Deadlock

William Stallings

Operating Systems (CS-2006)

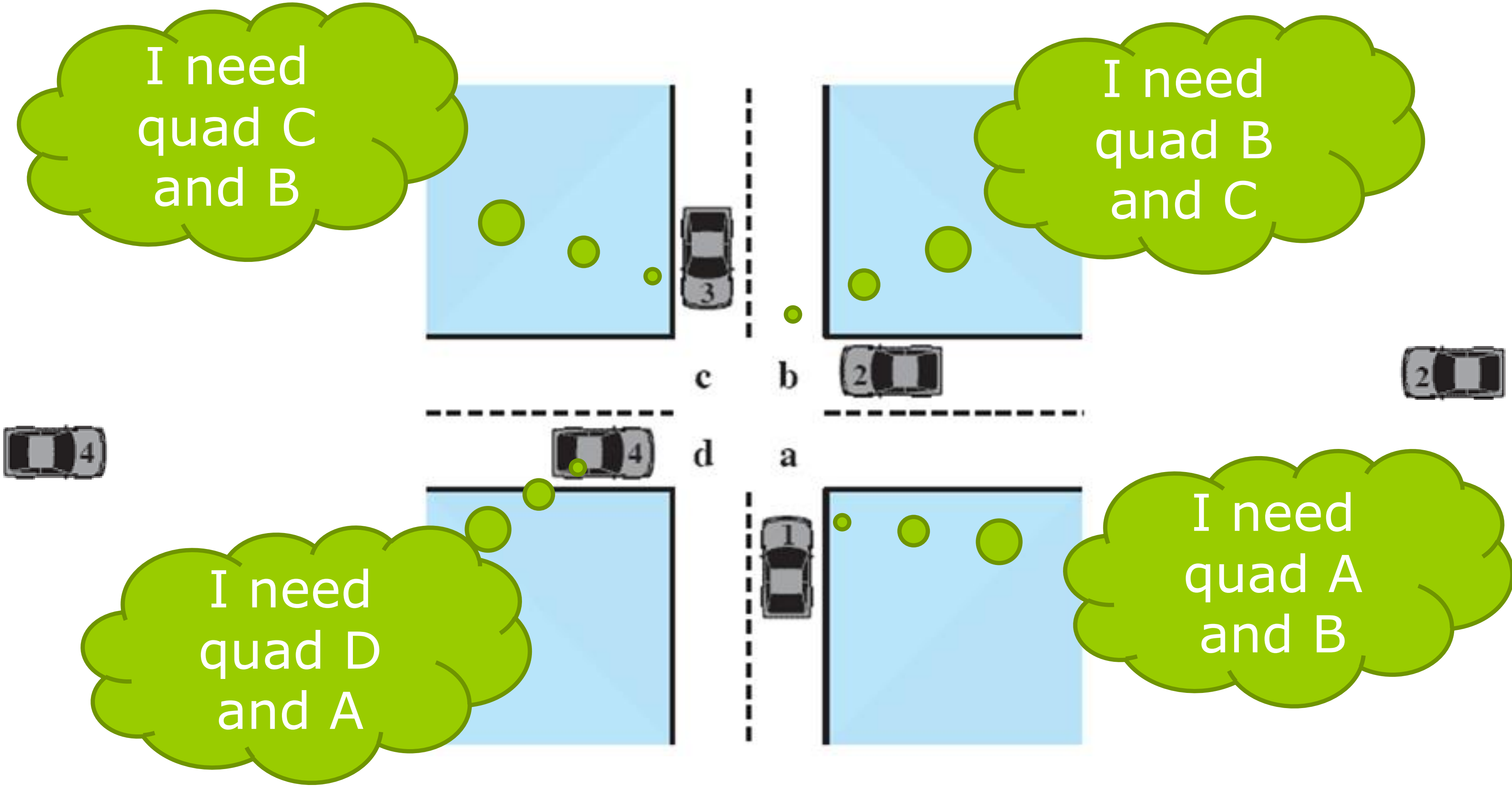
SPRING 2022, FAST NUCES

COURSE SUPERVISOR: ANAUM HAMID
anaum.hamid@nu.edu.pk

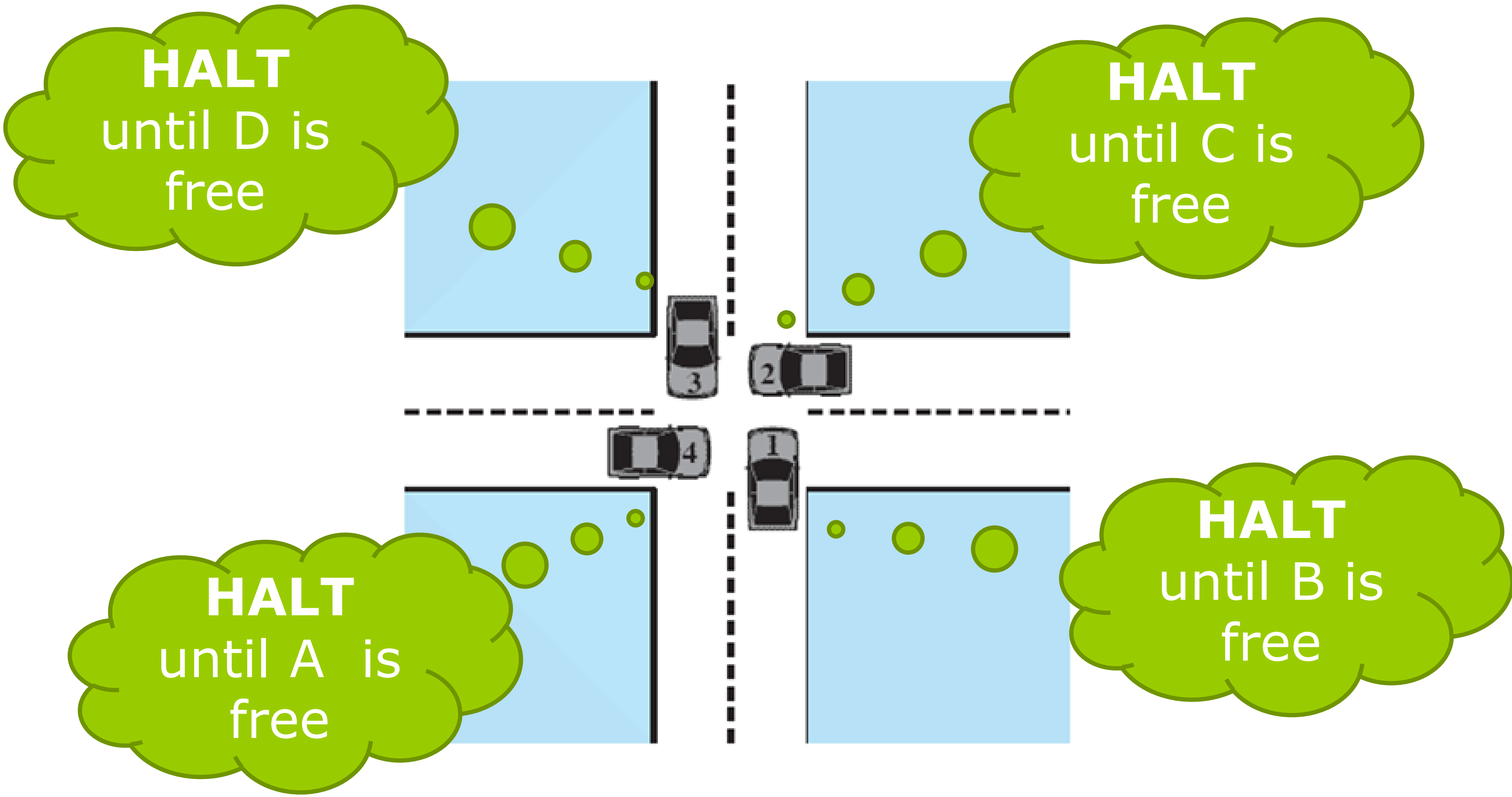
Deadlock

- A set of processes is deadlocked, when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.
  - Typically involves processes competing for the same set of resources
- No efficient solution

