DEADLOCKS

Introduction

In a computer system, we have a finite number of resources to be distributed among a number of competing processes.

System resources are classified into:

- Physical Resources
- Logical Resources

Introduction continue…

In a computer system, we have a finite number of resources to be distributed among a number of competing processes.

System resources are classified into:

- Physical Resources
- Logical Resources

- **Printers**
- **Tape drivers**
- **Memory space**
- **CPU cycles**

Introduction continue…

In a computer system, we have a finite number of resources to be distributed among a number of competing processes.

System resources are classified into:

- Physical Resources
- Logical Resources

- **Files**
- **Semaphores**

Introduction continue…

A process must request a resource before using it and release the resource after using it.

The number of resources requested can not exceed the total number of resources available in the system.

Introduction continue…

In a normal operation, a process may utilize a resource in the following sequence:

- Request the resource
- Use the resource
- Release the resource

Introduction continue…

In a normal operation, a process may utilize a resource in the following sequence:

- Request
- Use
- Release

If the request can not be immediately granted, then the requesting process must wait until it can get the resource.

Introduction continue…

In a normal operation, a process may utilize a resource in the following sequence:

- Request
- Use
- Release

The requesting process can operate on the resource.

Introduction continue…

In a normal operation, a process may utilize a resource in the following sequence:

- Request
- Use
- Release

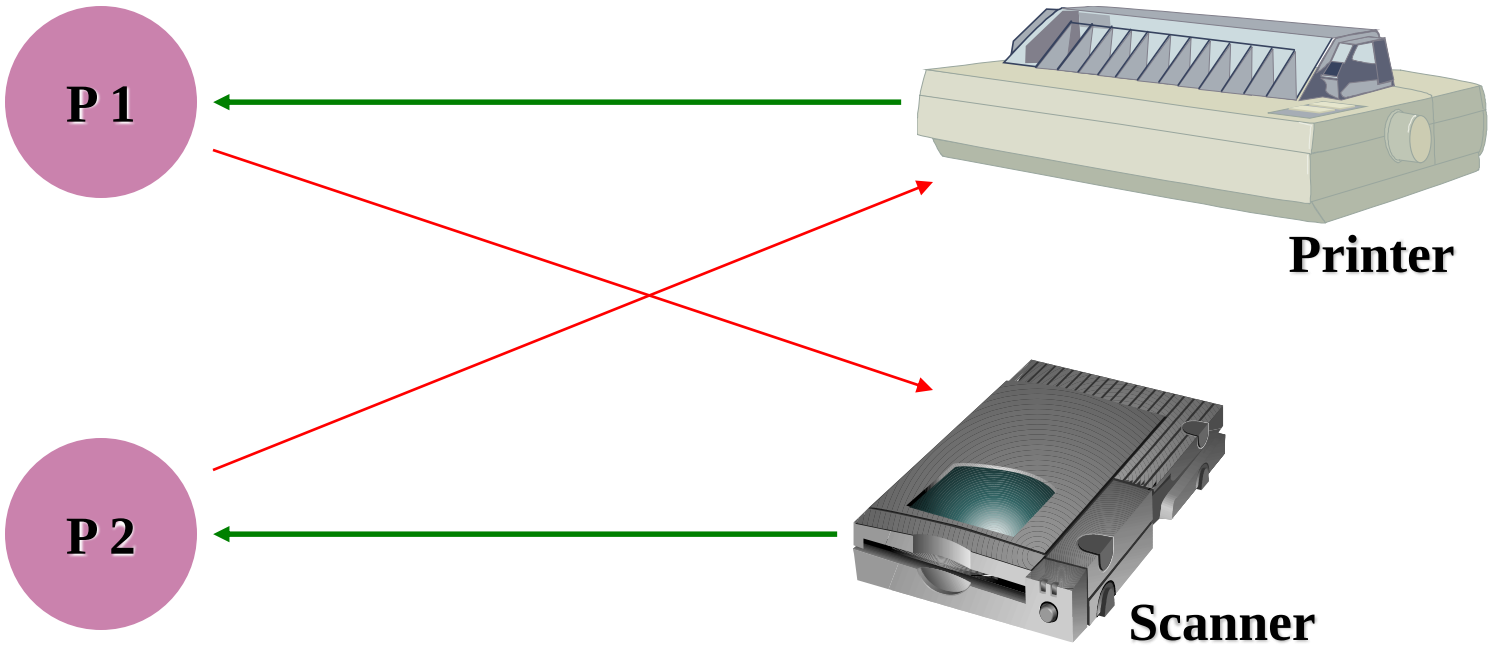
The process releases the resource after using it.

The OS is responsible for making sure that the requesting process has been allocated the resource. Request and release of resources can be accomplished through the wait and signal operations on semaphores. A system table records whether each resource is free or allocated, and if allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

