CS 153
Design of Operating Systems

Winter 2016

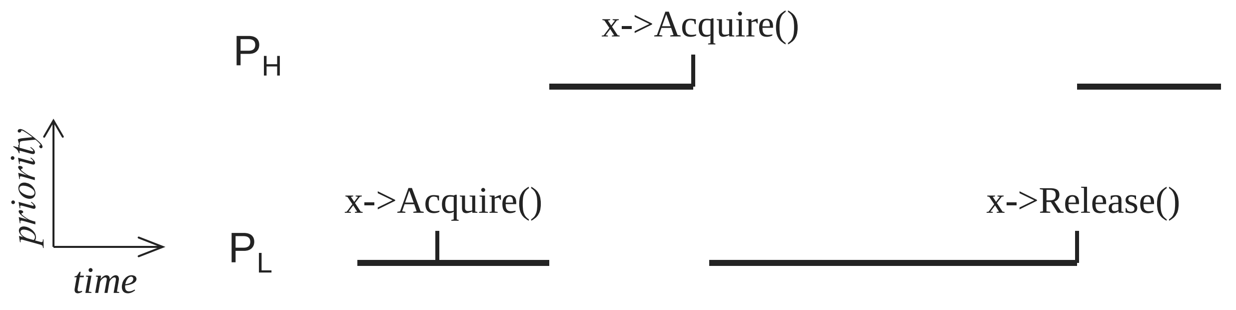
Lecture 12: Scheduling & Deadlock

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

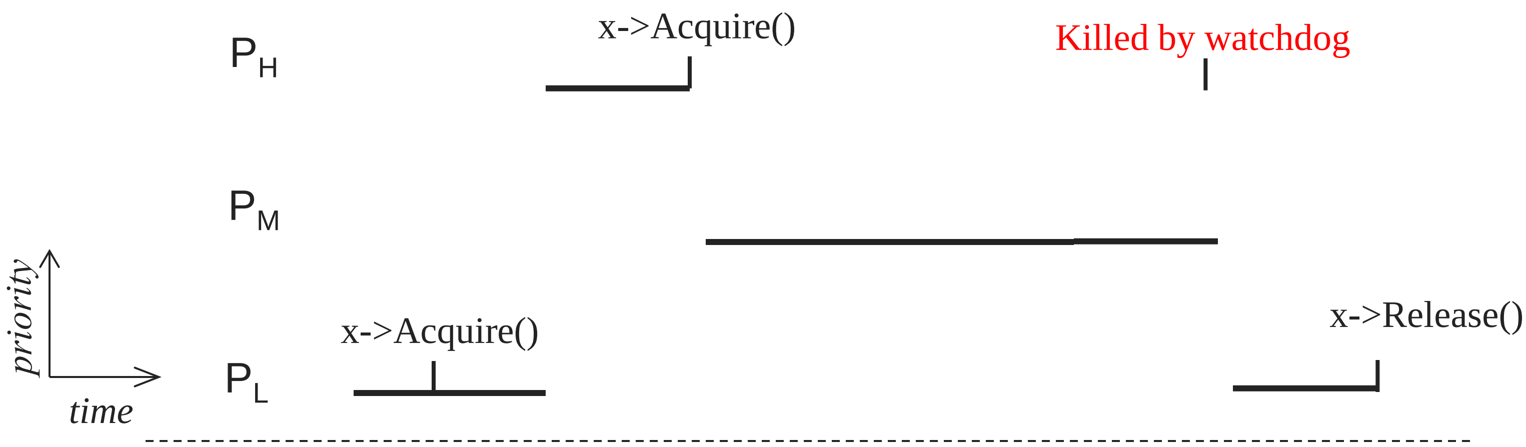
More on Priority Scheduling

- For real-time (predictable) systems, priority is often used to isolate a process from those with lower priority. *Priority inversion* is a risk unless all resources are jointly scheduled.