Potential Deadlock



Actual Deadlock



Conditions for *possible* Deadlock

- Mutual exclusion (non-sharable resources)
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others
- No pre-emption
  - No resource can be forcibly removed from a process holding it

Actual Deadlock Requires …

All previous 3 conditions plus:

- Circular wait
    - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

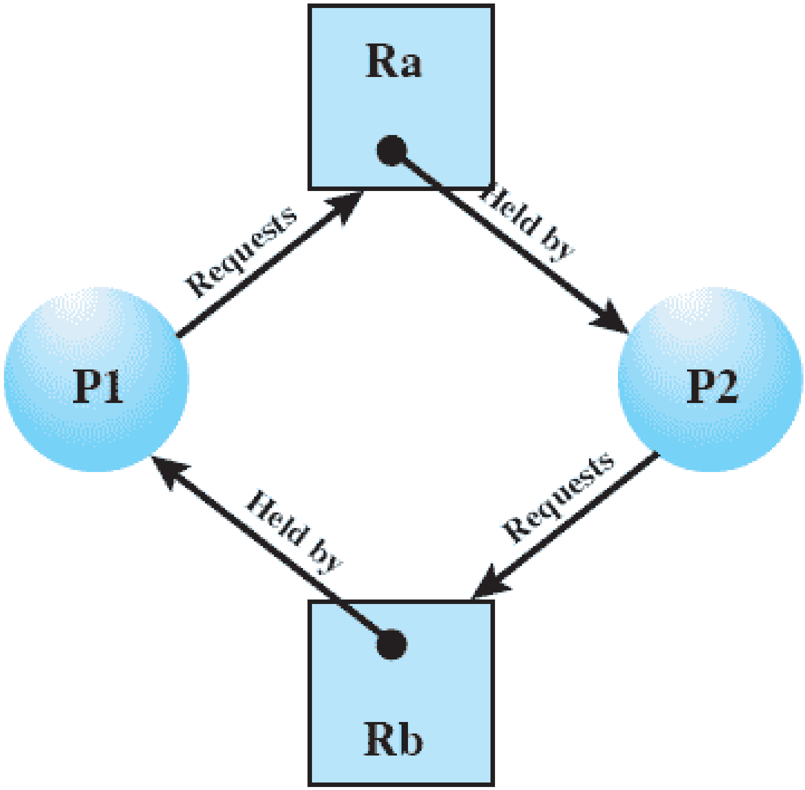
Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes

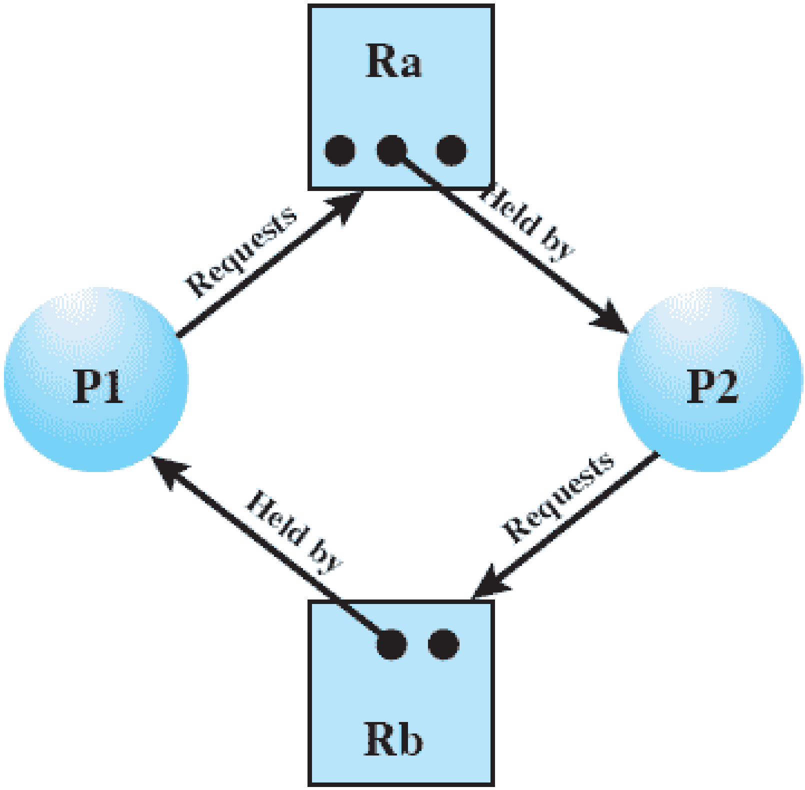
(a) Resouce is requested

(b) Resource is held

Resource Allocation Graphs of deadlock



(c) Circular wait

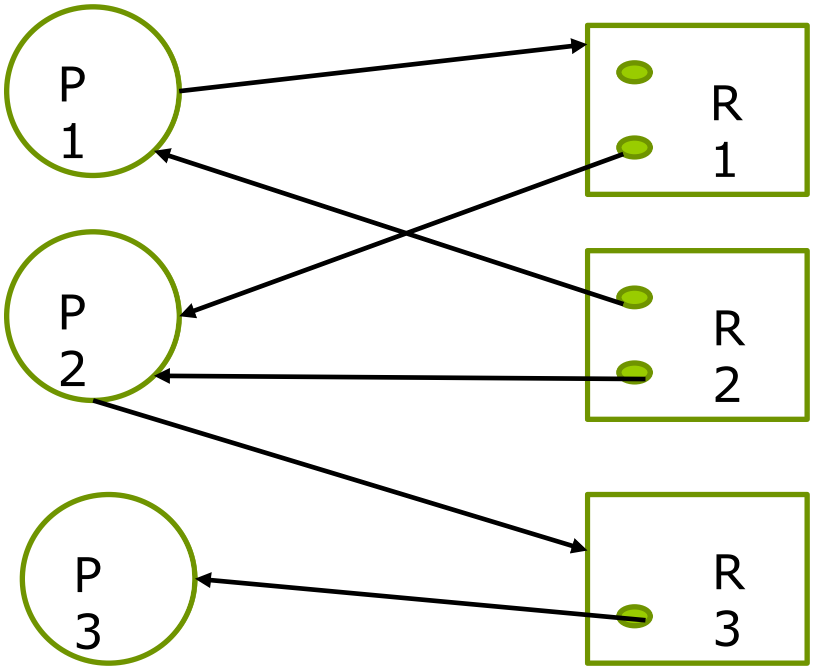


