CS 492: Operating Systems

Deadlocks (3)

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Goals for Today

- Deadlock
  - How to deal with deadlocks
    - Ignoring them: ostrich algorithm
    - Detecting & recovering from deadlock
    - Preventing deadlock
    - Avoiding deadlock

Four Necessary Conditions for Deadlock

1. Mutual exclusion condition
   - Each resource assigned to at most 1 process
2. Hold and wait condition
   - Process holding resources can request for additional resources
3. No preemption condition
   - Assigned resources cannot be claimed involuntarily
4. Circular wait condition
   - Must be a circular chain of 2 or more processes

- All four conditions must be present when a deadlock occurs
- If one condition is absent, the deadlock is impossible to happen

Deadlock Prevention

Principle:

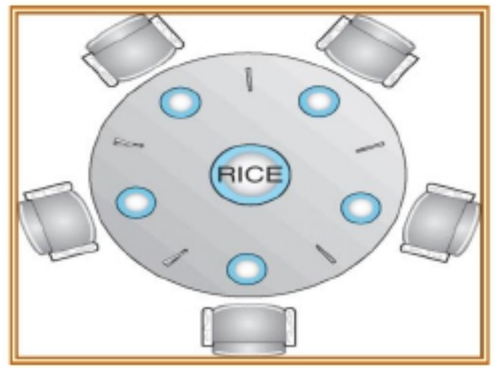
If any one of the four conditions for deadlock (with reusable resources) is denied, deadlock is impossible.

Deadlock Prevention

1. Attacking the Mutual Exclusion Condition

- Principle:
  - Allow resources to be shared, if possible
- Not all devices can be spooled
  - However, the mutual-exclusion condition must hold for non-sharable resources

Example of Deadlock: Dining Lawyers Problem



- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
  - May cause deadlock!
- Can we attack the mutual exclusion condition? If so, how?

Deadlock Prevention

2. Attacking the Hold and Wait Condition

- Goal: Prevent processes that hold resources from waiting for more resources

Breaking Deadlocks for Dining Lawyers Problem

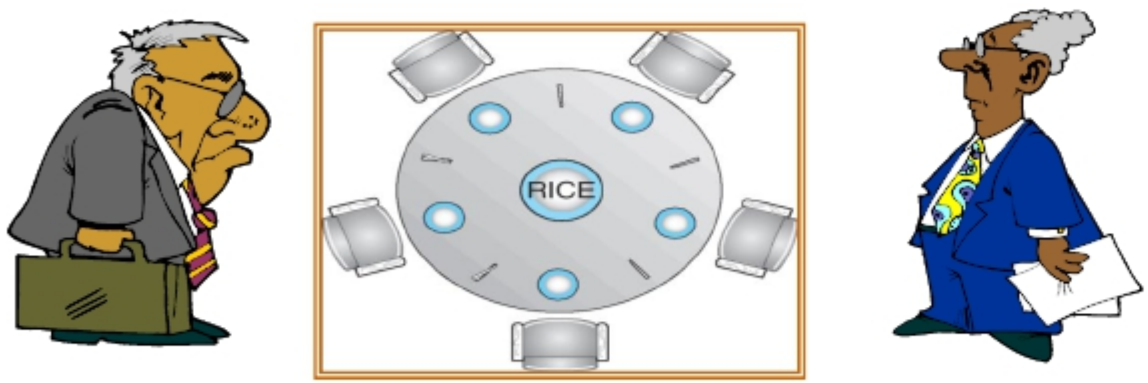

- Can we attack the hold and wait condition? If so, how?

Deadlock Prevention

3. Attacking the No Preemption Condition

- If a process is holding some resources and request another resource that is not available, then all resources the process is currently holding are preempted (released).
- Applied to resources whose state can be easily saved and restored later
  - CPU registers, memory space, etc.

Breaking Deadlocks for Dining Lawyers Problem



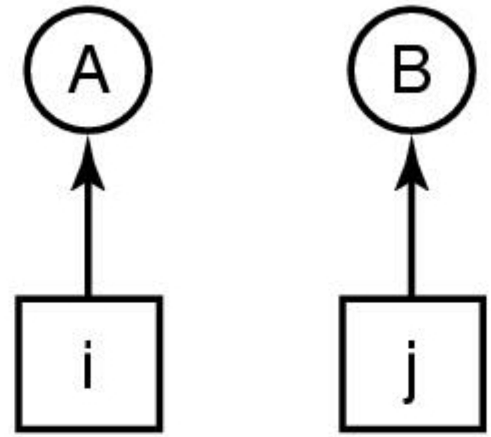
- Can we attack the no Preemption condition? If so, how?

