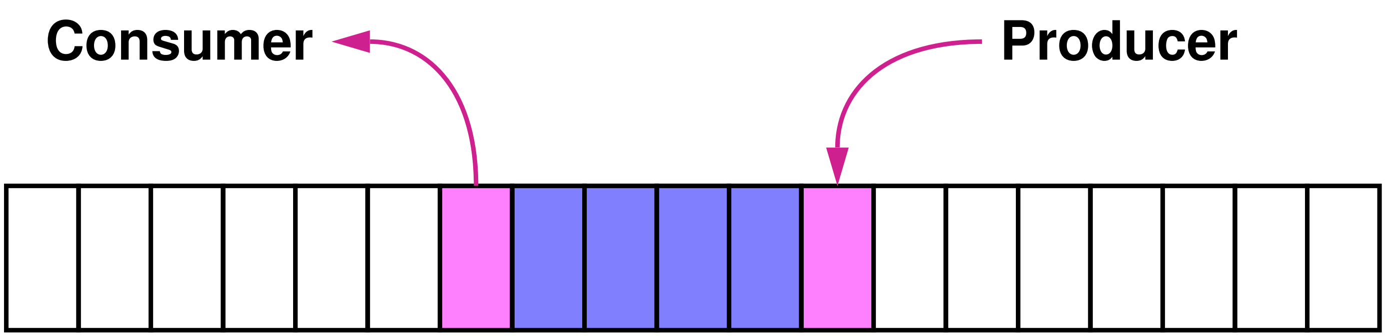
Housekeeping (Lecture 5 - 9/11/2013)

- **Warmup #1 due at 11:45pm this Friday, 9/13/2013**
  - 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
- *Grading guidelines* is the *ONLY* way we will grade and we will grade on `nunki.usc.edu` in our grading account (which you don't have access to)
  - due to our *fairness* policy
  - it's a good idea to run your code against the grading guidelines
- Please note that the jobs of the TAs and course producer is *NOT* to do your work for you (and not to teach you C or Unix)
  - that's your job (and learn some stuff on your own)
- Signing rowsheets will start next Monday
  - if you are registered for one section but would attend lecture in another section, please send me e-mail so I can add your name to the rowsheet

Beyond Mutexes

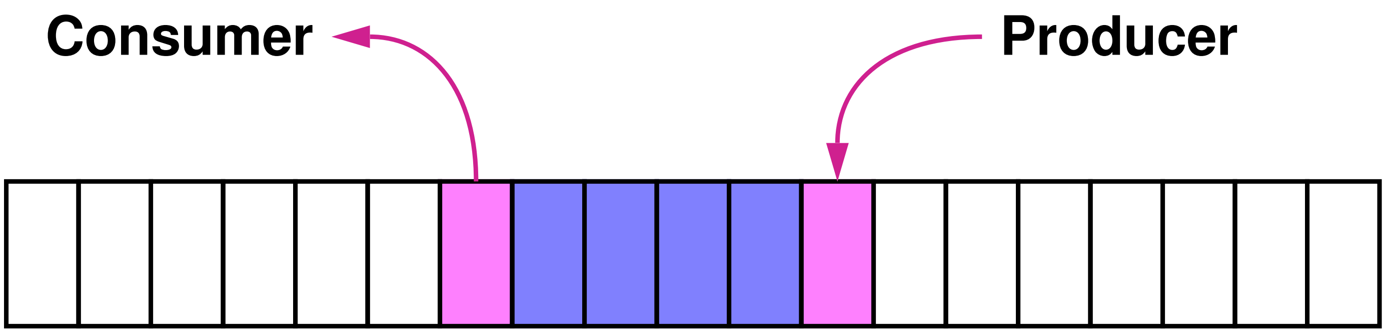
- **Mutex is necessary when shared data is being modified**
  - **what if some threads just want to look at (i.e., read) a piece of data?**
    - **mutex would work, but too restrictive and inefficient (lock threads out when it's not necessary)**
- **Producer-Consumer problem (a.k.a., bounded-buffer problem)**
- **Readers-Writers problem**
- **Barrier**

