

# CS 3204 Operating Systems

Lecture 16  
Godmar Back

## Announcements

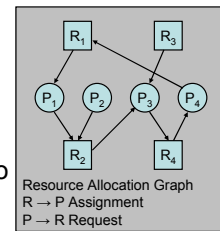
- Project 1 is due **Feb 27, 11:59pm**
  - 5 days left
  - Should be working on priority donation & BSD scheduler now
  - Can now attempt to parallelize some development
    - Merge early & often, regression test
  - priority donation: extra credit for handling & testing donation + priority change
  - fixed-point layer: use at least 14 binary digits after period.
- Office hours this week: 3-4 MWR, 4-5 F
- Reading assignments: Stallings Chapter 9.1-9.4

## Deadlock

Continued

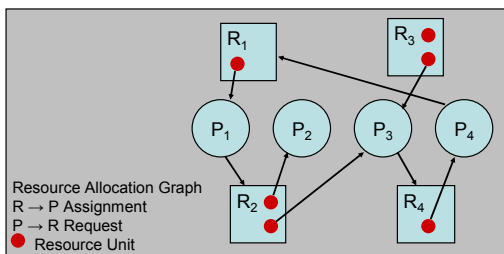
## Deadlocks, more formally

- 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



Variant: removing resources creates "wait-for" graph

## 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



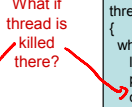

CS 3204 Spring 2006
2/22/2006
7

## Killing Threads or Processes


- Extremely difficult issue:
  - 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


CS 3204 Spring 2006
2/22/2006
8


## 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**
  - Request all resources at once
    - hard to know in modular system
  - *two-phase locking*: Drop all resources if additional request cannot be immediately granted – retry later
    - requires "try\_lock" facility
    - can be inefficient if lots of retries


CS 3204 Spring 2006
2/22/2006
9


## 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!)


CS 3204 Spring 2006
2/22/2006
10


## 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


CS 3204 Spring 2006
2/22/2006
11

## 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


CS 3204 Spring 2006
2/22/2006
12

## Deadlock in Pintos

- How would you implement a deadlock detection algorithm for Pintos?
- Could check that all threads are blocked, and none is blocked on console or disk
- If that happens, provide diagnostics; dump backtraces of all threads
  - Problem 1: can only get backtrace of currently running thread
  - Problem 2: must implement a version of `debug_backtrace()` based entirely on `serial_putc()` (`printf` requires ability to take console lock, so won't always work)
  - Set flag "exit\_all\_threads"
  - Unblock all threads that are blocked
  - In `schedule_tail`, check "exit\_all\_threads" flag and dump backtrace if so, then `thread_exit()`
    - Last thread is `idle_thread`, which calls `PANIC()`
- Can be done in < 100lines of code.
- Alternatively, use gdb macros I posted on forum & website