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: this is not the first procedures of producer and consumer threads
  - this is *synchronization code* (producer and consumer threads calls them to *synchronize* with each other)

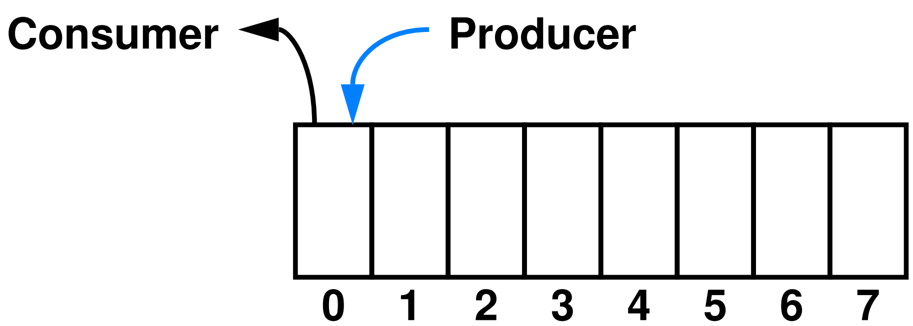
Producer/Consumer with Semaphores

```
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



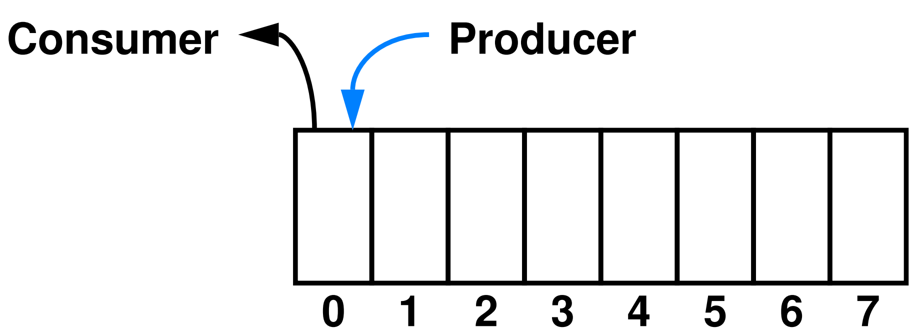
Producer/Consumer with Semaphores

```
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



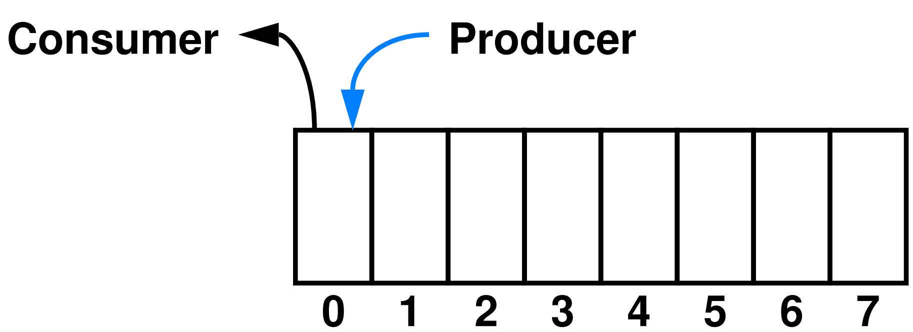
Producer/Consumer with Semaphores

```
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



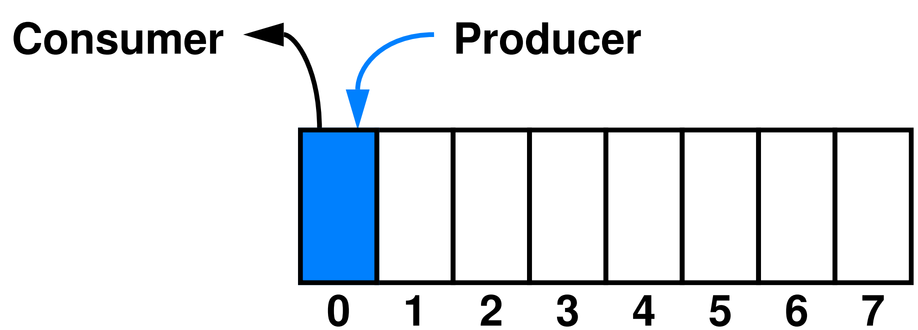
Producer/Consumer with Semaphores

```
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



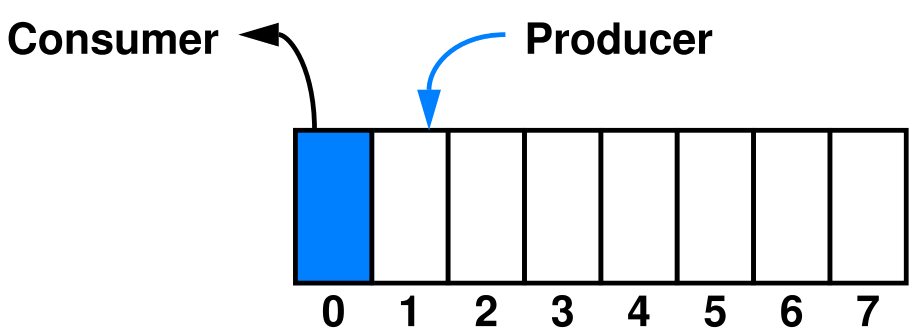
Producer/Consumer with Semaphores

```
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



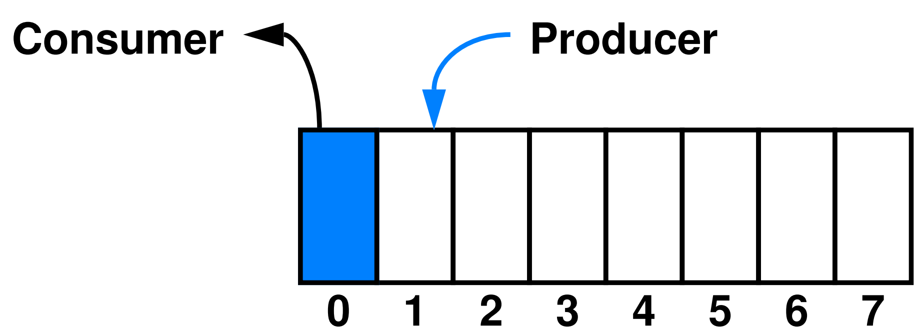
Producer/Consumer with Semaphores