Deadlock Prevention

4. Attacking the Circular Wait Condition

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)



(b)

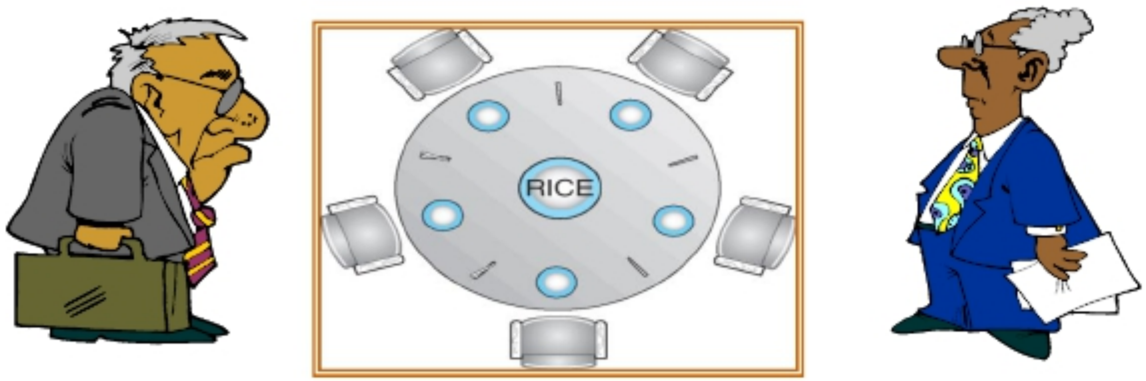
- Normally ordered resources
- A resource graph

Deadlock Prevention

4. Attacking the Circular Wait Condition

- Rule: All requests of a process must be made in numerical order
- Problem: impossible to find an ordering to satisfy everyone

Breaking Deadlocks for Dining Lawyers Problem



- Can we attack the circular wait condition? If so, how?

Summary of Deadlock Prevention

Condition	Approach
Mutual Exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Next..

- Deadlock
  - Concepts
  - How to deal with deadlocks
    - Ignoring them: ostrich algorithm
    - Detecting & recovering from deadlock
    - Preventing deadlock
    - Avoiding deadlock

Deadlock - What to do?

- Different strategies
  - Detection/Recovery
    - Clean up only when trouble really happens
  - Prevention
    - Break one of four necessary conditions
  - Avoidance
    - Processes ***pre-declare usage patterns***
    - Request manager avoids “unsafe states”

Assumptions

- Processes pre-declare usage
  - declare ***maximal resource usage***
    - E.g., P1 will never need more than 7 tape drives and 1 printer
- Processes proceed to completion
  - Don't hold onto resources forever
  - Complete in reasonable time
    - Worst-case: ok to stall P2 until P1 finishes

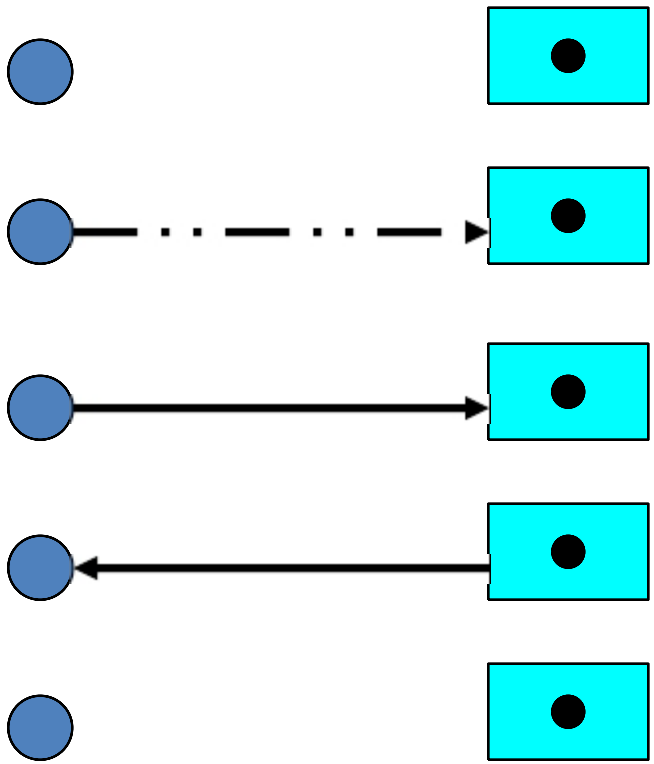
Deadlock Avoidance Algorithm

- Basic idea
  - Single instance for each resource: resource allocation graph
  - Multiple instances for each resource: Banker's algorithm

