Housekeeping (Lecture 7 - 9/18/2013)

- **Warmup #2 due at 11:45pm on Friday, 10/4/2013**
  - **if you have code from a previous semester, be very careful and *not copy any code from it***
    - **it's best if you just get rid of it**
  - **get started soon**
    - **if you are stuck, make sure you come to see the TAs, the course producer, or me during office hours**

- **Have you installed *Ubuntu 11.10* on your laptop/desktop?**
  - **you are required to do your kernel assignments on Ubuntu 11.10**
  - **if there are any problems, I need to know now so we can get it resolved *NOW!***

Warmup #2

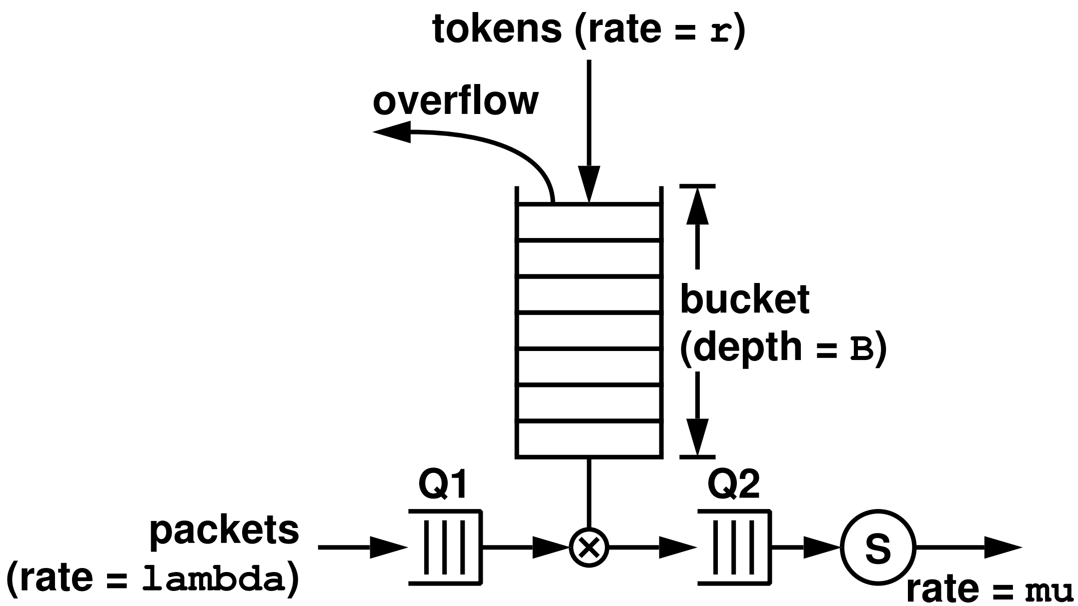
Bill Cheng

http://merlot.usc.edu/cs402-f13

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 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
    - `pthread_sigmask()/sigwait()`
    - `pthread_kill()`
  - if you want to use "thread cancellation"
    - `pthread_setcancelstate()`
    - `pthread_setcanceltype()`
    - `pthread_testcancel()`

