CS 3204
Operating Systems

Lecture 14

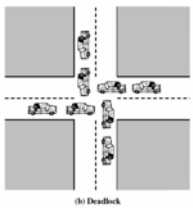
Godmar Back

Announcements

- Project 2 due March 20

Deadlock (Definition)

- A situation in which two or more threads or processes are blocked and cannot proceed
- "blocked" either on a resource request that can't be granted, or waiting for an event that won't occur
  - Possible causes: resource-related or communication-related
- Cannot easily back out

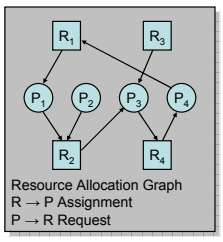


Reusable vs. Consumable Resources

- Distinguish two types of resources when discussing deadlock
- A resource:
  - "anything a process needs to make progress"
- (Serially) Reusable resources (*static, concrete, finite*)
  - CPU, memory, locks
  - Can be a single unit (CPU on uniprocessor, lock), or multiple units (e.g. memory, semaphore initialized with N)
- Consumable resources (*dynamic, abstract, infinite*)
  - Can be created & consumed: messages, signals
- Deadlock may involve reusable resources or consumable resources

Deadlocks, more formally

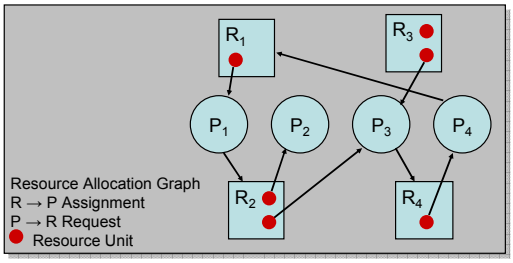
- 4 necessary conditions
  1) Exclusive Access
  2) Hold and Wait
  3) No Preemption
  4) Circular Wait
- Will look at strategies to
  - Prevent
  - Avoid
  - Detect & break deadlocks



Deadlock Detection

- Idea: Look for circularity in resource allocation graph
  - Q.: How do you find out if a directed graph has a cycle?
- Can be done eagerly
  - on every resource acquisition/release, resource allocation graph is updated & tested
- or lazily
  - when all threads are blocked & deadlock is suspected, build graph & test
- Windows provides this for its mutexes as an option
- Note: all processes in BLOCKED state is not sufficient to conclude existence of deadlock. (Why?)
- Note: circularity test is only sufficient criteria if there's only a single instance of each resource – see next slide for multi-unit resources

Multi-Unit Resources



- Note: Cycle, but no deadlock!

Deadlock Detection

- For reusable resources
  - If each resource has exactly one unit, deadlock iff cycle
  - If each resource has multiple units, existence of cycle may or may not mean deadlock
    - Must use reduction algorithm to determine if deadlock exists (Intuition: remove processes that don't have request edges, return their resource units and remove assignment edges, assign resources to remove request edges, repeat until out of processes without request edges. – If entire graph reduces to empty graph, no deadlock.)
- For consumable resources
  - analog algorithm possible
- Q.: What to do once deadlock is detected?

Deadlock Recovery

Increasing Severity

- Preempt resources (if possible)
- Back processes up to a checkpoint
  - Requires checkpointing or transactions (typically expensive)
- Kill processes involved until deadlock is resolved
- Kill all processes involved
- Reboot

Killing Threads or Processes

- Difficult question:
  - When is it safe to kill a thread?
- Consider:

```
thread_func()
{
  while (!done) {
    lock_acquire(&lock);
    // access shared state
    lock_release(&lock);
  }
}
```

What if thread is killed there?

```
thread_func()
{
  while (!done) {
    lock_acquire(&lock);
    p = queue.get();
    queue.put(p);
    lock_release(&lock);
  }
}
```

- Must guarantee full resource reclamation & consistency of all surviving system data structures

Deadlock Prevention (1)

- Idea: remove one of the necessary conditions!
- (C1) (Don't require) Exclusive Access
  - Duplicate resource or make it shareable (where possible)
- (C2) (Avoid) Hold and Wait
  - a) Request all resources at once
    - hard to know in modular system
  - b) Drop all resources if additional request cannot be immediately granted – retry later
    - requires "try_lock" facility
    - can be inefficient if lots of retries

Deadlock Prevention (2)

- (C3) (Allow) Preemption
  - Take resource away from process
    - Difficult: how should process react?
  - Virtualize resource so it can be taken away
    - Requires saving & restoring resource's state
- (C4) (Avoid) Circular Wait
  - Use partial ordering
    - Requires mapping to domain that provides an ordering function (addresses often work!)

Deadlock Avoidance

- Don't grant resource request if deadlock could occur in future
  - Or don't admit process at all
- Banker's Algorithm (Dijkstra 1965, see book)
  - Avoids "unsafe" states that might lead to deadlock
  - Need to know what future resource demands are ("credit lines" of all customers)
  - Need to capture all dependencies (no additional synchronization requirements – "loans" can be called back if needed)
- Mainly theoretical
  - Impractical assumptions
  - Tends to be overly conservative – inefficient use of resources

Deadlock in the Real World

- Most common strategy of handling deadlock
  - Test: fix all deadlocks detected during testing
  - Deploy: if deadlock happens, kill and rerun (easy!)
    - If it happens too often, or reproducibly, add deadlock detection code (see next slide for how to do that in Pintos)
- Weigh cost of preventing vs cost of (re-) occurring
- Static analysis tools detects some kinds of deadlocks before they occur
  - Example: Microsoft Driver Verifier
  - Idea: monitor order in which locks are taken, flag if not consistent lock order

Summary

- Deadlock:
  - 4 necessary conditions: mutual exclusion, hold-and-wait, no preemption, circular wait
- Strategies to deal with:
  - Detect & recover
  - Prevention: remove one of 4 necessary conditions
  - Avoidance: if you can't do that, avoid deadlock by being conservative