Deadlocks

Moumita Patra
Galvin & Gagne
Jul-Nov 2019

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- Resources may be partitioned into several types/classes.
- Each resource type has a number of identical instances.
- CPU cycles, files, and I/O devices are examples of resource types.

Each process utilizes a resource as follows:

- Request
- Use
- Release

Deadlock Characterization

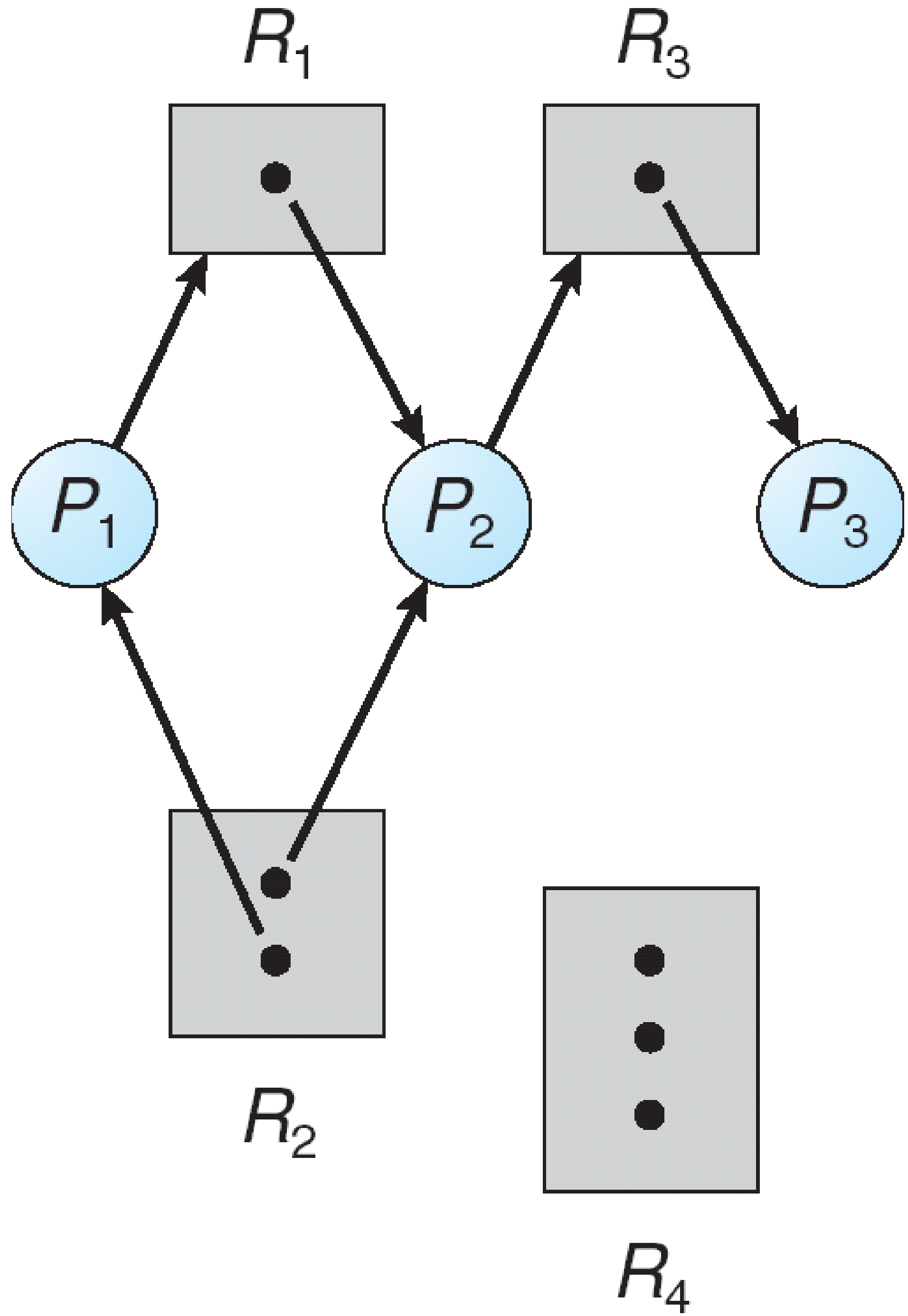
Deadlock can arise if the following conditions hold simultaneously:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Resource-Allocation Graph