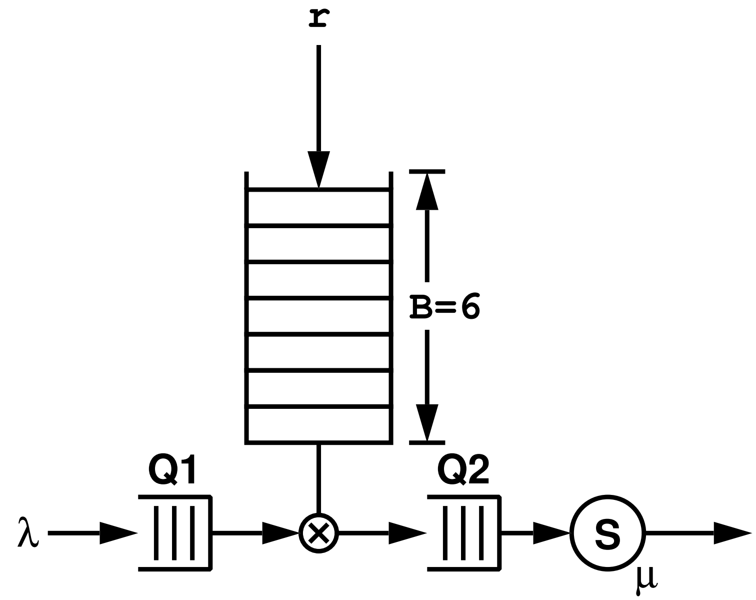
Token Bucket Filter

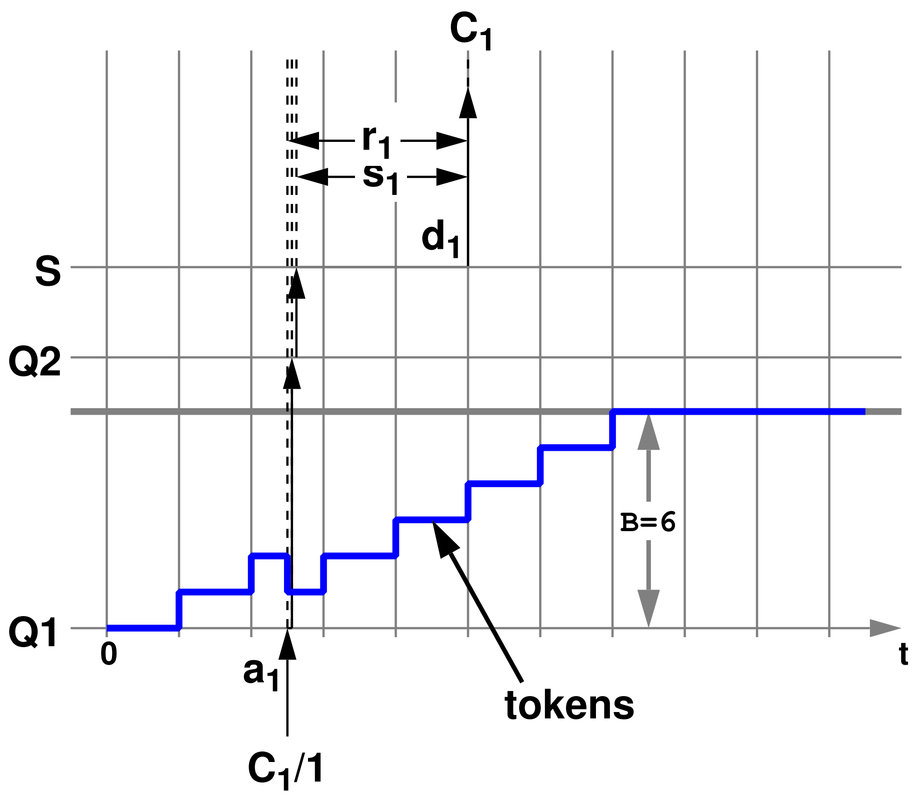


- Ex:
  - ticket scalper?!
  - traffic controller

Arrivals & Departures

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