Deadlock Situation

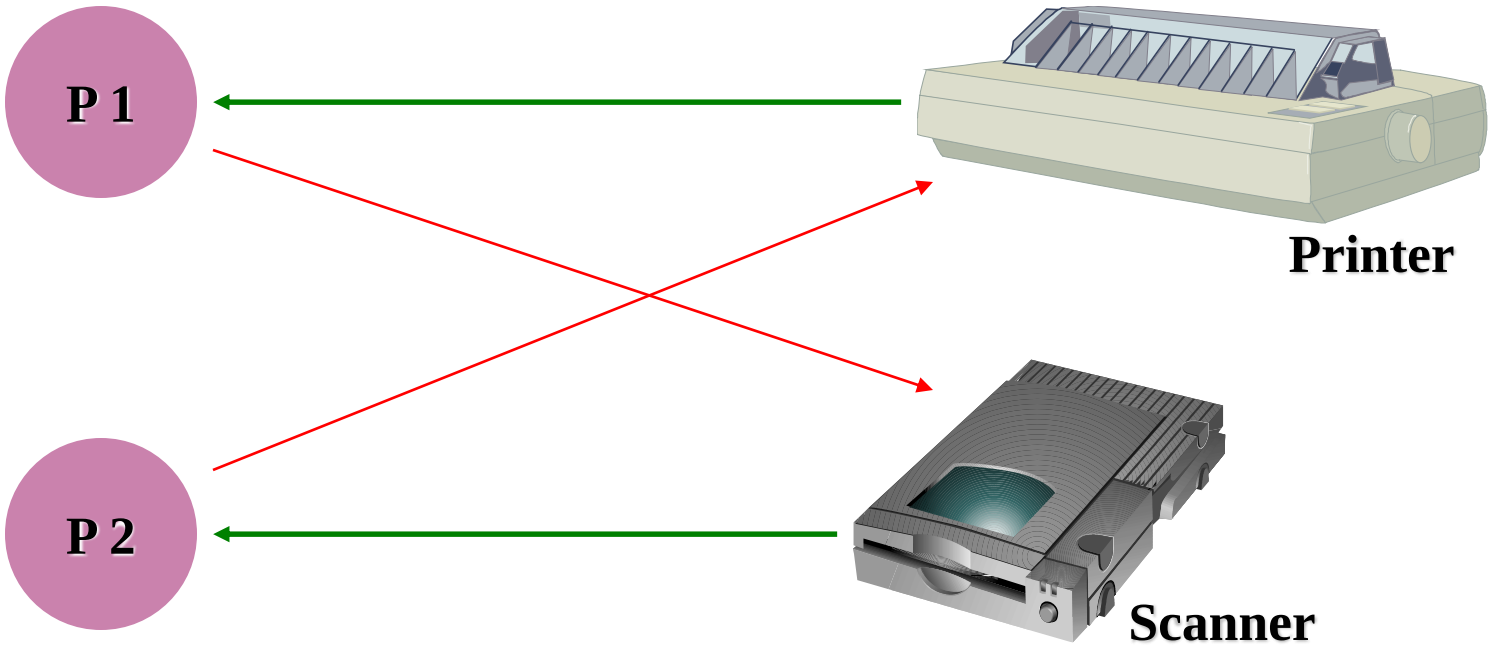
In a multi-programming environment, several processes may compete for a fixed number of resources. A process requests resources and if the resources are not available at that time, it enters a wait state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called deadlock.

Deadlock Situation



Consider a system with one printer and one scanner. Process P1 request the printer and process P2 requests the scanner. Both requests are granted. Now P1 requests the scanner (without giving up the printer) and P2 requests the printer (without giving up the scanner). Neither request can be granted so both processes enter a deadlock situation.

Deadlock Situation



Deadlock Characterization

❖ **Necessary Conditions**

A deadlock situation can arise if the following four conditions hold simultaneously in a system.

- **Mutual Exclusion**
- **Hold and Wait**
- **No Preemption**
- **Circular Wait**

Deadlock Characterization

- **Necessary Conditions** **Continue…**

Mutual Exclusion

At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Deadlock Characterization

- **Necessary Conditions** **Continue…**

Hold and Wait

A process must be holding at least one resource and waiting to acquire additional resources that are currently held by other processes.

Deadlock Characterization

❖ **Necessary Conditions** **Continue…**

No Preemption

Resources already allocated to a process cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after process has completed its task.

Deadlock Characterization

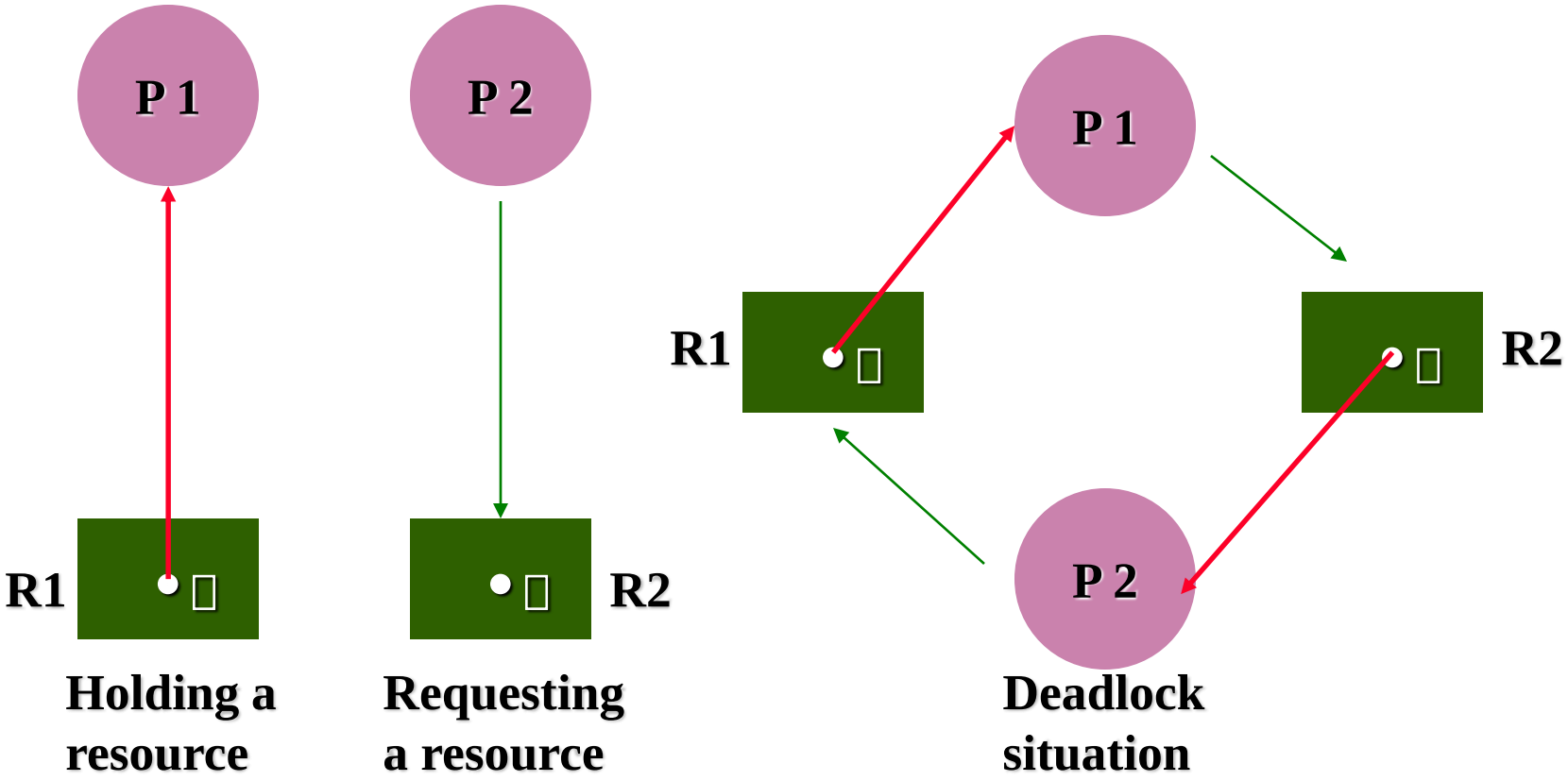
- **Necessary Conditions** **Continue…**

Circular Wait

A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist 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.

