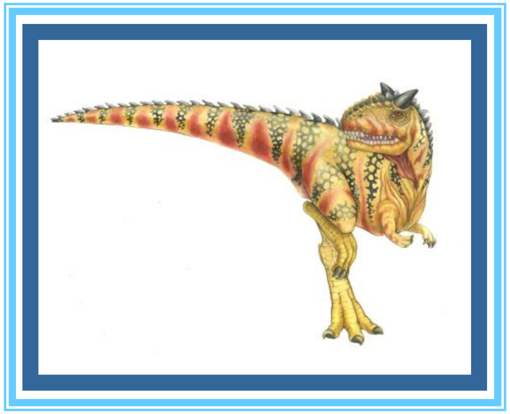
Chapter 7: Deadlocks



Deadlocks

- Necessary conditions
- Resource allocation graphs
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

- CO4 : Explain any one method for detection, prevention, avoidance and recovery for managing deadlocks in Operating Systems.

Deadlock

EXAMPLES:

- You can't get a job without experience; you can't get experience without a job.

BACKGROUND:

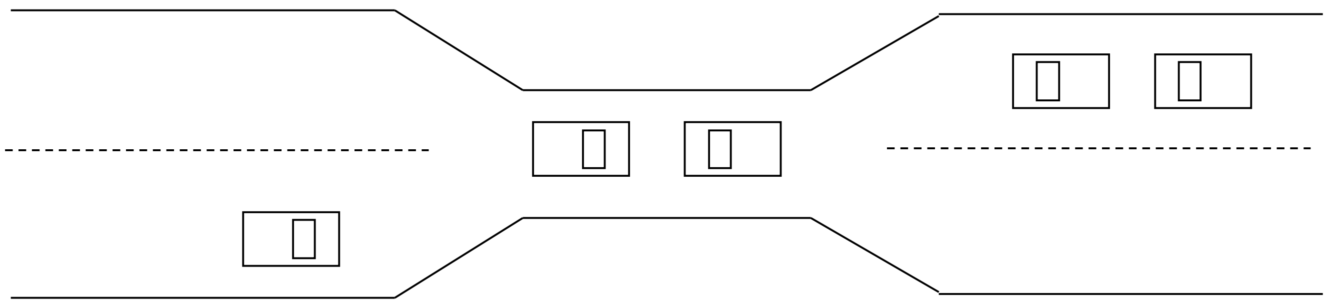
The cause of deadlocks: Each process needing what another process has.

Examples of computer resources

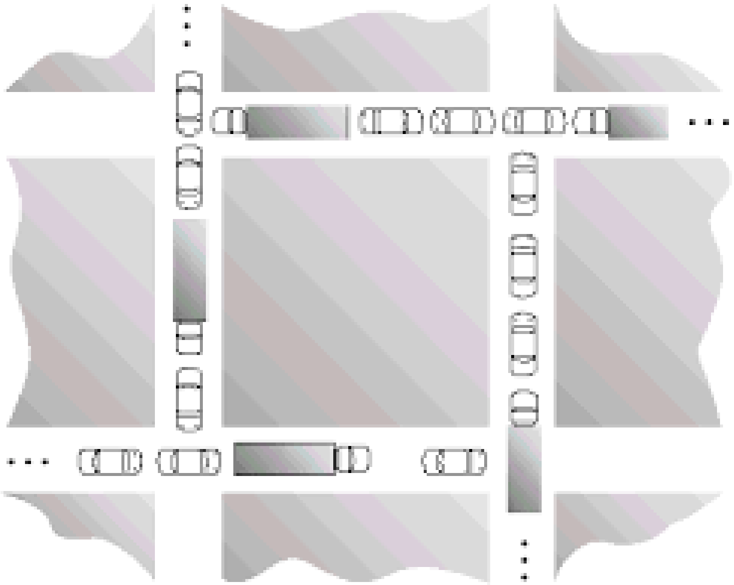
- printers
- tape drives
- tables

- Suppose a process holds resource A and requests resource B
  - at same time another process holds B and requests A
  - both are blocked and remain so

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.



System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:

1. **Request a resource** (suspend until available if necessary).
2. **Use the resource.**
3. **Release the resource.**

Introduction to Deadlocks

- Formal definition :
  A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Usually the event is release of a currently held resource
- None of the processes can …
  - run
  - release resources
  - be awakened

Deadlock Characterization

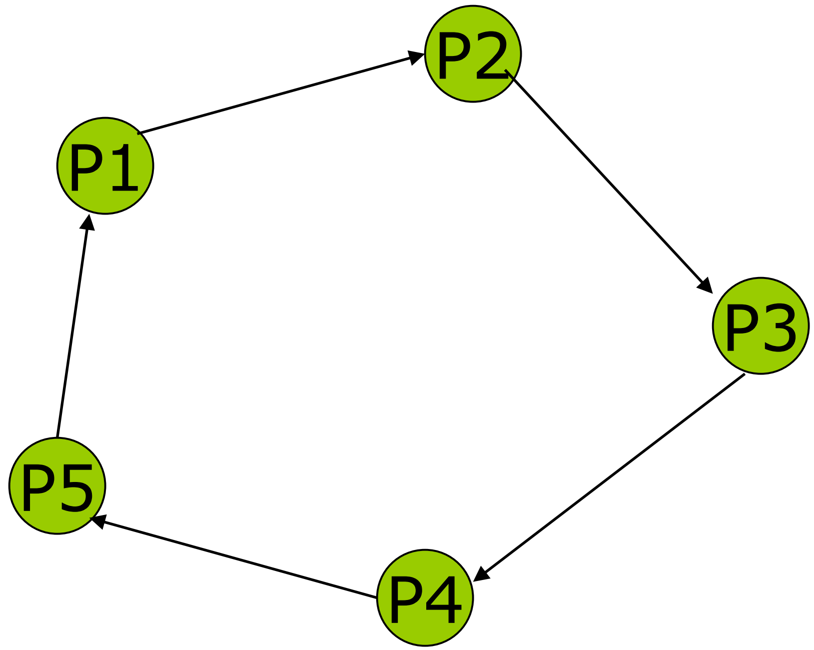
Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: 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 $\{P_0, P_1, \ldots, P_n\}$ 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.