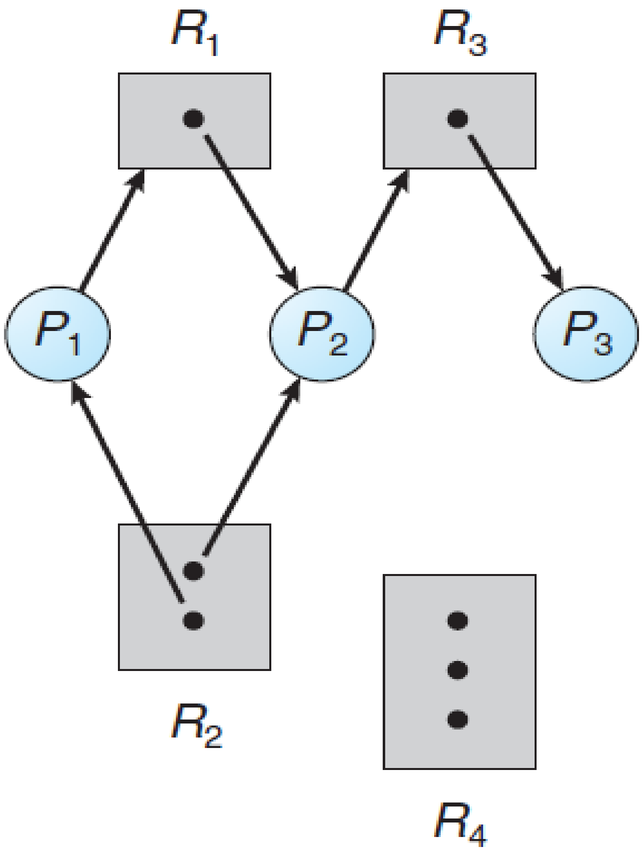
(d) No deadlock

Resource Allocation Graphs of deadlock

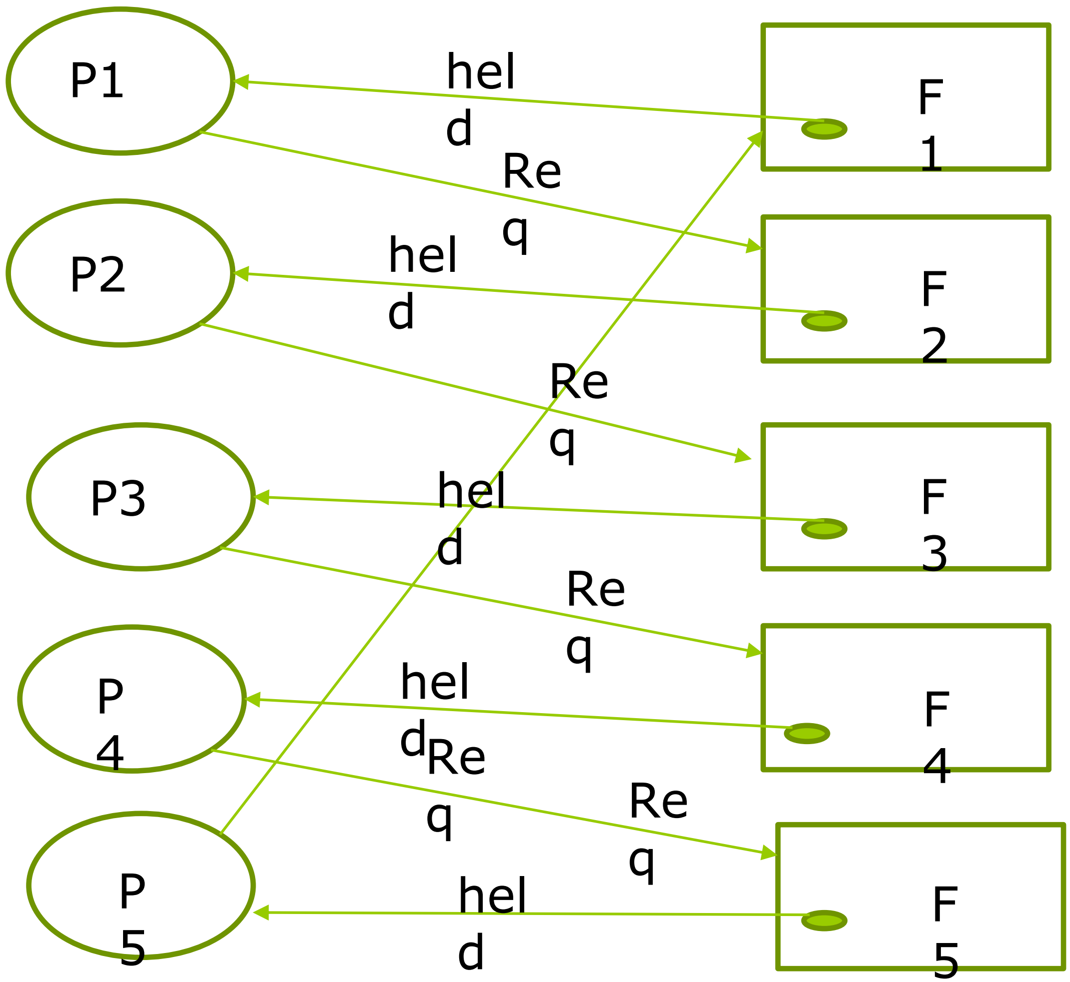
- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**

