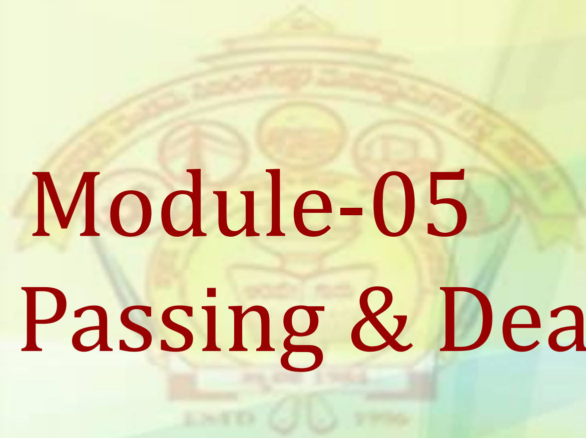
Department of Electronics & Communication Engg.

Course : Operating Systems -15EC553. **Sem.: 5th (2018-19 ODD)**

Course Coordinator:

Prof. Nyamatulla M Patel

Operating Systems-15EC553



Module-05
Message Passing & Deadlocks

Introduction

- What is a Deadlock?
- Deadlocks in Resource Allocation
- Handling Deadlocks
- Deadlock Detection and Resolution
- Deadlock Prevention
- Deadlock Avoidance
- Characterization of Resource Deadlocks by Graph Models
- Deadlock Handling in Practice

What is a Deadlock?

Definition 8.1 Deadlock A situation involving a set of processes D in which each process P_i in D satisfies two conditions:

1. Process P_i is blocked on some event e_j.
2. Event e_j can be caused only by actions of other process(es) in D.

- *Resource deadlock* → primary concern of OS

Process P_i	*Process P_j*
Request tape drive; Request printer; Use tape drive and printer; Release printer; Release tape drive;	Request printer; Request tape drive; Use tape drive and printer; Release tape drive; Release printer;

  - P_i, P_j are deadlocked after their second requests

- Deadlocks can also arise in synchronization and message communication → user concern

Deadlocks in Resource Allocation

- OS may contain several resources of a kind
  - *Resource unit* refers to a resource of a specific kind
  - *Resource class* to refers to the collection of all resource units of a kind
- Resource allocation in a system entails three kinds of events:
  - *Request* for the resource
  - Actual *allocation* of the resource
  - *Release* of the resource
    - Released resource can be allocated to another process

Deadlocks in Resource Allocation (continued)

Table 8.1 Events Related to Resource Allocation

Event	Description
Request	A process requests a resource through a system call. If the resource is free, the kernel allocates it to the process immediately; otherwise, it changes the state of the process to *blocked*.
Allocation	The process becomes the *holder* of the resource allocated to it. The resource state information is updated and the state of the process is changed to *ready*.
Release	A process releases a resource through a system call. If some processes are blocked on the allocation event for the resource, the kernel uses some tie-breaking rule, e.g., FCFS allocation, to decide which process should be allocated the resource.

Conditions for a Resource Deadlock

Table 8.2 Conditions for Resource Deadlock

Condition	Explanation
Nonshareable resources	Resources cannot be shared; a process needs exclusive access to a resource.
No preemption	A resource cannot be preempted from one process and allocated to another process.
Hold-and-wait	A process continues to hold the resources allocated to it while waiting for other resources.
Circular waits	A circular chain of hold-and-wait conditions exists in the system; e.g., process P_i waits for P_j, P_j waits for P_k, and P_k waits for P_i.

- Another condition is also essential for deadlocks:
  - *No withdrawal of resource requests:* A process blocked on a resource request cannot withdraw it

Modelling the Resource Allocation State

- *(Resource) allocation state:*
  - Information about resources allocated to processes and about pending resource requests
  - Used to determine whether a set of processes is deadlocked
- Two kinds of models are used to represent the allocation state of a system:
  - A *graph model*
  - A *matrix model*

Resource request and allocation graph (RRAG)

- Nodes and edges in an RRAG
  - Two kinds of *nodes* exist in an RRAG
    - A circle is a process
    - A rectangle is a resource class
      - Each bullet in a rectangle is one resource unit
  - *Edges* can also be of two kinds
    - An edge from a resource class to a process is a resource allocation
    - An edge from a process to a resource class is a pending resource request

Wait-for graph (WFG)