Resource Allocation Graph

- Used for deadlock avoidance when there is *only one instance* of each resource type.
- Three kinds of edges:
  - ***Claim*** edge Pi –..–> Rj indicates that process Pi ***may*** request resource Rj
  - ***Request*** edge: Pi → Rj
  - ***Assignment*** edge: Rj → Pi

Resource Allocation Graph



Process claims resource

Process requests resource

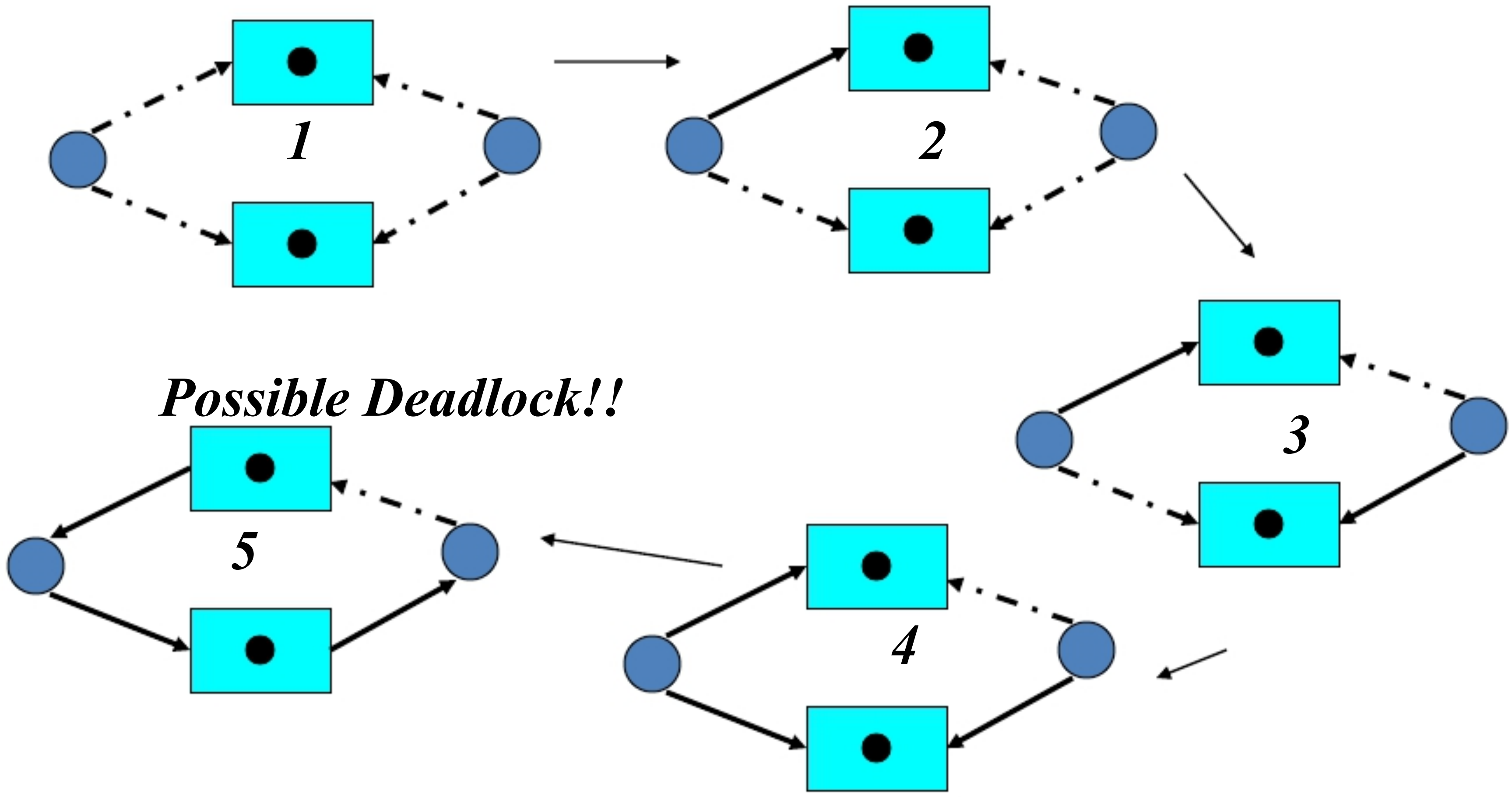
Process is assigned resource

Process releases resource

Resource Allocation Graph (Cont.)

