Synchronization Primitives

mutex, semaphore, condition variable

What is the final value of `data` if the threads run without preemption?

Race Conditions

```
int data = 0;

void *thread1(void* arg) {
        int a = data;
        a++;
        data = a;
        return NULL;
}

void *thread2(void *arg) {
        int b = data;
        b--;
        data = b;
        return NULL;
}

int main() {
        // Create and run both threads
}
```

Race Conditions

What is the final value of `data` if the threads run without preemption?

```c
int data = 0;

void *thread1(void* arg) {
        int a = data;
        a++;
        data = a;
        return NULL;
}

void *thread2(void *arg) {
        int b = data;
        b--;
        data = b;
        return NULL;
}

int main() {
        // Create and run both threads
}
```

Step	thread1	thread2	data
			0
1	`int a = data;`		0
2	`a++; // 1`		0
3		`int b = data;`	0
4		`b--; // -1`	0
5	`data = a;`		1
6		`data = b;`	-1

The final value for data could be: 1, 0, or -1.

What is the final value of `data` if the threads run without preemption?

How could we fix this program?

Race Conditions

```
int data = 0;
```
create lock

```
void *thread1(void* arg) {
```
acquire
```
    int a = data;
    a++;
    data = a;
```
release
```
    return NULL;
}

void *thread2(void *arg) {
```
acquire
```
    int b = data;
    b--;
    data = b;
```
release
```
    return NULL;
}

int main() {
    // Create and run both threads
}
```

Step	thread1	thread2	data
			0
1	`int a = data;`		0
2	`a++; // 1`		0
3		`int b = data;`	0
4		`b--; // -1`	0
5	`data = a;`		1
6		`data = b;`	-1

The final value for data could be: 1, 0, or -1.

Problems with Locks

- Problem 1: correct synchronization with locks is difficult

- Problem 2: locks are slow
  - Threads that fail to acquire a lock on the first attempt must "spin", which wastes CPU cycles
    - Replace no-op with `yield()`
  - Threads get scheduled and de-scheduled while the lock is still locked
    - Need a better synchronization primitive

Semaphores (`semaphore.h`)

- Semaphores are stateful synchronization primitives with:
  - a value `n` (non-negative integer),
  - a lock, and
  - a queue.

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`

- `int sem_wait(sem_t *sem);`
  - If s is nonzero, the P decrements s and returns immediately. If s is zero, then adds the thread to queue(s); after restarting, the P operation decrements s and returns.

- `int sem_post(sem_t *sem);`
  - Increments s by 1. If there are any threads in queue(s), then V restarts exactly one of these threads, which then completes the P operation.

```
sem_init(&sem_name, 0, 10);
```

Semantics of `wait` and `post`

OS

- `sem_wait`
  - block (suspend thread) until value `n` > 0
  - when `n` > 0, decrement `n` by 1

```
sem_wait(sem_t *s){
  while(s->n == 0){}
  s->n -= 1
}
```

- `sem_post`
  - increment value `n` by 1
  - resume a thread waiting on s (if any)

OS

```
sem_post(sem_t *s){
  s->n += 1
}
```

Binary Semaphore (aka mutex)

- A binary semaphore is a semaphore whose value is always 0 or 1
- Used for mutual exclusion---it's a more efficient lock!

```
sem_t s
sem_init(&s, 0, 1)
```

T1

```
sem_wait(&s)
CriticalSection()
sem_post(&s)
```

T2

```
sem_wait(&s)
CriticalSection()
sem_post(&s)
```

Example: Shared counter

H (head): code **before** the critical section
L (load): loading a critical value
U (update): updating a critical value
S (store): store a critical value
T (tail): code **after** the critical section

```
volatile long cnt = 0;
```

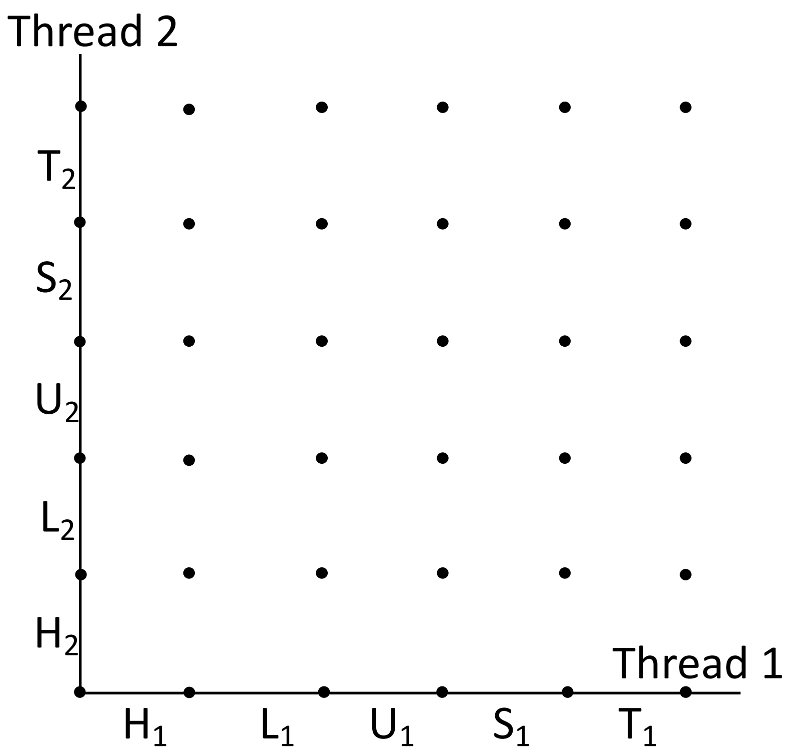
```
/* Thread routine */
void *thread(void *vargp)
{
  long num_iters = *((long *)vargp);

  for (long i = 0; i < num_iters; i++){

    cnt++;

  }

  return NULL;
}
```



Example: Shared counter

H (head): code **before** the critical section
L (load): loading a critical value
U (update): updating a critical value
S (store): store a critical value
T (tail): code **after** the critical section

```
volatile long cnt = 0;
```