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

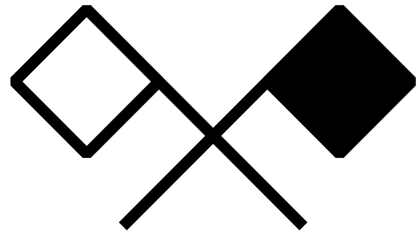
Guarded Commands

```
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
  - *atomically* mean that it's executed without interruption
  - since it's executed atomically and without interruption, it is as if it's executed in an instance of time (duration = 0), including the time to evaluate the guard
    - *evaluting the guard* and *executing the command sequence* altogether is an *atomic operation* if the guard is true

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);
  ...
  /* 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 *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 it
- 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);
}
```

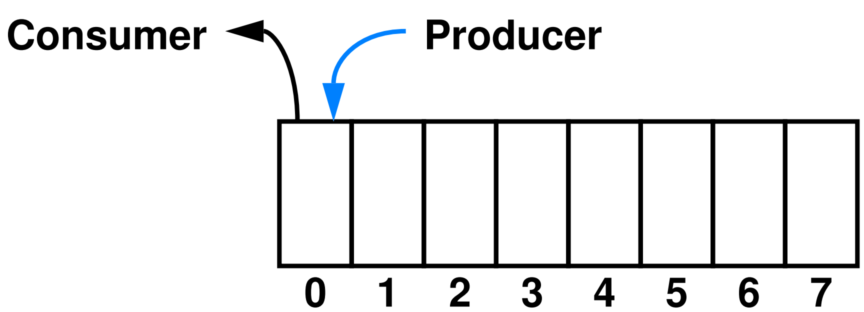
Producer/Consumer with Semaphores

```
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

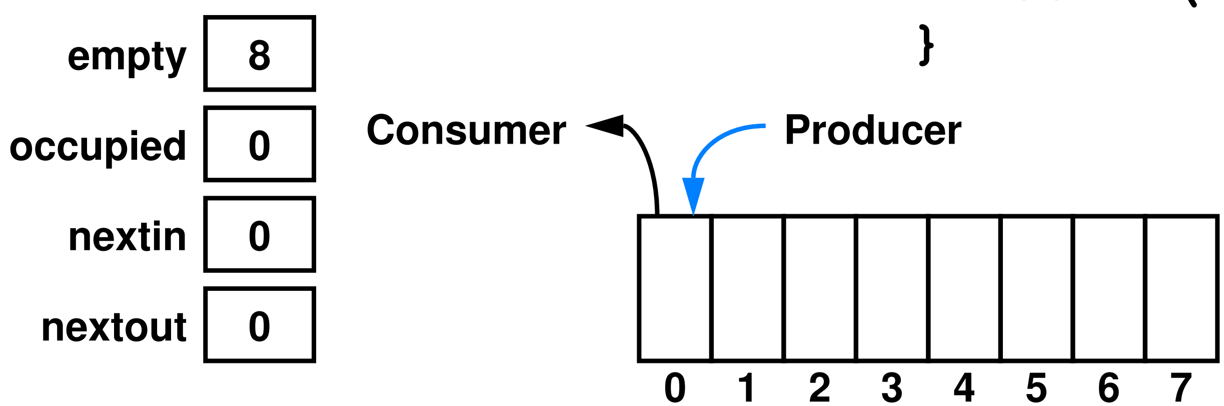


Producer/Consumer with Semaphores

```
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);
}
```



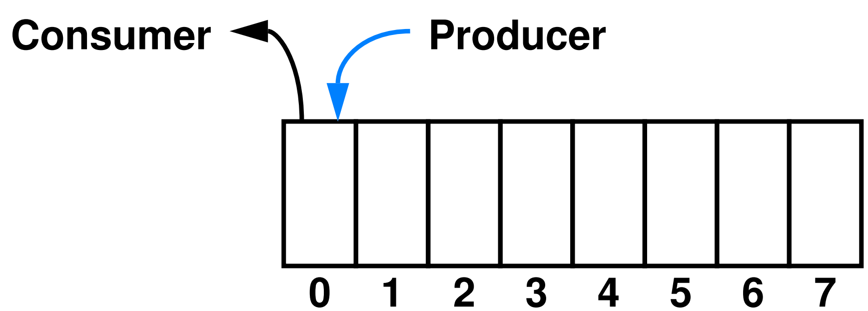
Producer/Consumer with Semaphores

