Operating Systems: Deadlocks

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapters 5)

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
  - Eg: You can have two instances of a printer available
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **Release**
- if the resources not available, the process enters a *waiting state*.

Deadlock

- **Deadlock –** when a waiting process has requested a resource held by other waiting processes.
- In a deadlock processes never finish executing
  - System resources are tied up.
  - Thus, preventing other processes from starting.

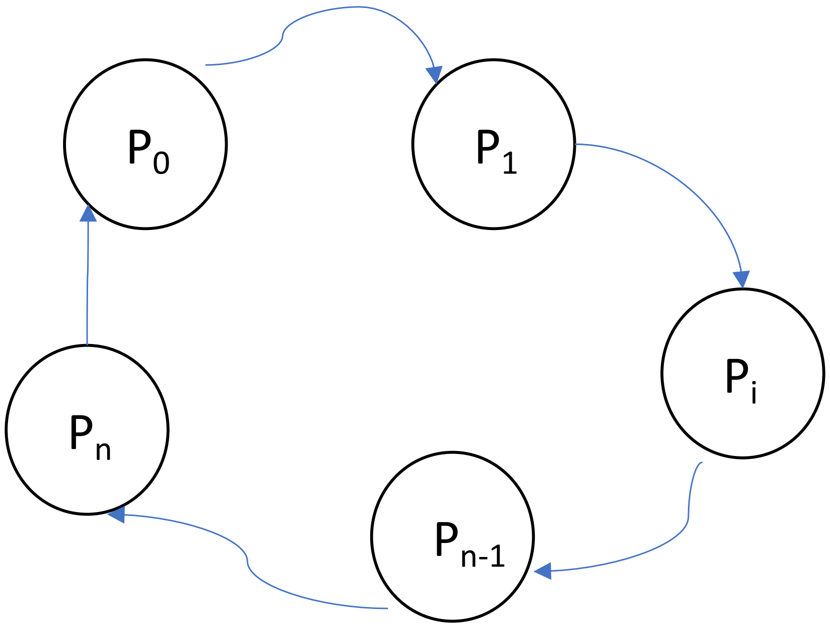
Deadlock Characterization

Deadlock ⇔ below **four conditions hold** at the same time.

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait**: a process holding at least one resource is **waiting to acquire additional resources** held by other processes.
- **No preemption**: a resource can be **released only voluntarily** by the process holding it, after that process finishes its task.

Deadlock Characterization contd..

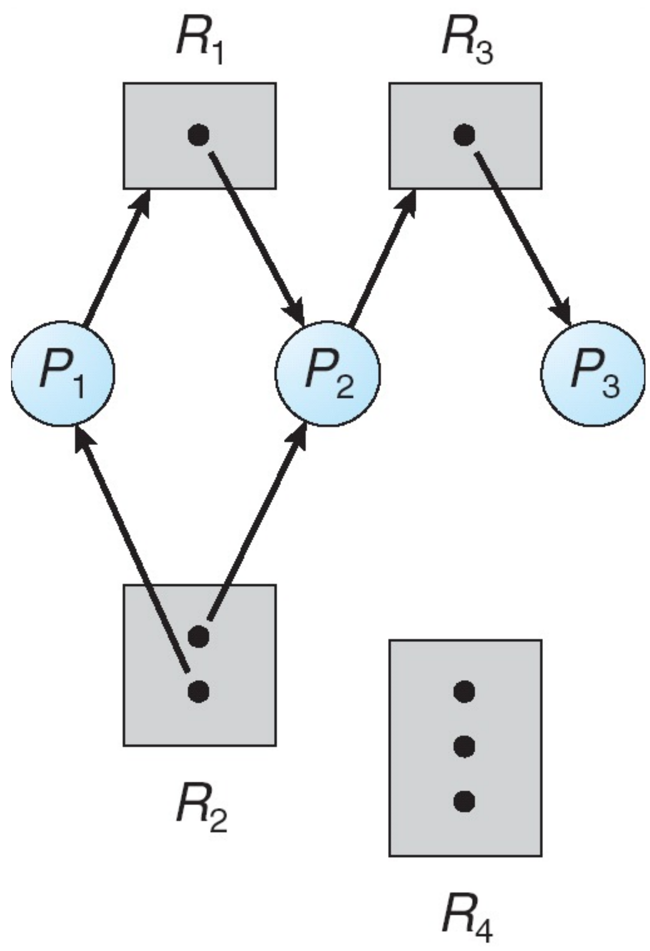
- **Circular wait:** there exists. a set $\{P_0, P_1, \ldots, P_n\}$ of **waiting processes** such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.