Graph-theoretic models

- Wait-for graph.
- Resource-allocation graph.

Wait-for graph



Resource-Allocation Graph

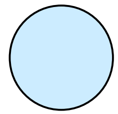
A set of vertices V and a set of edges E.

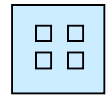
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system

    $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

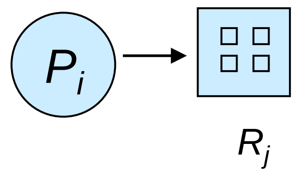
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

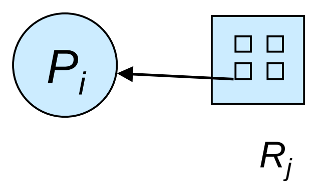
- Process 

- Resource Type with 4 instances 

- P_i requests instance of R_j

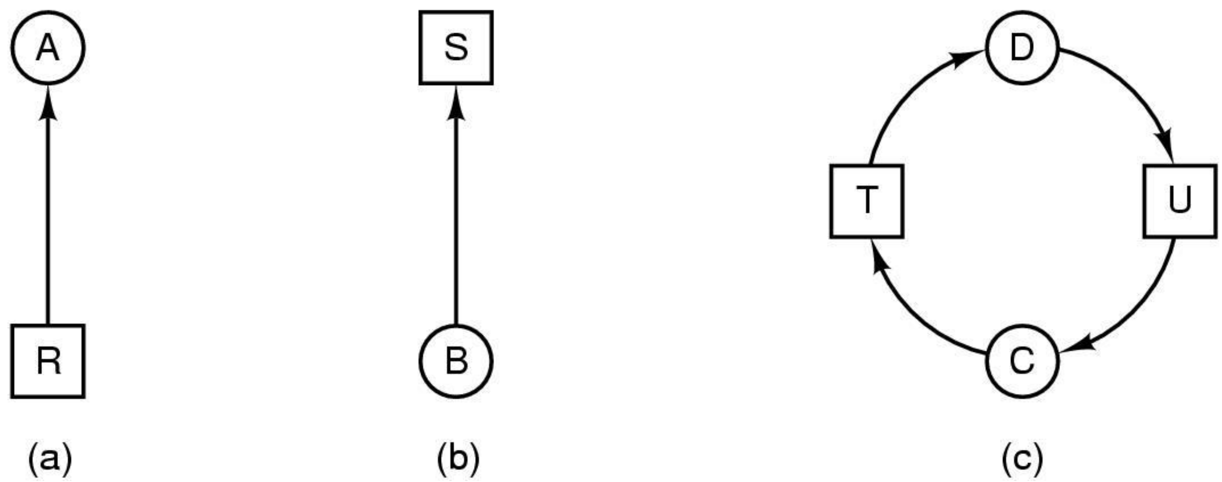


- P_i is holding an instance of R_j



Deadlock Modeling

- Modeled with directed graphs



- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U

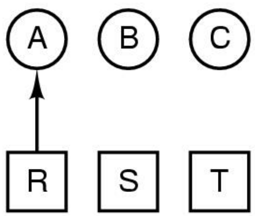
Deadlock Modeling

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

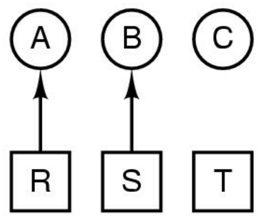
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R

deadlock

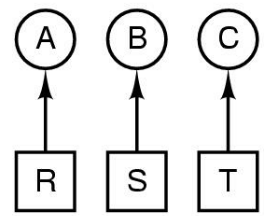
(d)



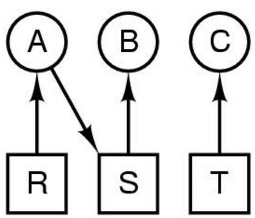
(e)



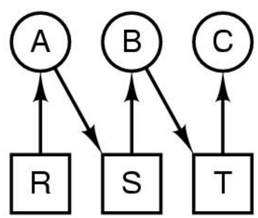
(f)



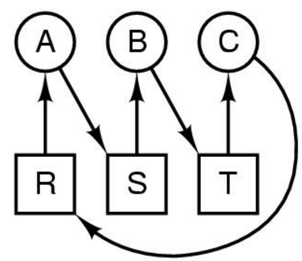
(g)



(h)



(i)



