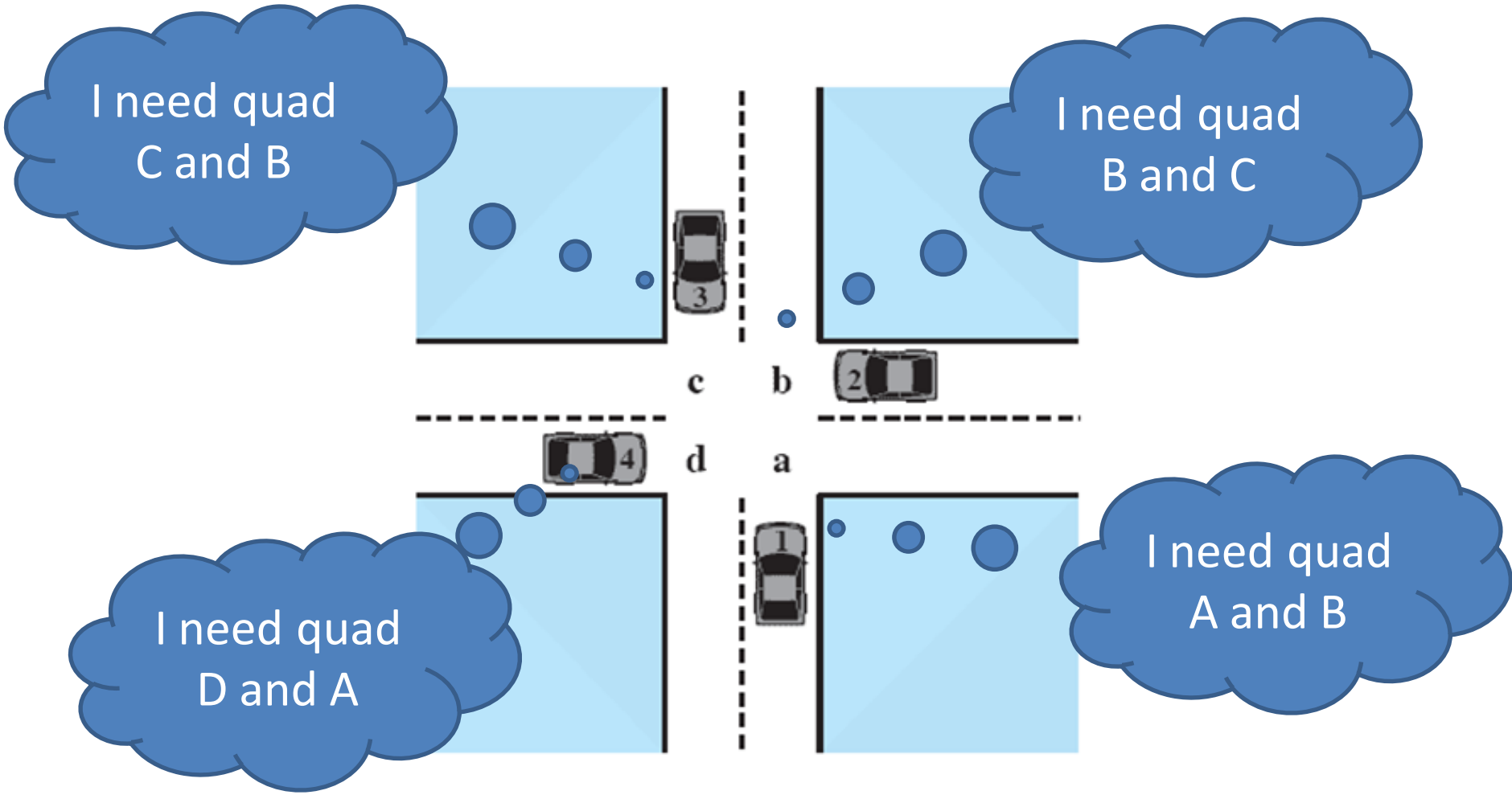