```
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



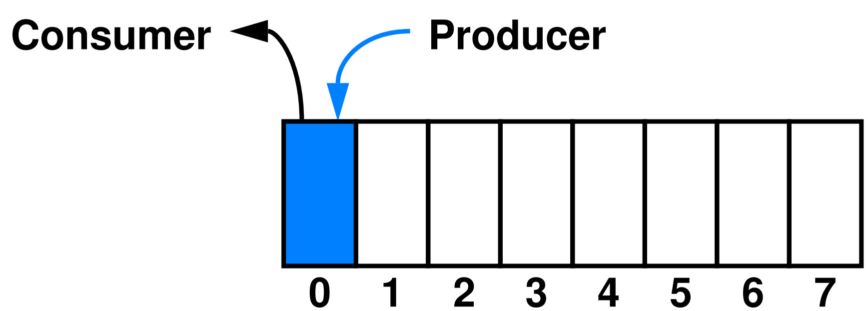
Producer/Consumer with Semaphores

```
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



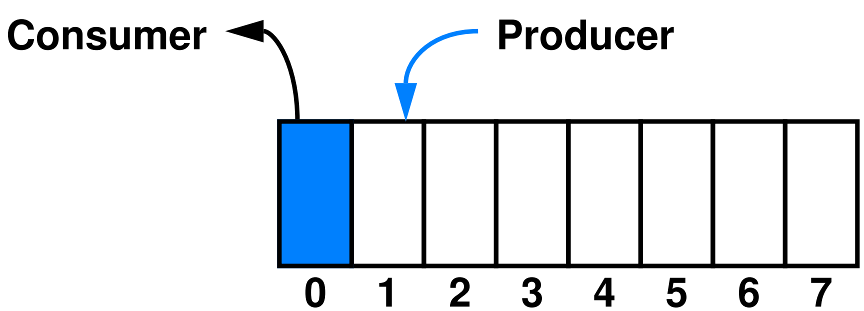
Producer/Consumer with Semaphores

```
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



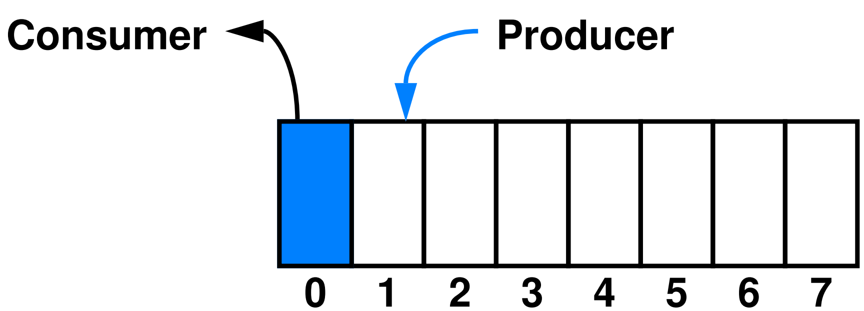
Producer/Consumer with Semaphores

```
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



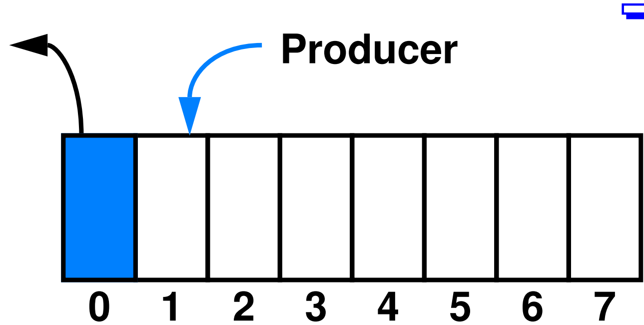
Producer/Consumer with Semaphores