```
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



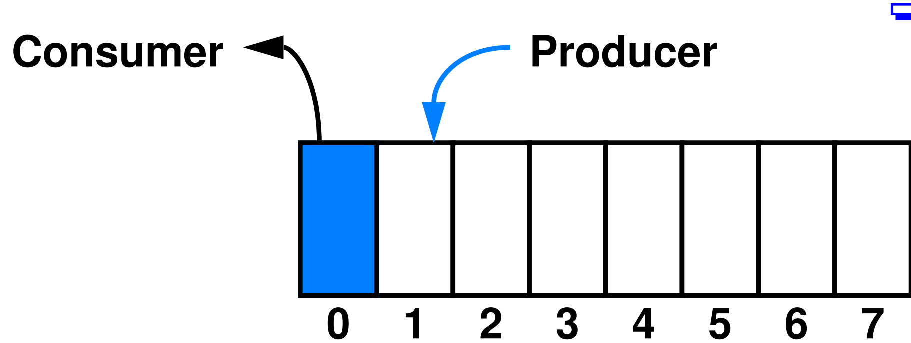
Producer/Consumer with Semaphores

```
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

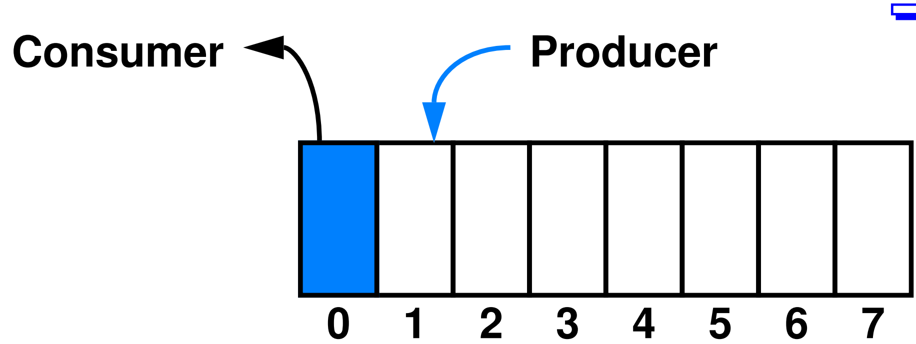
Producer/Consumer with Semaphores

```
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

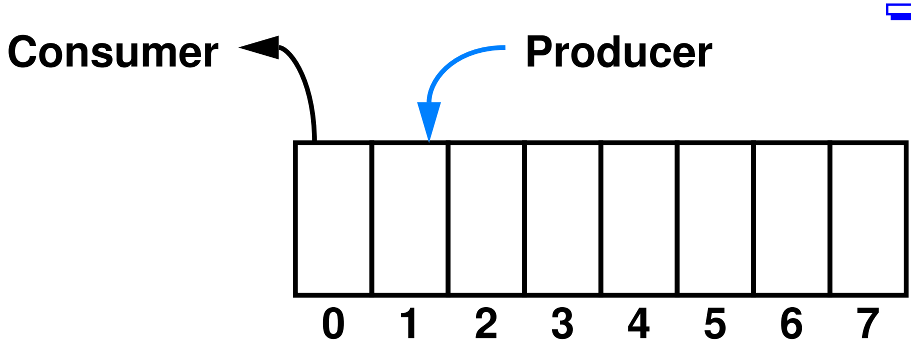
Producer/Consumer with Semaphores

```
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

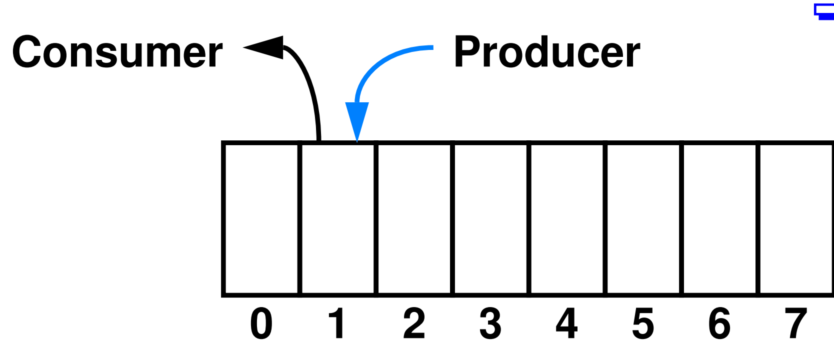
Producer/Consumer with Semaphores

```
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

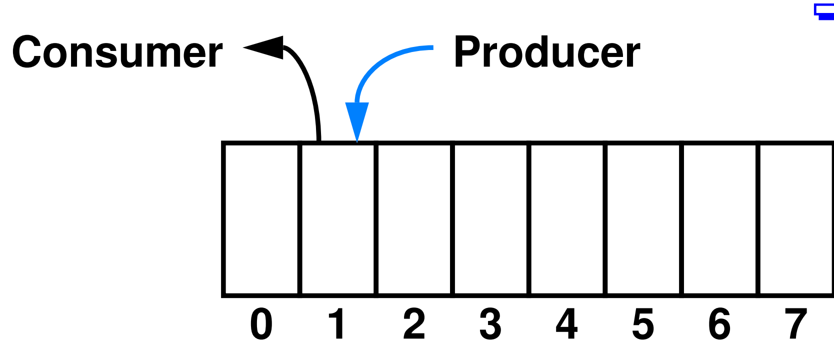
Producer/Consumer with Semaphores

```
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

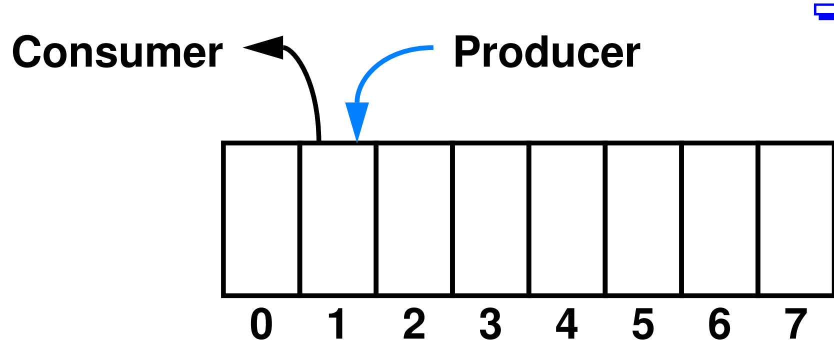
Producer/Consumer with Semaphores