- A WFG can be used to depict the resource state of a system in which every resource class contains only one resource unit
  - A *Node* in the graph is a process
  - An *edge* is a *wait-for* relationship between processes
    - A wait-for edge (P_i, P_j) indicates that
      - Process P_j holds the resource unit of a resource class
      - Process P_i has requested the resource and it has become blocked on it
      - In essence P_i waits for P_j to release the resource

Graph Models

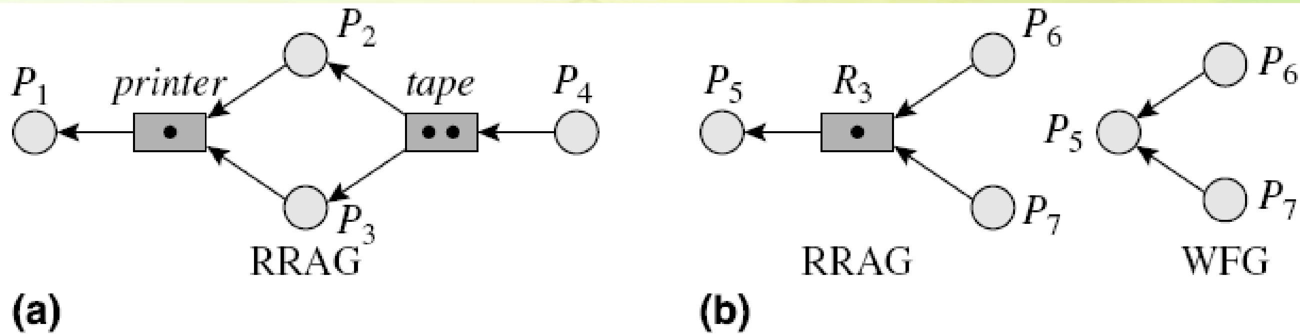


Figure 8.1 (a) Resource request and allocation graph (RRAG); (b) Equivalence of RRAG and wait-for graph (WFG) when each resource class contains only one resource unit.

Paths in WFG and RRAG

- A *path* in a graph is a sequence of edges such that the destination node of an edge is the source node of the subsequent edge
  - Consider an RRAG path $P_1 - R_1 - P_2 - R_2 \ldots P_{n-1} - R_{n-1} - P_n$

    This path indicates that

    - Process P_n has been allocated a resource unit of R_{n-1}
    - Process P_{n-1} has been allocated a resource unit of R_{n-2} and awaits a resource unit of R_{n-1}, etc.
  - In WFG, the same path would be $P_1 - P_2 - \ldots P_{n-1} - P_n$

Graph Models (continued)

- A deadlock cannot exist unless an RRAG, or a WFG, contains a cycle
- A cycle in an RRAG does not necessarily imply a deadlock if a resource class has multiple resource units

When P_k completes, its tape unit can be allocated to P_j

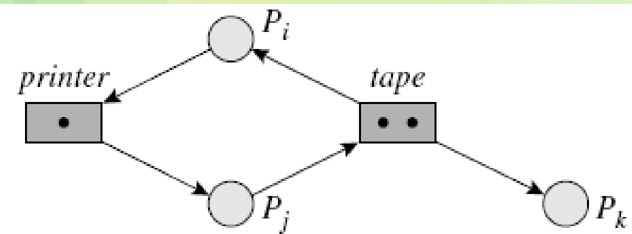


Figure 8.3 RRAG after all requests of Example 8.4 are made.

Matrix Model

- Allocation state represented by two matrices:
  - *Allocated_resources*
  - *Requested_resources*
- If system has *n* processes and *r* resource classes, each of these matrices is an $n \times r$ matrix

Allocated resources

	Printer	Tape
P_i	0	1
P_j	1	0
P_k	0	1

Requested resources

	Printer	Tape
P_i	1	0
P_j	0	1
P_k	0	0

	Printer	Tape
Total resources	1	2
Free resources	0	0

- Auxiliary: *Total_resources* and *Free_resources*

Handling Deadlocks

Table 8.3 Deadlock Handling Approaches

Approach	Description
Deadlock detection and resolution	The kernel analyzes the resource state to check whether a deadlock exists. If so, it aborts some process(es) and allocates the resources held by them to other processes so that the deadlock ceases to exist.
Deadlock prevention	The kernel uses a resource allocation policy that ensures that the four conditions for resource deadlocks mentioned in Table 8.2 do not arise simultaneously. It makes deadlocks impossible.
Deadlock avoidance	The kernel analyzes the allocation state to determine whether granting a resource request can lead to a deadlock in the future. Only requests that cannot lead to a deadlock are granted, others are kept pending until they can be granted. Thus, deadlocks do not arise.