Priority Inversion on Mars Pathfinder

- P_H = (Frequent) Bus Management
- P_M = (Long-Running) Communications
- P_L = (Infrequent and short) Data Gathering



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

Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation-free
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)

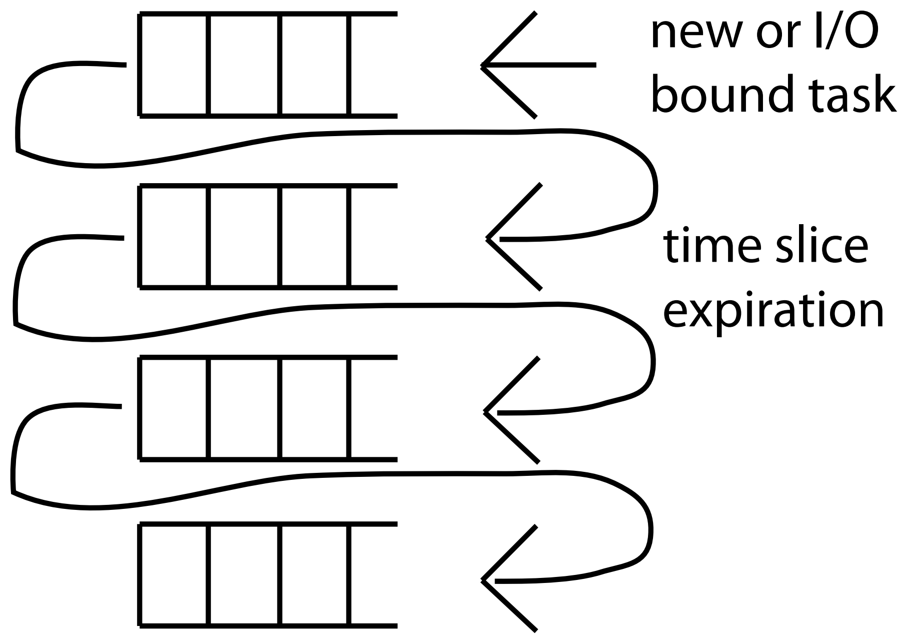
MFQ

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first task in highest priority queue
  - If time slice expires, task drops one level

MFQ

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



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

Multiprocessor Scheduling

- This is its own topic, we won't go into it in detail
  - Could come back to it towards the end of the quarter

- What would happen if we used MFQ on a multiprocessor?
  - Contention for scheduler spinlock
  - Multiple MFQ used – this optimization technique is called distributed locking and is common in concurrent programming
- A couple of other considerations
  - Co-scheduling for parallel programs
  - Core affinity

Scheduling Summary

- Scheduler (dispatcher) is the module (not a thread) 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

Deadlock!

Deadlock—the deadly embrace!

- Synchronization– 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 access 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

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

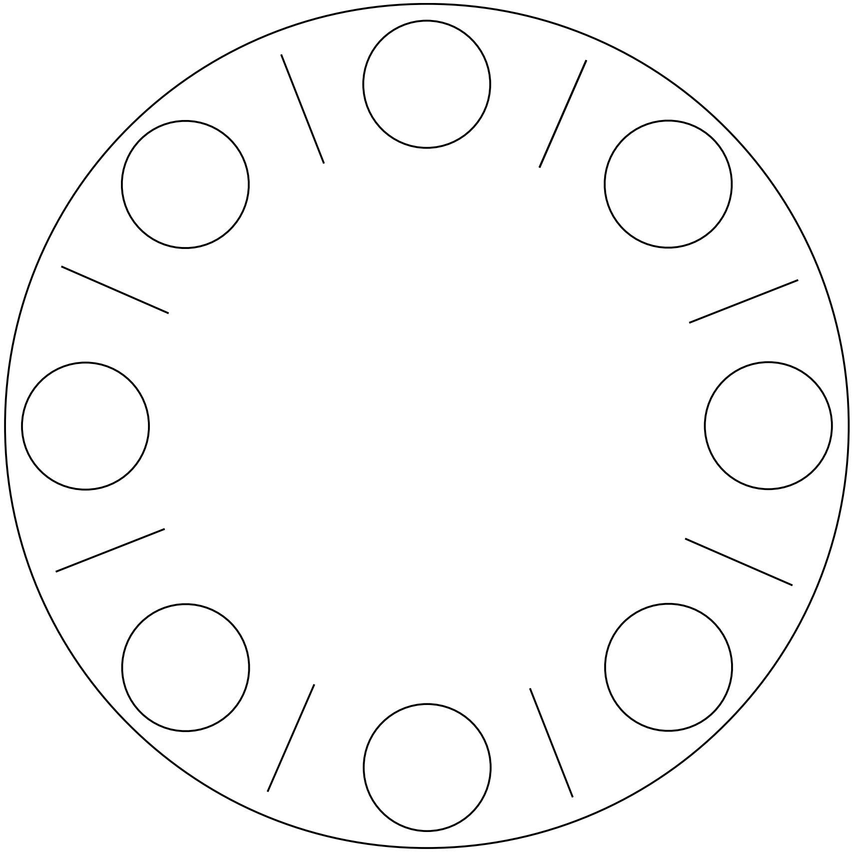
Process 1

```
lockA->Acquire();
…
lockB->Acquire();
```