```
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

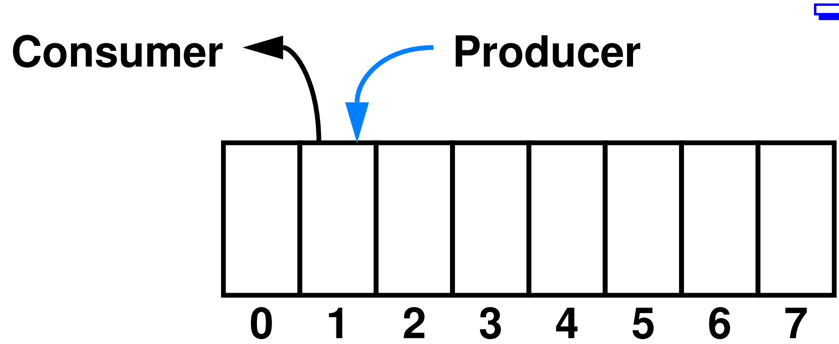
Producer/Consumer with Semaphores

```
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 may ever wait
  - parallelism of 2
- 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;
```

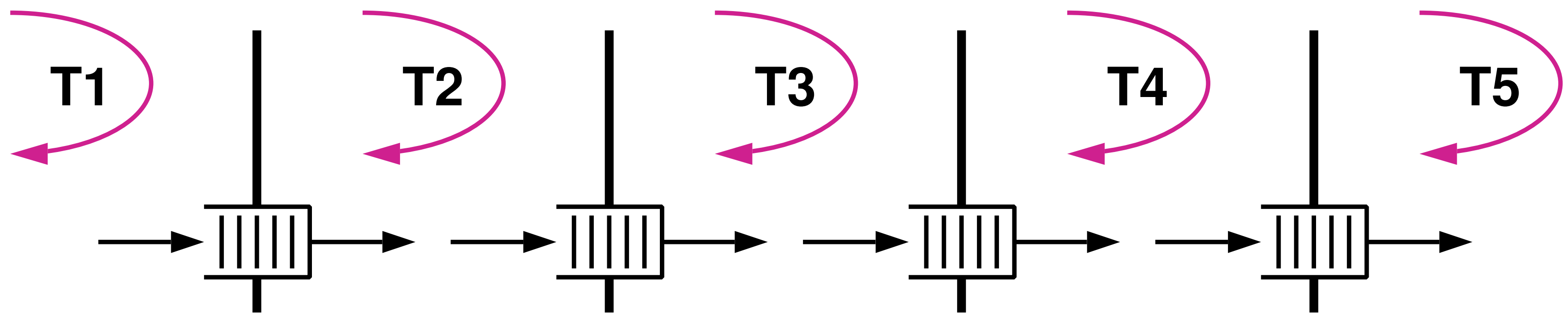
```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
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* gives more "*fine grain parallelism*"
  - but *not general* enough

Semaphore

- ***Semaphore*** **has limited use**
  - **pretty much the only place it's really good for is producer-consumer**
    - **although it's a very important application of semaphores**



    - **the queues above are** ***queues with bounded buffer space***
      - **recall that the "producer-consumer problem" is also known as the "bounded-buffer problem"**
  - ***"pipelined parallelism"***

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) {
  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);
}
```

```
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);
}
```

Implementing Semaphore With Mutex

<table>
<tr><th>Semaphore operation</th><th>POSIX implementation</th></tr>
<tr><td><code>when (S > 0) [
  S = S - 1;
]</code></td><td><code>while(1) {
  pthread_mutex_lock(&m);
  if (S > 0) {
    S = S - 1;
    pthread_mutex_unlock(&m);
    break;
  }
  pthread_mutex_unlock(&m);
}</code></td></tr>
<tr><td><code>[S = S + 1;]</code></td><td><code>pthread_mutex_lock(&m);
S = S + 1;
pthread_mutex_unlock(&m);</code></td></tr>
</table>

Implementing Semaphore With Mutex

Semaphore operation	*POSIX implementation*

`when (S > 0) [`
`  S = S - 1;`
`]`
	`while(1) {`
	`  pthread_mutex_lock(&m);`
	`  if (S > 0) {`
	`    S = S - 1;`
	`    pthread_mutex_unlock(&m);`
	`    break;`
	`  }`
	`  pthread_mutex_unlock(&m);`
	`}`

- Implementation of `P(S)` above works but not good - *inefficient*
  - what if `guard` is false (in this case, `S` = 0) and no other thread is waiting for the mutex?
    - *busy waiting:* your thread will not give up CPU while waiting for the `guard` to become true
      - it does not do anythnig "useful"
  - right way to wait is to *give up the CPU* when waiting

