Ch 2: Multithreaded Programming

Bill Cheng

http://merlot.usc.edu/cs402-s16

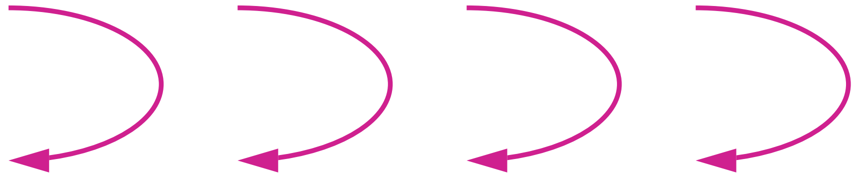
Overview

- Why threads?
- How to program with threads?
  - what is the API?
- Synchronization
  - mutual exclusion
  - semaphores
  - condition variables
- Pitfall of thread programmings

Concurrency

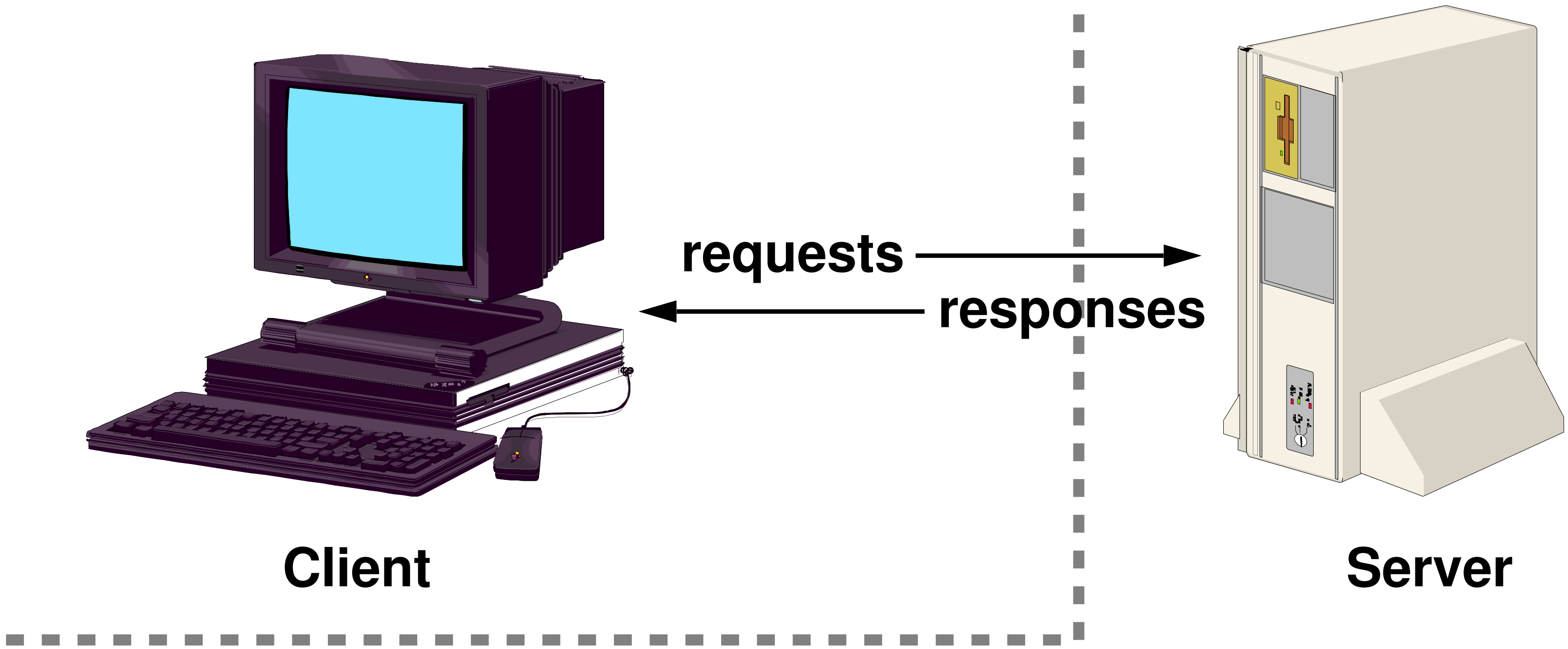
- Many things occur simultaneously in the OS
  - e.g., data coming from a disk, data coming from the network, data coming from the keyboard, mouse got clicked, jobs need to get executed
- If you have multiple processors, you may be able to handle things in parallel
  - that's real concurrency
- If you only have one processor, you may want to make it look like things are running in parallel
  - do multiplexing to create the illusion
  - as it turns out, it's a good idea to do this even if you have only have one processor
- The down side is that if you want concurrency, you have to have *concurrency control* or bad things can happen

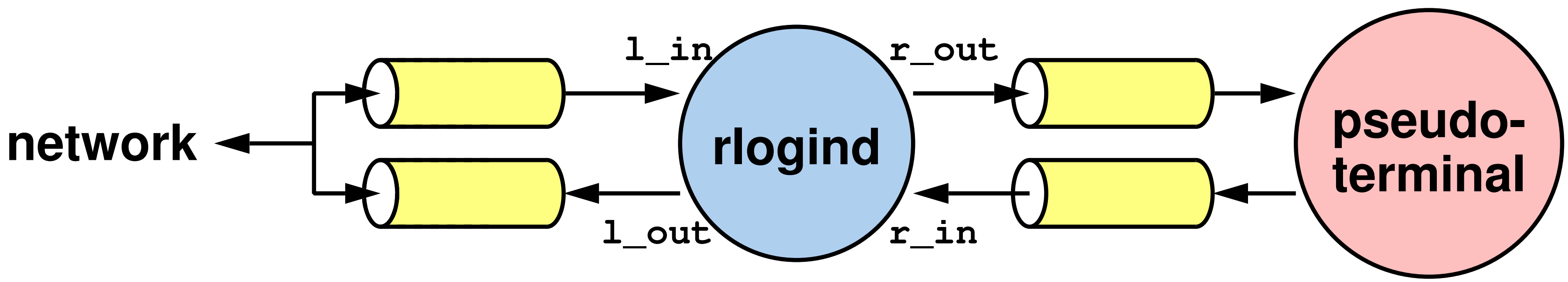
Why Threads?



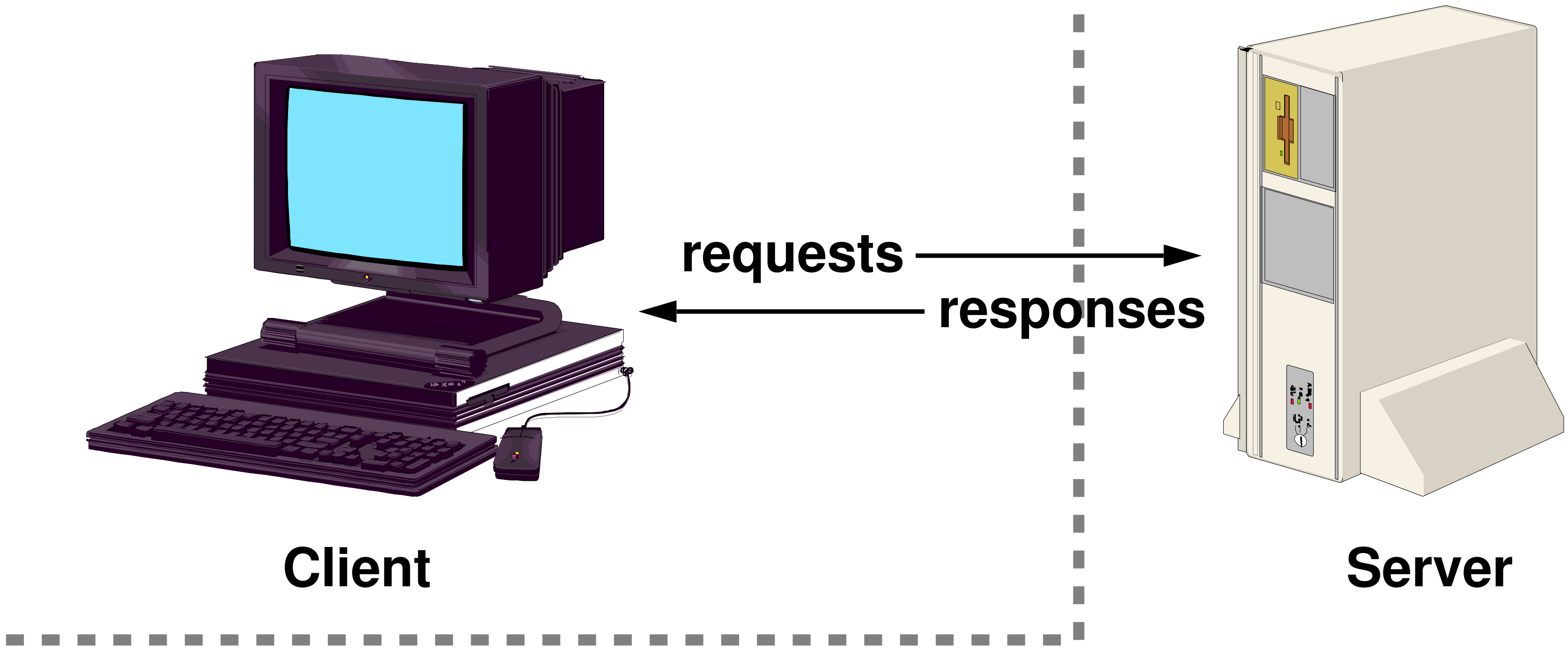
- Many things are easier to do with threads
  - *multithreading* is a *powerful paradigm*
  - makes your design *cleaner*, and therefore, less buggy
- Many things run faster with threads
  - if you are just waiting, don't waste CPU cycles, give the CPU to someone else, *without explicitly* giving up the CPU
- *Kernel threads* vs. *user threads*
  - basic concepts are the same
  - can easily do programming assignments for user-level threads
    - that's why we start here (to get your warmed up)!
    - for kernel programming assignments, you need to fill out missing parts of various kernel threads

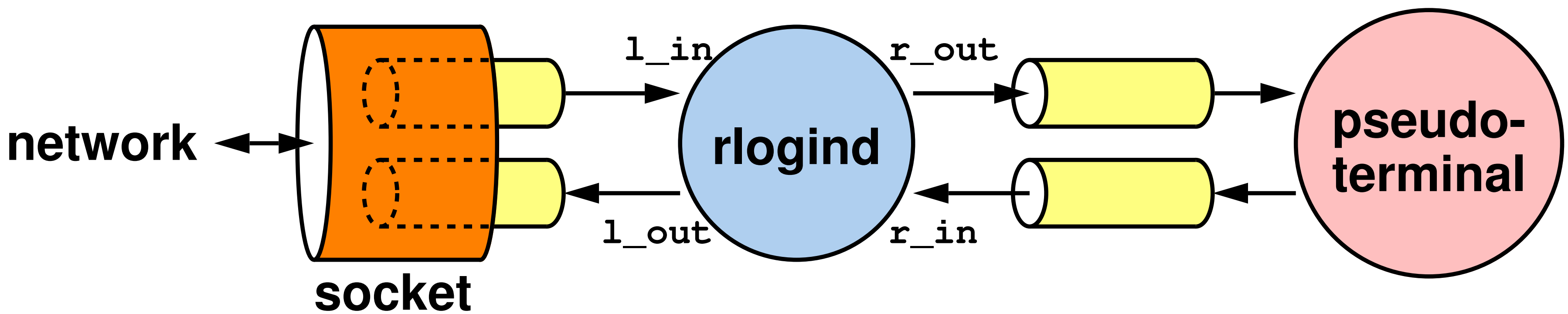
A Simple Example: `rlogind`





A Simple Example: `rlogind`

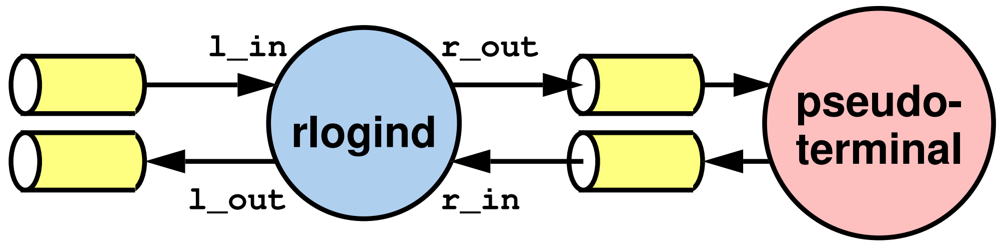




- for a socket, `l_in = l_out`, i.e., you read and write using the same file descriptor

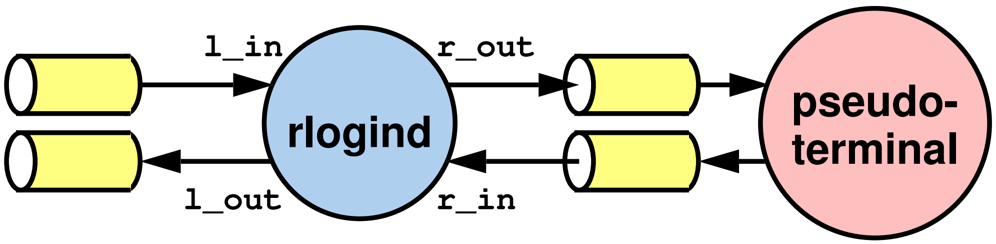
Life Without Threads

```
logind(int r_in, int r_out, int l_in, int l_out) {
  fd_set in = 0, out;
  int  want_l_write = 0, want_r_write = 0;
  int want_l_read = 1, want_r_read = 1;
  int eof = 0, tsize, fsize, wret;
  char fbuf[BSIZE], tbuf[BSIZE];

  fcntl(r_in, F_SETFL, O_NONBLOCK);
  fcntl(r_out, F_SETFL, O_NONBLOCK);
  fcntl(l_in, F_SETFL, O_NONBLOCK);
  fcntl(l_out, F_SETFL, O_NONBLOCK);

  while(!eof) {
    FD_ZERO(&in);
    FD_ZERO(&out);
    if (want_l_read) FD_SET(l_in, &in);
    if (want_r_read) FD_SET(r_in, &in);
    if (want_l_write) FD_SET(l_out, &out);
    if (want_r_write) FD_SET(r_out, &out);
    select(MAXFD, &in, &out, 0, 0);
    if (FD_ISSET(l_in, &in)) {
      if ((tsize = read(l_in, tbuf, BSIZE)) > 0) {
        want_l_read = 0;
        want_r_write = 1;
      } else { eof = 1; }
    }
```

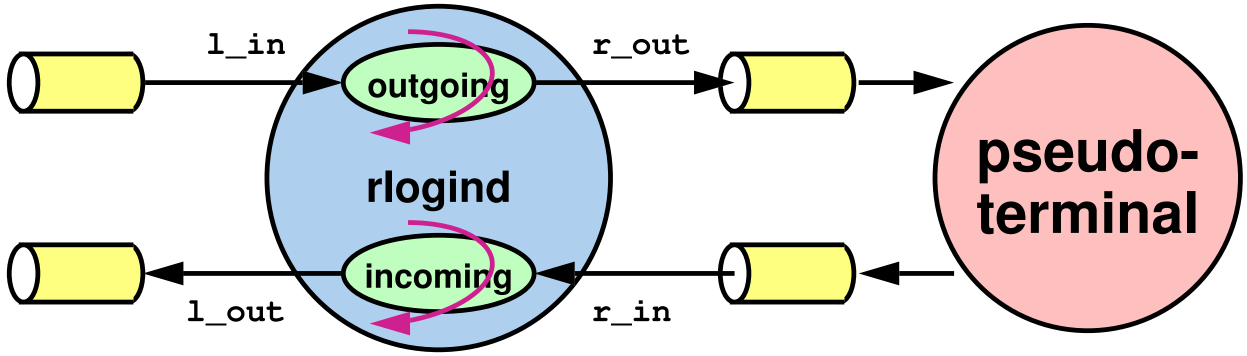


Life Without Threads

```
    if (FD_ISSET(r_in, &in)) {
      if ((fsize = read(r_in, fbuf, BSIZE)) > 0) {
        want_r_read = 0;
        want_l_write = 1;
      } else { eof = 1; }
    }
    if (FD_ISSET(l_out, &out)) {
      if ((wret = write(l_out, fbuf, fsize)) == fsize) {
        want_r_read = 1;
        want_l_write = 0;
      } else if (wret >= 0) {
        tsize -= wret;
      } else { eof = 1; }
    }
    if (FD_ISSET(r_out, &out)) {
      if ((wret = write(r_out, tbuf, tsize)) == tsize) {
        want_l_read = 1;
        want_r_write = 0;
      } else if (wret >= 0) {
        tsize -= wret;
      } else { eof = 1; }
    }
  }
}
```



