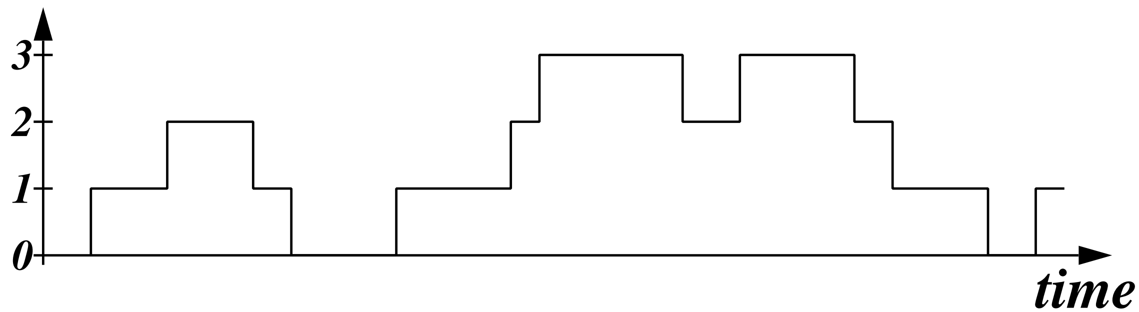
- A ***packet*** is not a thread, it's a ***data structure***
  - it should have 7 timestamps to store "measured" information
  - it should also store "packet specification" (such as specified inter-arrival time, token requirement, service time)
    - these are ***not*** "measured" values
- Some packets needs to be excluded from certain statistics
  - add to corresponding statistics only when a packet is being ejected

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 the *population variance* equation**