Locks

Chapter 28

Previously in CS212…

- We’ve talked about why we might want to use threads
    - Concurrency
    - Preventing the entirety of a process from being blocked
    - Sharing data between concurrently running threads of execution
- We’ve talked about how we can do this on Unix systems in C
    - pthreads
    - locks (mutex)
    - condition variables
- Tangent: I also tried (in vain) to remember name of the CAP Theorem when discussing webservices
- Now we need to talk more about how the OS deals with these things, starting with locks

Locks

- Essentially a variable that acts much like a lock on door
  - The door gives access to the critical section of code
  - Only one thread can enter at a time
  - As the thread enters, it locks the door to keep other threads out
  - When the thread is done, it unlocks the door for other threads to gain access
- We need to consider a few measures of how well a locking solution works:
  - Mutual exclusion – does it ensure only one thread for access?
  - Fairness – do all the threads get a fair chance at the lock (avoiding starvation)?
  - Performance – what is the overhead needed for the locking mechanism?

Approaches – Controlling Interrupts

- The core need for locks is that the scheduler can interrupt a thread at any time and thus stop them in the middle of important work potentially putting the system in a non-deterministic state between one or more threads

- What if we just disable interrupts when we lock and enable them when we unlock?
  - Malicious/greedy/buggy programs can dominate the system and lock out the OS
  - Doesn't work with multiprocessors as the threads might not be on the same CPU
  - Without interrupts other events (like I/O) might be missed

- Does have limited application within the OS kernel, but not for general purpose use

Approaches – Load and Store Flag

- We said previously that a lock is a variable, so why not just create a variable in our code, and have one thread check for the right value, and the other thread change it?

```
void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 28.1: **First Attempt: A Simple Flag**

If we get interrupted just as we are about to set the flag to 1, another process might be able to as well!

WE NEED HARDWARE SUPPORT!

- Remember that the hardware supports a specific set of low-level instructions (assembly)

- Most single lines of C code are multiple low-level instructions
  - We can be interrupted in between any of those instructions

- We need low-level support for a mechanism that maps or lock to a single uninterrupted instruction

Test-And-Set Operation

- Gets the current value of the lock and sets it to be the new value
- Behavior is like this C code (but runs as one instruction):

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

- If the lock is 0, TAS gives us 0, but sets the lock to 1 indicating we have the lock
- If the lock is already 1, TAS gives us 1 and sets the lock to 1 meaning the lock is in use

Using a Spin Lock with Test-And-Set

- Here we can see the functions for our lock

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

How do we feel about this?

Spin Lock Evaluation

- Correctness:

- Fairness:

- Performance:

Spin Lock Evaluation

- Correctness: **Yes**
  - The single test-and-set will provide a proper mutual exclusion
- Fairness:
- Performance:

Spin Lock Evaluation

- Correctness: **Yes**
  - The single test-and-set will provide a proper mutual exclusion
- Fairness: **No**
  - No guarantee for fairness
  - Possible to spin forever (starvation)
- Performance:

Spin Lock Evaluation

- Correctness: **Yes**
  - The single test-and-set will provide a proper mutual exclusion

- Fairness: **No**
  - No guarantee for fairness
  - Possible to spin forever (starvation)

- Performance: **It depends**… (assuming a short critical section)
  - Multiple CPUs where the number of threads roughly equals the number of CPUs – **Works Okay**
  - Single CPU - **No**

A More Robust Instruction

- We aren't limited to just setting 1 or 0, we can have an instruction that provides more flexibility
- One implementation is **Compare-and-swap**

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

- Can support the same behavior as test-and-set but allows for other defined value comparisons

More Advanced Checking

- **Load-Linked and Stored-Conditional** is a different take
- Here we load a value from memory, but we save where the value came from and its old value
- This means even if another thread stored the same value, or tries to load the same data but from a different address, it will fail

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Figure 28.5: **Load-linked And Store-conditional**

A Chance for Fairness

- The Fetch-and-add locking primitive takes an old value and increments it by one

- This can be used for locks where the lock value doesn’t determine whether the lock is active or not, but rather, which specific thread will get access

- The ticket lock can help ensure that all threads make process

```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```c
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Figure 28.7: **Ticket Locks**

Spinning

- Spinning can also be though of as busy waiting
- Essentially, no work is being done, but the thread is still using CPU time repeatedly checking if the lock is free
- Generally, we’d like to avoid this if we can
  - The more threads we have, the more valuable CPU time is wasted
- What else can we do?

Alternatives to Spinning

- Yield
  - When a thread can’t get the lock, give up CPU time voluntary
  - Simple solution to de-schedule a thread back to ready state
  - With a small number of threads, it works fine, but as the thread count increases the scheduler may take longer to return to the thread that has the lock
- Queues and Sleeping
  - If we can’t get the lock, we jump into a queue and wait to be given the lock by the thread who had it last
  - Need to make sure that we coordinate the park/sleep process

Linux Futex Lock

- A two-phase lock
  - Tries to spin first (quickest way to grab the lock)
  - If that fails, it goes to sleep
- Integer lock value
  - Used the high bit to indicate that the lock is in use, and the rest to indicate how many threads are waiting for the lock
- Hybrid Solution

```
void mutex_lock (int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex (the fastpath) */
  if (atomic_bit_test_set (mutex, 31) == 0)
    return;
  atomic_increment (mutex);
  while (1) {
      if (atomic_bit_test_set (mutex, 31) == 0) {
          atomic_decrement (mutex);
          return;
      }
      /* We have to waitFirst make sure the futex value
         we are monitoring is truly negative (locked). */
      v = *mutex;
      if (v >= 0)
        continue;
      futex_wait (mutex, v);
  }
}

void mutex_unlock (int *mutex) {
  /* Adding 0x80000000 to counter results in 0 if and
     only if there are not other interested threads */
  if (atomic_add_zero (mutex, 0x80000000))
    return;

  /* There are other threads waiting for this mutex,
     wake one of them up.  */
  futex_wake (mutex);
}
```

Figure 28.10: **Linux-based Futex Locks**

Next Time

- We look at locks with respect to certain common data structures.