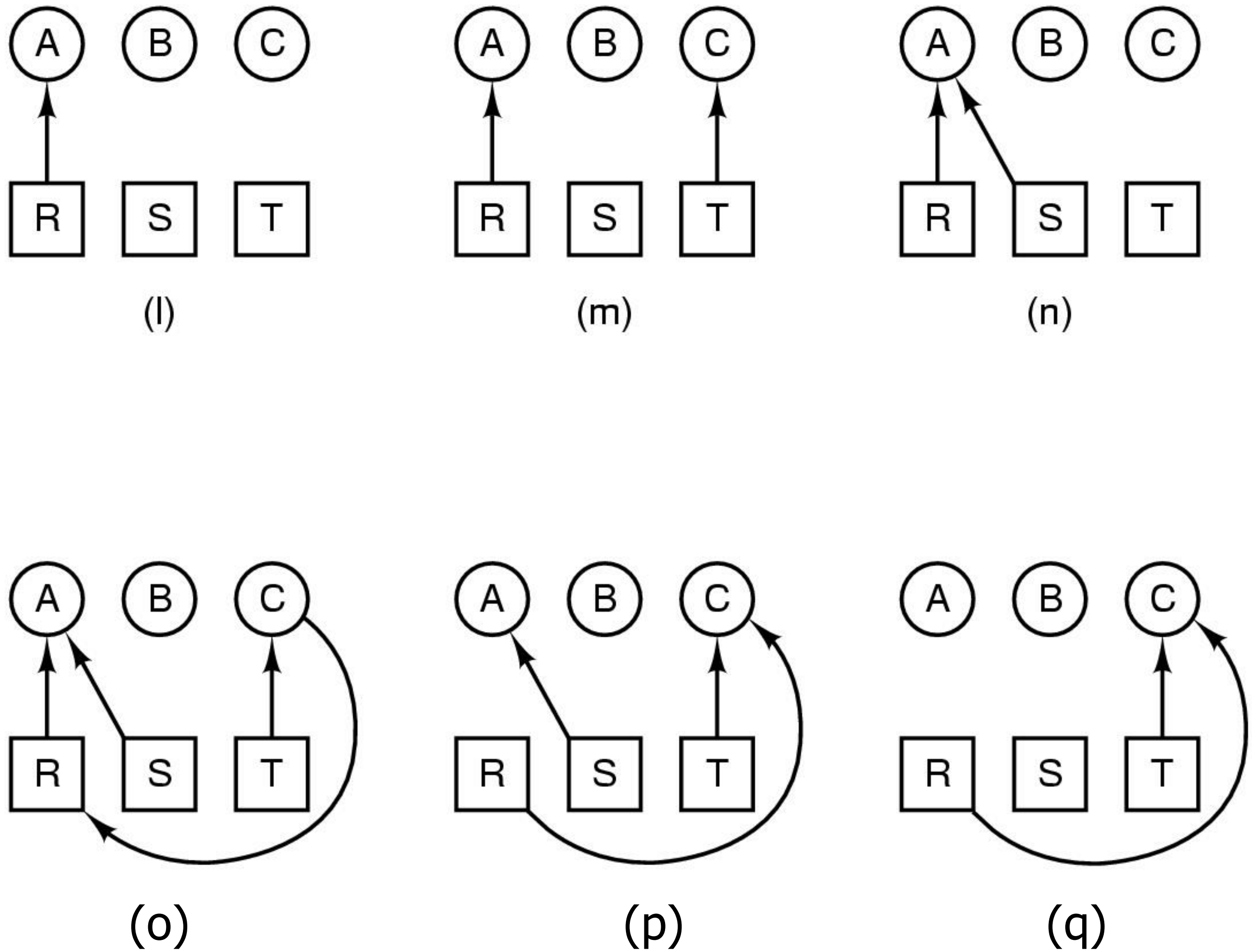
(j)

How deadlock occurs

Deadlock Modeling

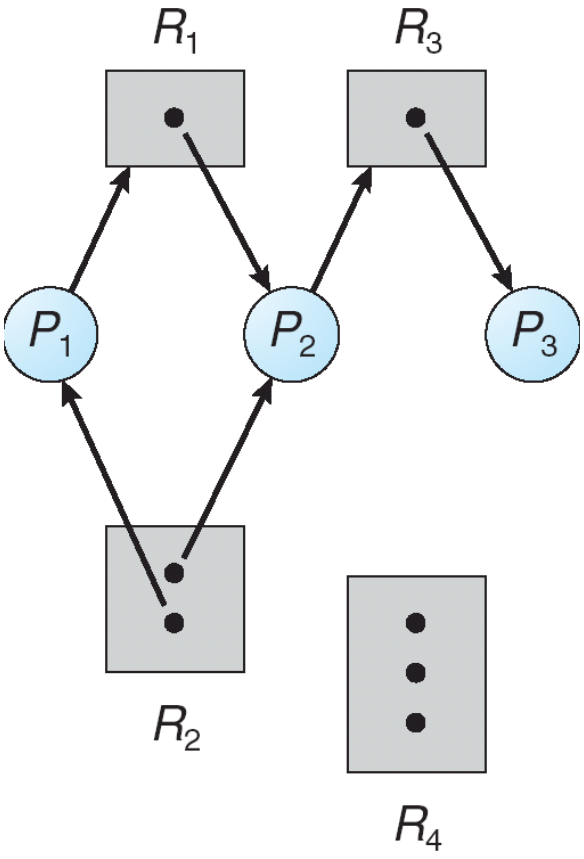
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

(k)

