CS 537
Lecture 9
Deadlock

Quiz Answers

- Use of disabling interrupts
  - Not allowed by processor --> requires system call
  - Not safe is usermode code buggy and allowed by processor
- Locking
  - Just lock manipulation of list, nothing else
- Double-checked locking
  - Is safe here - assuming fine never gets closed and CPU doesn't reorder things or leave values in registers

Readers and Writers Monitor Example

```
Monitor ReadersNWriters {
  int WaitingWriters, WaitingReaders,
        NReaders, NWriters;
  Condition CanRead, CanWrite;

Void BeginWrite()  {
   if(NWriters == 1 ||
      NReaders > 0) {
     ++WaitingWriters;
     wait(CanWrite);
     --WaitingWriters;
   }
   NWriters = 1;
}

Void EndWrite() {
  NWriters = 0;
  if(WaitingReaders)
    Signal(CanRead);
  else Signal(CanWrite);
}
```

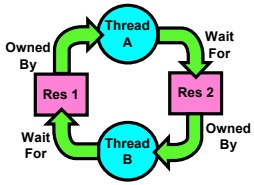
```
Void BeginRead() {
  if(NWriters == 1 ||
     WaitingWriters > 0) {
    ++WaitingReaders;
    Wait(CanRead);
    --WaitingReaders;
  }
  ++NReaders;
  Signal(CanRead);
}

Void EndRead() {
  if(--NReaders == 0)
    Signal(CanWrite);
}
```


What can go wrong?

- For example: dining philosphers
- Primarily, we worry about:
  - **Starvation**: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)
  - **Deadlock**: A policy that leaves all the philosophers "stuck", so that nobody can do anything at all
  - **Livelock**: A policy that makes them all do something endlessly without ever eating!

Starvation vs Deadlock

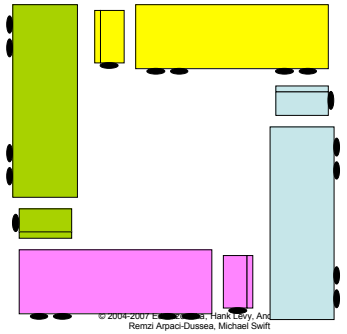
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock ⇒ Starvation but not vice versa
    - Starvation can end (but doesn't have to)
    - Deadlock can't end without external intervention

Real World Deadlocks?

- Gridlock



Testing for deadlock

- How do cars do it?
  - Never block an intersection
  - Must back up if you find yourself doing so
- Why does this work?
  - "Breaks" a wait-for relationship
  - Illustrates a sense in which intransigent waiting (refusing to release a resource) is one key element of true deadlock!

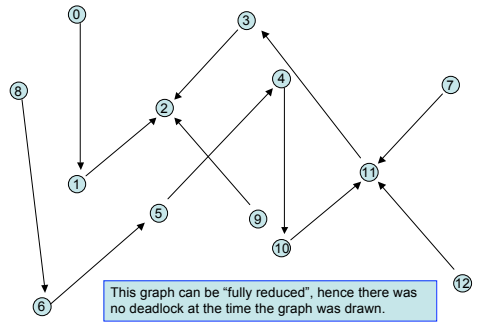
Testing for deadlock

- Steps
  - Collect "process state" and use it to build a graph
    - Ask each process "are you waiting for anything"?
    - Put an edge in the graph if so
  - We need to do this in a single instant of time, not while things might be changing
- Now need a way to test for cycles in our graph

Testing for deadlock

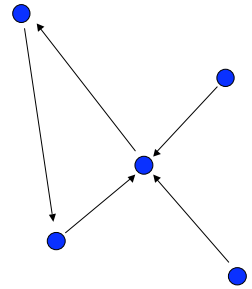
- One way to find cycles
  - Look for a node with no outgoing edges
  - Erase this node, and also erase any edges coming into it
    - Idea: This was a process people might have been waiting for, but it wasn't waiting for anything else
  - If (and only if) the graph has no cycles, we'll eventually be able to erase the whole graph!
- This is called a graph reduction algorithm

Graph reduction example



Graph reduction example

- This is an example of an "irreducible" graph
- It contains a cycle and represents a deadlock, although only some processes are in the cycle



Some questions you might ask

- If a system is deadlocked, could this go away?
  - No, unless someone kills one of the threads or something causes a process to release a resource
  - Many real systems put time limits on "waiting" precisely for this reason. When a process gets a timeout exception, it gives up waiting and this also can eliminate the deadlock
  - But that process may be forced to terminate itself because often, if a process can't get what it needs, there are no other options available!

