Deadlocks

System Model

- System consists of finite number of resources
- Resource types $R_1, R_2, \ldots, R_m$

  Physical: printers, tape drives, CPU cycles, memory space, I/O devices

  Logical: files, semaphores, monitors
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - Release
- Request and release are system calls
- Deadlock – two or more processes are waiting indefinitely for an event (resource acquisition and release) that can be caused by only one of the waiting processes
- Multithreaded programs are good candidates for deadlock because multiple threads compete for shared resources

Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** at least 1 resource must be in non-sharable mode, only one process at a time can use a resource. Other process must be delayed until release

- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

- **No preemption:** resources cannot be preempted. A resource can be released only voluntarily by the process holding it, after that process has completed its task

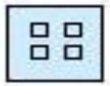
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is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

Resource-Allocation Graph

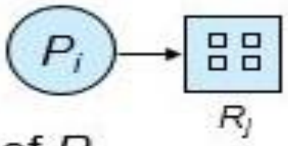
- Used to describe deadlocks more precisely in terms of a directed graph
- Consists of set of vertices V and a set of edges E.
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

- Process

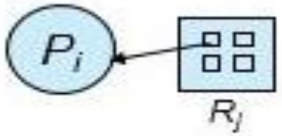
- Resource Type with 4 instances



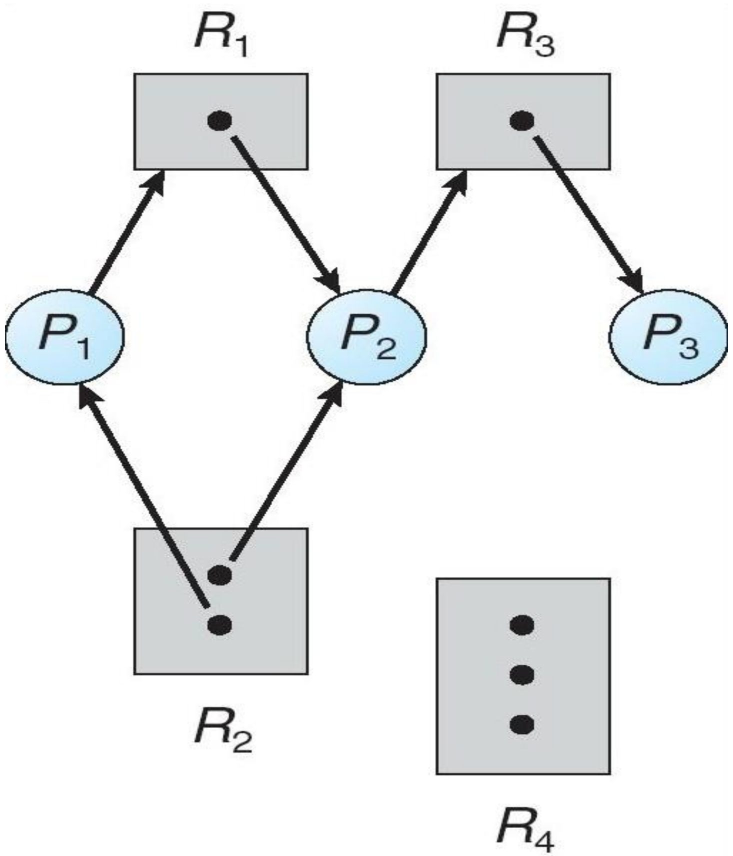
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