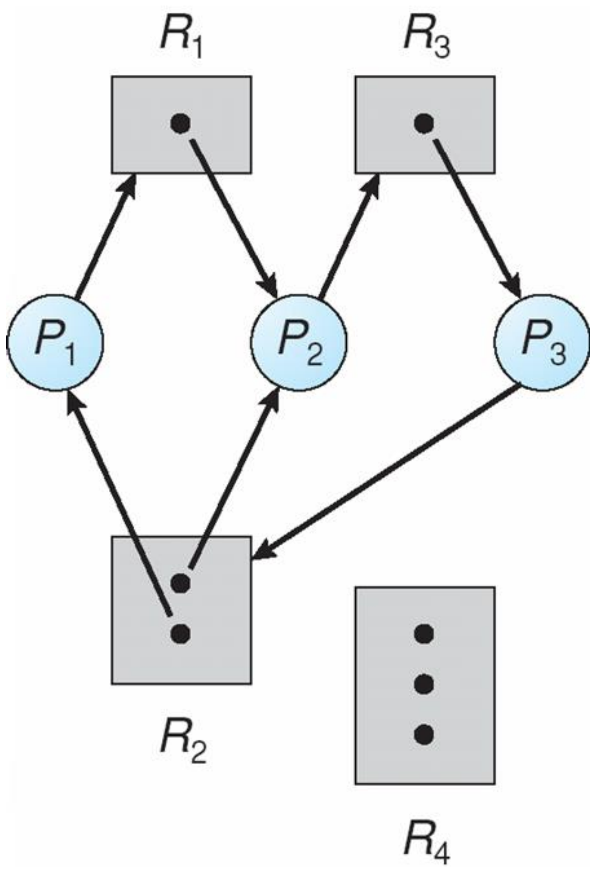


How deadlock can be avoided

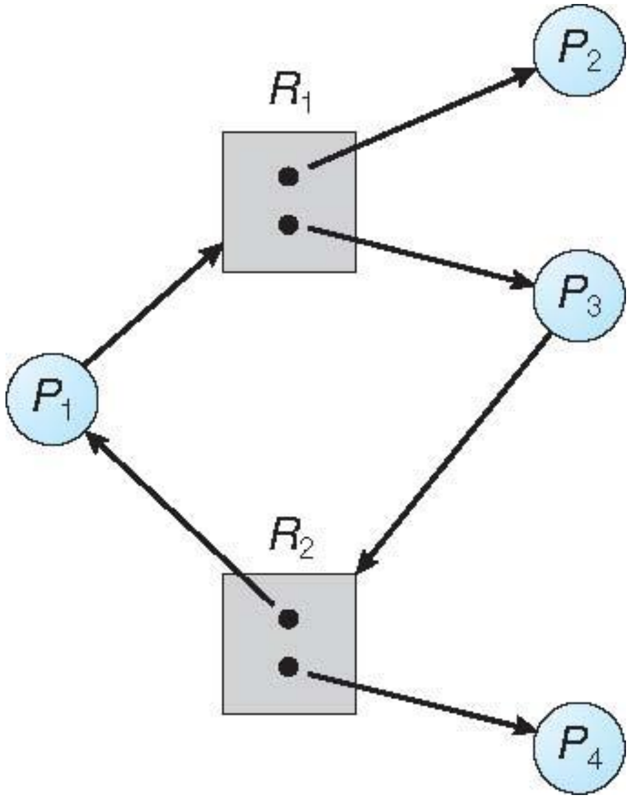
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock

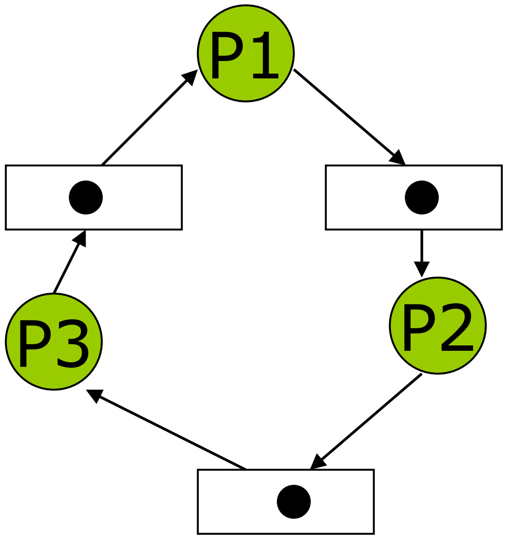


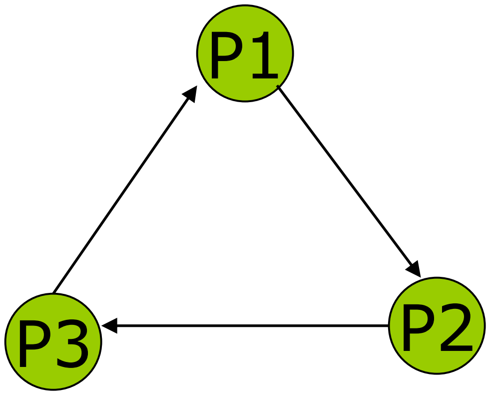
Basic Facts

- 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

Wait-for graph and Resource-allocation graph conversion

- Any resource allocation graph with a single copy of resources can be transferred to a wait-for graph.





Methods for Handling Deadlocks

There are three methods:

Ignore Deadlocks: ← **Most Operating systems do this!!**

Ensure deadlock **never** occurs using either

Prevention Prevent any one of the 4 conditions from happening.

Avoidance Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations..

Allow deadlock to happen. This requires using both:

Detection Know a deadlock has occurred.

Recovery Regain the resources.

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
  - convenience
  - correctness

Deadlock Prevention

Do not allow one of the four conditions to occur.

Mutual exclusion:

a) Automatically holds for printers and other non-sharables.

b) Shared entities (read only files) don't need mutual exclusion (and aren’t susceptible to deadlock.)

c) Prevention not possible, since some devices are intrinsically non-sharable.

Deadlock Prevention

Do not allow one of the four conditions to occur.

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

Attacking the Hold and Wait Condition

- Require processes to request resources before starting
  - a process never has to wait for what it needs
- Problems
  - may not know required resources at start of run
  - also ties up resources other processes could be using
- Variation:
  - process must give up all resources
  - then request all immediately needed

Deadlock Prevention

Do not allow one of the four conditions to occur.

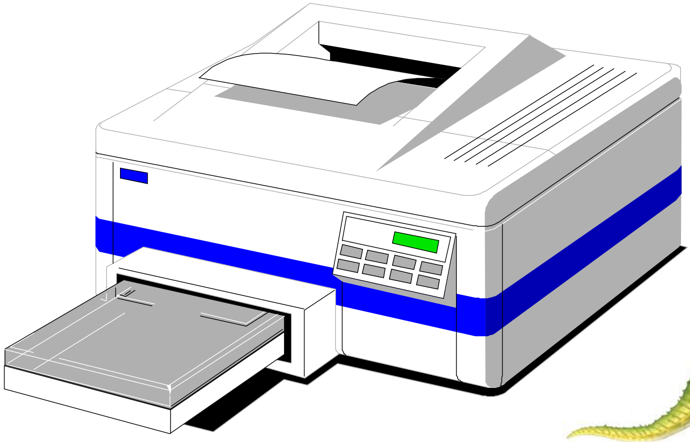
No preemption:

- 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

- Alternatively, if a process requests some resources, we first check whether they are available.
    - Allocate them.
    - Check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt them
- If the resources are neither available nor held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted.
- A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Attacking the No Preemption Condition

- This is not a viable option
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



Deadlock Prevention

Do not allow one of the four conditions to occur.

