CS 33

Multithreaded Programming II

Problem

```
    pthread_create(&thread, 0, start, 0);

    …


void *start(void *arg) {
    long BigArray[128*1024*1024];
    …
    return 0;
}
```

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);

...
```

Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 130*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

Execution Speed

```
// thread 1
lengthy_func();
x = 1;
```

```
// thread 2
x = 0;
assert(x==0);
```

Mutual Exclusion

Threads and Mutual Exclusion

Thread 1:

```
x = x+1;
 /*
   movl x,%eax
   incr %eax
   movl %eax,x
 */
```

Thread 2:

```
x = x+1;
 /*
   movl x,%eax
   incr %eax
   movl %eax,x
 */
```

Quiz 1

Suppose gcc produces the following code. Will it still be the case that x's value might not be incremented by 2?

a) yes
b) no

Thread 1:

```
x = x+1;
 /*
  incr x
 */
```

Thread 2:

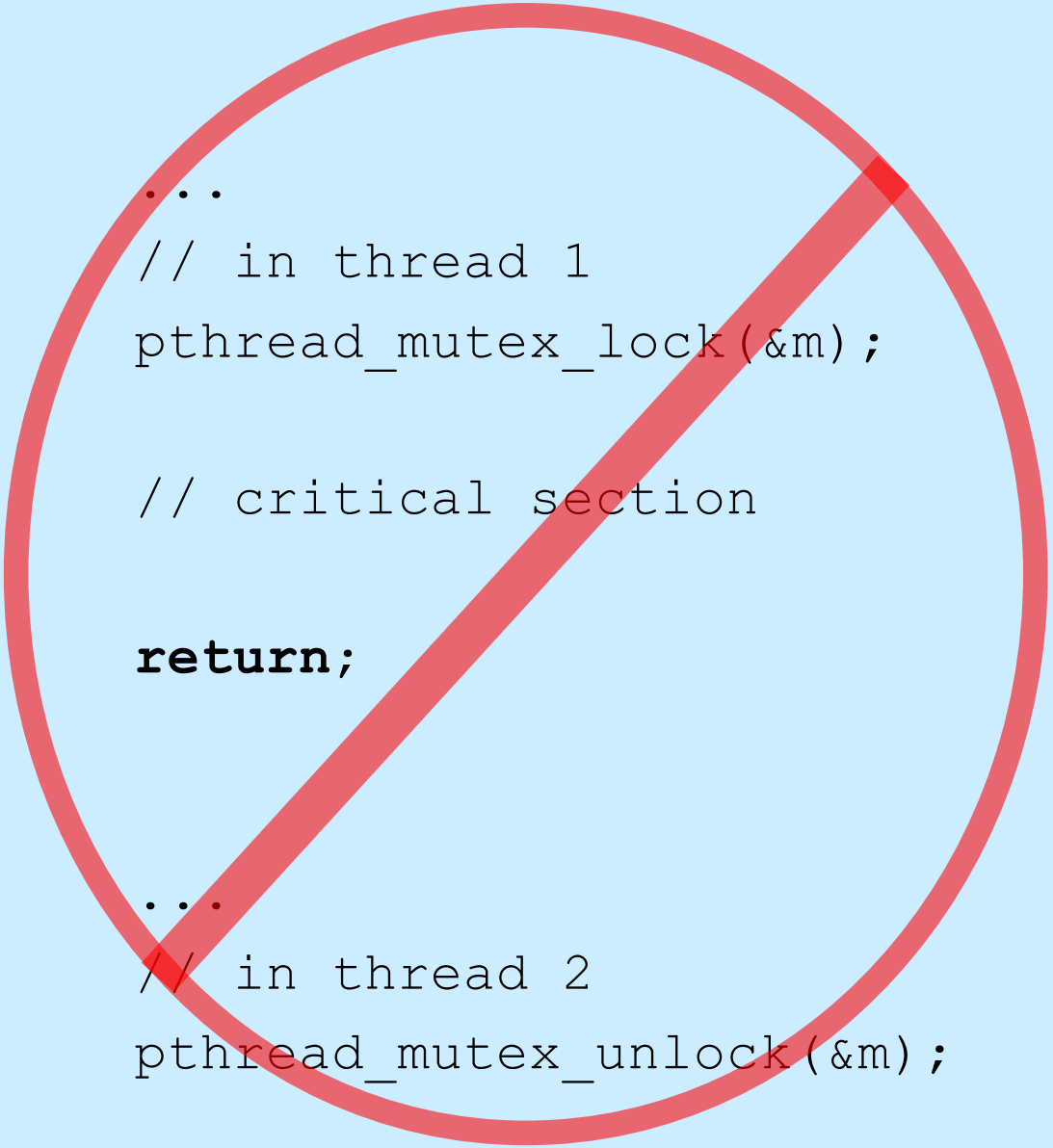
```
x = x+1;
 /*
  incr x
 */
```

POSIX Threads Mutual Exclusion