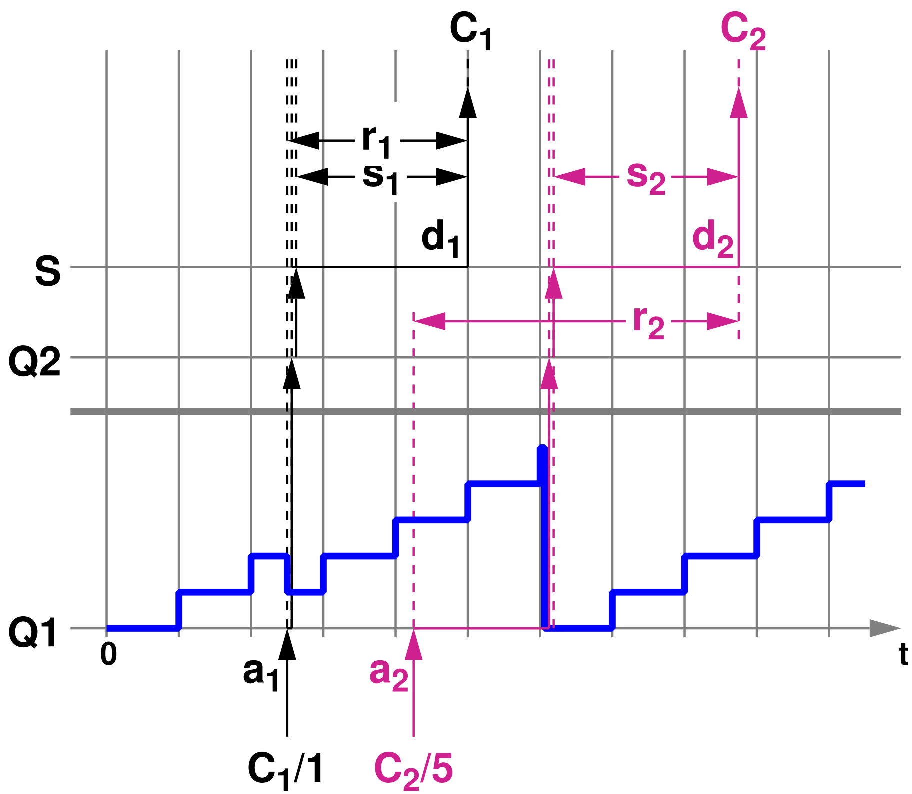


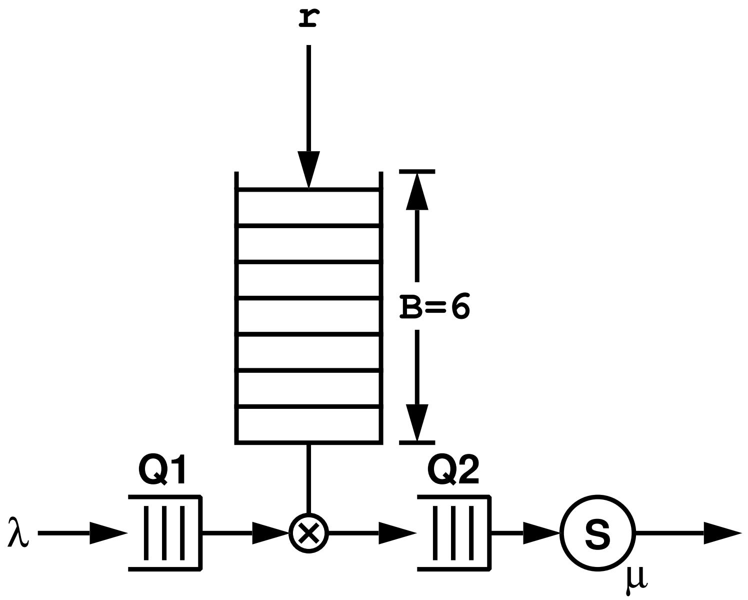


- $r_1 = d_1 - a_1$

Arrivals & Departures

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

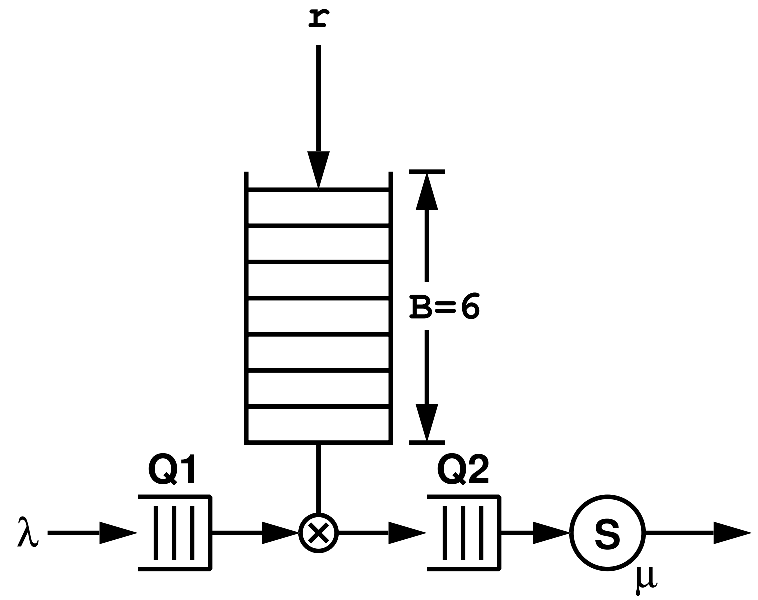


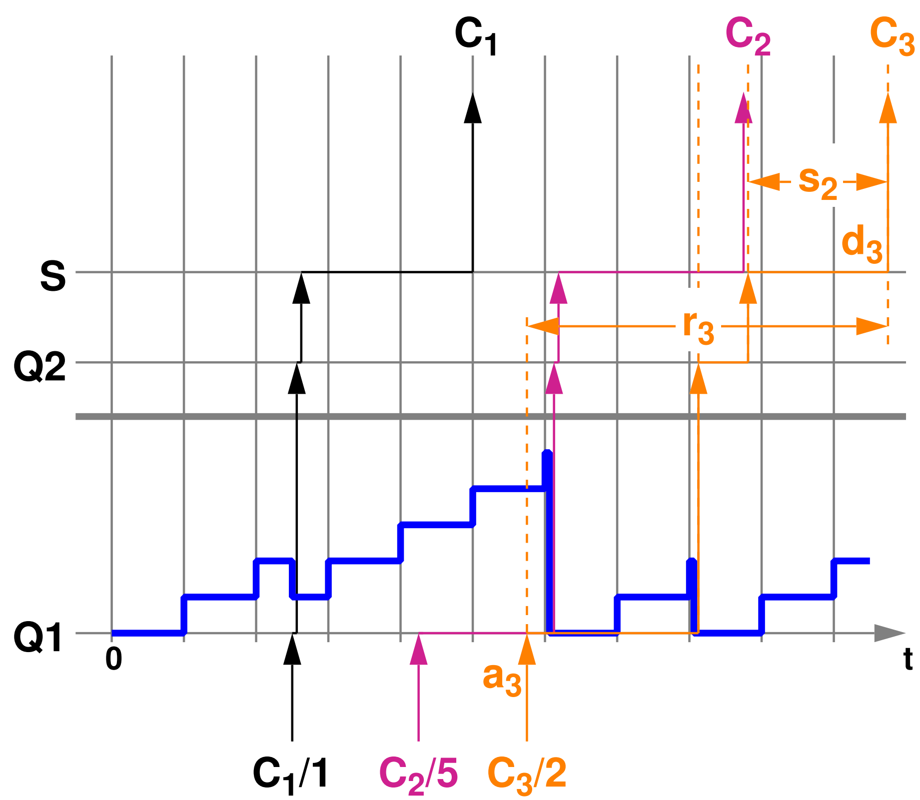