Circular wait:

- impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

No circular wait

- Let $R = \{R1, R2, ..., Rm\}$ be the set of resource types.
- Assign to each resource type a unique integer number F: $R \rightarrow N$
  - F(tape drive) = 1 F(disk drive) = 5 F(printer) = 12
- Each process can request resources only in an increasing order of enumeration.
- A process can initially request any number of instances of a resource type —say, Ri . After that, the process can request instances of resource type Rj if and only if $F(Rj) > F(Ri)$.
- A process requesting an instance of resource type Rj must have released any resources Ri such that $F(Ri) \geq F(Rj)$.

No circular wait

- If these two protocols are used, then the circular-wait condition cannot hold.
- Proof by contradiction
- Let the set of processes involved in the circular wait be $\{P_0, P_1, \ldots, P_n\}$, where P_i is waiting for a resource R_i , which is held by process P_{i+1}.
- P_n is waiting for a resource R_n held by P_0.
- Since process P_{i+1} is holding resource R_i while requesting resource R_{i+1}, we must have $F(R_i) < F(R_{i+1})$ for all i.
- But this condition means that $F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$.
- By transitivity, $F(R_0) < F(R_0)$, which is impossible.
- Therefore, there can be no circular wait.

- Each of these prevention techniques may cause a decrease in utilization and/or resources.
- For this reason, prevention isn't necessarily the best technique.
- Prevention is generally the easiest to implement.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Summary of approaches to deadlock prevention

Deadlock Avoidance

- If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.
- Possible states are:
  - **Deadlock** No forward progress can be made.
  - **Unsafe state** A state that **may** allow deadlock.
  - **Safe state** A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.
- The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.
- **NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.**

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

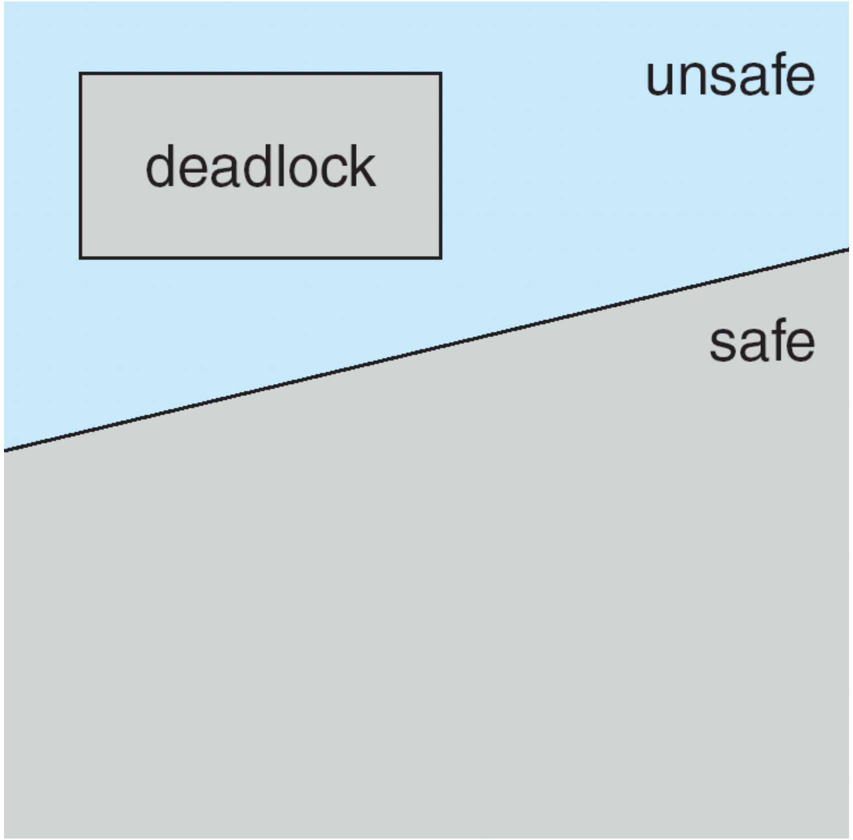
Safe State

- 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 sequence $<P_1, P_2, \ldots, P_n>$ 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

Basic Facts

- 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.