circular-wait condition => the hold-and-wait condition.

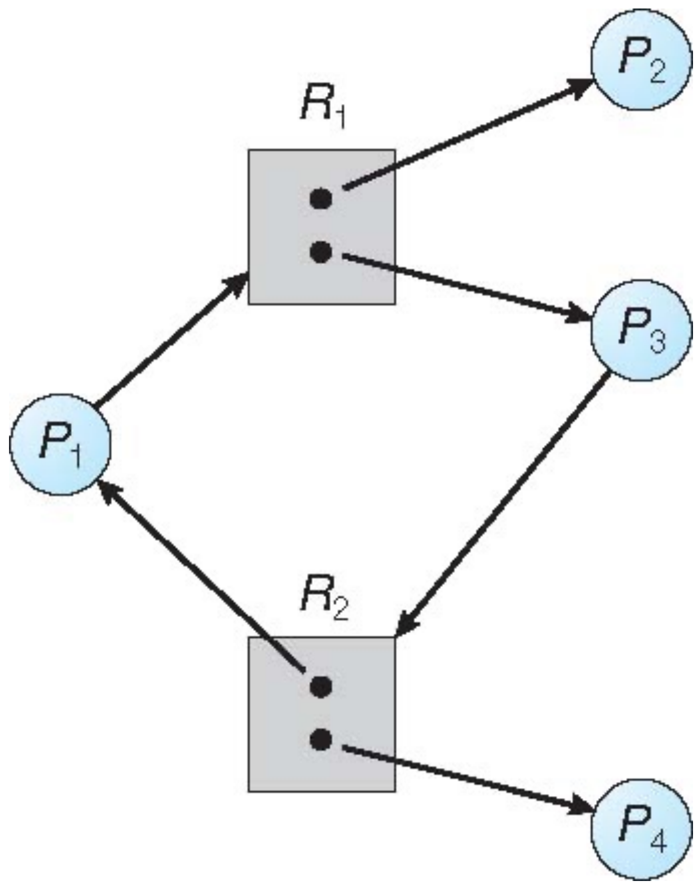
Resource-Allocation Graph

- **System resource-allocation graph** (directed graph G = (V, E)) used to describe deadlocks.
- V is partitioned into two sets:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- E has two types of edges:
  - **Request edge** – directed edge $P_i \rightarrow R_j$: Process P_i requested an instance of resource R_j and is waiting.
  - **Assignment edge** – directed edge $R_j \rightarrow P_i$: Resource R_j assigned to process P_i