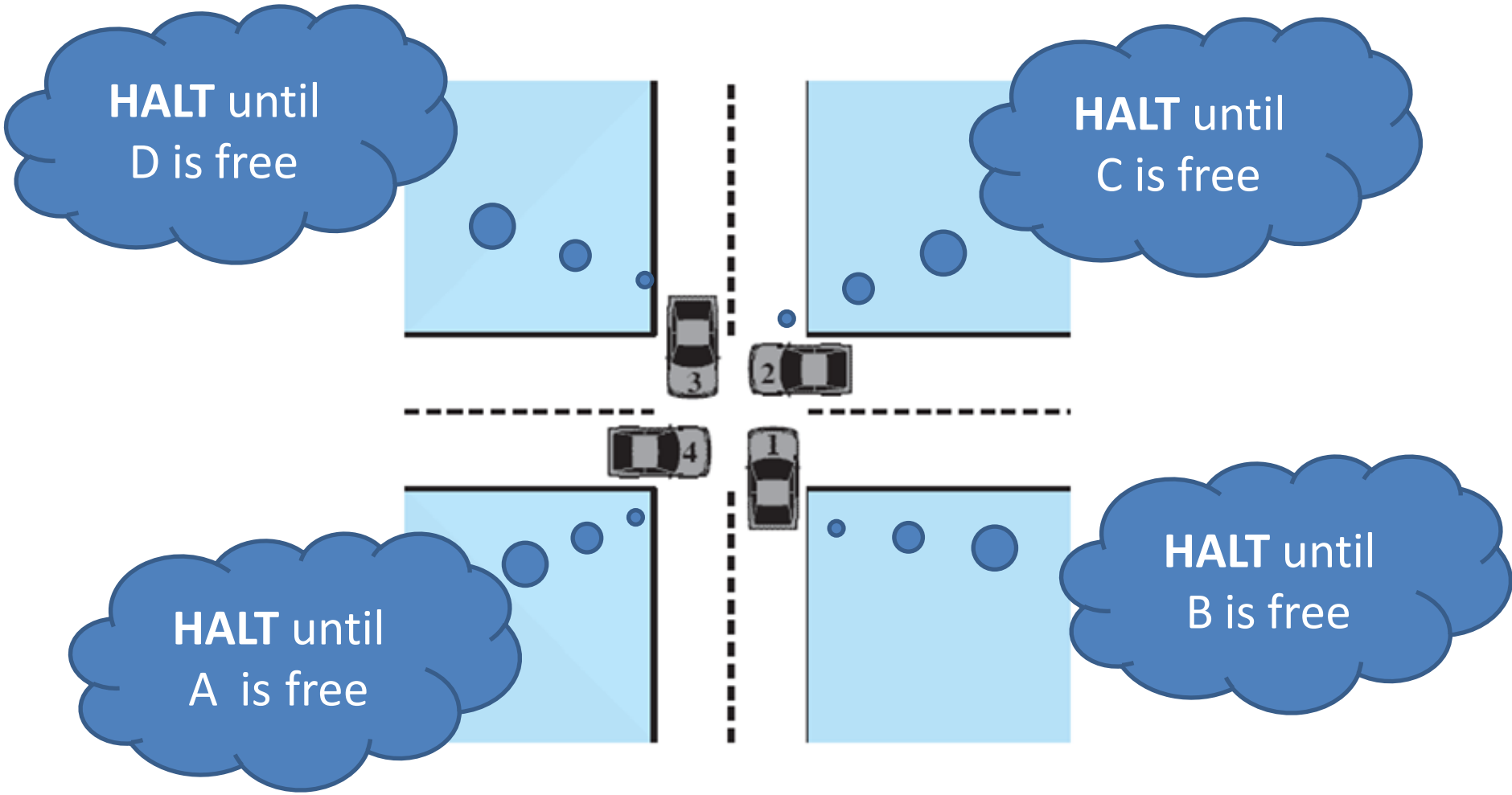
Deadlocks

