

Lecture 7 /

Chapter 7 (Deadlocks)

Deadlock Characterization

Methods for Handling Deadlocks

Deadlock Prevention / Avoidance

Deadlock Detection / Recovery



System Model

- System consists of resources
- Resource types $R_1, R_2, \dots, R_m$
CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - **request**
 - **use**
 - **release**



Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion** (استبعاد متبادل) : 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 has completed its task
- **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_n is waiting for a resource that is held by P_0 .



Resource-Allocation Graph

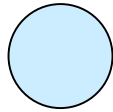
A set of vertices V and a set of edges E .

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, 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 edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

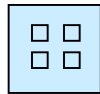


Resource-Allocation Graph (Cont.)

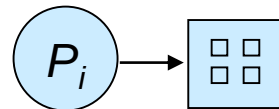
Process



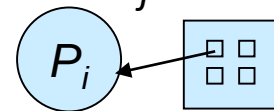
Resource Type with 4 instances



P_i requests instance of R_j

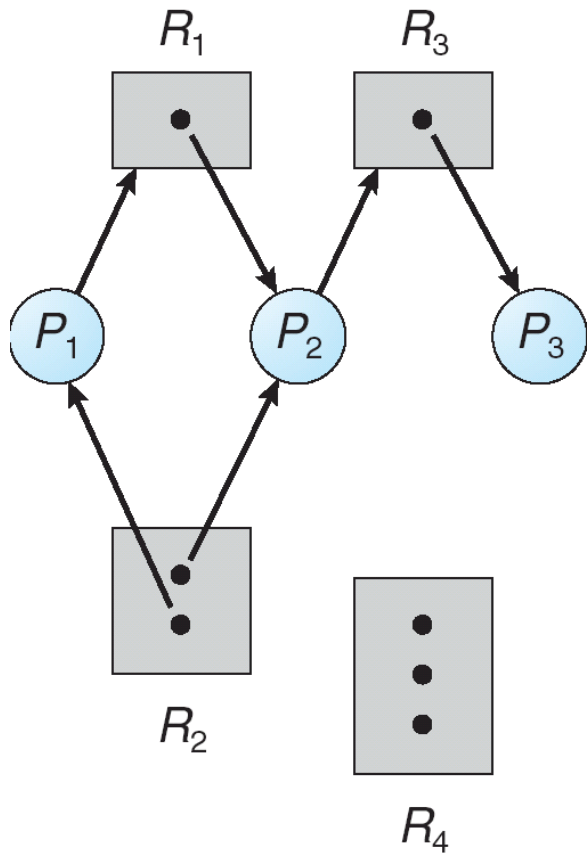


P_i is holding an instance of R_j

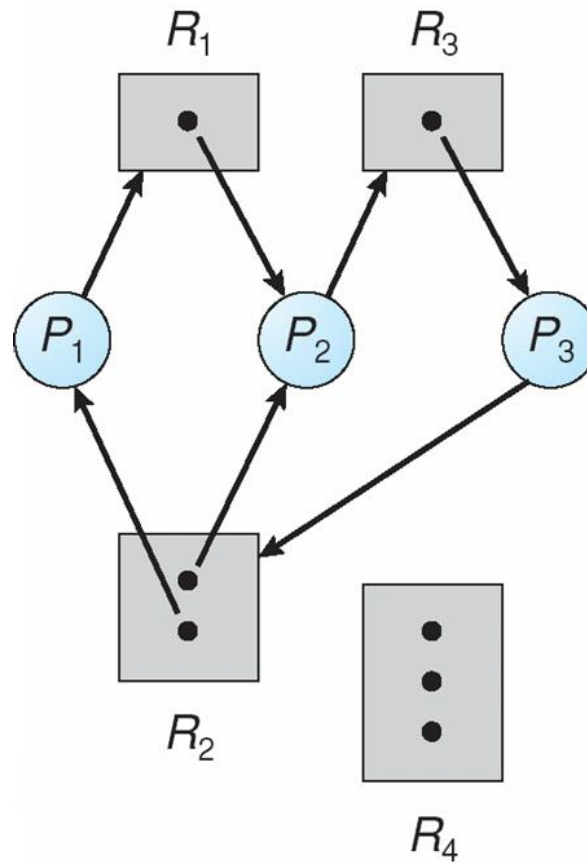


Example of a Resource Allocation Graph

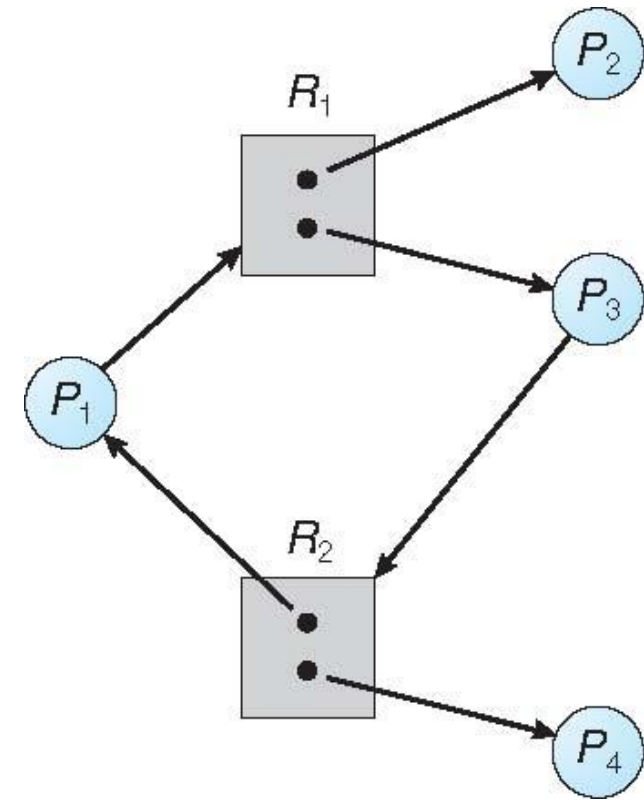
6



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



- If 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
 - Deadlock avoidance
- 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, etc.



Deadlock Prevention

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); 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 allocated to it.