```
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

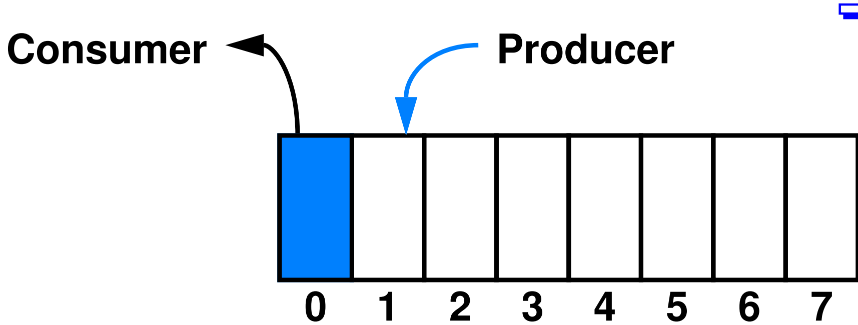
Producer/Consumer with Semaphores

```
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

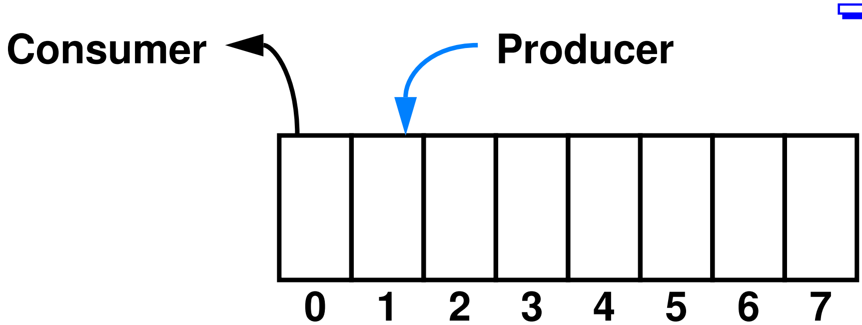
```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
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);
}
```

- note: producer continue to produce

empty	7
occupied	0
nextin	1
nextout	0



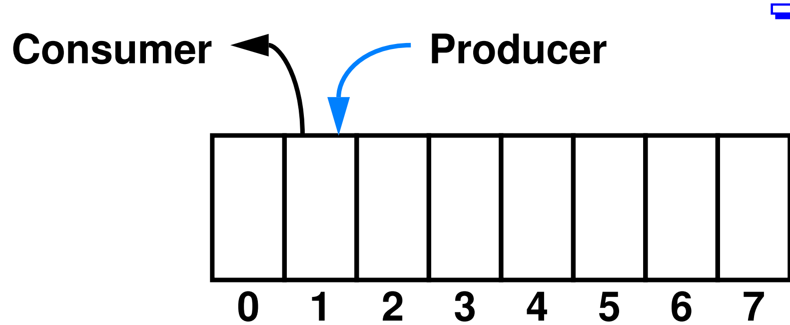
Producer/Consumer with Semaphores

```
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	1



- note: producer continue to produce

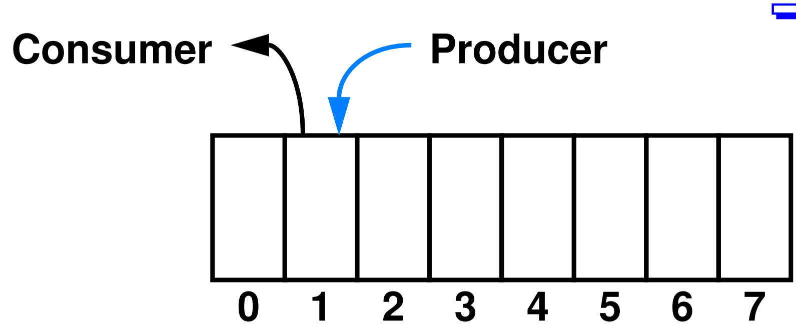
Producer/Consumer with Semaphores

```
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

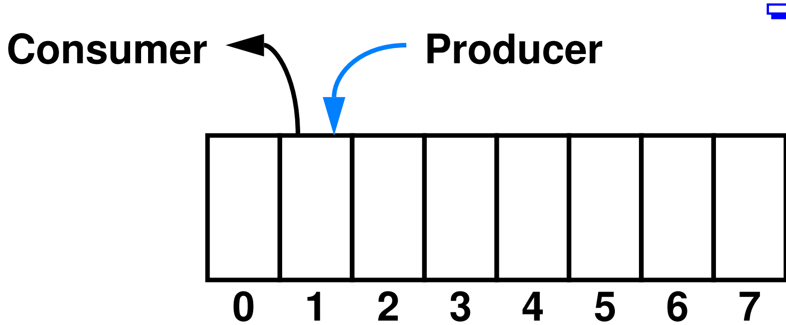
Producer/Consumer with Semaphores