- Principles of deadlock,
- Deadlock Prevention,
- Deadlock Avoidance,
- Deadlock Detection,
- Deadlock Recovery.

Potential Deadlock



Actual Deadlock



Example of deadlock

Dining Philosophers Problem

Example of deadlock

Boss said to secretary: For a week we will go abroad, so make arrangement.
Secretary make call to Husband: For a week my boss and I will be going abroad, you look after yourself.
Husband make call to secret lover: My wife is going abroad for a week, so lets spend the week together.
Secret lover make call to small boy whom she is giving private tution: I have work for a week, so you need not come for class.
Small boy make call to his grandfather: Grandpa, for a week I don't have class 'coz my teacher is busy. Lets spend the week together.
Grandpa make call to his secretary: This week I am spending my time with my grandson. We cannot attend that meeting.
Secretary make call to her husband: This week my boss has some work, we cancelled our trip.
Husband make call to secret lover: We cannot spend this week together, my wife has cancelled her trip.
Secret lover make call to small boy whom she is giving private tution: This week we will have class as usual.
Small boy make call to his grandfather: Grandpa, my teacher said this week I have to attend class. Sorry I can't give you company.
Grandpa make call to his secretary: Don't worry this week we will attend that meeting, so make arrangement .

DEADLOCKS

- Permanent blocking of a single or set of processes, competing for system resources or may want to cooperate for communication.
- 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.
- Generally it is because of the conflicting needs of different processes.
- There is no general solution to solve it completely.

Resource Categories

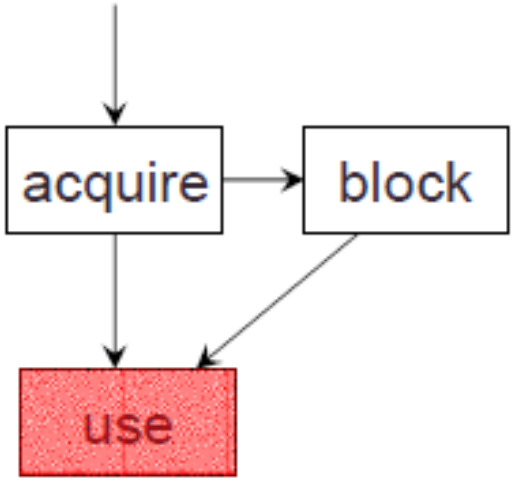
- Reusable resources vs. Consumable resources
- Physical vs. logical resources
- Preemptable resources vs. Non preemptable resources

Reusable Resources

- Can be safely used by only one process at a time and is not depleted by that use
- Processors, I/O channels, memory, devices and data structure such as database, files,semaphores etc.
- Examples of deadlock with reusable resources
  - If each process holds on resource and requests for the other
  - Dining Philosophers
- General access pattern:
  - Request
  - Lock
  - Use
  - Release

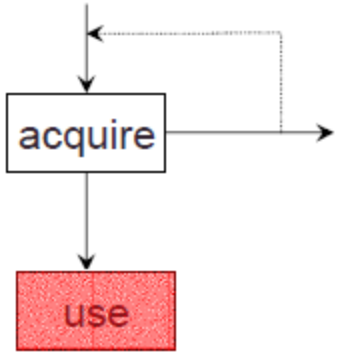
Resources

- Process must wait if request is denied
  - ❑ Requesting process may be blocked
  - ❑ May fail with error code



- Deadlocks

❑ Occur only when processes are granted exclusive access to resources



Example of accessing Disk file and Tape drive

Another Example

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1

. . .

Request 80 Kbytes;

. . .

Request 60 Kbytes;

P2

. . .

Request 70 Kbytes;

. . .

Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources

- One that can be created/produced and destroyed/ consumed
- Typically no limit on the number of consumable resources of a particular type
- Resource ceases to exist after consumption
- Examples
  - Interrupts, signals, messages, contents of I/O buffers
- Can there be deadlock involving consumable resources?

Example

- Each process attempts to receive a message from the other and then sends a message to it

```
P1
. . .
Receive(P2);
. . .
Send(P2, M1);
```

```
P2
. . .
Receive(P1);
. . .
Send(P1, M2);
```

- Deadlock occurs if the “receive” is blocking
- Takes a rare combination of events!
  - Ever heard that a s/w is never bug free?

