7

Synchronization

When multithreading gets interesting

Synchronization coordinates various tasks so that they all finishin the the correct state. In C, we have series of mechanisms to control what threads are allowed to perform at a given state. Most of the time, the threads can progress without having to communicate, but every so often two or more threads may want to access a critical section. A critical section is a section of code that can only be executed by one thread at a time if the program is to function correctly. If two threads (or processes) were to execute code inside the critical section at the same time, it is possible that the program may no longer have the correct behavior.

As we said in the previous chapter, race conditions happen when an operation touches a piece of memory at the same time as another thread. If the memory location is only accessible by one thread, for example the automatic variable `i` below, then there is no possibility of a race condition and no Critical Section associated with `i`. However, the `sum` variable is a global variable and accessed by two threads. It is possible that two threads may attempt to increment the variable at the same time.

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

int sum = 0; //shared

void *countgold(void *param) {
  int i; //local to each thread
  for (i = 0; i < 10000000; i++) {
    sum += 1;
  }
  return NULL;
}

int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, countgold, NULL);
  pthread_create(&tid2, NULL, countgold, NULL);
  //Wait for both threads to finish:
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  printf("ARRRRG sum is %d\n", sum);
  return 0;
}
```

A typical output of the above code is `ARGGGH sum is <some number less than expected>` because there is a race condition. The code allows two threads to read and write `sum` at the same time. For example, both threads copy the current value of sum into CPU that runs each thread (let's pick 123). Both threads increment one to their own copy. Both threads write back the value (124). If the threads had accessed the sum at different times then the count would have been 125. A few of the possible different orderings are below.

Permissible Pattern

Table 7.1: Good Thread Access Pattern

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	...
Store (i=1 globally)	...
...	Load Addr, Add 1 (i=2 locally)
...	Store (i=2 globally)

Partial Overlap

Table 7.2: Bad Thread Access Pattern

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	...
Store (i=1 globally)	Load Addr, Add 1 (i=1 locally)
...	Store (i=1 globally)

Full Overlap

Table 7.3: Horrible Thread Access Pattern

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	Load Addr, Add 1 (i=1 locally)
Store (i=1 globally)	Store (i=1 globally)

We would like the first pattern of the code being mutually exclusive. Which leads us to our first synchronization primitive, a Mutex.

7.1 Mutex

To ensure that only one thread at a time can access a global variable, use a mutex – short for Mutual Exclusion. If one thread is currently inside a critical section we would like another thread to wait until the first thread is complete. A mutex isn't a primitive in the truest sense, though it is one of the smallest that has useful threading API. A mutex also isn't a data structure. It is an abstract data type.

Let's think about a duck satisfying the mutex api. If someone has the duck then they are allowed to access a shared resource! We call it the mutex duck. Everyone else has to waddle around and wait. Once someone let's go of the duck, they have to stop interacting with the resource and the next grabber can interact with the shared resource. Now you know the origins of the duck.

There are many ways to implement a mutex, and we'll give a few in this chapter. For right now let's use the black box that the pthread library gives us. Here is how we declare a mutex.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; // global variable
pthread_mutex_lock(&m); // start of Critical Section
// Critical section
pthread_mutex_unlock(&m); //end of Critical Section
```

7.1.1 Mutex Lifetime

There are a few ways of initializing a mutex. A program can use the macro `PTHREAD_MUTEX_INITIALIZER` only for global ('static') variables. `m = PTHREAD_MUTEX_INITIALIZER` is functionally equivalent to the more general purpose `pthread_mutex_init(m,NULL)`. The init version includes options to trade performance for additional error-checking and advanced sharing options. The init version also makes sure that the mutex is correctly initialized after the call, global mutexes are initialized on the first lock. A program can also call the init function inside of a program for a mutex located on the heap.

```
pthread_mutex_t *lock = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
//later
pthread_mutex_destroy(lock);
free(lock);
```

Once we are finished with the mutex we should also call `pthread_mutex_destroy(m)` too. Note, a program can only destroy an unlocked mutex, destroy on a locked mutex is undefined behavior. Things to keep in mind about `init` and `destroy` A program doesn't need to destroy a mutex created with the global initializer.

1. Multiple threads init/destroy has undefined behavior

2. Destroying a locked mutex has undefined behavior

3. Keep to the pattern of one and only one thread initializing a mutex.

4. Copying the bytes of the mutex to a new memory location and then using the copy is *not* supported. To reference a mutex, a program *must* to have a pointer to that memory address.

7.1.2 Mutex Usages

How does one use a mutex? Here is a complete example in the spirit of the earlier piece of code.

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

// Create a mutex this ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int sum = 0;

void *countgold(void *param) {
  int i;

  //Same thread that locks the mutex must unlock it
  //Critical section is ’sum += 1’
  //However locking and unlocking ten million times
  //has significant overhead

  pthread_mutex_lock(&m);

  // Other threads that call lock will have to wait until we call unlock

  for (i = 0; i < 10000000; i++) {
    sum += 1;
  }
  pthread_mutex_unlock(&m);
  return NULL;
}

int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, countgold, NULL);
  pthread_create(&tid2, NULL, countgold, NULL);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  printf("ARRRRG sum is %d\n", sum);
  return 0;
}
```

In the code above, the thread gets the lock to the counting house before entering. The critical section is only the sum+=1 so the following version is also correct.

```
for (i = 0; i < 10000000; i++) {
  pthread_mutex_lock(&m);
  sum += 1;
  pthread_mutex_unlock(&m);
}
return NULL;
}
```

This process runs slower because we lock and unlock the mutex a million times, which is expensive - at least compared with incrementing a variable. In this simple example, we didn't need threads - we could have added up twice! A faster multi-thread example would be to add one million using an automatic (local) variable and only then adding it to a shared total after the calculation loop has finished:

```
int local = 0;
for (i = 0; i < 10000000; i++) {
  local += 1;
}

pthread_mutex_lock(&m);
sum += local;
pthread_mutex_unlock(&m);

return NULL;
}
```

If you know the Gaussian sum, you can avoid race conditions altogether, but this is for illustration.

Starting with the gotchas. Firstly, C Mutexes do not lock variables. A mutex is a simple data structure. It works with code, not data. If a mutex is locked, the other threads will continue. It's only when a thread attempts to lock a mutex that is already locked, will the thread have to wait. As soon as the original thread unlocks the mutex, the second (waiting) thread will acquire the lock and be able to continue. The following code creates a mutex that does effectively nothing.

```
int a;
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER,
m2 = = PTHREAD_MUTEX_INITIALIZER;
// later
// Thread 1
pthread_mutex_lock(&m1);
a++;
pthread_mutex_unlock(&m1);
```

```
// Thread 2
pthread_mutex_lock(&m2);
a++;
pthread_mutex_unlock(&m2);
```

Here are some other gotchas in no particular order

1. Don't cross the streams! If using threads, don't fork in the middle of your program. This means any time after your mutexes have been initialized.
2. The thread that locks a mutex is the only thread that can unlock it.
3. Each program can have multiple mutex locks. A thread safe design might include a lock with each data structure, one lock per heap, or one lock per set of data structures If a program has only one lock, then there may be significant contention for the lock. If two threads were updating two different counters, it isn't necessary to use the same lock.
4. Locks are only tools. They don't spot critical sections!
5. There will always be a small amount of overhead of calling `pthread_mutex_lock` and `pthread_mutex_unlock`. However, this is the price to pay for correctly functioning programs!
6. Not unlocking a mutex due to an early return during an error condition
7. Resource leak (not calling `pthread_mutex_destroy`)
8. Using an uninitialized mutex or using a mutex that has already been destroyed
9. Locking a mutex twice on a thread without unlocking first
10. Deadlock

7.1.3 Mutex Implementation

So we have this cool data structure. How do we implement it? A naive, incorrect implementation is shown below. The `unlock` function simply unlocks the mutex and returns. The lock function first checks to see if the lock is already locked. If it is currently locked, it will keep checking again until another thread has unlocked the mutex. For the time being, we'll avoid the condition that other threads are able to unlock a lock they don't own and focus on the mutual exclusion aspect.