Process 2

```
lockB->Acquire();
…
lockA->Acquire();
```

Dining Philosophers



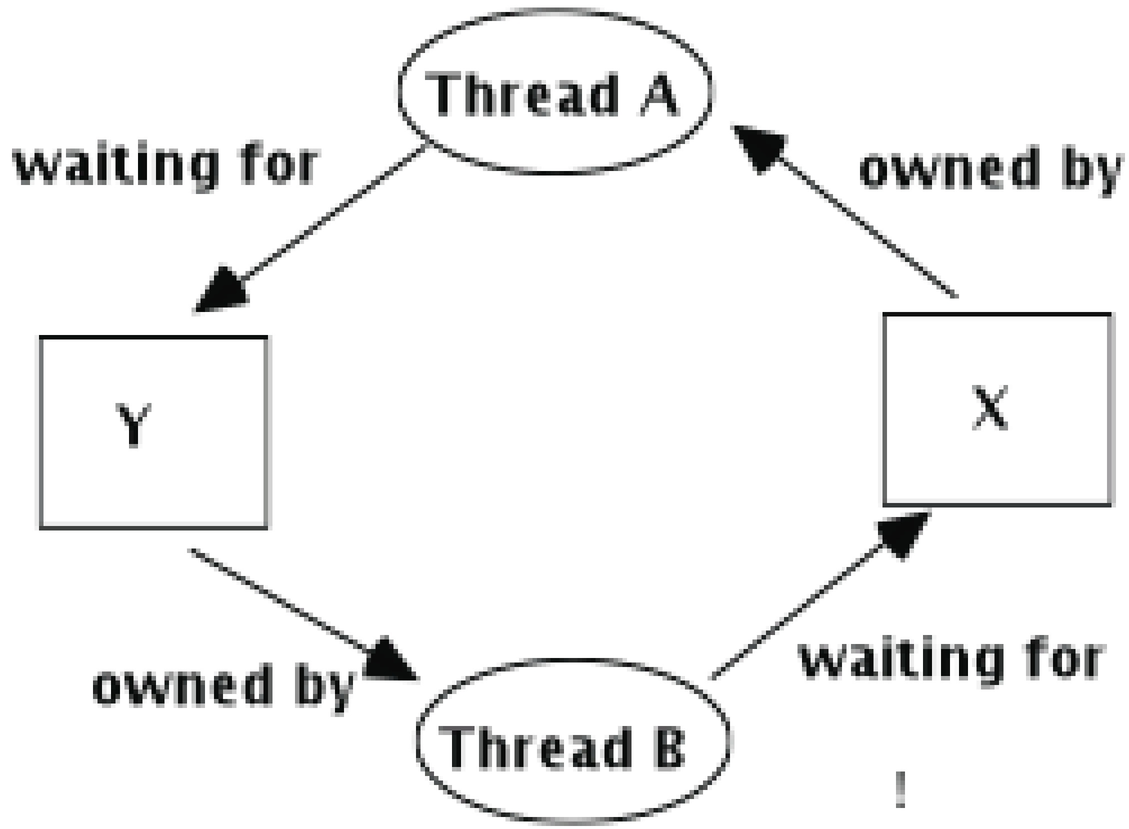
Each lawyer needs two chopsticks to eat.
Each grabs chopstick on the right first.

Real example!

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,…,P_n] such that P_1 is waiting for P_2, P_2 for P_3, etc.