Life With Threads

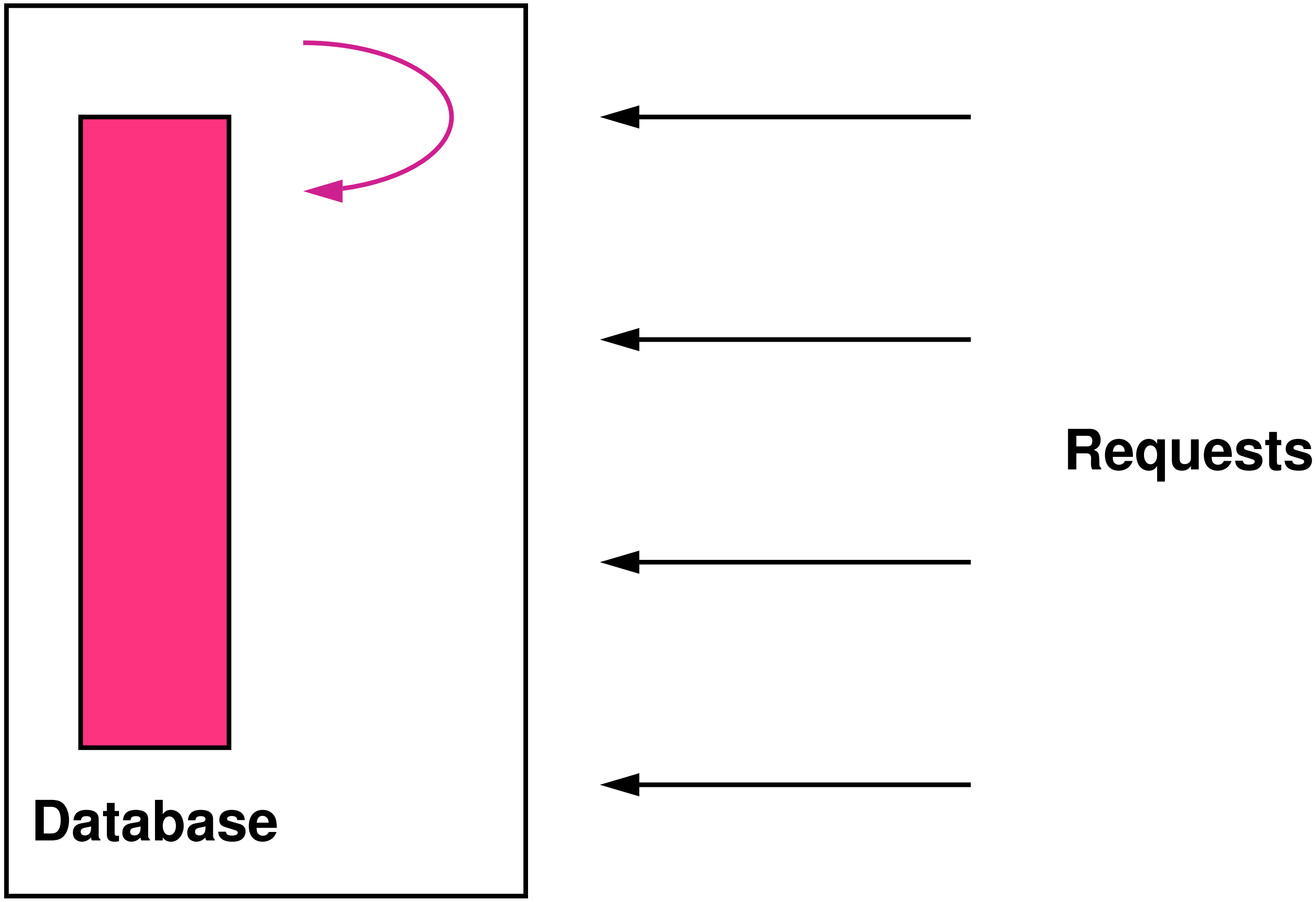



```
incoming(int r_in, int l_out) {
  int eof = 0;
  char buf[BSIZE];
  int size;

  while (!eof) {
    size = read(r_in, buf, BSIZE);
    if (size <= 0)
      eof = 1;
    if (write(l_out, buf, size) <= 0)
      eof = 1;
  }
}
```

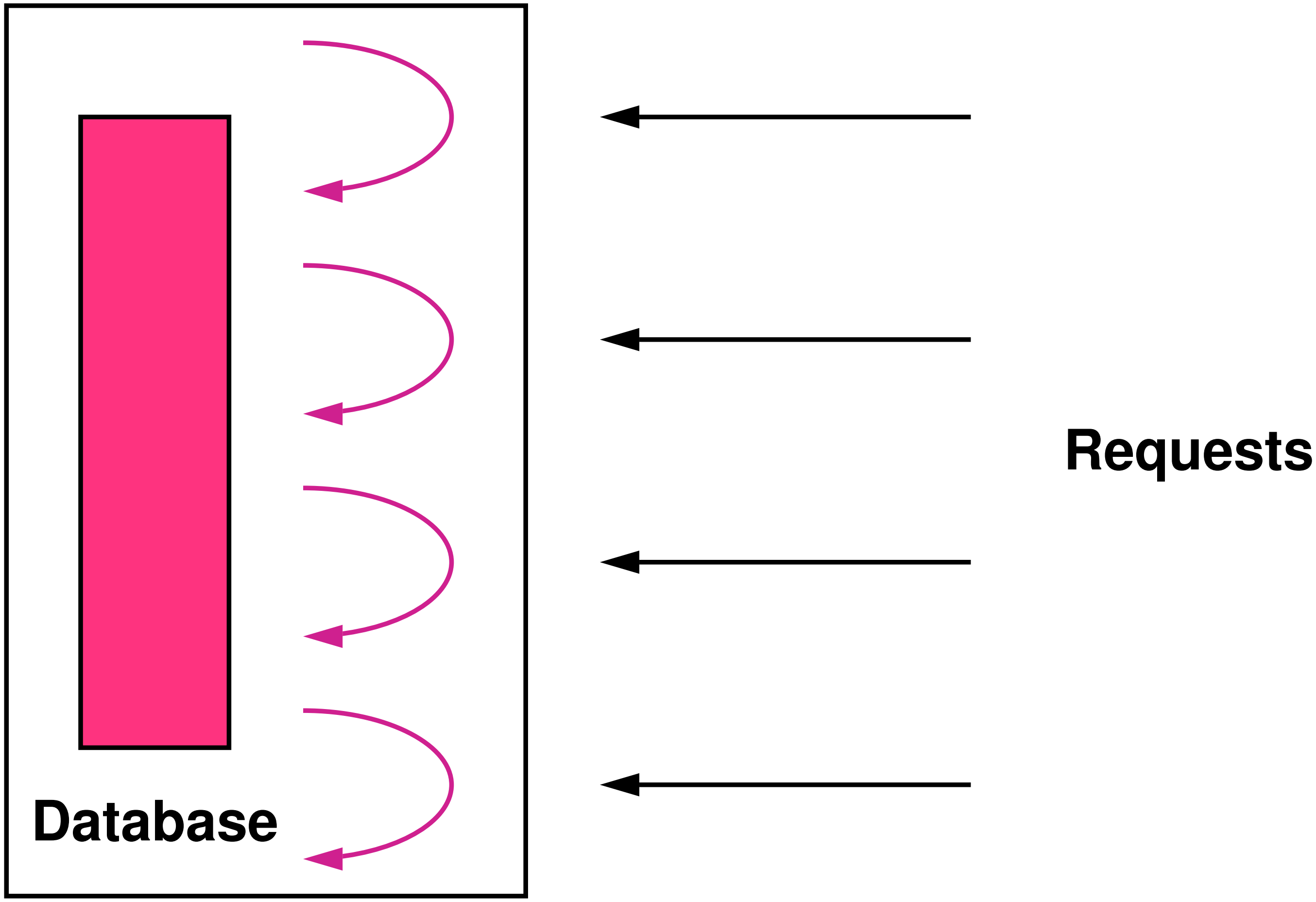
```
outgoing(int l_in, int r_out) {
  int eof = 0;
  char buf[BSIZE];
  int size;

  while (!eof) {
    size = read(l_in, buf, BSIZE);
    if (size <= 0)
      eof = 1;
    if (write(r_out, buf, size) <= 0)
      eof = 1;
  }
}
```

- don't have to call `select()`

Single-Threaded Database Server



Multithreaded Database Server



- will be very difficult to implement this without using threads if you want to handle a large number of requests simultaneously

2.2 Programming With Threads

- *Threads Creation & Termination*
- Threads & C++
- Synchronization
- Thread Safety
- Deviations

Creating a POSIX Thread

- `man pthread_create`

```
SYNOPSIS
   #include <pthread.h>

   int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void *),
        void *arg);

   Compile and link with -pthread.
```

- the `start_routine` is also known as the *"first procedure"* or *"thread function"* of the child thread
  - it's like `main()` for the child thread
- the "thread ID" of the newly created thread will be returned in the first argument of `pthread_create()`
  - may *not* be a *Thread Control Block*

Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i=0; i<nr_of_server_threads; i++)
    pthread_create(&thread,    // thread ID
                   0,          // default attributes
                   server,     // first procedure
                   argument);  // argument
}

void *server(void *arg) {
  // perform service
  return(0);
}
```

child thread *starts* executing here

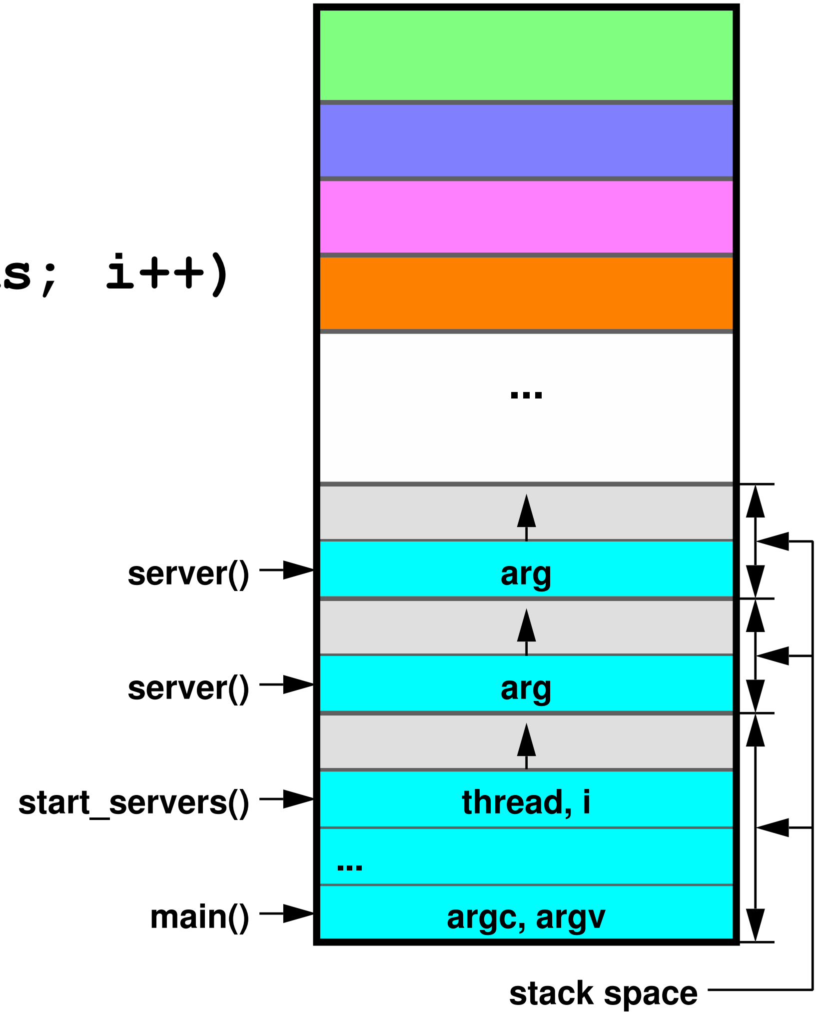
arg = argument (from caller)

child thread *ends* when *return* from its *start routine / first procedure*

- `pthread_create()` returns 0 if successful
- POSIX 1003.1c standard

Creating a POSIX Thread

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i=0; i<nr_of_server_threads; i++)
    pthread_create(&thread,
                   0,
                   server,
                   argument);
}

void *server(void *arg) {
  // perform service
  return(0);
}
```



- every thread needs a separate stack
  - first stack frame in every child thread corresponds to `server()`
    - one arg in each of these stack frames

Creating a POSIX Thread

- These are the same:
  - keep thread handle in the stack

```
pthread_t thread;
pthread_create(&thread, ...);
```

  - keep thread handle in the heap

```
pthread_t *thread_ptr =
    (pthread_t*)malloc(sizeof(pthread_t));
pthread_create(thread_ptr, ...);
```

  - need to make sure that eventually you will call the following to not leak memory

```
free(thread_ptr);
```

Creating a Win32 Thread

```
start_servers( ) {
  HANDLE thread;
  DWORD id;
  int i;
  for (i=0; i<nr_of_server_threads; i++)
    thread = CreateThread(
        0,       // security attributes
        0,       // default # of stack pages allocated
        server,  // first procedure
        arg,     // argument
        0,       // default attributes
        0,       // creation flags
        &id);    // thread ID
}

DWORD WINAPI server(void *arg) {
  // perform service
  return(0);
}
```

- We won't talk about Win32 much

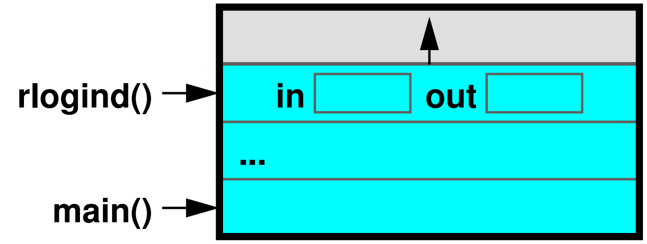
Complications

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  pthread_create(&in_thread,
                 0,
                 incoming,
                 r_in, l_out); // Cannot do this ...
  pthread_create(&out_thread,
                 0,
                 outgoing,
                 l_in, r_out); // Cannot do this ...
  /* How do we wait till they are done? */
}
```

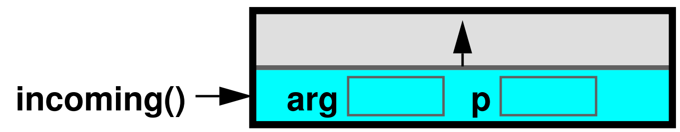
Multiple Arguments

```
typedef struct {
  int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
                 0,
                 incoming,
                 &in);
  ...
  /* How do we wait till they are done? */
}

void *incoming(void *arg) {
  two_ints_t *p=(two_ints_t*)arg;
  ... p->first ...
  return NULL;
}
```

Multiple Arguments




```
typedef struct {
  int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;

  two_ints_t in={r_in, l_out}, out={l_in, r_out};
  pthread_create(&in_thread,
                 0,
                 incoming,
                 &in);
  ...
  /* How do we wait till they are done? */
}

void *incoming(void *arg) {
  two_ints_t *p=(two_ints_t*)arg;
  p->first ...
  ...
}
```



Multiple Arguments

- Need to be careful how to pass argument to new thread when you call `pthread_create()`
  - there is no way to pass multiple arguments in either POSIX or Win32
  - passing address of a *local* variable (like the previous example) only works if we are certain the this storage doesn't go out of scope until the thread is done with it
  - passing address of a *static* or a *global* variable only works if we are certain that only one thread at a time is using the storage
  - passing address of a *dynamically* allocated storage only works if we can free the storage when, and only when, the thread is finished with it
    - this would not be a problem if the language supports garbage collection

- Ask yourself, "How can you be sure?"
  - if the answer is, "I hope it works", then you need a different solution

When Is The Child Thread Done?

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
  pthread_t in_thread, out_thread;
  two_ints_t in={r_in, l_out}, out={l_in, r_out};

  pthread_create(&in_thread, 0, incoming, &in);
  pthread_create(&out_thread, 0, outgoing, &out);

  pthread_join(in_thread, 0);
  pthread_join(out_thread, 0);
}
```

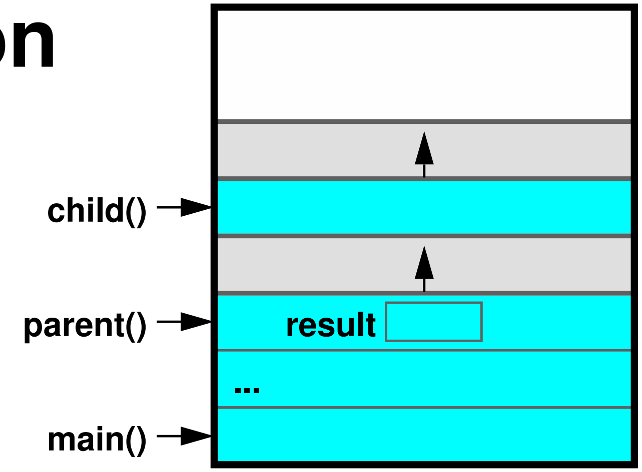
Thread Termination

- Thread return values
  - which threads receive these values
  - how do they do it?
    - clearly, receiving thread must wait until the producer thread produced it, i.e., producer thread has terminated
    - so we must have a way for one thread to wait for another thread to terminate
  - must have a way to say which thread you are waiting for
    - need a unique identifier
    - tricky if it can be reused

- To wait for another thread to terminate

```
int pthread_join(thread_t thread,
                 (void **)ret_value);
```

Thread Termination



- How does a thread *self-terminate*?
  1) return from its "first procedure"
     - return a value of type `(void*)`
  2) call `pthread_exit(ret_value)`
     - `ret_value` is of type `(void*)`

```
parent() {
  pthread_t thread;
  void *result;
  pthread_create(&thread,
      0, child, 0);
  pthread_join(thread,
      (void**)&result);
  switch ((int)result) {
  case 1: ...
  case 2: ...
  }
  ...
}
```

```
void *child(void *arg) {
  ...
  if (terminate_now) {
    pthread_exit((void*)1);
  }
  return((void*)2);
}
```

Thread Control Block

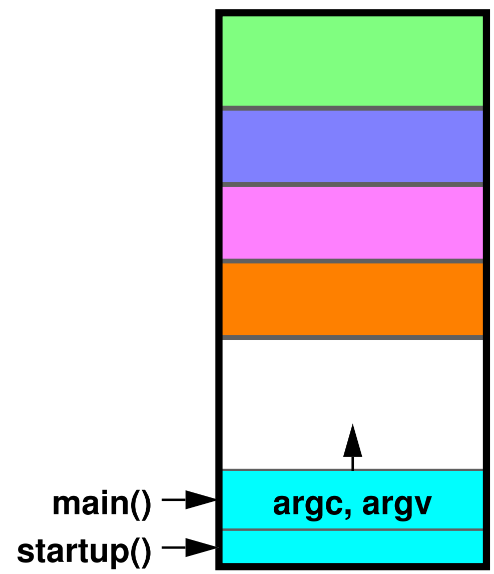
TID
Exit/Return Code
...