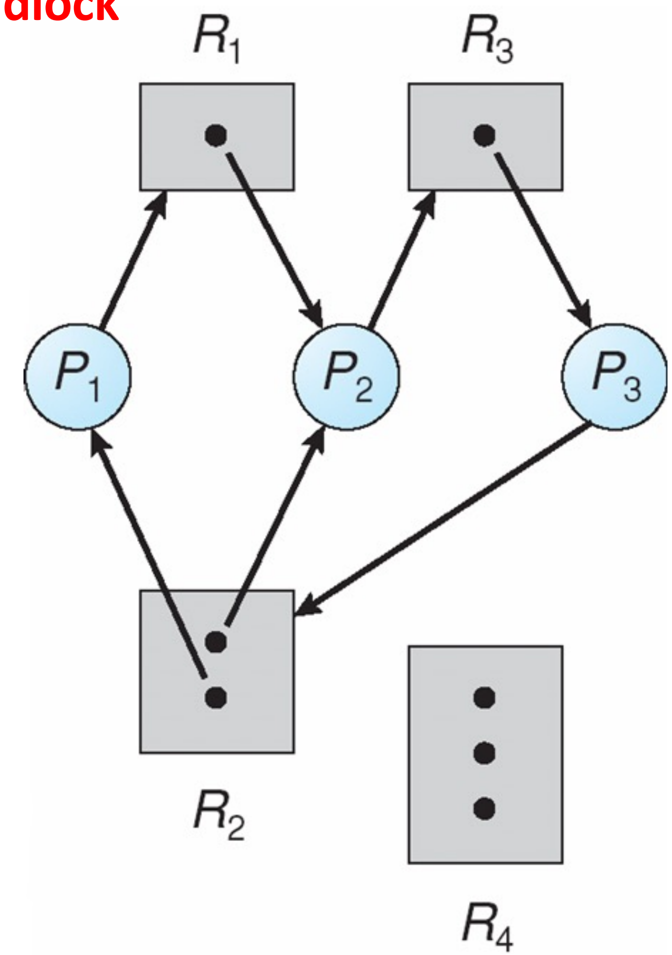


Resource Allocation Graph Examples

Resource allocation graph with a cycle but no Deadlock



Resource allocation graph with a cycle and a Deadlock



Cycles with circular wait condition satisfied

- *P1 → R1 → P2 → R3 → P3 → R2 → P1*
- *P2 → R3 → P3 → R2 → P2*

Basic Facts

- If a graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

Ensure that the system will ***never*** enter a deadlock state:

- **Deadlock prevention:** ensure that atleast one of the necessary conditions (stated in slide #4) cannot hold.
  - Only the **Circular Wait condition is a practical** option for deadlock prevention.
- **Deadlock avoidance:** requires that OS be given **additional information in advance** about the type and no. of resources a process will request. Based on this data OS **decides the waiting strategy for the process**.
- **Deadlock detection:** Allow the system to enter a deadlock state and then recover.
- **IGNORE!** the problem and pretend that deadlocks never occur; used by most operating systems such as UNIX, Linux, and Windows!

Deadlock Avoidance Strategy

- Requires that the OS be given **additional information in advance** about the type and no. of resources a process will request.
- Each process declares its ***maximum need = number* of resources** of each type that the process may need.
- The deadlock-avoidance algorithm dynamically examines the *resource-allocation state* to ensure that there can never be a **circular-wait condition.**

Resource allocation state

- **Resource-allocation state** is defined by
  - Number of available resources
  - Number of allocated resources
  - the maximum need of all the processes in the system.

Example: Consider a system with a single resource type – magnetic tapes= 12.

Resource allocation state of the system at time T_0:

	Maximum Need	Allocated resources	Available
P0	10	5	12-9 = 3
P1	4	2	
P2	9	2	

Safe State

- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i:
  - Resources P_i still needs $\leq$ total available resources + resources held by/allocated to all $P_1, P_2, \ldots, P_{i-1}$.
- To assess whether the system is in a safe state, we compute the needs of a process P_i as below

P_is Need = Max. Need of Process P_i – Allocated resources for process P_i

Safe State Example

Is the below system at time t_0 in safe state?

- Consider a system with 12 magnetic tapes.
- **Resource allocation state** of the system at time t_0:

	Maximum Needs	*Allocated resources*	*Available*
*P*0	10	5	12-9 = 3
*P*1	4	2	
*P*2	9	2	

Safe State Example – with Need vector

- ***P_is Need = Max. Need of Process P_i – Allocated resources for process P_i***
- **Resource allocation state** of the system at time t_0 with need vector:

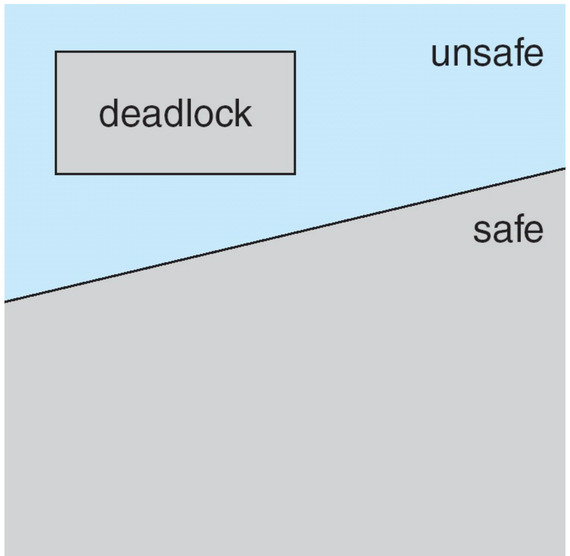
	Maximum Needs	*Allocated resources*	*Need*	*Available*
*P*0	10	5	5	3
*P*1	4	2	2	
*P*2	9	2	7	

Does there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i satisfying the state state requirement?

YES! The sequence is <P_1, P_0, P_2>

Basic Facts

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



Deadlock Avoidance algorithm Outline

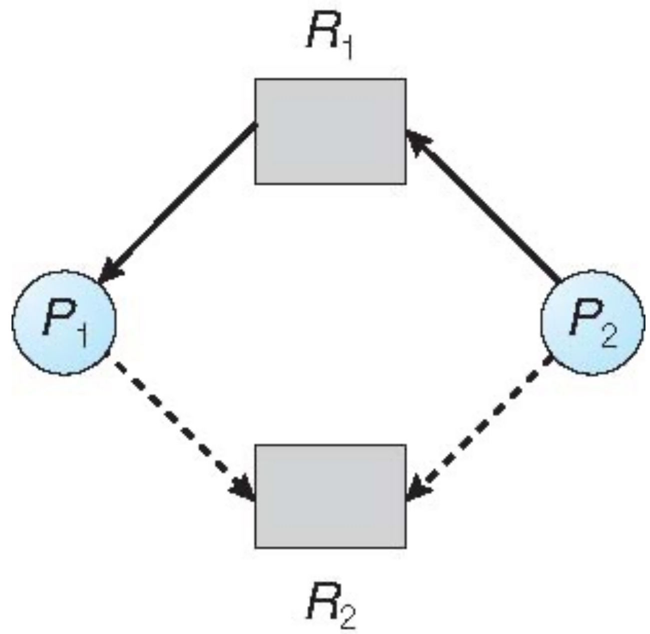
- Given the resource allocation state of a system,
  - The algorithm checks if the system is in a *safe state*.
  - If the system is in a safe state, whenever a process requests an instance of a resource type,
    - It check to see if allocating the resources continues to have the system in a safe state.
      - If yes -- allocate the resources.
      - If no -- have the process wait.

Deadlock Avoidance Algorithms

- Single instance of a resource type
  - Use a **resource-allocation graph**
- Multiple instances of a resource type
  - Use the **Banker's algorithm**