- $r_2 = d_2 - a_2$

Arrivals & Departures

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



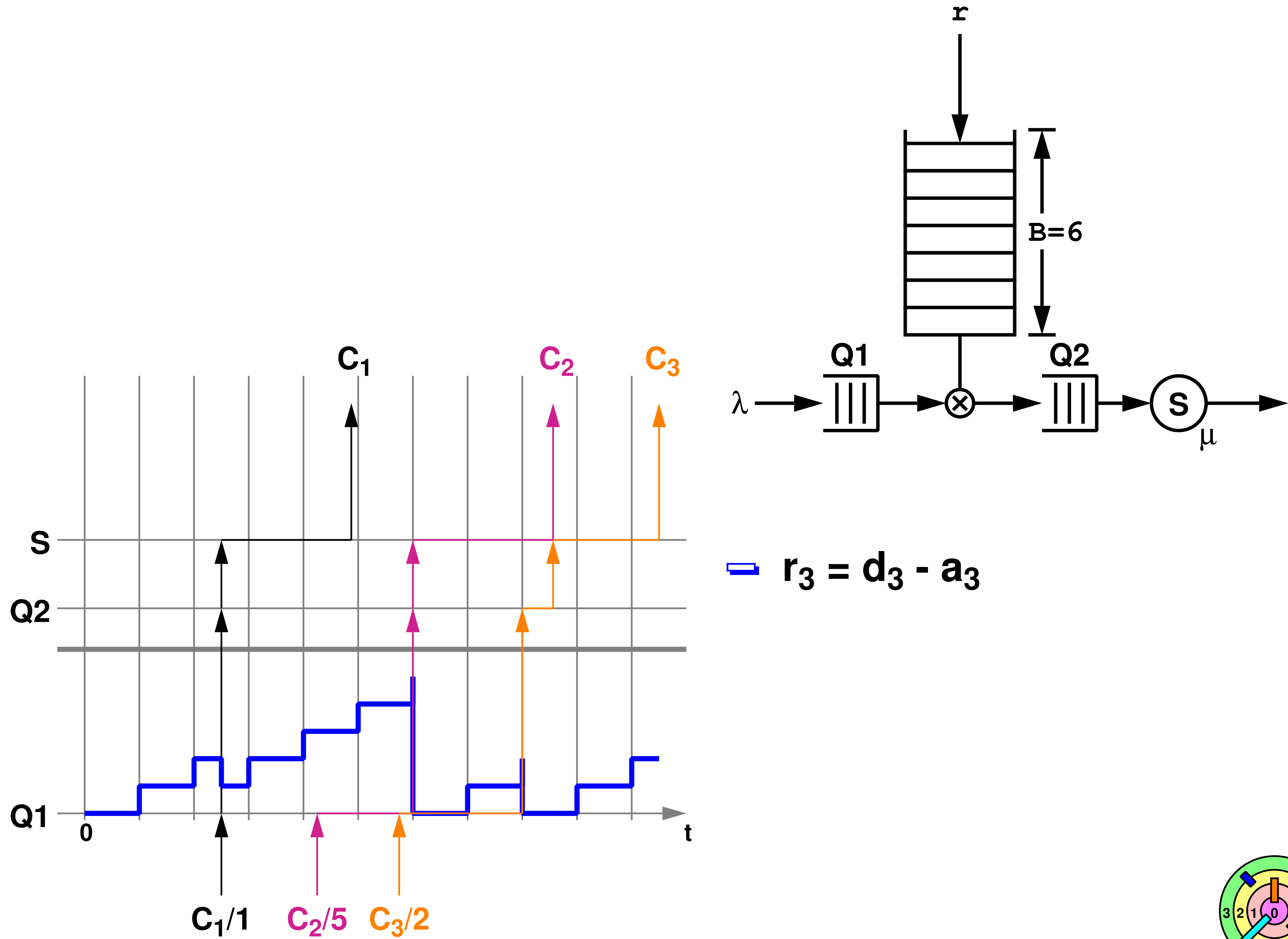


- $r_3 = d_3 - a_3$

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...)