```
// Version 1 (Incorrect!)

void lock(mutex_t *m) {
  while(m->locked) {/*Locked? Never-mind - loop and check again!*/ }

  m->locked = 1;
}
```

```
void unlock(mutex_t *m) {
  m->locked = 0;
}
```

Version 1 uses ‘busy-waiting’ unnecessarily wasting CPU resources. However, there is a more serious problem. We have a race-condition! If two threads both called `lock` concurrently, it is possible that both threads would read `m_locked` as zero. Thus both threads would believe they have exclusive access to the lock and both threads will continue.

We might attempt to reduce the CPU overhead a little by calling `pthread_yield()` inside the loop - pthread_yield suggests to the operating system that the thread does not use the CPU for a short while, so the CPU may be assigned to threads that are waiting to run. This still leaves the race-condition. We need a better implementation. We will talk about this later in the critical section part of this chapter. For now, we will talk about semaphores.

7.1.4 Advanced: Implementing a Mutex with hardware

We can use C11 Atomics to do that perfectly! A complete solution is detailed here. This is a spinlock mutex, futex implementations can be found online.

First the data structure and initialization code.

```
typedef struct mutex_{
  // We need some variable to see if the lock is locked
  atomic_int_least8_t lock;
  // A mutex needs to keep track of its owner so
  // Another thread can't unlock it
  pthread_t owner;
} mutex;

#define UNLOCKED 0
#define LOCKED 1
#define UNASSIGNED_OWNER 0

int mutex_init(mutex* mtx){
  // Some simple error checking
  if(!mtx){
    return 0;
  }
  // Not thread-safe the user has to take care of this
  atomic_init(&mtx->lock, UNLOCKED);
  mtx->owner = UNASSIGNED_OWNER;
  return 1;
}
```

This is the initialization code, nothing fancy here. We set the state of the mutex to unlocked and set the owner to locked.

```c
int mutex_lock(mutex* mtx){
  int_least8_t zero = UNLOCKED;
  while(!atomic_compare_exchange_weak_explicit
  (&mtx->lock,
  &zero,
  LOCKED,
  memory_order_seq_cst,
  memory_order_seq_cst)){
    zero = UNLOCKED;
    sched_yield(); // Use system calls for scheduling speed
  }
  // We have the lock now
  mtx->owner = pthread_self();
  return 1;
}
```

What does this code do? It initializes a variable that we will keep as the unlocked state. Atomic Compare and Exchange is an instruction supported by most modern architectures (on x86 it's `lock cmpxchg`). The pseudocode for this operation looks like this

```c
int atomic_compare_exchange_pseudo(int* addr1, int* addr2, int val){
  if(*addr1 == *addr2){
    *addr1 = val;
    return 1;
  }else{
    *addr2 = *addr1;
    return 0;
  }
}
```

Except it is all done *atomically* meaning in one uninterruptible operation. What does the *weak* part mean? Atomic instructions are prone to **spurious failures** meaning that there are two versions to these atomic functions a *strong* and a *weak* part, strong guarantees the success or failure while weak may fail even when the operation succeeds. These are the same spurious failures that you'll see in condition variables below. We are using weak because weak is faster, and we are in a loop! That means we are okay if it fails a little bit more often because we will keep spinning around anyway.

Inside the while loop, we have failed to grab the lock! We reset zero to unlocked and sleep for a little while. When we wake up we try to grab the lock again. Once we successfully swap, we are in the critical section! We set the mutex's owner to the current thread for the unlock method and return successfully.

How does this guarantee mutual exclusion? When working with atomics we are unsure! But in this simple example, we can because the thread that can successfully expect the lock to be UNLOCKED (0) and swap it to a LOCKED (1) state is considered the winner. How do we implement unlock?

```
int mutex_unlock(mutex* mtx){
  if(unlikely(pthread_self() != mtx->owner)){
    return 0; // Can't unlock a mutex if the thread isn't the owner
  }
  int_least8_t one = 1;
  //Critical section ends after this atomic
  mtx->owner = UNASSIGNED_OWNER;
  if(!atomic_compare_exchange_strong_explicit(
  &mtx->lock,
  &one,
  UNLOCKED,
  memory_order_seq_cst,
  memory_order_seq_cst)){
    //The mutex was never locked in the first place
    return 0;
  }
  return 1;
}
```

To satisfy the API, a thread can't unlock the mutex unless the thread is the one who owns it. Then we unassign the mutex owner, because critical section is over after the atomic. We want a strong exchange because we don't want to block. We expect the mutex to be locked, and we swap it to unlock. If the swap was successful, we unlocked the mutex. If the swap wasn't, that means that the mutex was UNLOCKED and we tried to switch it from UNLOCKED to UNLOCKED, preserving the behavior of unlock.

What is this memory order business? We were talking about memory fences earlier, here it is! We won't go into detail because it is outside the scope of this course but in the scope of this article. We need consistency to make sure no loads or stores are ordered before or after. A program need to create dependency chains for more efficient ordering.

7.1.5 Semaphore

A semaphore is another synchronization primitive. It is initialized to some value. Threads can either `sem_wait` or `sem_post` which lowers or increases the value. If the value reaches zero and a wait is called, the thread will be blocked until a post is called.

Using a semaphore is as easy as using a mutex. First, decide if on the initial value, for example the number of remaining spaces in an array. Unlike pthread mutex there are no shortcuts to creating a semaphore - use `sem_init`.

```
#include <semaphore.h>

sem_t s;
int main() {
  sem_init(&s, 0, 10); // returns -1 (=FAILED) on OS X
  sem_wait(&s); // Could do this 10 times without blocking
  sem_post(&s); // Announce that we've finished (and one more resource item is available; increment count)
  sem_destroy(&s); // release resources of the semaphore
}
```

When using a semaphore, wait and post can be called from different threads! Unlike a mutex, the increment and decrement can be from different threads.

This becomes especially useful if you want to use a semaphore to implement a mutex. A mutex is a semaphore that always `waits` before it `posts`. Some textbooks will refer to a mutex as a binary semaphore. You do have to be careful to never add more than one to a semaphore or otherwise your mutex abstraction breaks. That is usually why a mutex is used to implement a semaphore and vice versa.

- Initialize the semaphore with a count of one.
- Replace `pthread_mutex_lock` with `sem_wait`
- Replace `pthread_mutex_unlock` with `sem_post`

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

sem_wait(&s);
// Critical Section
sem_post(&s);
```

But be warned, it isn't the same! A mutex can handle what we call lock inversion well. Meaning the following code breaks with a traditional mutex, but produces a race condition with threads.

```
// Thread 1
sem_wait(&s);
// Critical Section
sem_post(&s);

// Thread 2
// Some threads want to see the world burn
sem_post(&s);

// Thread 3
sem_wait(&s);
// Not thread-safe!
sem_post(&s);
```

If we replace it with mutex lock, it won't work now.

```
// Thread 1
mutex_lock(&s);
// Critical Section
mutex_unlock(&s);

// Thread 2
// Foiled!
mutex_unlock(&s);

// Thread 3
mutex_lock(&s);
// Now it's thread-safe
mutex_unlock(&s);
```

Also, binary semaphores are different than mutexes because one thread can unlock a mutex from a different thread.

Signal Safety

Also, `sem_post` is one of a handful of functions that can be correctly used inside a signal handler `pthread_mutex_unlock` is not. We can release a waiting thread that can now make all of the calls that we disallowed to call inside the signal handler itself e.g. `printf`. Here is some code that utilizes this;

```
#include <stdio.h>
#include <pthread.h>
#include <signal.h>
#include <semaphore.h>
#include <unistd.h>

sem_t s;

void handler(int signal) {
  sem_post(&s); /* Release the Kraken! */
}

void *singsong(void *param) {
  sem_wait(&s);
  printf("Waiting until a signal releases...\n");
}

int main() {
  int ok = sem_init(&s, 0, 0 /* Initial value of zero*/);
  if (ok == -1) {
    perror("Could not create unnamed semaphore");
    return 1;
  }
  signal(SIGINT, handler); // Too simple! See Signals chapter

  pthread_t tid;
  pthread_create(&tid, NULL, singsong, NULL);
  pthread_exit(NULL); /* Process will exit when there are no more
      threads */
}
```

Other uses for semaphores are keeping track of empty spaces in arrays. We will discuss these in the thread-safe data structures section.

7.2 Condition Variables

Condition variables allow a set of threads to sleep until woken up. The API allows either one or all threads to be woken up. If a program only wakes one thread, the operating system will decide which thread to wake up. Threads don't wake threads other directly like by id. Instead, a thread 'signal's the condition variable, which then will wake up one (or all) threads that are sleeping inside the condition variable.

Condition variables are also used with a mutex and with a loop, so when woken up they have to check a condition in a critical section. If a thread needs to be woken up outside of a critical section, there are other ways to do this in POSIX. Threads sleeping inside a condition variable are woken up by calling `pthread_cond_broadcast` (wake up all) or `pthread_cond_signal` (wake up one). Note despite the function name, this has nothing to do with POSIX `signal`s!

