Lecture 7 — Race Conditions & Synchronization

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 22, 2019

Race Conditions

“Knock knock.”
“Race Condition.”
“Who’s there?”

Define A Race Condition

A race occurs when you have two concurrent accesses to the same memory location, at least one of which is a **write**.

This definition is a little bit strict.

We could also say that there is a race condition if there is some form of output, such as writing to the console.

If one thread is going to write “1” to the console and another is going to write “2”, then we could have a race condition.

If there is no co-ordination, we could get output of “12” or “21”.

If the order here is unimportant, there’s no issue; but if one order is correct, then the appearance of the other is a bug.

Matters of State

When there's a race, the final state may not be the same as running one access to completion and then the other.

But it "usually" is. It's nondeterministic.

The fact that the output is often "12" and only very occasionally "21" may make it very difficult to track down the source of the problem.

Hazards

In other situations (e.g., processor design) these are sometimes referred to as data hazards or dependencies.

1. **RAW** (Read After Write)
2. **WAR** (Write After Read)
3. **WAW** (Write After Write)
4. **RAR** (Read After Read) - No such hazard!

It's a Kind of Hazard

Race conditions typically arise between variables shared between threads.

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

void* run1(void* arg) {
    int* x = (int*) arg;
    *x += 1;
}

void* run2(void* arg) {
    int* x = (int*) arg;
    *x += 2;
}
```

```c
int main(int argc, char *argv[]) {
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;

}
```

It's a Kind of Hazard II

```
int main(int argc, char *argv[]) {
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Now do we have a race?

Trace the Race

What are the possible outputs? (Assume that initially $*x$ is 1.) We'll look at compiler intermediate code (three-address code) to tell.

```
run1                          run2
D.1 = *x;                     D.1 = *x;
D.2 = D.1 + 1;                D.2 = D.1 + 2
*x = D.2;                     *x = D.2;
```

Memory reads and writes are key in data races.

Trace the Race

Let's call the read and write from `run1` R1 and W1; R2 and W2 from `run2`.

Here are all possible orderings (under a sequentially consistent memory model):

Order				*x
R1	W1	R2	W2	4
R1	R2	W1	W2	3
R1	R2	W2	W1	2
R2	W2	R1	W1	4
R2	R1	W2	W1	2
R2	R1	W1	W2	3

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Yes.

- Variables y and z are independent.
- Variable x is only read.

RAR dependency allows parallelization.

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

No, z = 42 or z = 7.

RAW inhibits parallelization: can't change ordering.
Also known as a true dependency.

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

No. Again, z = 42 or z = 7.

- WAR is also known as a anti-dependency.
- But, we can modify this code to enable parallelization.

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Not always:

```
z = very_long_function(x) + 5
x = very_long_calculation()
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, z = 42 or z = 7.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, z = 42 or z = 7.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

```
z_copy = x + 5
z = x + 40
```

Synchronization

You'll need some sort of synchronization to get sane results from multithreaded programs.

Hopefully you remember semaphores and mutexes from earlier courses.

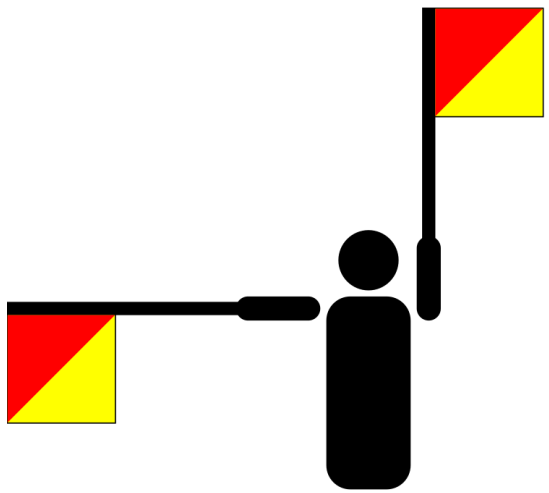


Image Credit: Denelson83

Mutual Exclusion

Mutexes are a very common of synchronization. As a reminder:

- Only one thread can access code protected by a mutex at a time.
- All other threads must wait until the mutex is free before they can execute the protected code.

Using a Mutex

Here's an example of using mutexes:

pthreads