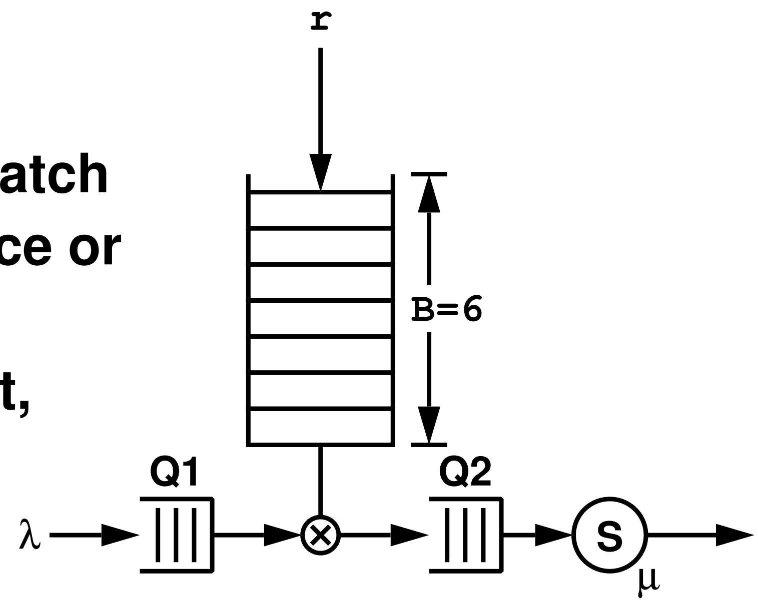
$r_3 = d_3 - a_3$

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

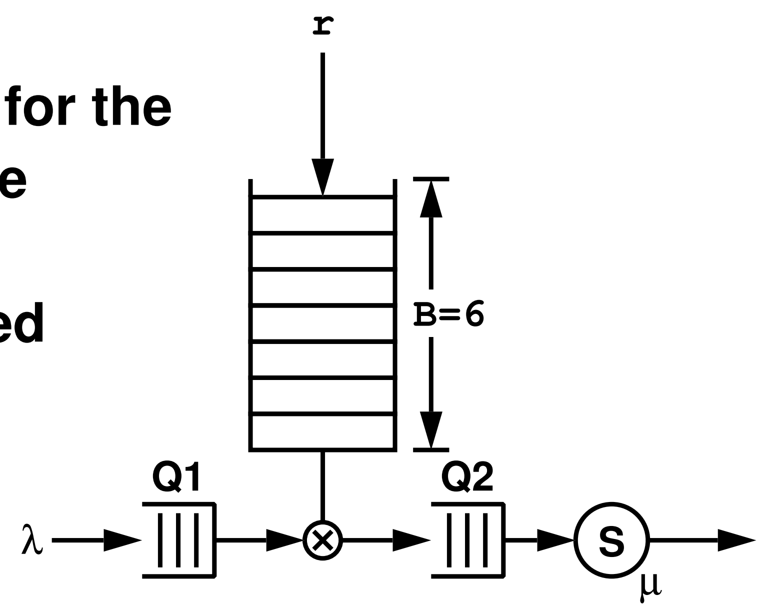
Time Driven Simulation (Cont...)

- You will need to implement 3 threads (or 1 main thread and 3 child threads)
  - the *arrival thread* sits in a loop
    - sleeps for an interval, trying to match a given interarrival 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



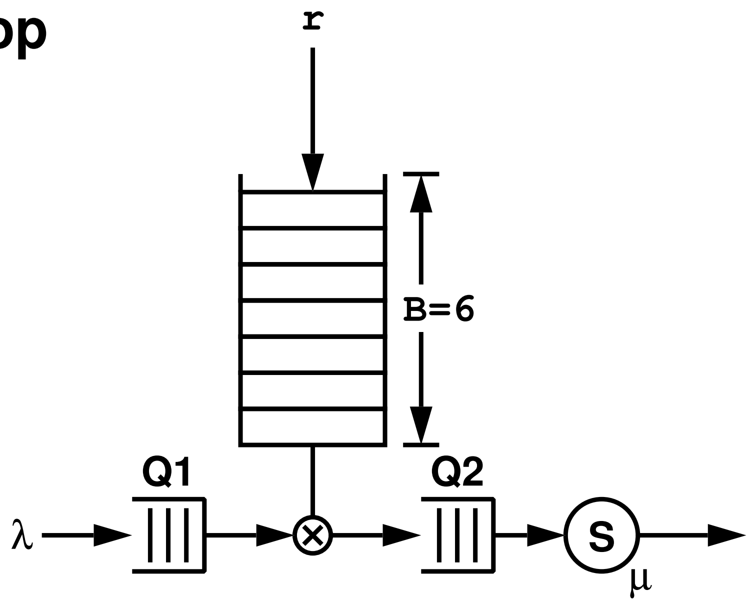
Time Driven Simulation (Cont...)

- the *server thread*
  - lock mutex, if Q2 is empty, *wait* for the *queue-not-empty condition* to be *signaled*
  - when unblocked, mutex is locked
  - if Q2 is not empty, dequeues a packet and unlock mutex
    - sleeps for an interval matching the service time of the packet; afterwards, eject the packet from the system
    - lock mutex, check if Q2 is empty, etc.
  - if Q2 is empty, go wait for the *queue-not-empty condition* to be *signaled*



Time Driven Simulation (Cont...)

- the *token arrival thread* sits in a loop
  - sleeps for an interval, trying to match a given interarrival 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



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

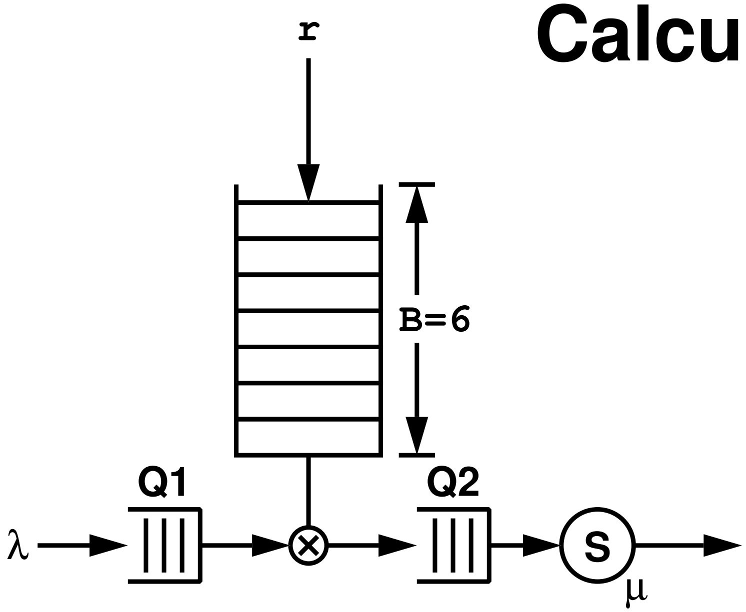
```
Emulation Parameters:
    lambda = 0.5           (if -t is not specified)
    mu = 0.35              (if -t is not specified)
    r = 1.5
    B = 10
    P = 3                  (if -t is not specified)
    number to arrive = 20  (if -t is not specified)
    tsfile = FILENAME          (if -t is specified)

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 begin service at S, time in Q2 = 0.216ms
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
...
```

