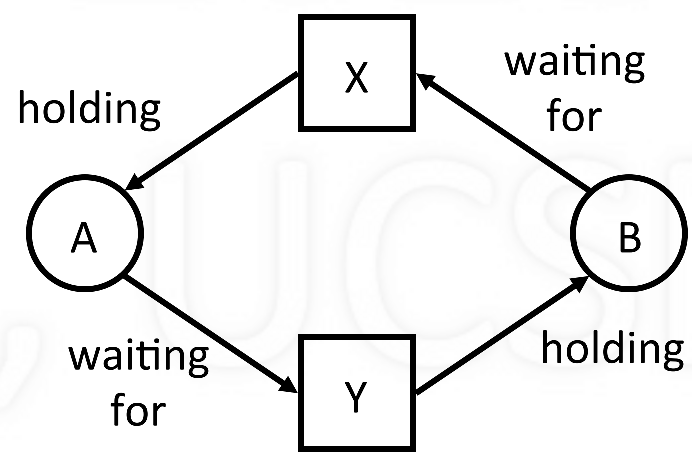
CSE 120: Principles of Operating Systems
Lecture 7: Deadlock

Prof. Joseph Pasquale
University of California, San Diego

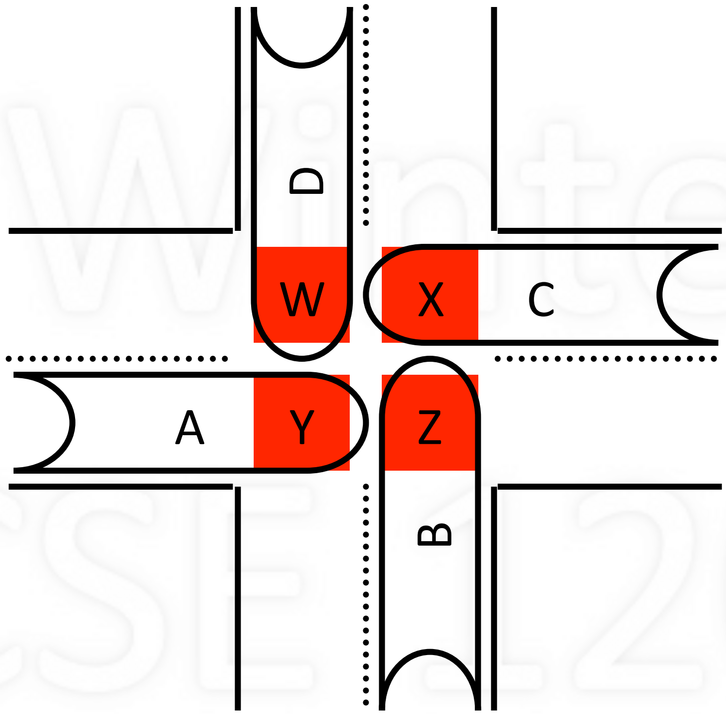
February 13, 2023

What is Deadlock?