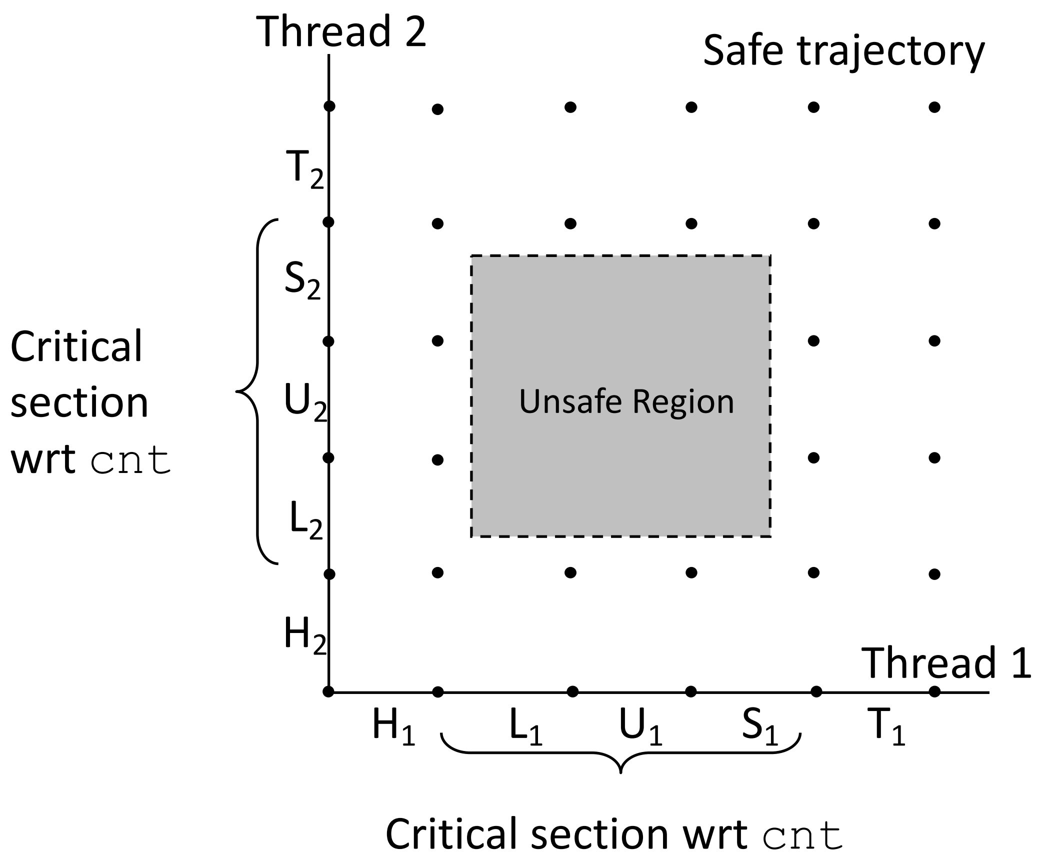
```
/* Thread routine */
void *thread(void *vargp)
{
  long num_iters = *((long *)vargp);

  for (long i = 0; i < num_iters; i++){

    cnt++;

  }

  return NULL;
}
```



Example: Shared counter

H (head): code **before** the critical section
L (load): loading a critical value
U (update): updating a critical value
S (store): store a critical value
T (tail): code **after** the critical section

```
volatile long cnt = 0;
```

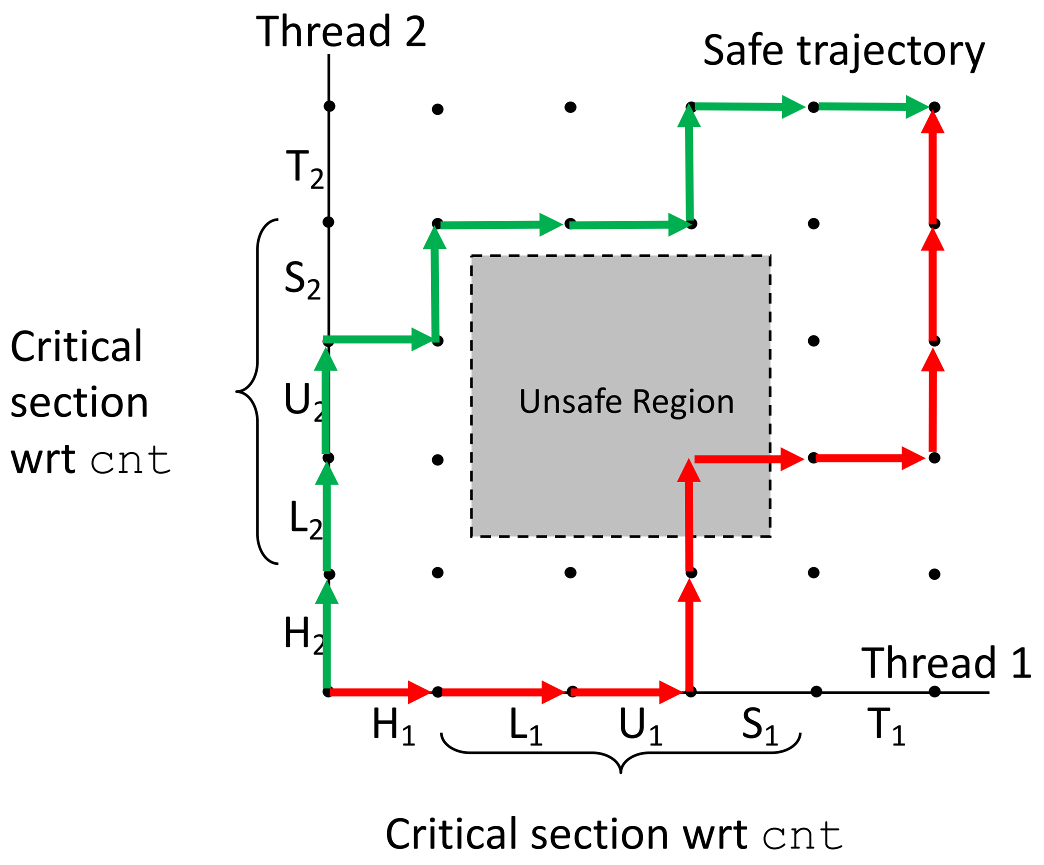
```
/* Thread routine */
void *thread(void *vargp)
{
  long num_iters = *((long *)vargp);

  for (long i = 0; i < num_iters; i++){

    cnt++;

  }

  return NULL;
}
```



What is the semaphore value n at each point?