```
pthread_mutex_t m1_static =
     PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2_dynamic;

pthread_mutex_init(&m2_dynamic, NULL);
...
pthread_mutex_destroy(&m1_static);
pthread_mutex_destroy(&m2_dynamic);
```

C++11

```
mutex m1;
mutex * m2;

m2 = new mutex();
// ...

delete (m2);
```

You can initialize mutexes statically (as with `m1_static`) or dynamically (`m2_dynamic`).

If you want to include attributes, you need to use the dynamic version.

Mutex Attributes

Both threads and mutexes use the notion of attributes.

- **Protocol**: specifies the protocol used to prevent priority inversions for a mutex.
- **Prioceiling**: specifies the priority ceiling of a mutex.
- **Process-shared**: specifies the process sharing of a mutex.

You can specify a mutex as *process shared* so that you can access it between processes.

Try, Try Again

There is also the idea of trylock: you attempt to lock the mutex in a way that you won't get blocked whether we acquire the lock or not.

The function returns a value to indicate if we succeeded and it is mandatory that we check.

If successful, proceed.

If unsuccessful then we'll have to try again at some point.

Protect the End Zone

Key idea: locks protect resources; only one thread can hold a lock at a time.

A second thread trying to obtain the lock (i.e. contending for the lock) has to wait, or block, until the first thread releases the lock.

So only one thread has access to the protected resource at a time.

The code between the lock acquisition and release is known as the critical region or critical section.

Don’t Lock Too Much

Excessive use of locks can serialize programs.