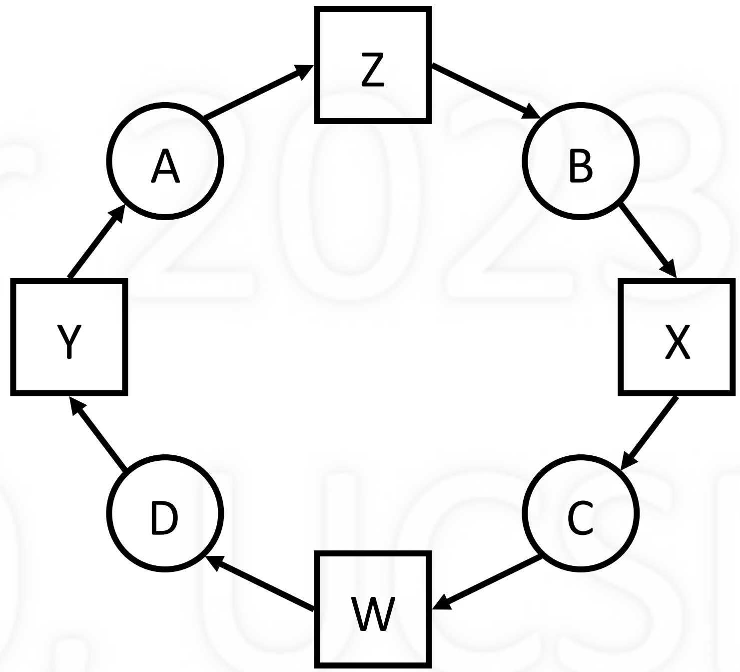
- Set of processes are permanently blocked
  - Unblocking of one relies on progress of another
  - But none can make progress!
- Example
  - Processes A and B
  - Resources X and Y
  - A holding X, waiting for Y
  - B holding Y, waiting for X
  - Each is waiting for the other; will wait forever



Traffic Jam as Example of Deadlock



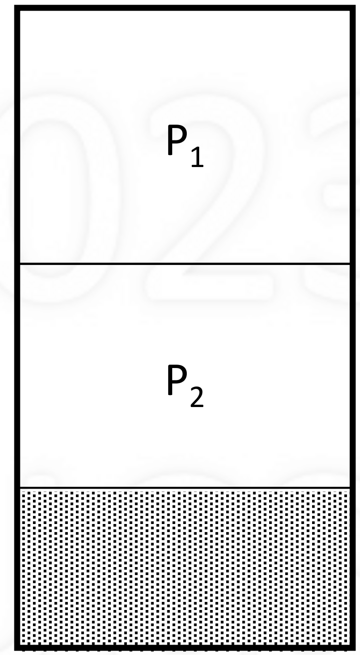
Cars deadlocked
in an intersection



Resource Allocation
Graph

Example of Deadlock: Memory

- Total memory = 200MB
- P_1 requests 80MB
- P_2 requests 70MB
- P_1 requests 60MB (wait)
- P_2 requests 80MB (wait)