Circular Waiting



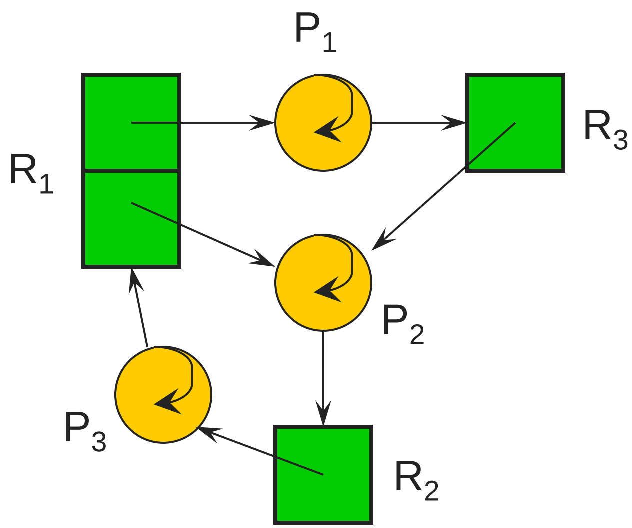
Dealing With Deadlock

- There are four approaches for dealing with deadlock:
  - Ignore it – responsibility of the developers. UNIX and Windows take this approach
  - Detection and Recovery – look for a cycle in dependencies
  - Prevention – make it impossible for deadlock to happen
  - Avoidance – control allocation of resources

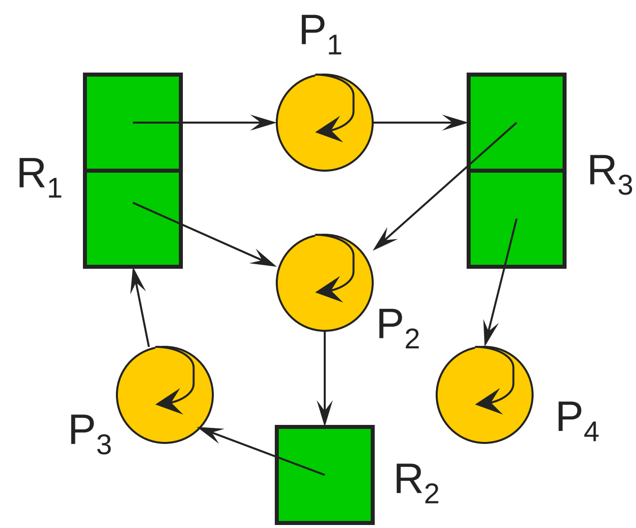
Resource Allocation Graph

- Deadlock can be described using a resource allocation graph (RAG)
- The RAG consists of a set of vertices P={P_1, P_2, …, P_n} of processes and R={R_1, R_2, …, R_m} of resources
  - A directed edge from a process to a resource, $P_i \rightarrow R_i$, means that P_i has requested R_j
  - A directed edge from a resource to a process, $R_i \rightarrow P_i$, means that R_j has been allocated to 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?

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)
- 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

#1: Detection and Recovery

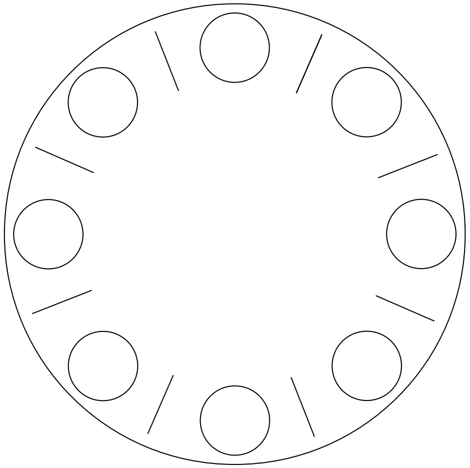
- Algorithm
  - Scan waits-for graph (WFG) or RAG
  - Detect cycles
  - Fix cycles
  - May be expensive
- How?
  - Remove one or more threads, reassign its resources
    - » Requires exception handling code to be very robust
  - Roll back actions of one thread (Preempt resources)
    - » Databases: all actions are provisional until committed
    - » Hard for general cases

#2: 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 all needed resources at once (in the beginning)
  - Preemption
    - OS can preempt resource (costly)
  - Circular wait
    - Impose an ordering (numbering) on the resources and request them in order (popular implementation technique)

#2: Deadlock Prevention

- How would you do each of the following for dining philosophers?
  - Don't enforce mutex?
  - Don't allow hold and wait?
  - Allow preemption?
  - Don't allow circular waiting?