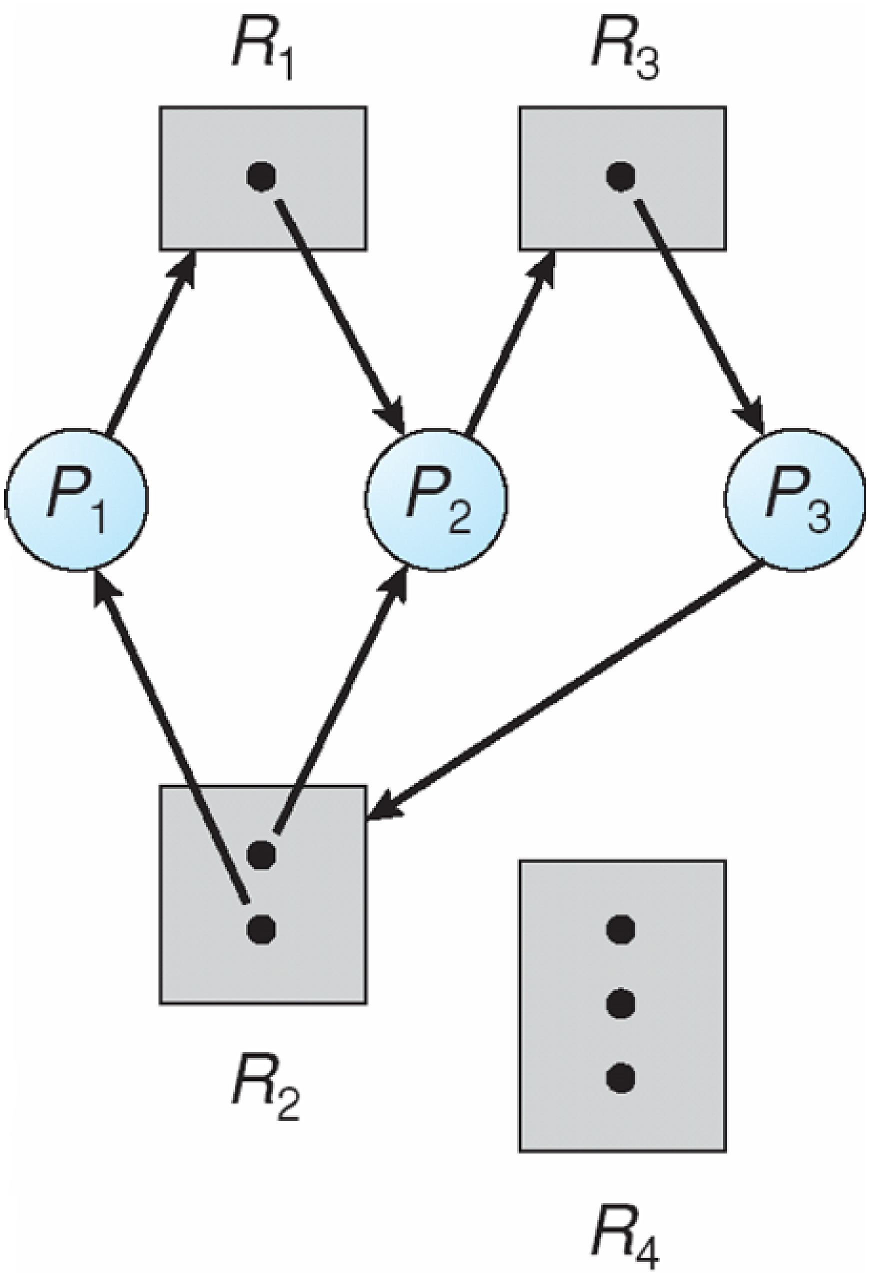
Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph.**

- Graph consists of a set of vertices V and a set of edges E.
- $P = \{P_1, P_2, \ldots, P_n\}$, 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* $Rj \rightarrow Pi$