Four Conditions for Deadlock

- Mutual Exclusion
  - Only one process may use a resource at a time
- Hold-and-Wait
  - Process holds resource while waiting for another
- No Preemption
  - Can't take a resource away from a process
- Circular Wait
  - The waiting processes form a cycle

How to Attack the Deadlock Problem

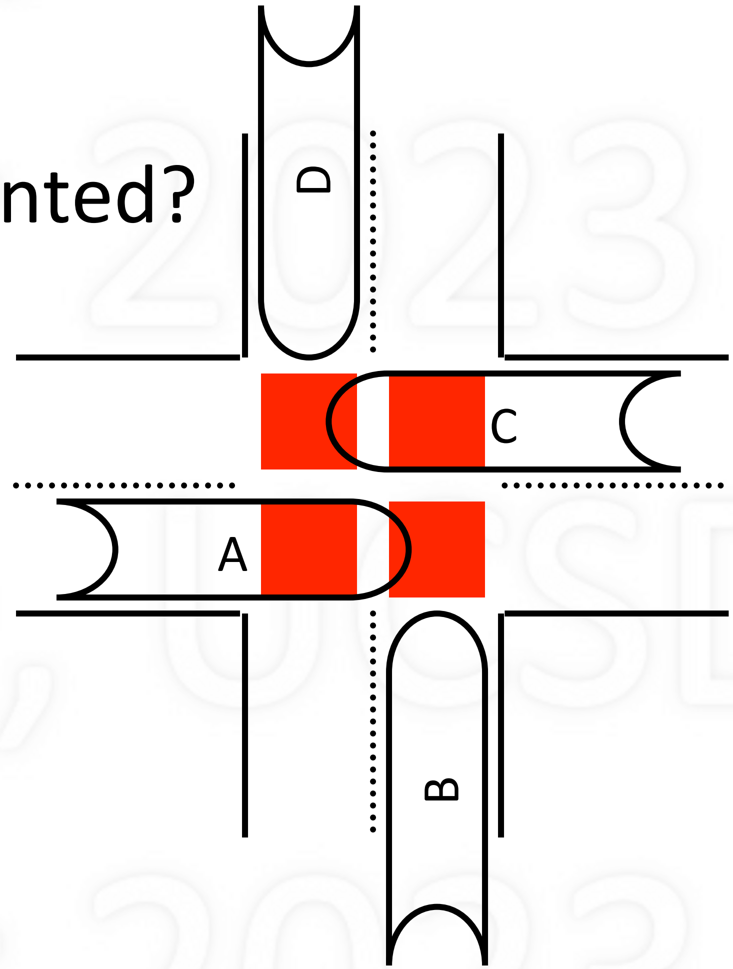
- Deadlock Prevention
  – Make deadlock impossible by removing condition
- Deadlock Avoidance
  – Avoid getting into situations that lead to deadlock
- Deadlock Detection
  – Don't try to stop deadlocks
  – Rather, if they happen, detect and resolve