```
pthread_mutex_t m =
      PTHREAD_MUTEX_INITIALIZER;
      // shared by both threads
int x; // ditto

  pthread_mutex_lock(&m);

  x = x+1;

  pthread_mutex_unlock(&m);
```

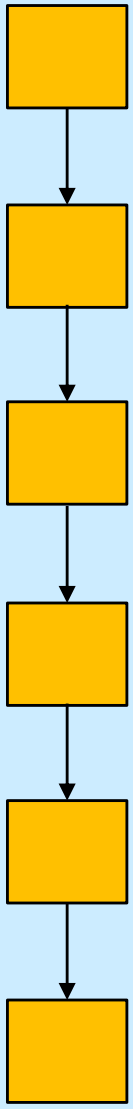
Correct Usage

```
pthread_mutex_lock(&m);

// critical section

pthread_mutex_unlock(&m);
```



```
...
// in thread 1
pthread_mutex_lock(&m);

// critical section

return;


...
// in thread 2
pthread_mutex_unlock(&m);
```

A Queue




```
void enqueue(node_t *item) {
    pthread_mutex_lock(&mutex);
    item->next = NULL;
    if (tail == NULL) {
        head = item;
        tail = item;
    } else {
        tail->next = item;
    }
    pthread_mutex_unlock(&mutex);
}
```

```
node_t *dequeue() {
    node_t *ret;
    pthread_mutex_lock(&mutex);
    if (head == NULL) {
        ret = NULL;
    } else {
        ret = head;
        head = head->next;
        if (head == NULL)
            tail = NULL;
    }
    pthread_mutex_unlock(&mutex);
    return ret;
}
```

Removing a Freelist Block

```
void pull_from_freelist(fblock_t *fbp) {
    ...
    fbp->blink->flink = fbp->flink;
    fbp->flink->blink = fbp->blink;
    ...
}
```

Parallelizing It

- Coarse grained
  - one mutex for the heap
  - threads lock the mutex before doing any operation
  - unlock it afterwards
  - only one thread at a time
- Fine grained
  - one mutex for each block
  - threads lock mutexes of only the blocks they are using
  - multiple threads at a time

Removing a Freelist Block: Coarse Grained

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&flist_mutex);
    ...
    fbp->blink->flink = fbp->flink;
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&flist_mutex);
}
```

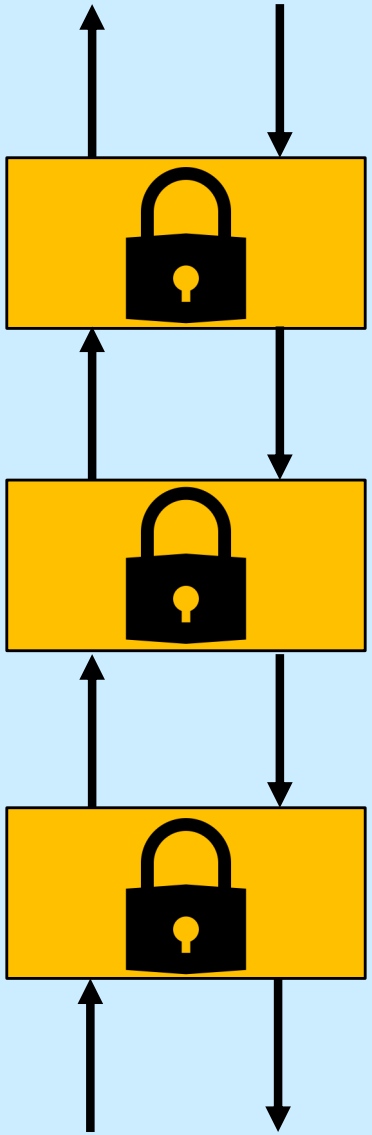
Removing a Freelist Block: Fine Grained (1)