Example: Shared counter

H (head): code **before** the critical section
L (load): loading a critical value
U (update): updating a critical value
S (store): store a critical value
T (tail): code **after** the critical section

```
volatile long cnt = 0;
sem_t s;
sem_init(&s, 0, 1);
```

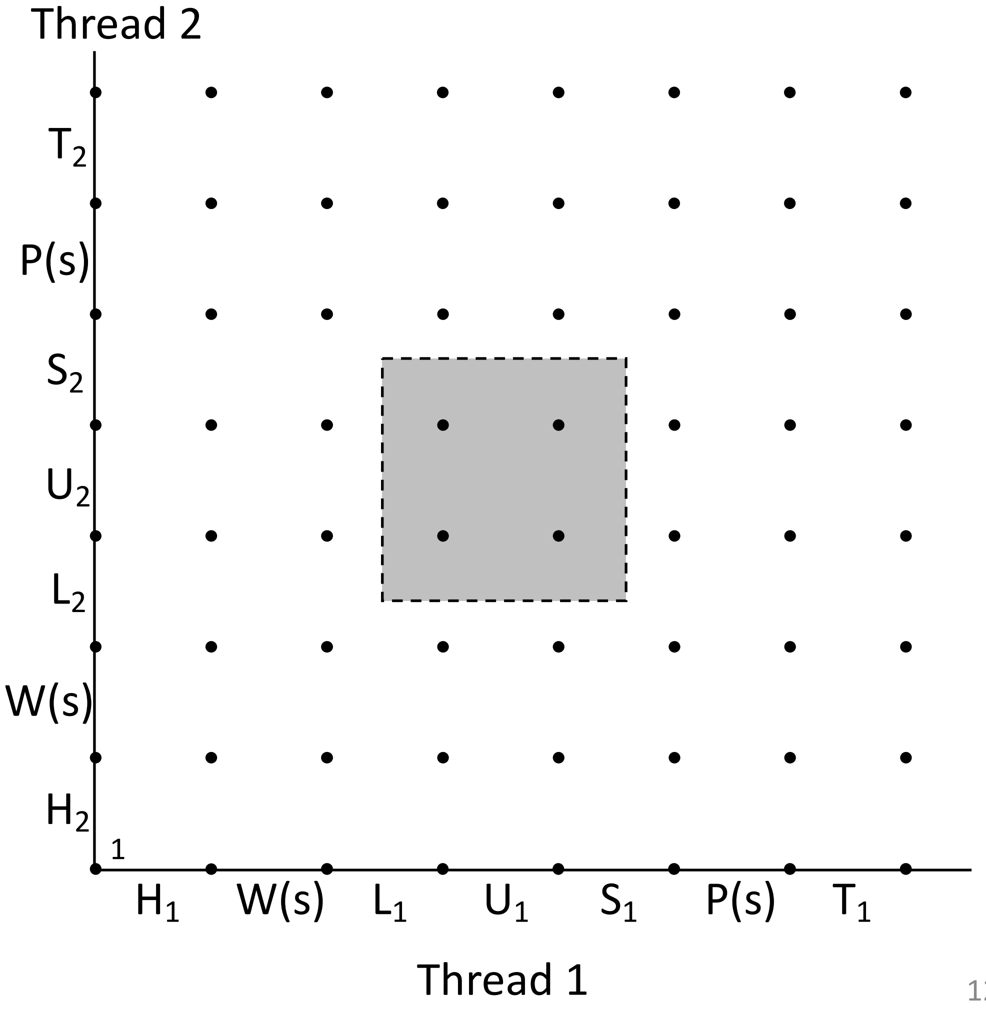
```
/* Thread routine */
void *thread(void *vargp)
{
  long num_iters = *((long *)vargp);

  for (long i = 0; i < num_iters; i++){
    sem_wait(&s);
    cnt++;
    sem_post(&s);
  }

  return NULL;
}
```

```
sem_wait(sem_t *s){
  while(s->n == 0){}
  s->n -= 1
}

sem_post(sem_t *s){
  s->n += 1
}
```



What is the semaphore value n at each point?

Example: Shared counter

H (head): code **before** the critical section
L (load): loading a critical value
U (update): updating a critical value
S (store): store a critical value
T (tail): code **after** the critical section

```
volatile long cnt = 0;
sem_t s;
sem_init(&s, 0, 1);
```

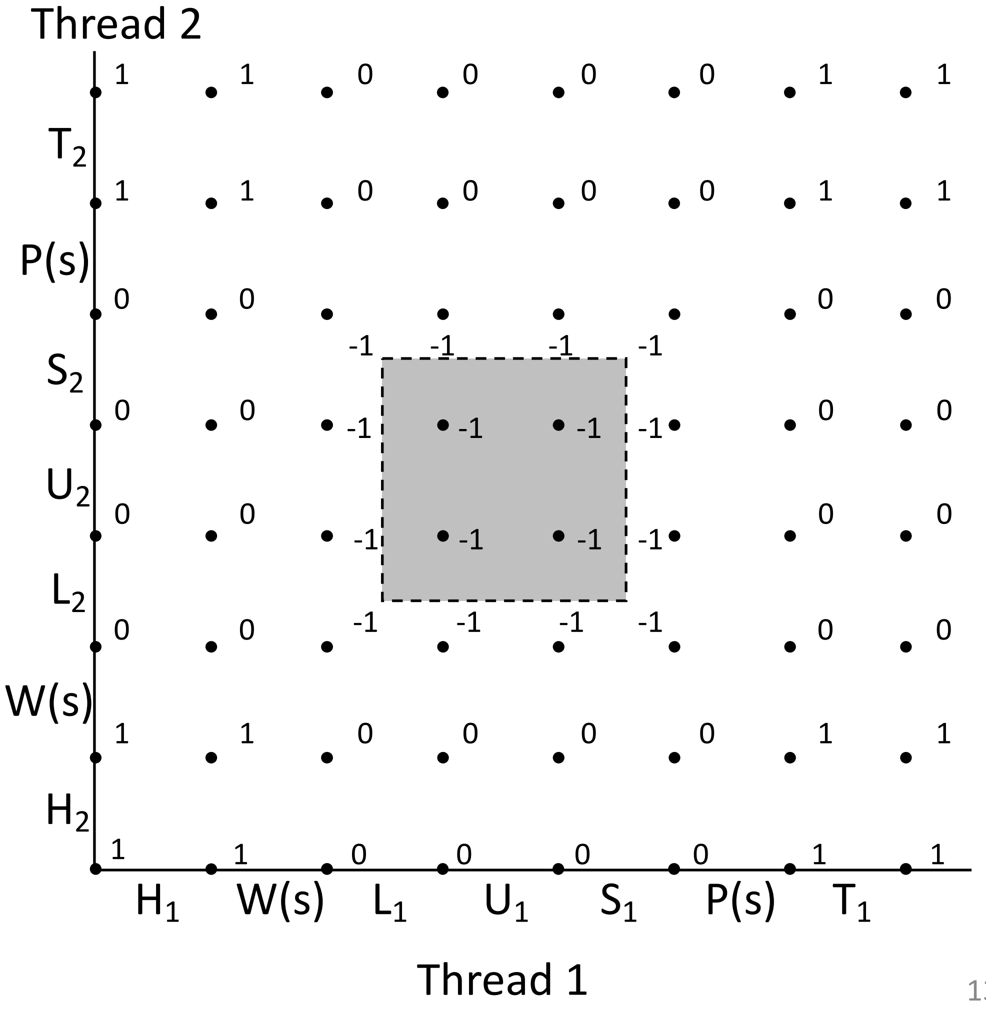
```
/* Thread routine */
void *thread(void *vargp)
{
  long num_iters = *((long *)vargp);

  for (long i = 0; i < num_iters; i++){
    sem_wait(&s);
    cnt++;
    sem_post(&s);
  }

  return NULL;
}
```

```
sem_wait(sem_t *s){
  while(s->n == 0){}
  s->n -= 1
}

sem_post(sem_t *s){
  s->n += 1
}
```