Deadlock Prevention

- Simply prevent any single condition for deadlock
- Mutual exclusion
  – Relax where sharing is possible
- Hold-and-wait
  – Get all resources simultaneously (wait until all free)
- No preemption
  – Allow resources to be taken away
- Circular wait
  – Order all the resources, force ordered acquisition

How Can We Prevent a Traffic Jam?

- Add a traffic light
- Which condition is prevented?
  - Mutual exclusion?
  - Hold-and-wait?
  - No Preemption?
  - Circular Wait?



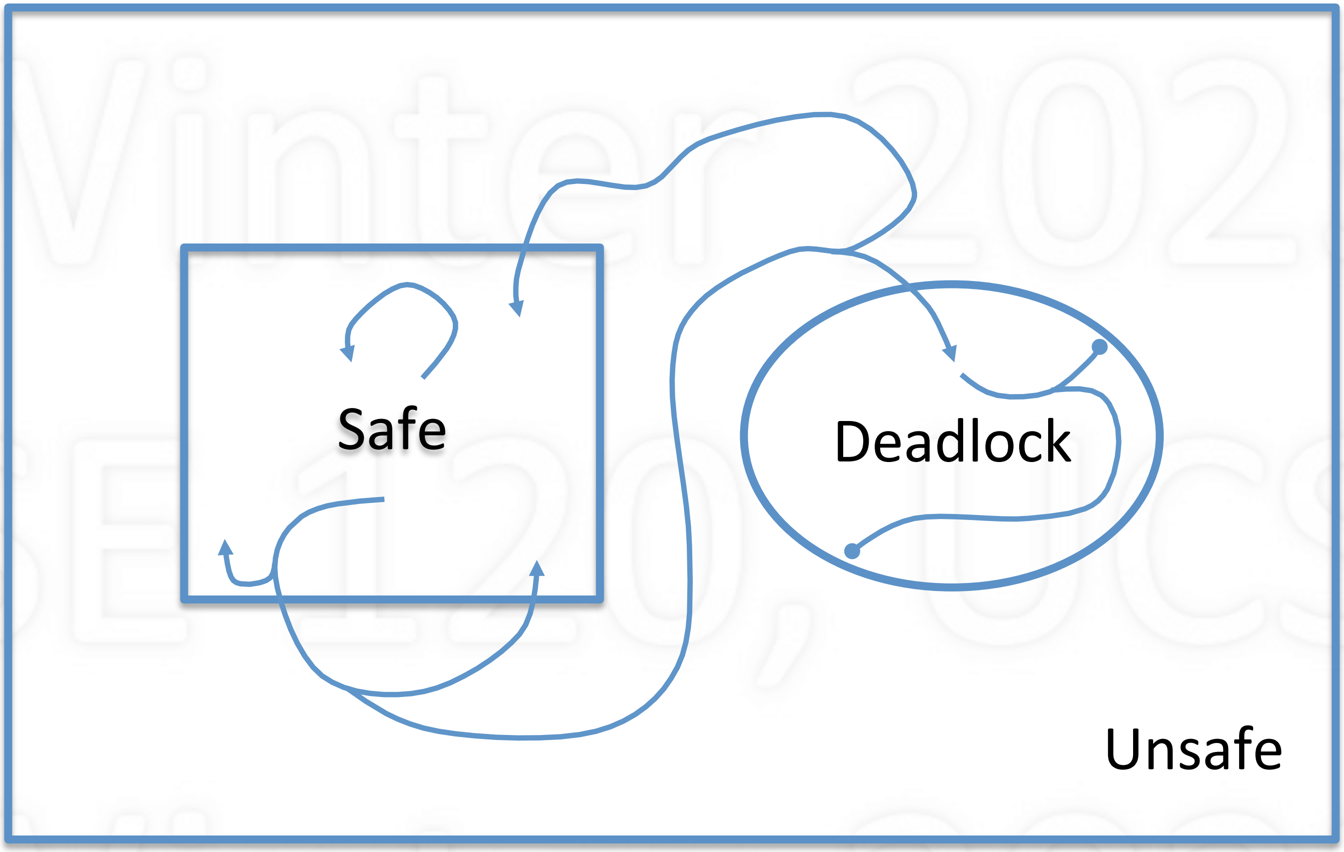
Deadlock Avoidance

- Avoid situations that lead to deadlock
  - Selective prevention
  - Remove condition only when deadlock is possible
- Works with incremental resource requests
  - Resources are asked for in increments
  - Do not grant request that can lead to a deadlock
- Need maximum resource requirements

Banker's Algorithm

- Fixed number of processes and resources
  - Each process has zero or more resources allocated
- System state: either safe or unsafe
  - Depends on allocation of resources to processes
- Safe: deadlock is absolutely avoidable
  - Can avoid deadlock by certain order of execution
- Unsafe: deadlock is possible (but not certain)
  - May not be able to avoid deadlock

Safe, Unsafe, and Deadlock States



Banker's Algorithm

- Given
  - process/resource claim matrix
  - process/resource allocation matrix
  - resource availability vector
- Is there a process ordering such that
  - a process can run to completion, return resources
  - resources can then be used by another process
  - eventually, all the processes complete

Example of a Safe State

	Claim				Allocation				Avail-ability	Total
	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4		
R_1	3	6	3	4	1	6	2	0	0	9
R_2	2	1	1	2	0	1	1	0	1	3
R_3	2	3	4	2	0	2	1	2	1	6

- This is a safe state
- Which process can run to completion? P_2
- After P_2 completes, its resources are returned
- Next select P_1, then P_3, then P_4

