Computer Systems III: Multitasking and Concurrency

Noah Singer, George Klees

Montgomery Blair High School Computer Team

October 5, 2017

Overview

Section 1: Multitasking

Processes and threads

- **Process**: A high-level task; an instance of a running program, with an address space and access to resources
- **Threads**: A low-level task; code running on the CPU, consisting of the actual execution state (CPU registers, stack, etc.)
- One or more threads per process
- **Multitasking**: CPU switches back and forth rapidly between threads

Program execution

1. Create blank address space
2. Load program from disk into address space
3. Create initial thread with stack
4. Begin executing

Scheduling

- **Scheduler** must select which threads to run and for how long
  - **Cooperative scheduling**: threads can run as long as they want until they yield
  - **Preemptive scheduling**: threads get a fixed amount of time (**quantum**)
- Threads are generally non-malicious; when any thread has no work to do, it often just yields to allow other threads a turn
- Threads often voluntarily yield when waiting for an operation to complete (**blocking**)
  - Most threads spend most of their time waiting for locks and I/O, so cooperative scheduling is common

Prioritization

- Some threads are more important than others
- **High-priority** threads must complete immediately
  - Examples: GUI, some drivers
  - Often simply receive, route, and transmit requests, sleeping almost immediately after awaking
- **Low-priority** threads can be scheduled whenever there is no more important work
  - Examples: Background processes, virus scanners, updates

Task switching

- When a thread exceeds its quantum, it is **preempted** and loses control
  - Triggered by a hardware timer
- Scheduler holds a queue of threads waiting to run
- When a yield or preemption occurs, the scheduler must switch **CPU contexts** (stack and registers)
  1. Save old CPU context
  2. Switch address spaces, etc. to new process
  3. Restore new CPU context

Section 2: Concurrency

Multiple CPUs

- Scheduler also selects which CPU to schedule a thread on
  - CPU load, power management status, temperature, etc.
  - Due to caches and **affinity**, threads are better scheduled on CPUs on which they have already/recently executed
- **Concurrency** issues occur when threads on two CPUs access the same resource simultaneously
  - Or one thread is working, gets preempted, and then another thread accesses the same resource
  - The kernel is one such resource

Locks

- **Spinlocks** simply record whether or not they are locked
  - Order in which threads attempt to acquire lock is irrelevant
  - Threads "spin" (in a while loop) while waiting for lock
- **Granularity**: the amount of code that is protected by locks
  - Too fine: doesn't fix the original issues
  - Too coarse: doesn't scale and difficult to develop/maintain

Atomic operations

- One thread can be interrupted *while* acquiring the spinlock
- Uninterrutible operations; another CPU cannot interfere
- Thread cannot be preempted in the process of accessing a resource
- In multi-step atomic operations, memory bus is locked, slowing down the rest of the system
- **Compare and exchange** compares the value of `var` and `old`, setting `var` to `new` if it is equal to `old` and returning the original value of `var`

Issues with spinlocks

- Threads cannot yield while waiting for spinlocks
- **Deadlock**: two threads both spinning to acquire resources the other thread has
- **Starvation**: a thread is continually outcompeted for locks, so it cannot access necessary resources
- Some more sophisticated locks solve these issues
  - **Ticket locks** á la deli counter or DMV