Safe, Unsafe, Deadlock State



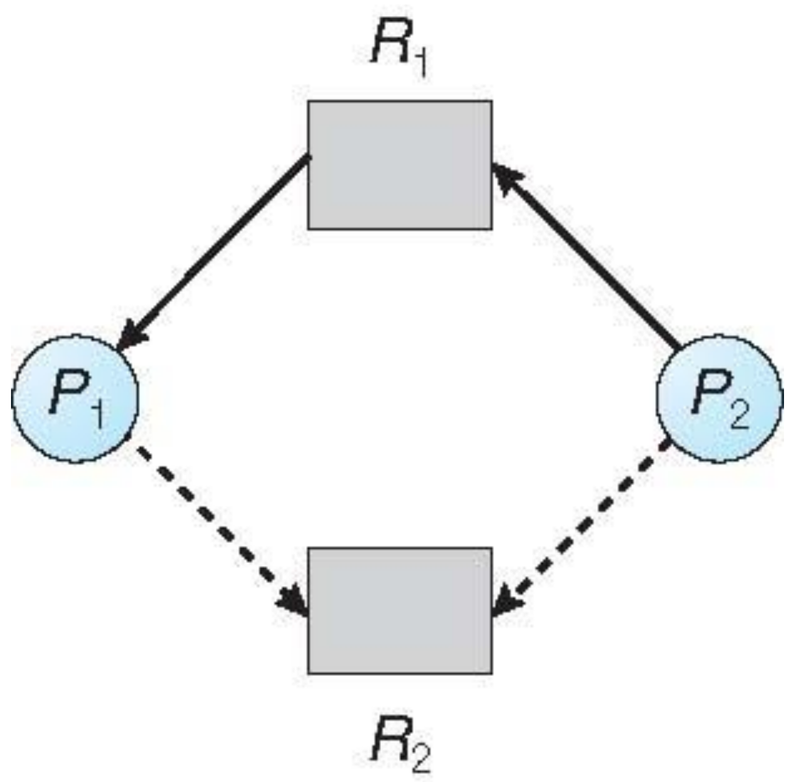
Avoidance Algorithms

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

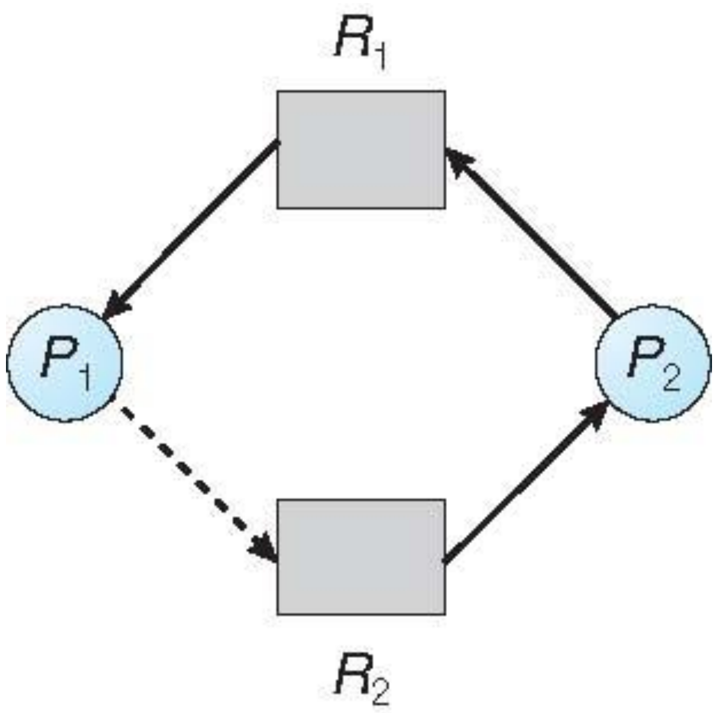
Resource-Allocation Graph Scheme

- **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 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



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

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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Banker's Algorithm

- Banker's Algorithm is used to determine whether a process's request for allocation of resources be safely granted immediately. or
- The grant of request be deferred to a later stage.

- For the banker's algorithm to operate, each process has to a priori specify its maximum requirement of resources.

- A process is admitted for execution only if its maximum requirement of resources is within the system capacity of resources.

- The Banker's algorithm is an example of resource allocation policy that avoids deadlock.

Safe State

- 1 resource with 12 units of that resource available.
- Current State: Free = (12 – (5 + 2 + 2)) = 3

	Alloc	Max.need	Still Need	
P0	5	10	5	
P1	2	4	2	
P2	2	9	7	

- This state is safe because, there is a sequence (P1 followed by P0 followed by P2) by which max needs of each process can be satisfied.
- This is called the reduction sequence.

- What if P2 requests 1 more and is allocated 1 more?

	Alloc	Max.need	Still Need
P0	5	10	5
P1	2	4	2
P2	2	9	7

	Alloc	Max.need	Still Need
P0	5	10	5
P1	2	4	2
P2	3	9	6

Unsafe State

- What if P2 requests 1 more and is allocated 1 more?

	Alloc	Max.need	Still Need
P0	5	10	5
P1	2	4	2
P2	3	9	6

- Only P1 can be reduced.

Free = 2 This is unsafe.

- If P0 and P2 then come and ask for their full needs, the system can become deadlocked.
- Hence, by granting P2's request for 1 more, we have moved from a safe to unsafe state.
- Deadlock avoidance algorithm will NOT allow such a transition, and will not grant P2's request immediately

Example:- Consider the following table of a system:

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R 1	R 2	R 3	R 4
P1	0	0	1	2	0	0	1	2	2	1	0	0
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	6				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

1. Compute NEED Matrix.
2. Is the system in safe state? Justify.

Solution:- Consider the following table of the system:

Process	Allocated				Max				Available			
	R1	R2	R3	R4	R1	R2	R3	R4	R 1	R 2	R 3	R 4
P1	0	0	1	2	0	0	1	2	2	1	0	0
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	6				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

1. Compute NEED Matrix = ?

Need [i] = Max[i] - Allocated[i],

Therefore,

Need Matrix

NEED MATRIX	R1	R2	R3	R4
P1	0	0	0	0
P2	0	7	5	0
P3	6	6	2	2
P4	2	0	0	0
P5	0	3	2	0

2. Is the system is Safe State?

By applying the Banker's Algorithm:

Let **Avail** = Available; i.e . Avail = {2,1,0,0}

Iteration 1. Check all processes from P1 to P5.

For P1:→

if (**P1 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P1]

= {2,1,0,0} + {0,0,1,2}

Avail = **{2,1,1,2}**

2. Is the system is Safe State?

By applying the Banker's Algorithm:

Iteration 1.

For P2:→

if (**P2 Need < Avail**)→FALSE

//then Check for next process.

For P3:→

if (**P3 Need < Avail**)→ FALSE

//then Check for next process.

2. Is the system is Safe State?

By applying the Banker's Algorithm:

Iteration 1.

For P4:→

if (**P4 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P4]

= {2,1,1,2} + {2,3,5,4}

Avail = **{4,4,6,6}**

2. Is the system is Safe State?

By applying the Banker's Algorithm:

Iteration 1.

For P5:→

if (**P5 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P5]

= {4,4,6,6} + {0,3,3,2}

Avail = **{4,7,9,8}**

2. Is the system is Safe State?

By applying the Banker's Algorithm:

Iteration 2. Check only process P2 to P3.