#3: Deadlock Avoidance

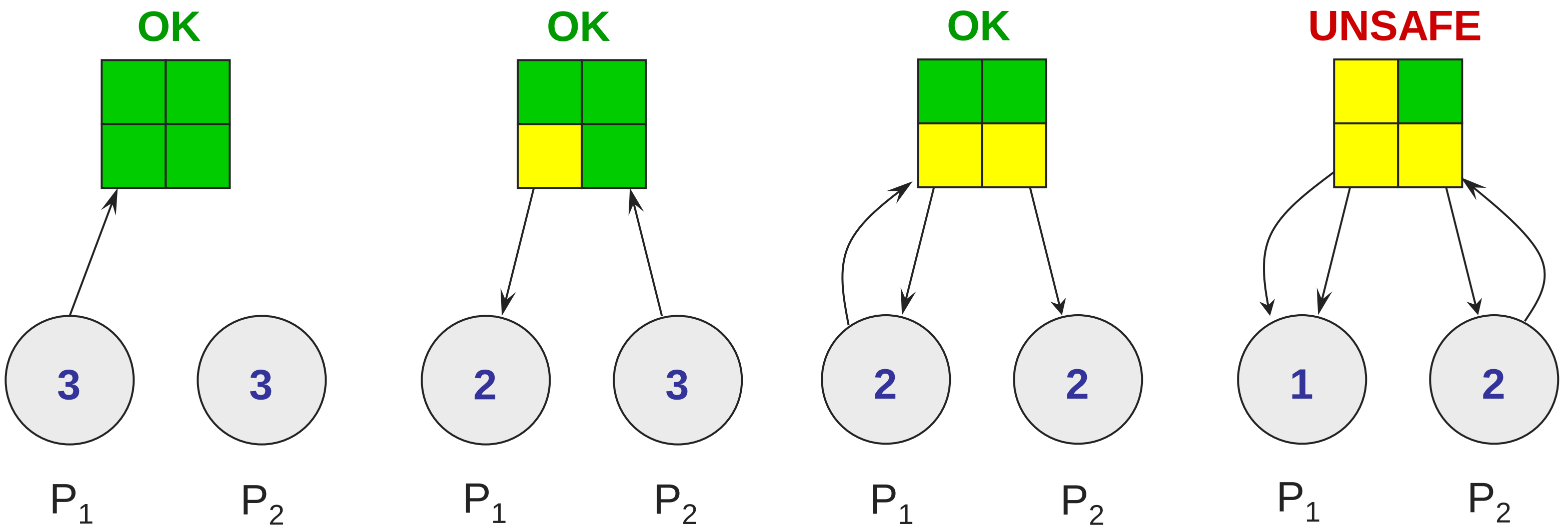
- Avoidance – dynamic strategy
  - Provide information in advance about what resources will be needed by processes to guarantee that deadlock will not happen
  - System only grants resource requests if it knows that the process can obtain all resources it needs in future requests
    - » Hint: it will release all resources eventually
  - Avoids circularities (wait dependencies)
- Tough
  - Hard to determine all resources needed in advance
  - Good theoretical problem, not as practical to use