Thread Termination

- **Difference between `pthread_exit()` and `exit()`**
  - `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what will this code do?

```
int main(int argc, char *argv[]) {
  // create all the threads
  return(0);
}
```

    - when `main()` returns, `exit()` will be called
      - as a result, none of the created child threads may get a chance to run



Thread Termination

- Difference between `pthread_exit()` and `exit()`
  - `pthread_exit()` terminates only the calling thread
  - `exit()` terminates the process, including all threads running in it
    - it will not wait for any thread to terminate
    - what about this code?

```
int main(int argc, char *argv[]) {
  // create all the threads
  pthread_exit(0); // exit the main thread
  return(0);
}
```

    - here, `pthread_exit()` will terminate the main thread, so `exit()` is never called
      - as it turns out, this special case is taken care of in the pthread library implemetation

- You should use `pthread_join()` unless you are absolutely sure

Thread Termination

- **Any thread can join with any other thread**
  - **there's *no parent/child relationships* among threads**
    - **unlike process termination and `wait()`**

- **What happens if a thread terminates and no other thread wants to join with this thread?**
  - **it also goes into a *zombie* state**
    - **all the thread related information is freed up, except for the thread ID and return code**

- **What if two threads want to join with the same thread?**
  - **after the first thread joins, the thread ID and return code are freed up and the thread ID may get reused**
  - **so don't do this!**

Detached Threads

```
start_servers( ) {
  pthread_t thread;
  int i;
  for (i=0; i<nr_of_server_threads; i++) {
    pthread_create(&thread, 0, server, 0);
    pthread_detach(thread);
  }
  ...
}

server( ) {
  ...
}
```

Types

```
pthread_create(&tid,
               0,
               (void *(*)(void *))func,
               (void *)1);

int func = 4; // func definition 1

void func(int i) { // func definition 2
  ...
}

void *func(void *arg) { // func definition 3
  int i = (int)arg;
  ...
  return(0);
}
```

- a function is just an address (of something in the text/code segment)

Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
/* establish some attributes */
...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);
...
```

- thread attribute only needs to be valid when a thread is created
  - therefore, it can be destroyed as soon as the thread is created

Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);
...
pthread_create(&thread, &thr_attr, startroutine, arg);
pthread_attr_destroy(&thr_attr);
```

- **the above code set the stack size to 20MB**
- **the default stack size is very large**
  - **if you need to create a lot of threads, you need to control the stack size**
  - **default stack size is probably around 1MB in Solaris and 8MB in some Linux implementations**

Example

```c
#include <stdio.h>
#include <pthread.h>
#include <string.h>

#define M  3
#define N  4
#define P  5

int A[M][N];
int B[N][P];
int C[M][P];

void *matmult(void *arg) {
  int row = (int)arg, col;
  int i, t;

  for (col=0; col < P; col++) {
    t = 0;
    for (i=0; i<N; i++)
      t += A[row][i] * B[i][col];
    C[row][col] = t;
  }
  return(0);
}
```

```c
main( ) {
  int i;
  pthread_t thr[M];
  int error;

  /* initialize the matrices ... */
  ...
  // create the worker threads
  for (i=0; i<M; i++) {
    if (error = pthread_create(
        &thr[i],
        0,
        matmult,
        (void *)i)) {
      fprintf(stderr,
          "pthread_create: %s",
          strerror(error));
      exit(1);
    }
  }
  // wait for workers to finish
  for (i=0; i<M; i++)
    pthread_join(thr[i], 0)
  /* print the results ... */
}
```

Compiling It

```
% gcc -o mat mat.c -lpthread
```

2.2.3 Synchronization

- In real life, "synchronization" means that you want to do things at the same time
- In computer science, "synchronization" could meant the above, ***OR***, it means that you want to ***prevent*** do things at the same time

Mutual Exclusion

Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
```

Thread 2:

```
x = x+1;
```

- looks like it doesn’t matter how you execute, x will be incremented by 2 in the end
  - choices are
    - thread 1 executes x = x+1 then thread 2 executes x = x+1
    - thread 2 executes x = x+1 then thread 1 executes x = x+1
  - are there other choices?

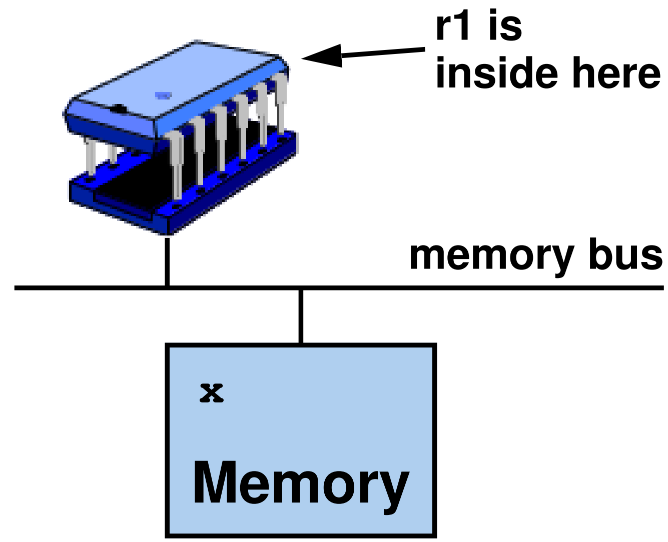
Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
   ld    r1,x
   add   r1,1
   st    r1,x
   */
```

Thread 2:

```
x = x+1;
 /*
   ld    r1,x
   add   r1,1
   st    r1,x
   */
```



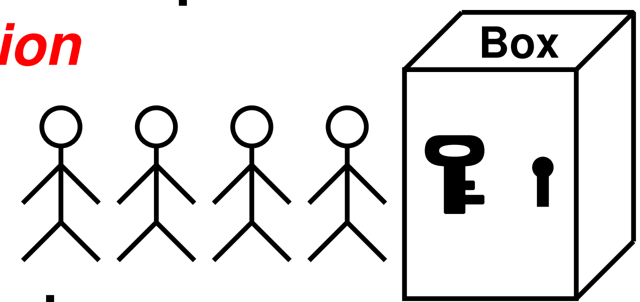
- Unfortunately, machines do not execute high-level language statements
  - they execute machine instructions
  - now if thread 1 executes the first (or two) machine instructions
  - context switch!
    - how can this happen?
  - then thread 2 executes all 3 machine instructions
  - then later thread 1 executes the remaining machine instructions
  - x would have only increased by 1

Threads and Synchronization

```
// shared by both threads
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int x;
...
pthread_mutex_lock(&m);
x = x+1;
pthread_mutex_unlock(&m);
```

critical section

Locking a mutex is like getting the key to a safe-deposit box



- code between `pthread_mutex_lock()` and `pthread_mutex_unlock()` for a particular *mutex* is called a *critical section with respect to that mutex*
  - *all* the critical sections *with respect to a particular mutex* are *"mutually exclusive"*
    - the system (not necessarily the OS) guarantees that only *one* critical section can be executing at any point in time
  - how it's really done will be covered in Ch 5

Set Up

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

- If a mutex cannot be initialized statically, do:

```
int pthread_mutex_init(
        pthread_mutex_t *mutexp,
        pthread_mutexattr_t *attrp)

int pthread_mutex_destroy(
        pthread_mutex_t *mutexp)
```

- Usually, mutex attributes are not used

Taking Multiple Locks

- Mutex is not a cure-all
  - when you have more than one locks, you may get into trouble

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

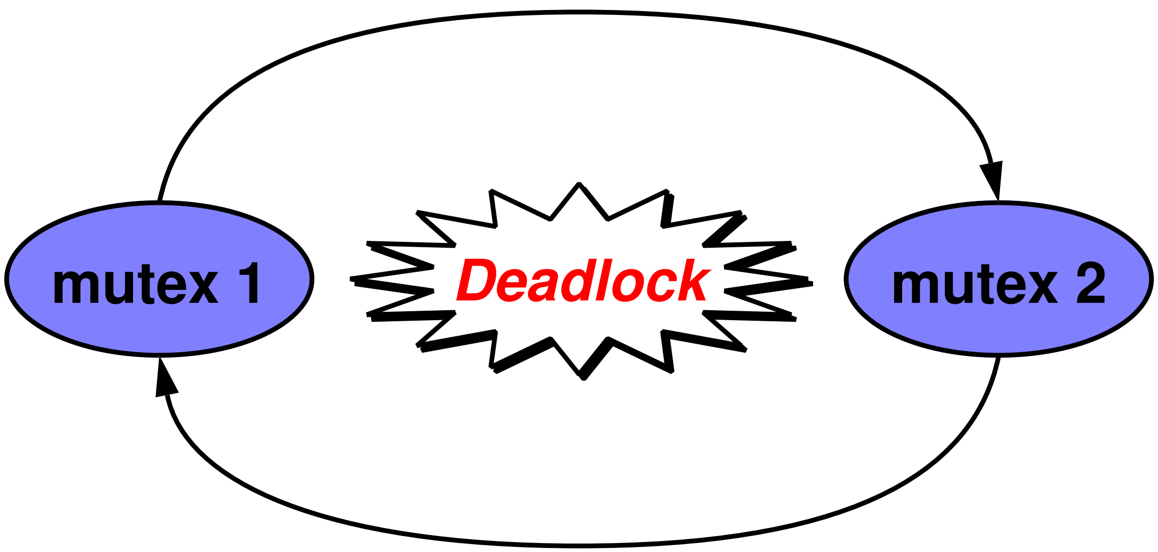
```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Taking Multiple Locks

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
proc2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

- Graph representation ("wait-for" graph)



Necessary Conditions For Deadlocks

- All 4 conditions below must be met in order for a deadlock to be possible (no guarantee that a deadlock may occur)

1) Bounded resources
   - only a finite number of threads can have concurrent access to a resource

2) Wait for resources
   - threads wait for resources to be freed up, without releasing resources that they hold

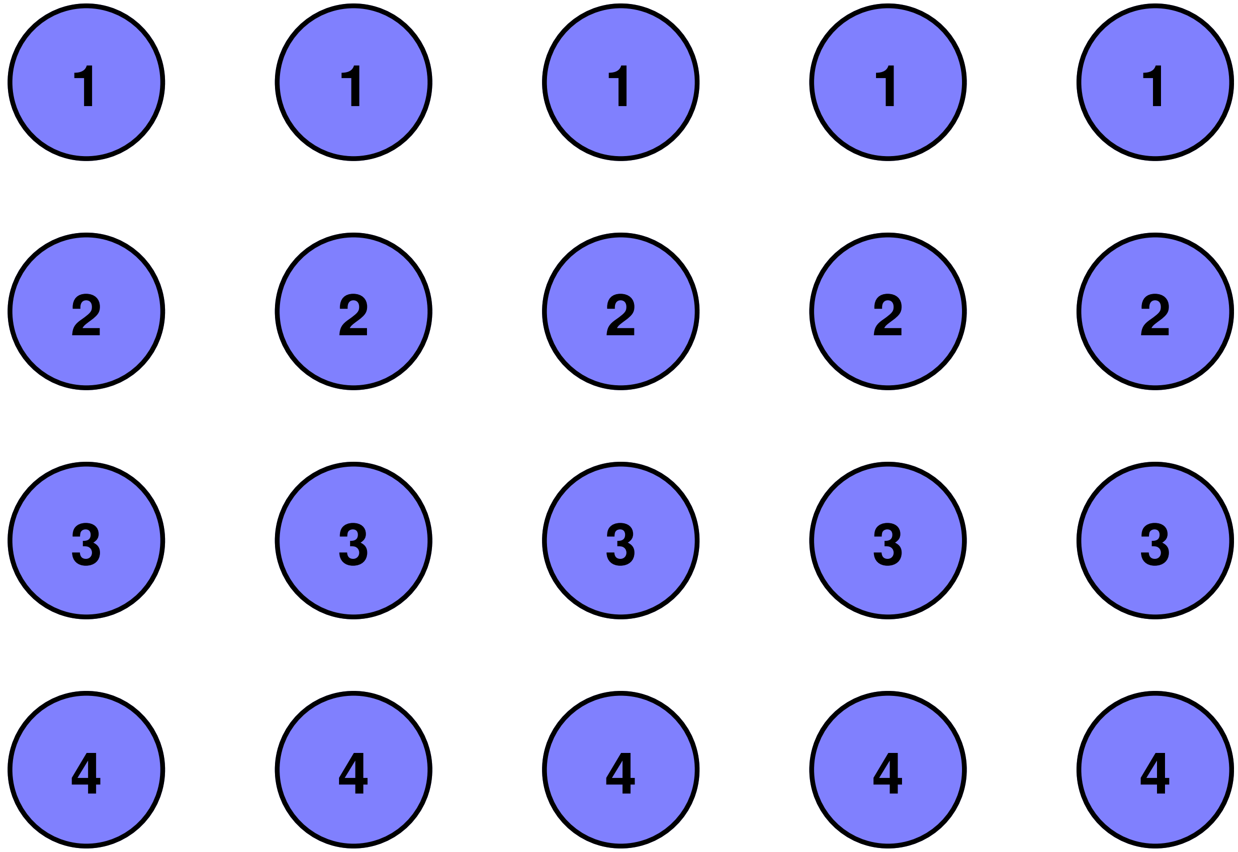
3) No preemption
   - resources cannot be *revoked* from a thread

4) Circular wait
   - there exists a set of waiting threads, such that each thread is waiting for a resource held by another

Dealing with Deadlock

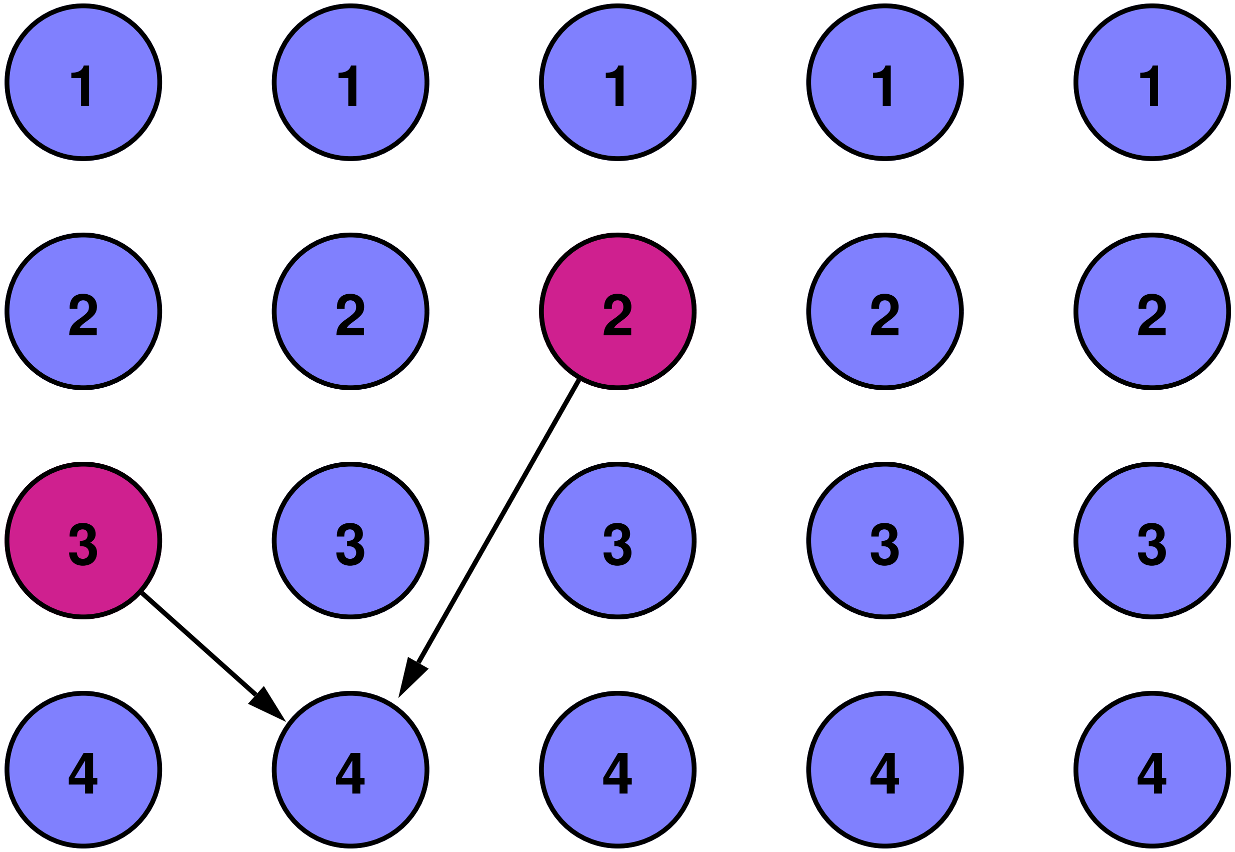
- Deadlock is a programming bug
  - one of the oldest bug
  - it's a tricky one because it only deadlocks *sometimes*
- Hard
  - is the system deadlocked?
  - will this move lead to deadlock?
  - this is *detection*
    - if you can detect deadlocks, what do you do after you have detected them?
- Easy
  - restrict use of mutexes so that deadlock cannot happen
  - this is *prevention*
- Deadlock is a complicated subject
  - some textbooks spend an entire chapter on deadlocks
  - we will only look at a couple of cases

Deadlock Prevention: Lock Hierarchies



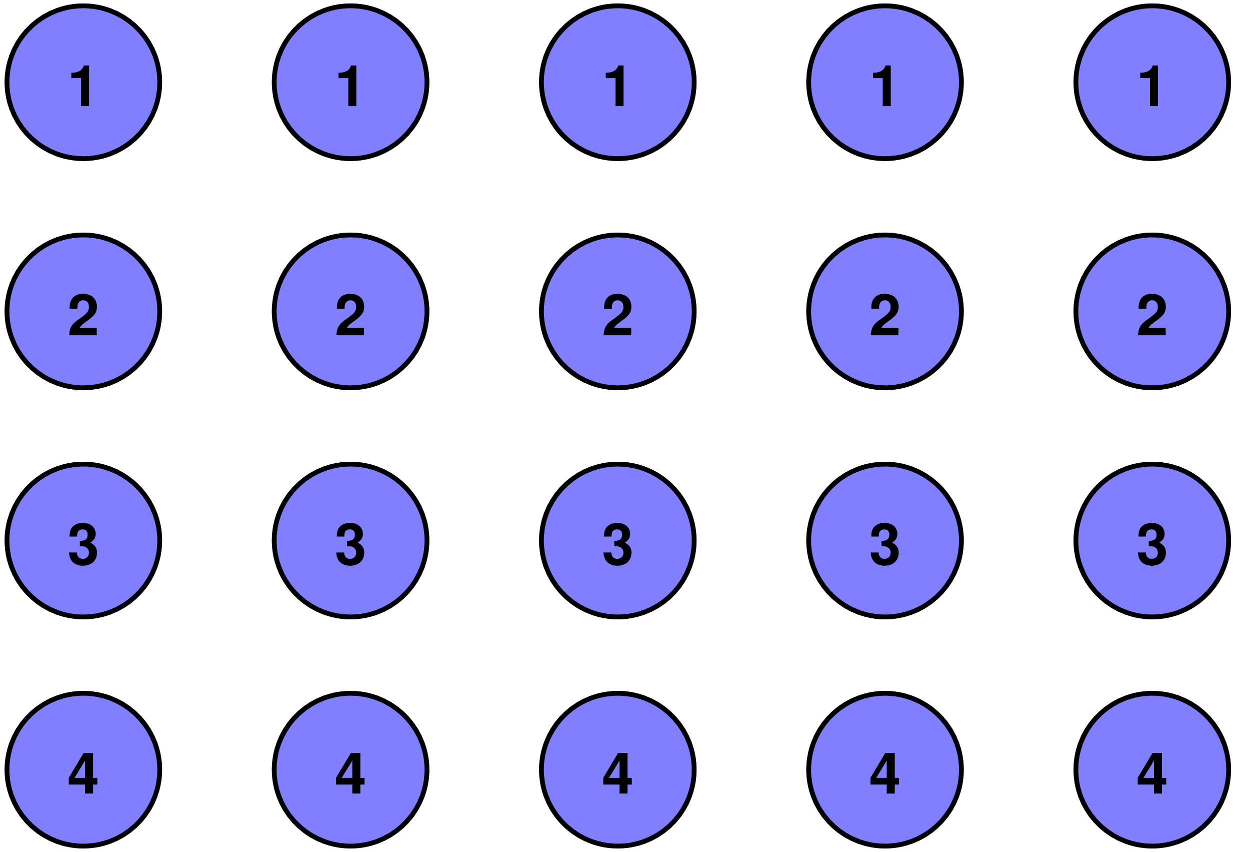
- organize mutexes into levels
- must not try locking a mutex at level i if already holding a mutex at level j if $i \le j$, otherwise it's okay
  - e.g., if hold mutexes at levels 2 and 3, can only wait for a mutex at levels 4 or higher

Deadlock Prevention: Lock Hierarchies



- organize mutexes into levels
- must not try locking a mutex at level *i* if already holding a mutex at level *j* if $i \leq j$, otherwise it's okay
  - e.g., if holding mutexes at levels 2 and 3, can only wait for a mutex at levels 4 or higher

Deadlock Prevention: Lock Hierarchies



- What if you cannot organize your mutexes in such strict order for deadlock detection?

Deadlock Prevention: Conditional Locking