For P2:→

if (**P2 Need < Avail**)→TRUE

then calculate

Avail= Avail + Allocated [P2]

= {4,7,9,8} + {2,0,0,0}

Avail = **{6,7,9,8}**

Since, all the processes got TRUE marked, no further iterations are required.

Therefore, Safe Sequence = P1, P4, P5, P2 , P3

Therefore, the System is in the Safe State.

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)

- Snapshot at time T_0:

MaxNeeds

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocated

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

StillNeeds

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Free

A	B	C
3	3	2

Example (Cont.)

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

MaxNeeds

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocated

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

StillNeeds

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Free

A	B	C
3	3	2

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

MaxNeeds

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocated

	A	B	C
P0	0	1	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2

StillNeeds

	A	B	C
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Free

A	B	C
2	3	0

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Exercise: Formally go through each of the steps that update these matrices for the reduction sequence.

- After this allocation, P0 then makes a request for (0,2,0).
- If granted the resulting state would be:

MaxNeeds

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocated

	A	B	C
P0	0	3	0
P1	3	0	2
P2	3	0	2
P3	2	1	1
P4	0	0	2

StillNeeds

	A	B	C
P0	7	2	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

Free

A	B	C
2	1	0

- This is an UNSAFE state.
- So this request should NOT be granted.

	Allocation			Max		
	X	Y	Z	X	Y	Z
P0	0	0	1	8	4	3
P1	3	2	0	6	2	0
P2	2	1	1	3	3	3

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in safe state. Consider the following independent requests for additional resources in the current state-

REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z

REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z

A system has 4 processes and 5 allocatable resource. The current allocation and maximum needs are as follows-

	Allocated					Maximum				
A	1	0	2	1	1	1	1	2	1	3
B	2	0	1	1	0	2	2	2	1	0
C	1	1	0	1	1	2	1	3	1	1
D	1	1	1	1	0	1	1	2	2	0

If Available = [0 0 X 1 1], what is the smallest value of x for which this is a safe state?

Deadlock Detection

- If a system does not use either deadlock-prevention or deadlock-avoidance algorithm then a deadlock may occur.
- In this environment, the system must provide
- Detection algorithm : An algorithm to examine the system-state to determine whether a deadlock has occurred.
- Recovery scheme : An algorithm to recover from the deadlock.

Single Instance of Each Resource Type

- If all the resources have only a single instance, then deadlock detection-algorithm can be defined using a wait-for-graph.
- The wait-for-graph is applicable to only a single instance of a resource type.
- A wait-for-graph (WAG) is a variation of the resource-allocation-graph.
- The wait-for-graph can be obtained from the resource-allocation-graph by
  - → removing the resource nodes and
  - → collapsing the appropriate edges.

Single Instance of Each Resource Type

- An edge from Pi to Pj implies that process Pi is waiting for process Pj to release a resource that Pi needs.
  - An edge Pi → Pj exists if and only if the corresponding graph contains two edges
  - $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$.

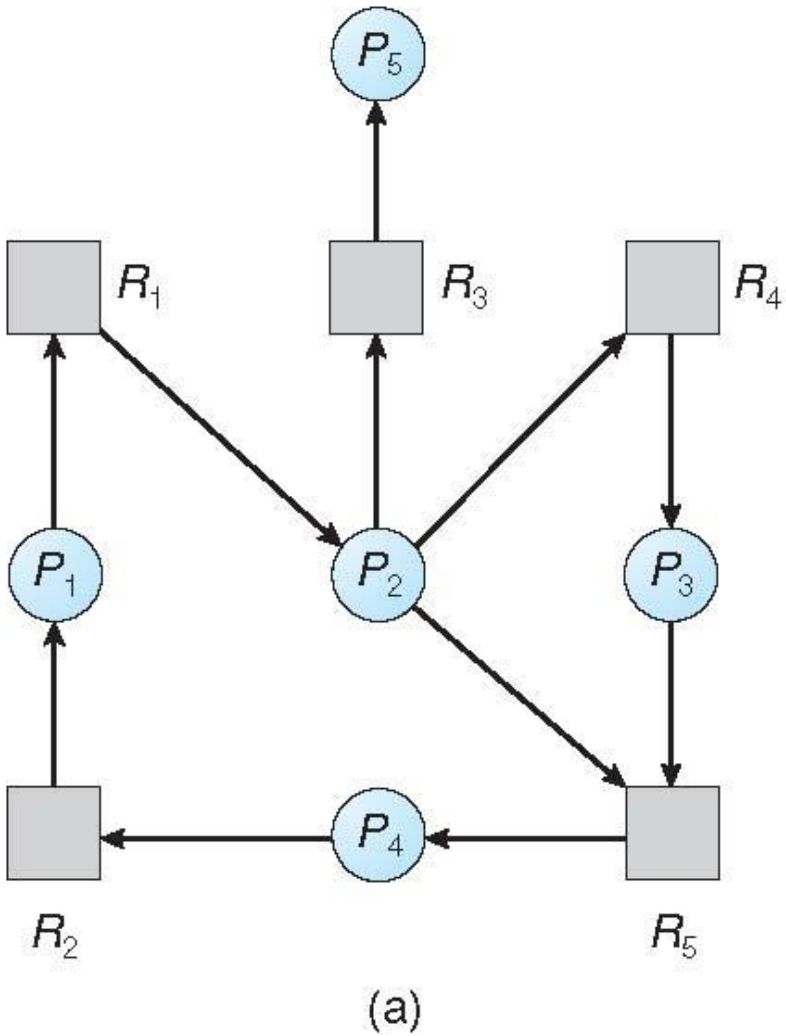
Single Instance of Each Resource Type

- A deadlock exists in the system if and only if the wait-for-graph contains a cycle.
- To detect deadlocks, the system needs to