The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

Deadlock Situation



Graphic representation of Resource Allocation

Resource-Allocation Graph

Resource-allocation graph is used to described the deadlock.

This graph consists of a set of vertices V and a set of edges E. The set vertices V is partitioned into two different types of nodes $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all active processes in the system, and $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

Resource-Allocation Graph continue…

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i.

A directed edge $P_i \rightarrow R_j$ is called a request edge and

a directed edge $R_j \rightarrow P_i$ is called an assignment edge.

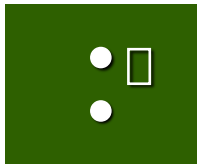
Resource-Allocation Graph continue…

Process node is represented by circles.

Resource node is represented by rectangles.

R1

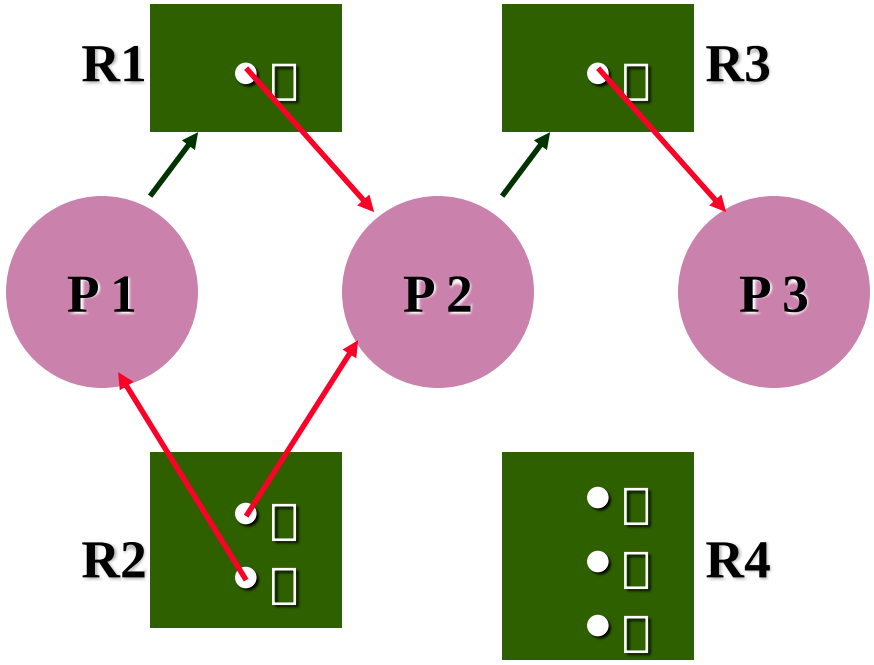
For each instances of a resource type, there is a dot in the resource node rectangle.



R2

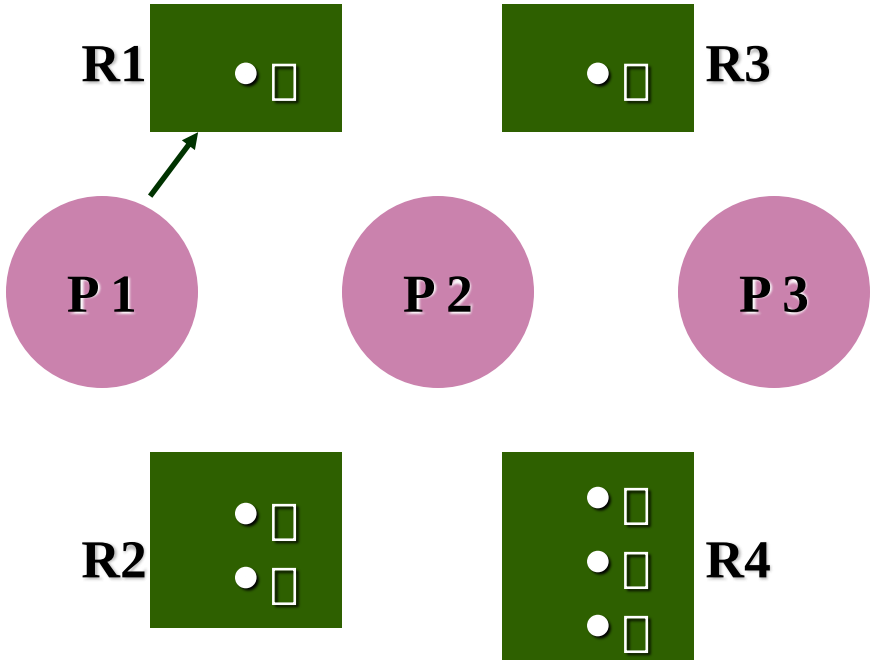
Resource-Allocation Graph continue…

A request edge points to only the square , where as an assignment edge must also designate one of the dots in the square.



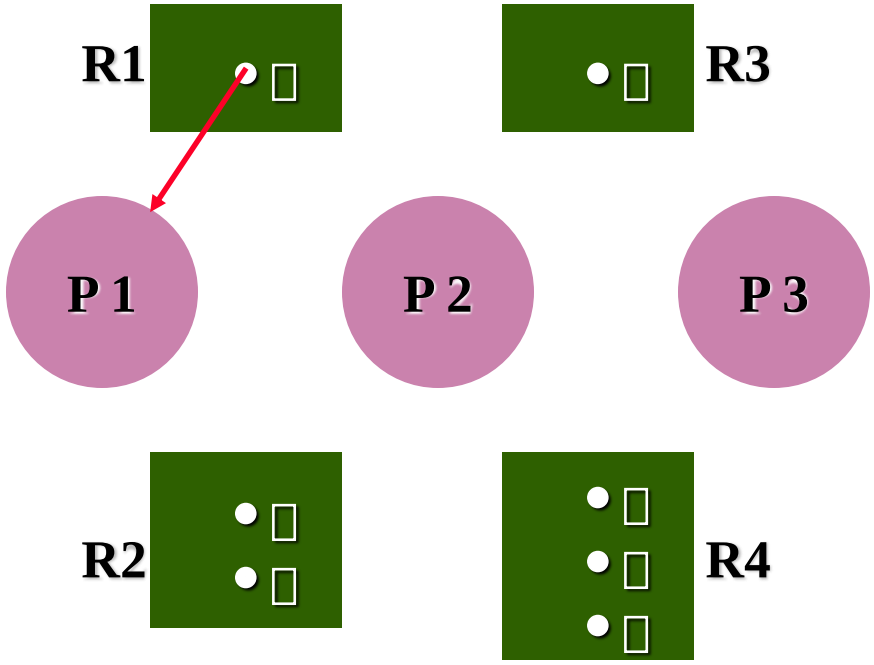
Resource-Allocation Graph continue…

When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource allocation graph.