Consider two resources *A* and *B* protected by a single lock ℓ.

Then a thread that’s just interested in *B* still has acquire ℓ, which requires it to wait for any other thread working with *A*.

Example: Linux Big Kernel Lock

Spinlocks

Spinlocks are a variant of mutexes, where the waiting thread repeatedly tries to acquire the lock instead of sleeping.

Use spinlocks when you expect critical sections to finish quickly.

Spinning for a long time consumes lots of CPU resources.

Many lock implementations use both sleeping and spinlocks: spin for a bit, then sleep longer.

Why Use Spinlocks?

When would we ever want to use a spinlock?

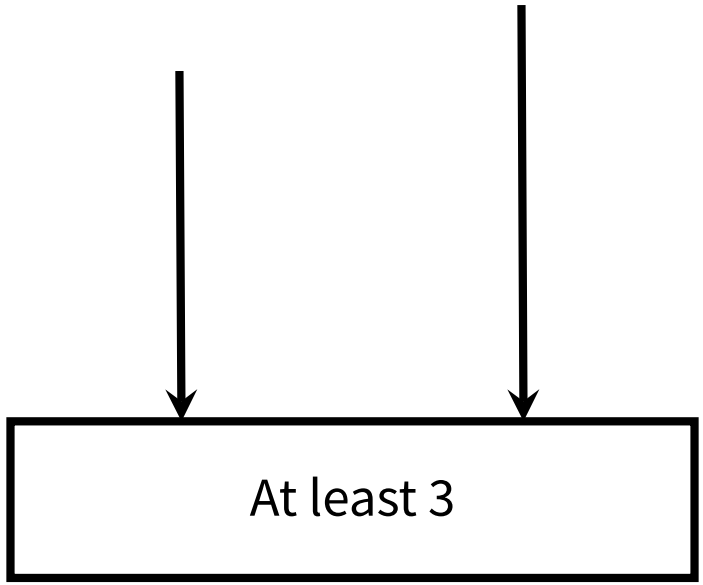
What we normally expect is to block until the lock becomes available.

But that means a process switch, and then a switch back in the future when the lock is available. This takes nonzero time.

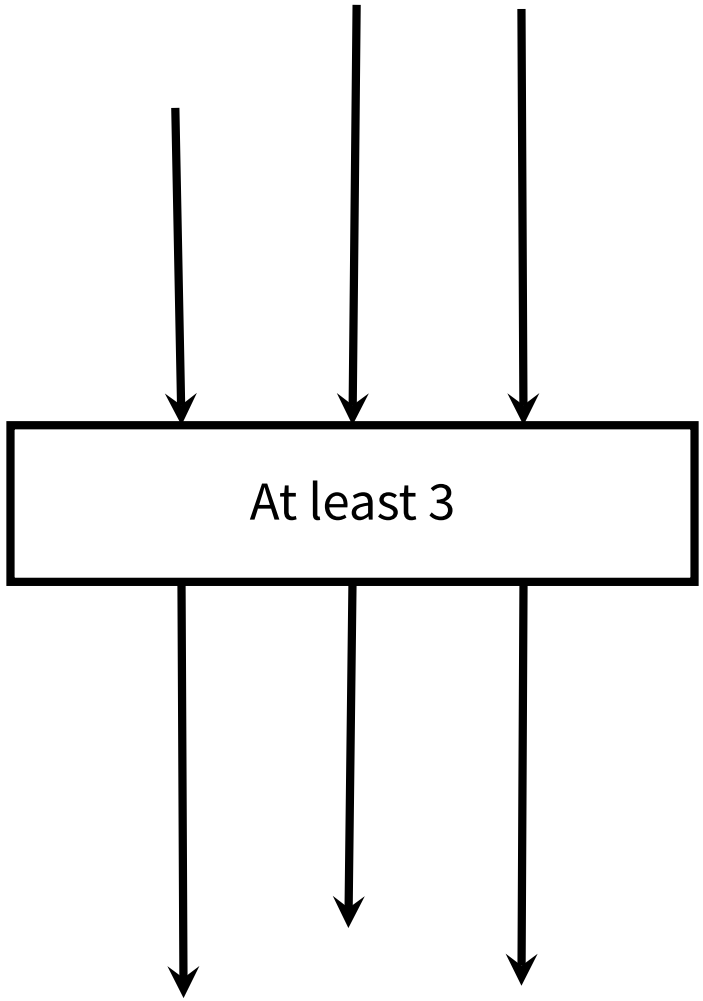
It's optimal to use a spinlock if the amount of time we expect to wait for the lock is less than the amount of time it would take to do two process switches.

As long as we have a multicore system.

Barriers



Barriers



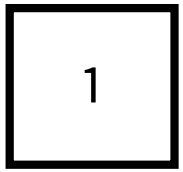
More about this later.

Modern CPUs support atomic operations, such as compare-and-swap; enable experts to write lock-free code.

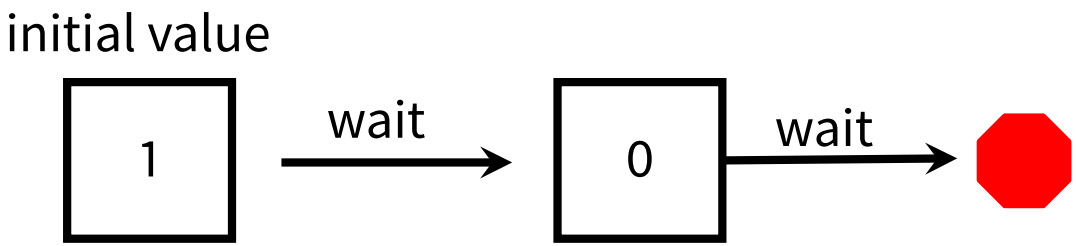
Lock-free implementations are complicated and must still contain certain synchronization constructs.

Semaphores: share # instances of a resource

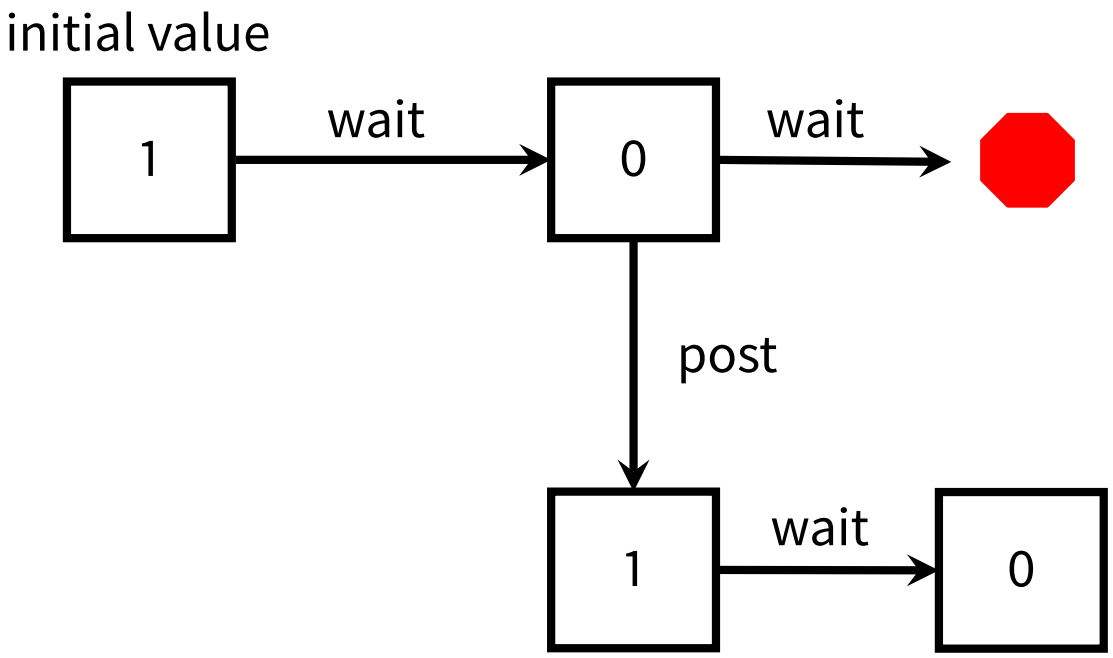
initial value



Semaphores: can wait until at least 1 available



Semaphores: another thread has posted, carry on



Semaphores Usage