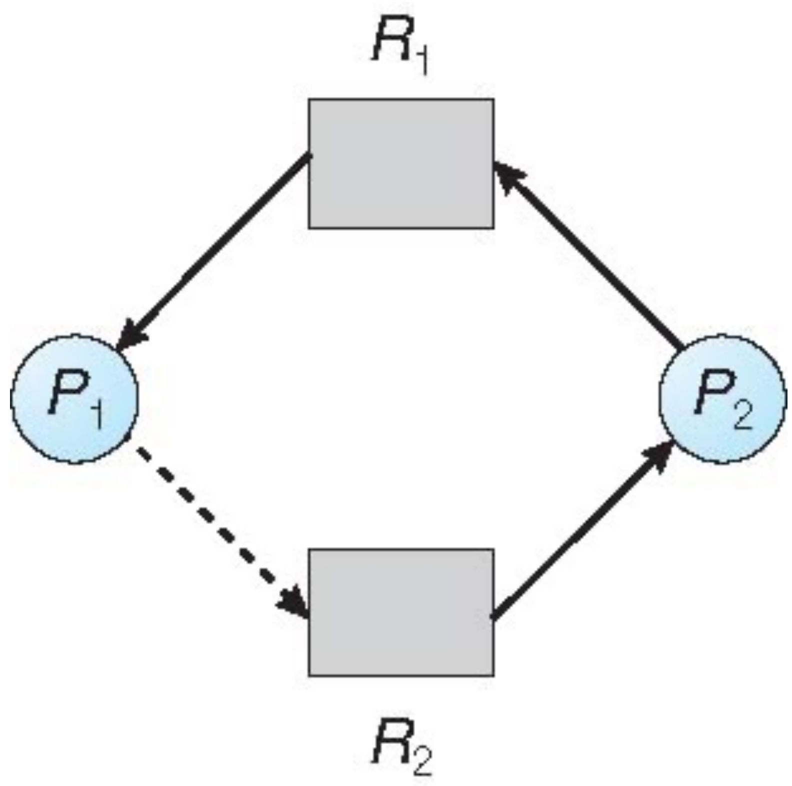
Resource-Allocation Graph Scheme

- Add claim edges to existing resource allocation graph.
- **Claim edge** $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; *represented by a dashed line*
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a **resource is released by a process**, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted **only if** converting the request edge to an assignment edge **does not result in the formation of a cycle** in the resource allocation graph
- We check for safety by using a cycle-detection algorithm.



Unsafe State In Resource-Allocation Graph

Although R_2 is free, we cannot allocate it to P_2 since this will create a cycle!

Banker's Algorithm Outline for *multiple instances* of a resource type

- The algorithm consists of two parts
  - **PART 1 - Safety Algorithm** **–** checks whether a system is in a safe state or not.
  - **PART 2 - Resource-Request Algorithm** – checks to see if resources requested by a process can be satisfied or not.
- Each process must a priori claim maximum use
- When a process gets all its resources it must return them in a *finite amount of time*
- When a process requests a resource, it *may have to wait*

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

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

Example of Data Structures for Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

Resource allocation state at time T_0:

	Max. Need			Allocation			Available		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	3	3	2
P_1	3	2	2	2	0	0			
P_2	9	0	2	3	0	2			
P_3	2	2	2	2	1	1			
P_4	4	3	3	0	0	2			

Need matrix for Banker's Algorithm

The content of the matrix ***Need*** is defined to be ***Max. Need – Allocation***

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

Snapshot at time T_0:

	Max. Need			Allocation			Need			Available		
	A	B	C	A	B	C	**A**	**B**	**C**	A	B	C
P_0	7	5	3	0	1	0	**7**	**4**	**3**	3	3	2
P_1	3	2	2	2	0	0	**1**	**2**	**2**			
P_2	9	0	2	3	0	2	**6**	**0**	**0**			
P_3	2	2	2	2	1	1	**0**	**1**	**1**			
P_4	4	3	3	0	0	2	**4**	**3**	**1**			

Part 1 - Safety Algorithm outline explained with an example

Step 1: Maintain ***Work*** and ***Finish*** vectors of length *m* and *n*, respectively.

Initially:

Work = Available = (3 3 2)

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

False	False	False	False	False

Step 2: Find a process P_i such that both:

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

(b) $Need_i \leq Work$

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

- P_1 satisfies conditions (a) and (b).

Part 1 - Safety Algorithm outline explained with an example – contd…

Step 3:

Work = Work + Allocation$_i$

Finish[i] = true

Work = Work + Allocation$_1$ = (3 3 2) + (2 0 0) = (5 3 2)

Finish[1] = true and Finish =

False	True	False	False	False

go to step 2

Next, we see that process P3, P4, P2, and P0 all satisfy the conditions in step 2.