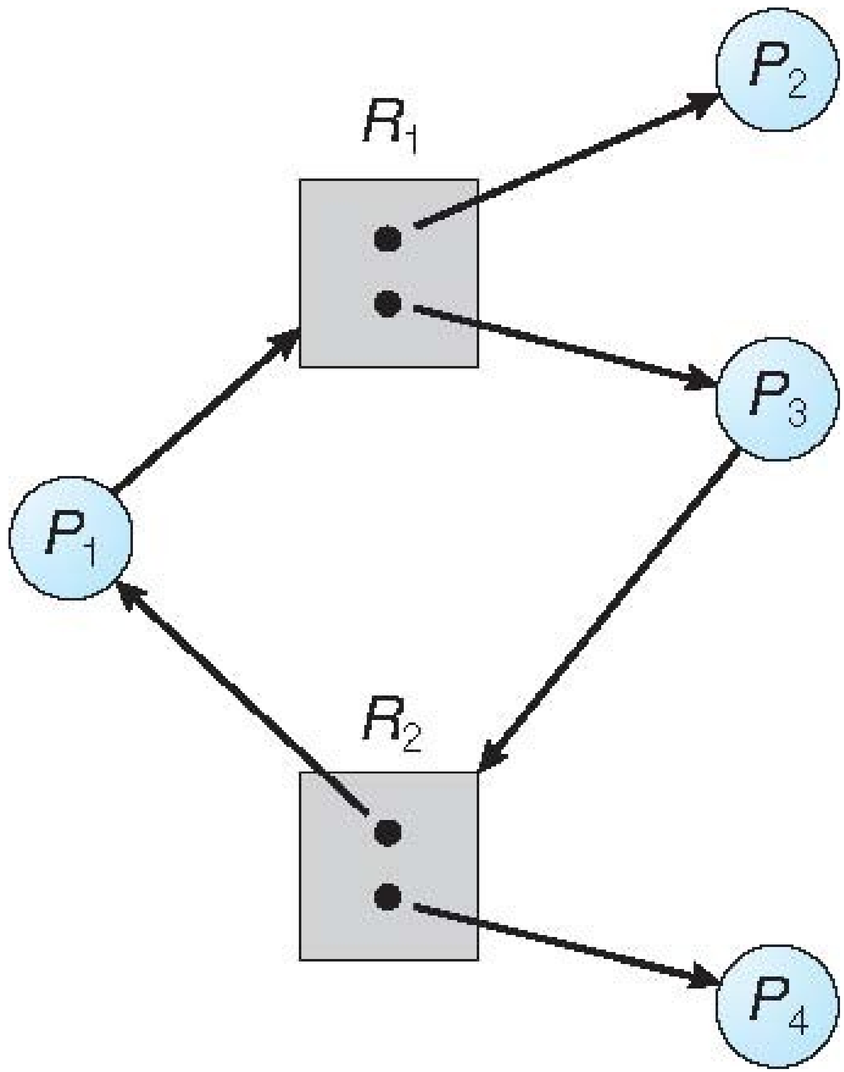


R_1
R_3
P_1
P_2
P_3
R_2
R_4

Resource Allocation Graph With Deadlock



Is there a deadlock here?



- If graph contains no cycles => no deadlock
- If graph contains a cycle =>
  - ➔ If only one instance per resource type, then deadlock
  - ➔ If several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

Deadlock problem can be dealt in one of three ways:

- Deadlock prevention, deadlock avoidance
- Deadlock detection and recovery
- Ignore the problem altogether and pretend that deadlocks never occur in the system

Deadlock Prevention

Ensure that at least one of the necessary conditions for deadlock does not hold.

- **Mutual Exclusion**- not required for sharable resources
- **Hold and Wait-**

➔ Whenever a process requests a resource, it must not hold any other

➔ May release all the resources before requesting additional resources

➔ Low resource utilization

➔ Starvation is possible

Deadlock Prevention

- **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.

Deadlock Prevention

- **Circular Wait**

➔ Impose a total ordering of all resource types.