```c
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

Reader/Writer Locks

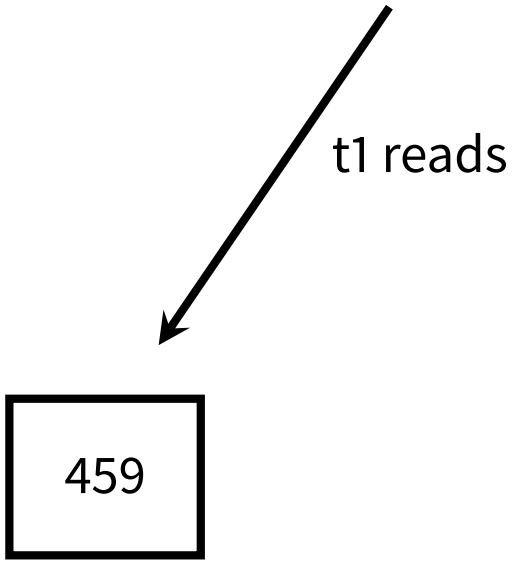
Reads don’t interfere with one another so we can let them run in parallel!

But sometimes writes occur, and nobody can read when this happens.

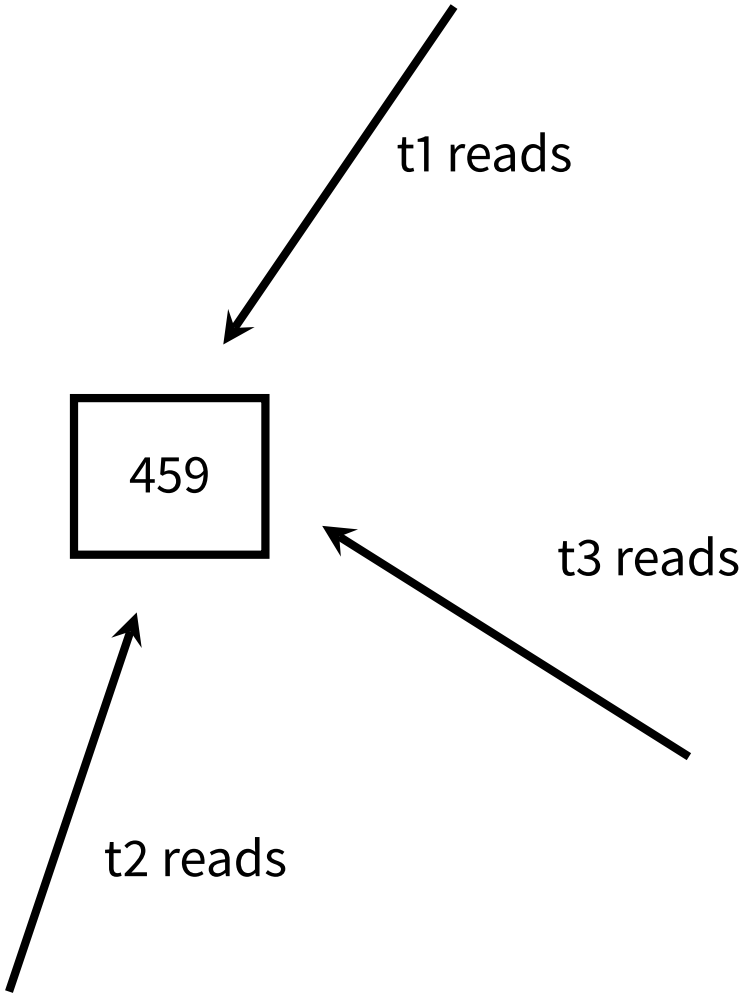
Reader/Writer Rules

1. Any number of readers may be in the critical section simultaneously.
2. Only one writer may be in the critical section (and when it is, no readers are allowed).