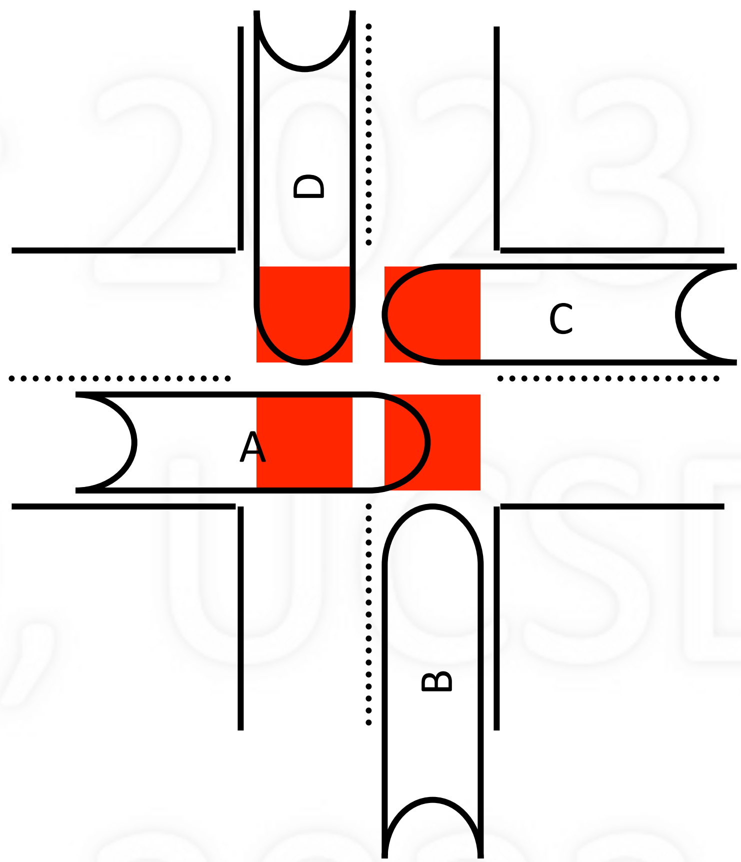
Example of an Unsafe State

	Claim				Allocation				Availability	Total
	P_1	P_2	P_3	P_4	P_1	P_2	P_3	P_4		
R_1	3	6	3	4	2	5	2	0	0	9
R_2	2	1	1	2	0	1	1	0	1	3
R_3	2	3	4	2	1	1	1	2	1	6

- This is an unsafe state
- No process can definitely run to completion
- P_1 may block asking for R_1; same for P_2, P_3, P_4
- Deadlock possible, but not necessarily certain

How Can We Prevent a Traffic Jam?

- What are the resources?
- What is a safe state?
- How to avoid deadlock?
  – At most 3 cars into intersection
- Which condition being prevented?

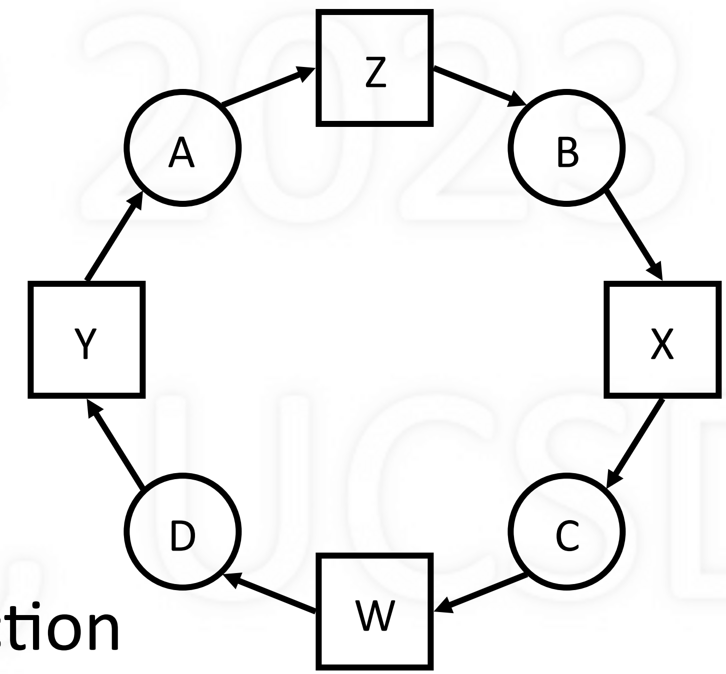


Deadlock Detection and Recovery

- Do nothing special to prevent/avoid deadlocks
  - If they happen, they happen
  - Periodically, try to detect if a deadlock occurred
  - Do something (or even nothing) about it
- Reasoning
  - Deadlocks rarely happen
  - Cost of prevention or avoidance not worth it
  - Deal with them in special way (may be very costly)
- Most general purpose OS's take this approach!