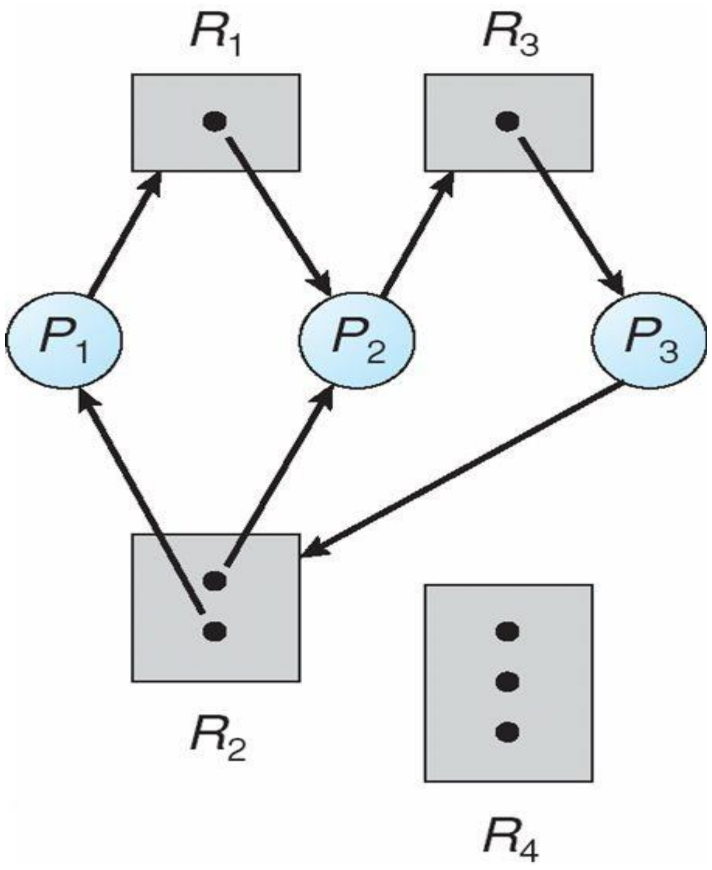


- Given the definition of RAG, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked
- If graph contains cycle, deadlock may exists

Basic Facts

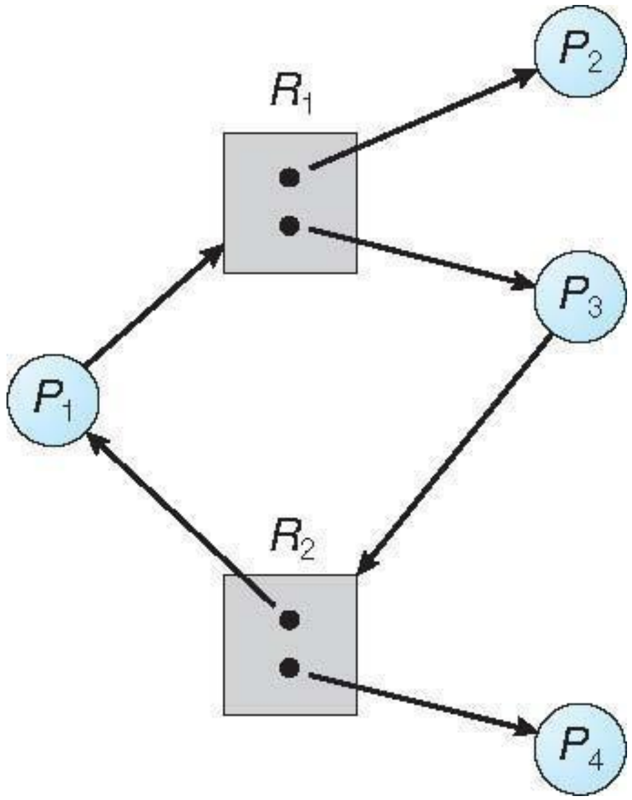
- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock (cycle is necessary and sufficient condition for existence of deadlock)
  - if several instances per resource type, possibility of deadlock (cycle is necessary but not sufficient condition for existence of deadlock)

Resource Allocation Graph With A Deadlock



- 2 cycles:
  - P1-R1-P2-R3-P3-R2-P1
  - P2-R3-P3-R2-P2
- **P2 is waiting** for R3, which is held by P3**. P3 is waiting** for P1/ P2 to release R2
- In addition**, P1 is waiting** for P2 to release R1

Graph With A Cycle But No Deadlock



- P1-R1-P3-R2-P1 (cycle but no deadlock)
- P4 may release its instance of R2 which can be allocated to P3 breaking the cycle

Methods for Handling Deadlocks

- Deal with deadlocks in 1 of the 3 ways:
  - Ensure that the system will *never* enter a deadlock state:
    - **Deadlock prevention-** set of methods ensuring at least 1 of the necessary conditions cannot hold
    - **Deadlock avoidance-** OS to be given additional information about resources a process will request and use during its lifetime depending on which it can decide whether or not a process should wait
  - Allow the system to enter a deadlock state and then **recover**
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX and windows

