2023 Fall ECE 344: Operating Systems
Lecture 19 1.0.0

Locks

Jon Eyolfson
2023 Fall

Data Races Can Occur When Sharing Data

A data race is when two concurrent actions access the same variable and at least one of them is a write operation

Atomic Operations are Indivisible

Any atomic instruction you may assume happens all at once

This means you can not preempt it

However, between two atomic instructions, you may be preempted

Three Address Code (TAC) is Intermediate Code Used by Compilers

TAC is mostly used for analysis and optimization by compilers

Statements represent one fundamental operation (assume each is atomic)
    Useful to reason about data races and easier to read than assembly

Statements have the form: *result* $:=$ *operand*$_1$ *operator operand*$_2$

GIMPLE is the TAC used by `gcc`

To see the GIMPLE representation of your compilation use:

`-fdump-tree-gimple` flag

To see all the three address code generated by the compiler (gcc) use:

`-fdump-tree-all` flag

GIMPLE is easier to reason about your code at a low-level without assembly

17-threads-implementation/pthread-datarace.c Data Race

Instead of `count`, we'll look at `pcount` (the pointer to count, which is a global)

The GIMPLE is the following:

```
D.1 = *pcount;
D.2 = D.1 + 1;
*pcount = D.2;
```

Assuming that two threads execute this once each and initially `*pcount = 0`

What are the possible values of `*pcount`?

To Analyze Data Races, You Have to Assume All Preemption Possibilities

Let's call the read and write from thread 1 R1 and W1 (R2 and W2 from thread 2)

We'll assume no re-ordering of instructions: always read then write in a thread

All possible orderings:

Order				*pcount
R1	W1	R2	W2	2
R1	R2	W1	W2	1
R1	R2	W2	W1	1
R2	W2	R1	W1	2
R2	R1	W2	W1	1
R2	R1	W1	W2	1

You Can Create Mutexes Statically or Dynamically

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;

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

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

Everything Within the Lock and Unlock is a Critical Section

```
// 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 a `pthread_mutex_trylock` if needed

Adding a Lock to Prevent the Data Race

```
...
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; /* New */
static int counter = 0;

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

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

A Critical Section Means Only One Thread Executes Instructions

Safety (aka mutual exclusion)
- There should only be a single thread in a critical section at once

Liveness (aka progress)
- If multiple threads reach a critical section, one must proceed
- The critical section can't depend on outside threads
  - You can mess up and deadlock (threads don't make progress)

Bounded waiting (aka starvation-free)
- A waiting thread must eventually proceed

Critical Sections Should Also Have Minimal Overhead

Efficient

You don’t want to consume resources while waiting

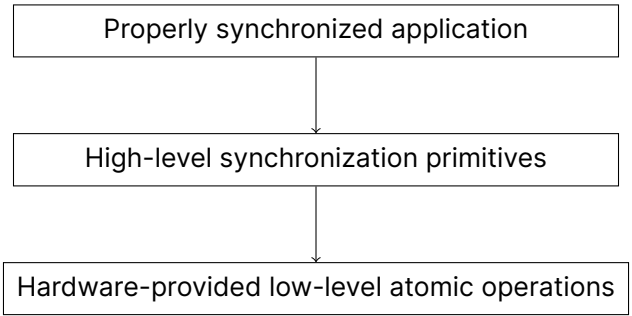
Fair

You want each thread to wait approximately the same time

Simple

It should be easy to use, and hard to misuse

Similar to Libraries, You Want Layers of Synchronization



You Could Use a Lock to Implement Critical Sections

Assuming a uniprocessor operating system, you could implement locks as follows:

```
void lock() {
  disable_interrupts();
}
void unlock() {
  enable_interrupts();
}
```

This would disable concurrency (assuming it ignores signals and interrupts)

Not going to work on multiprocessors (and OS won't let you change hardware)

Let’s Try to Implement a Lock in Software

```
void init(int *l) {
  *l = 0;
}
void lock(int *l) {
  while (*l == 1);
  *l = 1;
}
void unlock(int *l) {
  *l = 0;
}
```

What’s the issue with this implementation?

Let’s Try to Implement a Lock in Software

```
void init(int *l) {
  *l = 0;
}
void lock(int *l) {
  while (*l == 1);
  *l = 1;
}
void unlock(int *l) {
  *l = 0;
}
```

What’s the issue with this implementation?

- It’s not safe (both threads can be in the critical section)
- It’s not efficient, it wastes CPU cycles (busy wait)