```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}

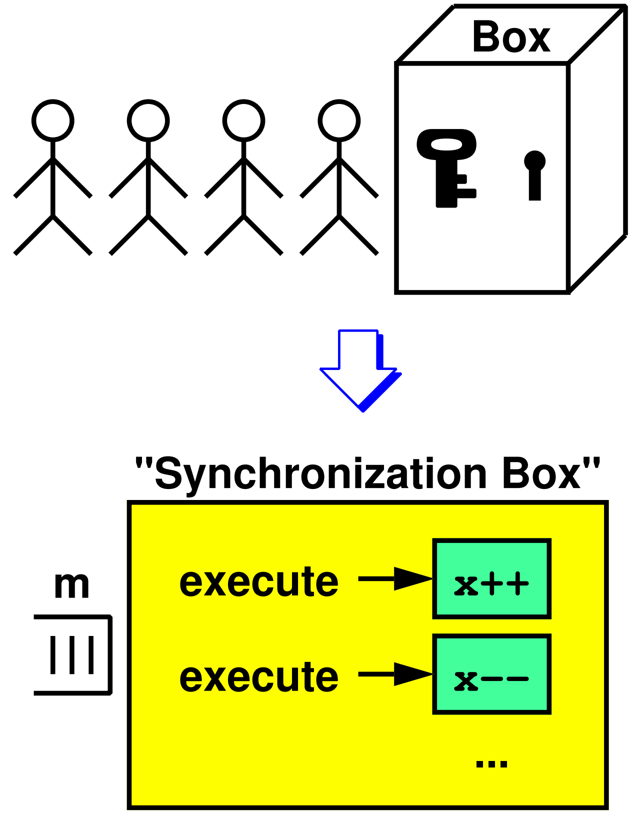
proc2( ) {
  while (1) {
    pthread_mutex_lock(&m2);

    if (!pthread_mutex_trylock(&m1))
      break;
    pthread_mutex_unlock(&m2);
  }
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

One Mutex, Multiple Critical Sections

```
f1() {
  pthread_mutex_lock(&m);
    x++;      critical section
  pthread_mutex_unlock(&m);
}

f2() {
  pthread_mutex_lock(&m);
    x--;      critical section
  pthread_mutex_unlock(&m);
}
```

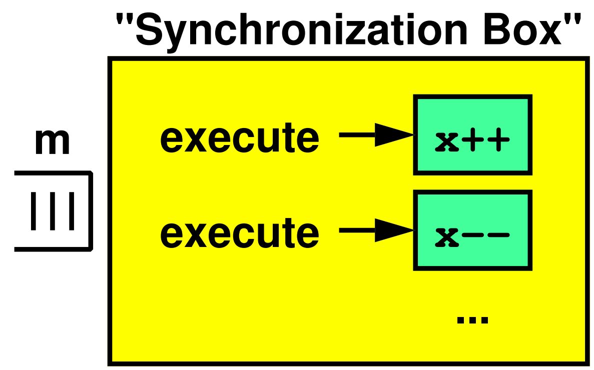


- I use "Synchronization Box" to mean executing one of many critical section code with respect to one particular mutex

One Mutex, Multiple Critical Sections

```
f1() {
  pthread_mutex_lock(&m);
  x++;          } critical section
  pthread_mutex_unlock(&m);
}

f2() {
  pthread_mutex_lock(&m);
  x--;          } critical section
  pthread_mutex_unlock(&m);
}
```



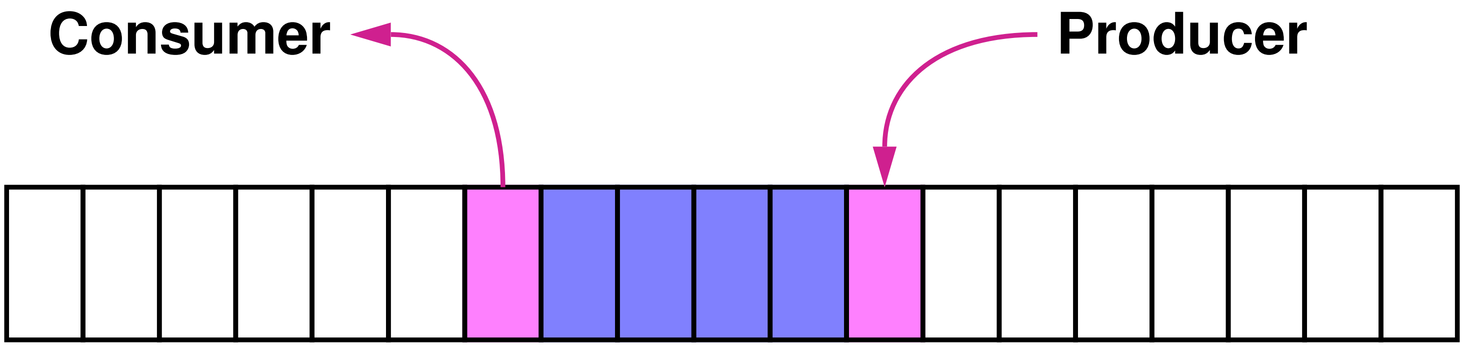
- I use "Synchronization Box" to mean executing one of many critical section code with respect to one particular mutex

⇨ By calling `pthread_mutex_lock(&m)`, a thread can be placed into a ***queue*** and ***wait*** there ***indefinitely*** for mutex `m` to become available
- multiple threads would join this queue
- queue is served one at a time, like a supermarket checkout
- when it's your thread's turn, `pthread_mutex_lock()` returns with the mutex locked, your thread can execute critical section code, and then release the mutex

Beyond Mutexes

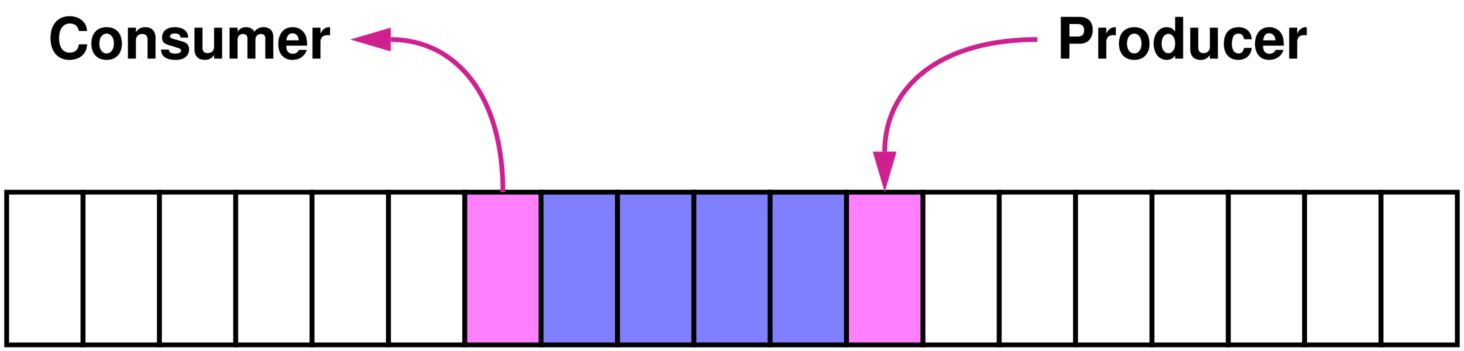
- **Mutex is necessary when shared data is being modified**
  - **although there are cases where using a mutex is an overkill (i.e., too restrictive and inefficient and would lock threads out when it's not necessary)**
    - **we would like to have better concurrency *(i.e., "fine-grained parallelism")* when complete mutual exclusion in not required**
  - **two major categories to illustrate this**
    1) **what if threads don't interfere one another most of the time and synchronization is only required occasionally?**
       - **e.g., *Producer-Consumer problem* (a.k.a., bounded-buffer problem)**
    2) **what if some threads just want to *look at* *(i.e., read)* a piece of data?**
       - **e.g., *Readers-Writers problem***
- **Barrier Synchronization**

Producer-Consumer Problem



- A circular buffer is used
- Most of the time, no interference
  - if you use a *single mutex* to lock the entire array of buffers, it's an overkill *(i.e., too inefficient)*
- When does it require synchronization?

Producer-Consumer Problem



- A circular buffer is used
- Most of the time, no interference
  - if you use a *single mutex* to lock the entire array of buffers, it's an overkill *(i.e., too inefficient)*
- When does it require synchronization?
  - producer needs to be blocked when all slots are full
  - consumer needs to be blocked when all slots are empty

Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

$\}$ *command sequence*

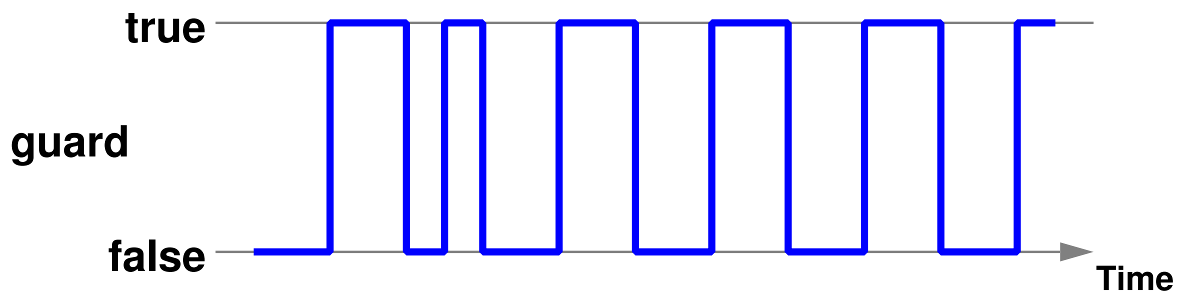
- this means that the command sequence *can be* executed (*atomically*) *at any time* the *guard* is evaluated to be *true*
  - a *guard* is a *boolean expression* (evaluates to true or false)
  - *atomically* mean that it's executed without interruption
    - *evaluating the guard* and *executing the command sequence* altogether is an *atomic operation* *if* the *guard is true*
    - you cannot evaluate the guard if your thread is not running

- For exams, you need to know how to write simple pesudo-code in the language of *Guarded Commands*

Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

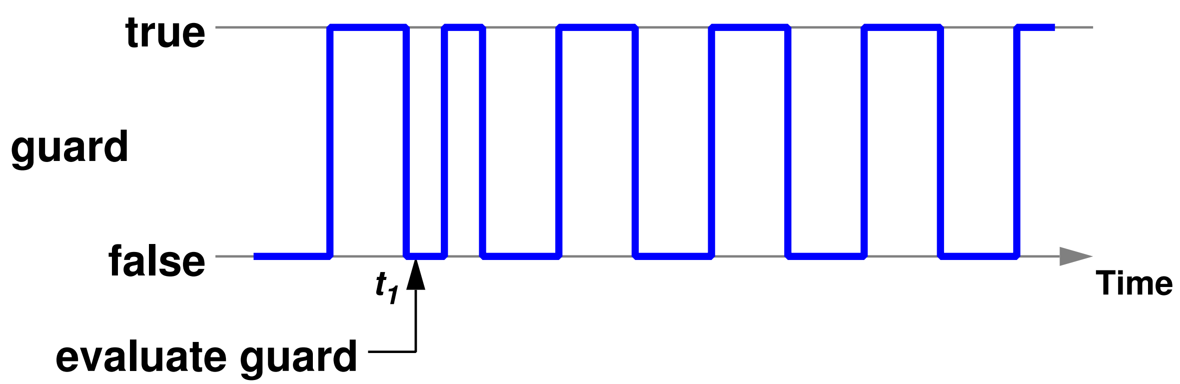
command sequence



Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

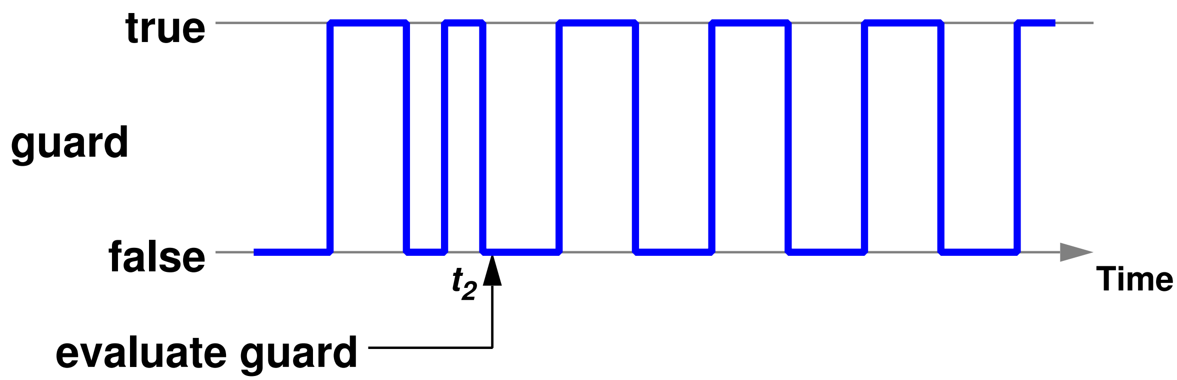
command sequence



Guarded Commands

```
when (guard) [
  /*
     once the guard is true,
     execute this code atomically
   */
  ...
]
```

