# LECTURE 1.5
## SCHEDULING CASE STUDIES AND INTRODUCTION TO IPC

COP4600

Dr. Matthew Gerber

2/8/2016

# SCHEDULING CASE STUDIES

# Windows Scheduling (6.7.2)

- Explicit multilevel queue with internal priorities mapped to linear numeric values that in turn provide an implicit multilevel queue
  - It's probably better to just show what we mean; we'll do that next
- In general, higher-priority threads immediately preempt lower-priority threads
- In particular, (soft) real-time threads *always* preempt other threads

# Windows Scheduling (6.7.2)

Priority Classes

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- IDLE_PRIORITY_CLASS

Relative Thread Priorities

- TIME_CRITICAL
- HIGHEST
- ABOVE_NORMAL
- NORMAL
- BELOW_NORMAL
- LOWEST
- IDLE

# Windows Scheduling (6.7.2)

| | REAL TIME | HIGH | ABOVE_ NORMAL | NORMAL | BELOW_ NORMAL | IDLE |
|---|---|---|---|---|---|---|
| TIME_ CRITICAL | 31 | 15 | 15 | 15 | 15 | 15 |
| HIGHEST | 26 | 15 | 12 | 10 | 8 | 6 |
| ABOVE_ NORMAL | 25 | 14 | 11 | 9 | 7 | 5 |
| NORMAL | 24 | 13 | 10 | 8 | 6 | 4 |
| BELOW_ NORMAL | 23 | 12 | 9 | 7 | 5 | 3 |
| LOWEST | 22 | 11 | 8 | 6 | 4 | 2 |
| IDLE | 16 | 1 | 1 | 1 | 1 | 1 |

# Windows Scheduling (6.7.2)

- When a thread returns from an I/O wait, its priority is raised
  - High amount for user interaction
  - Lower amount for disk I/O
- When a thread uses its entire quantum, its priority is lowered
  - Never lowered below base priority
- The process owning the foreground window gets a priority bump
  - Usually about 3

# Linux Scheduling: O(1)

- Linux scheduler for kernel version 2.6 up until 2.6.23
- The bad: Heuristics were pretty bad at figuring out which processes were interactive
- The good: An O(1) scheduler!
- Any priority queue “should” require O(log *n*), at least, and the previous Linux scheduler was O(*n*)
- So this scheduler implements priority with a guaranteed constant upper bound of execution time
  - How?
- A constant number of priority levels, *and a queue for each of them*

# Linux Scheduling: O(1)

- 140 priority levels
- Each processor is given two arrays of 140 doubly linked lists: the *active* and *expired* array
  - When a process's quantum expires it's re-inserted into the corresponding queue in the expired array
    - Its priority can change at this point
  - When the active array runs out of processes, the scheduler *switches arrays*
- When a process needs to be chosen, an active process with best priority is chosen
- Guaranteed to be doable in 140 compares or less (not counting dereferencing)…
  - So this is technically O(1)
  - Very, very, very technically

# Linux Scheduling: CFS (6.7.1)

- The Completely Fair Scheduler
- Implements the *weighted fair queuing* concept
  - Given equal priorities, processes should receive as close to the same amount of time from the processor as possible
- Each process is given a *virtual runtime*
- The scheduler selects whichever process currently has the *lowest* virtual runtime
- For processes with completely normal priority, virtual runtime is equal to physical runtime

- Virtual runtime is affected by multipliers
  - Priority is a direct multiplier
  - Long-running processes get de-prioritized
  - CPU-bound processes get de-prioritized
  - I/O bound processes get prioritized
- This *is* all done preemptively
- Guards are put in place (maximum time a process can wait to run, minimum time a process can run) to prevent silly results

# Linux Scheduling: CFS (6.7.1)

- The current processes and their runtimes are held in a red-black tree
  - That means it takes O(*log* n) to insert a process
  - That in turn means that CFS could start getting significantly slower than O(1)…
  - …if we reached a little less than 1.4 tredecillion processes
    - That's 1,400,000,000,000,000,000,000,000,000,000,000,000,000,000
    - We're not worried

# Solaris Scheduling (6.7.3)

- Solaris traditionally has four classes of thread:
  - Real-Time
  - System
  - Time-Sharing
  - Interactive
- Solaris 9 adds two more:
  - Fair Share
  - Fixed Priority
- *All of these use different scheduling methods*
- The scheduler maps each class-specific output to a global priority
- Real-Time threads always have priority over everything else
- System threads have priority over everything except Real-Time threads

# Solaris Scheduling (6.7.3)

- Interactive/Time-Sharing schedules are similar, and are the most interesting for our purposes
- Solaris uses a dynamic priority method somewhat similar to Windows'
  - (Actually, it's the other way around)
- Threads that return from I/O have their priority increased (in Solaris' case, radically)
- Threads that use up their time quanta have their priority decreased (more gradually)
- Threads with lower priority get longer time quanta

# INTER-PROCESS COMMUNICATION

# Inter-Process Communication (3.5)

- Two fundamental mechanisms available:
  - Shared Memory
  - Message Passing
- Both of these are useful, and many operating systems provide both
- Message-passing avoids conflicts and works better across networks
- Shared memory is more convenient for large amounts of data
- But we'll get back to this…

# The Producer-Consumer Problem

- Classic example of inter-process sharing
- A *producer* in one process creates chunks of data for a *consumer* to further operate on
- The producer needs to be able to place those chunks of data in a shared memory location that the consumer can read them from
- The producer needs to be delayed as appropriate to not overrun this buffer
- The consumer needs to be delayed as appropriate to wait for the producer to generate data for it to operate on
- So how do we make this happen?

# The Producer-Consumer Problem

- Classic example of inter-process sharing
- A *producer* in one process creates chunks of data for a *consumer* to further operate on
- The producer needs to be able to place those chunks of data in a shared memory location that the consumer can read them from
- The producer needs to be delayed as appropriate to not overrun this buffer
- The consumer needs to be delayed as appropriate to wait for the producer to generate data for it to operate on
- So how do we make this happen?

# Producer-Consumer Code

```
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

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

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

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

# Producer-Consumer Code

**Do you see the problem?**

```
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

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

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

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

# Producer-Consumer Code

**Here's a hint…**

```
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

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

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

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

# Producer-Consumer Code

**What happens if we context-switch inside one of these sections?**

```
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

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

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

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

# Producer-Consumer Code

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

p
```

**BAD THINGS**

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

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

```
    wait
  } // while in == out

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

# The Critical Section Problem

- This kind of section of code actually has a name
- It’s called a *critical section*
- The hard part of synchronization is keeping critical sections from interfering with each other

# NEXT TIME: SYNCHRONIZATION