Deadlock Detection and Resolution

- A *blocked* process is not currently involved in a deadlock if request on which it is blocked can be satisfied through a sequence of process completion, resource release, and resource allocation events
  - If each resource class in system contains a single resource unit, check for a cycle in RRAG or WFG
    - Not applicable if resource classes have multiple resource units
  - We will use matrix model
    - Applicable in all situations

Example: Deadlock Detection

- The allocation state of a system containing 10 units of a resource class R_1 and three processes:

	R_1
P_1	4
P_2	4
P_3	2

Allocated resources

	R_1
P_1	6
P_2	2
P_3	0

Requested resources

	R_1
Total resources	10
Free resources	0

- Process P_3 is in the *running* state
  - We simulate its completion
    - Allocate its resources to P_2
  - All processes can complete in this manner
    - No *blocked* processes exist when the simulation ends
      - Hence no deadlock

A Deadlock Detection Algorithm

Algorithm 8.1 *Deadlock Detection*

Inputs

n	:	Number of processes;
r	:	Number of resource classes;
Blocked	:	set of processes;
Running	:	set of processes;
Free_resources	:	**array** $[1..r]$ **of** *integer*;
Allocated_resources	:	**array** $[1..n, 1..r]$ **of** *integer*;
Requested_resources	:	**array** $[1..n, 1..r]$ **of** *integer*;

Data structures

Finished	:	**set of** *processes*;

1. **repeat until** *set Running is empty*
   - **a.** *Select a process* P_i *from set Running*;
   - **b.** *Delete* P_i *from set Running and add it to set Finished*;
   - **c.** **for** $k = 1..r$
     $Free_resources[k] := Free_resources[k] + Allocated_resources[i,k]$;
   - **d.** **while** *set Blocked contains a process* P_l *such that*
     for $k = 1..r$, $Requested_resources[l,k] \leq Free_resources[k]$
     - **i.** **for** $k = 1, r$
       $Free_resources[k] := Free_resources[k] - Requested_resources[l, k]$;
       $Allocated_resources[l, k] := Allocated_resources[l, k] + Requested_resources[l, k]$;
     - **ii.** *Delete* P_l *from set Blocked and add it to set Running*;
2. **if** *set Blocked is not empty* **then**
   declare processes in set Blocked to be deadlocked.

Example: Operation of a Deadlock Detection Algorithm

Allocated resources

	R_1	R_2	R_3
P_1	2	1	0
P_2	1	3	1
P_3	0	1	1
P_4	1	2	2

Requested resources

	R_1	R_2	R_3
P_1	2	1	3
P_2	1	4	0
P_3			
P_4	1	0	2

	R_1	R_2	R_3
Total resources	5	7	5
Free resources	1	0	1

(a) Initial state

	Allocated R_1	Allocated R_2	Allocated R_3	Requested R_1	Requested R_2	Requested R_3
P_1	2	1	0	2	1	3
P_2	1	3	1	1	4	0
P_3	1	1	1			
P_4	1	2	2	1	0	2

Free resources (R_1 R_2 R_3): 0 0 1

(b) After simulating allocation of resources to P_4 when process P_3 completes

	Allocated R_1	Allocated R_2	Allocated R_3	Requested R_1	Requested R_2	Requested R_3
P_1	2	1	0	2	1	3
P_2	1	3	1	1	4	0
P_3	0	0	0			
P_4	2	2	4			

Free resources: 0 1 0

(c) After simulating allocation of resources to P_1 when process P_4 completes

	Allocated R_1	Allocated R_2	Allocated R_3	Requested R_1	Requested R_2	Requested R_3
P_1	4	2	3			
P_2	1	3	1	1	4	0
P_3	0	0	0			
P_4	0	0	0			

Free resources: 0 2 1

(d) After simulating allocation of resources to P_2 when process P_1 completes

	Allocated R_1	Allocated R_2	Allocated R_3	Requested R_1	Requested R_2	Requested R_3
P_1	0	0	0			
P_2	2	7	1			
P_3	0	0	0			
P_4	0	0	0			

Free resources: 3 0 4

Figure 8.4 Operation of Algorithm 8.1, the deadlock detection algorithm.

Deadlock Resolution

- Deadlock resolution for a set of deadlocked processes *D* is breaking of deadlock to ensure progress for some processes in *D*
  - Achieved by aborting one or more processes in *D*
    - Each aborted process is called a *victim*
      - Choice of victim made using criteria such as process priority, resources consumed by it, etc.