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&fpp->mutex);
    ...
    fbp->blink->flink = fbp->flink;
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&fpp->mutex);
}
```

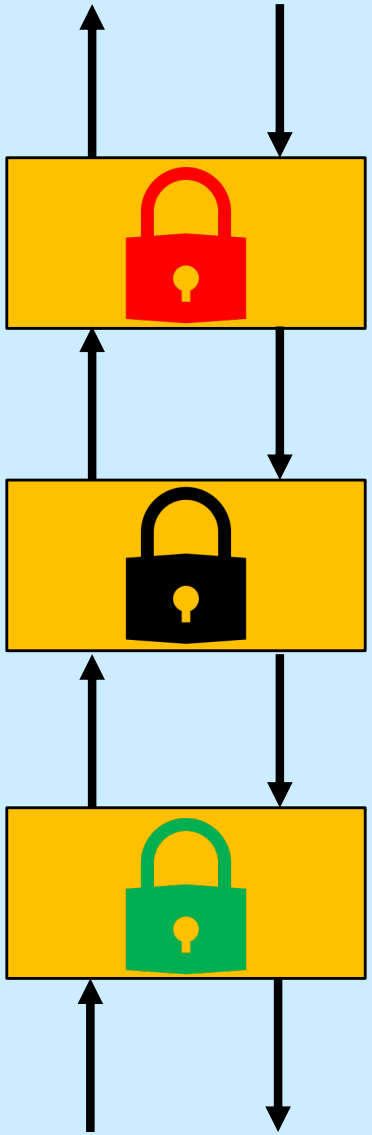
Removing a Freelist Block: Fine Grained (2)

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&fpp->mutex);
    ...
    pthread_mutex_lock(&fpp->blink->mutex);
    fbp->blink->flink = fbp->flink;
    pthread_mutex_lock(&fpp->flink->mutex);
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&fpp->blink->mutex);
    pthread_mutex_unlock(&fpp->flink->mutex);
    pthread_mutex_unlock(&fpp->mutex);
}
```

Multiple Pulls



Multiple Pulls

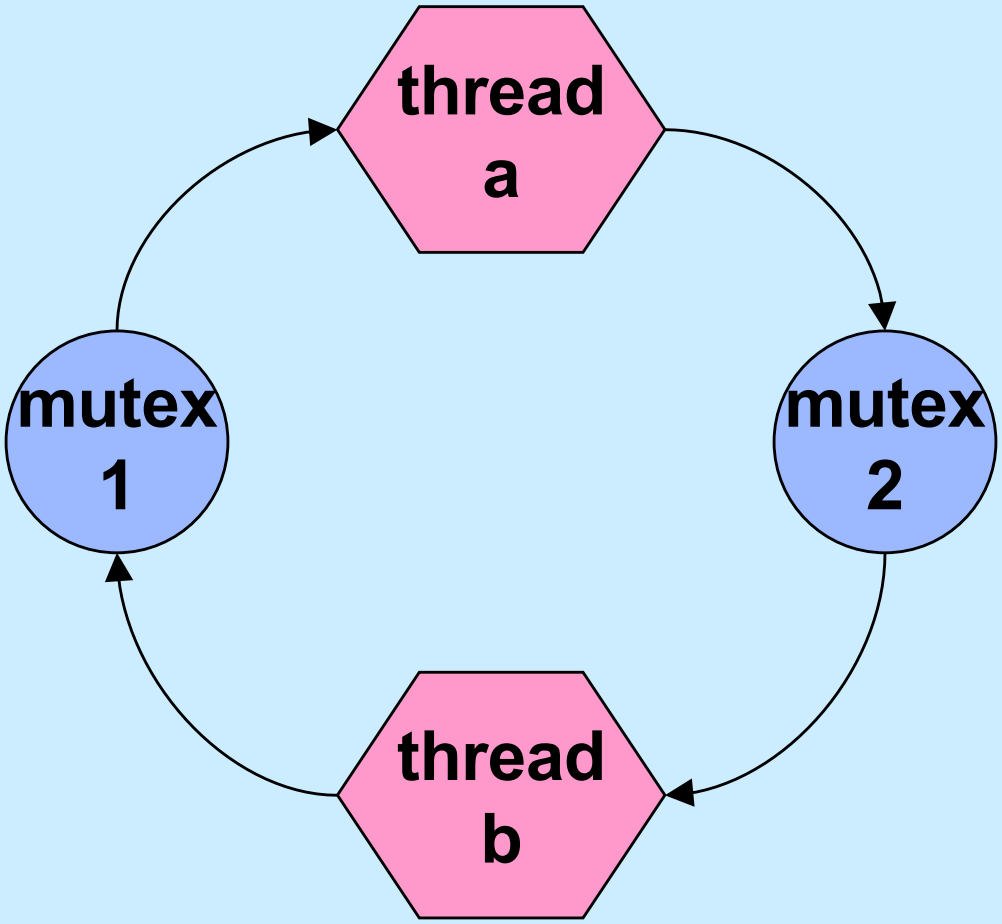


Taking Multiple Locks

```
func1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

