Lecture 6 — Race Conditions & Synchronization

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

Department of Electrical and Computer Engineering
University of Waterloo

January 25, 2018

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.

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!

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;
}

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?

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.

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

Here are all possible orderings:

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
```

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

We'll start by talking about how to use mutual exclusion in Pthreads.

Summary of Memory-carried Dependencies

		Second Access	
		Read	**Write**
First Access	**Read**	No Dependency Read After Read (RAR)	Anti-dependency Write After Read (WAR)
	Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)

Mutual Exclusion

Mutexes are the most basic type 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.

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.

Mutex Example

PThreads

```
// code
pthread_mutex_lock(&m1);
// protected code
pthread_mutex_unlock(&m1);
// more code
```

C++11 Threads

```
// code
m1.lock();
// protected code
m1.unlock();
// more code
```

- Everything within the `lock` and `unlock` is protected.
- Be careful to avoid deadlocks if you are using multiple mutexes (always acquire locks in the same order across threads).
- Another useful primitive is `pthread_mutex_trylock`. later.

Why are we bothering with locks? Data races. A data race occurs when two concurrent actions access the same variable and at least one of them is a **write**. (This shows up on Assignment 1!)

```c
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        ++counter;
    }
}

int main(int argc, char *argv[]) {
    // Create 8 threads
    // Join 8 threads
    printf("counter = %i\n", counter);
}
```

Use Mutexes

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        pthread_mutex_lock(&mutex);
        ++counter;
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char *argv[]) {
    // Create 8 threads
    // Join 8 threads
    pthread_mutex_destroy(&mutex);
    printf("counter = %i\n", counter);
}
```

- Call `lock` on mutex ℓ_1. Upon return from `lock`, your thread has exclusive access to ℓ_1 until it `unlock`s it.
- Other calls to `lock` ℓ_1 will not return until `m1` is available.

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.

Try, Try Again

Some mutex implementations also provide a “try-lock” primitive.

This grabs the lock if it’s available, or returns control to the thread if it’s not.

This enables the thread to do something else. (Kind of like non-blocking I/O!)

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?

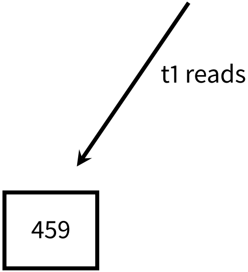
When would we ever want to use a spinlock?

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

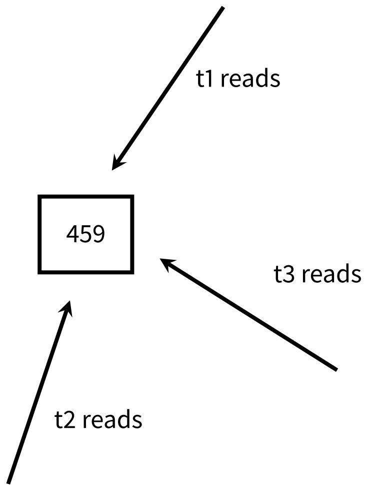
But that means a process switch, and then a switch back in the future when the lock is available. This takes nonzero time.

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.

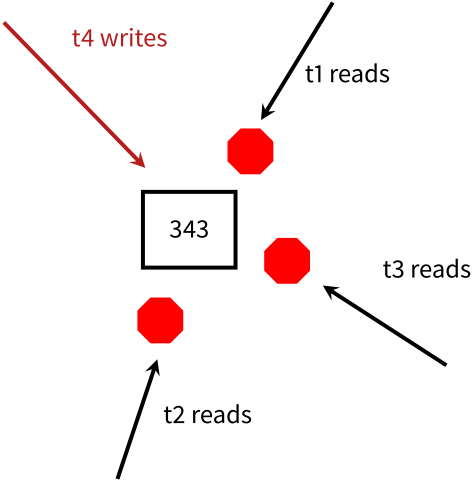
As long as we have a multicore system.


t1 reads
459

OK to read in parallel



Must wait for write to finish



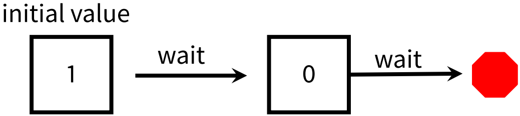
Multiple threads can hold a read lock
`(pthread_rwlock_rdlock)`
but only one thread may hold the associated write lock
`(pthread_rwlock_wrlock)`;
grabbing the write waits until current readers are done.