Deadlock Prevention

- For deadlock to occur 4 necessary conditions must hold. If 1 of them doesn’t we can prevent the occurrence of a deadlock
- **Mutual Exclusion (ME)** –
  - must hold for non-sharable resources (printer)
  - Sharable resources (e.g., read-only files) do not require ME and thus cannot be involved in a deadlock
  - However, deadlocks cannot be prevented by denying mutual exclusion as some resources are intrinsically non-sharable
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Disadvantages:
    - Low resource utilization-resources are allocated but unused for a longer time
    - starvation possible: wait indefinitely

- **No Preemption** –
  - To ensure this condition does not hold use the following protocol:
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted (implicitly released)
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

  F:R->N (one to one function)

  1.A process can initially request any no. of instances of Ri. Process can request instance of Rj iff

  F(Rj)>F(Ri)

  2. When a process request Rj, it should release Ri such that F(Ri)>=F(Rj)

  3.If several instances of same resource are needed, a single request for all of them must be issued

  If 1, 2 are used, circular wait cannot hold

- Eg: let F(tape drive)=1, F (disk drive)=5, F(printer)=12

1. Process wants to use printer and tape drive at the same time should request tape drive 1st and then printer bcz F(printer)>F(tape)
2. If process wants to use disk it should release printer bcz F(printer)>=F(disk)

Deadlock Avoidance

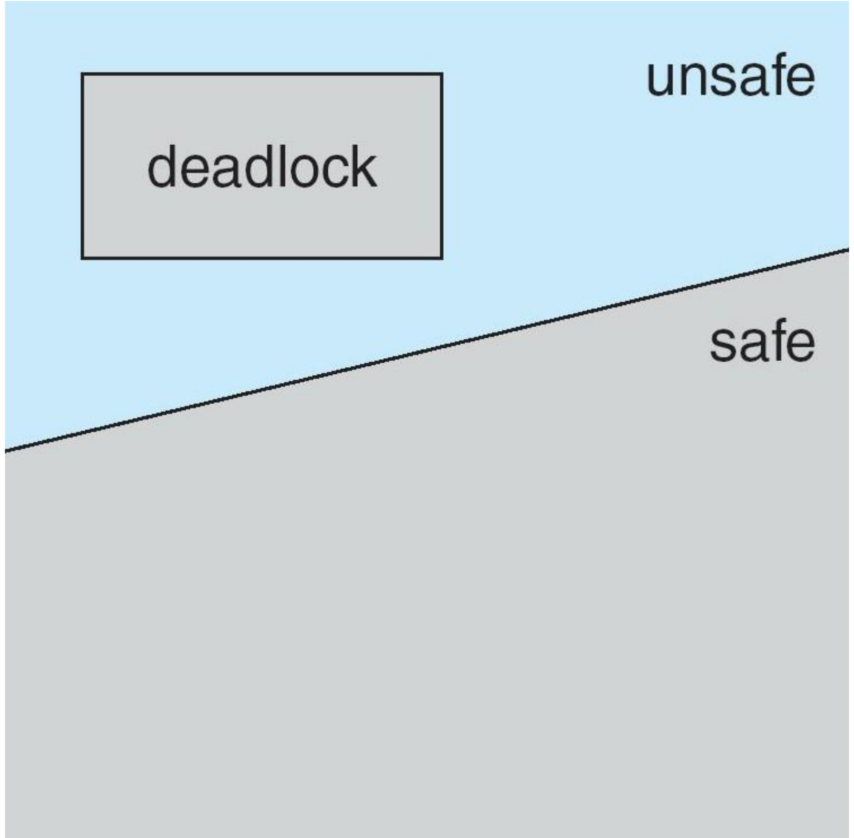
- Requires that the system has some additional ***a priori*** information available
- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock
- A system is in safe state only if there exists a safe sequence
- System is in **safe state** if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- Deadlock state=>unsafe
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.