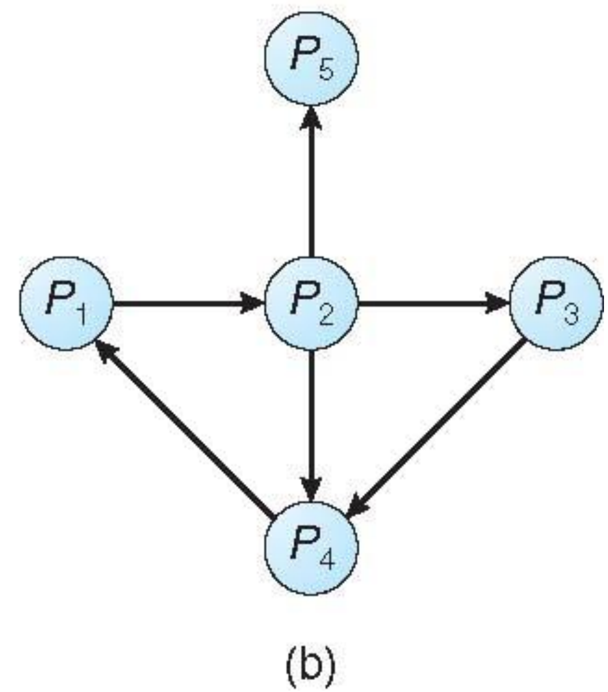
  → maintain the wait-for-graph and

  → periodically execute 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

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- The wait-for-graph is applicable to only a single instance of a resource type.
- Problem: However, the wait-for-graph is not applicable to a multiple instance of a resource type.
- Solution: The detection-algorithm can be used for a multiple instance of a resource type.
- Assumptions:
  - Let ‘n’ be the number of processes in the system
  - Let ‘m’ be the number of resources types.

Several Instances of a Resource Type

- **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.

Recap --- 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) ***$Need_i \leq Work$***

   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 in a safe state

Detection Algorithm

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

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

   (b) For ***i* = 1,2, …, *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

Detection Algorithm (Cont.)

3. ***Work = Work + Allocation$_i$***
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, $1 \leq$ ***i*** $\leq$ ***n***, then the system is in deadlock state.

   Moreover, if ***Finish*[*i*] == *false***, then ***P$_i$*** is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

----

7 2 6

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	0 0 0	0 1 0	P_0 completed
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 0		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	0 0 0	3 1 3	P_2 completed
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 0		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

Example of Detection Algorithm

- Five processes ***P*$_0$** through ***P*$_4$**; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time ***T*$_0$**:

	<u>*Allocation*</u>	<u>*Request*</u>	<u>*Available*</u>	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	0 0 0	5 2 4	P_3 completed
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 0		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	0 0 0	5 2 6	P_4 completed
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 0		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	0 0 0	7 2 6	P_1 completed
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 0		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

- Sequence <P_0, P_2, P_3, P_4, P_1> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- P_2 requests an additional instance of type ***C***

	Request		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Example

- Five processes ***P*$_0$** through ***P*$_4$**; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time ***T*$_0$**:

	Allocation	*Request*	*Available*	
	A B C	*A B C*	*A B C*	
P_0	0 1 0	0 0 0	0 1 0	P_0 completed
P_1	2 0 0	2 0 2		
P_2	3 0 3	0 0 1		
P_3	2 1 1	1 0 0		
P_4	0 0 2	0 0 2		

Example (Cont.)

- P_2 requests an additional instance of type C

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- The detection-algorithm must be executed based on following factors:
  - The frequency of occurrence of a deadlock.
  - The no. of processes affected by the deadlock.
- If deadlocks occur frequently, then the detection-algorithm should be executed frequently.
- Resources allocated to deadlocked-processes will be idle until the deadlock is broken.

- Problem:
- Deadlock occurs only when some processes make a request that cannot be granted immediately.
- Solution 1:
  - The deadlock-algorithm must be executed whenever a request for allocation cannot be granted immediately.
  - In this case, we can identify

    → set of deadlocked-processes and

    → specific process causing the deadlock.

- Solution 2:
- The deadlock-algorithm must be executed in periodic intervals.
- For example:

→ once in an hour

→ whenever CPU utilization drops below certain threshold

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

- Three approaches to recovery from deadlock:

1) Inform the system-operator for manual intervention.

2) Terminate one or more deadlocked-processes.

3) Preempt(or Block) some resources.

Process Termination

- Two methods to remove deadlocks:

1) Terminate all deadlocked-processes.

- This method will definitely break the deadlock-cycle.
- However, this method incurs great expense. This is because
  - Deadlocked-processes might have computed for a long time.
  - Results of these partial computations must be discarded.
  - Probably, the results must be re-computed later.

Process Termination

2) Terminate one process at a time until the deadlock-cycle is eliminated.

- This method incurs large overhead.
- This is because after each process is aborted, deadlock-algorithm must be executed to determine if any other process is still deadlocked

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?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Some resources are taken from one or more deadlocked-processes. These resources are given to other processes until the deadlock-cycle is broken.
- Three issues need to be considered:

1) Selecting a victim

- Which resources/processes are to be pre-empted (or blocked)?
- The order of pre-emption must be determined to minimize cost.
- Cost factors includes

1. The time taken by deadlocked-process for computation.
2. The no. of resources used by deadlocked-process.

Recovery from Deadlock: Resource Preemption

2) Rollback

- If a resource is taken from a process, the process cannot continue its normal execution.
- In this case, the process must be rolled-back to break the deadlock.
- This method requires the system to keep more info. about the state of all running processes.

Recovery from Deadlock: Resource Preemption

3) Starvation

- Problem: In a system where victim-selection is based on cost-factors, the same process may be always picked as a victim.
- As a result, this process never completes its designated task.
- Solution: Ensure a process is picked as a victim only a (small) finite number of times.