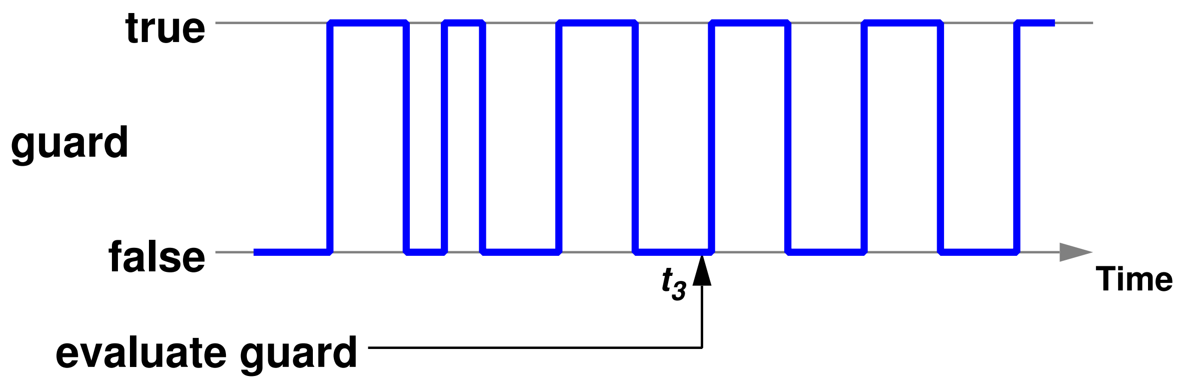
command sequence



Guarded Commands

```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
 ...
]
```

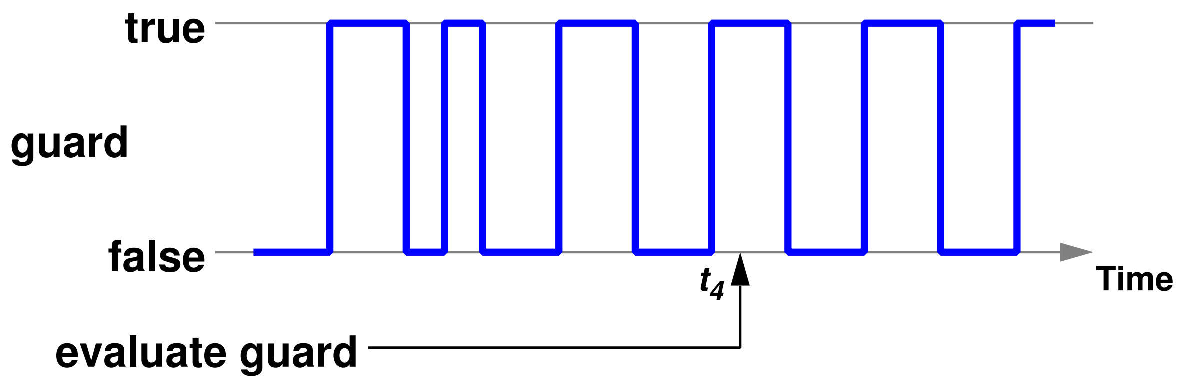
command sequence



Guarded Commands

```
when (guard) [
 /*
   once the guard is true,
   execute this code atomically
  */
 ...
]
```

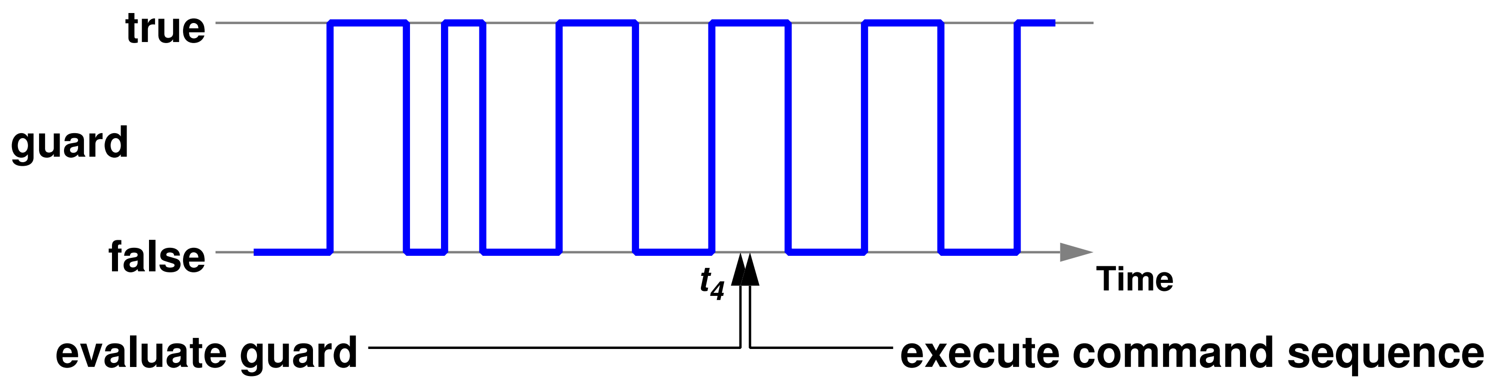
command sequence



Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

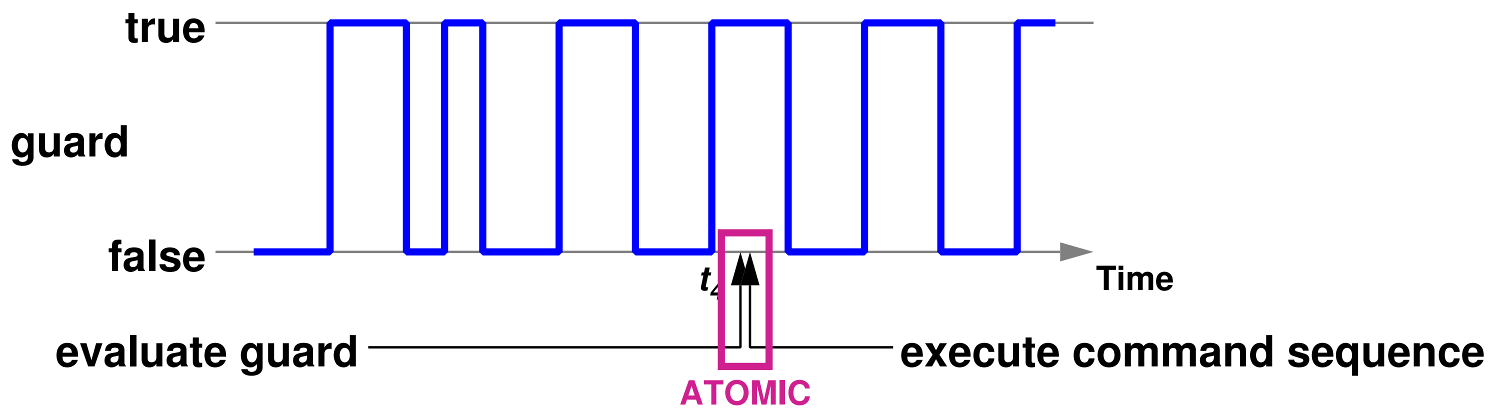
command sequence



Guarded Commands

```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

command sequence

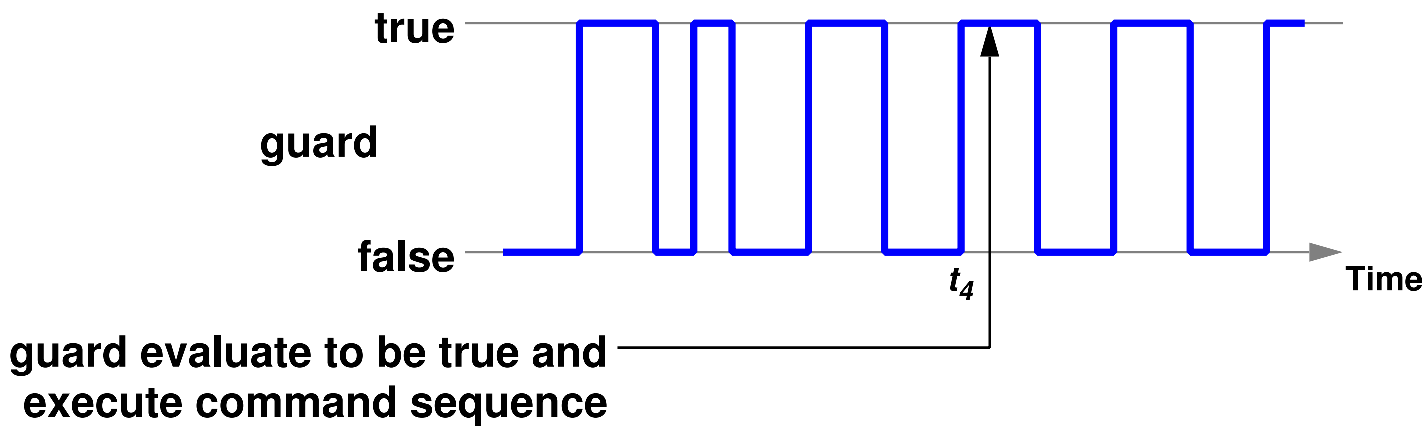


- please understand that *command sequence ≠ critical section*
  - evaluate the guard to be true *and* execute command sequence *together* is done inside one critical section

Guarded Commands

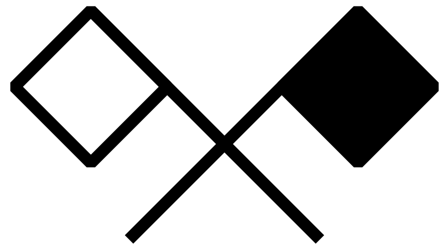
```
when (guard) [
  /*
    once the guard is true,
    execute this code atomically
   */
  ...
]
```

command sequence



- ***atomic:*** as if it's executed in an instance of time (duration = 0)
  - this is okay because it's just pseudo-code

Semaphores



- A *semaphore*, `S`, is a *nonnegative integer* on which there are exactly two operations defined by two garded commands
  - `P(S)` operation (implemented as a guarded command):
    - ```
      when (S > 0) [
        S = S - 1;
      ]
      ```
  - `V(S)` operation (implemented as a guarded command):
    - `[S = S + 1;]`
  - there are no other means for manipulating the value of `S`
    - other than initializing it
      ```

Mutexes with Semaphores

```
semaphore S = 1;

void OneAtATime( ) {
  P(S);
  ...
  /* code executed mutually
     exclusively */
  ...
  V(S);
}
```

- `P(S)` operation:
  - `when (S > 0) [`
    `S = S - 1;`
    `]`
- `V(S)` operation:
  - `[S = S + 1;]`

- this is known as a *binary semaphore*

Implement A Mutex With A Binary Semaphore

- Instead of doing

```
pthread_mutex_lock(&m);
x = x+1;
pthread_mutex_unlock(&m);
```

  - do:

```
S = 1;
P(S);
x = x+1;
V(S);
```

- So, you can lock a data structure using a binary semaphore
  - this looks just like mutex, what have we really gained?
    - if you use it this way, nothing

Mutexes with Semaphores

```
semaphore S = N;