Safe, Unsafe, Deadlock State

Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

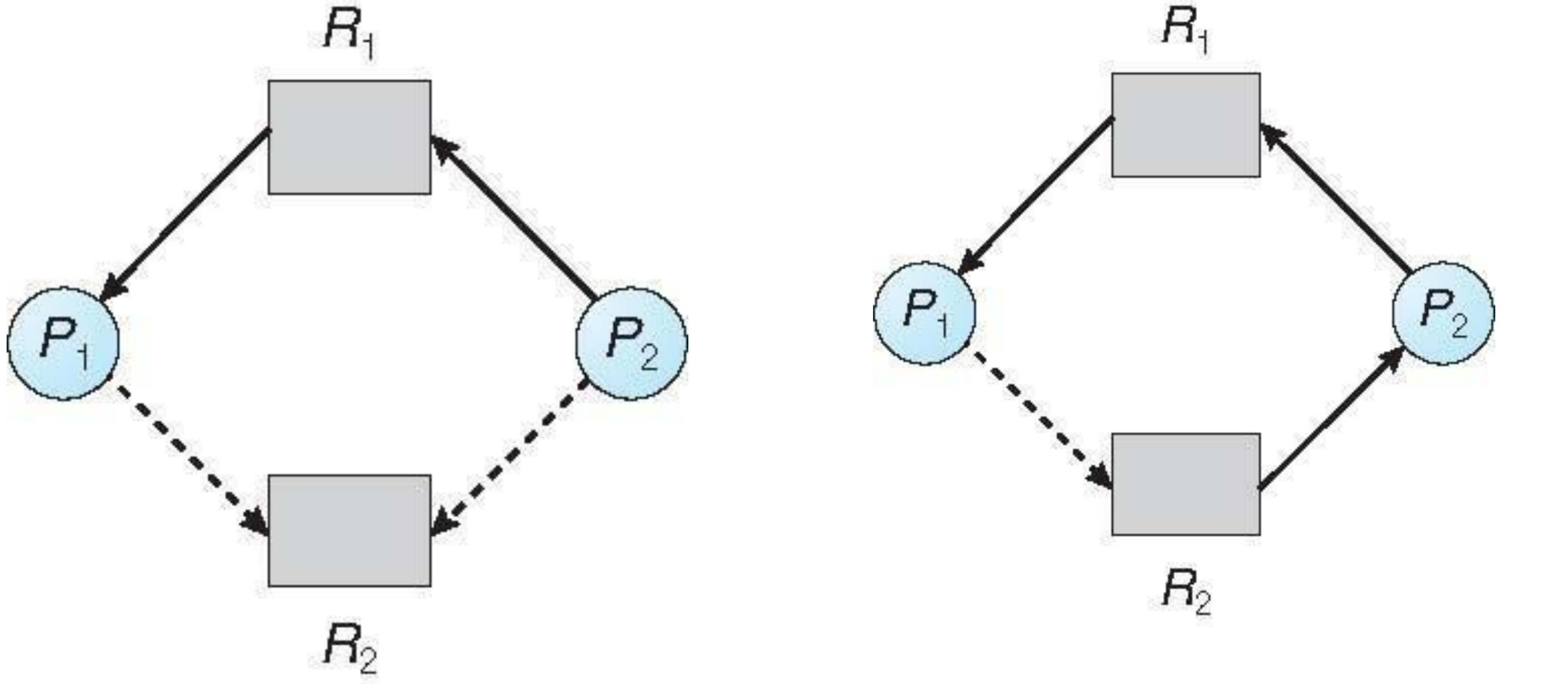
- Multiple instances of a resource type
  - Use the banker’s algorithm

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j 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. Before Pi starts its execution , all claim edges must be ready in RAG.
- A new claim edge may be added later only if all edges associates with pi are claim edges

- Suppose Pi requests Rj. Request will be granted only if converting request edge (Pi-Rj) to an assignment edge (Rj-Pi) does not result in the formation of a cycle in RAG
- If no cycle->safe state
- Cycle->unsafe
- Pi will have to wait for its request to be satisfied

Claim Edge and Assigned Edge



Bankers Algorithm

- Multiple instances of each resource type (RAG is not applicable)
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- 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 x 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

Resource-Request Algorithm for Process P_i

***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 Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

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

- Snapshot at time T_0:

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

The system is in a safe state since the sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety criteria

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

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

P_1 Request (1,0,2)

- **Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true**

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

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

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

- request for (3,3,0) by $\boldsymbol{P_4}$ cannot be granted since resources are not available
  - (3,3,0)<=(4,3,1)
  - (3,3,0)>=(2,3,0)

- Request for (0,2,0) by P0 cannot be granted, even though resources are available, since the resulting state is unsafe