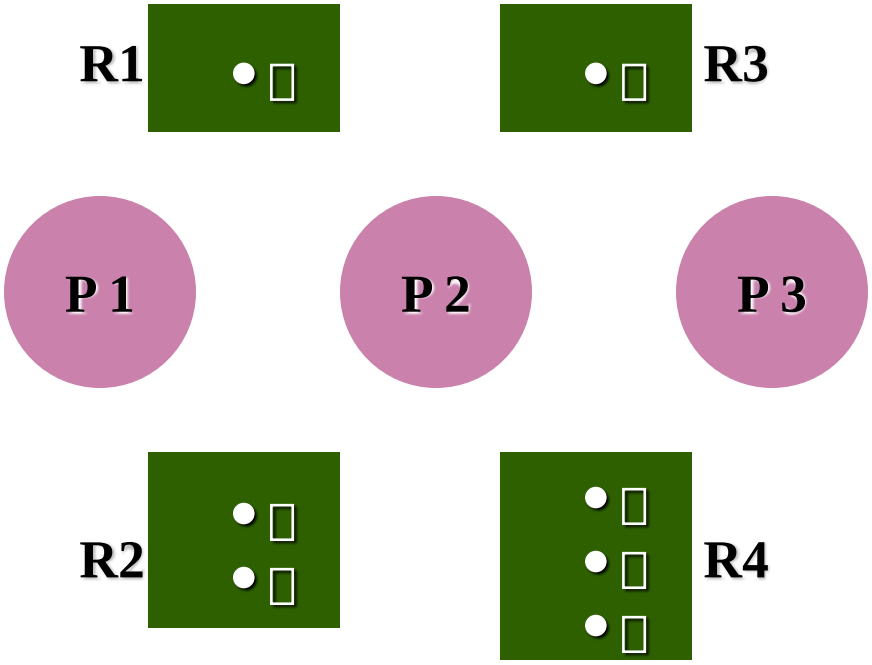
Resource-Allocation Graph continue…

When this requests can be fulfilled the request edge is instantaneously transformed to an assignment edge.



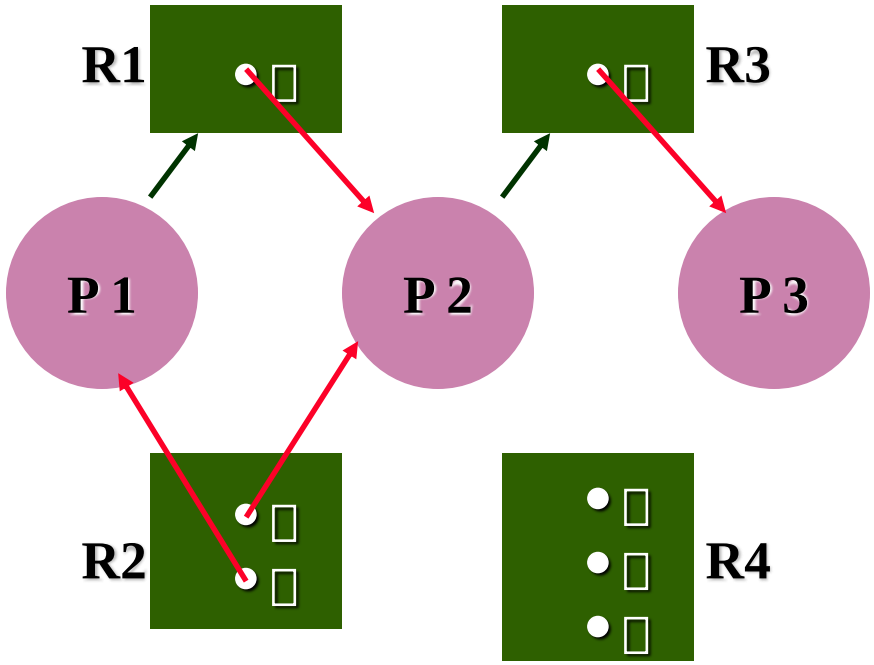
Resource-Allocation Graph continue…

When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted.



Resource-Allocation Graph continue…

The resource-allocation graph depicts the following:

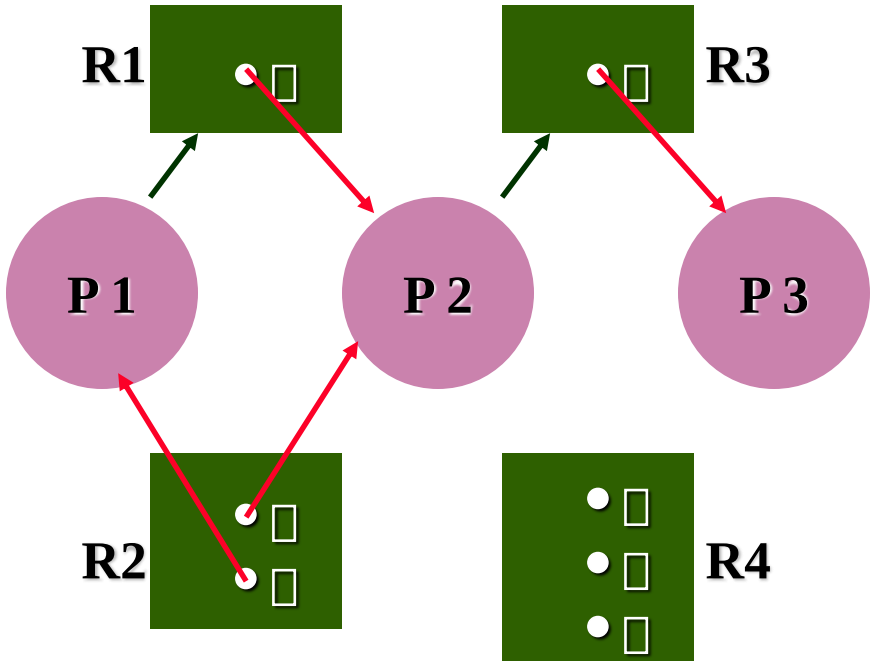


The sets *P*, *R*, and *E*

- $P = \{P_1, P_2, \ldots, P_n\}$
- $R = \{R_1, R_2, \ldots, R_m\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource-Allocation Graph continue…