void NAtATime( ) {
  P(S);
  ...
  /* no more than N threads
     here at once */
  ...
  V(S);
}
```

- `P(S)` operation:
  - `when (S > 0) [`
    `S = S - 1;`
    `]`
- `V(S)` operation:
  - `[S = S + 1;]`

- this is known as a *counting semaphore*
- can be used to solve the producer-consumer problem

Main difference between a semaphore and a mutex

- if a thread locks a mutex, it's holding the lock
  - therefore, it must be that thread that unlocks that mutex
- one thread performs a `P` operation on a semaphore, *another thread* performs a `V` operation on the same semaphore
  - this is often why you would use a semaphore

Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

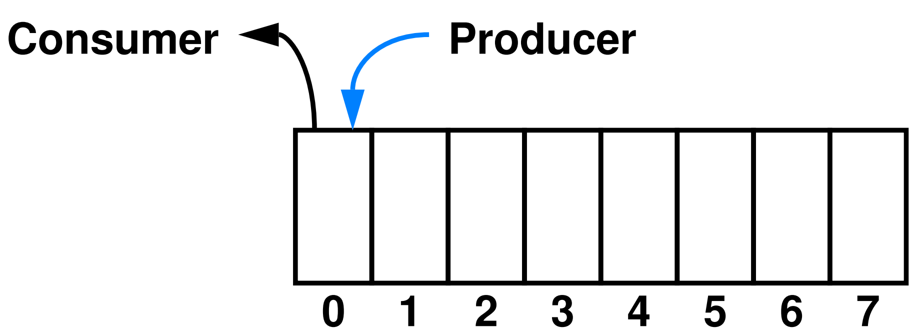
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	8
occupied	0
nextin	0
nextout	0



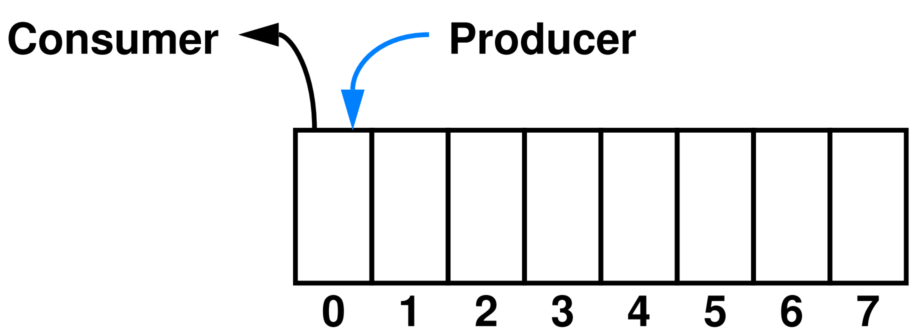
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	8
occupied	0
nextin	0
nextout	0



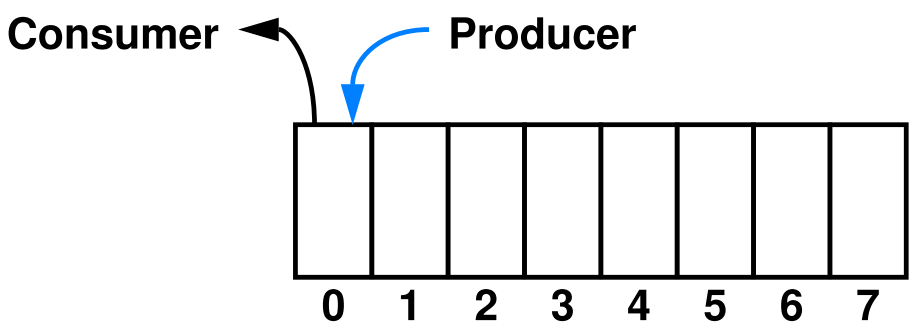
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	0
nextout	0



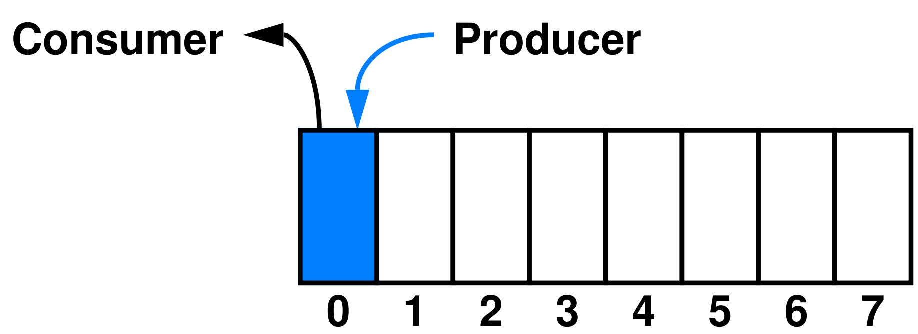
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	0
nextout	0



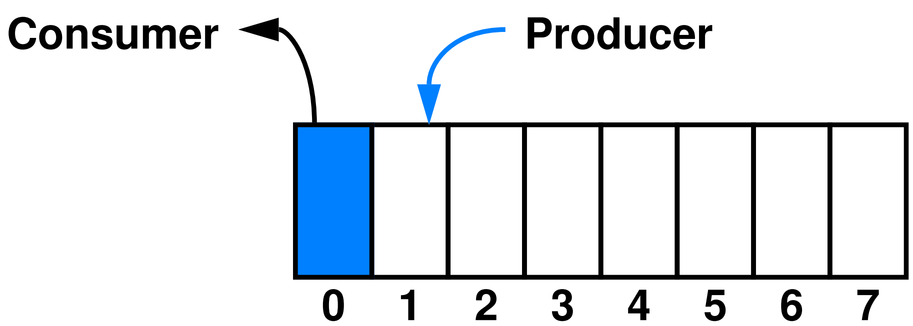
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	1
nextout	0



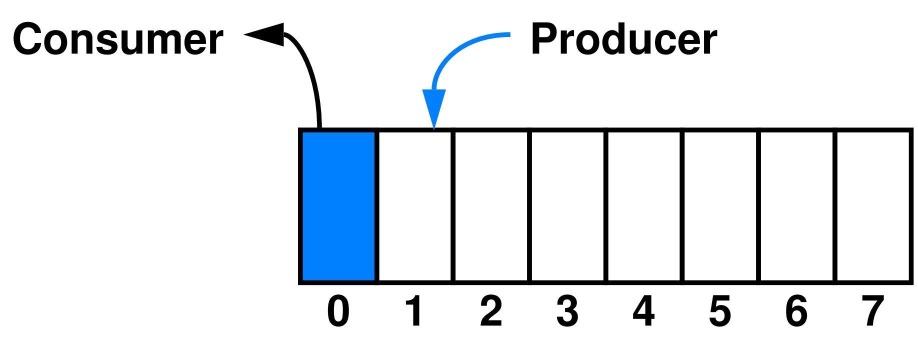
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	1
nextout	0



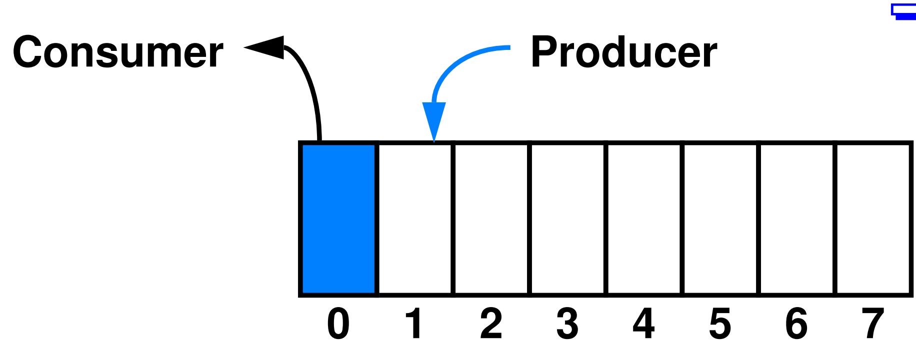
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	1
nextin	1
nextout	0



- note: producer continue to produce

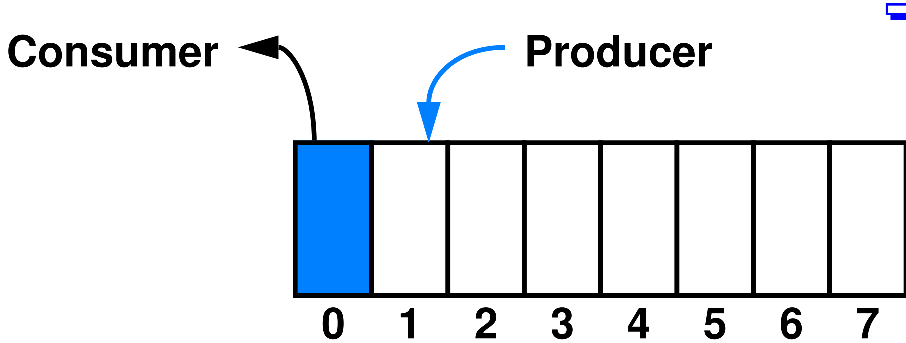
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	1
nextout	0



- note: producer continue to produce

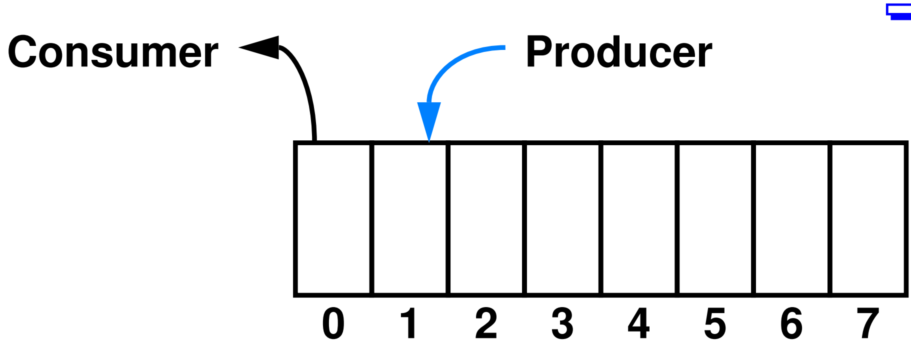
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	1
nextout	0



- note: producer continue to produce

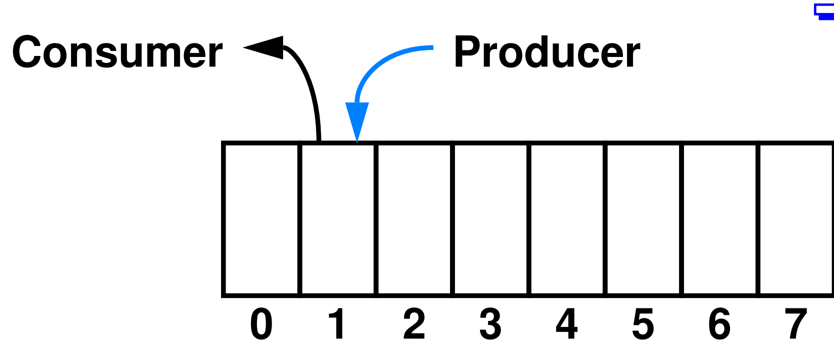
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	1
nextout	1



- note: producer continue to produce

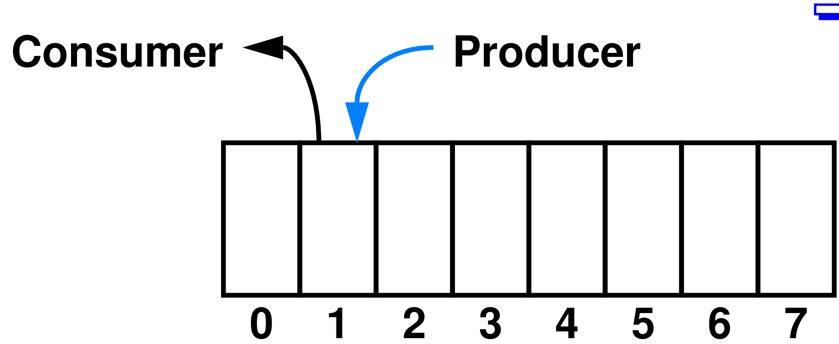
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	7
occupied	0
nextin	1
nextout	1



- note: producer continue to produce

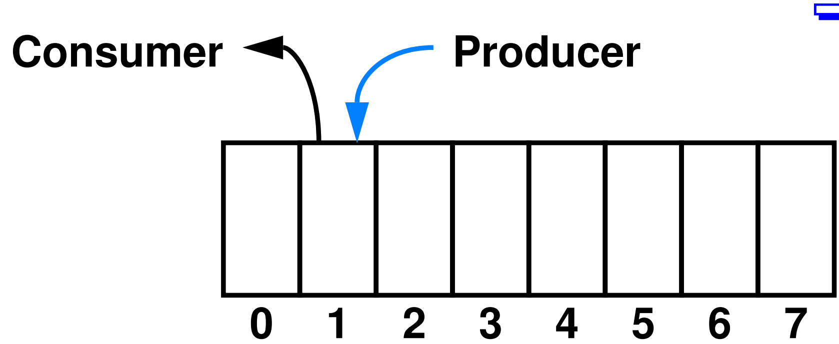
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	8
occupied	0
nextin	1
nextout	1



- note: producer continue to produce

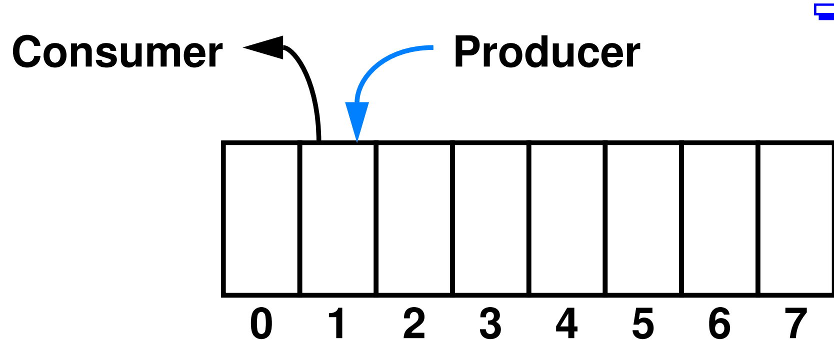
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

empty	8
occupied	0
nextin	1
nextout	1



- note: producer continue to produce

Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

- if produce and consume at same rate, no one waits
- if producer is fast and consumer slow, producer may wait
- if consumer is fast and producer slow, consumer may wait

Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

- ***Mutex*** by itself is more "***coarse grain***"
  - you may use one mutex to control access to the number of empty and occupied cells, `nextin`, and `nextout`
- ***Semaphore*** is more "***fine grain***"
  - but ***not general*** enough

POSIX Semaphores

```
#include <semaphore.h>
sem_t semaphore;
int   err;
int   pshared = 0;     // not shared among processes
int   init_value = B; // initial value

err = sem_init(&semaphore, pshared, init_value);
err = sem_destroy(&semaphore);
err = sem_wait(&semaphore);    /* P operation */
err = sem_trywait(&semaphore); /* conditional P
                                  operation
                                */
err = sem_post(&semaphore);    /* V operation */
```

Producer-Consumer with POSIX Semaphores

```
void produce(char item) {
  sem_wait(&empty);
  buf[nextin++] = item;
  if (nextin >= B)
    nextin = 0;
  sem_post(&occupied);
}
```

```
char consume() {
  char item;
  sem_wait(&occupied);
  item = buf[nextout++];
  if (nextout >= B)
    nextout = 0;
  sem_post(&empty);
  return(item);
}
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

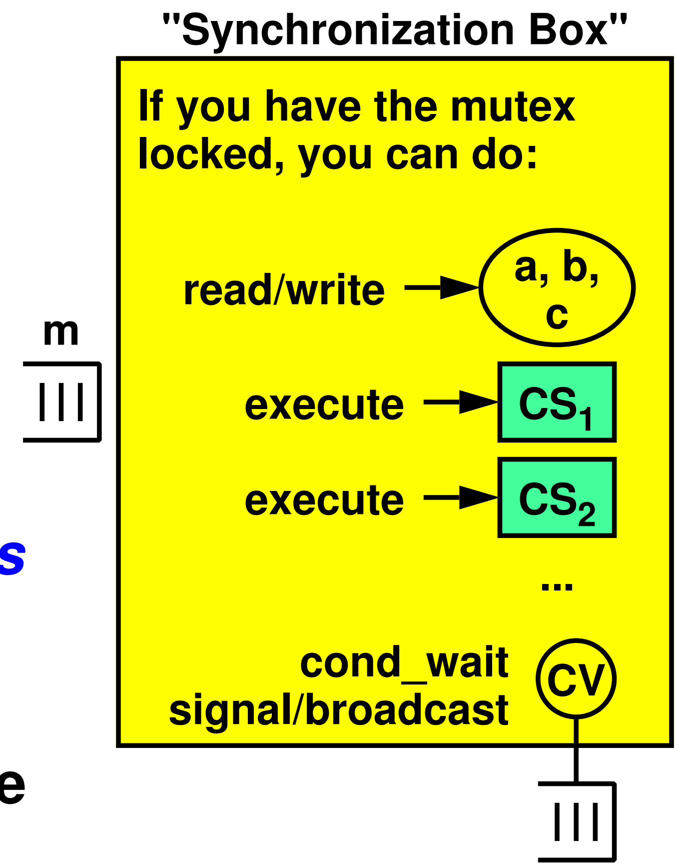
Implementation Of Guarded Commands

```
when (guard) [
  /* command sequence */
  ...
]
```

- In general, the *guard* can be *complicated* and involving the evaluation of several variables (e.g., `a > 3 && f(b) <= c`)
  - the guard (which evaluates to either true or false) *keeps changing its value*, *continuously* and by multiple threads *simultaneously*
  - how can we "capture" the instance of time when it evaluates to true so we can execute the command sequence atomically?
    - we have to "sample" it, i.e., take snap shot of all the variables that are involved and then evaluate it
    - a mutex is involved, but how?
      - need to be efficient
    - need something else *(known as condition variables)*
    - need a bunch of *rules* to follow

Implementation Of Guarded Commands

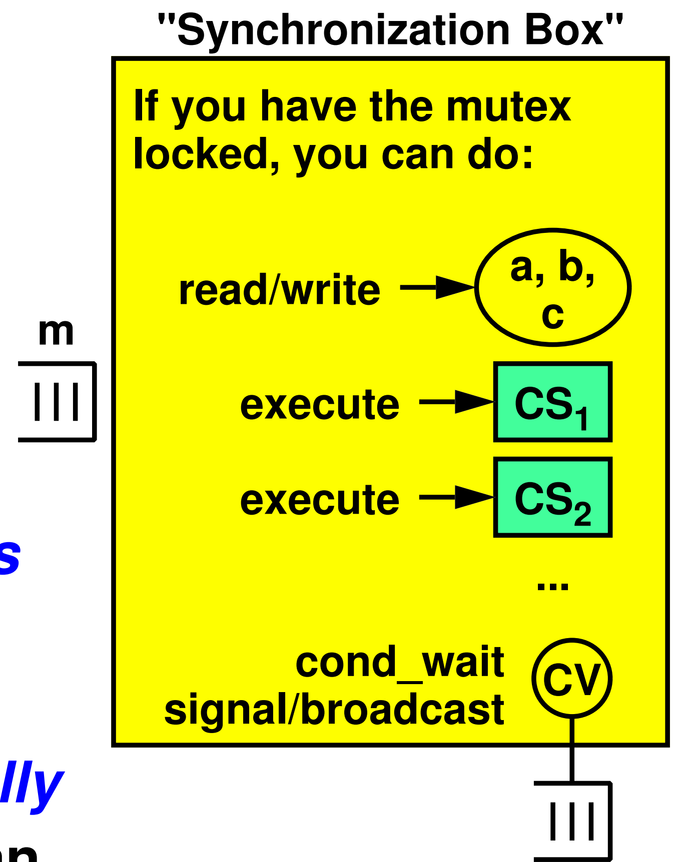
```
when (guard) [
  /* command sequence */
  . . .
]
```



- POSIX provides *condition variables* for programmers to implement guarded commands
  - a *condition variable* is a *queue of threads* waiting for some sort of notification *(an "event" or "condition")*
    - threads, waiting for a guard to become true, join such a queue
      - they wait for a specific *condition* to be *signaled*
      - they wait for *the right time* to *evaluate the guard*
    - (cont...)