- Resources must be claimed a priori in the system
  - Add claim edges at beginning.
- When a process requests a resource
  - The claim edge converts to the *request* edge (change dotted line to solid line).
  - Check cycles in the graph
    - If resource assignment does not result in the formation of a cycle in the resource allocation graph, safe state.
    - Otherwise, unsafe state.

Resource Allocation Graph



Deadlock Avoidance Algorithm

- Basic idea
  - Single instance for each resource: resource allocation graph
  - Multiple instances for each resource: Banker’s algorithm (can also be applied for single instance per resource type)

Safe Execution Sequence

- P1, P2, P3, ... Pn is a ***safe sequence*** if
  - Every process Pi can be satisfied using
    - currently-free resources; PLUS
    - resources held by P1, P2, ...Pi-1
- Bound Pi's waiting
  - P1 will run to completion, release resources
  - P2 can complete with now-free + P1's + P2's
  - P3 can complete with now-free + P1's + P2's + P3's
- Pi won't wait forever, no wait cycle, no deadlock

Safe State

- System in a ***safe state*** if
  - there exists one safe sequence
- Worst-case behavior
  - Everybody asks for everything at once
  - Follow the safe sequence
- Serial execution is ***worst-case***, not typical
  - Usually execute in parallel

Safe and Unsafe States (1)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Demonstration that the state in (a) is safe.

Safe and Unsafe States (2)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

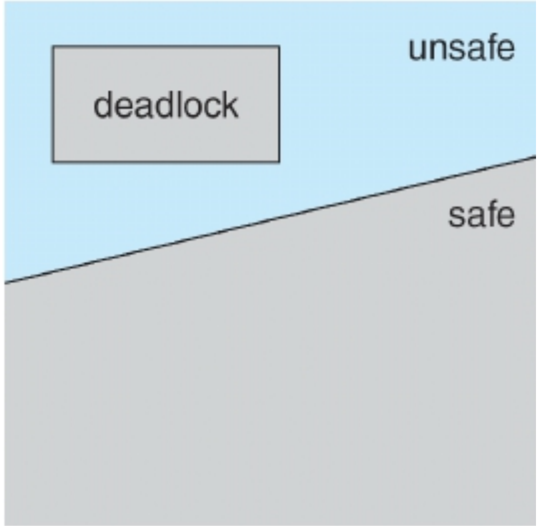
	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Demonstration that the state in (b) is not safe.

Basic Facts

- If a system is in a safe state $\Rightarrow$ no deadlocks.
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.
- Avoidance $\Rightarrow$ ensure that a system will never reach an unsafe state.



Avoidance - Key Ideas

- Safe state
  - Some safe sequence exists
  - Prove it by ***finding one***
- Unsafe state: No safe sequence exists

Deadlock Avoidence
Multiple Instances of resource type

- Banker’s algorithm
- Similar to deadlock detection algorithm
  – Consider each request as it occurs and see if granting it leads to a safe state
  – If a resource granting leads to a deadlock, then the state is unsafe and does not grant
- Key assumptions:
  – All processes need to declare their max resources
  – # of processes and resources are fixed.

Banker's Algorithm for Multiple Resources (1)

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, system will eventually deadlock.
2. Assume the process of row chosen requests all resources needed and finishes. Mark that process as terminated, add its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated (safe state) or no process is left whose resource needs can be met (deadlock)

Banker's Algorithm for Multiple Resources (2)

Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

The banker's algorithm with multiple resources.

Livelock

```
void process_A(void) {
        enter_region(&resource_1);
        enter_region(&resource_2);
        use_both_resources( );
        leave_region(&resource_2);
        leave_region(&resource_1);
}
```

```
void process_B(void) {
        enter_region(&resource_2);
        enter_region(&resource_1);
        use_both_resources( );
        leave_region(&resource_1);
        leave_region(&resource_2);
}
```

Figure 6-16. Busy waiting that can lead to livelock.