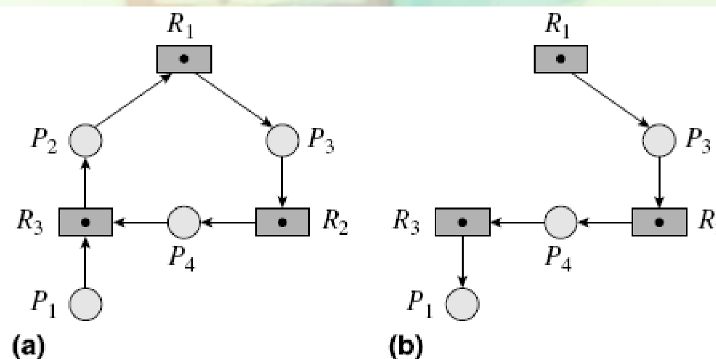


Figure 8.5 Deadlock resolution. (a) a deadlock; (b) resource allocation state after deadlock resolution.

Deadlock Prevention

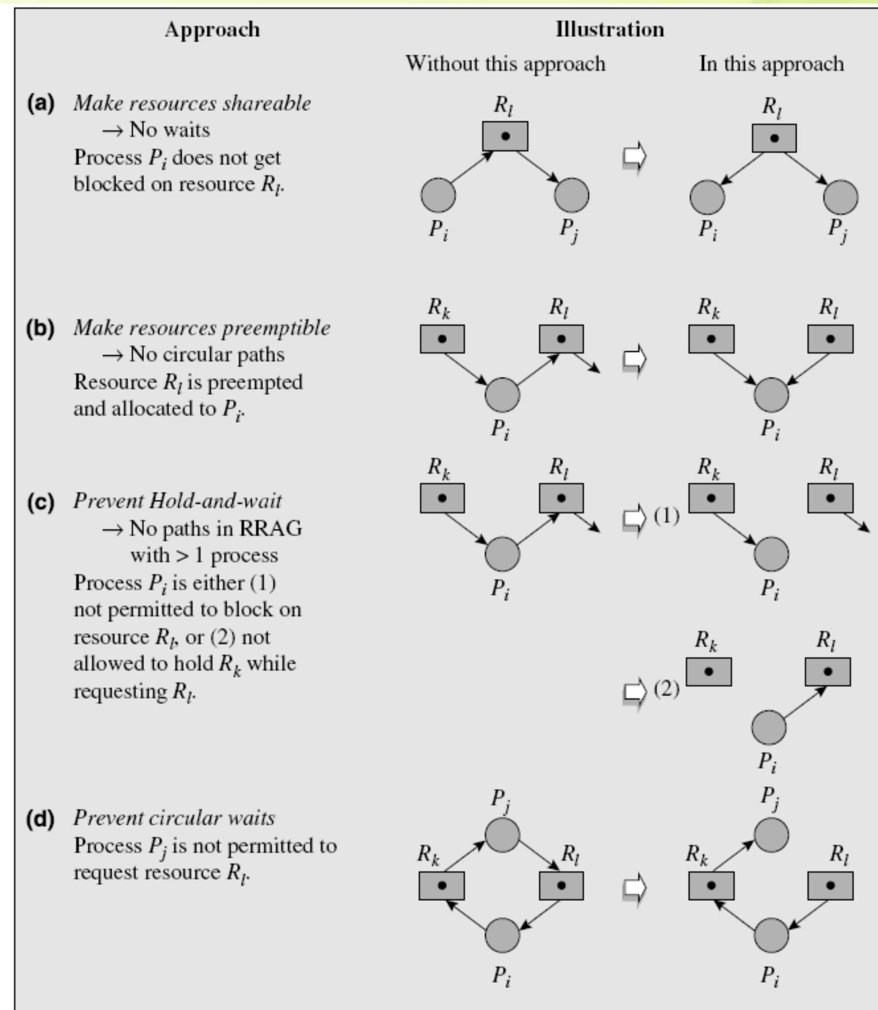


Figure 8.6 Approaches to deadlock prevention.