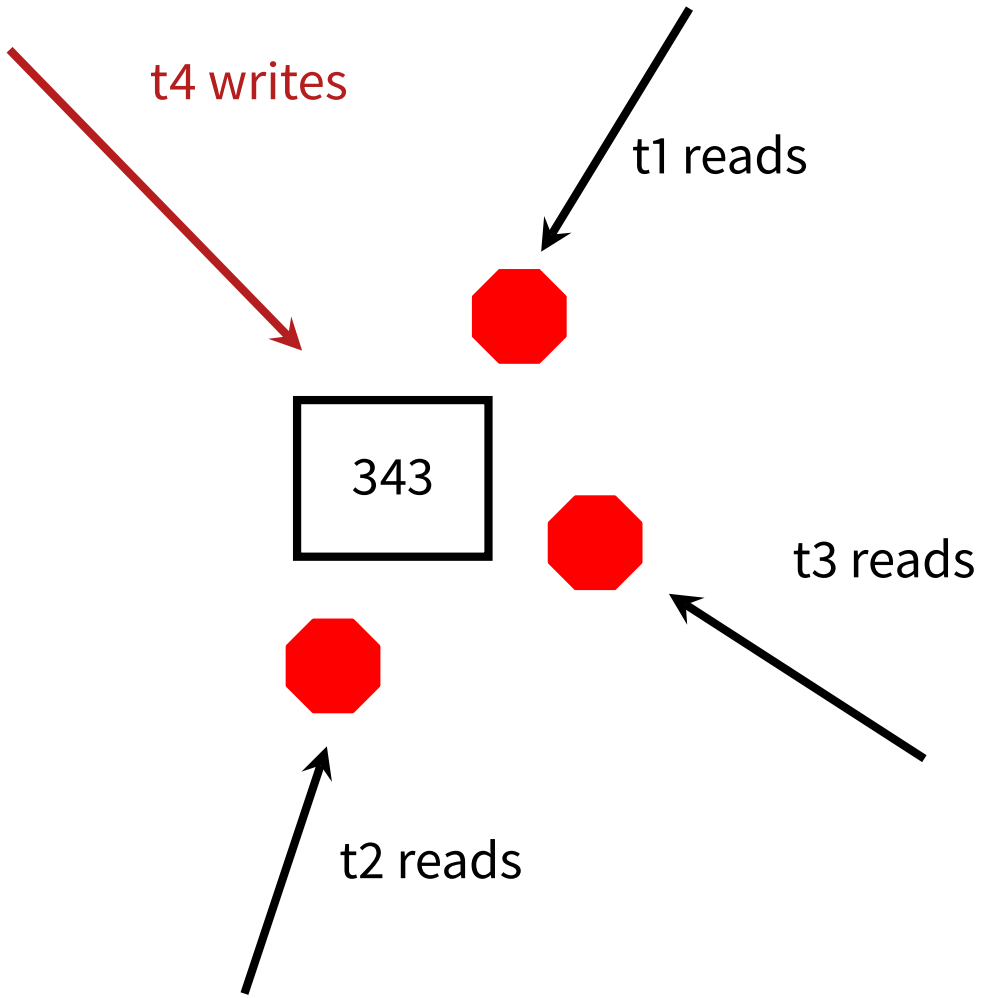
Read-Write Locks



OK to read in parallel



Must wait for write to finish



RW Lock Syntax

The type for the lock is `pthread_rwlock_t`.

```
pthread_rwlock_init( pthread_rwlock_t * rwlock, pthread_rwlockattr_t * attr )
pthread_rwlock_rdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_wrlock( pthread_rwlock_t * rwlock )
pthread_rwlock_trywrlock( pthread_rwlock_t  *  rwlock )
pthread_rwlock_unlock( pthread_rwlock_t * rwlock )
pthread_rwlock_destroy( pthread_rwlock_t * rwlock )
```

Mine, Mine, Mine!

In theory, the same thread may lock the same rwlock *n* times.

Just remember to unlock it *n* times as well.

Readers get priority? Implementation defined.

Without RW Locks

```
int readers;
pthread_mutex_t mutex;
sem_t roomEmpty;

void init( ) {
  readers = 0;
  pthread_mutex_init( &mutex, NULL );
  sem_init( &roomEmpty, 0, 1 );
}

void cleanup( ) {
  pthread_mutex_destroy( &mutex );
  sem_destroy( &roomEmpty );
}
```

```
void* writer( void* arg ) {
  sem_wait( &roomEmpty );
  write_data( arg );
  sem_post( &roomEmpty );
}

void* reader( void* read ) {
  pthread_mutex_lock( &mutex );
  readers++;
  if ( readers == 1 ) {
    sem_wait( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
  read_data( arg );
  pthread_mutex_lock( &mutex );
  readers--;
  if ( readers == 0 ) {
    sem_post( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
}
```

With RW Locks

```
pthread_rwlock_t rwlock;