```
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

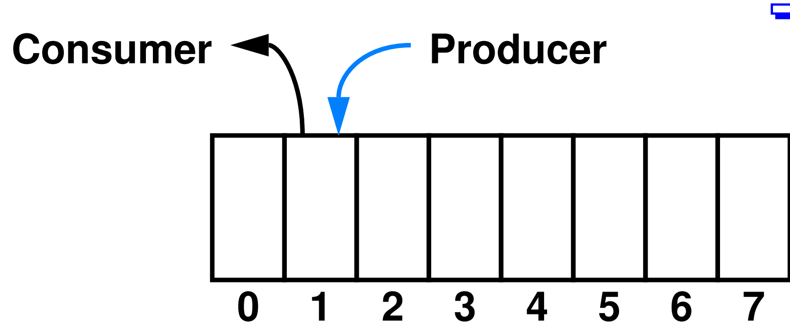
Producer/Consumer with Semaphores

```
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 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"

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

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

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

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

```c
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
  - can think of the guard as a predicate (i.e., evaluates to either true or false) that *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
    - once we see that it evaluates to "true", we have to execute the command sequence atomically
    - a mutex is involved, but how?
      - need to be efficient

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")
    - threads waiting for a guard to become true join such a queue
      - they wait for a *condition* to be *singaled*
    - threads that do something to potentially change the value of a guard can then wake up the threads that were waiting in the queue
      - they can *signal* or *broadcast* the *condition* (sometimes called an "event")
      - *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, the "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?

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 unblocks `mutex` and wait for the "event"
- when the event is signaled, `pthread_cond_wait()` returns with the `mutex` *locked*

Implementation Of Guarded Commands

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

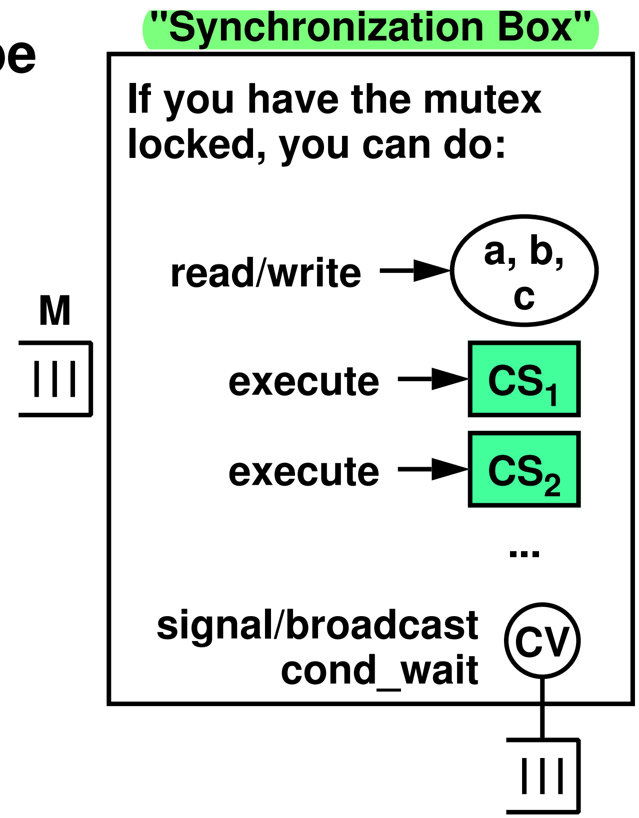
- POSIX provides *condition variables* for programmers to implement guarded commands
  - conceptually, the "event" (signaling/broadcasting of a condition) happens in an instance of time (duration of this "event" is zero)
    - if you are not waiting for it, you'll miss it
    - how do you make sure you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

