

Course Name: Operating System – Deadlocks

Reference: Stallings W., "Operating Systems: Internals and Design Principles", 7th Edition, Pearson Education Limited 2012, ISBN 10:0-273-75150-6.

Lecturer: Dr. Sahar A. AL-Talib

----- **Deadlocks** -----

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set. For example:

- A system has 2 disk drives
- P_1 and P_2 each hold one disk drive and each needs another one

System Model

- Resource types R_1, R_2, \dots, R_3 [CPU cycles, memory space, I/O devices]
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 1. request → the process acquires the resource it needs from the system which scans the table of resources.
 2. Use → If the resource is available, then use it, otherwise, wait till it released.
 3. Release → free the resource after the process finished.

Deadlock can arise if four conditions hold simultaneously.

1. **Mutual exclusion:** only one process at a time can use a resource.
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
3. **No preemption:** a resource can be released only voluntarily (طوعاً) by the process holding it after that process has completed its task.
4. **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ 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_0 is waiting for a resource that is held by P_0 .

Example of Resource Allocation Graph

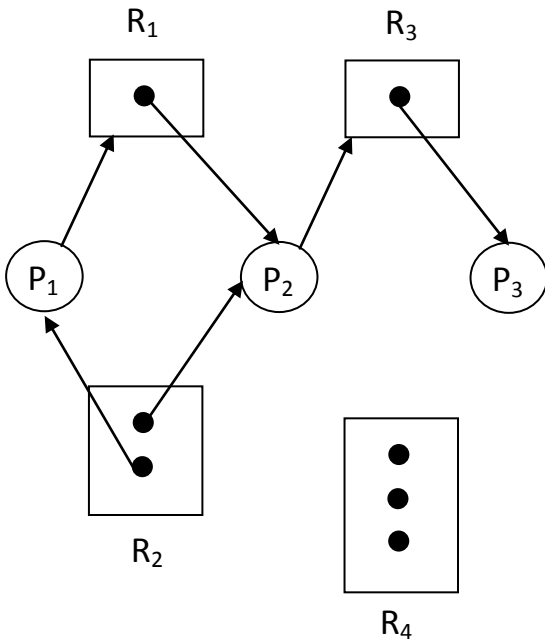


Figure 1

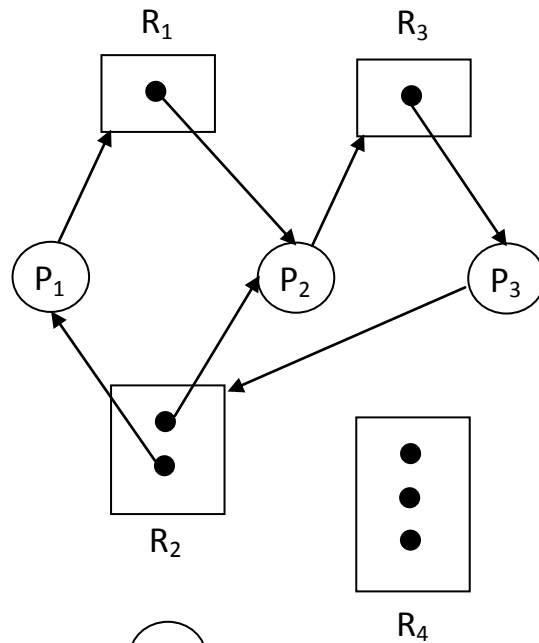


Figure 2

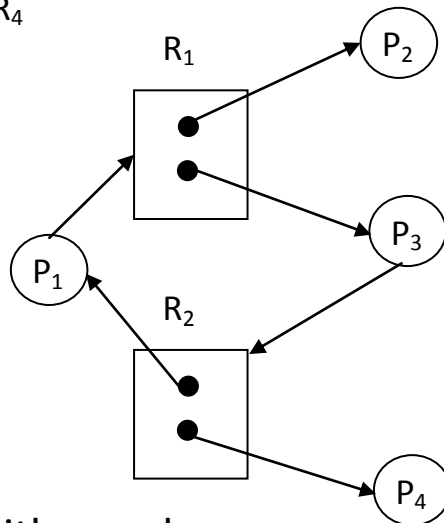


Figure 3: Graph with a cycle

Resource Allocation Graph

A set of vertices \mathbf{V} and a set of edges \mathbf{E} :

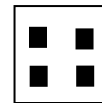
- \mathbf{V} is partitioned into two types:
 - $\mathbf{P} = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system