Occasionally, a waiting thread may appear to wake up for no reason. This is called a *spurious wakeup*. If you read the hardware implementation of a mutex section, this is similar to the atomic failure of the same name.

Why do spurious wakeups happen? For performance. On multi-CPU systems, it is possible that a race condition could cause a wake-up (signal) request to be unnoticed. The kernel may not detect this lost wake-up call but can detect when it might occur. To avoid the potentially lost signal, the thread is woken up so that the program code can test the condition again. If you want to know why, check out the appendix.

7.3 Thread-Safe Data Structures

Naturally, we want our data structures to be thread-safe as well! We can use mutexes and synchronization primitives to make that happen. First a few definitions. Atomicity is when an operation is thread-safe. We have atomic instructions in hardware by providing the lock prefix

```
lock ...
```

But Atomicity also applies to higher orders of operations. We say a data structure operation is atomic if it happens all at once and successfully or not at all.

As such, we can use synchronization primitives to make our data structures thread-safe. For the most part, we will be using mutexes because they carry more semantic meaning than a binary semaphore. Note, this is an introduction. Writing high-performance thread-safe data structures requires its own book! Take for example the following thread-unsafe stack.

```
// A simple fixed-sized stack (version 1)
#define STACK_SIZE 20
int count;
double values[STACK_SIZE];

void push(double v) {
  values[count++] = v;
}

double pop() {
  return values[--count];
}

int is_empty() {
  return count == 0;
}
```

Version 1 of the stack is thread-unsafe because if two threads call push or pop at the same time then the results or the stack can be inconsistent. For example, imagine if two threads call pop at the same time then both threads may read the same value, both may read the original count value.

To turn this into a thread-safe data structure we need to identify the *critical sections* of our code, meaning we need to ask which section(s) of the code must only have one thread at a time. In the above example the `push`, `pop`, and `is_empty` functions access the same memory and all critical sections for the stack. While `push` (and `pop`) is executing, the data structure is an inconsistent state, for example the count may not have been written to, so it may still contain the original value. By wrapping these methods with a mutex we can ensure that only one thread at a time can update (or read) the stack. A candidate 'solution' is shown below. Is it correct? If not, how will it fail?

```
// An attempt at a thread-safe stack (version 2)
#define STACK_SIZE 20
int count;
double values[STACK_SIZE];

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void push(double v) {
  pthread_mutex_lock(&m1);
  values[count++] = v;
  pthread_mutex_unlock(&m1);
}

double pop() {
  pthread_mutex_lock(&m2);
  double v = values[--count];
  pthread_mutex_unlock(&m2);

  return v;
}

int is_empty() {
  pthread_mutex_lock(&m1);
  return count == 0;
  pthread_mutex_unlock(&m1);
}
```

Version 2 contains at least one error. Take a moment to see if you can the error(s) and work out the consequence(s).

If three threads called `push()` at the same time, the lock `m1` ensures that only one thread at time manipulates the stack on push or is_empty – Two threads will need to wait until the first thread completes A similar argument applies to concurrent calls to `pop`. However, Version 2 does not prevent push and pop from running at the same time because `push` and `pop` use two different mutex locks. The fix is simple in this case - use the same mutex lock for both the push and pop functions.

The code has a second error. `is_empty` returns after the comparison and leaves the mutex unlocked. However, the error would not be spotted immediately. For example, suppose one thread calls `is_empty` and a second thread later calls `push`. This thread would mysteriously stop. Using debugger, you can discover that the thread is stuck at the lock() method inside the `push` method because the lock was never unlocked by the earlier `is_empty` call. Thus an oversight in one thread led to problems much later in time in an arbitrary other thread. Let's try to rectify these problems

```
// An attempt at a thread-safe stack (version 3)
int count;
double values[count];
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void push(double v) {
  pthread_mutex_lock(&m);
  values[count++] = v;
  pthread_mutex_unlock(&m);
}
double pop() {
  pthread_mutex_lock(&m);
  double v = values[--count];
  pthread_mutex_unlock(&m);
  return v;
}
int is_empty() {
  pthread_mutex_lock(&m);
  int result = count == 0;
  pthread_mutex_unlock(&m);
  return result;
}
```

Version 3 is thread-safe. We have ensured mutual exclusion for all of the critical sections. There are a few things to note.

- `is_empty` is thread-safe but its result may already be out-of-date. The stack may no longer be empty by the time the thread gets the result! This is usually why in thread-safe data structures, functions that return sizes are removed or deprecated.
- There is no protection against underflow (popping on an empty stack) or overflow (pushing onto an already-full stack)

The last point can be fixed using counting semaphores. The implementation assumes a single stack. A more general-purpose version might include the mutex as part of the memory structure and use `pthread_mutex_init` to initialize the mutex. For example,

```
// Support for multiple stacks (each one has a mutex)
typedef struct stack {
  int count;
  pthread_mutex_t m;
  double *values;
} stack_t;

stack_t* stack_create(int capacity) {
  stack_t *result = malloc(sizeof(stack_t));
  result->count = 0;
  result->values = malloc(sizeof(double) * capacity);
  pthread_mutex_init(&result->m, NULL);
  return result;
}
void stack_destroy(stack_t *s) {
  free(s->values);
  pthread_mutex_destroy(&s->m);
  free(s);
}

// Warning no underflow or overflow checks!

void push(stack_t *s, double v) {
  pthread_mutex_lock(&s->m);
  s->values[(s->count)++] = v;
  pthread_mutex_unlock(&s->m);
}

double pop(stack_t *s) {
  pthread_mutex_lock(&s->m);
  double v = s->values[--(s->count)];
  pthread_mutex_unlock(&s->m);
  return v;
}

int is_empty(stack_t *s) {
  pthread_mutex_lock(&s->m);
  int result = s->count == 0;
  pthread_mutex_unlock(&s->m);
  return result;
}

int main() {
  stack_t *s1 = stack_create(10 /* Max capacity*/);
  stack_t *s2 = stack_create(10);
  push(s1, 3.141);
  push(s2, pop(s1));
  stack_destroy(s2);
  stack_destroy(s1);
}
```

Before we fix the problems with semaphores. How would we fix the problems with condition variables? Try it out before you look at the code in the previous section. We need to wait in push and pop if our stack is full or empty respectively. Attempted solution:

```
// Assume cv is a condition variable
// correctly initialized

void push(stack_t *s, double v) {
  pthread_mutex_lock(&s->m);
  if(s->count == 0) pthread_cond_wait(&s->cv, &s->m);
  s->values[(s->count)++] = v;
  pthread_mutex_unlock(&s->m);
}

double pop(stack_t *s) {
  pthread_mutex_lock(&s->m);
  if(s->count == 0) pthread_cond_wait(&s->cv, &s->m);
  double v = s->values[--(s->count)];
  pthread_mutex_unlock(&s->m);
  return v;
}
```

Does the following solution work? Take a second before looking at the answer to spot the errors. So did you catch all of them?

1. The first one is a simple one. In push, our check should be against the total capacity, not zero.

2. We only have if statement checks. wait() could spuriously wake up

3. We never signal any of the threads! Threads could get stuck waiting indefinitely.

Let's fix those errors Does this solution work?

```
void push(stack_t *s, double v) {
  pthread_mutex_lock(&s->m);
  while(s->count == capacity) pthread_cond_wait(&s->cv, &s->m);
  s->values[(s->count)++] = v;
  pthread_mutex_unlock(&s->m);
  pthread_cond_signal(&s->cv);
}

double pop(stack_t *s) {
  pthread_mutex_lock(&s->m);
  while(s->count == 0) pthread_cond_wait(&s->cv, &s->m);
  double v = s->values[--(s->count)];
  pthread_cond_broadcast(&s->cv);
  pthread_mutex_unlock(&s->m);
  return v;
}
```

This solution doesn't work either! The problem is with the signal. Can you see why? What would you do to fix it?

Now, how would we use counting semaphores to prevent over and underflow? Let's discuss it in the next section.