Some questions you might ask

- Suppose a system isn't deadlocked at time T.
- Can we assume it will still be free of deadlock at time T+1?
  - No, because the very next thing it might do is to run some process that will request a resource…
    - … establishing a cyclic wait
    - … and causing deadlock

Deadlocks

- Definition: Deadlock exists among a set of processes if
  - Every process is waiting for an event
  - This event can be caused only by another process in the set
    - Event is the acquire or release of another resource

One-lane bridge

Four Conditions for Deadlock

- Coffman et. al. 1971
- Necessary conditions for deadlock to exist:
  - **Mutual Exclusion**
    - At least one resource must be held is in non-sharable mode
  - **Hold and wait**
    - There exists a process holding a resource, and waiting for another
  - **No preemption**
    - Resources cannot be preempted
  - **Circular wait**
    - There exists a set of processes $\{P_1, P_2, \ldots P_N\}$, such that
      - P_1 is waiting for P_2, P_2 for P_3, …. and P_N for P_1

All four conditions must hold for deadlock to occur

Dealing with Deadlocks

- Reactive Approaches: detect and recover
  - Periodically check for evidence of deadlock
    - For example, using a graph reduction algorithm
  - Then need a way to recover
    - Could blue screen and reboot the computer
    - Could pick a "victim" and terminate that thread
      - But this is only possible in certain kinds of applications
      - Basically, thread needs a way to clean up if it gets terminated and has to exit in a hurry!
    - Often thread would then "retry" from scratch
- Despite drawbacks, database systems do this

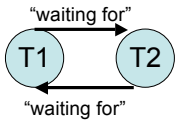
Dealing with Deadlocks

- Proactive Approaches:
  - Deadlock Prevention
    - Prevent one of the 4 necessary conditions from arising
    - …. This will prevent deadlock from occurring
  - Deadlock Avoidance
    - Carefully allocate resources based on future knowledge
    - Deadlocks are prevented
- Ignore the problem
  - Pretend deadlocks will never occur
  - Ostrich approach… but surprisingly common!

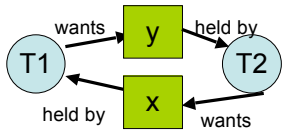
Representing Deadlock

- Two common ways of representing deadlock
  - Vertices:
    - Threads (or processes) in system
    - Resources (anything of value, including locks and semaphores)
  - Edges: Indicate thread is waiting for the other
  - WFG: good for locks, RAG: good for buffers, devices

Wait-For Graph



Resource-Allocation Graph



Deadlock Prevention #1

- Approach
  - Ensure 1 of 4 conditions cannot occur
  - Negate each of the 4 conditions
- No single approach is appropriate (or possible) for all circumstances
- No mutual exclusion --> Make resource sharable
  - Example: Read-only files

Deadlock Prevention #2

- No Hold-and-wait --> Two possibilities
- 1) Only request resources when have none
  - Release resource before requesting next one

Thread 1

```
lock(x);
A += 10;
unlock(x);
lock(y);
B += 20;
unlock(y);
lock(x);
A += 30;
unlock(x);
```

Thread 2

```
lock(y);
B += 10;
unlock(y);
lock(x);
A += 20;
unlock(x);
lock(y);
B += 30;
unlock(y);
```

Deadlock Prevention #2

- No Hold-and-wait
- 2) Atomically acquire all resources at once
  - Example #1: Single lock to protect all

Thread 1
```
lock(z);
A += 10;
B += 20;
A += B;
A += 30;
unlock(z);
```

Thread 2
```
lock(z);
B += 10;
A += 20;
A += B;
B += 30;
unlock(z);
```


Deadlock Prevention #2

- No Hold-and-wait
- 2) Atomically acquire all resources at once
  - Example #2: New primitive to acquire two locks

Thread 1
```
lock(x,y);
A += 10;
B += 20;
A += B;
unlock(y);
A += 30;
unlock(x);
```

Thread 2
```
lock(x,y);
B += 10;
A += 20;
A += B;
unlock(x);
B += 30;
unlock(y);
```


