Lecture 5: Scheduling

CSE 120: Principles of Operating Systems

UC San Diego: Summer Session I, 2009
Frank Uyeda
Announcements

• Homework 2 is due Monday.
• Project 1:
  – Milestone (*tonight*) Friday 7/10
  – Deadline Monday 7/13

• Midterm Exam
  – Friday 7/17 (*one week from today*)
  – Covers everything through Monday’s lecture
What does the value of a semaphore denote?

- Number of waiting processes
- Number of available resources
- Priority of waiting process
- Number of waiting threads

Requires insight about what the value of a semaphore “means”, not just how a semaphore behaves.
PeerWise

- A situation in which processes wait indefinitely within a semaphore is called:
  - Priority Inversion
  - Starvation
  - Busy Waiting
  - Bounded Waiting
  - Mutual Exclusion

- Again, good wrong answers
  - Several terms that we’ve heard before + a few new ones.
Homework

• Due in class on Mondays
• Late policy: I will accept homework until the solutions are posted online.
  – If your assignment is not in my hands when the solution gets posted, I will not accept it.
Homework (cont)

• Question 2 was ambiguously worded.
  – Gave points if you only listed the operations that should be privileged.
  – Gave points if you justified your answer well.

• 5 points/question.
  – Question 5 was worth 1 extra point.

• Memcpy - “Protecting memory state”
  – Not privileged.
  – Memory state = meta-data concerning memory
    • TLB, virtual memory manager
  – Normal reads/writes don’t need to be privileged.
  – Bad reads/writes cause a fault that is handled by the OS.
Homework (cont)

• What you want to hear about:
  – Security (8)
  – Future of OS’s (2)
  – Real-time systems (2)
  – Scheduling Algorithms (2)
  – How to build your own OS
  – Multithreading & performance
  – Influential OS’s (Linux, Windows)
  – File Systems
  – Interaction between OS and hardware
  – Virtual memory
Review

• Locks
  – Spin-locks are inefficient
  – Disabling interrupts only for short critical sections

• Higher-level Synchronization Primitives
  – Blocked waiters using queue
  – Interrupts can be enabled in critical section
Review (2)

• **Semaphores**
  – `wait()`/`signal()` implement blocking mutual exclusion
  – Also used as atomic counters (counting semaphores)

• **Monitors**
  – Define module of encapsulated data and procedures
  – Synchronizes execution within procedures
    • Only one thread can execute within a monitor at a time
  – Relies upon high-level language support

• **Condition variables**
  – Used by threads as a synchronization point to wait for events
  – Inside monitors, or outside with locks
Synchronization Case Study: Therac-25

- Radiation therapy machine (1976)
  - Completely computer controlled by 1982
  - Therac-6 and Therac-20 were predecessors
    - “25” stood for 25 MeV that machine can disperse
- Machine has 2 modes of operation
  - Electron and photon (X-Ray) treatments
  - Software written for PDP-11 computer

Note: Images and information from [gallagher]
Case Study: Therac-25 (2)
Case Study: Therac-25 (3)

• Accidents
  – 6 accidents between 1985 and 1987
    • 3 fatal
  – Cryptic error messages to technicians
    • “Dose not delivered” on error -> overdoses
      – e.g., 13,000-17,000 rads delivered instead of 200-1000 rads
    • Mode switches occurred on accident
      – Protective plate (beam spreader) not deployed either

• What happened?
  – Race conditions
Race Condition 1

- The race occurred when an operator edited the dosage on the console too quickly
- The machine had two basic software tasks
  - Treatment task: checks to see if a dosage entry has been entered on the terminal
  - Configuration task: checks to see if machinery aligned correctly (e.g., the dispersion plate, dispersion modes)
- Race condition:
  - Operator “enters” dosage
    - Treatment task asks Configuration task to set the hardware
  - Operator edits dosage while the hardware is being configured
    - Configuration task signals “ready” to Treatment task without edits being accepted
  - Race condition occurs when “machine ready” display is in response to the first dosage, not the edits