void init( ) {
  pthread_rwlock_init( &rwlock, NULL
       );
}

void cleanup( ) {
  pthread_rwlock_destroy( &rwlock );
}
```

```
void* writer( void* arg ) {
  pthread_rwlock_wrlock( &rwlock );
  write_data( arg );
  pthread_rwlock_unlock( &rwlock );
}

void* reader( void* read ) {
  pthread_rwlock_rdlock( &rwlock );
  read_data( arg );
  pthread_rwlock_unlock( &rwlock );
}
```

Conclusion: don’t reinvent the wheel!

`volatile` Keyword

- Used to notify the compiler that the variable may be changed by "external forces". For instance,

```
int i = 0;

while (i != 255) {
  ...
```

`volatile` prevents this from being optimized to:

```
int i = 0;

while (true) {
  ...
```

- Usually wrong unless there is a **very** good reason for it.

He’s a little volatile...

I read that `volatile` variables aren’t stored in registers, should I worry?

He’s a little volatile...

I read that `volatile` variables aren’t stored in registers, should I worry?

He's a little volatile...

I read that `volatile` variables aren't stored in registers, should I worry?

`volatile` in C was only designed to:

- Allow access to memory mapped devices.
- Allow uses of variables between `setjmp` and `longjmp`.
- Allow uses of `sig_atomic_t` variables in signal handlers.

Remember, things can also be reordered by the compiler, `volatile` doesn't prevent this.

Also, it's likely your variables could be in registers the majority of the time, except in critical areas.