Deadlock Prevention #2

- Problems w/ acquiring many resources atomically
  - Low resource utilization
    - Must make pessimistic assumptions about resource usage

```
if (cond1) {
  lock(x);
}
if (cond2) {
  lock(y);
}
```

  - Starvation
    - If need many resources, others might keep getting one of them

Deadlock Prevention #3

- No “no preemption” --> Preempt resources
- Example: A waiting for something held by B, then take resource away from B and give to A
  - Only works for some resources (e.g., CPU and memory)
  - Not possible if resource cannot be saved and restored
    - Can’t take away a lock without causing problems

Deadlock Prevention #4

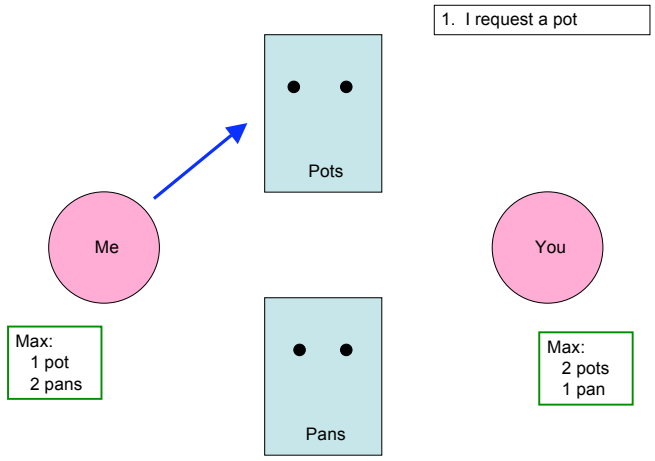
- No circular wait --> Impose ordering on resources
  - Give all resources a ranking; must acquire highest ranked first
  - How to change Example?
- Problems?

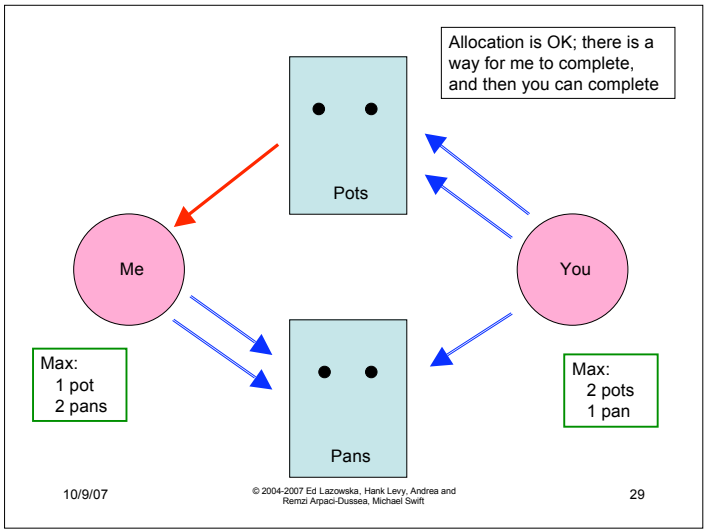
Deadlock Avoidance

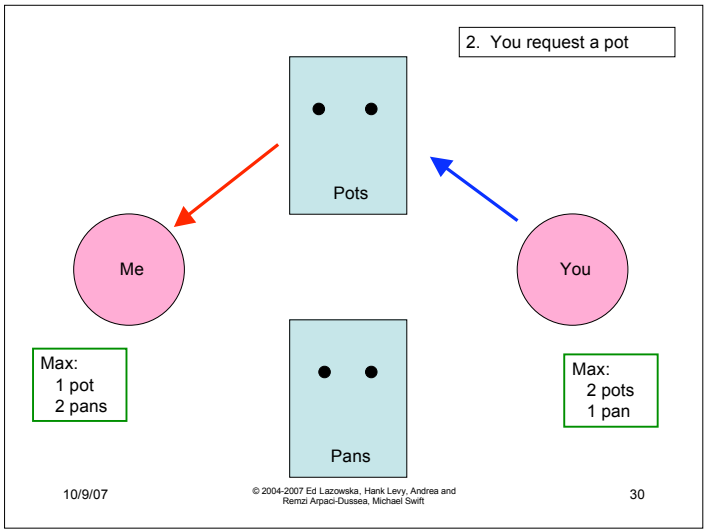
- Dijkstra's Banker's Algorithm
- Avoid **unsafe states** of processes holding resources
  - Unsafe states **might** lead to deadlock if processes make certain future requests
  - When process requests resource, only give if doesn't cause unsafe state
  - Problem: Requires processes to specify all possible future resource demands