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*

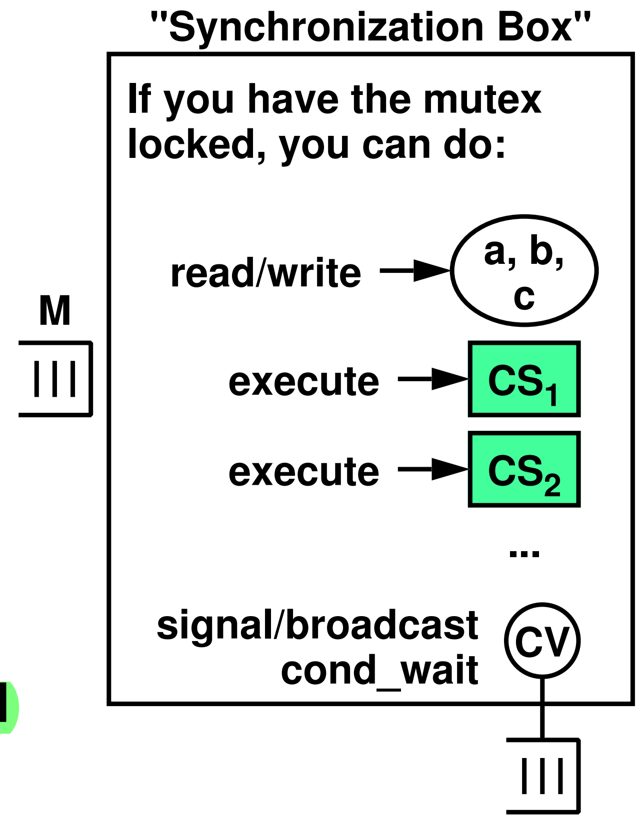
Implementation Of Guarded Commands

- ***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 predicate
    - can only access them if a thread is inside the "synchronization box"



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



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);