```
func2( ) {
  pthread_mutex_lock(&m2);
  /* use object 2 */
  pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
}
```

Preventing Deadlock

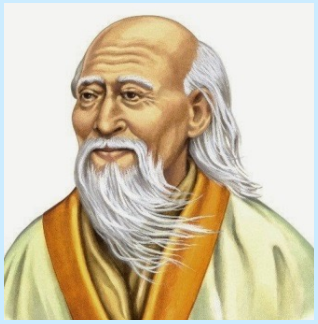


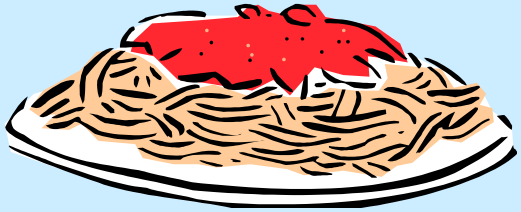
Taking Multiple Locks, Safely

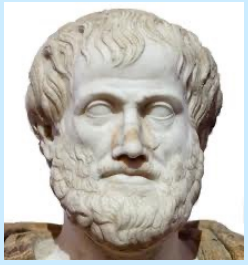
```
proc1( ) {
  pthread_mutex_lock(&m1);
  /* use object 1 */
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
}
```

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

Dining Philosophers Problem







Practical Issues with Mutexes

- Used a lot in multithreaded programs
  - speed is really important
    - » shouldn't slow things down much in the success case
  - checking for errors slows things down (a lot)
    - » thus errors aren't checked by default

Set Up

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

int pthread_mutex_destroy(pthread_mutex_t *mutexp)

int pthread_mutexattr_init(pthread_mutexattr_t *attrp)

int pthread_mutexattr_destroy(pthread_mutexattr_t *attrp)
```

Stupid (i.e., Common) Mistakes ...

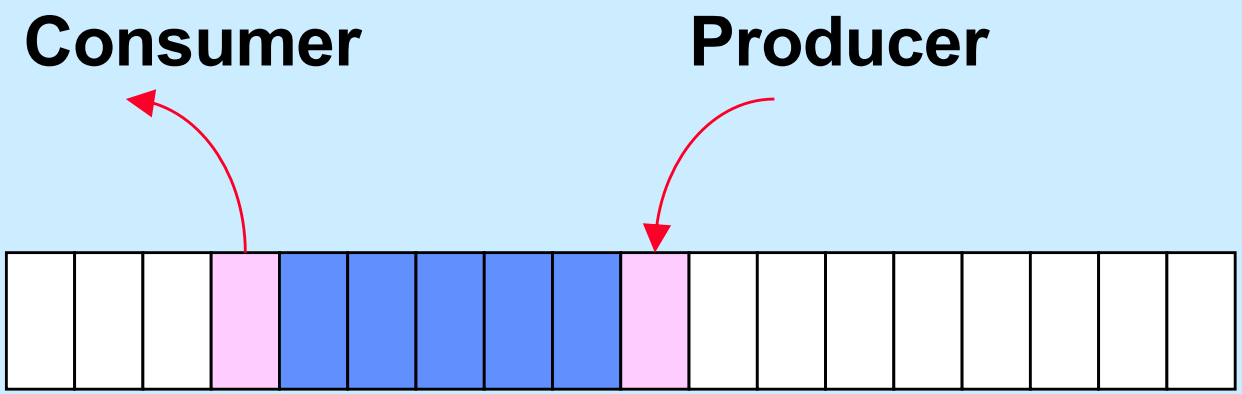
```
pthread_mutex_lock(&m1);
pthread_mutex_lock(&m1);
  // really meant to lock m2 ...
```

```
pthread_mutex_lock(&m1);
  ...
pthread_mutex_unlock(&m2);
  // really meant to unlock m1 ...
```

Runtime Error Checking

```
pthread_mutexattr_t err_chk_attr;
pthread_mutexattr_init(&err_chk_attr);
pthread_mutexattr_settype(&err_chk_attr,
       PTHREAD_MUTEX_ERRORCHECK);

pthread_mutex_t mut1;
pthread_mutex_init(&mut1, &err_chk_attr);

pthread_mutex_lock(&mut1);

if (pthread_mutex_lock(&mut1) == EDEADLK)
  fprintf(stderr, "error caught at runtime\n");

if (pthread_mutex_unlock(&mut2) == EPERM)
  fprintf(stderr, "another error: you didn't lock it!\n");
```

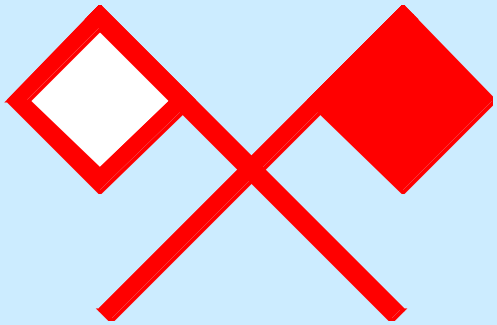
Producer-Consumer Problem



Guarded Commands

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

  ...