The resource-allocation graph depicts the following:



Resource instances

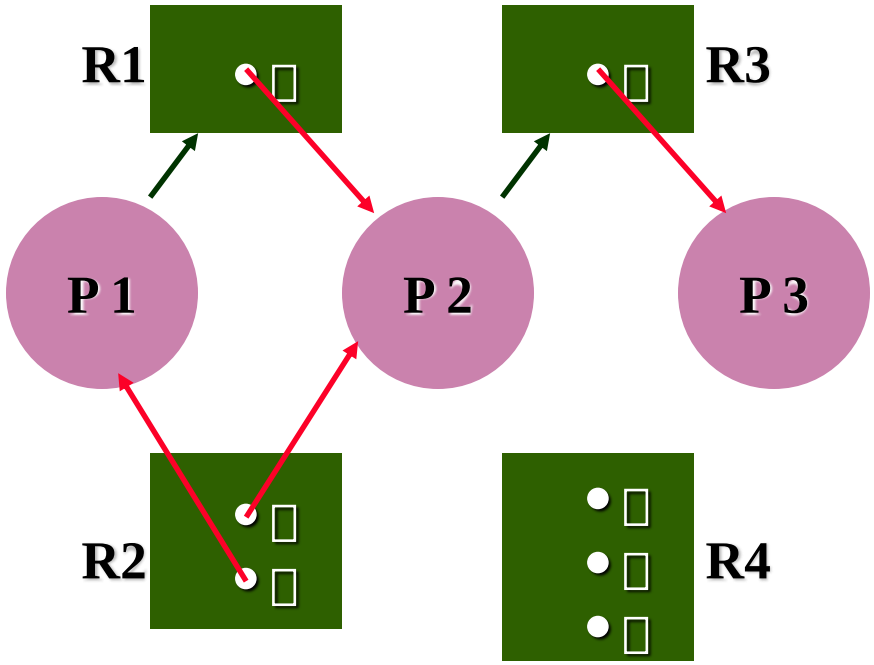
- *One instance of resource type R_1*
- *Two instances of resource type R_2*
- *One instance of resource type R_3*
- *Three instances of resource type R_4*

Resource-Allocation Graph continue…

The resource-allocation graph depicts the following:

Process states

Process P_1 is holding an instance of resource type R_2, and is waiting for an instance of resource type R_1.



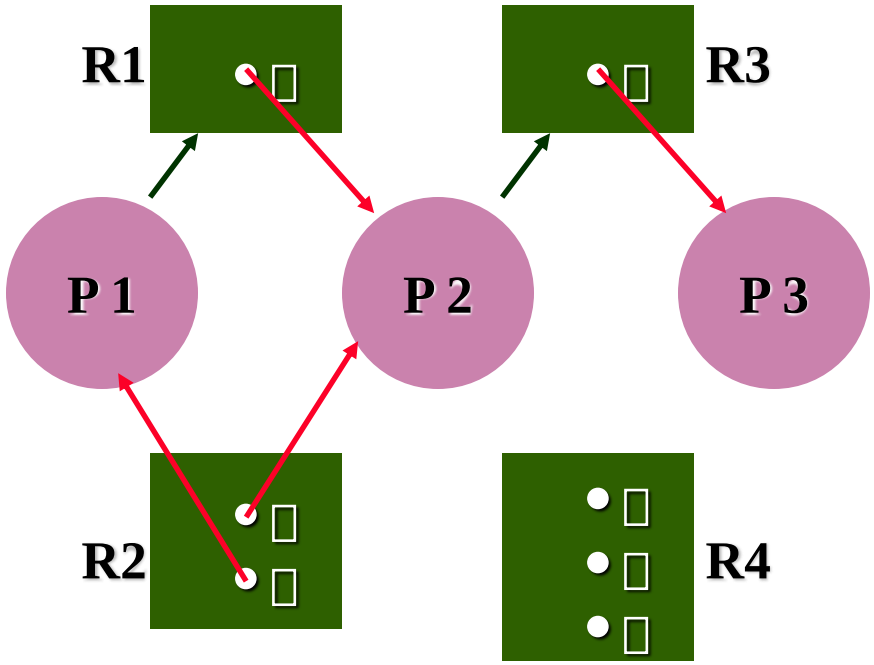
Resource-Allocation Graph continue…

The resource-allocation graph depicts the following:

Process states

Process P_1 is holding an

Process P_2 is holding an instance of resource types R_1 and R_2, and is waiting for an instance of resource type R_3 .



Resource-Allocation Graph continue…

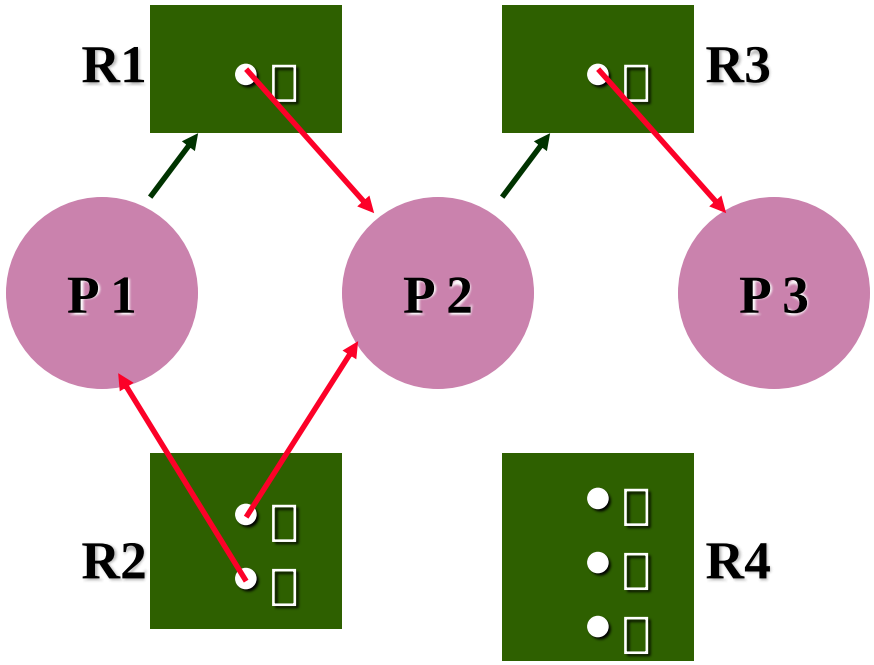
The resource-allocation graph depicts the following:

Process states

Process P_1 is holding an

Process P_2 is holding an

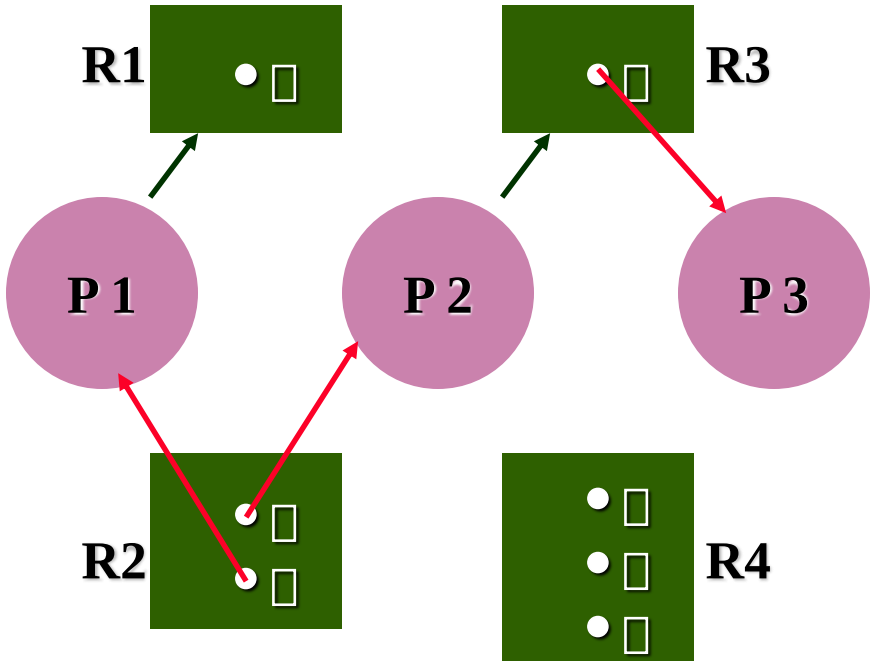
Process P_3 is holding an instance of resource type R_3.



Resource-Allocation Graph continue…

Tips to check deadlock:

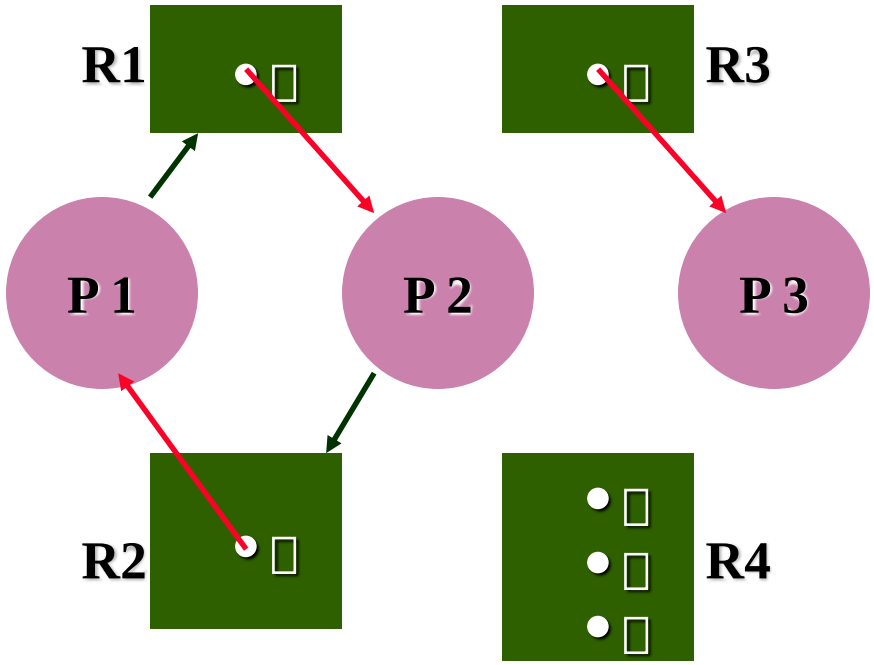
If no cycle exists in the resource allocation graph, there is no deadlock.



Resource-Allocation Graph continue…

Tips to check deadlock:

If there is a cycle in the graph and each resource has only one instance, then there is a deadlock. Here a cycle is a necessary and sufficient condition for deadlock.



Handling Deadlocks

Possible strategies to deal with deadlocks:

- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection and Recovery**

Handling Deadlocks continue…

Possible strategies to deal with deadlocks:

- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection and Re**

Is a set methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Handling Deadlocks continue…

Possible strategies to deal with deadlocks:

- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection and R**

DA requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this knowledge, we can decide for each request whether or not the process should wait.

Handling Deadlocks continue…

Possible strategies to deal with deadlocks:

- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection and Recovery**

Here the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.

Deadlock Prevention

We can prevent the occurrence of a deadlock by ensuring that at least one of the four necessary conditions can not hold.

- **Mutual Exclusion**
- **Hold and Wait**
- **No Preemption**
- **Circular Wait**

Deadlock Prevention continue…

Mutual Exclusion:

Shareable resources do not require mutually exclusive access, and thus cannot be involved in a deadlock.

Deadlock Prevention continue…

Hold and Wait:

When a process requests a resource, it does not hold any other resources. Two protocols:

- **Requires each process to request and be allocated all its resources before it begins execution.**
- **Allows a process to request resources only when the process has none.**

Deadlock Prevention continue…

Hold and Wait:

When a process requests a res

other resources. Two proto

- **Requires each process to rec**

  resources before it begins exe

- **Allows a process to request resources only when the process has none.**

A process may request some resources and use them. However it must release all the resources that is currently allocated before it can request any additional resources.

Deadlock Prevention continue…

No Preemption:

If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted. The process will be restarted only when it can regain its old resources, as well as the new ones that is requesting.

Deadlock Prevention continue…

Circular Wait:

One way to ensure that circular wait never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Deadlock prevention algorithms prevent deadlocks by restraining how request can be made. This ensures that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this methods are low utilization and reduced system throughput.

Deadlock Avoidance

Deadlock-Avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this knowledge, it can decide for each request whether or not the process should wait.