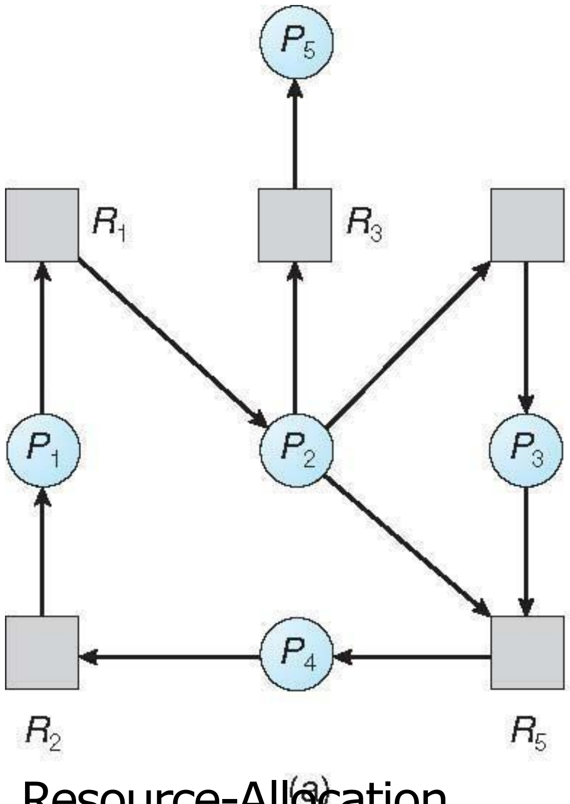
Deadlock Detection

- If there is no deadlock prevention/ avoidance algorithm, then a deadlock situation may occur. In such cases system must provide:
  - Detection algorithm
  - Recovery scheme

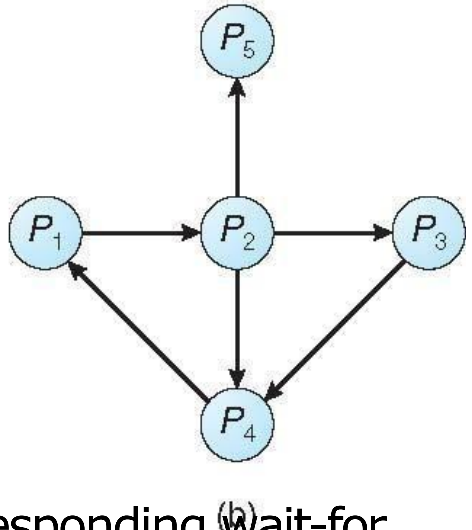
Single Instance of Each Resource Type

- Maintain **wait-for** graph (obtained by removing resource nodes and collapsing the edges)
  - Nodes are processes
  - $P_i \rightarrow P_j$ **(**P_i is waiting for P_j to release a resource that P_i **needs)**
- An edge $P_i \rightarrow P_j$ exist in WFG if and only if the corresponding RAG contains 2 edges $P_i \rightarrow R_q$ **and** $R_q \rightarrow P_j$
- 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



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- Wait for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. Foll. algorithm is used for deadlock detection

- **Available***:* A vector of length ***m*** indicates the number of available resources of each type

- **Allocation***:* An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process

- **Request***:* An ***n* x *m*** matrix indicates the current request of each process. If ***Request* [*i*][*j*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

Deadlock Detection Algorithm

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

   (a) Work = Available

   For $\boldsymbol{i = 1,2, \ldots, n}$, 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) $\boldsymbol{Request_i \leq Work}$

   (c) If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$
   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 $\boldsymbol{P_i}$ is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

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	
P_2	3 0 3	0 0 0	
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***

- $\boldsymbol{P_2}$ 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

- Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
- Deadlocks occur when request cannot be granted immediately. Invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
  - Invoking Deadlock detection algorithm for every resource request incurs overhead in computation time.
- Invoke the algorithm at defined intervals
  - Once/ hr, when CPU utilization drops below 40%

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

- Successively pre-empt some resources from processes and give to other processes until the deadlock cycle is broken
- 3 issues to be addressed:
  - Selecting a victim: which resources and processes are to be pre-empted?
    - Determine the order of pre-emption to minimize cost
  - Rollback: if resource is pre-empted from a process, what should be done with that process?
    - It cannot continue with normal execution
    - Rollback the process to some safe state , restart process for that state
    - Determining safe state is difficult, do total rollback (abort the process and restart it)

- **Starvation –**
  - How to ensure no starvation? (resource should not be preempted from same process)
  - same process may always be picked as victim-this process never completes its designated task
  - Therefore ensure that a process can be picked as a victim only a finite number of times.
  - Common solution is to include number of rollback in cost factor