Resource Allocation Graphs of deadlock





RAG of Dinning Philosopher Problem



Dealing with Deadlock

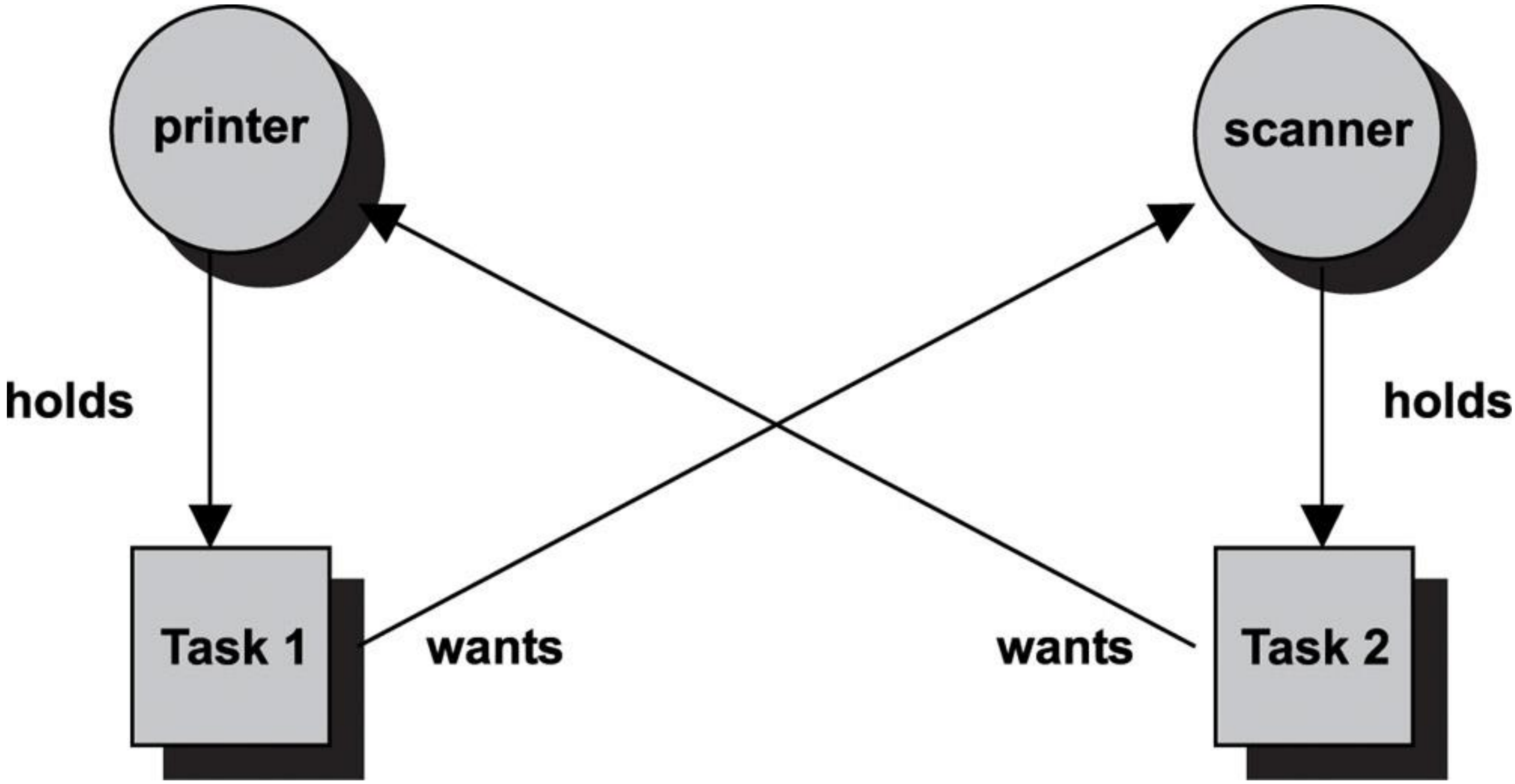
- Three general approaches exist for dealing with deadlock.