All Resources Together

- Simplest of all deadlock prevention policies
- Process must ask for all resources it needs in a single request
  - Kernel allocates all of them together
    - A blocked process does not hold any resources
      - Hold-and-wait condition is never satisfied
- Attractive policy for small operating systems
- Has one practical drawback:
  - Adversely influences resource efficiency

Resource Ranking

- *Resource rank* associated with each resource class
- Upon resource request, kernel applies a validity constraint to decide if it should be considered
  - Rank of requested resource must be larger than rank of highest ranked resource allocated to the process
- Result: absence of circular wait-for relationships
- Works best when all processes require their resources in the order of increasing resource rank
  - In worst case, policy may degenerate into the "all resources together" policy of resource allocation

Deadlock Avoidance

- Banker's algorithm
  - Analogy: bankers admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments
  - Uses notion of a *safe allocation state*
    - When system is in such a state, all processes can complete their operation without possibility of a deadlock
  - Deadlock avoidance implemented by taking system from one safe allocation state to another

Deadlock Avoidance

Table 8.4 Notation Used in the Banker's Algorithm

Notation	Explanation
$Requested_resources_{j,k}$	Number of units of resource class R_k currently requested by process P_j
$Max_need_{j,k}$	Maximum number of units of resource class R_k that may be needed by process P_j
$Allocated_resources_{j,k}$	Number of units of resource class R_k allocated to process P_j
$Total_alloc_k$	Total number of allocated units of resource class R_k, i.e., Σ_j $Allocated_resources_{j,k}$
$Total_resources_k$	Total number of units of resource class R_k existing in the system

Definition 8.2 Safe Allocation State An allocation state in which it is possible to construct a sequence of process completion, resource release, and resource allocation events through which each process P_j in the system can obtain $Max_need_{j,k}$ resources for each resource class R_k and complete its operation.

Deadlock Avoidance (continued)

- Outline of the approach:

1. When a process makes a request, compute *projected allocation state*
   - This would be the state if the request is granted
2. If projected allocation state is safe, grant request by updating *Allocated_resources* and *Total_alloc*; otherwise, keep request pending
   - Safety is checked through simulation
   - A process is assumed to complete only if it can get its maximum requirement of each resource satisfied simultaneously
3. When a process releases any resource(*s*) or completes its operation, examine pending requests and allocate those that would put the system in a new safe allocation state

Example: Banker's Algorithm for a Single Resource Class

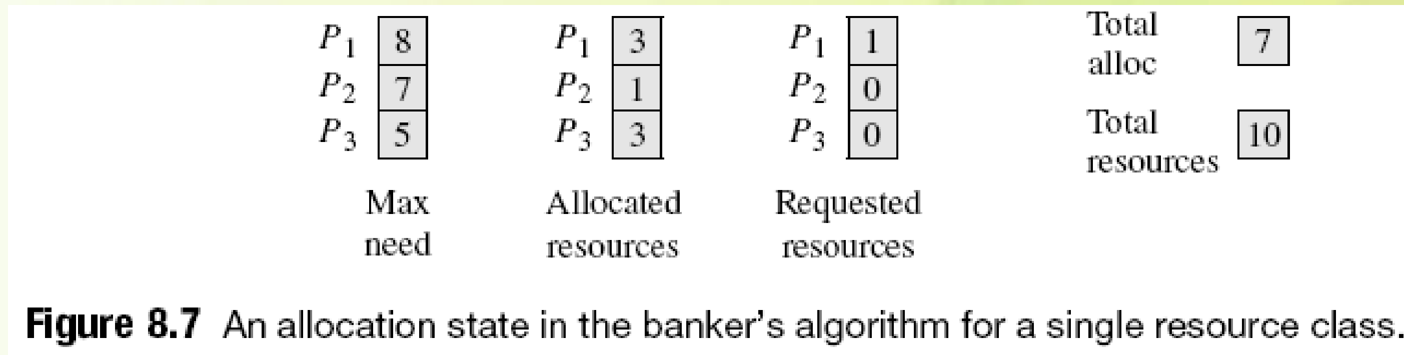


Figure 8.7 An allocation state in the banker's algorithm for a single resource class.

- Now consider the following requests:
  1. P_1 makes a request for 2 resource units
  2. P_2 makes a request for 2 resource units
  3. P_3 makes a request for 2 resource units
  - Requests by P_1 and P_2 do not put the system in safe allocation states, hence they will not be granted
  - Request by P_3 will be granted

Algorithm 8.2 *Banker's Algorithm*

Inputs

n	:	Number of processes;
r	:	Number of resource classes;
Blocked	:	**set of** processes;
Running	:	**set of** processes;
$P_{requesting_process}$	:	Process making the new resource request;
Max_need	:	**array** [1..n, 1..r] **of** *integer*;
Allocated_resources	:	**array** [1..n, 1..r] **of** *integer*;
Requested_resources	:	**array** [1..n, 1..r] **of** *integer*;
Total_alloc	:	**array** [1..r] **of** *integer*;
Total_resources	:	**array** [1..r] **of** *integer*;