Note: Information from [mdwelsh, harvard]
“Race” Condition 2

• Software task used to move aperture that controls intensity and direction of beam
• Operator must press “set” button to configure turntable once prescription entered
• “Set up test” task runs periodically to check position of turntable
  – Increment a variable “Class3” on each iteration
  – Otherwise, a series of interlock checks are performed to ensure aperture incorrect position
    • These checks will set Class3 to 0 when they are complete
    – If “Class3 == 0” everything is ready and the dosage can begin
• Integer overflow will cause Class3 to “start” at 0.
  – 8-bit variable -> 256 iterations can cause overflow
Case Study: Therac-25 (4)

• Why weren’t these errors caught?
  – Race Condition 1
    • Hardware interlocks masked previous (reused) code
      – Previously, the physical hardware would not allow x-ray mode to occur without beam spreader (plate)
    • Errors occurred only on strange (including erroneous keystrokes)
      – Not caught during testing
  – Race Condition 2
    • Only reproducible after 256 iterations, and the right sequence of keystrokes on that wrap-around

• Unfortunately even 1 error isn’t acceptable
Goals Today

• Describe where the Operating System must make scheduling decisions
• Describe possible goals of a scheduling policy
• Learn basic scheduling policies an OS may use
• Understand the basic limitations of each policy
Scheduling Decisions

- Processes, Threads, Scheduling Queues, WaitQueues

...
Scheduling Overview

• In discussing process management and synchronization, we talked about context switching among processes/threads on the ready queue
• But we have glossed over the details of exactly which thread is chosen from the ready queue
• Making this decision is called scheduling
• In this lecture, we’ll look at:
  – The goals of scheduling
  – Starvation
  – Various well-known scheduling algorithms
  – Standard Unix scheduling algorithms
Multiprogramming

- In a multiprogramming system, we try to increase CPU utilization and job throughput by overlapping I/O and CPU activities
  - Doing this requires a combination of mechanisms and policy
- We have covered the mechanisms
  - Context switching, how and when it happens
  - Scheduling queues and execution states
- Now we’ll look at the policies
  - Which process (thread) to run, for how long, etc.
- We’ll refer to schedulable entities as jobs (standard usage) – could be processes, threads, people, etc.
Scheduling Horizons

• Scheduling works at two levels in an operating system
  – To determine the **multiprogramming level** – the number of jobs loaded into primary memory
    • Moving jobs to/from memory is often called swapping
  – To decide what job to run next to guarantee “good service”
    • Good service could be one of many different criteria

• These decisions are known as long-term and short-term scheduling decisions, respectively
  – **Long-term** scheduling happens relatively infrequently
    • Significant overhead in swapping a process out to disk
  – **Short-term** scheduling happens relatively frequently
    • Want to minimize the overhead of scheduling
Scheduling Goals

• Scheduling algorithms can have many different goals:
  – CPU utilization
  – I/O utilization
  – Job throughput (# jobs/unit time)
  – Turnaround time ($T_{\text{finish}} - T_{\text{arrival}}$)
  – Waiting time ($\text{Avg}(T_{\text{wait}})$: avg time spent on wait queues)
  – Response time ($\text{Avg}(T_{\text{ready}})$: avg time spent on ready queue)

• Batch systems
  – Strive for job throughput, turnaround time (supercomputers)

• Interactive systems
  – Strive to minimize response time for interactive jobs (PC)
Starvation

- **Starvation** occurs when a job cannot make progress because some other job has the resource it requires
  - We’ve seen locks, Monitors, Semaphores, etc.
  - The same thing can happen with the CPU!
- Starvation can be a side effect of synchronization
  - Constant supply of readers always blocks out writers
  - Well-written critical sections should ensure bounded waiting
- Starvation usually a side effect of the scheduling algorithm
  - A high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock
OS Scheduler

- The **scheduler** (aka dispatcher) is the module that manipulates the queues, moving jobs to and from states
- The **scheduling algorithm** determines which jobs are chosen to run next and what queues they wait on
- In general, the scheduler runs:
  - When a job switches states (running, waiting, etc.)
  - When an interrupt occurs (e.g., timer, preemption)
  - When a job is created or terminated
- We’ll discuss scheduling algorithms in two contexts
  - A **preemptive** scheduler can interrupt a running job
  - A **non-preemptive** scheduler waits for running job to block
FCFS/FIFO Algorithms

• First-come first-served (FCFS), first-in first-out (FIFO)
  – Jobs are scheduled in order of arrival to ready queue
  – “Real-world” scheduling of people in lines (e.g., supermarket)
  – Typically non-preemptive (no context switching at market)
  – Jobs treated equally, no starvation