    1. Prevent deadlock

    1. Avoid deadlock

    1. Detect Deadlock

Dealing with Deadlock



Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded.
- Two main methods
  - Indirect – prevent one of the three necessary conditions from occurring
  - Direct – prevent circular waits

Deadlock Prevention Conditions

1. Mutual Exclusion

2. Hold and Wait

1. No Preemption

1. Circular Wait

Deadlock Prevention Conditions

Mutual Exclusion

- This condition says, “There exist resources in the system that can be used by only one process at a time.”
- Examples include printer, write access to a file or record, entry into a section of code
- Best not to get rid of this condition
  - some resources are intrinsically nonsharable

Deadlock Prevention Conditions

Hold and Wait (1/2)

- This condition says, "Some process holds one resource while waiting for another."
- To attack the hold and wait condition:
  - Force a process to acquire all the resources it needs before it does anything; if it can't get them all, get none
- Each philosopher tries to get both chopsticks, but if only one is available, put it down and try again later

Deadlock Prevention Conditions

No Preemption (1/2)

- This condition says, "Once a process has a resource, it will not be forced to give it up."
- To attack the no preemption condition:
  - If a process asks for a resource not currently available, block it but also take away all of its other resources
  - Add the preempted resources to the list of resource the blocked process is waiting for

Deadlock Prevention Conditions

Circular Wait (1/2)

- This condition says, "A is blocked waiting for B, B for C, C for D, and D for A"
- Note that the number of processes is actually arbitrary
- To attack the circular wait condition:
  - Assign each resource a priority
  - Make processes acquire resources in priority order

Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests

Two Approaches to Deadlock Avoidance

1. Process Initiation Denial
2. Resource Allocation Denial

Process Initiation Denial

- A process is only started, if the maximum claim of all current processes plus those of the new process can be met.
- Not optimal,
  - Assumes the worst: that all processes will make their maximum claims together.

Resource Allocation Denial

- Referred to as the banker’s algorithm
  - A strategy of resource allocation denial
- Consider a system with fixed number of resources
  1. ***State*** of the system is the current allocation of resources to process
  2. ***Safe state*** is where there is at least one sequence that does not result in deadlock
  3. ***Unsafe state*** is a state that is not safe

Basic Facts for deadlock avoidance

1. If a system is in safe state, ⇒ no deadlocks

2. If a system is in unsafe state ⇒ possibility of deadlock

3. Avoidance ⇒ ensure that a system will never enter an unsafe state.

Banker's Algorithm

1. Multiple instances

2. Each process must a priori claim maximum use

3. When a process requests a resource, it may have to wait

4. When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available

- **Max**: $n \times m$ matrix. If $Max\,[i,j] = k$, then process P_i may request at most k instances of resource type R_j

- **Allocation**: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j

- **Need**: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) ***Need**$_i$* **≤ *Work***

   If no such ***i*** exists, go to step 4

3. ***Work* = *Work* + *Allocation*$_i$**
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Example of Banker's Algorithm

- Discussed in Class

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process P_i. If $Request_i\,[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Example of Resource Request Algorithm