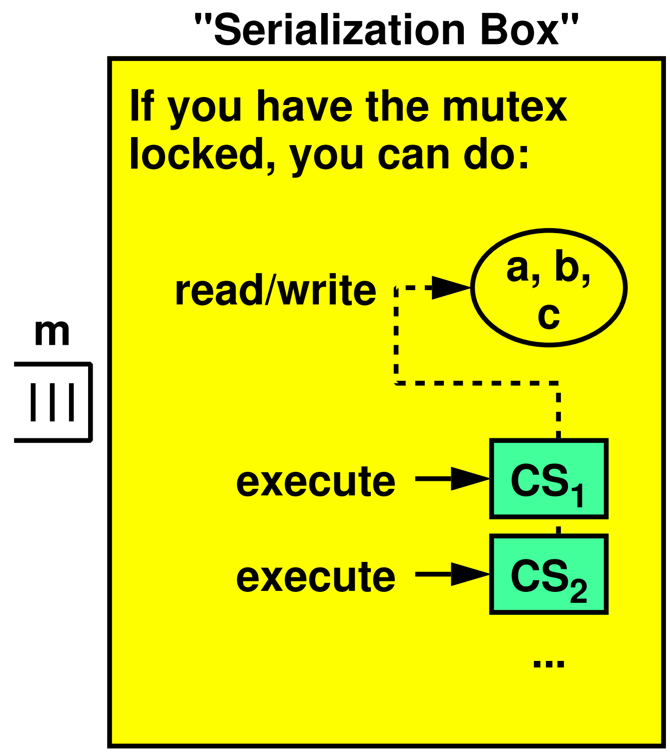
Implementation Of General Guarded Commands

```
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 in multiple CPUs *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?

Implementation Of General Guarded Commands

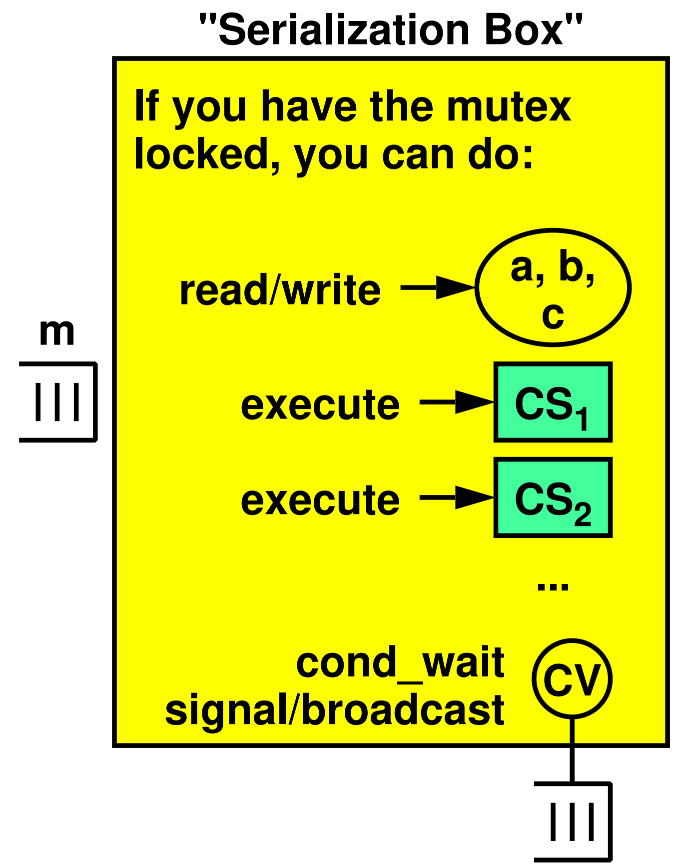
```
when (guard) [
  /* command sequence */
  . . .
]
```



- 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 in multiple CPUs *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?
    - this would work, but can lead to *busy-waiting*

Implementation Of General Guarded Commands

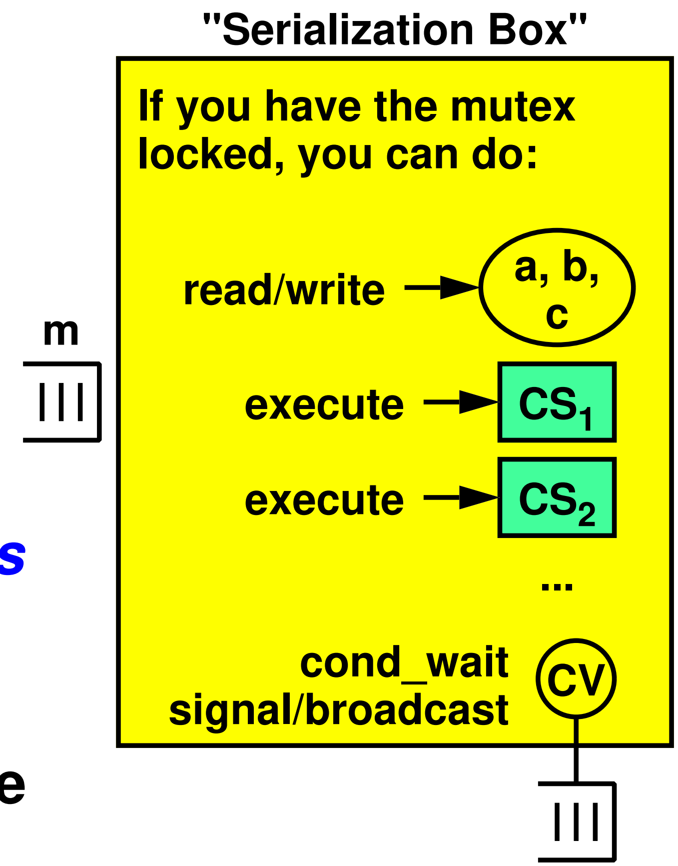
```
when (guard) [
  /* command sequence */
  ...
]
```



- 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 in multiple CPUs *simultaneously*
- Need something else *(known as condition variables)*
  - and a bunch of *rules* to follow

Implementation Of General Guarded Commands

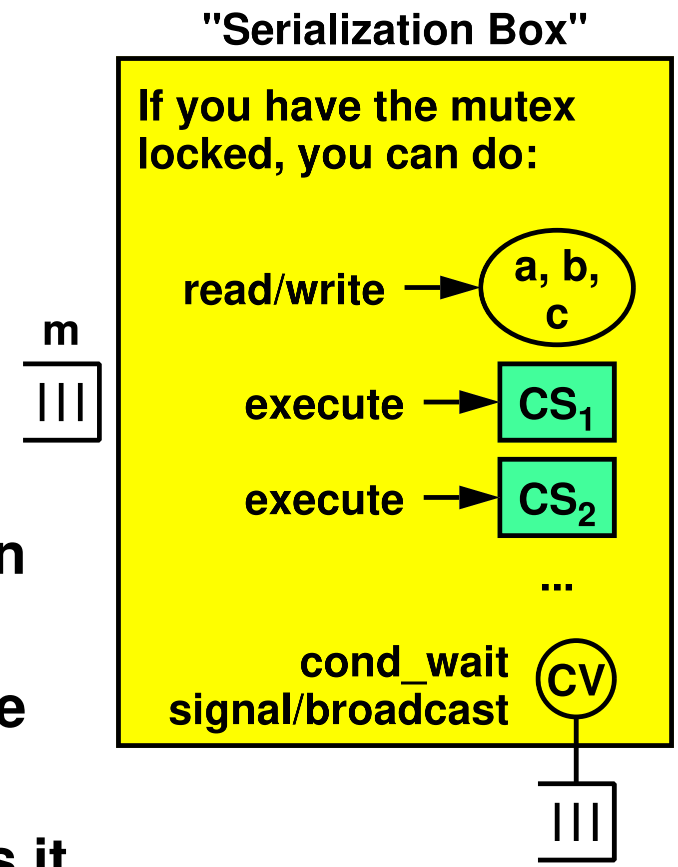
```
when (guard) [
  /* command sequence */
  ...
]
```



- POSIX provides *condition variables (CV)* 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, go to *sleep in such a queue*