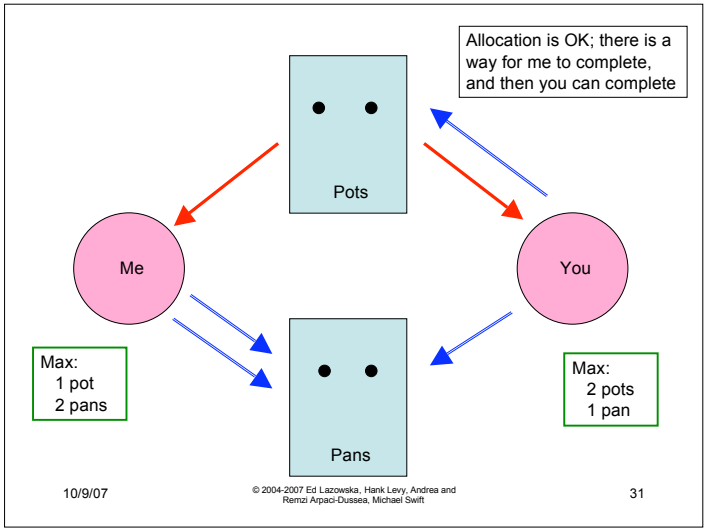
Banker's Algorithm example

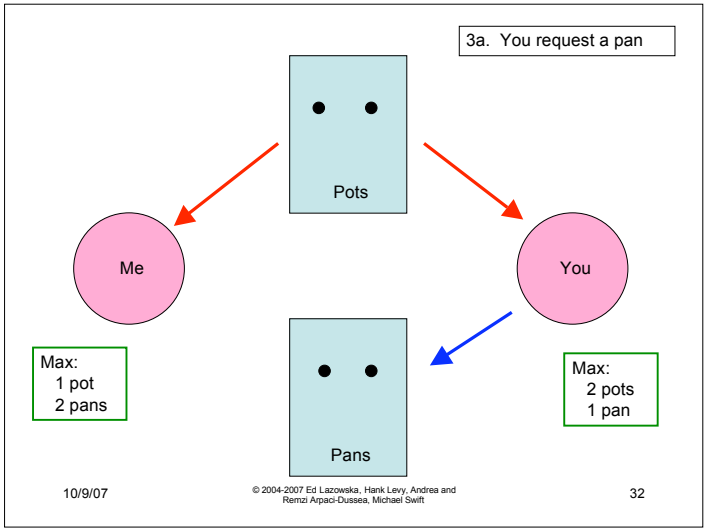
- When a request is made
  - pretend you granted it
  - pretend all other legal requests were made
  - can the graph be reduced?
    - if so, allocate the requested resource
    - if not, block the thread