Detecting a Deadlock

- Construct resource allocation "wait-for" graph
  - If cycle, deadlock
- Requires
  - Identifying all resources
  - Tracking their use
  - Periodically running detection algorithm



Recovery from Deadlock

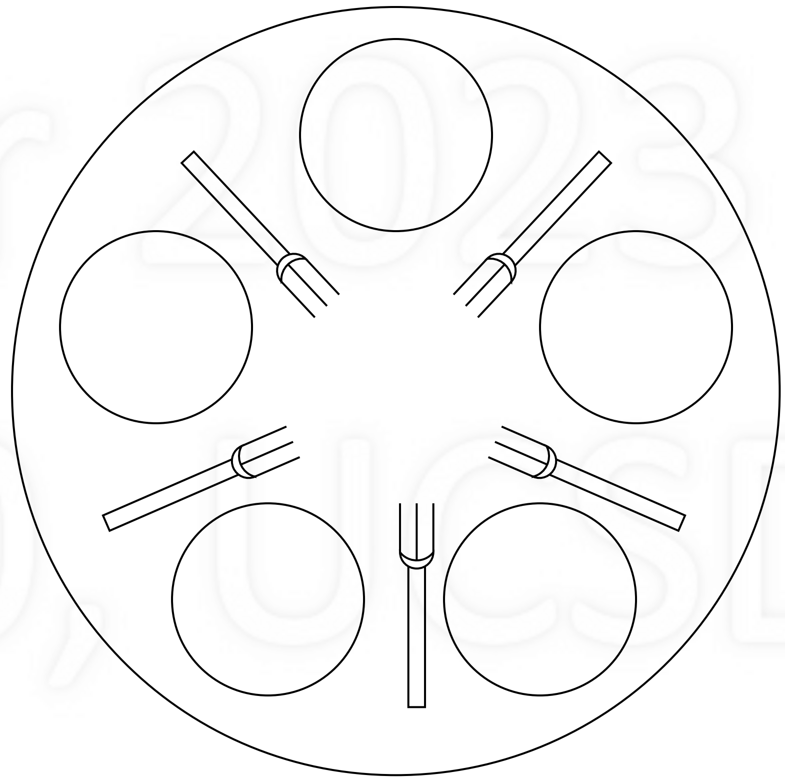
- Terminate all deadlocked processes
  - Will remove deadlock, but drastic and costly
- Terminate deadlocked processes one at a time
  - Do until deadlock goes away (need to detect)
  - What order should processes be ended?
- What about resources in inconsistent states
  - Such as files that are partially written?
  - Or interrupted message (e.g., file) transfers?

Classical Synchronization Problems

- Producer/Consumer (Bounded Buffer)
- Dining Philosophers
- Readers/Writers
- Study these problems and their solutions!

The Dining Philosopher's Problem

- Five philosophers
  - Think, eat, think, eat, …
- To eat
  - Pick up two forks
    - one at a time
  - Eat
  - Put down forks
- Mutual exclusion
  - Avoid deadlock or starvation



Implementing Dining Philosophers

- Identify critical section(s)
- How to achieve mutual exclusion?
- How to avoid deadlock?
- How to avoid starvation?
- How to generalize to *n* philosophers?

```
DoPhilosopher (int i) {
  while (TRUE) {
    Think ();
    PickupFork (i);
    PickupFork ((i+1)%5);
    Eat ();
    PutdownFork ((i+1)%5);
    PutdownFork (i);
  }
}
```

The Readers-Writers Problem

- Readers: processes that only read files
- Writers: processes that modify files
- Rules
  - Allow multiple simultaneous readers
  - A writer gets exclusive access
  - Avoid starvation: Once a writer arrives, wait until current readers leave, and then do not allow any new readers while writing

Textbook

- OSP: Chapter 7
- OSC: Chapter 8 (on Deadlocks)
  - Lecture-related: 8.1-8.9 (Deadlocks), 7.1 (Classic Synchronization Problems)
  - Recommended: 7.2-7.6