Implementation Of Guarded Commands

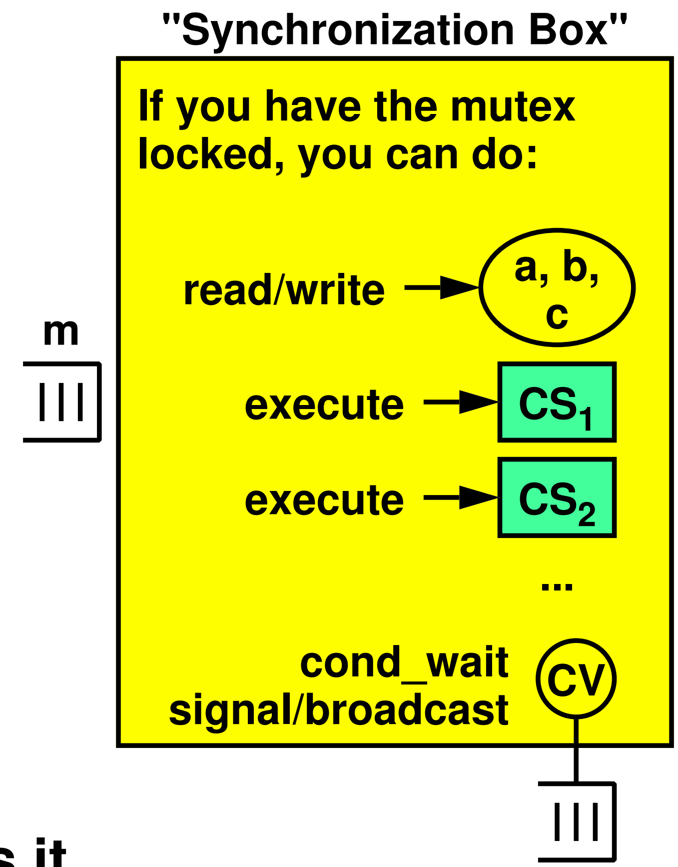
```
when (guard) [
  /* command sequence */
  . . .
]
```



- POSIX provides *condition variables* for programmers to implement guarded commands
  - a *condition variable* is a *queue of threads* waiting for some sort of notification *(an "event" or "condition")*
    - threads that do something to *potentially* change the truth value of the guard can then wake up the threads that were waiting in the queue
      - they can *signal* or *broadcast* the *condition*
      - *no guarantee* that the guard will be true when it's time for *another thread* to evaluate the guard

Implementation Of Guarded Commands

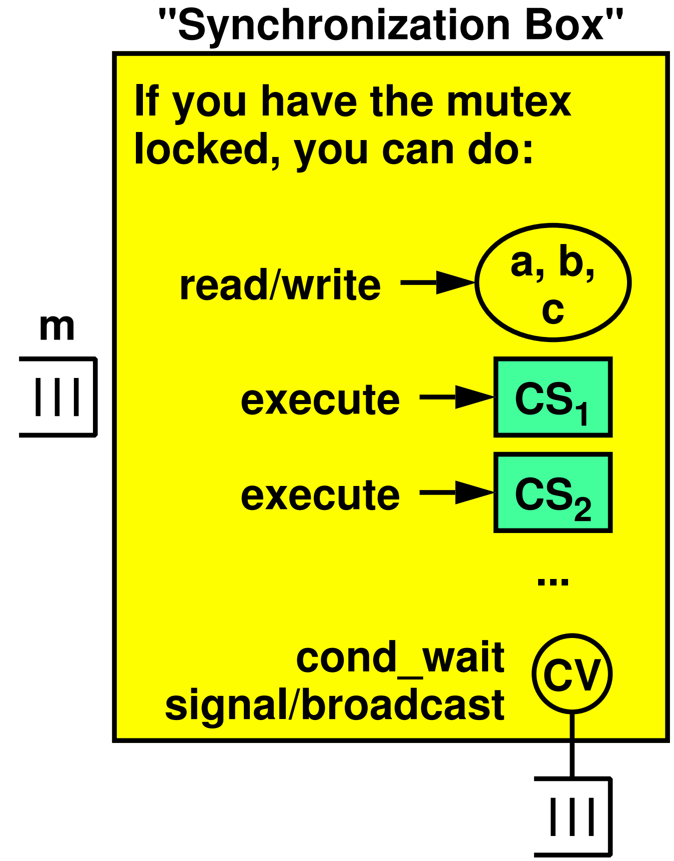
```
when (guard) [
  /* command sequence */
  . . .
]
```



- POSIX provides *condition variables* for programmers to implement guarded commands
  - conceptually, an "event" (signaling/broadcasting of a condition) happens in an instance of time (duration of this "event" is zero)
    - if you are not waiting for it, you'll miss it
    - how do you make sure you won't miss an event?
      - you have to *follow the protocol* (for multiple interacting threads to follow) described here

Implementation Of Guarded Commands

```
when (guard) [
  /* command sequence */
  ...
]
```



- POSIX provides *condition variables* for programmers to implement guarded commands

1) `pthread_cond_wait(`
```
        pthread_cond_t *cv,
        pthread_mutex_t *mutex)
```

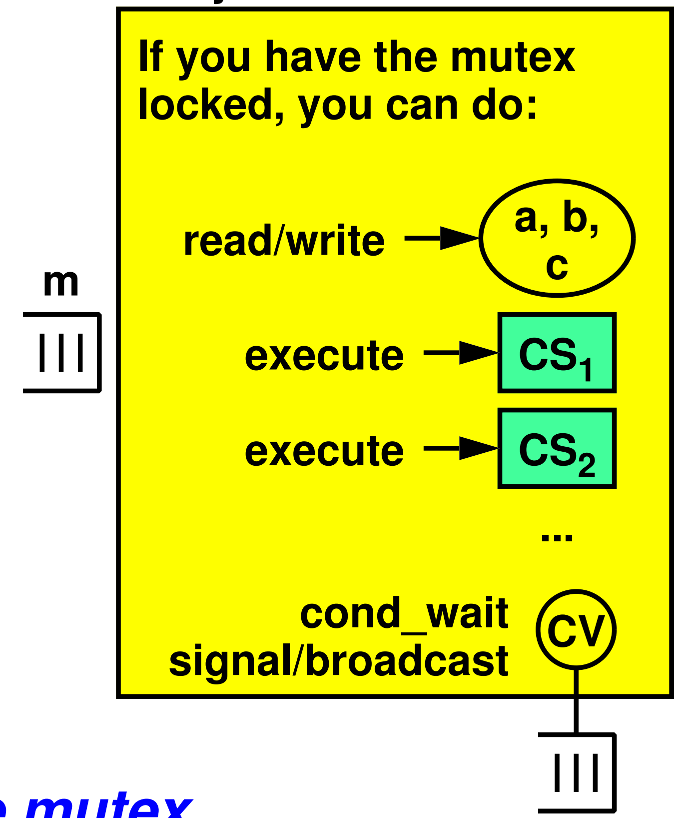
- should only call `pthread_cond_wait()` if you have the `mutex` *locked*
- *atomically* unlocks `mutex` and wait for the "event"
- when the event is signaled/broadcasted, `pthread_cond_wait()` returns with the `mutex` *locked*

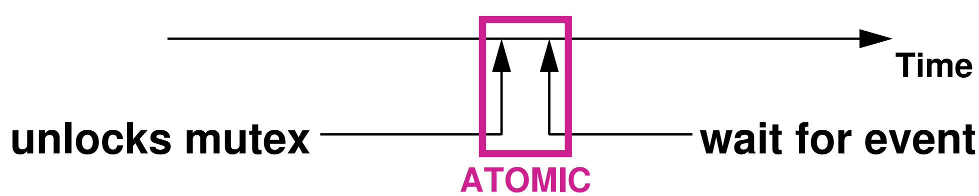
Implementation Of Guarded Commands

```
when (guard) [
  /* command sequence */
  ...
]
```

- POSIX provides *condition variables* for programmers to implement guarded commands
  - *atomically* unlocks `mutex` and wait for the "event"
    - with respect to the *operation of the mutex*

"Synchronization Box"



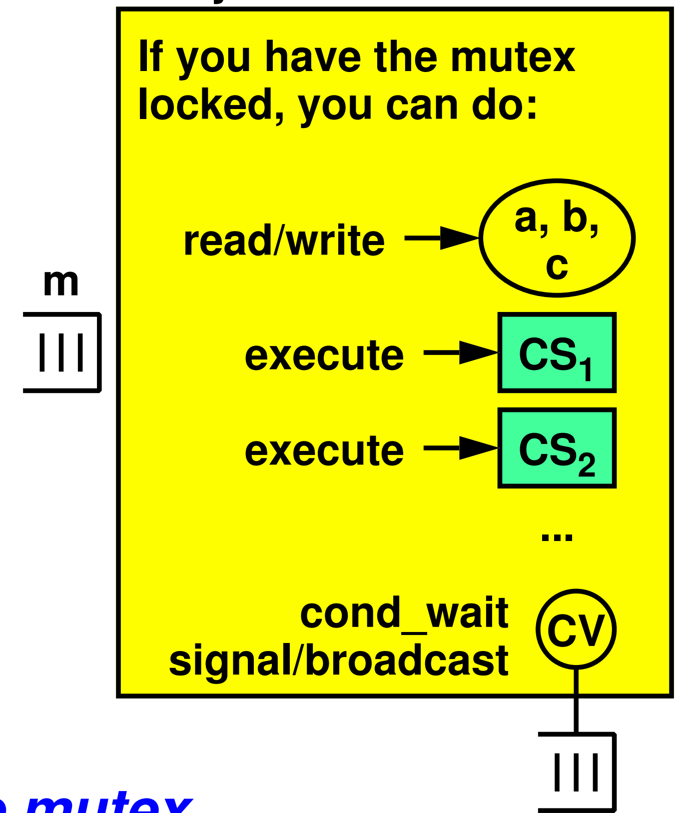


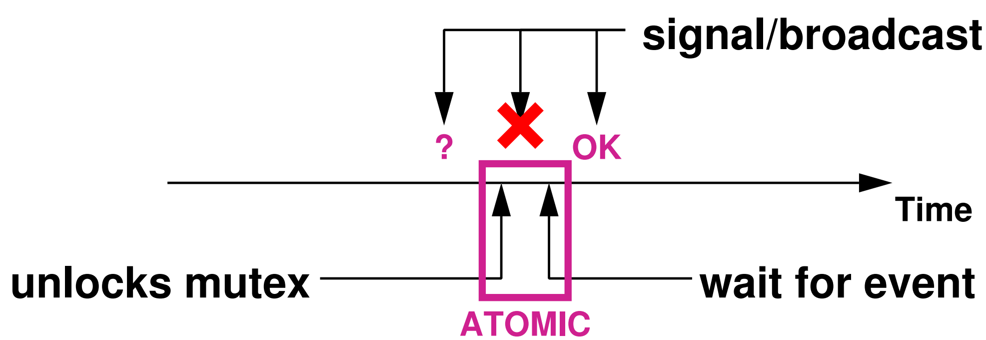
Implementation Of Guarded Commands

```
when (guard) [
  /* command sequence */
  . . .
]
```

- POSIX provides *condition variables* for programmers to implement guarded commands
  - *atomically* unlocks `mutex` and wait for the "event"
    - with respect to the *operation of the mutex*

"Synchronization Box"

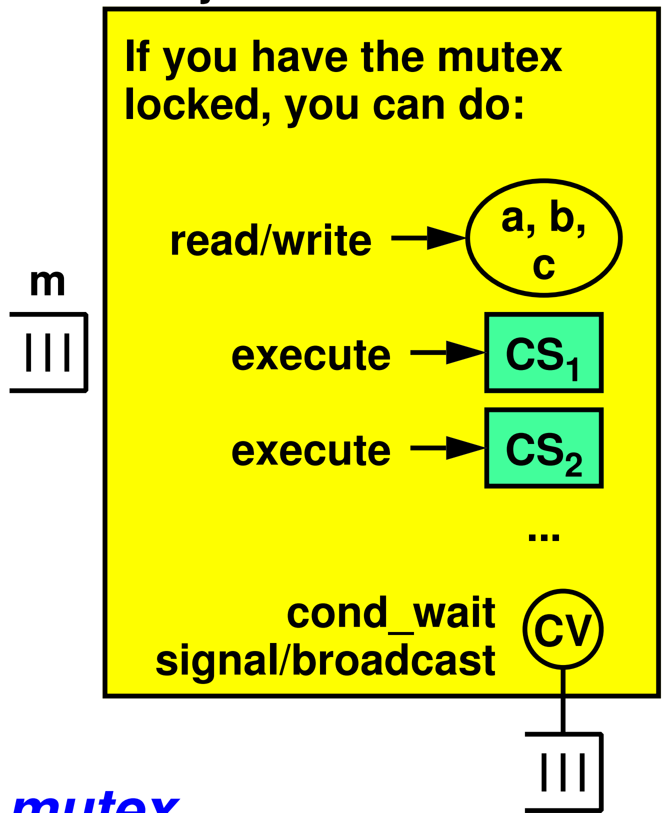




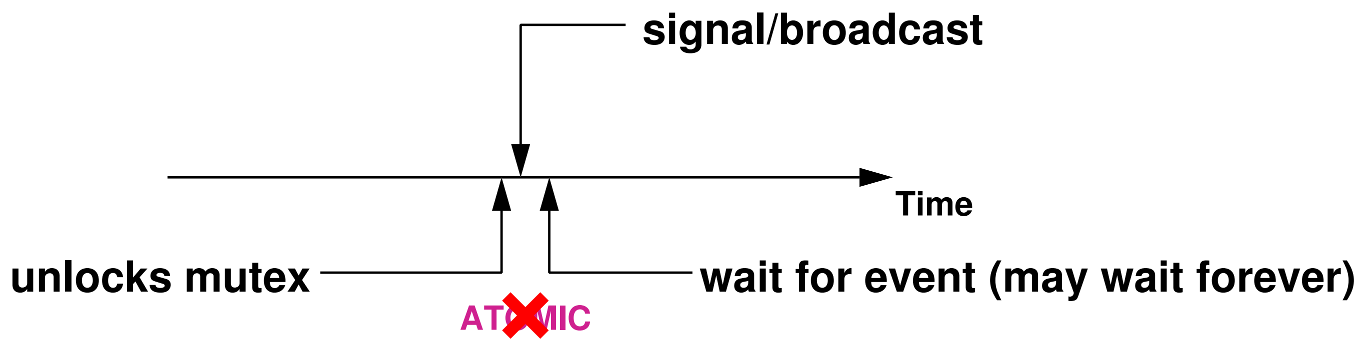
Implementation Of Guarded Commands

```
when (guard) [
  /* command sequence */
  . . .
]
```

"Synchronization Box"



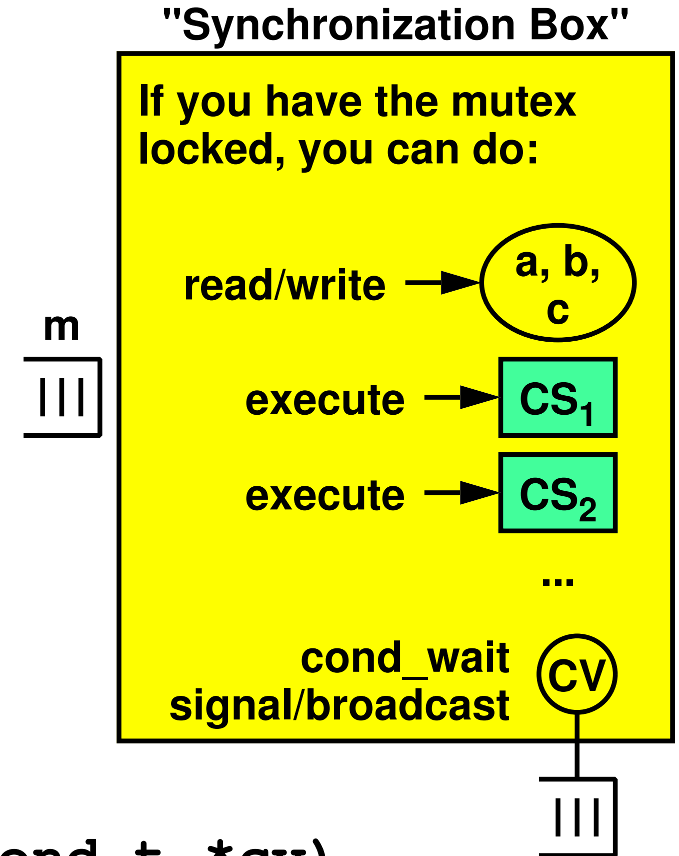
- **POSIX provides *condition variables* for programmers to implement guarded commands**
  - ***atomically* unlocks `mutex` and wait for the "event"**
    - **with respect to the *operation of the mutex***



Implementation Of Guarded Commands

```
when (guard) [
  /* command sequence */
  ...
]
```

- POSIX provides *condition variables* for programmers to implement guarded commands

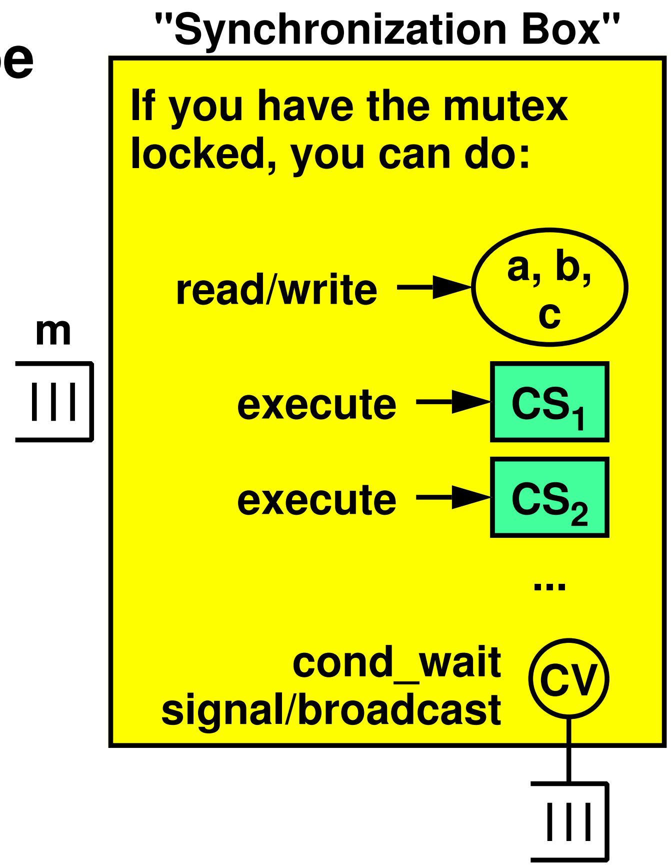


2) `pthread_cond_broadcast(pthread_cond_t *cv)`
   `pthread_cond_signal(pthread_cond_t *cv)`

   - should only call `pthread_cond_broadcast()` or `pthread_cond_signal()` if you have the corresponding mutex *locked*