Program Output

Program output must look like what's in the spec

```
...
00003612.843ms: p1 departs from S, service time = 2859.911ms, time in system = 3109.731ms
00003613.504ms: p2 begin service at S, time in Q2 = 2104.743ms
...
????????.???ms: p20 departs from S, service time = ???.???ms, time in system = ???.???ms

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 S = <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

arrival thread timeout (read clock)
lock & unlock stdout to print arrival msg
call to lock mutex to enter Q1
enter Q1 (read clock)
leave Q1 (read clock)
remove token(s)
enter Q2 (read clock)
leave Q2 (read clock)
unlock mutex
lock & unlock stdout to print leave queue and begin service msgs
begin service
leave server
lock & unlock stdout to print msg
time

overhead?
time in Q1
time in Q2
time in server
`select()` ?
charge to no one

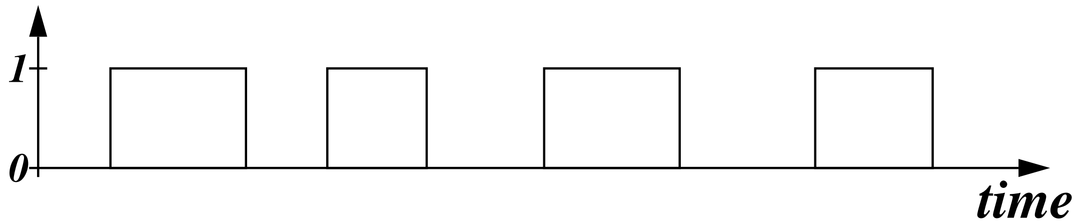
NOTE: this is just one scenario
- other scenarios are possible

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

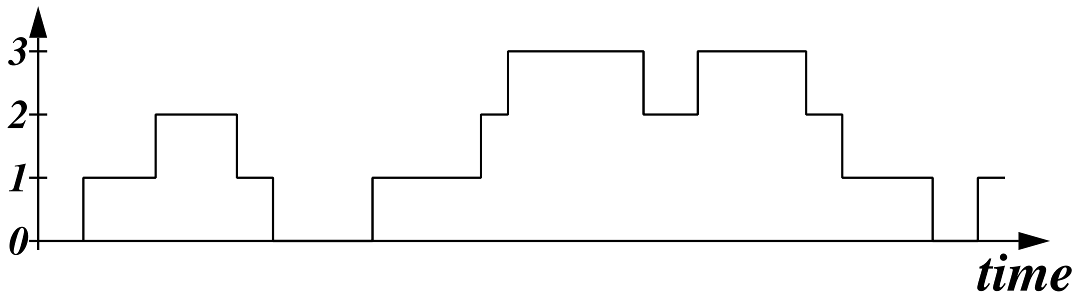
Some packets needs to be excluded from certain statistics
- e.g., if a packet is dropped, it should not participate in the time-in-system statistics

Mean and Standard Deviation

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



- Average number of customer at Q1



- Standard deviation is the squareroot of variance
  - $Var[X] = E[X^2] - (E[X])^2$

SIGINT

- **<Cntrl+C>**
  - arrival thread will stop generating packets and terminate
    - the arrival thread needs to stop the token thread
    - the arrival thread needs to clear out Q1 and Q2
  - server threads must finish serving its current packet
  - must print statistics for all packet seen
    - need to make sure that packets deleted this way do not participate in certain statistics calculation
      - you need to decide which ones and justify them

Designate A Thread To Catch A Signal

- Look at the man pages of `pthread_sigmask()` on `nunki` and try to understand the example there
  - designate child thread to handler SIGINT
  - parent thread blocks SIGINT

```
#include <pthread.h>
/* #include <thread.h> */

thread_t user_threadID;
sigset_t new;

void *handler(), interrupt();

main( int argc, char *argv[] )  {
    sigemptyset(&new);
    sigaddset(&new, SIGINT);

    pthread_sigmask(SIG_BLOCK, &new, NULL);
    pthread_create(&user_threadID, NULL, handler, argv[1]);
    pthread_join(user_threadID, NULL);

    printf("thread handler, %d exited\n",user_threadID);
    sleep(2);
    printf("main thread, %d is done\n", thr_self());
} /* end main */
```

pthread_sigmask()

- Child thread example
  - child thread unblocks SIGINT

```
struct sigaction act;

void *
handler(char argv1[])
{
    act.sa_handler = interrupt;
    sigaction(SIGINT, &act, NULL);
    pthread_sigmask(SIG_UNBLOCK, &new, NULL);
    printf("\n Press CTRL-C to deliver SIGINT\n");
    sleep(8);  /* give user time to hit CTRL-C */
}

void
interrupt(int sig)
{
    printf("thread %d caught signal %d\n", thr_self(), sig);
}
```

  - child thread is designated to handle SIGINT, no other thread will get SIGINT

Cancellation

- The user pressed <Cntrl+C>
  - or a request is generated to terminate the process
  - the chores being performed by the remaining threads are no longer needed
  - in general, we may just want to cancel a bunch of threads and not the entire process

- Concerns
  - getting cancelled at an inopportune moment
    - should not leave a mutex locked
    - or leave a data structure in an inconsistent state
      - e.g., you get a cancellation request when you are in the middle of a `insert()` operation into a doubly-linked list and `insert()` is protected by a mutex
  - cleaning up

Cancellation State

- **Send cancellation request to a thread**