What is the semaphore value n at each point?

Example: Shared counter

H (head): code **before** the critical section
L (load): loading a critical value
U (update): updating a critical value
S (store): store a critical value
T (tail): code **after** the critical section

```
volatile long cnt = 0;
sem_t s;
sem_init(&s, 0, 1);
```

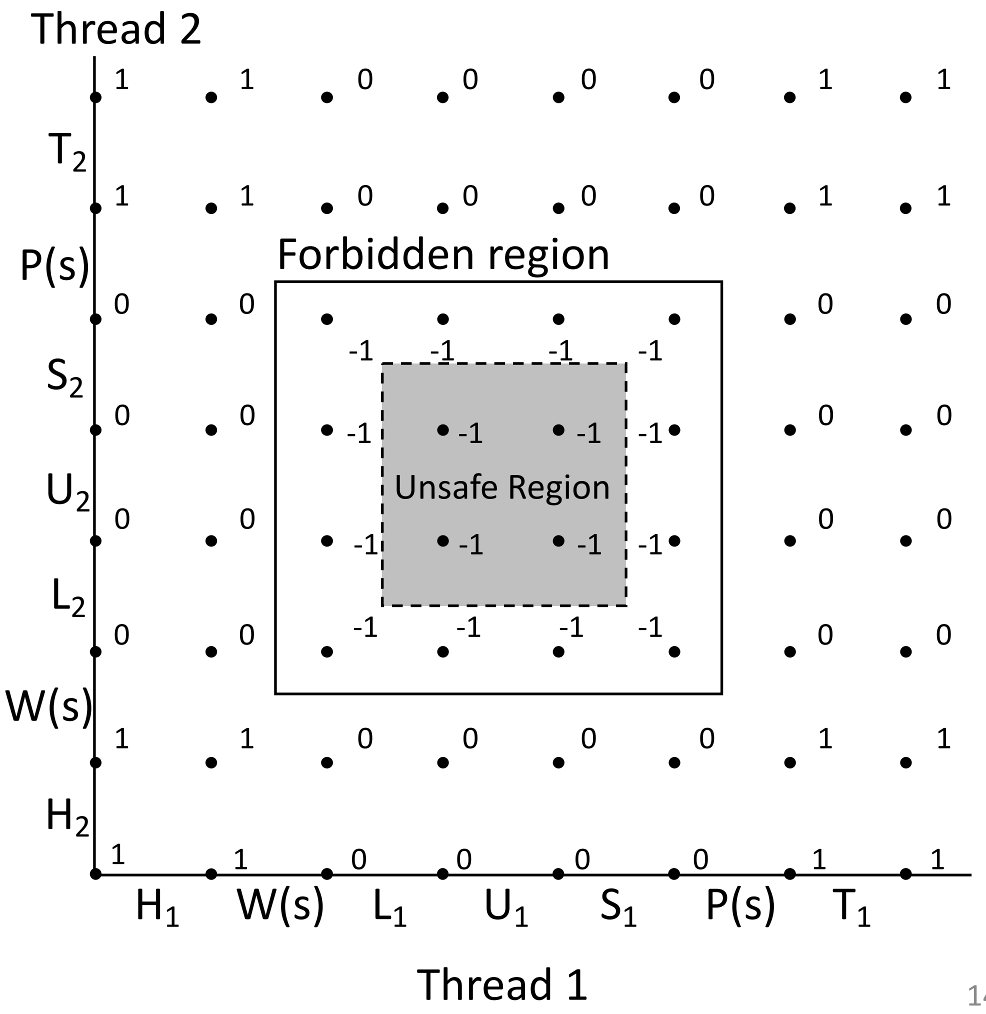
```
/* Thread routine */
void *thread(void *vargp)
{
  long num_iters = *((long *)vargp);

  for (long i = 0; i < num_iters; i++){
    sem_wait(&s);
    cnt++;
    sem_post(&s);
  }

  return NULL;
}
```

```
sem_wait(sem_t *s){
  while(s->n == 0){}
  s->n -= 1
}

sem_post(sem_t *s){
  s->n += 1
}
```



Example: Synchronization Barrier

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data.
- Results can be safely combined once all threads end.
  - MapReduce is an example of this!
- To do this safely, we need a way to check whether all threads are done.

```
volatile int results = 0;
volatile int done_count = 0;

sem_t done_count_semaphore;
sem_init(&done_count_semaphore, 0, 1);

sem_t barrier;
sem_init(&barrier, 0, 0);
```

```
void *thread(void *args){
    parallel_computation(args);

    sem_wait(&done_count_semaphore);
    done_count++;
    sem_post(&done_count_semaphore);

    if(done_count == n){
        sem_post(&barrier);
    }

    sem_wait(&barrier);
    sem_post(&barrier);
    use_results();
}
```

Counting Semaphores

- A semaphore with a value that goes above 1 is called a counting semaphore

- A more flexible primitive for mediating access to shared resources

Example: Bounded Buffers

finite capacity (e.g., 20 loaves)
implemented as a queue

Nobody waits until the queue is empty

Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

Example: Bounded Buffers

finite capacity (e.g., 20 loaves)
implemented as a queue

Nobody waits until the queue is empty

Separation of concerns:
1. How do you implement a bounded (circular) buffer?
2. How do you synchronize concurrent access to a bounded buffer?

Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

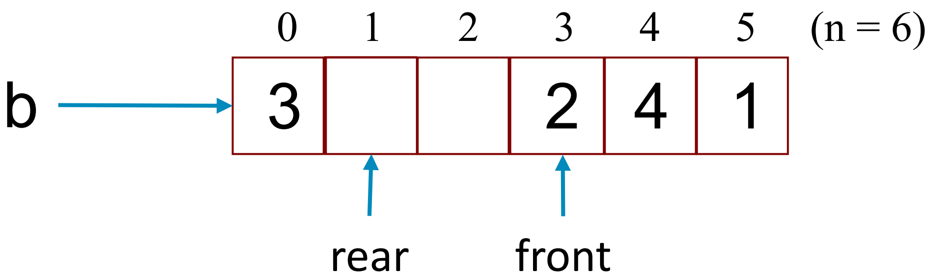
Example: Bounded Buffers

0	1	2	3	4	5
3			2	4	1

Values wrap around!!

```
typedef struct {
  int *b;         // ptr to buffer containing the queue
  int n;          // length of array (max # slots)
  int front;      // index of first element, 0 <= front < n
  int rear;       // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

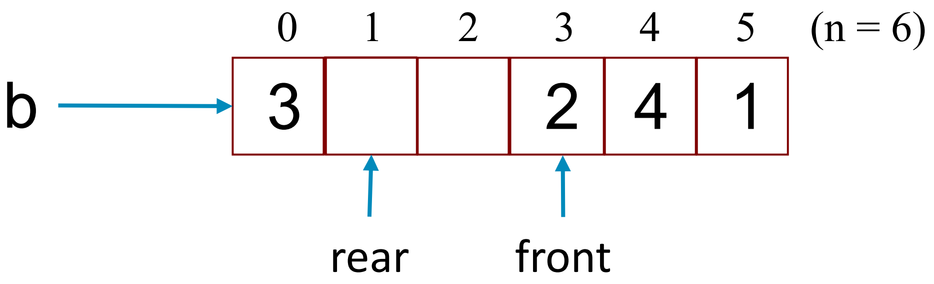
Example: Bounded Buffers



Values wrap around!!

```
typedef struct {
  int *b;        // ptr to buffer containing the queue
  int n;         // length of array (max # slots)
  int front;     // index of first element, 0 <= front < n
  int rear;      // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

Example: Bounded Buffers

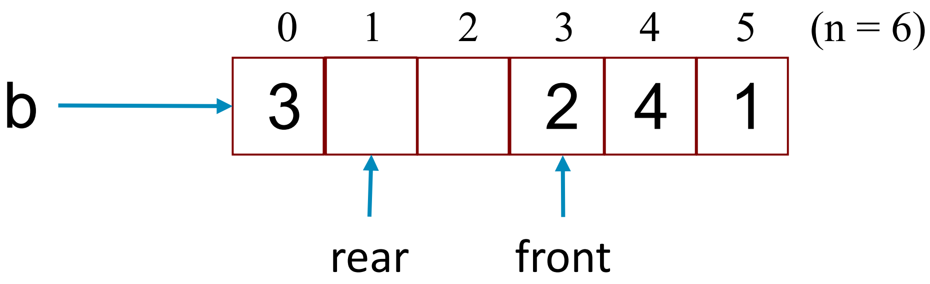


Values wrap around!!

```
typedef struct {
  int *b;        // ptr to buffer containing the queue
  int n;         // length of array (max # slots)
  int front;     // index of first element, 0 <= front < n
  int rear;      // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

Example: Bounded Buffers



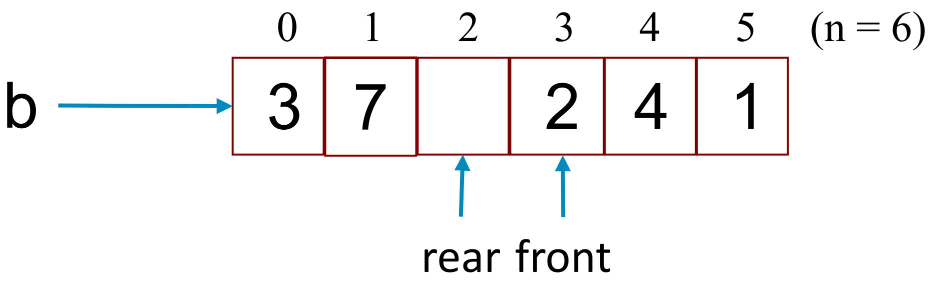
Values wrap around!!

```
typedef struct {
  int *b;        // ptr to buffer containing the queue
  int n;         // length of array (max # slots)
  int front;     // index of first element, 0 <= front < n
  int rear;      // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```
void put(bbuf_t * ptr, int val){
  ptr->b[ptr->rear] = val;
  ptr->rear = ((ptr->rear)+1)%(ptr->n);
}
```