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

Finish **=**

True	True	True	True	True

Therefore, the system is in a safe state and the sequence of processes satisfying the safety requirement is -

<P$_1$, P$_3$, P$_4$, P$_2$, P$_0$>

Part – 2 Resource-Request Algorithm for Process P_i

- ***Request**$_i$* = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$
- If $\boldsymbol{Request_i \le Need_i}$ go to step 2. *Otherwise, raise error condition*, since process has exceeded its maximum claim
- If $\boldsymbol{Request_i \le Available}$, go to step 3. *Otherwise,* $\boldsymbol{P_i}$ *must wait*, since resources are not available
- Pretend to allocate requested resources to $\boldsymbol{P_i}$ and update the system state as follows:

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

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

PART 2 - Resource-Request Algorithm Explained with an Example

- **P_1 requests resources (1 0 2)**
- Check if **$Request_1$** $\leq$ **$Need_1$**
  - $(1\ 0\ 2) \leq (1\ 2\ 2) \Rightarrow$ true
- Check if **$Request_1$** $\leq$ **Available**
  - $(1\ 0\ 2) \leq (3\ 2\ 2) \Rightarrow$ true
- Pretend that resources requested have be granted.
- Update system state. Max need, $Allocation_1$ and $Need_1$ data structures
  - $Available_1 = (3\ 2\ 2) - (1\ 0\ 2) = (2\ 2\ 0)$
  - $Allocation_1 = (2\ 0\ 0) + (1\ 0\ 2) = (3\ 0\ 2)$
  - $Need_1 = (1\ 2\ 2) - (1\ 0\ 2) = (0\ 2\ 0)$

PART 2 - Resource-Request Algorithm Explained with an Example contd…

Updated resource allocation state:

	Max. Need			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	**2**	**2**	**0**
P_1	3	2	2	**3**	**0**	**2**	**0**	**2**	**0**			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

PART 2 - Resource-Request Algorithm Explained with an Example contd…

- Run safety algorithm on the **updated resource allocation state**.
- System is in safe state and the sequence of processes satisfying the safety requirement is **<P_1, P_3, P_4, P_2, P_0>**

Updated Resource allocation state after request has been granted for P_1

	Max. Need			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3	**2**	**2**	**0**
P_1	3	2	2	**3**	**0**	**2**	**0**	**2**	**0**			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

Resource-Request Algorithm Example – Cont…

- When system in this state, can request for (3 3 0) by P_4 be granted?
  - Check if $\textbf{Request}_4 \le \textbf{Available}$
    - $(3\ 3\ 0) \le (\ 2\ 2\ 0) \Rightarrow$ false
    - The request cannot be granted.
- When system in this state, can request for (0 2 0) by P_0 be granted?
  - Check if $\textbf{Request}_0 \le \textbf{Available}$
    - $(0\ 2\ 0) \le (\ 2\ 2\ 0) \Rightarrow$ true
  - Check if $\textbf{Request}_0 \le \textbf{Need}_0$
    - $(0\ 2\ 0) \le (\ 7\ 4\ 3) \Rightarrow$ true
  - Pretend to grant the resources requested

Resource-Request Algorithm Example – Cont…

Updated Resource allocation state

	Max. Need			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	**0**	**3**	**0**	**7**	**2**	**3**	**2**	**0**	**0**
P_1	3	2	2	3	0	2	0	2	0			
P_2	9	0	2	3	0	2	6	0	0			
P_3	2	2	2	2	1	1	0	1	1			
P_4	4	3	3	0	0	2	4	3	1			