```
pthread_cancel(thread)
```

- **Cancels enabled or disabled**

```
int pthread_setcancelstate(
    { PTHREAD_CANCEL_DISABLE,
      PTHREAD_CANCEL_ENABLE},
    &oldstate)
```

- **Asynchronous vs. deferred cancels**

```
int pthread_setcanceltype(
  { PTHREAD_CANCEL_ASYNCHRONOUS,
    PTHREAD_CANCEL_DEFERRED},
  &oldtype)
```

- **By default, a thread has cancellation enabled and deferred**
  - **it's for a good reason**
  - **if you are going to change it, you must ask yourself, "Why?" and "Are you sure this is really a good idea?"**

Cancellation State

- **POSIX threads rules:**
  - what `pthread_cancel()` gets called, the target thread is marked as having a *pending cancel*
  - if the target thread has cancellation *disabled*, the target thread stays in the pending cancel state
  - if the target thread has cancellation *enabled* ...
    - if the cancellation type is *asynchronous*, the target thread immediately acts on the cancel
    - if the cancellation type is *deferred*, cancellation is delayed until it reaches a *cancellation point* in its execution
      - cancellation points correspond to points in the thread's execution at which it is safe to act on the cancel
  - when a thread acts on the cancel
    - walks through a *stack* of *cleanup handlers*
    - the threads that called `pthread_cancel()` does not wait for the cancel to take effect
    - it may join and wait for the target to terminate

Cancellation Points

```
aio_suspend
close
creat
fcntl (when F_SETLCKW
       is the command)
fsync
mq_receive
mq_send
msync
nanosleep
open
pause
pthread_cond_wait
pthread_cond_timedwait
```

```
pthread_join
pthread_testcancel
read
sem_wait
sigsuspend
sigtimedwait
sigwait
sigwaitinfo
sleep
system
tcdrain
wait
waitpid
write
```

- `pthread_mutex_lock()` **is not on the list!**
- `pthread_testcancel()` creates a cancellation point
  - useful if a thread contains no other cancellation point

Cleaning Up

```
pthread_cleanup_push(
        (void)(*routine)(void *),
        void *arg)
pthread_cleanup_pop(int execute)
```

Example

```
list_item_t list_head;

void *GatherData(void *arg) {
  list_item_t *item;
  item = (list_item_t*)malloc(sizeof(list_item_t));
  pthread_cleanup_push(free, item);
  GetDataItem(&item->value);
  pthread_cleanup_pop(0);
  insert(item);
  return 0;
}
```

must match up *(like a pair of brackets)*

- in C library, `free()` is defined as:

  ```
  void free(void *ptr);
  ```

  - perfectly matches the argument types of `pthread_cleanup_push()`

Cancellation and Cleanup

```
void close_file(int fd) {
  close(fd);
}

fd = open(file, O_RDONLY);
pthread_cleanup_push(close_file, fd);
while(1) {
  read(fd, buffer, buf_size);
  // ...
}
pthread_cleanup_pop(0);
```

- should close any opened files when you clean up
- `int` is compatible with `void*`
  - well, sort of
  - `void*` can be a 64-bit quantity, so may need to be careful

Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(CleanupHandler, argument);

while(should_wait)
  pthread_cond_wait(&cv, &m);
// ... (code containing other cancellation points)
pthread_cleanup_pop(0);
pthread_mutex_unlock(&m);
```

- what should `CleanupHandler()` do?
- remember, if the thread is canceled between `push()` and `pop()`, we need to ensure that the mutex is left ***unlocked***
- can `CleanupHandler()` just call `pthread_mutex_unlock()`?
  - `pthread_cond_wait()` is a cancellation point
  - must not unlock the mutex twice!
  - should `CleanupHandler()` call `pthread_mutex_lock()` then call `pthread_mutex_unlock()`?
    - what if the mutex is locked?

Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);

while(should_wait)
  pthread_cond_wait(&cv, &m);
// ... (code containing other cancellation points)
pthread_cleanup_pop(1);
```

- pthreads library implementation ensures that a thread, when acting on a cancel within `pthread_cond_wait()`, would first lock the mutex, before calling the cleanup routines
  - this way, the above code would work correctly

Cancellation & C++