- Discussed in Class

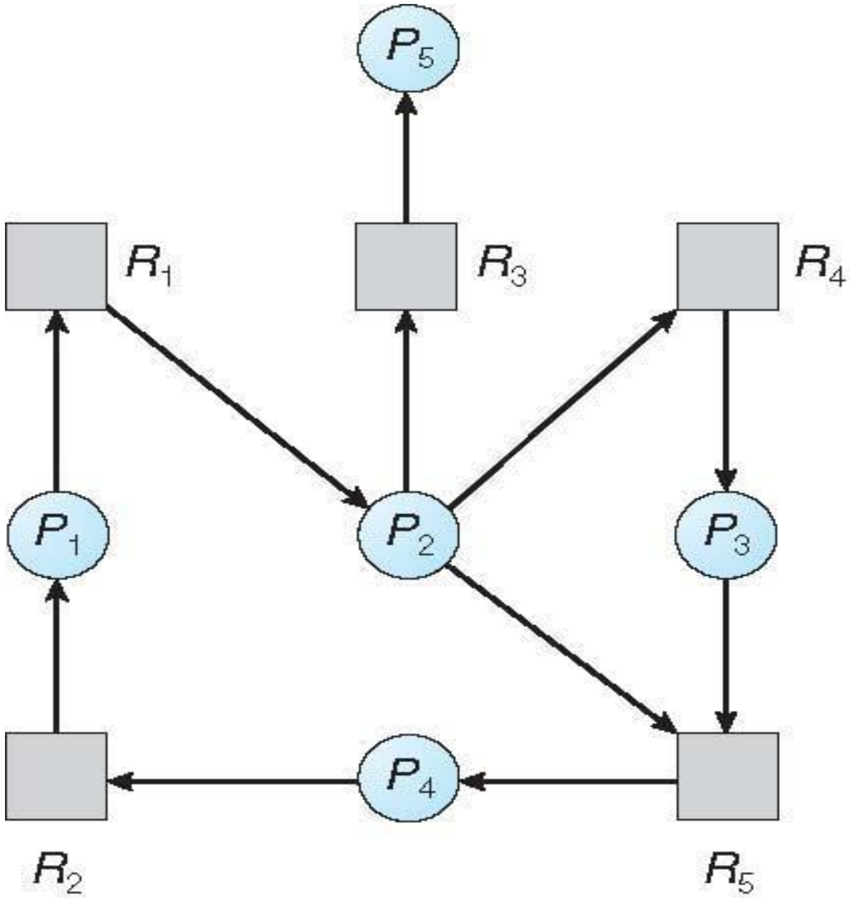
Deadlock Detection

1. Allow system to enter deadlock state
2. Detection algorithm
3. Recovery scheme

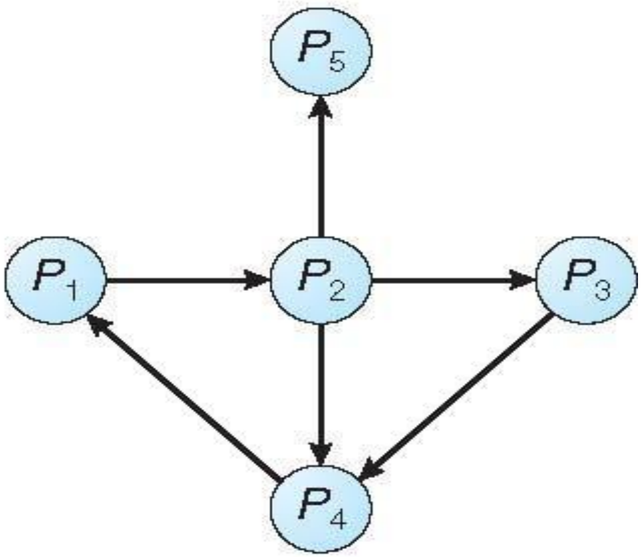
Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:

   (a) ***Work* = *Available***

   (b) For ***i* = 1,2, …, *n***, if $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:

   (a) ***Finish*[*i*] == *false***

   (b) $\boldsymbol{Request_i \leq Work}$

   If no such ***i*** exists, go to step 4

Detection Algorithm (Cont.)

3. ***Work*** **=** ***Work*** **+** ***Allocation$_i$***
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, 1 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P$_i$*** is deadlocked

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	0 0 0	0 0 0	
P_1	2 0 0	2 0 2	3 1 3	
P_2	3 0 3	0 0 0	5 2 4	
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- P_2 requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes

- Abort one process at a time until the deadlock cycle is eliminated

- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

1. **Selecting a victim** – minimize cost

2. **Rollback** – return to some safe state, restart process for that state

3. **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Thank You!