      - they wait for a specific *condition* to be *signaled*
      - they wait for *the right time* to *re-evaluate the guard*

Implementation Of General Guarded Commands

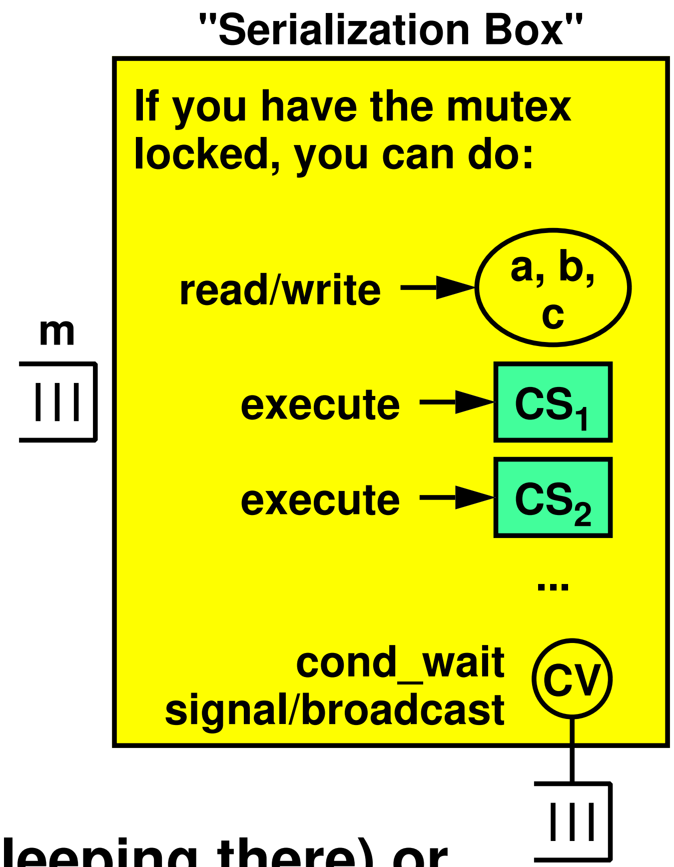
```
when (guard) [
  /* command sequence */
  . . .
]
```



- POSIX provides *condition variables (CV)* for programmers to implement guarded commands
  - unlike notifications on your cellphone, 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 rules/protocol* (for multiple interacting threads to follow) described here

Implementation Of General Guarded Commands

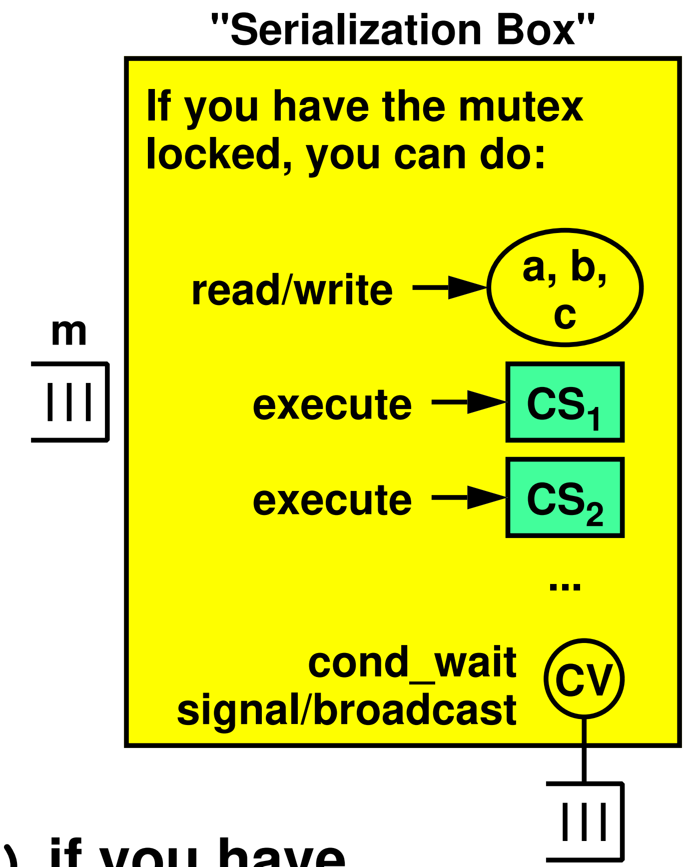
```
when (guard) [
  /* command sequence */
  . . .
]
```



- POSIX provides *condition variables (CV)* for programmers to implement guarded commands
  - threads that do something to *potentially* change the truth value of the guard can then wake up the threads that were sleeping in the queue
    - they can *signal* (wake up *one* thread sleeping there) or *broadcast* (wake up *all* threads sleeping there) the *condition*
      - *wake up* thread(s) by *moving it into the mutex queue*