```
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

Example: Bounded Buffers



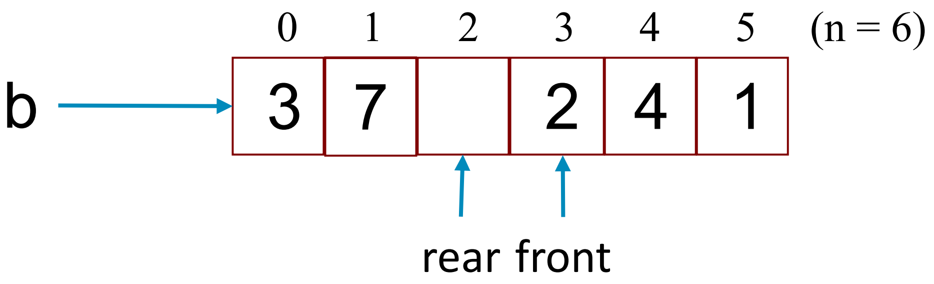
Values wrap around!!

```c
typedef struct {
  int *b;       // ptr to buffer containing the queue
  int n;        // length of array (max # slots)
  int front;    // index of first element, 0 <= front < n
  int rear;     // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```c
void put(bbuf_t * ptr, int val){
  ptr->b[ptr->rear] = val;
  ptr->rear = ((ptr->rear)+1)%(ptr->n);
}
```

```c
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

Example: Bounded Buffers



Values wrap around!!

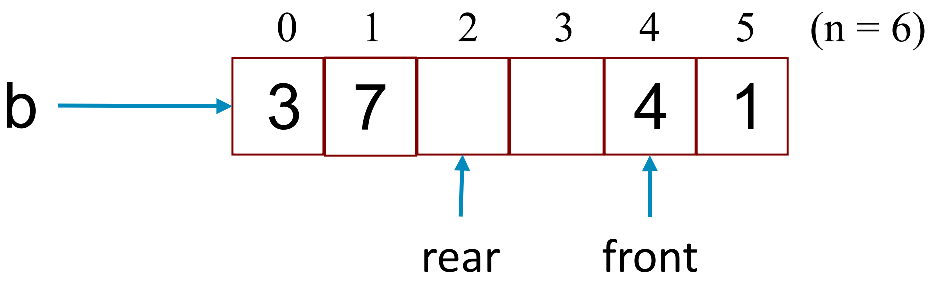
```
typedef struct {
  int *b;         // ptr to buffer containing the queue
  int n;          // length of array (max # slots)
  int front;      // index of first element, 0 <= front < n
  int rear;       // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

```
void put(bbuf_t * ptr, int val){
  ptr->b[ptr->rear] = val;
  ptr->rear = ((ptr->rear)+1)%(ptr->n);
}
```

```
int get(bbuf_t * ptr){
  int val = ptr->b[ptr->front];
  ptr->front = ((ptr->front)+1)%(ptr->n);
  return val;
}
```

Example: Bounded Buffers



Values wrap around!!

```
typedef struct {
  int *b;         // ptr to buffer containing the queue
  int n;          // length of array (max # slots)
  int front;      // index of first element, 0 <= front < n
  int rear;       // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

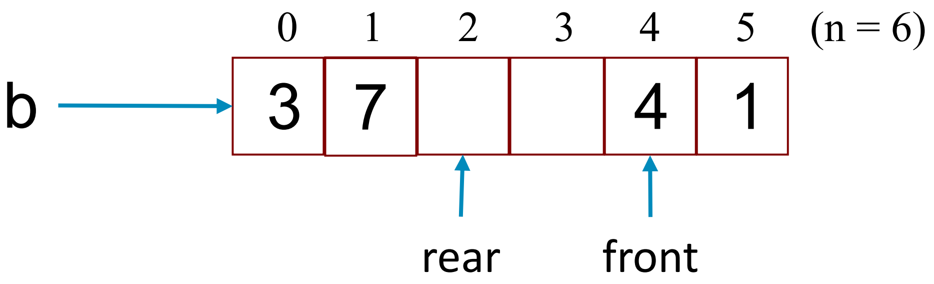
```
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

```
void put(bbuf_t * ptr, int val){
  ptr->b[ptr->rear] = val;
  ptr->rear = ((ptr->rear)+1)%(ptr->n);
}
```

```
int get(bbuf_t * ptr){
  int val = ptr->b[ptr->front];
  ptr->front = ((ptr->front)+1)%(ptr->n);
  return val;
}
```

Example: Bounded Buffers

What can go wrong if multiple threads are using the buffer?



Values wrap around!!

```
typedef struct {
  int *b;        // ptr to buffer containing the queue
  int n;         // length of array (max # slots)
  int front;     // index of first element, 0 <= front < n
  int rear;      // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

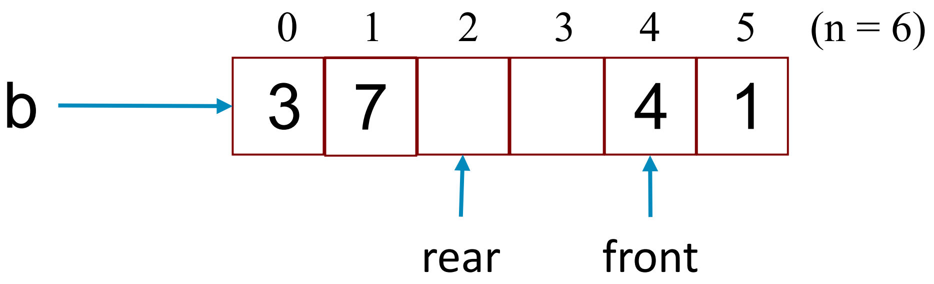
```
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

```
void put(bbuf_t * ptr, int val){
  ptr->b[ptr->rear] = val;
  ptr->rear = ((ptr->rear)+1)%(ptr->n);
}
```

```
int get(bbuf_t * ptr){
  int val = ptr->b[ptr->front];
  ptr->front = ((ptr->front)+1)%(ptr->n);
  return val;
}
```

Example: Bounded Buffers

What can go wrong if multiple threads are using the buffer?



Values wrap around!!

```
typedef struct {
  int *b;         // ptr to buffer containing the queue
  int n;          // length of array (max # slots)
  int front;      // index of first element, 0 <= front < n
  int rear;       // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

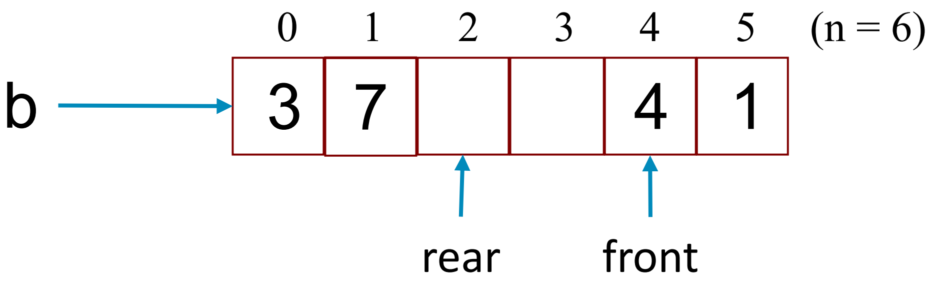
```
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

```
void put(bbuf_t * ptr, int val){
  ptr->b[ptr->rear] = val;
  ptr->rear = ((ptr->rear)+1)%(ptr->n);
}
```

```
int get(bbuf_t * ptr){
  int val = ptr->b[ptr->front];
  ptr->front = ((ptr->front)+1)%(ptr->n);
  return val;
}
```

Example: Bounded Buffers

What can go wrong if multiple threads are using the buffer?



