Lecture 8:
Scheduling & Deadlock

CSE 120: Principles of Operating Systems

Alex C. Snoeren

HW 2 Due NOW

UCSDCSE

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 states (running, waiting, etc.)
  - When an interrupt occurs
  - 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

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
  - This is what you're implementing in Nachos in Project 1
- 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

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

Deadlock

- Processes that acquire multiple resources are dependent on those resources
  - E.g., locks, semaphores, monitors, etc.
- What if one process tries to allocate a resource that a second process holds, and vice-versa?
  - Neither can ever make progress!
  - Dining philosphers problem from Homework 2
- 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

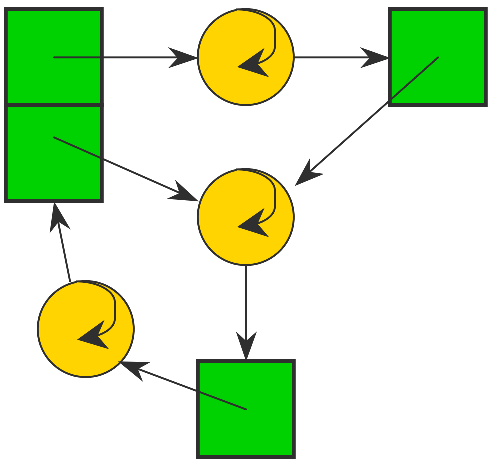
Deadlock can exist if and only if four conditions hold:

1. Mutual exclusion – At least one resource must be held in a non-sharable mode. (*I.e.*, only one instance)
2. Hold and wait – There must be one process holding one resource and waiting for another resource
3. No preemption – Resources cannot be preempted (*I.e.*, 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 a resource held by P_2, P_2 is waiting for P_3, … , and P_n for P_1

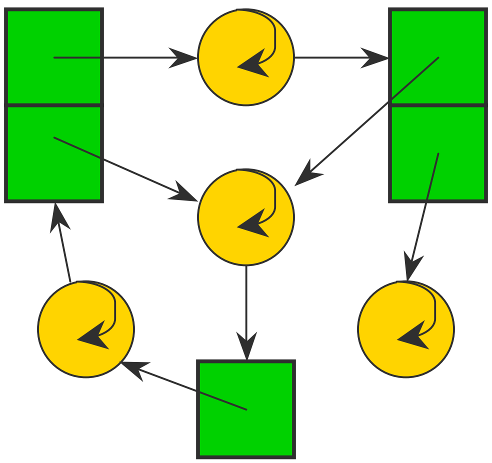
Resource Allocation Graph

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

RAG Example



A cycle…and deadlock!



Same cycle…but no deadlock. Why?

Dealing With Deadlock

There are four ways to deal 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

Prevent at least one condition from happening:

- 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 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 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

Next time…

- Work on Project 1
- We'll review material for the midterm on Thursday