- $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- request (طلب) edge -- directed edge $P_i \rightarrow R_j$
- assignment (حجز) edge – directed edge $R_j \rightarrow P_i$

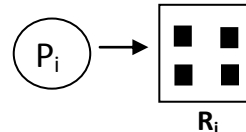
- Process



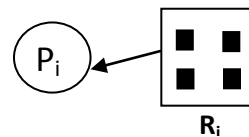
- Resource type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



Basic Facts

- If the graph contains no cycle \implies no deadlock
- If the graph contains a cycle \implies
 - 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 and avoidance)
 - Prevention: ensuring that at least one of the necessary conditions cannot hold.

- Avoidance: using additional info to decide whether the process must wait or not to request a resource.
- 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, WINDOWS.

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
- **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
 - Low resource utilization; starvation possible
- **No preemption** –
 - 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 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

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

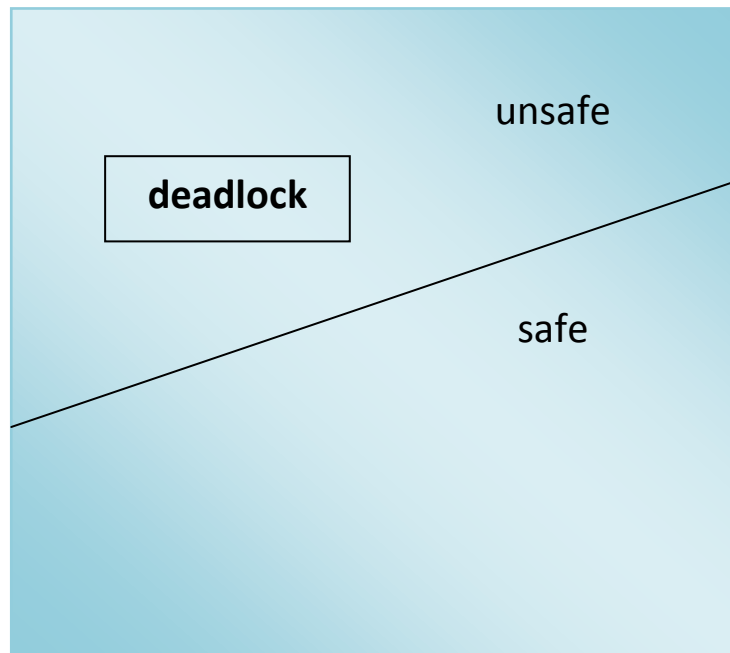
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a safe sequence $\langle P_1, P_2, P_3, \dots, P_n \rangle$ of all the processes in the system 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 .
- 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_j can obtain its needed

Basic facts

- If a system is in safe state == > no deadlocks
- 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

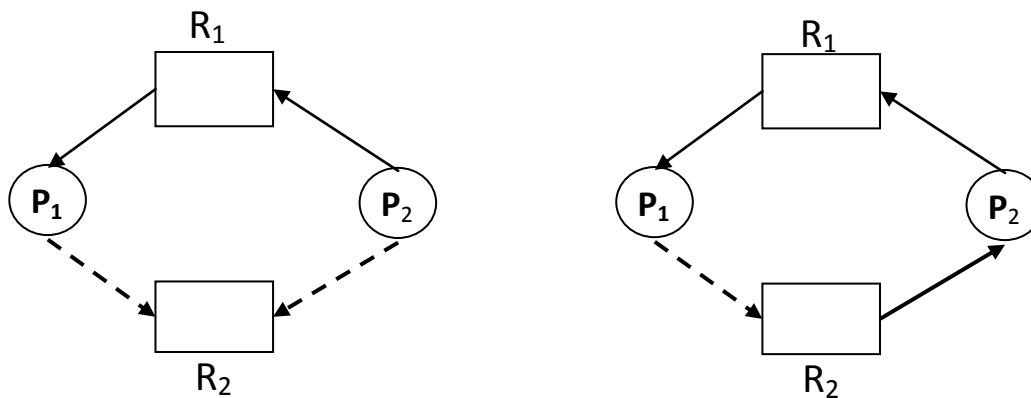


Avoidance algorithms

- Single instance of a source type
 - Use a resource-allocation graph
- Multiple instances of a resource type
 - Use banker's algorithm

Resource Allocation Graph Scheme (single instance)

- 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

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

Banker's Algorithm (multiple instances)

- Multiple instances
- 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:** $(n \times m)$ matrix. If $\text{Max}[i, j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** $(n \times m)$ matrix. If $\text{Allocation}[i, j] = k$ then P_i is currently allocated k instances of R_j
- **Need:** $(n \times m)$ matrix. If $\text{Need}[i, j] = k$, then P_i may need k more instances of R_j to complete its task

$$\text{Need}[i, j] = (\text{max}[i, j] - \text{Allocation}[i, j])$$

Safety Algorithm

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

Work = Available

Finish $[i] = \text{false}$ for $i = 0, 1, \dots, n-1$.