```
void tcode() {
  A a1;
  pthread_cleanup_push(handler, 0);
  foo();
  pthread_cleanup_pop(0);
}

void foo() {
  A a2;
  pthread_testcancel();
}
```

- are the destructors of `a1` and `a2` getting called?
  - not sure
  - they should get called
  - some C++ implementation does not do this correctly!

Ch 3: Basic Concepts

Bill Cheng

http://merlot.usc.edu/cs402-f13

What's Next?

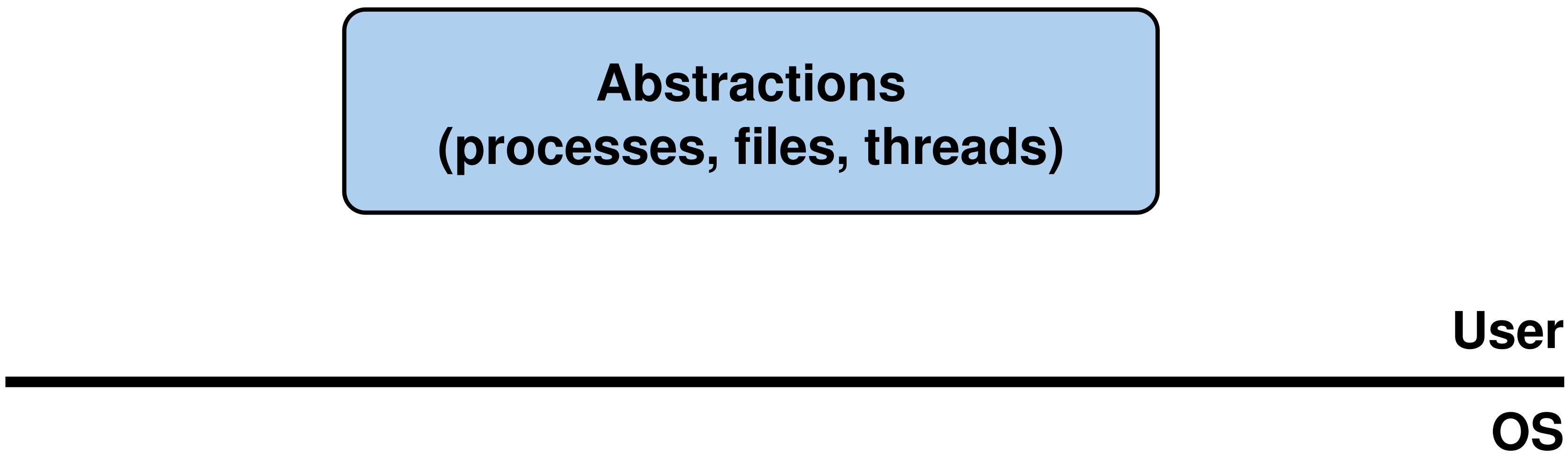
- So far, we have talked about abstractions

User

OS

What's Next?

- So far, we have talked about abstractions
  - processes, files, threads
    - stuff at the user level



What's Next?

- So far, we have talked about abstractions
  - processes, files, threads
    - stuff at the user level
- We are not ready to talk about the OS yet

Abstractions
(processes, files, threads)

User

OS

What's Next?

- So far, we have talked about abstractions
  - processes, files, threads
    - stuff at the user level
- We are not ready to talk about the OS yet
- Next step is something in between

Abstractions
(processes, files, threads)

- context for execution
- I/O architecture
- dynamic storage allocation
- linking & loading
- booting

User

OS

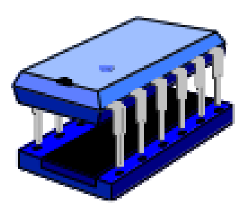
3.1 Context Switching

- Procedures
- Threads & Coroutines
- Systems Calls
- Interrupts

Context Switching

- The magic of OS
  - to provide the illusion that applications run concurrently and each application thinks it's the only application running on the processor
- The OS switches the processor from one application to another
  - switching happens transparently to the applications

Application1 Application2 Application3



Context

- What's the execution context of a thread?
  - why should we care?
    - if we are going to talk about context switching, we need to know what we are switching and how to get back
- The execution context of a thread is the *current state* of our thread
  - what does it include?
    - CPU registers, including the *instruction pointer*, *stack pointer*, and *base/frame pointer*
    - stack
    - open files
    - etc.
    - i.e., things that may affect the execution of the thread
  - turns out the stack is complicated
    - in reality, it's just the *current stack frame* of the current thread

3.1 Context Switching

- *Procedures*
- Threads & Coroutines
- Systems Calls
- Interrupts

Subroutines

```
int main() {
  int i;
  int a;
  ...
  i = sub(a, 1);
  ...
  return(0);
}
```

```
int sub(int x, int y) {
  // computers x^y
  int i;
  int result = 1;
  for (i=0; i<y; i++)
    result *= x;
  return(result);
}
```

- You are in `main()` and are ready to call `sub()`
  - how do you make sure that `sub()` has the right context to execute the code in `sub()`?
    - you need to prepare the context for `sub()`
  - how do you make sure that you can return from `sub()` and restore the `main()` context and continue to execute properly?
  - what is the context of `main()`?

Subroutines

```
int main() {
  int i;
  int a;
  ...
  i = sub(a, 1);
  ...
  return(0);
}
```

```
int sub(int x, int y) {
  // computers x^y
  int i;
  int result = 1;
  for (i=0; i<y; i++)
    result *= x;
  return(result);
}
```

- The context of `main()` includes any global variables (none here) and its local variables, `i` and `a`
- The context of `sub()` includes
  - any global variables, none here
  - its local variables, `i` and `result`
  - its arguments, `x` and `y`
- Global variables are in fixed location in the address space
- ***Local variables*** and ***arguments*** are in ***current stack frame***