7.3.1 Using Semaphores

Let's use a counting semaphore to keep track of how many spaces remain and another semaphore to track the number of items in the stack. We will call these two semaphores sremain and sitems. Remember sem_wait will wait if the semaphore's count has been decremented to zero (by another thread calling sem_post).

```
// Sketch #1

sem_t sitems;
sem_t sremain;
void stack_init(){
  sem_init(&sitems, 0, 0);
  sem_init(&sremain, 0, 10);
}


double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  ...

  void push(double v) {
    // Wait until there's at least one space
    sem_wait(&sremain);
    ...
  }
```

Sketch #2 has implemented the `post` too early. Another thread waiting in push can erroneously attempt to write into a full stack. Similarly, a thread waiting in the pop() is allowed to continue too early.

```
// Sketch #2 (Error!)
double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  sem_post(&sremain); // error! wakes up pushing() thread too early
  return values[--count];
}
void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  sem_post(&sitems); // error! wakes up a popping() thread too early
  values[count++] = v;
}
```

Sketch 3 implements the correct semaphore logic, but can you spot the error?

```
// Sketch #3 (Error!)
double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  double v= values[--count];
  sem_post(&sremain);
  return v;
}

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  values[count++] = v;
  sem_post(&sitems);
}
```

Sketch 3 correctly enforces buffer full and buffer empty conditions using semaphores. However, there is no *mutual exclusion*. Two threads can be in the *critical section* at the same time, which would corrupt the data structure or least lead to data loss. The fix is to wrap a mutex around the critical section:

```
// Simple single stack - see the above example on how to convert this
    into multiple stacks.
// Also a robust POSIX implementation would check for EINTR and error
    codes of sem_wait.

// PTHREAD_MUTEX_INITIALIZER for statics (use pthread_mutex_init() for
    stack/heap memory)
#define SPACES 10
pthread_mutex_t m= PTHREAD_MUTEX_INITIALIZER;
int count = 0;
double values[SPACES];
sem_t sitems, sremain;

void init() {
  sem_init(&sitems, 0, 0);
  sem_init(&sremains, 0, SPACES); // 10 spaces
}

double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);

  pthread_mutex_lock(&m); // CRITICAL SECTION
  double v= values[--count];
  pthread_mutex_unlock(&m);

  sem_post(&sremain); // Hey world, there's at least one space
  return v;
}

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  pthread_mutex_lock(&m); // CRITICAL SECTION
  values[count++] = v;
  pthread_mutex_unlock(&m);

  sem_post(&sitems); // Hey world, there's at least one item
}
// Note a robust solution will need to check sem_wait's result for
    EINTR (more about this later)
```

What happens when we start inverting the lock and wait orders?

```
double pop() {
  pthread_mutex_lock(&m);
  sem_wait(&sitems);

  double v= values[--count];
  pthread_mutex_unlock(&m);

  sem_post(&sremain);
  return v;
}

void push(double v) {
  sem_wait(&sremain);

  pthread_mutex_lock(&m);
  values[count++] = v;
  pthread_mutex_unlock(&m);

  sem_post(&sitems);
}
```

Rather than giving you the answer, we'll let you think about this. Is this a permissible way to lock and unlock? Is there a series of operations that could cause a race condition? How about deadlock? If there is, provide it. If there isn't, provide a short justification proof of why that won't happen.

7.4 Software Solutions to the Critical Section

As already discussed, there are critical parts of our code that can only be executed by one thread at a time. We describe this requirement as 'mutual exclusion'. Only one thread (or process) may have access to the shared resource. In multi-threaded programs, we can wrap a critical section with mutex lock and unlock calls:

```
pthread_mutex_lock() // one thread allowed at a time! (others will have
    to wait here)
// ... Do Critical Section stuff here!
pthread_mutex_unlock() // let other waiting threads continue
```

How would we implement these lock and unlock calls? Can we create a pure software algorithm that assures mutual exclusion? Here is our attempt from earlier.

```
pthread_mutex_lock(p_mutex_t *m) {
  while(m->lock) ;
  m->lock = 1;
}
pthread_mutex_unlock(p_mutex_t *m) {
  m->lock = 0;
}
```

As we touched on earlier, this implementation *does not satisfy Mutual Exclusion* even considering that threads can unlock other threads locks. Let's take a close look at this 'implementation' from two threads running around the same time.

To simplify the discussion, we consider only two threads. Note these arguments work for threads and processes and the classic CS literature discusses these problems in terms of two processes that need exclusive access to a critical section or shared resource. Raising a flag represents a thread/process's intention to enter the critical section.

There are three main desirable properties that we desire in a solution to the critical section problem.

1. Mutual Exclusion. The thread/process gets exclusive access. Others must wait until it exits the critical section.

2. Bounded Wait. A thread/process cannot get superseded by another thread infinite amounts of time.

3. Progress. If no thread/process is inside the critical section, the thread/process should be able to proceed without having to wait.

With these ideas in mind, let's examine another candidate solution that uses a turn-based flag only if two threads both required access at the same time.

7.4.1 Naive Solutions

Remember that the pseudo-code outlined below is part of a larger program. The thread or process will typically need to enter the critical section many times during the lifetime of the process. So, imagine each example as wrapped inside a loop where for a random amount of time the thread or process is working on something else.

Is there anything wrong with the candidate solution described below?

```
// Candidate #1
```

```
wait until your flag is lowered
raise my flag
// Do Critical Section stuff
lower my flag
```

Answer: Candidate solution #1 also suffers from a race condition because both threads/processes could read each other's flag value as lowered and continue.

This suggests we should raise the flag *before* checking the other thread's flag, which is candidate solution #2 below.

```
// Candidate #2
raise my flag
wait until your flag is lowered
// Do Critical Section stuff
lower my flag
```

Candidate #2 satisfies mutual exclusion. It is impossible for two threads to be inside the critical section at the same time. However, this code suffers from deadlock! Suppose two threads wish to enter the critical section at the same time.

Table 7.4: Candidate Solution #2 Analysis

Time	Thread 1	Thread 2
1	Raise Flag	
2		Raise Flag
3	Wait	Wait

Both processes are now waiting for the other one to lower their flags. Neither one will enter the critical section as both are now stuck forever! This suggests we should use a turn-based variable to try to resolve who should proceed.

7.4.2 Turn-based solutions

The following candidate solution #3 uses a turn-based variable to politely allow one thread and then the other to continue

```
// Candidate #3
wait until my turn is myid
// Do Critical Section stuff
turn = yourid
```

Candidate #3 satisfies mutual exclusion. Each thread or process gets exclusive access to the Critical Section. However, both threads/processes must take a strict turn-based approach to use the critical section. They are forced into an alternating critical section access pattern. If thread 1 wishes to read a hash table every millisecond, but another thread writes to a hash table every second, then the reading thread would have to wait another 999ms before being able to read from the hash table again. This 'solution' is ineffective because our threads should be able to make progress and enter the critical section if no other thread is currently in the critical section.

7.4.3 Turn and Flag solutions

Is the following a correct solution to CSP?

```
\\ Candidate #4
raise my flag
if your flag is raised, wait until my turn
// Do Critical Section stuff
turn = yourid
lower my flag
```

Analyzing these solutions is tricky. Even peer-reviewed papers on this specific subject contain incorrect solutions [?]! At first glance, it appears to satisfy Mutual Exclusion, Bounded Wait and Progress The turn-based flag is only used in the event of a tie, so Progress and Bounded Wait is allowed and mutual exclusion appears to be satisfied. Perhaps you can find a counter-example?

Candidate #4 fails because a thread does not wait until the other thread lowers its flag. After some thought or inspiration, the following scenario can be created to demonstrate how Mutual Exclusion is not satisfied.

Imagine the first thread runs this code twice. The turn flag now points to the second thread. While the first thread is still inside the Critical Section, the second thread arrives. The second thread can immediately continue into the Critical Section!

Table 7.5: Candidate Solution #4

Time	Turn	Thread # 1	Thread # 2
1	2	Raise my flag	
2	2	If your flag is raised, wait until my turn	Raise my flag
3	2	// Do Critical Section Stuff	If your flag is raised, wait until my turn (TRUE!)
4	2	// Do Critical Section Stuff	Do Critical Section Stuff - OOPS

7.5 Working Solutions

The first solution to the problem was Dekker's Solution. Dekker's Algorithm (1962) was the first provably correct solution. Though, it was in an unpublished paper, so it was not discovered until later [1] (this is an English transcribed version released in 1965). A version of the algorithm is below.

```
raise my flag
while (your flag is raised) :
   if it is your turn to win :
     lower my flag
     wait while your turn
     raise my flag
// Do Critical Section stuff
set your turn to win
lower my flag
```

Notice how the process's flag is always raised during the critical section no matter if the loop is iterated zero, once or more times. Further, the flag can be interpreted as an immediate intent to enter the critical section. Only if the other process has also raised the flag will one process defer, lower their intent flag and wait. Let's check the conditions.

1. Mutual Exclusion. Let's try to sketch a simple proof. The loop invariant is that at the start of checking the condition, your flag has to be raised – this is by exhaustion. Since the only way that a thread can leave the loop is by having the condition be false, the flag must be raised for the entirety of the critical section. Since the loop prevents a thread from exiting while the other thread's flag is raised and a thread has its flag raised in the critical section, the other thread can't enter the critical section at the same time.