Data structures

Active	:	**set of** *processes*;
feasible	:	*boolean*;
New_request	:	**array** [1..r] **of** *integer*;
Simulated_allocation	:	**array** [1..n, 1..r] **of** *integer*;
Simulated_total_alloc	:	**array** [1..r] **of** *integer*;

1. $Active := Running \bigcup Blocked$;
for $k = 1..r$
$New_request[k] := Requested_resources[requesting_process, k]$;

```
2. Simulated_allocation := Allocated_resources;
   for k = 1..r        /* Compute projected allocation state */
       Simulated_allocation[requesting_process, k] :=
               Simulated_allocation[requesting_process, k] + New_request[k];
       Simulated_total_alloc[k] := Total_alloc[k] + New_request[k];
3. feasible := true;
   for k = 1..r        /* Check whether projected allocation state is feasible */
       if Total_resources[k] < Simulated_total_alloc[k] then feasible := false;
4. if feasible = true
   then        /* Check whether projected allocation state is a safe allocation state */
      while set Active contains a process P_l such that
                  For all k, Total_resources[k] − Simulated_total_alloc[k]
                     ≥ Max_need[l, k] − Simulated_allocation[l, k]
          Delete P_l from Active;
          for k = 1..r
              Simulated_total_alloc[k] :=
                  Simulated_total_alloc[k] − Simulated_allocation[l, k];
5. if set Active is empty
   then        /* Projected allocation state is a safe allocation state */
      for k = 1..r        /* Delete the request from pending requests */
          Requested_resources[requesting_process, k] := 0;
          for k = 1..r        /* Grant the request */
              Allocated_resources[requesting_process, k] :=
                  Allocated_resources[requesting_process, k] + New_request[k];
              Total_alloc[k] := Total_alloc[k] + New_request[k];
```

Example: Banker's Algorithm for Multiple Resource Classes

(a) State after Step 1

Max need

	R_1	R_2	R_3	R_4
P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Allocated resources

	R_1	R_2	R_3	R_4
P_1	1	1	1	1
P_2	2	0	1	0
P_3	2	0	2	2
P_4	0	2	1	1

Requested resources

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

	R_1	R_2	R_3	R_4
Total alloc	5	3	5	4
Total exist	6	4	8	5

Active $\{P_1, P_2, P_3, P_4\}$

(b) State before while loop of Step 4

Max need

	R_1	R_2	R_3	R_4
P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Simulated allocation

	R_1	R_2	R_3	R_4
P_1	1	1	1	1
P_2	2	1	2	0
P_3	2	0	2	2
P_4	0	2	1	1

Requested resources

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

	R_1	R_2	R_3	R_4
Simulated total_alloc	5	4	6	4
Total exist	6	4	8	5

Active $\{P_1, P_2, P_3, P_4\}$

Figure 8.8 Operation of the banker's algorithm for Example 8.11.

(c) State after simulating completion of Process P_1

Max need

P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Simulated allocation

P_1	1	1	1	1
P_2	2	1	2	0
P_3	2	0	2	2
P_4	0	2	1	1

Requested resources

P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

Simulated total_alloc	4	3	5	3
Total exist	6	4	8	5

Active $\{P_2, P_3, P_4\}$

(d) State after simulating completion of Process P_4

Max need

P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Simulated allocation

P_1	1	1	1	1
P_2	2	1	2	0
P_3	2	0	2	2
P_4	0	2	1	1

Requested resources

P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

Simulated total_alloc	4	1	4	2
Total exist	6	4	8	5

Active $\{P_2, P_3\}$

(e) State after simulating completion of Process P_2

Max need

P_1	2	1	2	1
P_2	2	4	3	2
P_3	5	4	2	2
P_4	0	3	4	1

Simulated allocation

P_1	1	1	1	1
P_2	2	1	2	0
P_3	2	0	2	2
P_4	0	2	1	1

Requested resources

P_1	0	0	0	0
P_2	0	1	1	0
P_3	0	0	0	0
P_4	0	0	0	0

Simulated total_alloc	2	0	2	2
Total exist	6	4	8	5

Active $\{P_3\}$

Figure 8.8 Operation of the banker's algorithm for Example 8.11.