    - *no guarantee* that the guard will be true when it's time for *another thread* to evaluate the guard again

Implementation Of General Guarded Commands

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



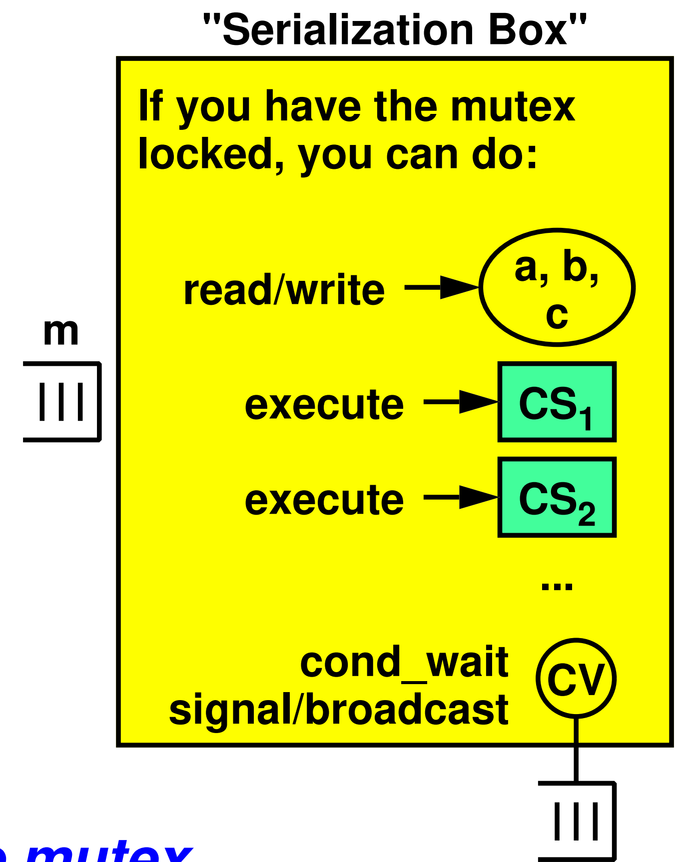
- POSIX provides *condition variables (CV)* for programmers to implement guarded commands

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

- must 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 General Guarded Commands

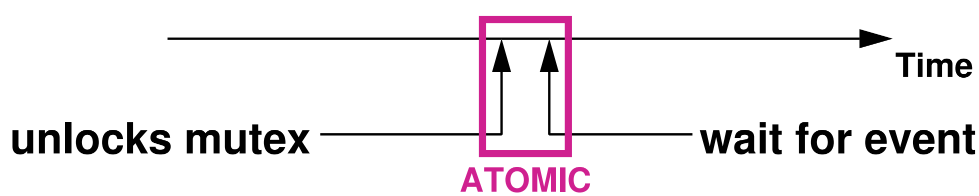
```
when (guard) [
  /* command sequence */
  ...
]
```



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

1) `pthread_cond_wait(cv, mutex)`

- *atomically* unlocks `mutex` and wait for the "event"
  - with respect to the *operation of the mutex*

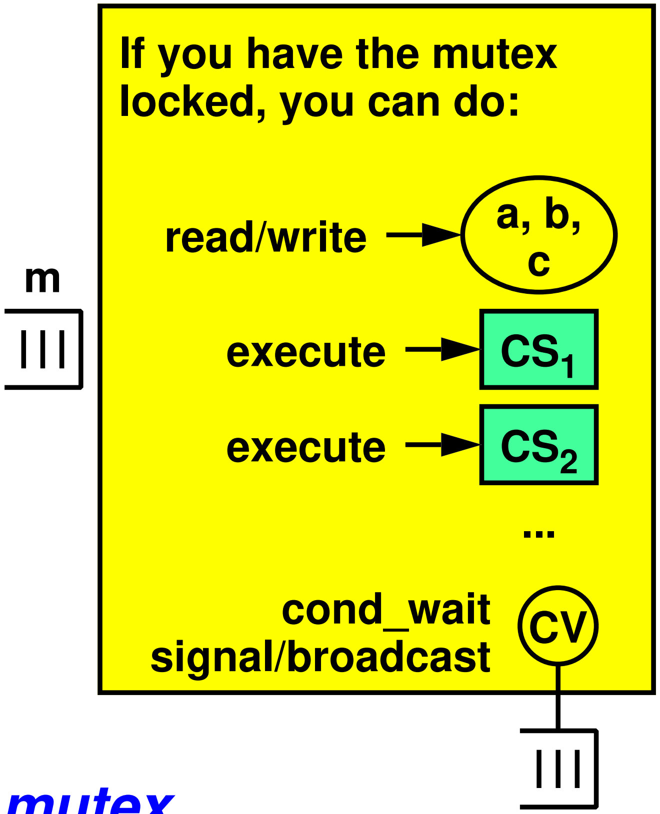


Implementation Of General Guarded Commands

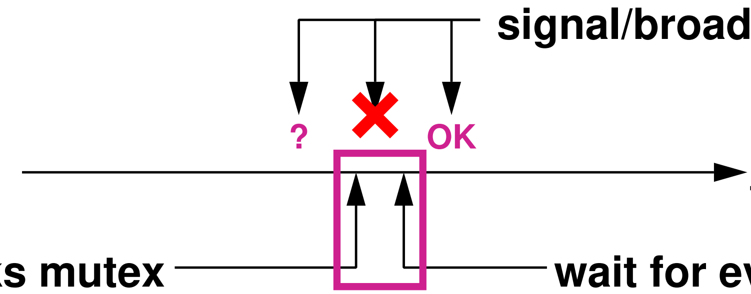
```
when (guard) [
  /* command sequence */
  ...
]
```

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

"Serialization Box"

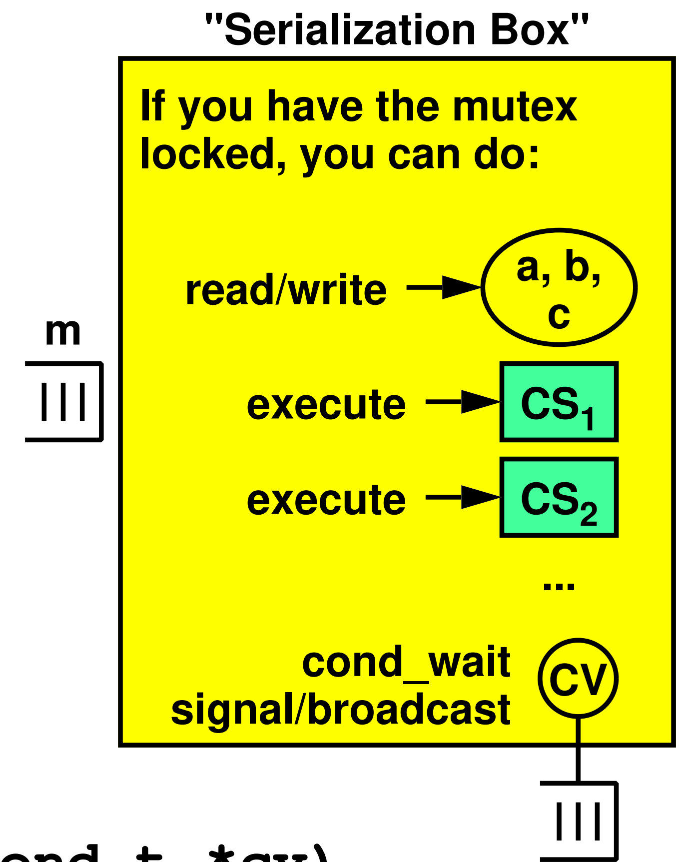


1) `pthread_cond_wait(cv, mutex)`
   - *atomically* unlocks `mutex` and wait for the "event"
     - with respect to the *operation of the mutex*