• Problem
  – Average waiting time can be large if small jobs wait behind long ones (high turnaround time)
    • You have a basket, but you’re stuck behind someone with a cart
    • This is called “Head-of-line (HoL) Blocking”
Shortest Job First (SJF)

• Shortest Job First (SJF)
  – Choose the job with the smallest expected CPU burst
    • Person with smallest number of items to buy
  – Provably optimal minimum average waiting time

• Problem
  – Usually impossible to know size of CPU burst
    • Like choosing person in line without looking inside basket/cart
    • How can you make a reasonable guess?
    • Can potentially starve
  – Flavors
    • Can be either preemptive or non-preemptive
    • Preemptive SJF is called shortest remaining time first (SRTF)
Round Robin (RR)

• Round Robin
  – Excellent for timesharing
  – Ready queue is treated as a circular queue (FIFO)
  – Each job is given a time slice called a quantum
    • A job executes for the duration of the quantum, or until it blocks or is interrupted
    • If it blocks, it is placed on the appropriate queue, otherwise, it is put back on the ready queue (almost like a thread_yield())
  – Preemptive
  – No starvation

• Problem
  – Context switches are frequent and need to be very fast
  – How to set the time quantum?
Example

- FCFS
- SJF
- SRTF (quantum = 2)
- Round Robin (quantum = 2)

P1: time = 10
P2: time = 1
P3: time = 5
P4: time = 3
Priority Scheduling

• Priority Scheduling
  – Choose next job based on priority
    • Airline check-in for first class passengers
  – Can implement SJF, priority = 1/(expected CPU burst)
  – Also can be either preemptive or non-preemptive

• Problem
  – Starvation – low priority jobs can wait indefinitely

• Solution
  – “Age” processes
    • Increase priority as a function of waiting time
    • Decrease priority as a function of CPU consumption
Lottery Scheduling

• A randomized priority scheme
  – Each job is given some number of lottery tickets
    • Number of tickets is proportional to their priority (in this case, higher priority means more tickets)
  – On each scheduling interval, a ticket is chosen
    • Job holding the winning ticket is scheduled
• How does this avoid starvation?
  – Even low priority jobs have a chance to run
Combining Algorithms

• Scheduling algorithms can be combined
  – Have multiple queues
  – Use a different algorithm for each queue
  – Move processes among queues

• Example: Multiple Level Feedback Queues (MLFQ)
  – Multiple queues representing different job types
    • Interactive, CPU-bound, batch, system, etc.
  – Queues have priorities, jobs on same queue scheduled RR
  – Jobs can move among queues based upon execution history
    • Feedback: Switch from interactive to CPU-bound behavior
Unix Scheduler

• The canonical Unix scheduler uses a MLFQ
  – 3-4 classes spanning ~170 priority levels
    • Timesharing: first 60 priorities
    • System: next 40 priorities
    • Real-time: next 60 priorities
    • Interrupt: next 10 (Solaris)

• Priority scheduling across queues, RR within a queue
  – The process with the highest priority always runs
  – Processes with the same priority are scheduled RR

• Processes dynamically change priority
  – Increases over time if process blocks before end of quantum
  – Decreases over time if process uses entire quantum
Motivation of Unix Scheduler

• The idea behind the Unix scheduler is to reward interactive processes over CPU hogs
• Interactive processes (shell, editor, etc.) typically run using short CPU bursts
  – They do not finish quantum before waiting for more input
• Want to minimize response time
  – Time from keystroke (putting process on ready queue) to executing keystroke handler (process running)
  – Don’t want editor to wait until CPU hog finishes quantum
• This policy delays execution of CPU-bound jobs
  – But that’s ok
Scheduling Summary

- Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen
- Scheduling algorithm determines which process runs, where processes are placed on queues
- Many potential goals of scheduling algorithms
  - Utilization, throughput, wait time, response time, etc.
- Various algorithms to meet these goals
  - FCFS/FIFO, SJF, Priority, RR
- Can combine algorithms
  - Multiple-level feedback queues
  - Unix example
Next Time

• Read Chapter 7 (Deadlocks)
• Homework 2 due on Monday.
• Check Web site for course announcements
  – http://www.cs.ucsd.edu/classes/su09/cse120