Characterization of Resource Deadlocks by Graph Models

- A *deadlock characterization* is a statement of the essential features of a deadlock
  - We discuss characterization using graph models of allocation state and elements of graph theory
    - A cycle in a RRAG or WFG is a *sufficient* condition for a deadlock in some systems, but not in others

		Resource request models	
		Single request (SR) model	Multiple request (MR) model
Resource instance models	Multiple instance (MI) model	Multiple-instance, single-request (MISR)	Multiple-instance, multiple-request (MIMR)
	Single instance (SI) model	Single-instance, single-request (SISR)	Single-instance, multiple-request (SIMR)

Figure 8.9 Classification of systems according to resource class and resource request models.

Single-Instance, Single-Request (SISR) Systems

- Each resource class contains a single instance of the resource and each request is a single request
- A cycle in an RRAG implies a mutual wait-for relationship for a set of processes
  - Since each resource class contains a single resource unit
    - Each blocked process P_i in cycle waits for exactly one other process, say P_k, to release required resource
    - Hence a cycle that involves P_i also involves P_k
- A cycle is thus a necessary and sufficient condition to conclude that a deadlock exists in the system

Multiple-Instance, Single-Request (MISR) Systems

- A *knot* in RRAG is a necessary and sufficient condition for the existence of a deadlock in an MISR system

Definition 8.3 Knot A nontrivial subgraph $G' \equiv (N', E')$ of an RRAG in which every node $n_i \in N'$ satisfies the following conditions:

1. For every edge of the form (n_i, n_j) in E: (n_i, n_j) is included in E' and n_j is included in N'.
2. If a path $n_i - \ldots - n_j$ exists in G', a path $n_j - \ldots - n_i$ also exists in G'.

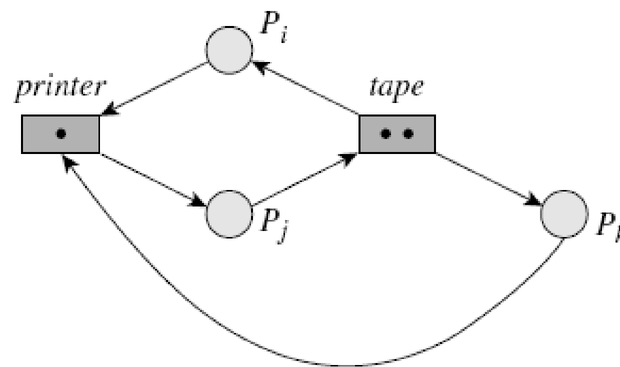


Figure 8.10 A knot in the RRAG of an MISR system implies a deadlock.

Single-Instance, Multiple-Request (SIMR) Systems

- A process making a multiple request has > 1 out-edge
  - It remains blocked until each of the requested resources is available
  - A cycle is a necessary and sufficient condition for a deadlock in an SIMR system

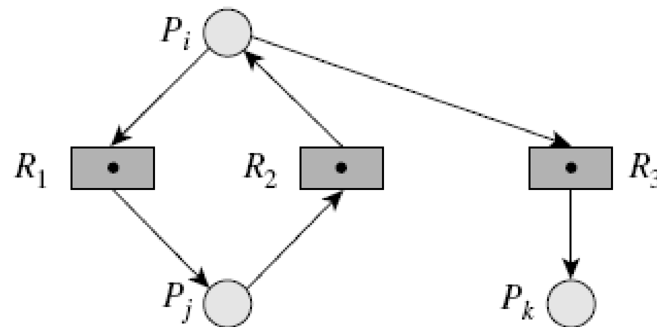


Figure 8.11 A cycle is a necessary and a sufficient condition for a deadlock in an SIMR system.

Multiple-Instance, Multiple-Request (MIMR) Systems

- We must differentiate between process and resource nodes in the RRAG of an MIMR system
  - All out-edges of a resource node must be involved in cycles for a deadlock to arise
  - A process node needs to have only one out-edge involved in a cycle
- A *resource knot* incorporates these conditions

Definition 8.4 Resource Knot A nontrivial subgraph $G' \equiv (N', E')$ of an RRAG in which every node $n_i \in N'$ satisfies the following conditions:

1. If n_i is a resource node, for every edge of the form (n_i, n_j) in E: (n_i, n_j) is included in E' and n_j is included in N'.
2. If a path $n_i - \ldots - n_j$ exists in G', a path $n_j - \ldots - n_i$ also exists in G'.

Multiple-Instance, Multiple-Request (MIMR) Systems (continued)

- A resource knot is a necessary and sufficient condition for the existence of a deadlock in an MIMR system...