Given a prior information about the maximum number of resources of each type that may be requested for each process, it is possible to construct an algorithm that ensures the deadlock-avoidance approach.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The 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(up to its maximum) in some order and still avoid a deadlock. A system is in a safe state only if there exists a safe sequence.

A sequence of processes <P1, P2, …, Pn> is a safe sequence for the current allocation state if, for each Pi, the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all the Pj, with j < i.

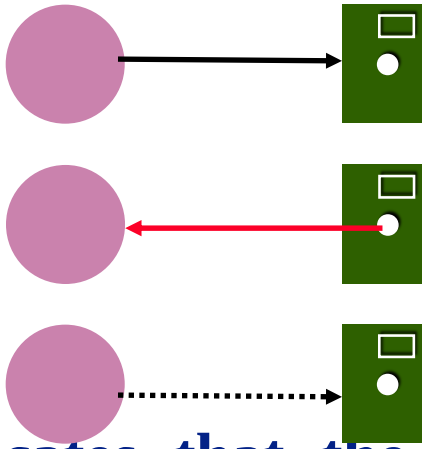
In this situation the resources that process Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources and terminate. When Pi terminates, Pi+1 can obtain its needed resources and so on. If no sequence exists, then the system state is said to be unsafe.

Avoidance algorithms ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.

Resource-Allocation Graph Algorithm

This graph has three types of edges:

- **Request edge**
- **Assignment edge**
- **Claim edge**



A claim edge $P_i \rightarrow R_j$ indicates that the process P_i may request resource R_j at some time in the future.

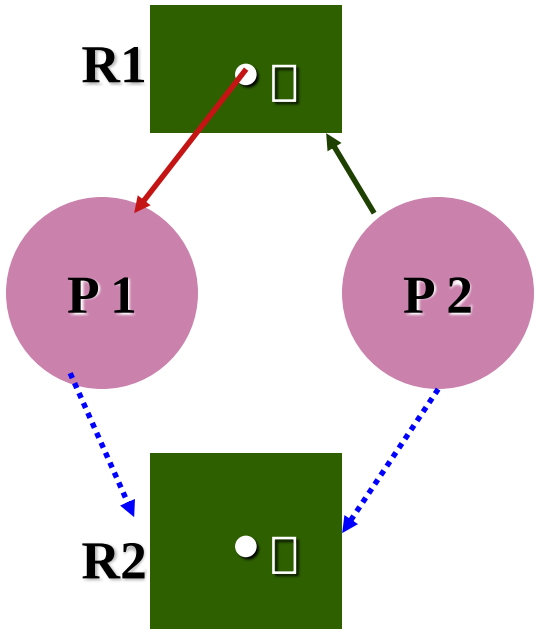
When process Pi request resource Rj, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. When the resource Rj is released by Pi, the assignment edge $R_j \rightarrow P_i$ is converted to a claim edge $P_i \rightarrow R_j$.

Before process P_i starts executing, all its claim edges must appear in the resource-allocation graph.

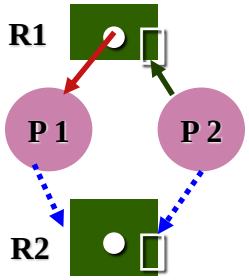
Suppose, process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph.

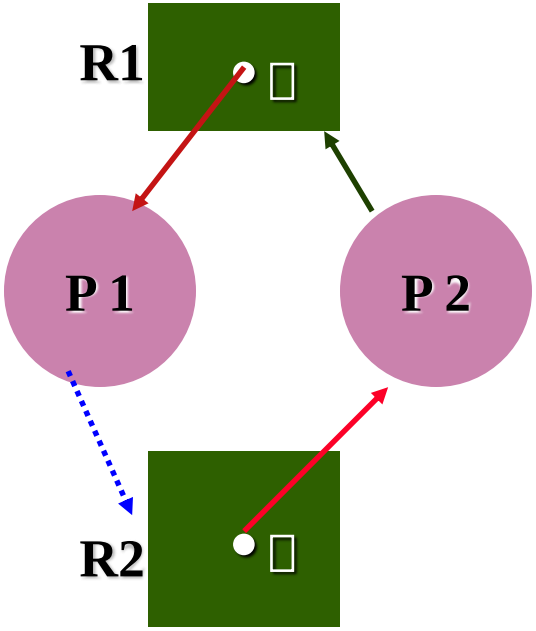
Safety is provided by the use of cycle detection algorithm.

If P2 requests R2, we cannot allocate it to P2 even if R2 is currently free, since this action will create a cycle in the graph.



If P2 requests R2, we cannot allocate it to P2 even if R2 is currently free, since this action will create a cycle in the graph.





Banker's Algorithm

Resource-Allocation Graph Algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

Banker's Algorithm deals multiple instances of each resource type.

This algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

Banker's Algorithm continue…

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number resources in the system.

Banker's Algorithm continue…

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Banker's Algorithm continue…

To implement Banker's algorithm, we need the following data structures:

- **Available**
- **Max**
- **Allocation**
- **Need**

Banker's Algorithm continue…

To implement Banker's algorithm, we need the following data structures:

- **Available**
- **Max**
- **Allocation**
- **Need**

A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.

Banker's Algorithm continue…

To implement Banker's algorithm, we need the following data structures:

- **Available**
- **Max**
- **Allocation**
- **Need**

An n x m matrix defines the maximum demand of each process. If max[i,j] = k, then process Pi may request at most k instances of resource type Rj.

Banker's Algorithm continue…

To implement Banker's algorithm, we need the following data structures:

- **Available**
- **Max**
- **Allocation**
- **Need**

An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process Pi is currently allocated k instances of resource type Rj.

Banker's Algorithm continue…

To implement Banker's algorithm, we need the following data structures:

- **Available**
- **Max**
- **Allocation**
- **Need**

An n x m matrix indicates the remaining resource need of each process. If Need[i, j] = k, then process Pi may need k more instances of resource type Rj to complete its tasks. Need[i,j] = Max[i,j] – Allocation[i,j].

Banker's Algorithm continue…

We can treat each row in the matrices Allocation and Need as vectors and refer to them as $Allocation_i$ and $Need_i$. The vector $Allocation_i$ specifies the resources currently allocated to process Pi; the vector $Need_i$ specifies the additional resources that process Pi may still request to complete its task.

Banker's Algorithm continue…

Safety Algorithm:

1. Let Work and Finish be vectors of length m and n respectively. Initialize Work := Available and Finish[i] := false for i=1,2, …, n.
2. Find an i such that both
   - Finish[i] = false
   - $Need_i \leq Work$.