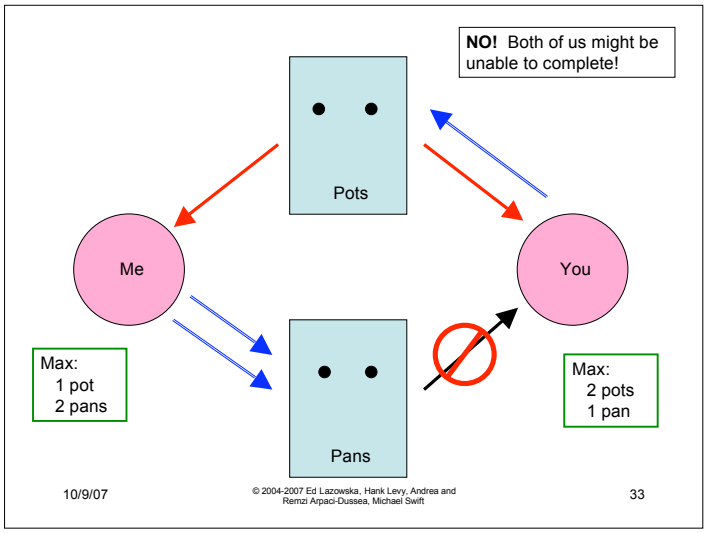


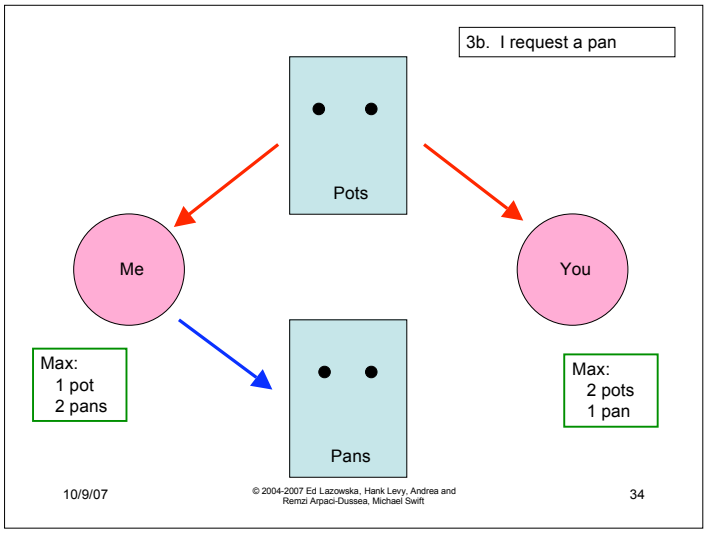

Allocation is OK; there is a way for me to complete, and then you can complete
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan
10/9/07
© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dussea, Michael Swift
29

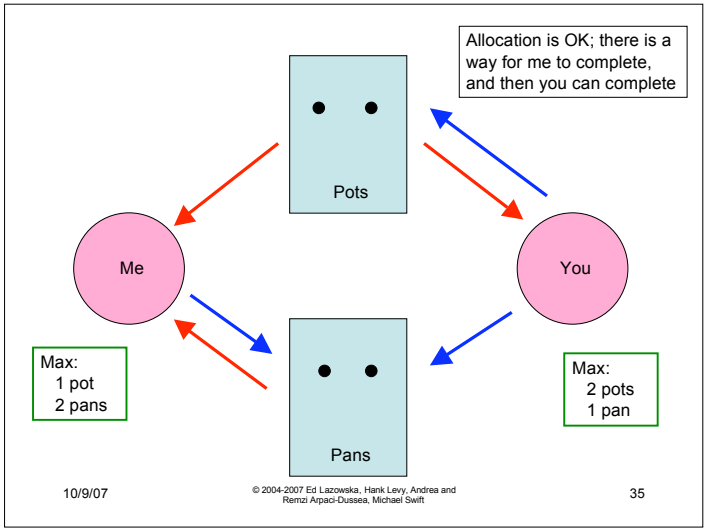

2. You request a pot
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan
10/9/07
© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dussea, Michael Swift
30


Allocation is OK; there is a way for me to complete, and then you can complete
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan
10/9/07
© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dussea, Michael Swift
31


3a. You request a pan
Pots
Me
You
Pans
Max:
1 pot
2 pans
Max:
2 pots
1 pan
10/9/07
© 2004-2007 Ed Lazowska, Hank Levy, Andrea and Remzi Arpaci-Dussea, Michael Swift
32







Deadlock Detection and Recovery

- Detection
  - Maintain wait-for graph of requests
  - Run algorithm looking for cycles
    - When should algorithm be run?
- Recovery: Terminate deadlock
  - Reboot system (Abort all processes)
  - Abort all deadlocked processes
  - Abort one process in cycle
- Challenges
  - How to take resource away from process? Undo effects of process (e.g., removing money from account)
    - Must roll-back state to safe state (checkpoint memory of job)
  - Could starve process if repeatedly abort it

When to run Detection Algorithm?

- For every resource request?
- For every request that cannot be immediately satisfied?
- Once every hour?
- When CPU utilization drops below 40%?

Deadlock Recovery

- Killing one/all deadlocked processes
  - Crude, but effective
  - Keep killing processes, until deadlock broken
  - Repeat the entire computation
- Preempt resource/processes until deadlock broken
  - Selecting a victim (# resources held, how long executed)
  - Rollback (partial or total)
  - Starvation (prevent a process from being executed)

Summary: Handing Deadlock

- Deadlock prevention
  - Ensure deadlock does not happen
  - Ensure at least one of 4 conditions does not occur
- Deadlock avoidance
  - Ensure deadlock does not happen
  - Use information about resource requests to dynamically avoid unsafe situations
- Deadlock detection and recovery
  - Allow deadlocks, but detect when occur
  - Recover and continue
- Ignore
  - Easiest and most common approach