➔ Requirement- each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

- OS requires additional information in advance concerning which resources a process will request and use during its lifetime.
- Requires that process declare the maximum number of resources of each type that it may need.
- Dynamically examines resource allocation state to ensure that circular-wait never holds.
- Resource allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

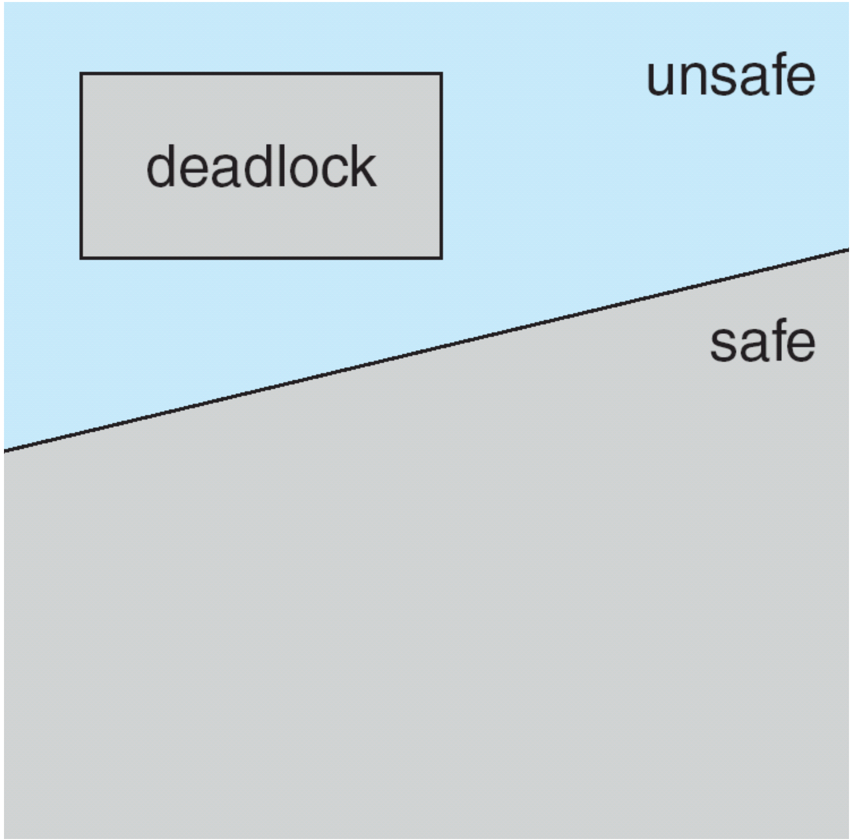
Safe State

- A state is ***safe*** if the system can allocate all resources requested by all processes (upto their stated maximums) without entering a deadlock state.
- System is in ***safe state*** if there exists a ***safe sequence***

if there exists a sequence $<P_1, P_2, \ldots, P_n>$ *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*, with* $j < i$

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



Avoidance Algorithms

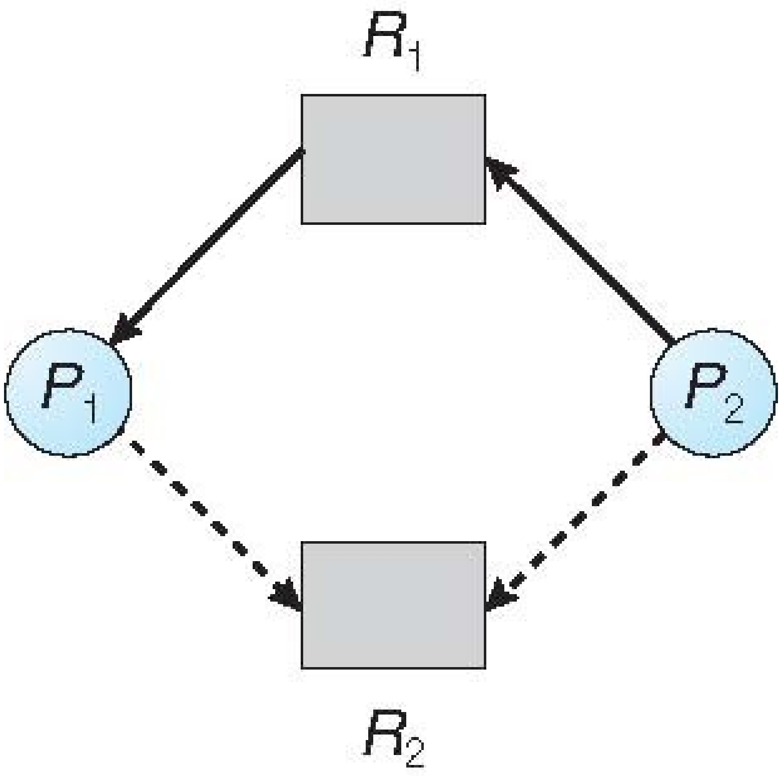
- Single instance of a resource type => use a resource-allocation graph
- Multiple instances of a resource type => use the banker's algorithm