   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] = true for all i, then the system is an a safe state.

Banker's Algorithm continue…

Resource-Request Algorithm: continue…

1. **If Requesti ≤ Needi, go to step 2. Otherwise, raise an error condition, since the process has exceeded it maximum claim.**
2. **If Requesti ≤ Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.**
3. **Pretend to have allocated the requested resources to process Pi by modifying the state as follows:**
   - **$Available := Available – Request_i$;**
   - **$Allocation_i := Allocation_i + Request_i$;**
   - **$Need_i := Need_i – Request_i$;**

Banker's Algorithm continue…

Resource-Request Algorithm: continue…

If the resulting resource-allocation state is safe, the transaction is completed and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti and the old resource-allocation state is restored.

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then it should provide:

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

Deadlock Detection continue…

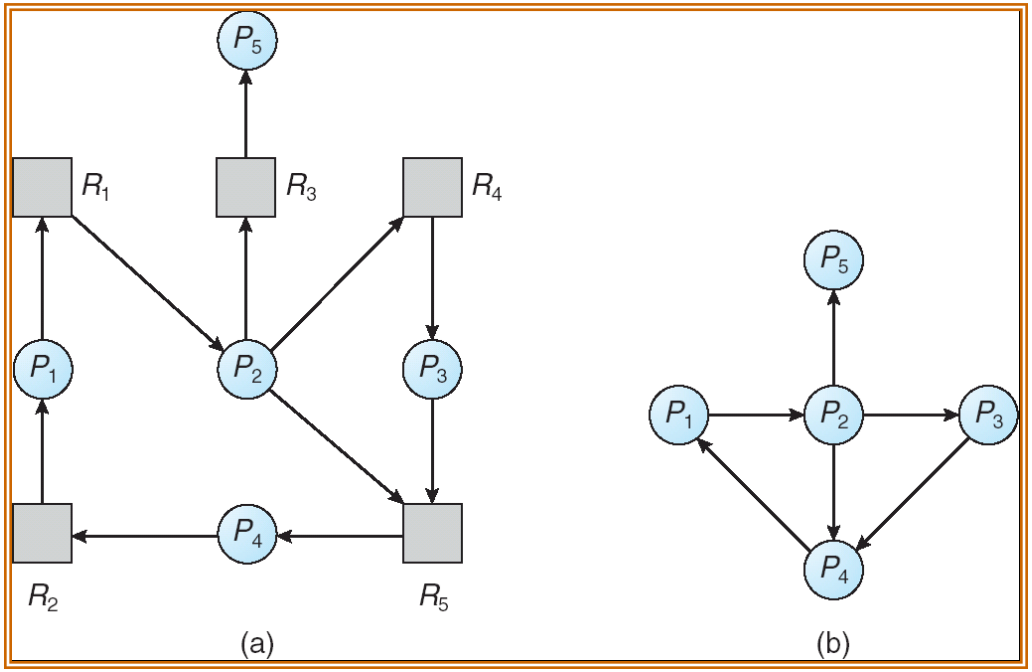
Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph.

We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

Deadlock Detection continue…

Single Instance of Each Resource Type



(a) Resource-allocation graph (b) Corresponding wait-for graph

Deadlock Detection continue…

Single Instance of Each Resource Type

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.

Deadlock Detection continue…

Single Instance of Each Resource Type

If there exists a cycle in wait-for graph, there is a deadlock in the system, and the processes forming the part of cycle are blocked in the deadlock. To take appropriate action to recover from this situation, an algorithm needs to be called periodically to detect existence of cycle in wait-for graph.

Deadlock Detection continue…

Multiple Instances of a Resource Type

When multiple instances of a resource type exist, the wait-for graph becomes inefficient to detect the deadlock in the system.

An algorithm which uses certain data structures similar to ones used in Banker's algorithm is applied.

Deadlock Detection continue…

Multiple Instances of a Resource Type

The following data structures are used:

- **Available Resources**
- **Current Allocation**
- **Request**

Deadlock Detection continue…

Multiple Instances of a Resource Type

The following data structures are used:

- **Available Resources, A: A vector of size q stores information about the number of available resources of each type.**
- **Current Allocation**
- **Request**

Deadlock Detection continue…

Multiple Instances of a Resource Type

The following data structures are used:

- **Available Resources**
- **Current Allocation, C: A matrix of order pxq stores information about the number of resources of each type allocated to each process. C[i][j] indicates the number of resources of type j currently held by the process i.**
- **Request**

Deadlock Detection continue…

Multiple Instances of a Resource Type

The following data structures are used:

- **Available Resources**
- **Current Allocation**
- **Request, R: A matrix of order pxq stores information about the number of resources of each type currently requested by each process. R[i][j] indicates the number of resources of type j currently requested by the process i.**

Deadlock Detection continue…

Multiple Instances of a Resource Type

Consider a vector Complete of size p.

Steps to detect the deadlock:

1. **Initialize Complete[i]=False for all i=1,2,3, …., p. Complete[i]=False indicates that the ith process is still not completed.**
2. **Search for an i, such that Complete[i]=False and (R<=A), that is , resources currently requested by this process is less than the available resources. If no such process exists, then go to step 4.**

Deadlock Detection continue…

Multiple Instances of a Resource Type

3. **Allocate the required resources and let the process finish its execution and set Complete[i]=True for that process. Go to Step 2.**

4. **If Complete[i]=False for some i, then the system is in the state of deadlock and the ith process is deadlocked.**

Deadlock Recovery

When a detection algorithm determines that a deadlock exists, then:

- **let the operator deal with the deadlock manually.**
- **let the system recover from the deadlock automatically.**

There are two options for breaking a deadlock:

- **to abort one or more processes to break the circular wait.**
- **to preempt some resources from one or more of the deadlocked processes.**

Deadlock Recovery continue…

Process Termination:

There are two methods that can be used for terminating the processes to recover from the deadlock:

- **Terminating one process at a time until the circular-wait condition is eliminated.**

- **Terminating all processes involved in the deadlock.**

Deadlock Recovery continue…

Resource Preemption:

An alternate method to recover system from the state of deadlock is to preempt the resources from the processes one by one and allocate them to other processes until the circular-wait condition is eliminated.

Deadlock Recovery continue…

Resource Preemption:

Steps involved in the preemption of resources from processes are:

1. Select a process for preemption.
2. Roll back of the process.
3. Prevent starvation.