Values wrap around!!

Note: we can do a bit better than this if we consider that `put` and `get` operate on different sections of the buffer.

```
typedef struct {
  int *b;        // ptr to buffer containing the queue
  int n;         // length of array (max # slots)
  int front;     // index of first element, 0 <= front < n
  int rear;      // (index of last elem)+1 % n, 0 <= rear < n
} bbuf_t
```

```
sem_t mutex;
```

```
void init(bbuf_t * ptr, int n) {
  ptr->b = malloc(n*sizeof(int));
  ptr->n = n;
  ptr->front = 0;
  ptr->rear = 0;
}
```

```
sem_init(&mutex, 0, 1);
```

```
void put(bbuf_t * ptr, int val
  ptr->b[ptr->rear] = val;
  ptr->rear = ((ptr->rear)+1)%(ptr->n);
}
```

```
sem_wait(&(ptr->slots));
sem_post(&(ptr->mutex));
```

```
int get(bbuf_t * ptr){
  int val = ptr->b[ptr->front];
  ptr->front = ((ptr->front)+1)%(ptr->n);
  return val;
}
```

```
sem_wait(&(ptr->slots));
sem_post(&(ptr->mutex));
```

Practice with multiple Readers/Writers

- Consider a collection of concurrent threads that have access to a shared object
- Some threads are readers, some threads are writers
  - an unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```
// global variables
int num_readers = 0;
```

```
void init(){

}
```

```
int reader(void *shared){

    num_readers++;



    int x =  read(shared);


    num_readers--;



    return x
}
```

```
void writer(void *shared, int val){

    write(shared, val);

}
```

Practice with multiple Readers/Writers

- Consider a collection of concurrent threads that have access to a shared object

- Some threads are readers, some threads are writers
  - an unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```
// global variables
int num_readers = 0;
sem_t num_lock;
sem_t ojb_lock;
```

```
void init(){
    sem_init(&num_lock, 0, 1);
    sem_init(&ojb_lock, 0, 1);
}
```

```
int reader(void *shared){
    sem_wait(&num_lock);
    num_readers++;
    if (num_readers == 1)
        sem_wait(&obj_lock);
    sem_post(&num_lock);

    int x =  read(shared);

    sem_wait(&num_lock);
    num_readers--;
    if (num_readers == 0)
        sem_post(&obj_lock);
    sem_post(&num_lock);

    return x
}
```

```
void writer(void *shared, int val){
    sem_wait(&ojb_lock);
    write(shared, val);
    sem_post(&ojb_lock);
}
```

This thread is the first reader.

This thread was the last reader.

Limitations of Semaphores

- Semaphores are a very spartan mechanism
  - They are simple, and have few features
  - More designed for proofs than synchronization
- They lack many practical synchronization features
  - It is easy to deadlock with semaphores
  - One cannot check the lock without blocking
  - They are flat out hard to use
  - POSIX doesn’t even include an implementation
- Strange interactions with OS scheduling (priority inheritance)

Condition Variables

- A condition variable (CV) is a stateless synchronization primitive that is used in combination with locks (mutexes)

- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - atomically releases the lock, suspends execution of the calling thread, and places that thread on cv's waitlist
  - after the thread is awoken, it re-acquires the lock before returning

- `int pthread_cond_signal(pthread_cond_t *cond);`
  - takes one thread off CV waitlist and marks it as eligible to run (No-op if waitlist is empty.)

- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - takes all threads off CV waitlist and marks them all as eligible to run (No-op if waitlist is empty.)

Using Condition Variables

- Add a lock. *Each shared value needs a lock to enforce mutually exclusive access to the shared value.*

- Add code to acquire and release the lock. *All code access the shared value must hold the objects lock.*

- Identify and add condition variables. *A good rule of thumb is to add a condition variable for each situation in a function might need to wait.*

- Add loops to wait. *Threads might not be scheduled immediately after they are eligible to run. Even if a condition was true when signal/broadcast was called, <u>it might not be true when a thread resumes execution</u>.*

Example: Synchronization Barrier (Semaphore)

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data.
- Results can be safely combined once all threads end.
  - MapReduce is an example of this!
- To do this safely, we need a way to check whether all threads are done.

```
volatile int results = 0;
volatile int done_count = 0;

sem_t done_count_semaphore;
sem_init(&done_count_semaphore, 0, 1);

sem_t barrier;
sem_init(&barrier, 0, 0);
```

```
void *thread(void *args){
    parallel_computation(args);

    sem_wait(&done_count_semaphore);
    done_count++;
    sem_post(&done_count_semaphore);

    if(done_count == n){
        sem_post(&barrier);
    }

    sem_wait(&barrier);
    sem_post(&barrier);
    use_results();
}
```

Example: Synchronization Barrier (CV)

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data.

- Results can be safely combined once all threads end.
  - MapReduce is an example of this!

- To do this safely, we need a way to check whether all threads are done.

```
volatile int results = 0;
volatile int done_count = 0;

sem_t done_count_semaphore;
sem_init(&done_count_semaphore, 0, 1);

sem_t barrier;
sem_init(&barrier, 0, 0);
```

```
void *thread(void *args){
    parallel_computation(args);

    sem_wait(&done_count_semaphore);
    done_count++;
    sem_post(&done_count_semaphore);

    if(done_count == n){
        sem_post(&barrier);
    }

    sem_wait(&barrier);
    sem_post(&barrier);
    use_results();
}
```

Example: Synchronization Barrier (CV)

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data.
- Results can be safely combined once all threads end.
  - MapReduce is an example of this!
- To do this safely, we need a way to check whether all threads are done.

```
volatile int results = 0;
volatile int done_count = 0;
```

```
void *thread(void *args){
    parallel_computation(args);


    done_count++;








    use_results();
}
```

Example: Synchronization Barrier (CV)