2. Find an index i such that both
 - a. *Finish* $[i] == \text{false}$
 - b. $\text{Need}_i \leq \text{Work}$

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.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Resource – Request Algorithm for Process P_i

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

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe == > the resources are allocated to P_i
- If unsafe == > 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 (5 instances), and C (7 instances), snapshot at time T_0 :

	<i>Allocation</i>	<i>Max</i>	<i>Available</i>	<i>Need</i>
	<i>A B C</i>	<i>ABC</i>	<i>ABC</i>	<i>ABC</i>
P_0	010	753	332	743
P_1	200	322		122
P_2	302	902		600
P_3	211	222		011
P_4	002	433		431

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

	<i>Need</i>
	<i>ABC</i>
P_0	743
P_1	122
P_2	600
P_3	011
P_4	431

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Deadlock Detection

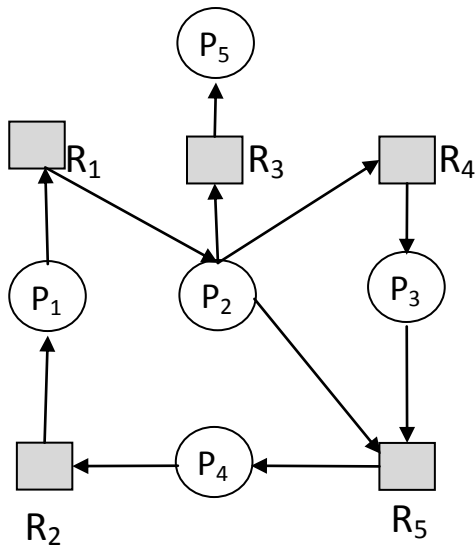
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. So the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

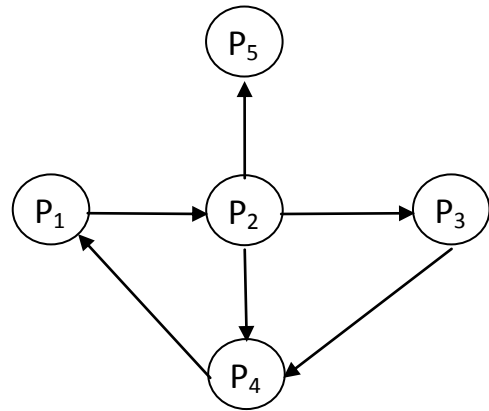
(a) Single instance of each resource type

A detection algorithm called a wait-for graph is used which is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q . A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically to recover from the deadlock invoke an algorithm that searches for a cycle in the graph.

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.



(a) Resource-allocation graph



(b) Corresponding wait-for graph

(b) Several instances of a resource type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. Instead, a deadlock detection algorithm similar to those used in banker's algorithm is applicable. The algorithm employs several time-varying data structures:

- **Available.** A vector of length m indicates the number of available resources of each type.
- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request.** An $n \times m$ matrix indicates the current request of each process. If $Request[i][j]$ equals k , then process P_i is requesting k more instances of resource type R_j .

The detection algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively. Initialize:
Work = Available. For $i = 0, 1, \dots, n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$;
Otherwise, $Finish[i] = true$.

2. Find an index i such that both
a. $Finish[i] == false$
b. $Request_i \leq Work$

If no such i exists, go to step 4.

3. $Work = Work + Allocation_i$

$Finish[i] = true$

Go to step 2.

4. If $Finish[i] == false$ for some i , $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Detection –Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur? If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
2. How many processes will be affected by deadlock when it happens?

Deadlocks occur only when some process makes a request that cannot be granted immediately.

We can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately.

Invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time. A less expensive alternative is simply to invoke the algorithm at defined intervals- for example, once per hour or

whenever CPU utilization drops below 40%. If the detection-algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles. In this case, we generally cannot tell which of the many deadlocked processes caused the deadlock.