LECTURE 1.6

SYNCHRONIZATION

COP4600

Dr. Matthew Gerber

2/10/2015

Review: Producer/Consumer

- Two threads that each use a shared, bounded buffer
- The producer produces data items for the consumer to consume
- The producer needs to avoid overrunning the buffer
  - Not hard; just use shared variables
- The consumer needs to avoid trying to consume items that aren't ready yet
  - Also not hard; see above
- Both need to avoid messing up each others' updating of the aforementioned shared variables
- *That's* the hard part
- Again, this problem has a general name

Critical Sections (5.2)

- A *critical section* is:
  - A *segment of code*…
  - In a *cooperating process*…
  - That *modifies shared information* in a way that *either*…
    - …can *destructively interfere* with other critical sections, *or*
    - …can *be destructively interfered with by* other critical sections
- It is considered (and really only makes sense) in terms of a *given* set of inter-process interactions
- We have three fundamental ways to deal with critical sections
- The first way is really, really simple…

Producer-Consumer with Disabling Interrupts

```
constant BUFSIZE = 8
shared data buffer[BUFSIZE]
shared int in = 0
shared int out = 0
```

```
producer_thread {
  data d
  while true {
    while ((in+1)%BUFSIZE) == out {
      wait
    } // while in + 1 == out

    disable_interrupts()
    d = produce_item()
    buffer[out] = d
    in = in + 1
    in = in % BUFSIZE
    enable_interrupts()
  } // while true
}
```

```
consumer_thread {
  data d
  while true {
    while in == out {
      wait
    } // while in == out

    disable_interrupts()
    data d = buffer[out]
    consume_item(d)
    out = out + 1
    out = out % BUFSIZE
    enable_interrupts()
  } // while true
}
```

Producer-Consumer with Disabling Interrupts

```
constant BUFSIZE = 8
shared data buffer[BUFSIZE]
shared int in = 0
shared int out = 0

p
```

Now with 100% less undefined behavior, but…

```
    disable_interrupts()
    d = produce_item()
    buffer[out] = d
    in = in + 1
    in = in % BUFSIZE
    enable_interrupts()
  } // while true
}
```

```
    disable_interrupts()
    data d = buffer[out]
    consume_item(d)
    out = out + 1
    out = out % BUFSIZE
    enable_interrupts()
  } // while true
}
```

The Problems with Disabling Interrupts

- Performance
  - If the timer is controlled by interrupts, it's going to mess them up
- Potential unpredictable consequences
  - Disabling fundamental bits of the operating system any more often than absolutely necessary is not a good idea
- Absolute non-starter on multiprocessor systems
  - Consider what it takes to disable interrupts on those…
- So we need a different way to make this happen

The Mutex

- A primitive function provided by the operating system
- The following functions are implemented by the operating system…

```
signal (mutex m) {
  m = true
}
```

```
wait (mutex m) {
  while m == false {
    wait
  }
  m = false
}
```

- …in such a way that `wait` *cannot be interrupted* between *testing* that `m` is true and *setting* it to false (more on this in a bit…)
- As with anything else, a mutex can start in either position

```
mutex m = true
```

Producer-Consumer with a Mutex

```
constant BUFSIZE = 8
shared data buffer[BUFSIZE]
shared int in = 0
shared int out = 0
mutex m = true
```

```
producer_thread {
  data d
  while true {
    while ((in+1)%BUFSIZE) == out {
      wait
    } // while in + 1 == out

    wait(m)
    d = produce_item()
    buffer[out] = d
    in = in + 1
    in = in % BUFSIZE
    signal(m)
  } // while true
}
```

```
consumer_thread {
  data d
  while true {
    while in == out {
      wait
    } // while in == out

    wait(m)
    data d = buffer[out]
    consume_item(d)
    out = out + 1
    out = out % BUFSIZE
    signal(m)
  } // while true
}
```

Producer-Consumer with a Mutex

```
constant BUFSIZE = 8
shared data buffer[BUFSIZE]
shared int in = 0
```

Now with 100% less undefined behavior *and* system instability

```
    } // while in + 1 == out

    wait(m)
    d = produce_item()
    buffer[out] = d
    in = in + 1
    in = in % BUFSIZE
    signal(m)
  } // while true
}
```

```
    } // while in == out

    wait(m)
    data d = buffer[out]
    consume_item(d)
    out = out + 1
    out = out % BUFSIZE
    signal(m)
  } // while true
}
```

Implementing Mutexes (5.5)

- Remember that a mutex basically provides the following functions:

```
signal (mutex m) {
  m = true
}
```

```
wait (mutex m) {
  while m == false {
    wait
  }
  m = false
}
```