Semaphores: share # instances of a resource

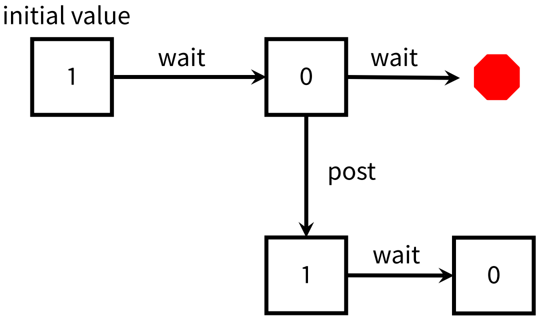
initial value

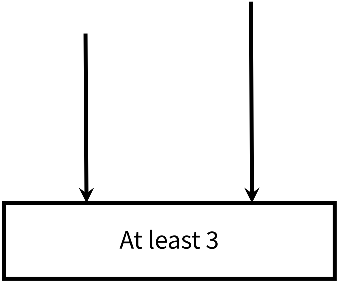
$\boxed{1}$

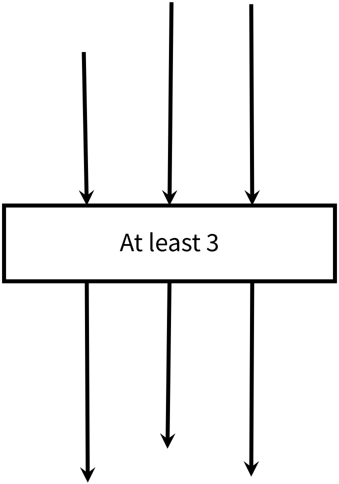
Semaphores: can wait until at least 1 available



Semaphores: another thread has posted, carry on




At least 3


At least 3

Lock-Free Algorithms

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 Usage

```
#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);
```

Semaphores As Mutexes

How could you use as `semaphore` as a `mutex`?

How could you use as `semaphore` as a `mutex`?

- If the initial `value` is 1 and you use `wait` to lock and `post` to unlock, it's equivalent to a `mutex`.

Semaphores for Signalling

Here's an example from the recommended book. How would you make this always print "Thread 1" then "Thread 2" using semaphores?

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

void* p1 (void* arg) { printf("Thread 1\n"); }

void* p2 (void* arg) { printf("Thread 2\n"); }

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return EXIT_SUCCESS;
}
```

Here's a possible solution. Is it actually correct?

```
sem_t sem; // new
void* p1 (void* arg) {
  printf("Thread␣1\n"); // new
  sem_post(&sem);
}
void* p2 (void* arg) {
  sem_wait(&sem);        // new
  printf("Thread␣2\n");
}

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    sem_init(&sem, 0, /* value: */ 1); // new
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    sem_destroy(&sem);
}
```

Semaphores for Signalling

1. `value` is initially 1.
2. Say `p2` hits its `sem_wait` first and succeeds.
3. `value` is now 0 and `p2` prints "Thread 2" first.

- If `p1` happens first, it would just increase `value` to 2.

1. `value` is initially 1.
2. Say `p2` hits its `sem_wait` first and succeeds.
3. `value` is now 0 and `p2` prints "Thread 2" first.

- If `p1` happens first, it would just increase `value` to 2.
- Fix: set the initial `value` to `0`.

  Then, if `p2` hits its `sem_wait` first, it will not print until `p1` posts (and prints "Thread 1") first.

`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.