Synchronization so far...

We looked at how to build software locks
- Difficult
- Worse: it doesn’t really work
  » Compilers don’t think multi-threaded
  » Hardware reorders memory ops: memory consistency models

Let’s get help from the hardware!
Hardware to the rescue

- Crux of the problem:
  - We get interrupted between checking the lock and setting it to 1
  - Software locks reordered by compiler/hardware

- Possible solutions?
  - Atomic instructions: create a new assembly language instruction that checks and sets a variable atomically
    » Cannot be interrupted!
    » How do we use them?
  - Disable interrupts altogether (no one else can interrupt us)
Atomic Instructions: Test-And-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to indicate available
  - Return the old value
- Hardware executes it atomically!

```c
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- When executing test-and-set on “flag”
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?
Using Test-And-Set

- Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it satisfy critical region requirements? (mutex, progress, bounded wait, performance?)
The problem with spinlocks is that they are wasteful
- Although still useful in some cases; let's discuss advantages and disadvantages

If a thread is spinning on a lock, then the scheduler thinks that this thread needs CPU and puts it on the ready queue

If N threads are contending for the lock, the thread which holds the lock gets only $1/N$ th of the CPU
Another solution: Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```c
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?
On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer).
- In a “real” system, this is only available to the kernel.
  - Why?

- Disabling interrupts is insufficient on a multiprocessor.
  - Back to atomic instructions.
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives.
  - Don’t want interrupts disabled between acquire and release.
Goal: Use mutual exclusion to protect critical sections of code that access shared resources

Method: Use locks (spinlocks or disable interrupts)

Problem: Critical sections can be long

Spinlocks:
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted
- Memory consistency model causes problems (out of scope of this class)

Disabling Interrupts:
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Higher-Level Synchronization

- Spinlocks and disabling interrupts are useful for short and simple critical sections
  - Can be wasteful otherwise
  - These primitives are “primitive” – don’t do anything besides mutual exclusion

- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section

- All synchronization requires atomicity

- So we’ll use our “atomic” locks as primitives to implement them
Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion
- Locks work, but they have some drawbacks when critical sections are long
  - Spinlocks – inefficient
  - Disabling interrupts – can miss or delay important events

- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
Implementing a Blocking Lock

- Can use a spinlock instead of disabling interrupts

```
struct lock {
    int held = 0;
    queue Q;
}
void acquire (lock) {
    spinlock->acquire();
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    spinlock->release();
}

void release (lock) {
    spinlock->acquire();
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    spinlock->release();
}
```

**acquire(lock)**

- **spinning**
- **Running or Blocked**

**release(lock)**

- **spinning**
Semaphores

- Semaphores are an **abstract data type** that provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within critical section
  - Described by Dijkstra in THE system in 1968

- Semaphores are integers that support two operations:
  - **wait** *(semaphore)*: decrement, block until semaphore is open
    » **P()**, after the Dutch word for test, or **down()**
  - **signal** *(semaphore)*: increment, allow another thread to enter
    » **V()** after the Dutch word for increment, or **up()**
  - That's it! No other operations – not even just reading its value – exist

- Semaphore safety property: the semaphore value is always greater than or equal to 0
Blocking in Semaphores

- Associated with each semaphore is a queue of waiting threads/processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
Semaphores come in two types

- **Mutex semaphore (or binary semaphore)**
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section

- **Counting semaphore (or general semaphore)**
  - Multiple threads pass the semaphore determined by count
    - mutex has count = 1, counting has count = N
  - Represents a resource with many units available
  - or a resource allowing some unsynchronized concurrent access (e.g., reading)
Using Semaphores

- Use is similar to our locks, but semantics are different

```c
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

It is undefined which thread runs after a signal
Using Semaphores

- We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account

- We’re going to use semaphores to look at more interesting examples
  - Counting critical region
  - Ordering threads
  - Readers/Writers
  - Producer consumer with bounded buffers
  - More general examples
Beyond Mutual Exclusion

- We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account

- We’re going to use semaphores to look at more interesting examples
  - Counting critical region
  - Ordering threads
  - Readers/Writers
  - Producer consumer with bounded buffers
  - More general examples
Example Problem(s)

- Create a critical region where up to three threads (but no more) may enter at a time
  - Exploits the counting feature of semaphores

- Order operations across two threads; thread A executes first, then thread B executes
  - Exploits the ability to initialize semaphores to different values
Readers/Writers Problem

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers but only one writer
    » Let \( r \) be the number of readers, \( w \) be the number of writers
    » Safety: \( (r \geq 0) \land (0 \leq w \leq 1) \land ((r > 0) \Rightarrow (w = 0)) \)

- Use three variables
  - int \texttt{readcount} – number of threads reading object
  - Semaphore \texttt{mutex} – control access to \texttt{readcount}
  - Semaphore \texttt{w_or_r} – exclusive writing or reading
Readers/Writers

// number of readers
int readcount = 0;

// mutual exclusion to readcount
Semaphore mutex = 1;

// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}

reader {
    wait(mutex);     // lock readcount
    readcount += 1;  // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);     // lock readcount
    readcount -= 1;  // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount
}
**Readers/Writers Notes**

- `w_or_r` provides mutex between readers and writers
  - Readers wait/signal when `readcount` goes from 0 to 1 or 1 to 0
- If a writer is writing, where will readers be waiting?
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
  - Is it guaranteed that all readers will fall through?
- If readers and writers are waiting, and a writer exits, who goes first?
- Why do readers use `mutex`?
- What if the signal is above “if (readcount == 1)”?
- If read in progress when writer arrives, when can writer get access?