- The only thing that needs to *not be interrupted* is the period in `wait` between determining that `m` is true and setting it to false
- There are a few ways to do this; let’s start with the obvious one

Mutex Locks with Disabling Interrupts

```
signal (mutex m) {
  m = true
}
```

```
wait (mutex m) {
  disable interrupts
  while m == false {
    enable interrupts
    yield
    disable interrupts
  }
  m = false
  enable interrupts
}
```

- `wait` is never interrupted between determining that `m` is true and setting it to false
- In one version of this, `wait` yields; in another version, `wait` just keeps checking whenever it can get the CPU
- Which version to use depends on how long you expect this all to take
- This method – especially the version where you don't yield—is called a *spin wait*
- This all has at least *some* of the problems of disabling interrupts in general

The Test-And-Set instruction (5.4)

- Better way: Get help from the hardware
- The **processor** implements the following instruction **atomically**

```
boolean test_and_set (boolean &target) {
  boolean rv = target
  target = true

  return rv
}
```

- This defines a sort of minimum possible critical section in hardware
- If we call `test_and_set` on a true value, it stays true and we get back true
- If we call `test_and_set` on a false value, it becomes true and we get back false
- It's up to the hardware to make sure no other processors interfere

Mutex Locks with Test-And-Set

```
signal (mutex m) {
  m = false
}
```

```
wait (mutex m) {
  while test_and_set(m) {
    yield
  }
}
```

- Note that the logic on this is inverted
- The mutex is free if `m` is *false*; releasing it sets it false, and acquiring its lock sets it true
- When `wait` calls `test_and_set`, it gets back false if `m` *was* false and true if `m` was true; **`m` is true afterward either way**
- The good news is we're no longer disabling interrupts
- The bad news is we're still yielding or spin-waiting

Mutex Locks: Avoiding Spin-Waits (5.6.2)

- The easy way to solve this is by turning mutexes into queues
- When a process waits on a mutex that isn't available, it's inserted into a queue for that mutex
- When a process signals a mutex that has any processes waiting for it, one is dequeued and given the lock
- The queuing is a critical section in and of itself…
  - …but we can guard *it* with a spin-wait mutex without any trouble, since it's a very small O(1) problem

Counting Semaphores (5.6)

- Just like mutex locks, except integer instead of boolean
- Signaling increments, waiting decrements; processes are blocked when they wait on zero
- Logic is more complex
  - Requires multiple levels of guarding…
  - …just like the queued mutex we just discussed
- By the way…
  - How easily could producer-consumer be implemented using counting semaphores?
  - (Hint: Pretty easily)

LOOSE ENDS 1: DEADLOCKS

Deadlocks (5.6.3)

```
shared mutex m1, m2

define p1 {                        define p2 {
  wait(m1)                           wait(m2)
  wait(m2)                           wait(m1)
  …                                 …
```

- What's going to happen here?

Deadlocks (5.6.3)

```
shared mutex m1, m2
```

```
define p1 {
  wait(m1)
  wait(m2)
  …
```

```
define p2 {
  wait(m2)
  wait(m1)
  …
```

- What's going to happen here?

BAD THINGS

- A set of processes is *deadlocked* if every process in a set is waiting on an event that can only be caused by another process in the same set
- We'll cover deadlocks in our last lecture of the unit

LOOSE ENDS 2: CASE APPROACHES

Windows Synchronization (5.9.1)

- Uses simple spinlocks and mutex locks in the kernel
- Adapts spinlocks to interrupt-disabling for single-processor systems
- For thread synchronization, Windows provides *dispatcher objects* with:
  - Mutex locks
  - Semaphores
  - Timers
  - Conditions (called Events)
- Lock-guarded shared memory is provided
- *Critical Section Objects* provide mutexes at the user level that only allocate kernel mutexes if they have to

Solaris Synchronization (5.9.3)

- Solaris provides semaphores and conditions as we’ve already described
- *Adaptive mutexes* shift themselves from spinlocks to yielding if the kernel determines they’re going to have to wait any real length of time for the lock
- *Turnstiles* are Solaris’ method of queuing for adaptive mutexes
  - Turnstiles are dynamically allocated queues that implement priority inheritance
- Every synchronization mechanism available to the kernel is available to the user
  - …except priority inheritance

LOOSE ENDS 3: BEYOND SEMAPHORES

Other Issues with Semaphores

- Mutexes and semaphores are useful, but they’re low-level constructs
- Think of them as like malloc() and free() – they have all the same problems that can cause resource “leaks”
- So can we come up with the equivalent of garbage-collected objects?
- As a matter of fact…

NEXT TIME:
MONITORS, PIPES AND SOCKETS