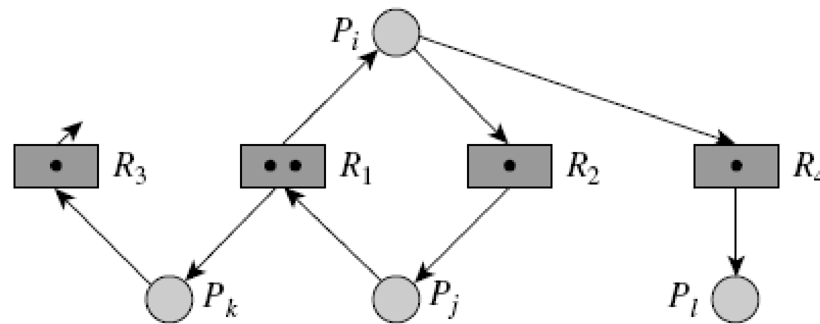


Figure 8.12 RRAG for an MIMR system.

  - And, in all classes of systems discussed in this section

Processes in Deadlock

RR_i	The set of resource classes requested by process P_i.
HS_k	The *holder set* of resource class R_k, i.e., set of processes to which units of resource class R_k are allocated.
KS	The set of process nodes in resource knot(s) (we call it the *knot-set* of RRAG).
AS	An *auxiliary set* of process nodes in RRAG that face indefinite waits. These nodes are not included in a resource knot.

$$AS = \{ P_i \mid RR_i \text{ contains } R_k \text{ such that } HS_k \subseteq (KS \cup AS) \} \quad (8.4)$$

$$D = KS \cup AS \quad (8.5)$$

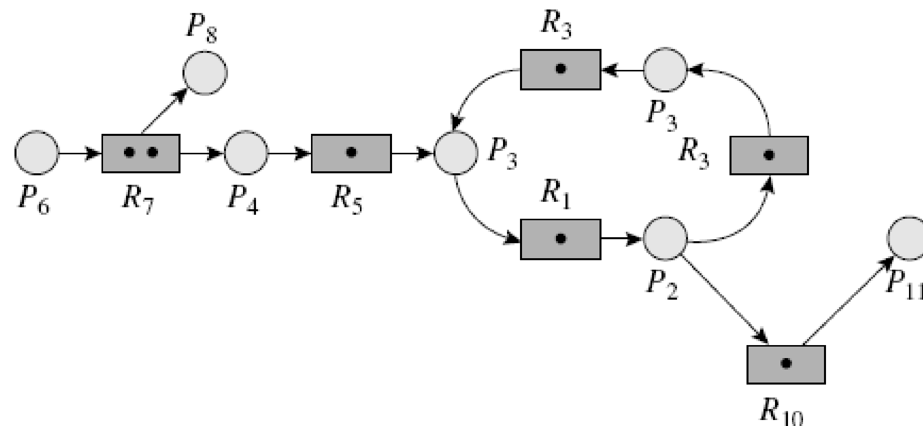


Figure 8.13 Processes in deadlock.

Deadlock Handling in Practice

- Deadlock detection-and-resolution and deadlock avoidance are unattractive in practice (overhead)
  - OS uses deadlock prevention approach or simply does not care about possibility of deadlocks
- OSs tend to handle deadlock issues separately for each kind of resource
  - Memory: Explicit deadlock handling is unnecessary
  - I/O devices: Resources are not limited (virtual devices)
  - Files: Deadlocks are handled by processes, not OS
  - Control blocks: Resource ranking or all-resources-together

Deadlock Handling in Unix

- Most operating systems simply ignore the possibility of deadlocks involving user processes
  - Unix is no exception
- Unix addresses deadlocks due to sharing of kernel data structures by user processes
  - Kernel uses resource ranking (deadlock prevention) by requiring processes to set locks on kernel data structures in a standard order
    - There are exceptions to this rule; deadlocks can arise
      - Special deadlock handling for buffer cache and file system

Deadlock Handling in Windows

- Vista has feature called *wait chain traversal* (WCT)
  - Assists applications and debuggers in detecting deadlocks
  - A wait chain starts on a thread and is analogous to a path in the RRAG
- Debugger can investigate cause of a freeze by invoking **`getthreadwaitchain`** with the id of a thread to retrieve a chain starting on that thread

Summary

- *Deadlock:* set of processes wait indefinitely for events because each of the events can be caused only by other processes in the set
- Resource deadlock arises when:
  - Resources are nonshareable and nonpreemptible
  - *Hold-and-wait*
  - Circular wait exists
- OS can discover a deadlock by analyzing the *allocation state* of a system
  - Use RRAG, WFG or *matrix model*
- Deadlocks can be detected, prevented and avoided

Queries …?