Banker's Algorithm

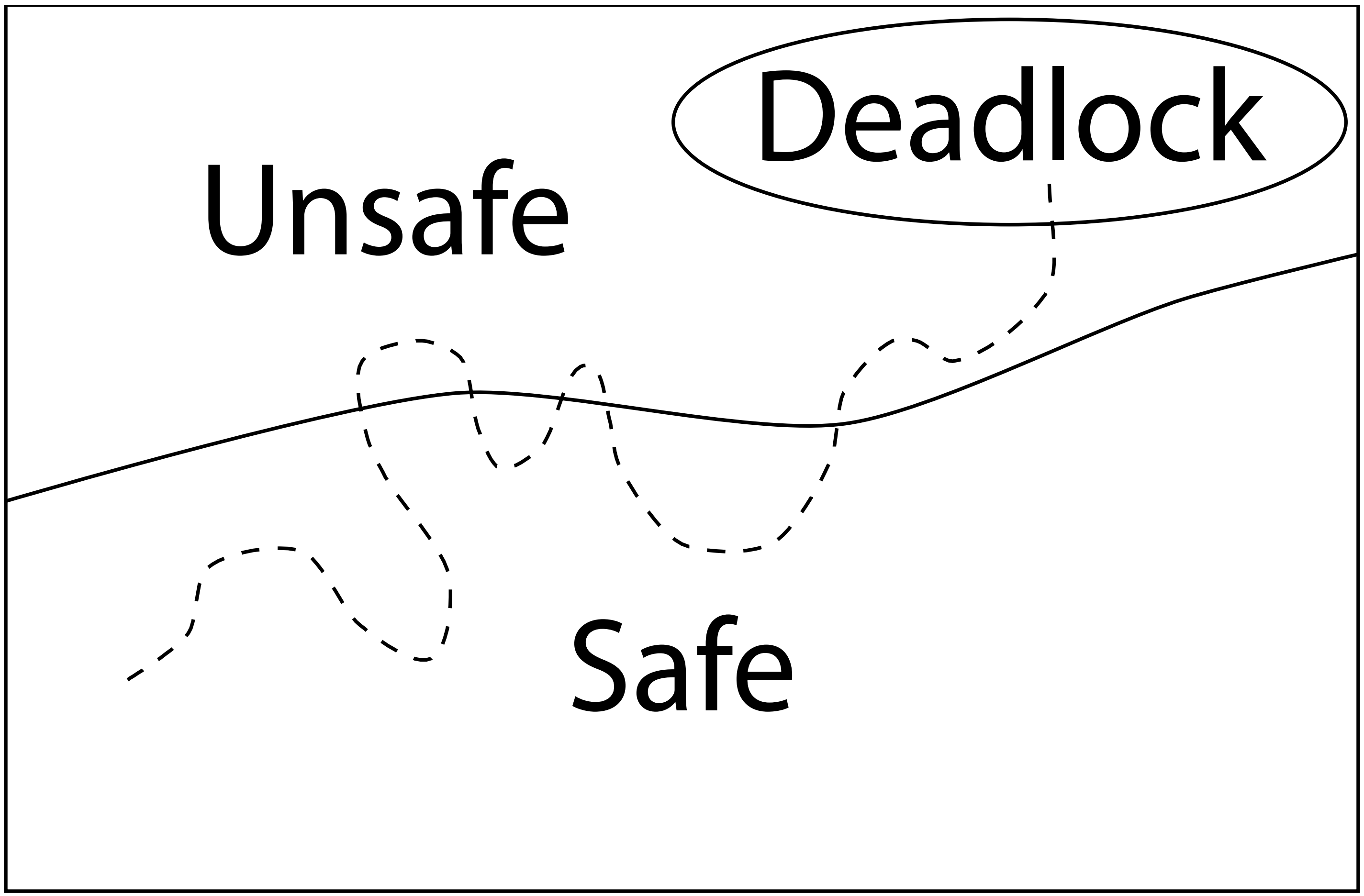
- The Banker's Algorithm is the classic approach to deadlock avoidance for resources with multiple units

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

Banker's Algorithm Simplified



Possible System States



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
  - Detection and Recovery – Look for a cycle, preempt or abort
  - Prevention – Make one of the four conditions impossible
  - Avoidance – Banker's Algorithm (control allocation)

Concurrency Bugs

- Subtle to detect compared to deterministic bugs
- A huge problem in critical infrastructure (airplane control, power systems)
- Extensive research still ongoing
- Hardest traditional OS research problem, the others:
  - Memory problems
  - File system problems

Example Concurrency Bug

Thread 1

```
nsFileTransport::Process()
{
    ...
W1  mStatus =
        mOutputStream->WriteFrom();

R1  if(mStatus == STREAM_WOULD_BLOCK)
    {
W2    mStatus = NS_OK;
      return;
    }
    ...
}
```

Thread 2

```
HandleEvent()
{

R2  if(mStatus != NS_OK)
      return; // ignore event

}
```

Incorrect Interleaving

nsFileTransport.cpp

A concurrency bug in Mozilla

Example Concurrency Bug

Thread 1

```
    void LoadScript(nsSpt *aspt)
    {
      Lock(l);
W1    gCurrentScript = aspt;
      LaunchLoad(aspt);
      Unlock(l);
    }

    void OnLoadComplete()
    {
      /* callback */
      Lock(l);
R1    gCurrentScript->compile();
      Unlock(l);
    }
```

Thread 2

```
      Lock(l);
W2    gCurrentScript = NULL;
      Unlock(l);
```

nsXULDocument.cpp

Incorrect Interleaving

Another concurrency bug in Mozilla

Research on Concurrency Problems

- *Jie Yu and Satish Narayanasamy*, A Case for an Interleaving Constrained Shared-Memory Multi-Processor, **ISCA 09**
  - Observe safe thread inter-leavings during software testing phase
  - Disallow any unseen inter-leavings
  - Safe but may be too conservative

Next Class

- Preparation for Exam