Resourc Allocation Graph Scheme

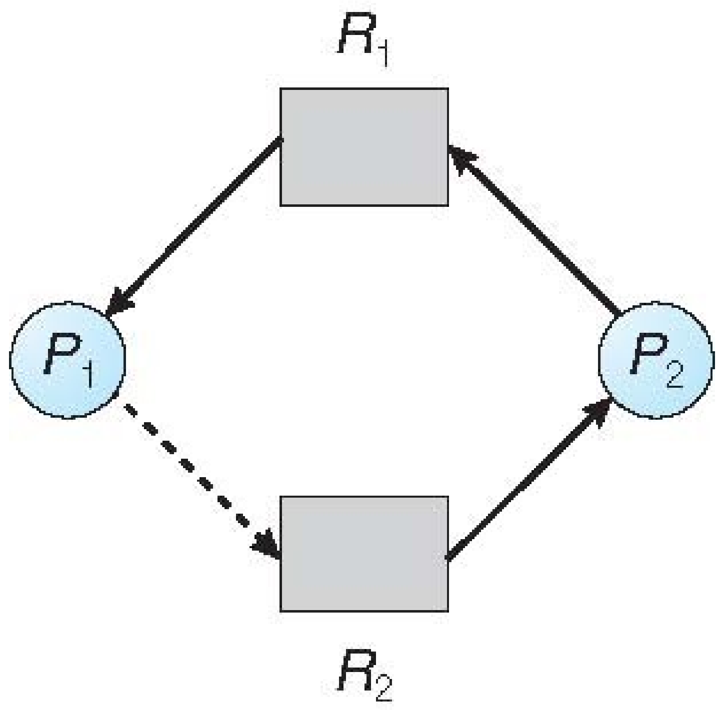
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge is converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge

Resources must be claimed *apriori* in the system

Resource-Allocation Graph



Unsafe State In 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 of resources
- Each process must declare maximum number of each resource type required apriori.
- 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

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 Pi is currently allocated k instances of Rj
- **Need**: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj 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

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i\,[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \leq Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