2. Bounded Wait. Assuming that the critical section ends in finite time, a thread once it has left the critical section cannot then get the critical section back. The reason being is the turn variable is set to the other thread, meaning that that thread now has priority. That means a thread cannot be superseded infinitely by another thread.

3. Progress. If the other thread isn't in the critical section, it will simply continue with a simple check. We didn't make any statement about if threads are randomly stopped by the system scheduler. This is an idealized scenario where threads will keep executing instructions.

7.5.1 Peterson's Solution

Peterson published his novel and surprisingly simple solution in 1981 [2]. A version of his algorithm is shown below that uses a shared variable `turn`.

```
// Candidate #5
raise my flag
turn = other_thread_id
while (your flag is up and turn is other_thread_id)
    loop
// Do Critical Section stuff
lower my flag
```

This solution satisfies Mutual Exclusion, Bounded Wait and Progress. If thread #2 has set turn to 2 and is currently inside the critical section. Thread #1 arrives, *sets the turn back to 1* and now waits until thread 2 lowers the flag.

1. Mutual Exclusion. Let's try to sketch a simple proof again. A thread doesn't get into the critical section until the turn variable is yours or the other thread's flag isn't up. If the other thread's flag isn't up, it isn't trying to enter the critical section. That is the first action the thread does and the last action the thread undoes. If the turn variable is set to this thread, that means that the other thread has given the control to this thread. Since my flag is raised and the turn variable is set, the other thread has to wait in the loop until the current thread is done.

2. Bounded Wait. After one thread lowers, a thread waiting in the while loop will leave because the first condition is broken. This means that threads cannot win all the time.

3. Progress. If no other thread is contesting, other thread's flags are not up. That means that a thread can go past the while loop and do critical section items.

Unfortunately, we can't implement a software mutex in the same way today because of out of order instructions. Check the appendix for a solution to the problem.

7.6 Implementing Counting Semaphore

Now that we have a solution to the critical section problem, We can reasonably implement a mutex. How would we implement other synchronization primitives? Let's start with a semaphore. To implement a semaphore with efficient CPU usage, we will say that we have implemented a condition variable. Implementing an O(1) space condition variable using only a mutex is not trivial, or at least an O(1) heap condition variable is not trivial. We don't want to call malloc while implementing a primitive, or we may deadlock!

- We can implement a counting semaphore using condition variables.
- Each semaphore needs a count, a condition variable and a mutex

```
typedef struct sem_t {
  ssize_t count;
  pthread_mutex_t m;
  pthread_condition_t cv;
} sem_t;
```

Implement `sem_init` to initialize the mutex and condition variable

```
int sem_init(sem_t *s, int pshared, int value) {
  if (pshared) {
    errno = ENOSYS /* 'Not implemented'*/;
    return -1;
  }
  s->count = value;
  pthread_mutex_init(&s->m, NULL);
  pthread_cond_init(&s->cv, NULL);
  return 0;
}
```

Our implementation of sem_post needs to increment the count. We will also wake up any threads sleeping inside the condition variable. Notice we lock and unlock the mutex so only one thread can be inside the critical section at a time.

```
void sem_post(sem_t *s) {
  pthread_mutex_lock(&s->m);
  s->count++;
  pthread_cond_signal(&s->cv);
  /* A woken thread must acquire the lock, so it will also have to wait
      until we call unlock*/

  pthread_mutex_unlock(&s->m);
}
```

Our implementation of sem_wait may need to sleep if the semaphore's count is zero. Just like sem_post, we wrap the critical section using the lock, so only one thread can be executing our code at a time. Notice if the thread does need to wait then the mutex will be unlocked, allowing another thread to enter sem_post and awaken us from our sleep!

Also notice that even if a thread is woken up before it returns from pthread_cond_wait, it must re-acquire the lock, so it will have to wait until sem_post finishes.

```
void sem_wait(sem_t *s) {
  pthread_mutex_lock(&s->m);
  while (s->count == 0) {
    pthread_cond_wait(&s->cv, &s->m); /*unlock mutex, wait, relock
        mutex*/
  }
  s->count--;
  pthread_mutex_unlock(&s->m);
}
```

That is a complete implementation of a counting semaphore Notice that we are calling sem_post every single time. In practice, this means sem_post would unnecessary call pthread_cond_signal even if there are no waiting threads. A more efficient implementation would only call pthread_cond_signal when necessary i.e.

```
/* Did we increment from zero to one- time to signal a thread sleeping
    inside sem_post */
if (s->count == 1) /* Wake up one waiting thread!*/
pthread_cond_signal(&s->cv);
```

7.6.1 Other semaphore considerations

- A production semaphore implementation may include a queue to ensure fairness and priority. Meaning, we wake up the highest-priority and/or longest sleeping thread.
- An advanced use of `sem_init` allows semaphores to be shared across processes. Our implementation only works for threads inside the same process. We could fix this by setting the condition variable and mutex attributes.

Implementing a condition variable with a mutex is complex, so we've left that in the appendix.

7.7 Barriers

Suppose we wanted to perform a multi-threaded calculation that has two stages, but we don't want to advance to the second stage until the first stage is completed. We could use a synchronization method called a **barrier**. When a thread reaches a barrier, it will wait at the barrier until all the threads reach the barrier, and then they'll all proceed together.

Think of it like being out for a hike with some friends. You make a mental note of how many friends you have and agree to wait for each other at the top of each hill. Say you're the first one to reach the top of the first hill. You'll wait there at the top for your friends. One by one, they'll arrive at the top, but nobody will continue until the last person in your group arrives. Once they do, you'll all proceed.

Pthreads has a function `pthread_barrier_wait()` that implements this. You'll need to declare a `pthread_barrier_t` variable and initialize it with `pthread_barrier_init()`. `pthread_barrier_init()` takes the number of threads that will be participating in the barrier as an argument. Here is a sample program using barriers.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>

#define THREAD_COUNT 4

pthread_barrier_t mybarrier;

void* threadFn(void *id_ptr) {
  int thread_id = *(int*)id_ptr;
  int wait_sec = 1 + rand() % 5;
  printf("thread %d: Wait for %d seconds.\n", thread_id, wait_sec);
  sleep(wait_sec);
  printf("thread %d: I'm ready...\n", thread_id);

  pthread_barrier_wait(&mybarrier);

  printf("thread %d: going!\n", thread_id);
  return NULL;
}


int main() {
  int i;
  pthread_t ids[THREAD_COUNT];
  int short_ids[THREAD_COUNT];

  srand(time(NULL));
  pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT + 1);

  for (i=0; i < THREAD_COUNT; i++) {
    short_ids[i] = i;
    pthread_create(&ids[i], NULL, threadFn, &short_ids[i]);
  }

  printf("main() is ready.\n");

  pthread_barrier_wait(&mybarrier);

  printf("main() is going!\n");

  for (i=0; i < THREAD_COUNT; i++) {
    pthread_join(ids[i], NULL);
  }

  pthread_barrier_destroy(&mybarrier);

  return 0;
}
```

Now let's implement our own barrier and use it to keep all the threads in sync in a large calculation. Here is our thought process,

1. Threads do first calculation (use and change values in data)
2. Barrier! Wait for all threads to finish first calculation before continuing
3. Threads do second calculation (use and change values in data)

The thread function has four main parts-

```
// double data[256][8192]

void *calc(void *arg) {
  /* Do my part of the first calculation */
  /* Is this the last thread to finish? If so wake up all the other
      threads! */
  /* Otherwise wait until the other threads have finished part one */
  /* Do my part of the second calculation */
}
```

Our main thread will create the 16 threads, and we will divide each calculation into 16 separate pieces. Each thread will be given a unique value (0,1,2,..15), so it can work on its own block. Since a (void*) type can hold small integers, we will pass the value of `i` by casting it to a void pointer.

```
#define N (16)
double data[256][8192] ;
int main() {
  pthread_t ids[N];
  for(int i = 0; i < N; i++) {
    pthread_create(&ids[i], NULL, calc, (void *) i);
  }
  //...
}
```

Note, we will never dereference this pointer value as an actual memory location.
We will cast it straight back to an integer.

```
void *calc(void *ptr) {
  // Thread 0 will work on rows 0..15, thread 1 on rows 16..31
  int x, y, start = N * (int) ptr;
  int end = start + N;
  for(x = start; x < end; x++) {
    for (y = 0; y < 8192; y++) {
      /* do calc #1 */
    }
  }
}
```

After calculation 1 completes, we need to wait for the slower threads unless we are the last thread! So, keep track of the number of threads that have arrived at our barrier ‘checkpoint’.