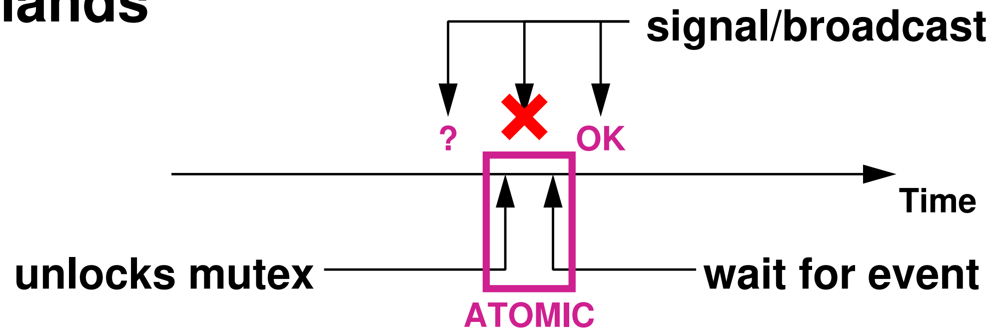


Implementation Of General Guarded Commands

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



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

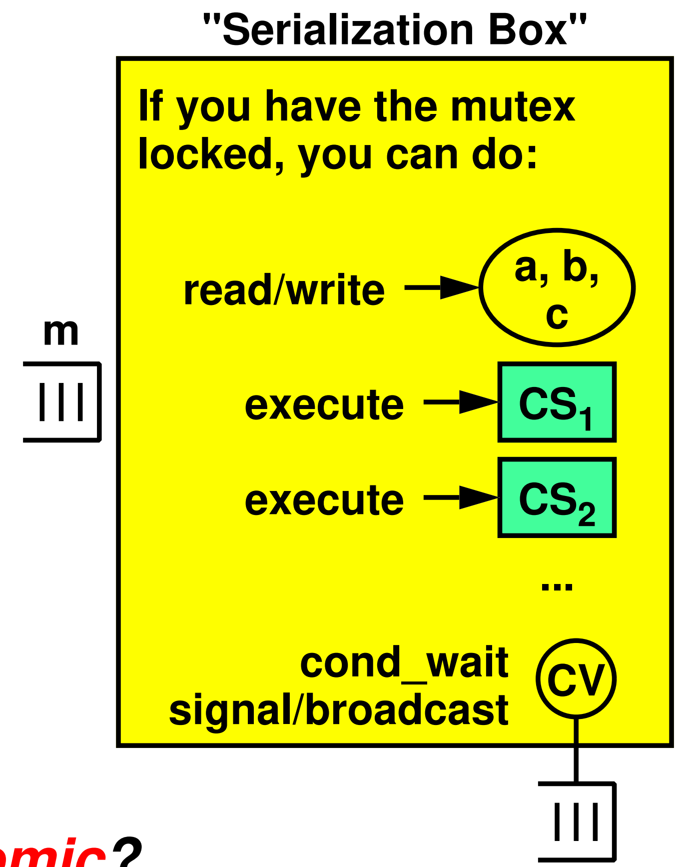


2) `pthread_cond_broadcast(pthread_cond_t *cv)`
`pthread_cond_signal(pthread_cond_t *cv)`

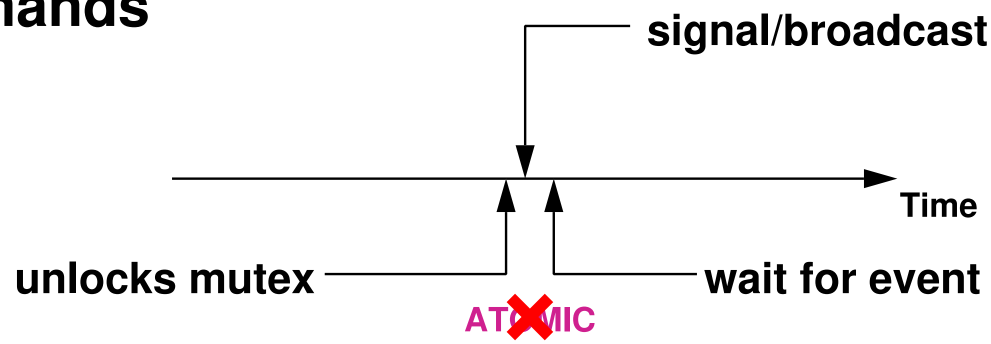
- must only call `pthread_cond_wait()`, `pthread_cond_broadcast()` or `pthread_cond_signal()` if you have the corresponding mutex *locked*

Implementation Of General Guarded Commands

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



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



- what if `pthread_cond_wait()` it *not atomic*?
  - your thread may miss the event and sleep forever in the CV queue
  - is this a deadlock?
    - no, this is a *race condition* (i.e., bad timing-dependent behavior)

Set Up

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

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

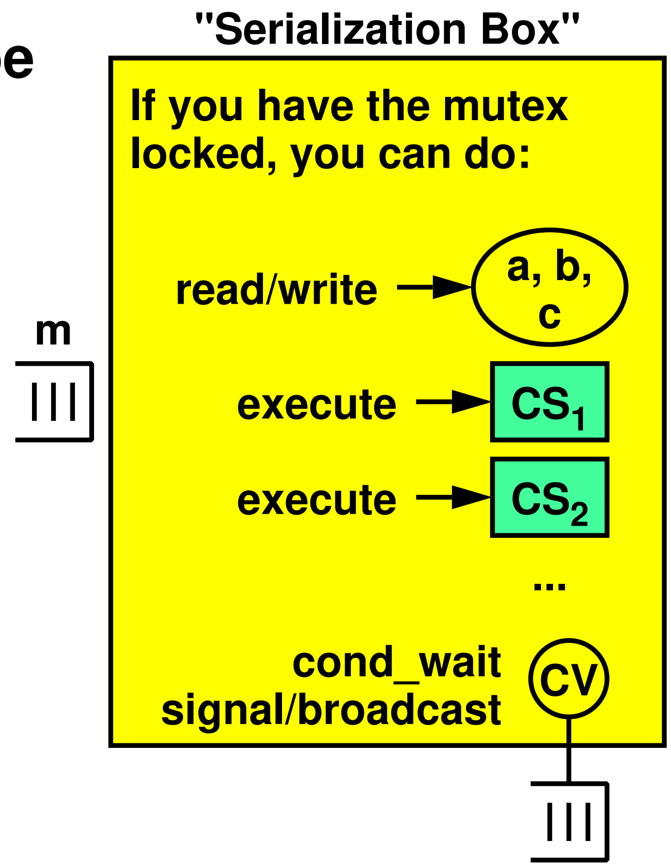
```
int pthread_cond_init(
        pthread_cond_t *cvp,
        pthread_condattr_t *attrp)

