CS 111: Operating System Principles
Lecture 11

Locks
3.0.0

Jon Eyolfson
November 2, 2021

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

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 of the three address code generated by the compiler use:

`-fdump-tree-all` flag

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

lecture-10/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;
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);
}
```

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

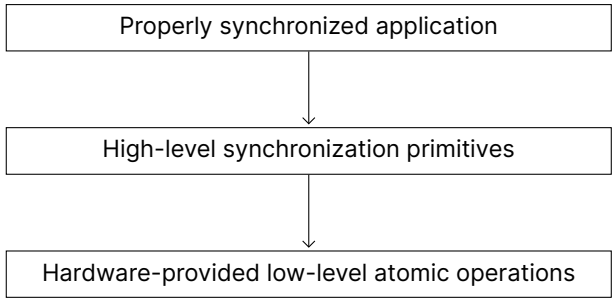
Fair

- 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)

You Can Implement Locks in Software with Minimal Hardware

You hardware requirements just have to ensure:

- Loads and stores are atomic
- Instructions execute in order

There's 2 main algorithms you could use:

Peterson's algorithm and Lamport's bakery algorithm

The problem is that they don't scale well, and processors execute out-of-order

Let's Assume a Magical Atomic Function — `compare_and_swap`

`compare_and_swap(int *p, int old, int new)` is atomic

- It returns the original value pointed to
- It only swaps if the original value equals `old`, and changes it to `new`

Let's give it another shot:

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

What We Implement is Essentially a Spinlock

Compare and swap is a common atomic hardware instruction

On x86 this is the `cmpxchg` instruction (compare and exchange)

However it still has this “busy wait” problem

Consider a uniprocessor system, if you can’t get the lock, you should yield

Let the kernel schedule another process, that may free the lock

On a multiprocessor machine, it depends

Let's Add a Yield

```
void lock(int *l) {
  while (compare_and_swap(l, 0, 1)) {
    yield();
  }
}
```

Now we have a thundering herd problem

- Multiple threads may be waiting on the same lock

We have no control over who gets the lock next

- We need to be able to reason about it (FIFO is okay)

We Can Add a Wait Queue to the Lock

```
void lock(int *l) {
  while (compare_and_swap(l, 0, 1)) {
    // add myself to the wait queue
    yield();
  }
}
void unlock(int *l) {
  *l = 0;
  if (/* threads in wait queue */) {
    // wake up one thread
  }
}
```

There are 2 issues with this: 1) lost wakeup, and 2) the wrong thread gets the lock

Lost Wakeup Example

```
1  void lock(int *l) {
2    while (compare_and_swap(l, 0, 1)) {
3      // add myself to the wait queue
4      yield();
5    }
6  }
7  void unlock(int *l) {
8    *l = 0;
9    if (/* threads in wait queue */) {
10     // wake up one thread
11   }
12 }
```

Assume we have thread 1 (T1) and thread 2 (T2), thread 2 holds the lock

T1 runs line 2 and fails, swap to T2 that runs lines 10-12, T1 runs lines 3 -4

T1 will never get woken up!

Wrong Thread Getting the Lock Example

```
1  void lock(int *l) {
2    while (compare_and_swap(l, 0, 1)) {
3      // add myself to the wait queue
4      yield();
5    }
6  }
7  void unlock(int *l) {
8    *l = 0;
9    if (/* threads in wait queue */) {
10     // wake up one thread
11   }
12 }
```

Assume we have T1, T2, and T3. T2 holds the lock, T3 is in queue.

T2 runs line 9, swap to T1 which runs line 2 and succeeds

T1 just stole the lock from T3!

To Fix These Problems, We Can Use Two Variables (One to Guard)

```
typedef struct {
  int lock;
  int guard;
  queue_t *q;
} mutex_t;

void lock(mutex_t *m) {
  while (
    compare_and_swap(m->guard, 0, 1)
  );
  if (m->lock == 0) {
    m->lock = 1; // acquire mutex
    m->guard = 0;
  } else {
    enqueue(m->q, self);
    m->guard = 0;
    yield();
    // wakeup transfers the lock here
  }
}
```

```
void unlock(mutex_t *m) {
  while (
    compare_and_swap(m->guard, 0, 1)
  );
  if (queue_empty(m->q)) {
    // release lock, no one needs it
    m->lock = 0;
  }
  else {
    // direct transfer mutex
    // to next thread
    wakeup(dequeue(m->q));
  }
  m->guard = 0;
}
```

Remember What Causes a Data Race

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

We could have any many readers as we want

We don’t need a mutex as long as nothing writes at the same time

We need different lock modes for reading and writing

Read-Write Locks

With mutexes/spinlocks, you have to lock the data,
even for a read since you don't know if a write could happen

Reads can happen in parallel, as long as there's no write

Multiple threads can hold a read lock (`pthread_rwlock_rdlock`),
but only one thread may hold a write lock (`pthread_rwlock_wrlock`)
and will wait until the current readers are done

We Can Use A Guard To Keep Track of Readers

```
typedef struct {
  int nreader;
  lock_t guard;
  lock_t lock;
} rwlock_t;

void write_lock(rwlock_t *l) (
  lock(&l->lock);
}

void write_unlock(rwlock_t *l) (
  unlock(&l->lock);
}
```

```
void read_lock(rwlock_t *l) (
  lock(&l->guard);
  ++nreader;
  if (nreader == 1) { // first reader
    lock(&l->lock);
  }
  unlock(&l->guard);
}
void read_unlock(rwlock_t *l) (
  lock(&l->guard);
  --nreader;
  if (nreader == 0) { // last reader
    unlock(&l->lock);
  }
  unlock(&l->guard);
}
```

We Want Critical Sections to Protect Against Data Races

We should know what data races are, and how to prevent them:

- Mutex or spinlocks are the most straightforward locks
- We need hardware support to implement locks
- We need some kernel support for wake up notifications
- If we know we have a lot of readers, we should use a read-write lock