```
// Global:
int remain = N;

// After calc #1 code:
remain--; // We finished
if (remain == 0) {/*I'm last! - Time for everyone to wake up! */ }
else {
  while (remain != 0) {/* spin spin spin*/ }
}
```

However, the code has a few flaws. One is two threads might try to decrement `remain`. The other is the loop is a busy loop. We can do better! Let's use a condition variable and then we will use a broadcast/signal functions to wake up the sleeping threads.

A reminder, a condition variable is similar to a house! Threads go there to sleep (`pthread_cond_wait`). A threa can choose to wake up one thread (`pthread_cond_signal`) or all of them (`pthread_cond_broadcast`). If there are no threads currently waiting then these two calls have no effect.

A condition variable version is usually similar to a busy loop incorrect solution - as we will show next. First, let's add a mutex and condition global variables and don't forget to initialize them in `main`.

```
//global variables
pthread_mutex_t m;
pthread_cond_t cv;

int main() {
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&cv, NULL);
```

We will use the mutex to ensure that only one thread modifies `remain` at a time. The last arriving thread needs to wake up *all* sleeping threads - so we will use `pthread_cond_broadcast(cv)` not `pthread_cond_signal`

```
pthread_mutex_lock(&m);
remain--;
if (remain == 0) {
  pthread_cond_broadcast(&cv);
}
else {
  while(remain != 0) {
    pthread_cond_wait(&cv, &m);
  }
}
pthread_mutex_unlock(&m);
```

When a thread enters `pthread_cond_wait`, it releases the mutex and sleeps. After, the thread will be woken up. Once we bring a thread back from its sleep, before returning it must wait until it can lock the mutex. Notice that even if a sleeping thread wakes up early, it will check the while loop condition and re-enter wait if necessary.

The above barrier is not reusable. Meaning that if we stick it into any old calculation loop there is a good chance that the code will encounter a condition where the barrier either deadlocks or thread races ahead one iteration faster. Why is that? Because of the ambitious thread.

We will assume that one thread is much faster than all the other threads. With the barrier API, this thread should be waiting, but it may not be. To make it concrete, let's look at this code

```
void barrier_wait(barrier *b) {
  pthread_mutex_lock(&b->m);
  // If it is 0 before decrement, we should be on
  // another iteration right?
  if (b->remain == 0) b->remain = NUM_THREADS;
  b->remain--;
  if (b->remain == 0) {
    pthread_cond_broadcast(&cv);
  }
  else {
    while(b->remain != 0) {
      pthread_cond_wait(&cv, &m);
    }
  }
  pthread_mutex_unlock(&b->m);
}

for (/* ... */) {
  // Some calc
  barrier_wait(b);
}
```

What happens if a thread becomes ambitious. Well

1. Many other threads wait on the condition variable
2. The last thread broadcasts.
3. A single thread leaves the while loop.
4. This single thread performs its calculation before any other threads *even wake up*
5. Reset the number of remaining threads and goes back to sleep.

All the other threads who should've woken up never do and our implementation deadlocks. How would you go about solving this? Hint: If multiple threads call `barrier_wait` in a loop then one can guarantee that they are on the same iteration.

7.7.1 Reader Writer Problem

Imagine you had a key-value map data structure that is used by many threads. Multiple threads should be able to look up (read) values at the same time provided the data structure is not being written to. The writers are not so gregarious. To avoid data corruption, only one thread at a time may modify (`write`) the data structure and no readers may be reading at that time.

This is an example of the *Reader Writer Problem*. Namely, how can we efficiently synchronize multiple readers and writers such that multiple readers can read together, but a writer gets exclusive access?

An incorrect attempt is shown below ("lock" is a shorthand for `pthread_mutex_lock`):

7.7.2 Attempt #1

```
void read() {
  lock(&m)
  // do read stuff
  unlock(&m)
}

void write() {
  lock(&m)
  // do write stuff
  unlock(&m)
}
```

At least our first attempt does not suffer from data corruption. Readers must wait while a writer is writing and vice versa! However, readers must also wait for other readers. Let's try another implementation.

7.7.3 Attempt #2:

```
void read() {
  while(writing) {/*spin*/}
  reading = 1
  // do read stuff
  reading = 0
}

void write() {
  while(reading || writing) {/*spin*/}
  writing = 1
  // do write stuff
  writing = 0
}
```

Our second attempt suffers from a race condition. Imagine if two threads both called **read** and **write** or both called write at the same time. Both threads would be able to proceed! Secondly, we can have multiple readers and multiple writers, so let's keep track of the total number of readers or writers Which brings us to attempt #3.

7.7.4 Attempt #3

Remember that `pthread_cond_wait` performs *Three* actions. Firstly, it atomically unlocks the mutex and then sleeps (until it is woken by `pthread_cond_signal` or `pthread_cond_broadcast`). Thirdly, the awoken thread must re-acquire the mutex lock before returning. Thus only one thread can actually be running inside the critical section defined by the lock and unlock() methods.

Implementation #3 below ensures that a reader will enter the `cond_wait` if any writers are writing.

```
read() {
  lock(&m)
  while (writing)
  cond_wait(&cv, &m)
  reading++;

  /* Read here! */

  reading--
  cond_signal(&cv)
  unlock(&m)
}
```

However, only one reader a time can read because candidate #3 did not unlock the mutex. A better version unlocks before reading.

```
read() {
  lock(&m);
  while (writing)
  cond_wait(&cv, &m)
  reading++;
  unlock(&m)

  /* Read here! */

  lock(&m)
  reading--
  cond_signal(&cv)
  unlock(&m)
}
```

Does this mean that a writer and read could read and write at the same time? No! First of all, remember cond_wait requires the thread re-acquire the mutex lock before returning. Thus only one thread can be executing code inside the critical section (marked with **) at a time!

```
read() {
  lock(&m);
  ** while (writing)
  **     cond_wait(&cv, &m)
  ** reading++;
  unlock(&m)
  /* Read here! */
  lock(&m)
  ** reading--
  ** cond_signal(&cv)
  unlock(&m)
}
```

Writers must wait for everyone. Mutual exclusion is assured by the lock.

```
write() {
  lock(&m);
  ** while (reading || writing)
  **     cond_wait(&cv, &m);
  ** writing++;
  **
  ** /* Write here! */
  ** writing--;
  ** cond_signal(&cv);
  unlock(&m);
}
```

Candidate #3 above also uses `pthread_cond_signal`. This will only wake up one thread. If many readers are waiting for the writer to complete, only one sleeping reader will be awoken from their slumber. The reader and writer should use `cond_broadcast` so that all threads should wake up and check their while-loop condition.

7.7.5 Starving writers

Candidate #3 above suffers from starvation. If readers are constantly arriving then a writer will never be able to proceed (the 'reading' count never reduces to zero). This is known as *starvation* and would be discovered under heavy loads. Our fix is to implement a bounded-wait for the writer. If a writer arrives they will still need to wait for existing readers however future readers must be placed in a “holding pen” and wait for the writer to finish. The “holding pen” can be implemented using a variable and a condition variable so that we can wake up the threads once the writer has finished.

The plan is that when a writer arrives, and before waiting for current readers to finish, register our intent to write by incrementing a counter ‘writer’

```
write() {
  lock()
  writer++

  while (reading || writing)
  cond_wait
  unlock()
  ...
}
```

And incoming readers will not be allowed to continue while writer is nonzero. Notice ‘writer’ indicates a writer has arrived, while ‘reading’ and ‘writing’ counters indicate there is an *active* reader or writer.

```
read() {
  lock()
  // readers that arrive *after* the writer arrived will have to wait
      here!
  while(writer)
  cond_wait(&cv,&m)

  // readers that arrive while there is an active writer
  // will also wait.
  while (writing)
  cond_wait(&cv,&m)
  reading++
  unlock
  ...
}
```

7.7.6 Attempt #4

Below is our first working solution to the Reader-Writer problem. Note if you continue to read about the “Reader Writer problem” then you will discover that we solved the “Second Reader Writer problem” by giving writers preferential access to the lock. This solution is not optimal. However, it satisfies our original problem of N active readers, single active writer, and avoiding starvation of the writer if there is a constant stream of readers.

Can you identify any improvements? For example, how would you improve the code so that we only woke up readers or one writer?

