Lecture 8: Scheduling and Deadlock
Geoffrey M. Voelker
• Friday
  ♦ Project #1 due at 11:59pm
• Saturday
  ♦ Homework #2 due at 11:59pm
  ♦ Will post solutions Sunday morning
• Monday
  ♦ Q&A review session 3pm (more details on Piazza)
• Tuesday
  ♦ Midterm
Edward Wu (UCSD Alumnus)
Director & Technical Lead
Niantic

Tuesday, May 1 @ 9:30am
CSE 1242
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:
  ♦ Goals of scheduling
  ♦ Starvation
  ♦ Various well-known scheduling algorithms
  ♦ Standard Unix scheduling algorithm
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
  ♦ Process queues and process 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 Goals

- Scheduling works at two levels in an operating system
  - To determine the multiprogramming level, the number of jobs loaded into memory
    » Moving jobs to/from memory is 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
      - Fast context switches, fast queue manipulation
Scheduling

• The scheduler (aka dispatcher) is the module that manipulates the queues, moving jobs to and fro

• 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 from running to waiting
  ♦ When an interrupt occurs (e.g., I/O completes)
  ♦ When a job is created or terminated

• We’ll discuss scheduling algorithms in two contexts
  ♦ In preemptive systems the scheduler can interrupt a running job (involuntary context switch)
  ♦ In non-preemptive systems, the scheduler waits for a running job to explicitly block (voluntary context switch)
Scheduling Goals

- Scheduling algorithms can have many different goals:
  - CPU utilization (%CPU)
  - Job throughput (# jobs/time)
  - Turnaround time ($T_{\text{finish}} - T_{\text{start}}$)
  - 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 is a scheduling “non-goal”:

- **Starvation** is a situation where a process is prevented from making progress because some other process has the resource it requires
  - Resource could be the CPU, or a lock (recall readers/writers)
- **Starvation usually a side effect of the sched. 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
- **Starvation can be a side effect of synchronization**
  - Constant supply of readers always blocks out writers
FCFS/FIFO

• First-come first-served (FCFS), first-in first-out (FIFO)
  ♦ Jobs are scheduled in order of arrival to ready Q
  ♦ “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
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 (AWT)

AWT = \((8 + (8+4)+(8+4+2))/3 = 11.33\)

AWT = \((4 + (4+8)+(4+8+2))/3 = 10\)

AWT = \((4+ (4+2)+(4+2+8))/3 = 8\)

AWT = \((2 + (2+4)+(2+4+8))/3 = 7.33\)
Shortest Job First (SJF)

- Problems
  - 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)
Priority Scheduling

• Priority Scheduling
  ♦ Choose next job based on priority
    » Airline checkin 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
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
  - No starvation
  - Can be preemptive or non-preemptive

- Problem
  - Context switches are frequent and need to be very fast
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 Overhead

- Operating systems aim to minimize overhead
  - Context switching takes non-zero time, so it is pure overhead
  - Overhead includes context switch + choosing next process
- Modern time-sharing OSes (Unix, Windows, …) time-slice processes in ready list
  - A process runs for its quantum, OS context switches to another, next process runs, etc.
  - A CPU-bound process will use its entire quantum (e.g., 10ms)
  - An IO-bound process will use part (e.g., 1ms), then issue IO
  - The IO-bound process goes on a wait queue, the OS switches to the next process to run, the IO-bound process goes back on the ready list when the IO completes
Utilization

• CPU utilization is the fraction of time the system is doing useful work (e.g., not context switching)

• If the system has
  ♦ Quantum of 10ms + context-switch overhead of 0.1ms
  ♦ 3 CPU-bound processes + round-robin scheduling

• In steady-state, time is spent as follows:
  ♦ 10ms + 0.1ms + 10ms + 0.1ms + 10ms + 0.1ms
  ♦ CPU utilization = time doing useful work / total time
  ♦ CPU utilization = (3*10ms) / (3*10ms + 3*0.1ms) = 30/30.3

• If one process is IO-bound, it will not use full quantum
  ♦ 10ms + 0.1ms + 10ms + 0.1ms + 1ms + 0.1ms
  ♦ CPU util = (2*10 + 1) / (2*10 + 1 + 3*0.1) = 21/21.3
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
Deadlock

- Synchronization is a live gun – we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - You have likely been intuitively avoiding this situation already
- More generally, processes that allocate multiple resources generate dependencies on those resources
  - Locks, semaphores, monitors, etc., just represent the resources that they protect
- If one process tries to allocate a resource that a second process holds, and vice-versa, they can never make progress
- We call this situation **deadlock**, and we’ll look at:
  - Definition and conditions necessary for deadlock
  - Representation of deadlock conditions
  - Approaches to dealing with deadlock
Deadlock Definition

- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized
- Definition:
  - Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.

```
Process 1
lockA->Acquire();
...
lockB->Acquire();

Process 2
lockB->Acquire();
...
lockA->Acquire();
```
Conditions for Deadlock

- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. **Mutual exclusion** – At least one resource must be held in a non-sharable mode
  2. **Hold and wait** – There must be one process holding one resource and waiting for another resource
  3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)
  4. **Circular wait** – There must exist a set of processes \([P_1, P_2, P_3, \ldots, P_n]\) such that \(P_1\) is waiting for \(P_2\), \(P_2\) for \(P_3\), etc.
Resource Allocation Graph

- Deadlock can be described using a resource allocation graph (RAG)
- The RAG consists of a set of vertices \( P = \{P_1, P_2, \ldots, P_n\} \) of processes and \( R = \{R_1, R_2, \ldots, R_m\} \) of resources
  - A directed edge from a process to a resource, \( P_i \rightarrow R_j \), means that \( P_i \) has requested \( R_j \)
  - A directed edge from a resource to a process, \( R_i \rightarrow P_j \), means that \( R_j \) has been allocated by \( P_i \)
  - Each resource has a fixed number of units
- If the graph has no cycles, deadlock \textbf{cannot exist}
- If the graph has a cycle, deadlock \textbf{may exist}
RAG Example

A cycle...and deadlock!

Same cycle...but no deadlock. Why?
A Simpler Case

• If all resources are single unit and all processes make single requests, then we can represent the resource state with a simpler waits-for graph (WFG)
  ♦ Useful for tracking locks

• The WFG consists of a set of vertices $P = \{P_1, P_2, \ldots, P_n\}$ of processes
  ♦ A directed edge $P_i \rightarrow P_j$ means that $P_i$ has requested a resource that $P_j$ currently holds

• If the graph has no cycles, deadlock cannot exist
• If the graph has a cycle, deadlock exists
Dealing With Deadlock

• There are four approaches for dealing with deadlock:
  ♦ Ignore it – how lucky do you feel?
  ♦ Prevention – make it impossible for deadlock to happen
  ♦ Avoidance – control allocation of resources
  ♦ Detection and Recovery – look for a cycle in dependencies
Deadlock Prevention

• Prevention – Ensure that at least one of the necessary conditions cannot happen
  ♦ Mutual exclusion
    » Make resources sharable (not generally practical)
  ♦ Hold and wait
    » Process cannot hold one resource when requesting another
    » Process requests, releases all needed resources at once
  ♦ Preemption
    » OS can preempt resource (costly)
  ♦ Circular wait
    » Impose an ordering (numbering) on the resources and request them in order (popular implementation technique)
Deadlock Avoidance

• Avoidance
  ♦ Provide information in advance about what resources will be needed by processes to guarantee that deadlock will not happen
  ♦ System only grants resource requests if it knows that the process can obtain all resources it needs in future requests
  ♦ Avoids circularities (wait dependencies)

• Tough
  ♦ Hard to determine all resources needed in advance
  ♦ Good theoretical problem, not as practical to use
### Banker’s Algorithm

- The Banker’s Algorithm is the classic approach to deadlock avoidance for resources with multiple units

1. Assign a **credit/claim limit** to each customer (process)
   - Maximum credit claim must be stated in advance

2. Reject any request that leads to a **dangerous state**
   - A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
   - A recursive reduction procedure recognizes dangerous states

3. In practice, the system must keep resource usage well below capacity to maintain a **resource surplus**
   - Rarely used in practice due to low resource utilization
Detection and Recovery

- Detection and recovery
  - If we don’t have deadlock prevention or avoidance, then deadlock may occur
  - In this case, we need to detect deadlock and recover from it
- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock
- Possible, but expensive (time consuming)
  - Implemented in VMS
  - Run detection algorithm when resource request times out
Deadlock Detection

- Detection
  - Traverse the resource graph looking for cycles
  - If a cycle is found, preempt resource (force a process to release)
- Expensive
  - Many processes and resources to traverse
- Only invoke detection algorithm depending on
  - How often or likely deadlock is
  - How many processes are likely to be affected when it occurs
Deadlock Recovery

Once a deadlock is detected, we have two options...

1. Abort processes
   - Abort all deadlocked processes
     » Processes need to start over again
   - Abort one process at a time until cycle is eliminated
     » System needs to rerun detection after each abort

2. Preempt resources (force their release)
   - Need to select process and resource to preempt
   - Need to rollback process to previous state
   - Need to prevent starvation
Deadlock Summary

• Deadlock occurs when processes are waiting on each other and cannot make progress
  ✷ Cycles in Resource Allocation Graph (RAG)

• Deadlock requires four conditions
  ✷ Mutual exclusion, hold and wait, no resource preemption, circular wait

• Four approaches to dealing with deadlock:
  ✷ Ignore it – Living life on the edge
  ✷ Prevention – Make one of the four conditions impossible
  ✷ Avoidance – Banker’s Algorithm (control allocation)
  ✷ Detection and Recovery – Look for a cycle, preempt or abort