]
```

Semaphores



- **P(S) operation:**

```
when (S > 0) [
    S = S - 1;
]
```

- **V(S) operation:**

```
[S = S + 1;]
```

Quiz 2

```
semaphore S = 1;
int count = 0;

void func( ) {
   P(S);
   count++;
   ...
   count--;
   V(S);
}
```

The function func is called concurrently by n threads. What's the maximum value that count will take on?

a) 1
b) 2
c) n
d) indeterminate

- **P(S) operation:**

```
when (S > 0) [
   S = S - 1;
]
```

- **V(S) operation:**

```
[S = S + 1;]
```

Producer/Consumer with Semaphores

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

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

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

POSIX Semaphores

```
#include <semaphore.h>

int sem_init(sem_t *semaphore, int pshared, int init);
int sem_destroy(sem_t *semaphore);
int sem_wait(sem_t *semaphore);
    /* P operation */
int sem_trywait(sem_t *semaphore);
    /* conditional P operation */
int sem_post(sem_t *semaphore);
    /* V operation */
```

Producer-Consumer with POSIX Semaphores

```
sem_init(&empty, 0, BSIZE);
sem_init(&occupied, 0, 0);
int nextin = 0;
int nextout = 0;
```

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

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

Quiz 3

Does the POSIX version of the producer-consumer solution work with multiple producers and consumers?

a) **Yes**

b) **No, but it can be made to work by using mutexes to make sure that only one thread is executing the producer code at a time and only one thread is executing the consumer code at a time**

c) **It can't easily be made to work**

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s);

void start(state_t *s);

void stop(state_t *s);
```


Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  if (s->state == stopped)
    sleep();
}
void start(state_t *s) {
  state = started;
  wakeup_all();
}
void stop(state_t *s) {
  state = stopped;
}
```


Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  if (s->state == stopped) {
    pthread_mutex_unlock(&s->mutex);
    sleep();
  else pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  state = started;
  wakeup_all();
  pthread_mutex_unlock(&s->mutex);
}
```

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  if (s->state == stopped) {
    sleep();
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  state = started;
  wakeup_all();
  pthread_mutex_unlock(&s->mutex);
}
```


Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

Condition Variables

```
when (guard) [
  statement 1;
  …
  statement n;
]
```

```
pthread_mutex_lock(&mutex);
while(!guard)
  pthread_cond_wait(
     &cond_var, &mutex);
statement 1;
…
statement n;
pthread_mutex_unlock(&mutex);
```

```
// code modifying the guard:
…
```

```
pthread_mutex_lock(&mutex);
// code modifying the guard:
…
pthread_cond_broadcast(
    &cond_var);
pthread_mutex_unlock(&mutex);
```

Set Up

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

int pthread_cond_destroy(pthread_cond_t *cvp)

int pthread_condattr_init(pthread_condattr_t *attrp)

int pthread_condattr_destroy(pthread_condattr_t *attrp)
```

PC with Condition Variables (1)

```
typedef struct buffer {
   pthread_mutex_t m;
   pthread_cond_t  more_space;
   pthread_cond_t  more_items;
   int             next_in;
   int             next_out;
   int             empty;
   char            buf[BSIZE];
} buffer_t;
```

PC with Condition Variables (2)

```
void produce(buffer_t *b,
     char item) {

  pthread_mutex_lock(&b->m);
  while (!(b->empty > 0))
    pthread_cond_wait(
        &b->more_space, &b->m);
  b->buf[b->nextin] = item;
  if (++(b->nextin) == BSIZE)
    b->nextin = 0;
  b->empty--;
  pthread_cond_signal(
     &b->more_items);
  pthread_mutex_unlock(&b->m);
}
```

```
char consume(buffer_t *b) {
  char item;
  pthread_mutex_lock(&b->m);
  while (!(b->empty < BSIZE))
    pthread_cond_wait(
        &b->more_items, &b->m);
  item = b->buf[b->nextout];
  if (++(b->nextout) == BSIZE)
    b->nextout = 0;
  b->empty++;
  pthread_cond_signal(
     &b->more_space);
  pthread_mutex_unlock(&b->m);
  return item;
}
```