```
int writers; // Number writer threads that want to enter the critical
    section (some or all of these may be blocked)
int writing; // Number of threads that are actually writing inside the
    C.S. (can only be zero or one)
int reading; // Number of threads that are actually reading inside the
    C.S.
// if writing !=0 then reading must be zero (and vice versa)

reader() {
  lock(&m)
  while (writers)
  cond_wait(&turn, &m)
  // No need to wait while(writing here) because we can only exit the
      above loop
  // when writing is zero
  reading++
  unlock(&m)

  // perform reading here

  lock(&m)
  reading--
  cond_broadcast(&turn)
  unlock(&m)
}

writer() {
  lock(&m)
  writers++
  while (reading || writing)
  cond_wait(&turn, &m)
  writing++
  unlock(&m)
  // perform writing here
  lock(&m)
  writing--
  writers--
  cond_broadcast(&turn)
  unlock(&m)
}
```

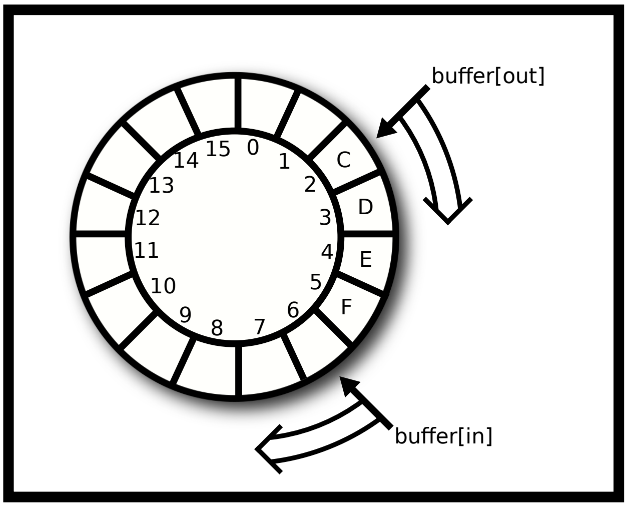

Figure 7.1: Ring Buffer Visualization

7.8 Ring Buffer

A ring buffer is a simple, usually fixed-sized, storage mechanism where contiguous memory is treated as if it is circular, and two index counters keep track of the current beginning and end of the queue. As array indexing is not circular, the index counters must wrap around to zero when moved past the end of the array. As data is added (enqueued) to the front of the queue or removed (dequeued) from the tail of the queue, the current items in the buffer form a train that appears to circle the track

A simple (single-threaded) implementation is shown below. Note, enqueue and dequeue do not guard against underflow or overflow. It's possible to add an item when the queue is full and possible to remove an item when the queue is empty. If we added 20 integers (1, 2, 3, . . . , 20) to the queue and did not dequeue any items then values, `17,18,19,20` would overwrite the `1,2,3,4`. We won't fix this problem right now, instead of when we create the multi-threaded version we will ensure enqueue-ing and dequeue-ing threads are blocked while the ring buffer is full or empty respectively.

```
void *buffer[16];
unsigned int in = 0, out = 0;

void enqueue(void *value) {/* Add one item to the front of the queue*/
  buffer[in] = value;
  in++; /* Advance the index for next time */
  if (in == 16) in = 0; /* Wrap around! */
}

void *dequeue() {/* Remove one item to the end of the queue.*/
  void *result = buffer[out];
  out++;
  if (out == 16) out = 0;
  return result;
}
```

7.8.1 Ring Buffer Gotchas

It's tempting to write the enqueue or dequeue method in the following compact form.

```
// N is the capacity of the buffer
void enqueue(void *value)
b[ (in++) % N ] = value;
}
```

This method would appear to work but contains a subtle bug. With more than four billion enqueue operations, the int value of in will overflow and wrap around to 0! Thus, you might end up writing into b[0] for example!

A compact form is correct uses bit masking provided N is a power of two. (16,32,64,…)

```
b[ (in++) & (N-1) ] = value;
```

This buffer does not yet prevent overwrites. For that, we'll turn to our multi-threaded attempt that will block a thread until there is space or there is at least one item to remove.

7.8.2 Multithreaded Correctness

The following code is an incorrect implementation. What will happen? Will enqueue and/or dequeue block? Is mutual exclusion satisfied? Can the buffer underflow? Can the buffer overflow? For clarity, pthread_mutex is shortened to p_m and we assume sem_wait cannot be interrupted.

```
#define N 16
void *b[N]
int in = 0, out = 0
p_m_t lock
sem_t s1,s2
void init() {
  p_m_init(&lock, NULL)
  sem_init(&s1, 0, 16)
  sem_init(&s2, 0, 0)
}

enqueue(void *value) {
  p_m_lock(&lock)

  // Hint: Wait while zero. Decrement and return
  sem_wait( &s1 )

  b[ (in++) & (N-1) ] = value
  // Hint: Increment. Will wake up a waiting thread
  sem_post(&s1)
  p_m_unlock(&lock)
}
void *dequeue(){
  p_m_lock(&lock)
  sem_wait(&s2)
  void *result = b[(out++) & (N-1) ]
  sem_post(&s2)
  p_m_unlock(&lock)
  return result
}
```

7.8.3 Analysis

Before reading on, see how many mistakes you can find. Then determine what would happen if threads called the enqueue and dequeue methods.

- The enqueue method waits and posts on the same semaphore (s1) and similarly with enqueue and (s2) i.e. we decrement the value and then immediately increment the value, so by the end of the function the semaphore value is unchanged!
- The initial value of s1 is 16, so the semaphore will never be reduced to zero - enqueue will not block if the ring buffer is full - so overflow is possible.
- The initial value of s2 is zero, so calls to dequeue will always block and never return!
- The order of mutex lock and sem_wait will need to be swapped; however, this example is so broken that this bug has no effect!

7.8.4 Another Analysis

The following code is an incorrect implementation. What will happen? Will `enqueue` and/or `dequeue` block? Is mutual exclusion satisfied? Can the buffer underflow? Can the buffer overflow? For clarity `pthread_mutex` is shortened to `p_m` and we assume sem_wait cannot be interrupted.

```
void *b[16]
int in = 0, out = 0
p_m_t lock
sem_t s1, s2
void init() {
  sem_init(&s1,0,16)
  sem_init(&s2,0,0)
}
```

```
enqueue(void *value){
  sem_wait(&s2)
  p_m_lock(&lock)

  b[ (in++) & (N-1) ] = value

  p_m_unlock(&lock)
  sem_post(&s1)
}

void *dequeue(){
  sem_wait(&s1)
  p_m_lock(&lock)
  void *result = b[(out++) & (N-1)]
  p_m_unlock(&lock)
  sem_post(&s2)

  return result;
}
```

Here are a few problems that we hope you've found.

- The initial value of s2 is 0. Thus enqueue will block on the first call to sem_wait even though the buffer is empty!
- The initial value of s1 is 16. Thus dequeue will not block on the first call to sem_wait even though the buffer is empty - Underflow! The dequeue method will return invalid data.
- The code does not satisfy Mutual Exclusion. Two threads can modify `in` or `out` at the same time! The code appears to use mutex lock. Unfortunately, the lock was never initialized with `pthread_mutex_init()` or `PTHREAD_MUTEX_INITIALIZER` - so the lock may not work (`pthread_mutex_lock` may simply do nothing)

7.8.5 Correct implementation of a ring buffer

As the mutex lock is stored in global (static) memory it can be initialized with `PTHREAD_MUTEX_INITIALIZER`. If we had allocated space for the mutex on the heap, then we would have used `pthread_mutex_init(ptr, NULL)`

```
#include <pthread.h>
#include <semaphore.h>
// N must be 2^i
#define N (16)

void *b[N]
int in = 0, out = 0
p_m_t lock = PTHREAD_MUTEX_INITIALIZER
```

```
sem_t countsem, spacesem

void init() {
  sem_init(&countsem, 0, 0)
  sem_init(&spacesem, 0, 16)
}
```

The enqueue method is shown below. Make sure to note.

1. The lock is only held during the critical section (access to the data structure).
2. A complete implementation would need to guard against early returns from `sem_wait` due to POSIX signals.

```
enqueue(void *value){
  // wait if there is no space left:
  sem_wait( &spacesem )

  p_m_lock(&lock)
  b[ (in++) & (N-1) ] = value
  p_m_unlock(&lock)

  // increment the count of the number of items
  sem_post(&countsem)
}
```

The `dequeue` implementation is shown below. Notice the symmetry of the synchronization calls to `enqueue`. In both cases, the functions first wait if the count of spaces or count of items is zero.