   $\boldsymbol{Available = Available - Request_i;}$

   $\boldsymbol{Allocation_i = Allocation_i + Request_i;}$

   $\boldsymbol{Need_i = Need_i - Request_i;}$

If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$

If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

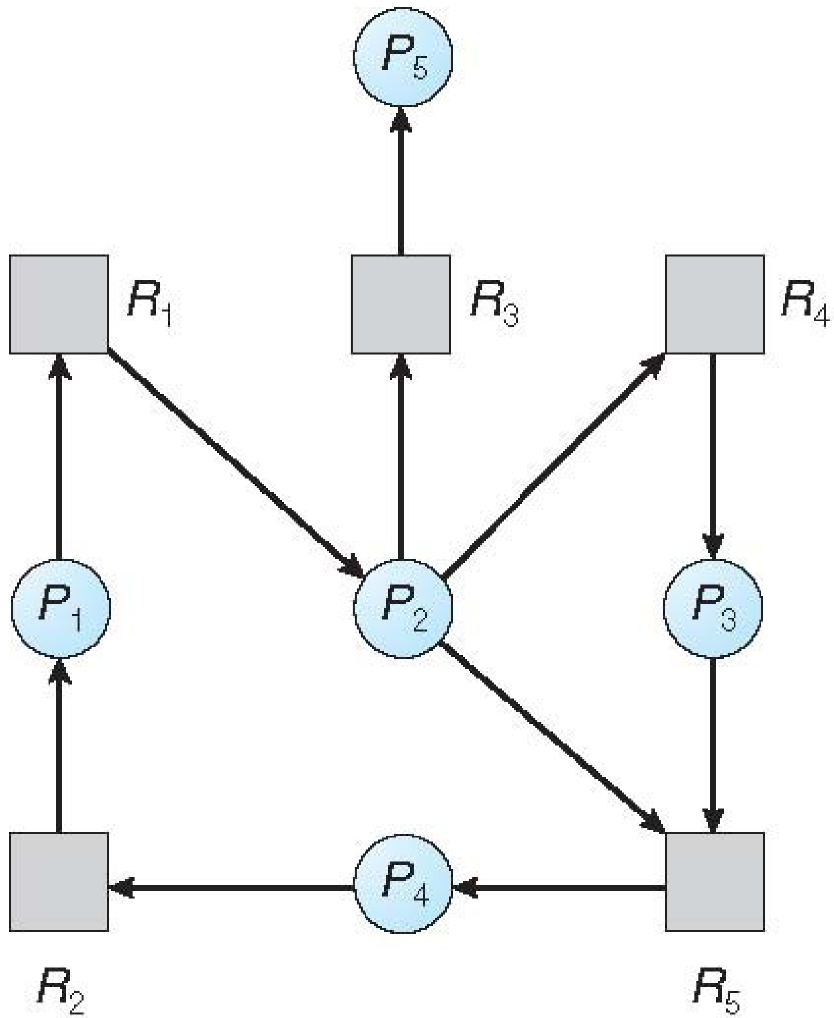
Deadlock Detection

- Allow system to enter deadlock state
- Detect
- Recover

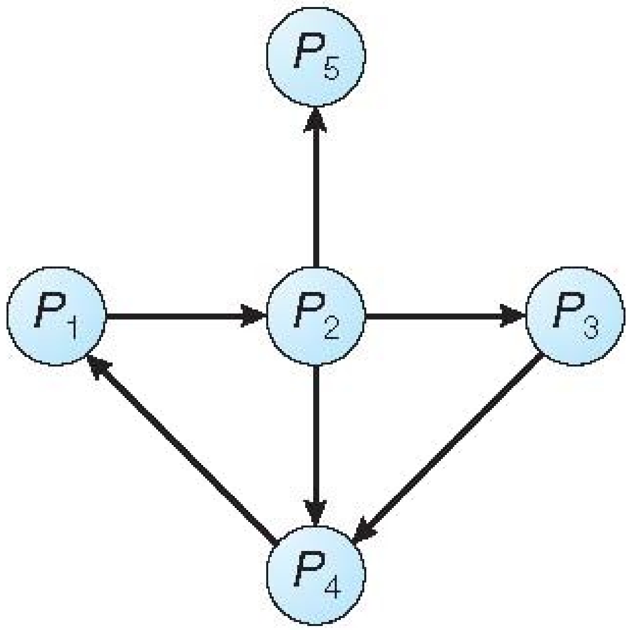
Single Instance of Each Resource Type

- 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 and Wait-for Graph



(a)



(b)

Resource-Allocation Graph

Corresponding wait-for graph

Data Structures

- **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.

Detection Algorithm

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

   (a) ***Work = Available***

   (b) 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) $\boldsymbol{Request_i \leq Work}$

   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***, 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

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 the deadlock?

  => one for each disjoint cycle

- 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.

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?
  - ➔ Priority of the process
  - ➔ How long process has computed, and how much longer to completion
  - ➔ Resources the process has used
  - ➔ Resources process needs to complete
  - ➔ How many processes will need to be terminated
  - ➔ Is process interactive or batch?

Resource Preemption

Successively preempt some processes and give the resources to other processes until the deadlock cycle is broken.

Issues:

- **Selecting a victim**- select a process to terminate. Which one?
- **Rollback**- return to some safe state. What to do with rollbacked process?

  restart process from that state
- **Starvation**- same process may always be picked as victim. Include number of rollback in cost factor (solution).

  A process can be picked as a victim only a finite number of times.