- With data parallel programming, a computation proceeds in parallel, with each thread operating on a different section of the data.
- Results can be safely combined once all threads end.
  - MapReduce is an example of this!
- To do this safely, we need a way to check whether all threads are done.

```
volatile int results = 0;
volatile int done_count = 0;

pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t all_done =
    PTHREAD_COND_INITIALIZER;
```

```
void *thread(void *args){
    parallel_computation(args);

    pthread_mutex_lock(&lock);
    done_count++;

    if (done_count < n){
        pthread_cond_wait(&all_done, &lock);
    } else {
        pthread_cond_broadcast(&all_done);
    }

    pthread_mutex_unlock(&lock);
    use_results();
}
```

Practice with multiple Readers/Writers (Sema)

- Consider a collection of concurrent threads that have access to a shared object
- Some threads are readers, some threads are writers
  - an unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```c
// global variables
int num_readers = 0;
sem_t num_lock;
sem_t ojb_lock;
```

```c
void init(){
    sem_init(&num_lock, 0, 1);
    sem_init(&ojb_lock, 0, 1);
}
```

```c
int reader(void *shared){
    sem_wait(&num_lock);
    num_readers++;
    if (num_readers == 1)
        sem_wait(&obj_lock);
    sem_post(&num_lock);

    int x =  read(shared);

    sem_wait(&num_lock);
    num_readers--;
    if (num_readers == 0)
        sem_post(&obj_lock);
    sem_post(&num_lock);

    return x
}
```

```c
void writer(void *shared, int val){
    sem_wait(&ojb_lock);
    write(shared, val);
    sem_post(&ojb_lock);
}
```

This thread is the first reader.

This thread was the last reader.

Practice with multiple Readers/Writers (CV)

- Consider a collection of concurrent threads that have access to a shared object
- Some threads are readers, some threads are writers
  - an unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```
// global variables
int num_readers = 0;
```

```
void init(){

}
```

```
int reader(void *shared){

    num_readers++;

    int x =  read(shared);

    num_readers--;

    return x
}
```

```
void writer(void *shared, int val){

    write(shared, val);

}
```

Implement this program using condition variables.

Practice with multiple Readers/Writers (CV)

- Consider a collection of concurrent threads that have access to a shared object
- Some threads are readers, some threads are writers
  - an unlimited number of readers can access the object at same time
  - a writer must have exclusive access to the object

```
// global variables
int num_readers = 0;
int num_writers = 0;
pthread_mutex_t lock;
pthread_cond_t readable;
pthread_cond_t writeable;
```

```
void init(){
    lock = PTHREAD_MUTEX_INITIALIZER;
    readable = PTHREAD_COND_INITIALIZER;
    writeable = PTHREAD_COND_INITIALIZER;
}
```

```
int reader(void *shared){
    // Increment num_readers; lock shared
    pthread_mutex_lock(&lock);
    while(num_writers > 0)
        pthread_cond_wait(&readable, &lock);
    num_readers++;
    pthread_mutex_unlock(&lock);

    // Read from shared resource
    int x =  read(shared);

    // Decrement num_readers; unlock shared
    pthread_mutex_lock(&lock);
    num_readers--;
    if(num_readers == 0)
        pthread_cond_signal(writeable);

    pthread_mutex_unlock(&lock);
    return x
}
```

```
void writer(void *shared, int val){
    // Wait for shared resource
    pthread_mutex_lock(&lock);
    while (num_readers > 0)
        pthread_cond_wait(writeable, &lock);
    num_writers = 1;
    pthread_mutex_unlock(&lock);

    // Use shared resource
    write(shared, val);

    // Release shared resource
    pthread_mutex_lock(&lock);
    num_writers = 0;
    pthread_cond_signal(writeable);
    pthread_cond_broadcast(readable);
    pthread_mutex_unlock(&lock);
}
```

Linux Example (General Pattern for any OS)

```
pthread_mutex_t lock;
pthread_cond_t cv;
```

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs.

```
void PerformOperationOnSharedData()
{
    pthread_mutex_lock(&lock);
    while (TestPredicate() == FALSE)
        pthread_cond_wait(& cv, &lock);

    ChangeSharedData();

    pthread_mutex_unlock(&lock);

    // If necessary, signal the condition variable
}
```

Verify that the current lock request is compatible with the existing owners.

Condition variables are subject to spurious wakeups (those not associated with an explicit wake) and stolen wakeups (another thread manages to run before the woken thread). Therefore, you should recheck a predicate (typically in a while loop) after a sleep operation returns.

Atomically release a lock and enter the sleeping state.

After a thread is woken, it re-acquires the lock it released when the thread entered the sleeping state.

The data can be changed safely because we own the critical section

You can wake other threads using `pthread_cond_signal` *or* `pthread_cond_broadcast` *either inside or outside the lock associated with the condition variable. It is usually better to release the lock before waking other threads to reduce the number of context switches.*

Windows Example (General Pattern for any OS)

Microsoft: typical usage pattern of condition variables

```
CRITICAL_SECTION CritSection;

CONDITION_VARIABLE ConditionVar;
```

Condition variables are synchronization primitives that enable threads to wait until a particular condition occurs.

```
void PerformOperationOnSharedData()

{

   EnterCriticalSection(&CritSection);

   while (TestPredicate() == FALSE)

      SleepConditionVariableCS(&ConditionVar, &CritSection, INFINITE);

   ChangeSharedData();

   LeaveCriticalSection(&CritSection);

   // If necessary, signal the condition variable

}
```

Verify that the current lock request is compatible with the existing owners.

Condition variables are subject to spurious wakeups (those not associated with an explicit wake) and stolen wakeups (another thread manages to run before the woken thread). Therefore, you should recheck a predicate (typically in a while loop) after a sleep operation returns.

Atomically release a lock and enter the sleeping state.

After a thread is woken, it re-acquires the lock it released when the thread entered the sleeping state.

The data can be changed safely because we own the critical section

You can wake other threads using WakeConditionVariable or WakeAllConditionVariable either inside or outside the lock associated with the condition variable. It is usually better to release the lock before waking other threads to reduce the number of context switches.

Summary

- Shared resources (often) need protection
- Mutexes (locks) are an inefficient way to protect shared resources
  - They “spin” and wait instead of blocking
- Semaphores add blocking to mutexes, but they are still difficult to use
- Condition variables are more ergonomic and flexible