```
void *dequeue(){
  // Wait if there are no items in the buffer
  sem_wait(&countsem)

  p_m_lock(&lock)
  void *result = b[(out++) & (N-1)]
  p_m_unlock(&lock)

  // Increment the count of the number of spaces
  sem_post(&spacesem)

  return result
}
```

Food for thought:

- What would happen if the order of `pthread_mutex_unlock` and `sem_post` calls were swapped?
- What would happen if the order of `sem_wait` and `pthread_mutex_lock` calls were swapped?

7.9 Extra: Process Synchronization

You thought that you were using different processes, so you don't have to synchronize? Think again! You may not have race conditions within a process but what if your process needs to interact with the system around it? Let's consider a motivating example

```
void write_string(const char *data) {
  int fd = open("my_file.txt", O_WRONLY);
  write(fd, data, strlen(data));
  close(fd);
}

int main() {
  if(!fork()) {
    write_string("key1: value1");
    wait(NULL);
  } else {
    write_string("key2: value2");
  }
  return 0;
}
```

If none of the system calls fail then we should get something that looks like this given the file was empty to begin with.

```
key1: value1
key2: value2
```

```
key2: value2
key1: value1
```

7.9.1 Interruption

But, there is a hidden nuance. Most system calls can be `interrupted` meaning that the operating system can stop an ongoing system call because it needs to stop the process. So barring `fork` `wait` `open` and `close` from failing – they typically go to completion – what happens if `write` fails? If write fails and no bytes are written, we can get something like `key1: value1` or `key2: value2`. This is data loss which is incorrect but won't corrupt the file. What happens if write gets interrupted after a partial write? We get all sorts of madness. For example,

```
key2: key1: value1
```

7.9.2 Solution

A program can create a mutex before fork-ing - however the child and parent process will not share virtual memory and each one will have a mutex independent of the other. Advanced note: There are advanced options using shared memory that allow a child and parent to share a mutex if it's created with the correct options and uses a shared memory segment. See stackoverflow example

So what should we do? We should use a shared mutex! Consider the following code.

```
pthread_mutex_t * mutex = NULL;
pthread_mutexattr_t attr;

void write_string(const char *data) {
  pthread_mutex_lock(mutex);
  int fd = open("my_file.txt", O_WRONLY);
  int bytes_to_write = strlen(data), written = 0;
  while(written < bytes_to_write) {
    written += write(fd, data + written, bytes_to_write - written);
  }
  close(fd);
  pthread_mutex_unlock(mutex);
}

int main() {
  pthread_mutexattr_init(&attr);
  pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
  pmutex = mmap (NULL, sizeof(pthread_mutex_t),
  PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, 0);
  pthread_mutex_init(pmutex, &attrmutex);
  if(!fork()) {
    write_string("key1: value1");
    wait(NULL);
    pthread_mutex_destroy(pmutex);
    pthread_mutexattr_destroy(&attrmutex);
    munmap((void *)pmutex, sizeof(*pmutex));
  } else {
    write_string("key2: value2");
  }
  return 0;
}
```

What the code does in main is initialize a process shared mutex using a piece of `shared` memory. You will find out what this call to `mmap` does later – assume for the time being that it creates memory that is shared between processes. We can initialize a `pthread_mutex_t` in that special piece of memory and use it as normal. To counter `write` failing, we have put the `write` call inside a while loop that keeps writing so long as there are bytes left to write. Now if all the other system calls function, there should be more race conditions.

Most programs try to avoid this problem entirely by writing to separate files, but it is good to know that there are mutexes across processes, and they are useful. A program can use all of the primitives that were mentioned previouslty! Barriers, semaphores, and condition variables can all be initialized on a shared piece of memory and used in similar ways to their multithreading counterparts.

- You don't have to worry about arbitrary memory addresses becoming race condition candidates. Only areas that specifically mapped are in danger.
- You get the nice isolation of processes so if one process fails the system can maintain intact.
- When you have a lot of threads, creating a process might ease the system load

There are other ways to synchronize as well, check out goroutines or higher orders of synchronization in the appendix.

7.10 External Resources

Guiding questions for the man pages

- How is a recursive mutex different than a default mutex?
- How is mutex trylock different than mutex lock?
- Why would a mutex lock fail? What's an example?
- What happens if a thread tries to destroy a locked mutex?
- Can a thread copy the underlying bytes of a mutex instead of using a pointer?
- What is the lifecycle of a semaphore?
- pthread_mutex_lock man page
- pthread_mutex_init man page
- sem_init
- sem_wait
- sem_post
- sem_destroy

7.11 Topics

- Atomic operations
- Critical Section
- Producer Consumer Problem
- Using Condition Variables
- Using Counting Semaphore
- Implementing a barrier
- Implementing a ring buffer
- Using pthread_mutex
- Implementing producer consumer
- Analyzing multi-threaded coded

7.12 Questions

- What is atomic operation?
- Why will the following not work in parallel code

  ```
  //In the global section
  size_t a;
  //In pthread function
  for(int i = 0; i < 100000000; i++) a++;
  ```

  And this will?

  ```
  //In the global section
  atomic_size_t a;
  //In pthread function
  for(int i = 0; i < 100000000; i++) atomic_fetch_add(a, 1);
  ```

- What are some downsides to atomic operations? What would be faster: keeping a local variable or many atomic operations?
- What is the critical section?

- Once you have identified a critical section, what is one way of assuring that only one thread will be in the section at a time?
- Identify the critical section here

```
struct linked_list;
struct node;
void add_linked_list(linked_list *ll, void* elem){
  node* packaged = new_node(elem);
  if(ll->head){
    ll->head =
  }else{
    packaged->next = ll->head;
    ll->head = packaged;
    ll->size++;
  }
}

void* pop_elem(linked_list *ll, size_t index){
  if(index >= ll->size) return NULL;

  node *i, *prev;
  for(i = ll->head; i && index; i = i->next, index--){
    prev = i;
  }

  //i points to the element we need to pop, prev before
  if(prev->next) prev->next = prev->next->next;
  ll->size--;
  void* elem = i->elem;
  destroy_node(i);
  return elem;
}
```

- How tight can you make the critical section?
- What is a producer consumer problem? How might the above be a producer consumer problem be used in the above section? How is a producer consumer problem related to a reader writer problem?
- What is a condition variable? Why is there an advantage to using one over a `while` loop?
- Why is this code dangerous?

```
if(not_ready){
  pthread_cond_wait(&cv, &mtx);
}
```

- What is a counting semaphore? Give me an analogy to a cookie jar/pizza box/limited food item.
- What is a thread barrier?
- Use a counting semaphore to implement a barrier.
- Write up a Producer/Consumer queue, How about a producer consumer stack?
- Give me an implementation of a reader-writer lock with condition variables, make a struct with whatever you need, it needs to be able to support the following functions

```
typedef struct {


} rw_lock_t;

void reader_lock(rw_lock_t* lck) {

}

void writer_lock(rw_lock_t* lck) {

}

void reader_unlock(rw_lock_t* lck) {

}

void writer_unlock(rw_lock_t* lck) {

}
```

  The only specification is that in between `reader_lock` and `reader_unlock`, no writers can write. In between the writer locks, only one writer may be writing at a time.

- Write code to implement a producer consumer using ONLY three counting semaphores. Assume there can be more than one thread calling enqueue and dequeue. Determine the initial value of each semaphore.
- Write code to implement a producer consumer using condition variables and a mutex. Assume there can be more than one thread calling enqueue and dequeue.
- Use CVs to implement add(unsigned int) and subtract(unsigned int) blocking functions that never allow the global value to be greater than 100.
- Use CVs to implement a barrier for 15 threads.
- What does the following code do?

```
void main() {
    pthread_mutex_t mutex;
    pthread_cond_t cond;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_cond_broadcast(&cond);
    pthread_cond_wait(&cond,&mutex);

    return 0;
}
```

- Is the following code correct? If it isn't, could you fix it?

```
extern int money;
void deposit(int amount) {
  pthread_mutex_lock(&m);
  money += amount;
  pthread_mutex_unlock(&m);
}

void withdraw(int amount) {
  if (money < amount) {
    pthread_cond_wait(&cv);
  }

  pthread_mutex_lock(&m);
  money -= amount;
  pthread_mutex_unlock(&m);
}
```

- Sketch how to use a binary semaphore as a mutex. Remember in addition to mutual exclusion, a mutex can only ever be unlocked by the thread who called it.

```
sem_t sem;

void lock() {

}

void unlock() {

}
```

- How many of the following statements are true?
    - There can be multiple active readers
    - There can be multiple active writers
    - When there is an active writer the number of active readers must be zero
    - If there is an active reader the number of active writers must be zero
    - A writer must wait until the current active readers have finished

Bibliography

[1] T.J. Dekker and Edsgar Dijkstra. Over de sequentialiteit van procesbeschrijvingen, 1965. URL `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD35.html`.

[2] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12:115–116, 1981.