Set Up

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

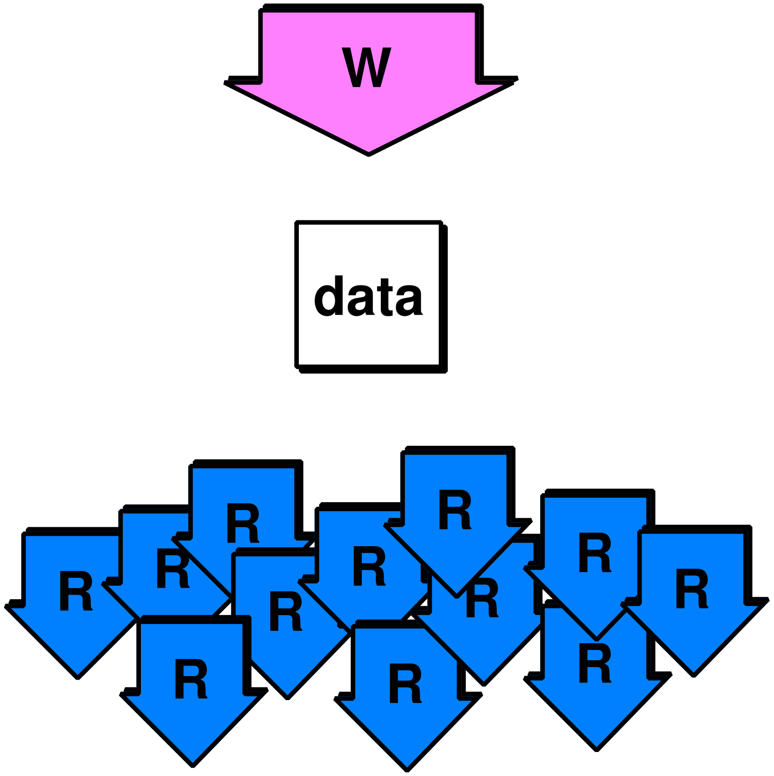
- If a condition variable cannot be initialized statically, do:

```
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

Pseudocode with Assertions

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

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

- remember, 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

Pseudocode with Assertions

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

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

- also, since `writers` is part of the guard in a guarded commend, in the implementation of `[writers--;]`, you must signal/broadcast the corresponding condition used to implement that guard

Pseudocode with Assertions

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

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

- don't have to worry about this `readers`
  - you need to look at your program logic and figure when signal/broad conditions won't be useful

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 write 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?

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!

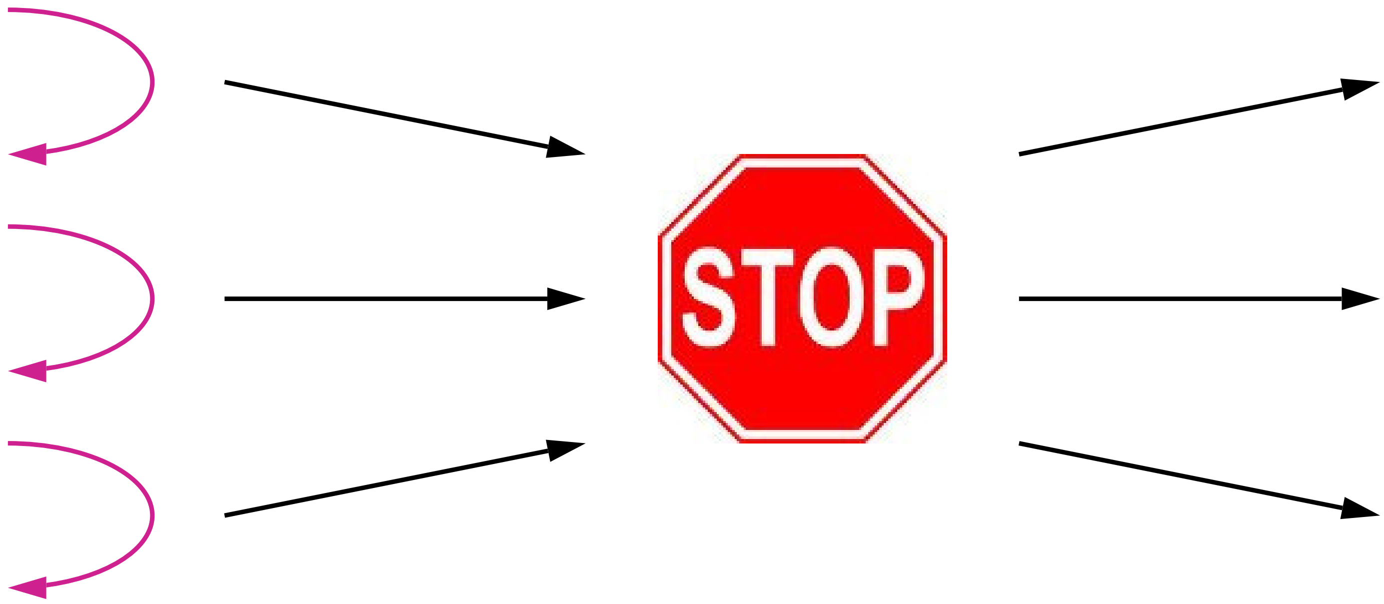
Improved Writer

```
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() {
  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() {
  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, not all these threads will see `count == n`**
  - **moving `count = 0` around won't help**

Barrier in POSIX Threads

```
int count = 0;
pthread_mutex_t m;
pthread_cond_t BarrierQueue;

void barrier() {
  pthread_mutex_lock(&m);
  if (++count < number) {
    int my_generation = generation;
    while(my_generation == generation)
      pthread_cond_wait(&waitQ, &m);
  } else {
    count = 0;
    generation++;
    pthread_cond_broadcast(&waitQ);
  }
  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);
```