# Lecture 08—Race Conditions; Mutexes

January 21, 2015

# Roadmap

Past: Non-blocking I/O;

Now: Race Conditions, Locking.

# curl_multi_wait

example:
`https://gist.github.com/clemensg/4960504`

# Part I

# Race Conditions

# Race Conditions

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

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

Race conditions arise between variables which are shared between threads.

# Example Data Race (Part 1)

```
#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;
}
```

# Example Data Race (Part 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;
}
```

Do we have a data race? Why or why not?

# Example Data Race (Part 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;
}
```

Do we have a data race? Why or why not?

- No, we don't. Only one thread is active at a time.

# Example Data Race (Part 2B)

```
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;
}
```

Do we have a data race now? Why or why not?

# Example Data Race (Part 2B)

```
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;
}
```

Do we have a data race now? Why or why not?

- Yes, we do. We have 2 threads concurrently accessing the same data.

# Tracing our Example Data Race

What are the possible outputs? (initially `*x` is 1).

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

- Memory reads and writes are key in data races.

# Outcome of Example Data Race

- Let's call the read and write from `run1` R1 and W1; R2 and W2 from `run2`.
- Assuming a sane[1] memory model, $R_n$ must precede $W_n$.

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 |

[1] sequentially consistent

# Detecting Data Races Automatically

Dynamic and static tools can help find data races in your program.

- `helgrind` is one such tool. It runs your program and analyzes it (and causes a large slowdown).

Run with `valgrind --tool=helgrind <prog>`.

It will warn you of possible data races along with locations.

For useful debugging information, compile with debugging information (`-g` flag for `gcc`).

# Helgrind Output for Example

```
==5036== Possible data race during read of size 4 at
         0x53F2040 by thread #3
==5036== Locks held: none
==5036==    at 0x400710: run2 (in datarace.c:14)
...
==5036==
==5036== This conflicts with a previous write of size 4 by
         thread #2
==5036== Locks held: none
==5036==    at 0x400700: run1 (in datarace.c:8)
...
==5036==
==5036== Address 0x53F2040 is 0 bytes inside a block of size
         4 alloc'd
...
==5036==    by 0x4005AE: main (in datarace.c:19)
```

# Mutual Exclusion

Mutexes are the most basic type of synchronization.

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

# Live Coding Example: Mutual Exclusion

# Creating Mutexes—Pthreads Example

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;

pthread_mutex_init(&m2, NULL);
...
pthread_mutex_destroy(&m1);
pthread_mutex_destroy(&m2);
```

- Two ways to initialize mutexes: statically and dynamically
- If you want to include attributes, you need to use the dynamic version

# Creating Mutexes—C++ Example

```
mutex m1;
mutex *m2;

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

delete(m2);
```

# Mutex 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. In that case, you need to use shared memory and `mmap`, which we won't get into.

# Using Mutexes: Pthreads Example

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

- Everything within the `lock` and `unlock` is protected.
- Be careful to avoid deadlocks if you are using multiple mutexes.
- Also you can use `pthread_mutex_trylock`, if needed.

# Using Mutexes: Pthreads Example

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

# Data Race Example

Recall that dataraces occur when two concurrent actions access the same variable and at least one of them is a **write**.

```
...
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);
}
```

Is there a datarace in this example? If so, how would we fix it?

# Example Problem Solution

```
...
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);
}
```