 - Low resource utilization; starvation possible



Deadlock Prevention (Cont.)

- **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 operating system be given additional information (*a priori* information available) in advance concerning which resources a process will request and use during its lifetime.

- 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



- 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 sequence $\langle P_1, P_2, \dots, P_n \rangle$ 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



- 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.

Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per year)

In some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority.

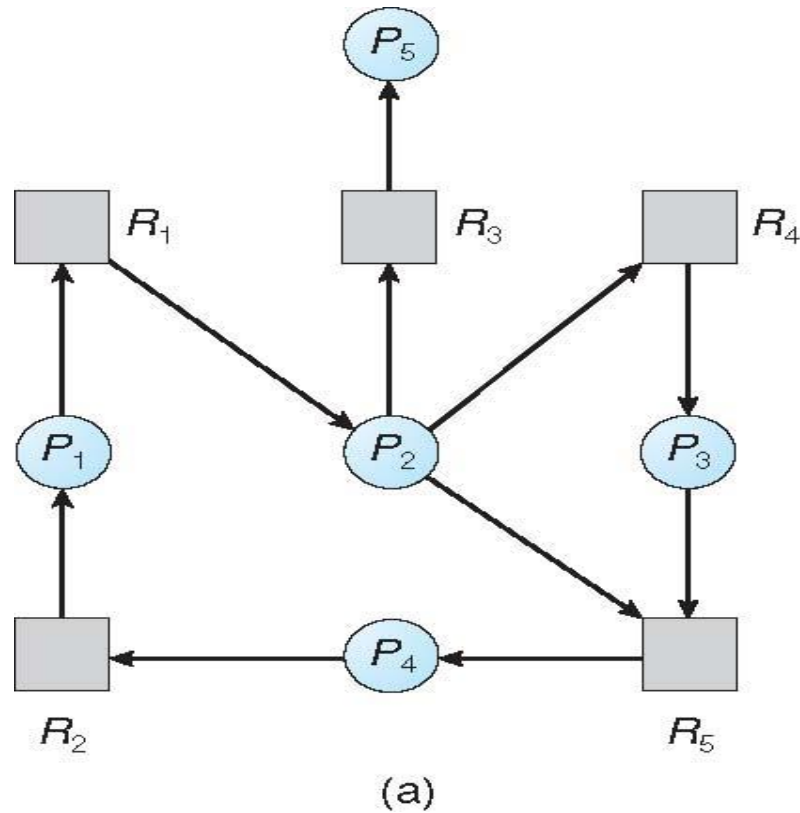


- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

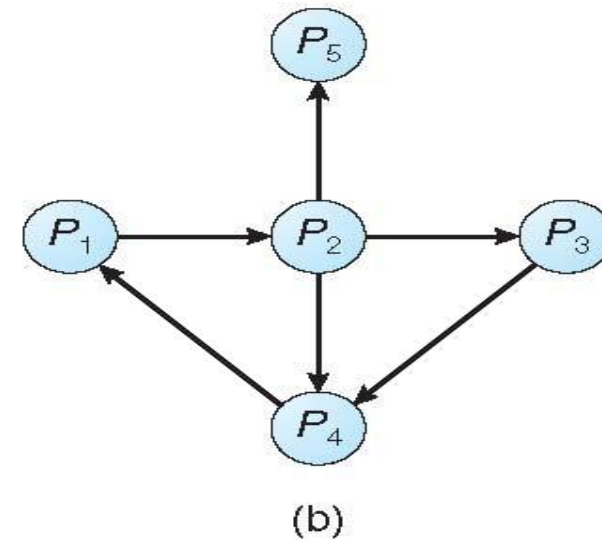


- Maintain **wait-for** graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations,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





Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- **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 $Request[i][j] = k$, then process P_i is requesting k more instances of resource type R_j .



- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily استدعاء تعسفي, 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.



- Process Termination
 - Abort all deadlocked processes
 - Abort one process at a time until the deadlock cycle is eliminated
- Resource Preemption
 - Selecting a victim
 - Rollback
 - Starvation





Chapter 8

Main Memory

Second Exam