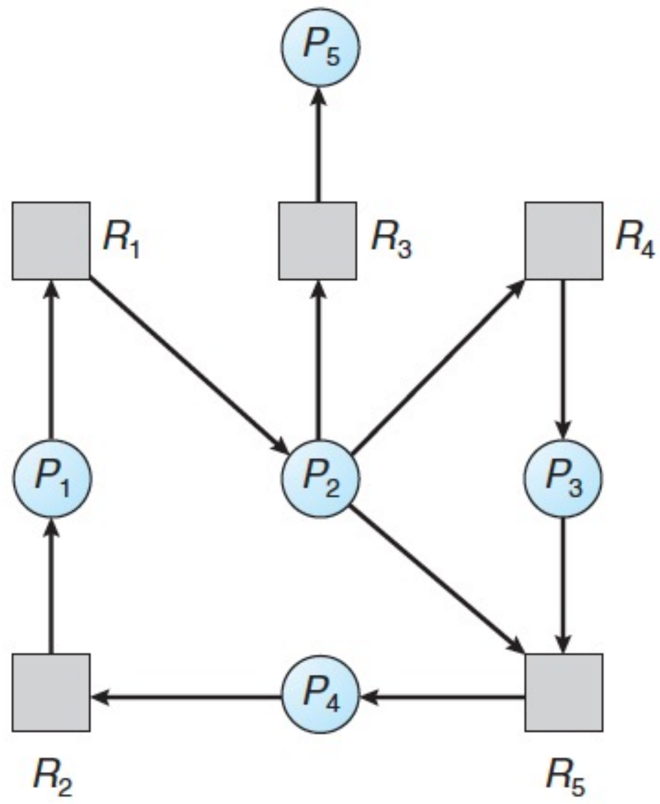
- However, since no sequence of processes exist satisfying the safe state requirement, this request cannot be granted as doing so will leave the system in an unsafe state.

Deadlock Detection

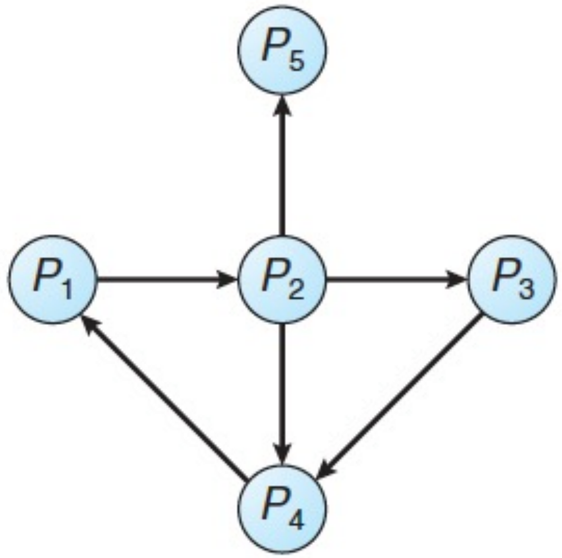
- Allow system to enter deadlock state
- Use deadlock detection algorithm to check if a deadlock exists
- If deadlock exists, use a recovery scheme to recover from the deadlock

Deadlock Detection - Single Instance of Each Resource Type

- A variant of the resource-allocation graph if all resources have only **a single instance** - used for deadlock detection
- Nodes are processes, and an edge $\boldsymbol{P_i \rightarrow P_j}$ in the wait for graph implies that $\boldsymbol{P_i}$ is waiting for $\boldsymbol{P_j}$ to release a resource that P_j needs.