Implementation Of Guarded Commands

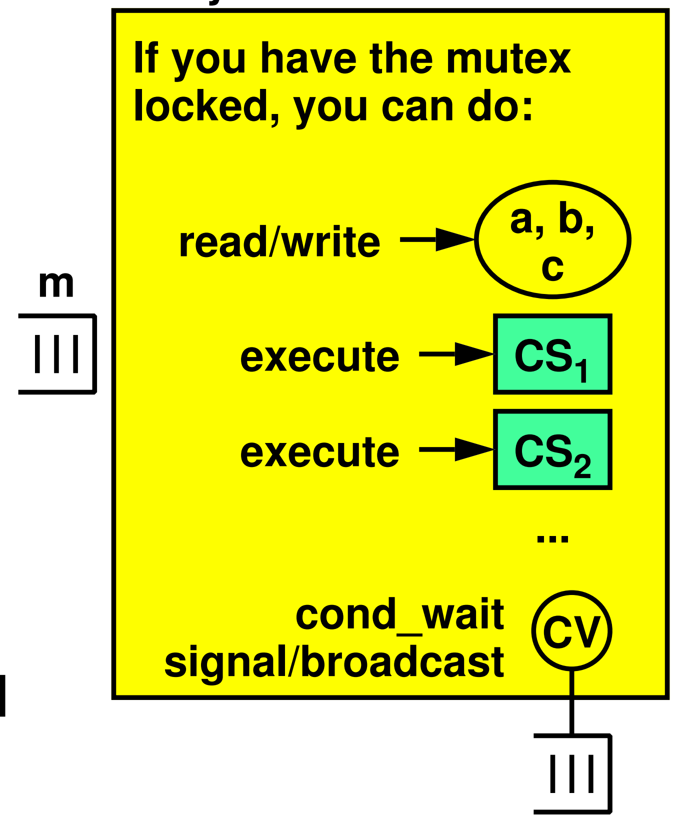
- ***Synchronization:*** **mutex, condition variables, guards, critical sections**
  - with respect to a mutex, a thread can be
    - waiting in the mutex queue
    - got the lock and inside the "synchronization box"
      - only one thread can be inside the "synchronization box"
    - waiting in the CV queue
    - or outside
  - with respect to a mutex, a, b, c are variables that can affect the value of the guard
    - can only access (i.e., read/write) them if a thread is inside the "synchronization box" (i.e., has the mutex locked)



Implementation Of Guarded Commands

- ***Synchronization:*** **mutex, condition variables, guards, critical sections**
  - **when you *signal* CV**
    - ***one* thread in the CV queue gets moved to the mutex queue**
  - **when you *broadcast* CV**
    - ***all* threads in the CV queue get moved to the mutex queue**
  - **you can only get *added* to the CV queue if you have the mutex locked**
  - **you can only *modify* the variables in the guard if you have the mutex locked**
  - **you can only *read* the variables in the guard (i.e., evaluate the guard) if you have the mutex locked**
  - **you can only *execute* critical section code if you have the mutex locked**

"Synchronization Box"



POSIX Condition Variables

<table>
<tr><th>Guarded command</th><th>POSIX implementation</th></tr>
<tr><td>when (guard) [
  statement 1;
  ...
  statement n;
]</td><td>pthread_mutex_lock(&mutex);
while(!guard)
  pthread_cond_wait(
      &cv,
      &mutex);
statement 1;
...
statement n;
pthread_mutex_unlock(&mutex);</td></tr>
<tr><td>[/* code
   * modifying
   * the guard
   */
]</td><td>pthread_mutex_lock(&mutex);
/*code modifying the guard:*/
...
pthread_cond_broadcast(&cv);
pthread_mutex_unlock(&mutex);</td></tr>
</table>

- if you don’t follow these rules, your code will have *race conditions* (i.e., timing-dependent behavior)

POSIX Condition Variables

Guarded command	*POSIX implementation*

`when (guard) [`
`  statement 1;`
`  ...`
`  statement n;`
`]`
	`pthread_mutex_lock(&mutex);`
	`while(!guard)`
	`  pthread_cond_wait(`
	`      &cv,`
	`      &mutex);`
	`statement 1;`
	`...`
	`statement n;`
	`pthread_mutex_unlock(&mutex);`

`[ /* code`
`   * modifying`
`   * the guard`
`   */`
`]`
	`pthread_mutex_lock(&mutex);`
	`/*code modifying the guard:*/`
	`...`
	`pthread_cond_broadcast(&cv);`
	`pthread_mutex_unlock(&mutex);`

- **don't believe that `pthread_cond_signal/broadcast()` can be called without locking the mutex**

Set Up

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

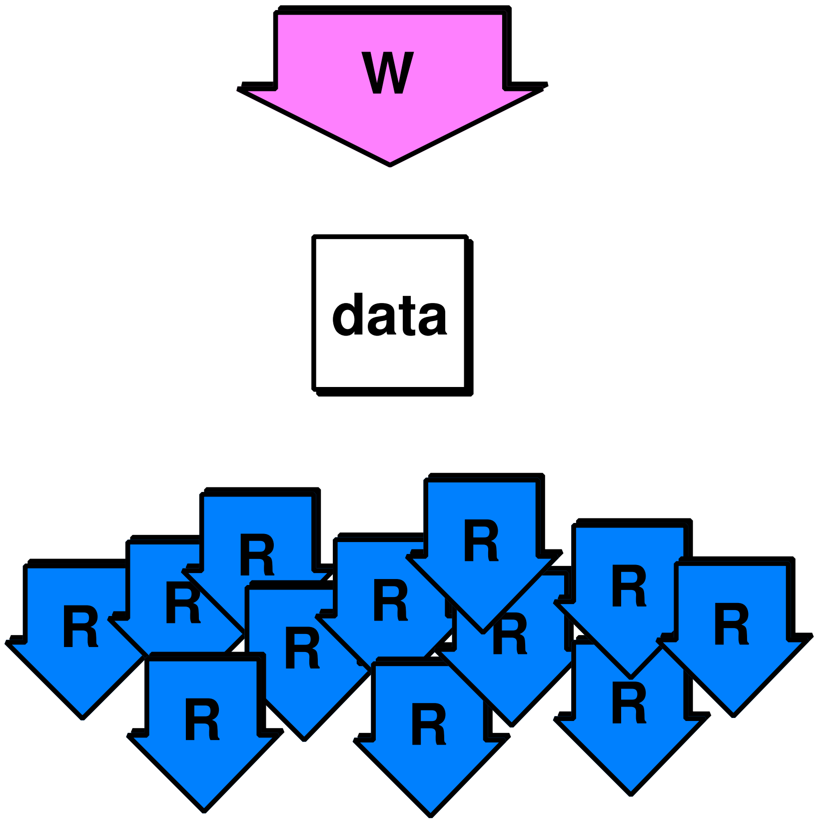
- If a condition variable cannot be initialized statically, do:

```
int pthread_cond_init(
        pthread_cond_t *cvp,
        pthread_condattr_t *attrp)

int pthread_cond_destroy(
        pthread_cond_t *cvp)
```

- Usually, condition variable attributes are not used

Readers-Writers Problem



Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

Pseudocode with Assertions

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  // sanity check
  assert((writers == 0) &&
      (readers > 0));
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  // sanity check
  assert((readers == 0) &&
        (writers == 1));
  /* write */
  [writers--;]
}
```

- the sanity checks are really not necessary

Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- since `readers` is part of the guard in a guarded commend, in the implementation of `[readers--;]`, you must signal/broadcast the corresponding condition used to implement that guard
  - in this case, only have to signal if `readers` becomes 0

Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
       (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- also, since `writers` is part of the guards in guarded commends (and these two guards are not identical), in the implementation of `[writers--;]`, you must signal/broadcast the corresponding conditions used to implement these guards

Readers-Writers Pseudocode

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
      (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

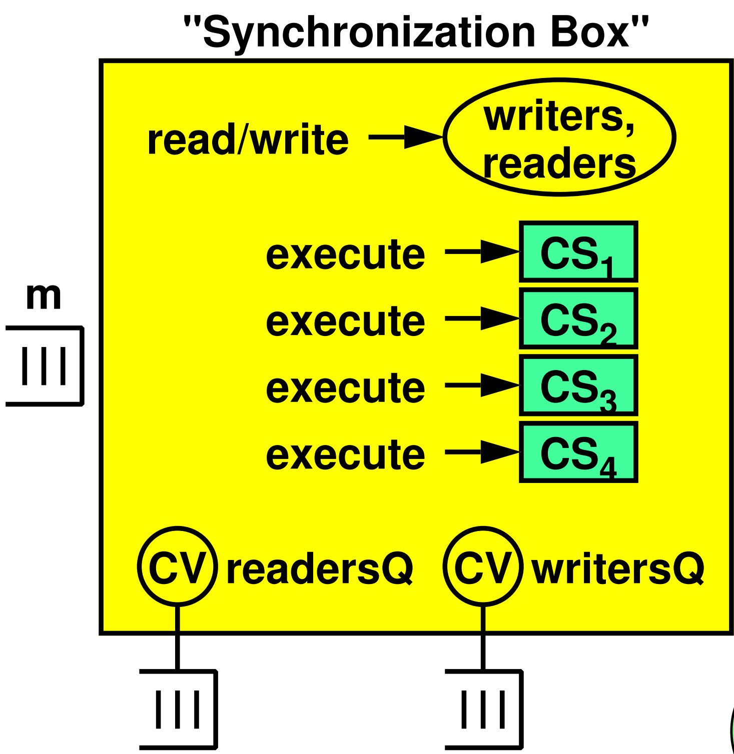
- don't have to worry about this `readers`
- don't have to worry about this `writers`
  - you need to look at your program logic and figure when signal/broadcast conditions won't be useful
    - it's ***not wrong*** to signal/broadcast here, it's just ***wasteful/inefficient***

Solution with POSIX Threads

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
        (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- to be even more "efficient", can use multiple CVs
  - so you don't have to wake up a thread unnecessarily
    - here we use one CV for reader's guard and one CV for writer's guard (since their expressions are different)



Solution with POSIX Threads

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

```
writer( ) {
  pthread_mutex_lock(&m);
  while(!((readers == 0) &&
      (writers == 0)))
    pthread_cond_wait(
        &writersQ, &m);
  writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  pthread_cond_signal(
      &writersQ);
  pthread_cond_broadcast(
      &readersQ);
  pthread_mutex_unlock(&m);
}
```

- one mutex (`m`) and two condition variables (`readersQ` and `writersQ`)

The Starvation Problem

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
      (readers == 0)) [
    writers++;
  ]
  /* write */
  [writers--;]
}
```

- **Can the writer never get a chance to write?**
  - yes, if there are always readers
  - so, this implementation can be unfair to writers
- **Solution**
  - once a writer arrives, shut the door on new readers
    - `writers` now means the number of writers *wanting* to write
    - use `active_writers` to make sure that only one writer can do the actual writing at a time

Solving The Starvation Problem

```
reader( ) {
  when (writers == 0) [
    readers++;
  ]
  /* read */
  [readers--;]
}
```

```
writer( ) {
  [writers++;]
  when ((readers == 0) &&
        (active_writers == 0))
  [
    active_writers++;
  ]
  /* write */
  [ writers--;
    active_writers--;
  ]
}
```

- now it's unfair to the readers
- isn't writing more important than reading anyway?

➡ **This is an example of how to give threads priority *without* assigning priorities to threads!**

Improved Reader

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

- exactly the same as before!

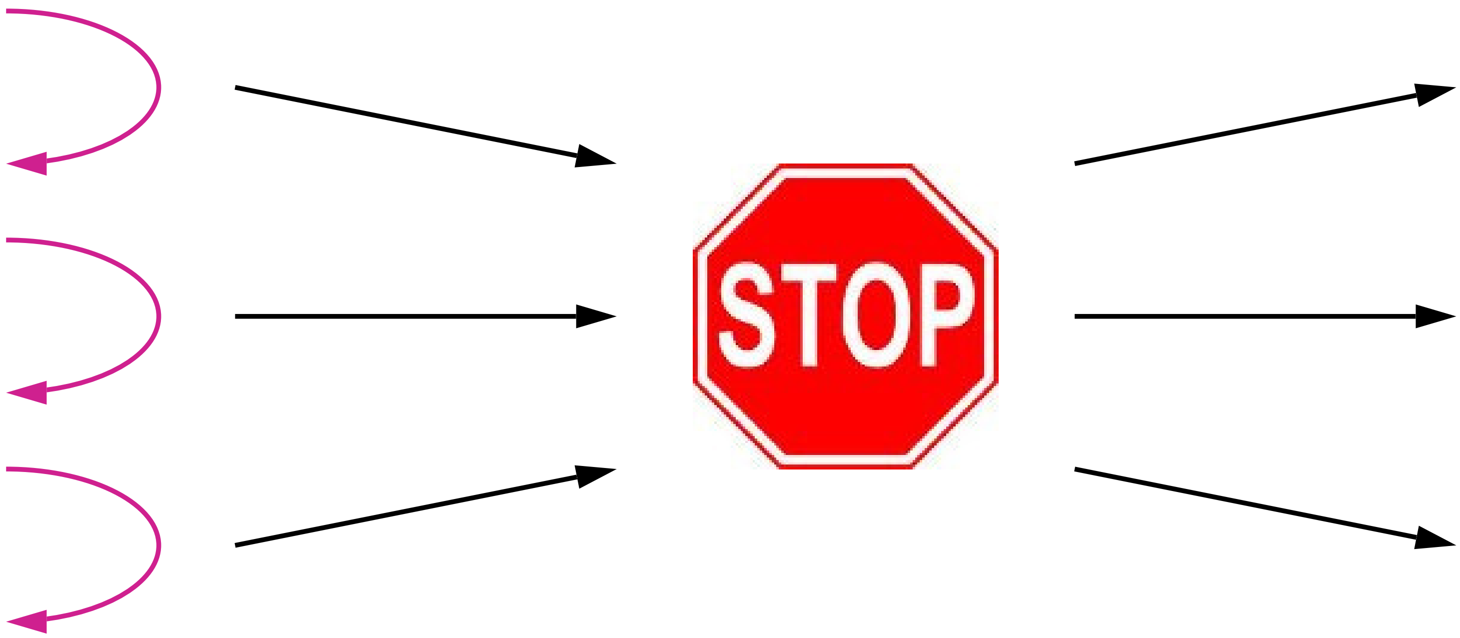
Improved Writer

```
writer( ) {
  pthread_mutex_lock(&m);
  writers++;
  while (!((readers == 0) &&
      (active_writers == 0))) {
    pthread_cond_wait(&writersQ, &m);
  }
  active_writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  active_writers--;
  if (writers > 0)
    pthread_cond_signal(&writersQ);
  else
    pthread_cond_broadcast(&readersQ);
  pthread_mutex_unlock(&m);
}
```

New, From POSIX!

```
int pthread_rwlock_init(
        pthread_rwlock_t *lock,
        pthread_rwlockattr_t *att);
int pthread_rwlock_destroy(
        pthread_rwlock_t *lock);
int pthread_rwlock_rdlock(
        pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(
        pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(
        pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(
        pthread_rwlock_t *lock);
int pthread_timedrwlock_rdlock(
        pthread_rwlock_t *lock, struct timespec *ts);
int pthread_timedrwlock_wrlock(
        pthread_rwlock_t *lock, struct timespec *ts);
int pthread_rwlock_unlock(
        pthread_rwlock_t *lock);
```

Barriers



- When a thread reaches a barrier, it must stop (do nothing) and simply wait for other threads to arrive at the same barrier
  - when all the threads that were suppose to arrive at the barrier have all arrived at the barrier, they are all given the signal to proceed forward
    - the barrier is then reset
- Ex: fork/join (fork to create parallel execution)

A Solution?

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < n) {
    pthread_cond_wait(&BarrierQueue, &m);
  } else {
    count = 0;
    pthread_cond_broadcast(&BarrierQueue);
  }
  pthread_mutex_unlock(&m);
}
```

- the idea here is to have the last thread broadcast the condition while all the other threads are blocked at waiting for the condition to be signaled
- as it turns out, `pthread_cond_wait()` might return *spontaneously*, so this won't work
  - http://pubs.opengroup.org/onlinepubs/009604599/functions/pthread_cond_signal.html

A Solution?

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < n) {
    while (count < n)
      pthread_cond_wait(&BarrierQueue, &m);
  } else {
    pthread_cond_broadcast(&BarrierQueue);
    count = 0;
  }
  pthread_mutex_unlock(&m);
}
```

- **if the `n`th thread wakes up all the other blocked threads, most likely, *none* of these threads will see `count == n`**

A Solution?

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < n) {
    while (count < n)
      pthread_cond_wait(&BarrierQueue, &m);
  } else {
    pthread_cond_broadcast(&BarrierQueue);
  }
  pthread_mutex_unlock(&m);
  count = 0;
}
```

- if the `n` th thread wakes up all the other blocked threads, most likely, *none* of these threads will see `count == n`
- moving `count = 0` around won't help
  - cannot guarantee all `n` threads will exit the barrier

Barrier in POSIX Threads

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;
void barrier_sync() {
  pthread_mutex_lock(&m);
  if (++count < number) {
    int my_generation = generation;
    while(my_generation == generation)
      pthread_cond_wait(&BarrierQueue, &m);
  } else {
    count = 0;
    generation++;
    pthread_cond_broadcast(&BarrierQueue);
  }
  pthread_mutex_unlock(&m);
}
```

- don't use `count` in the guard since its problematic!
- introduce a new guard

More From POSIX!

```
int pthread_barrier_init(
        pthread_barrier_t *barrier,
        pthread_barrierattr_t *attr,
        unsigned int count);
int pthread_barrier_destroy(
        pthread_barrier_t *barrier);
int pthread_barrier_wait(
        pthread_barrier_t *barrier);
```