int pthread_cond_destroy(
        pthread_cond_t *cvp)
```

- Usually, condition variable attributes are not used

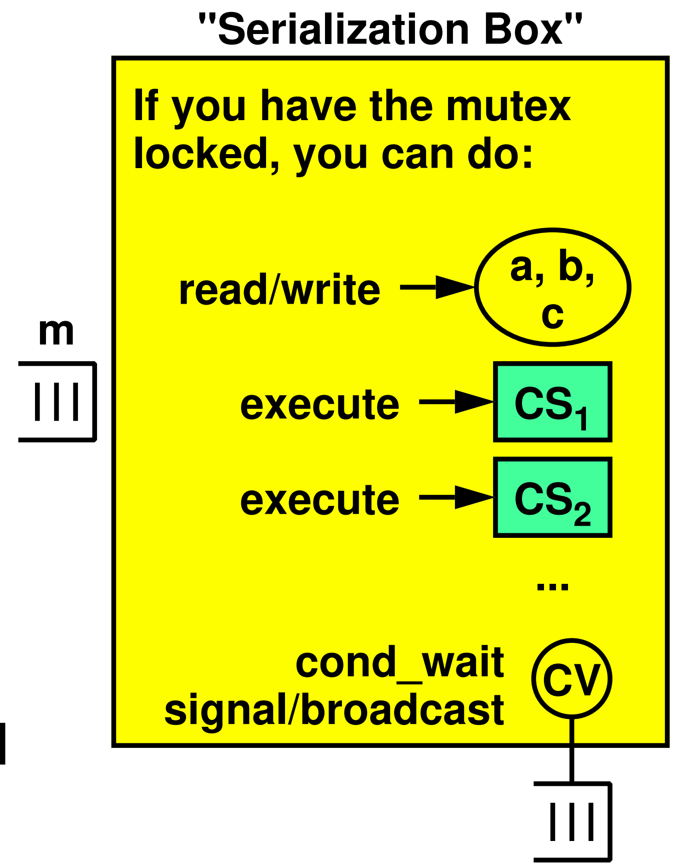
Implementation Of General Guarded Commands

- ***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 "serialization box"
      - only one thread can be inside the "serialization 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 "serialization box" (i.e., has the mutex locked)



Implementation Of General 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);`

- **if you don't follow these rules, your code will have *race conditions* (i.e., bad 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