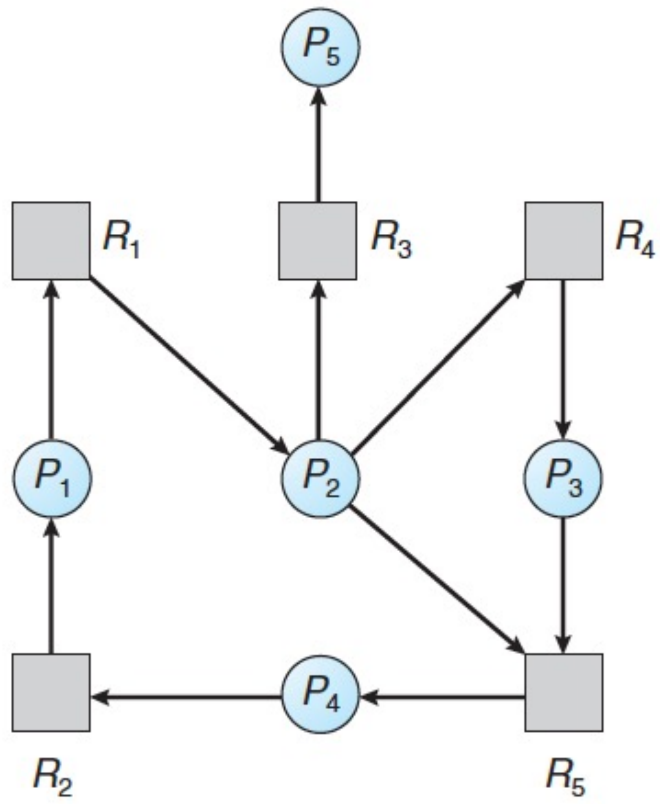
Resource-Allocation Graph



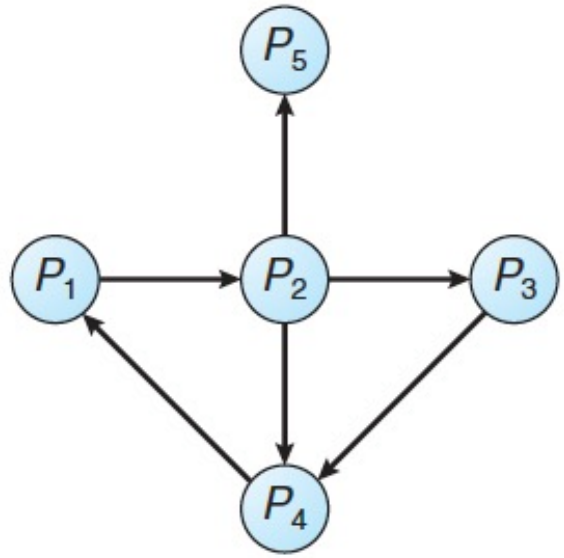
Corresponding wait-for graph

Deadlock Detection - Single Instance of Each Resource Type

If a cycle exists in the wait-for graph, then the system is in deadlock.



Resource-Allocation Graph



Corresponding wait-for graph

Deadlock Detection Algorithm for Multiple Instances of a Resource Type Outline

- The algorithms needs to know
  - The number of *available resources* for each resource type.
  - The number of *allocated resources* for each resource type.
  - The number of *requested resources* by all processes in the system.
- Given the above,
  - The deadlock detection algorithm checks whether the system is ***in a deadlocked*** state or not.
  - If the system is in a deadlocked state, then the algorithm also identifies the processes involved in the deadlock.

Deadlock 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***,

   i. if $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***;

   ii. 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

Items in red highlight the differences in Deadlock detection algorithm and the safety algorithm described under Banker's algorithm.

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 for Deadlock Detection Algorithm

- Five processes $\mathbf{P_0}$ through $\mathbf{P_4}$
- Three resource types

  A (7 instances), B (2 instances), and C (6 instances)
- Snapshot of the system at time $\mathbf{T_0}$:

	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	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Example of Detection Algorithm Cont…

Step 1:

1. Work = Available = (0 0 0)
2. Finish =

False	False	False	False	False

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

1. Finish[i] == false
2. $Request_i \leq Work$

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

P_0 satisfies the above two conditions.

Example of Detection Algorithm Cont…

- Step 3:
  - **Work = Work + Allocation$_0$ = (0 0 0) + (0 1 0) = (0 1 0)**

    Finish[1] = true

    go to step 2

- We see that process P_2, P_3, P_1, and P_4 all satisfy the conditions in step 2.

- Finally, in **Step 4: Finish =**

True	True	True	True	True

- Therefore, the system is ***not in a deadlocked state*,** as the following sequence of processes results in all values of the **Finish** vector to be **True**:

  <P_0, P_2, P_3, P_1, P_4>

Example of Detection Algorithm Cont…

- *Suppose P_2* requests an additional instance of type *C.*
- *Then below is the updated snapshot of the system including this request:*

	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	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Can reclaim resources held by process $\boldsymbol{P_0}$, but insufficient resources to fulfill other processes' requests
- **Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$**

Recovery from Deadlock

- **Process Termination**
  - Abort all deadlocked processes
  - Abort one process at a time until the deadlock cycle is eliminated

- **Resource Preemption**
  - **Selecting a victim** – minimize cost
  - **Rollback** – return to some safe state, restart process for that state
  - **Starvation** – same process may always be picked as victim.

    Possible solution - include number of rollback in cost factor