More Locks and Semaphores

2025 Winter ECE353: Systems Software

Lecture 14

Jon Eyolfson

2.0.0

You Can Implement Locks in Software with Minimal Hardware

Your hardware requirements just have to ensure:

- Loads and stores are atomic
- Instructions execute in order

There are 2 main algorithms you could use:
Peterson's algorithm and Lamport's bakery algorithm

However, 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, you could try again

Let’s Add a Yield

```
void lock(int *l) {
  while (compare_and_swap(l, 0, 1)) {
    thread_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 lock wait queue
    thread_sleep();
  }
}
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      thread_sleep();
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     thread_sleep();
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!

We Can Use Two Variables to Fix This (One to Guard)

```c
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;
    thread_sleep();
    // wakeup transfers the lock here
  }
}
```

```c
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
    thread_wakeup(dequeue(m->q));
  }
  m->guard = 0;
}
```

There’s STILL A Data Race

After a thread calls `lock`, it could get interrupted right before the `thread_sleep`

However, it’s been added to the wait queue, so `thread_wakeup`
would try to wake up a thread that’s not sleeping yet (we know it’s about to)

We could simply retry the call to `thread_wakeup`
until the thread finally calls `thread_sleep`

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

Locks Ensure Mutual Exclusion

Only one thread at a time can be between the `lock` and `unlock` calls

It does not help you ensure ordering between threads

How would we ensure an ordering between two threads?

Problem: Make One Thread Always Print First

Thread 1 (`print_first`)
```
printf("This is first\n");
```

Thread 2 (`print_second`)
```
printf("I'm going second\n");
```

Try executing `./ordered-print` and see what happens

Recall: `printf` is thread safe, which you may need to ensure

- Try executing `./safe-print` which (oddly) prints using multiple system calls
- Practice: ensure `./safe-print` behaves the same as `./ordered-print`

Semaphores are Used for Signaling

Semaphores have a `value` that's shared between threads (optionally processes)

- Think of `value` as an integer that is always ≥ 0

It has two fundamental operations `wait` and `post`

- `wait` decrements the value atomically
- `post` increments the value atomically

If `wait` will not return until the value is greater than 0

You can initially set `value` to whatever you want

- That number of `wait` calls may occur without any `post` calls

Semaphore API is Similar to `pthread` Locks

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

int sem_post(sem_t *sem);
```

All functions return 0 on success

The `pshared` argument is a boolean, you can set it to 1 for IPC
- For IPC the semaphore needs to be in shared memory

Problem: Make One Thread Always Print First

See `ordered-print.c` for the full code

Note: `return` statements are removed for space

```
void* print_first(void* arg) {
  printf("This is first\n");
}

void* print_second(void* arg) {
  printf("I'm going second\n");
}

int main(int argc, char *argv[]) {
  /* Initialize, create, and join threads */
}
```

This Always Executes `print_first` Then `print_second`

```c
static sem_t sem; /* New */

void* print_first(void* arg) {
  printf("This is first\n");
  sem_post(&sem); /* New */
}

void* print_second(void* arg) {
  sem_wait(&sem); /* New */
  printf("I'm going second\n");
}

int main(int argc, char *argv[])
{
  sem_init(&sem, 0, 0); /* New */
  /* Initialize, create, and join threads */
}
```

No Matter Which Thread Executes First, We Get the Same Order

The `value` is initially 0

Assume `print_second` executes first
  It executes `sem_wait`, which is 0, and doesn't continue

`print_first` doesn't have to wait, it prints first before it increments the `value`

`print_second` can then execute its print statement

What happens if we initialized the `value` to 1?

We Can Use a Semaphore as a Mutex

How?

Using a Semaphore as a Mutex, Note the `value`

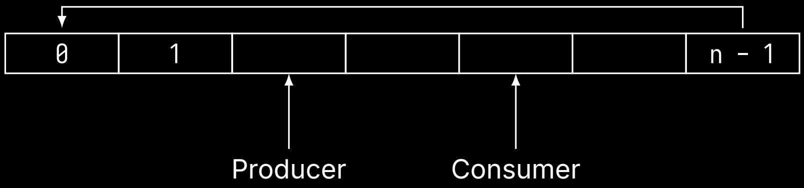
```c
static sem_t sem; /* New */
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        sem_wait(&sem); /* New */
        ++counter;
        sem_post(&sem); /* New */
    }
}

int main(int argc, char *argv[]) {
  sem_init(&sem, 0, 1); /* New */
  /* Initialize, create, and join multiple threads */
  printf("counter = %i\n", counter);
}
```

Can We Come Up with a Solution for a Producer/Consumer Problem?

Assume you have a circular buffer (each slot is either empty or filled):



The producer should write to the buffer (if the buffer is not full)
The consumer should read from the buffer (if the buffer is not empty)

All consumers share an index and all producers share an index
In both cases the index is initially 0 and increases sequentially

Problem 1: Ensure Producers Never Overwrite Filled Slots

```
static uint32_t buffer_size;

void init_semaphores() {
  sem_init(&empty_slots, 0, /* ? */);
}
void producer() {
  while (/* ... */) {
    /* spend time producing data */
    fill_slot();
  }
}
void consumer() {
  while (/* ... */) {
    empty_slot();
    /* spend time consuming data */
  }
}
```

Use a Semaphore to Track the Number of Empty Slots

```c
void init_semaphores() {
  sem_init(&empty_slots, 0, buffer_size);
}
void producer() {
  while (/* ... */) {
    /* spend time producing data */
    sem_wait(&empty_slots); /* New */
    fill_slot();
  }
}
void consumer() {
  while (/* ... */) {
    empty_slot();
    sem_post(&empty_slots); /* New */
    /* spend time consuming data */
  }
}
```

What is our next problem?

Problem 2: Ensure Consumers Never Consume Empty Slots

```
void init_semaphores() {
  sem_init(&empty_slots, 0, buffer_size);
  sem_init(&filled_slots, 0, /* ? */);
}
void producer() { while (/* ... */) {
  /* spend time producing data */
  sem_wait(&empty_slots);
  fill_slot();
} }
void consumer() { while (/* ... */) {
  empty_slot();
  sem_post(&empty_slots);
  /* spend time consuming data */
} }
```

Two Semaphores Ensure Proper Order for Producers and Consumers

```
void init_semaphores() {
  sem_init(&empty_slots, 0, buffer_size);
  sem_init(&filled_slots, 0, 0);
}
void producer() { while (/* ... */) {
  /* spend time producing data */
  sem_wait(&empty_slots);
  fill_slot();
  sem_post(&filled_slots); /* New */
} }
void consumer() { while (/* ... */) {
  sem_wait(&filled_slots); /* New */
  empty_slot();
  sem_post(&empty_slots);
  /* spend time consuming data */
} }
```

What Happens If We Initialize Both Semaphore Values to 0?

```
void init_semaphores() {
  sem_init(&empty_slots, 0, 0);
  sem_init(&filled_slots, 0, 0);
}
void producer() { while (/* ... */) {
  /* spend time producing data */
  sem_wait(&empty_slots);
  fill_slot();
  sem_post(&filled_slots);
} }
void consumer() { while (/* ... */) {
  sem_wait(&filled_slots);
  empty_slot();
  sem_post(&empty_slots);
  /* spend time consuming data */
} }
```

We Used Semaphores to Ensure Proper Order

Previously we ensured mutual exclusion, now we can ensure order

- Semaphores contain an initial value you choose
- You can increment the value using post
- You can decrement the value using wait (it blocks if the current value is 0)
- You still need to be prevent data races