Other Kinds

- Physical
  - Printer, tape drive
- Logical
  - File, semaphore, data structure
- Preemptable
  - Can be taken away from process for some time with no ill effects
  - CPU,memory
- Non Preemptable
  - Will cause process to fail if taken away
  - Printer
- Resource type (e.g. Printer) vs resource instances (e.g. 2)

Deadlock Example

- utility program

❑ Copies a file from a tape to disk

❑ Prints the file to a printer

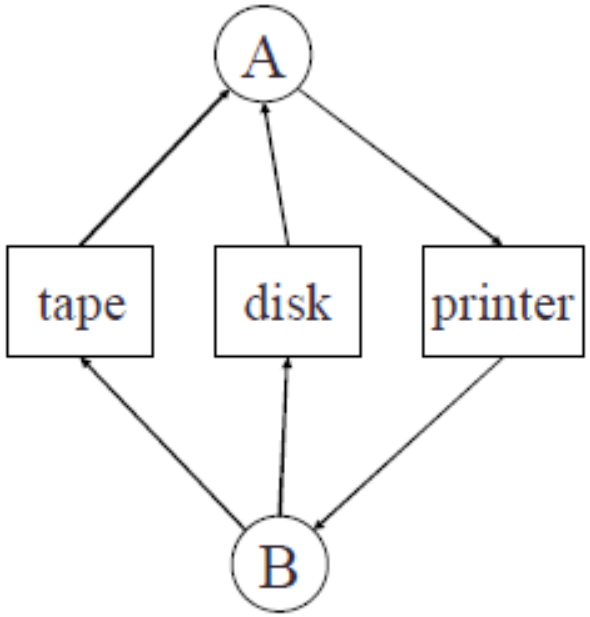
- Resources

❑ Tape

❑ Disk

❑ Printer

A deadlock



System Model

- Resource types R_1, R_2, . . ., 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**

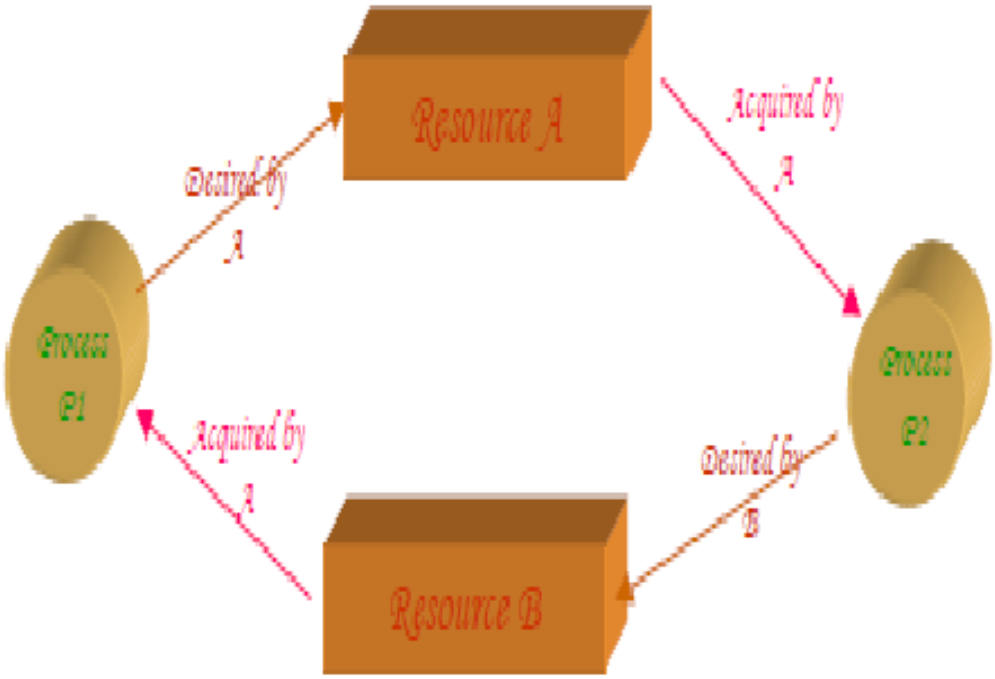
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

1. **Mutual exclusion** :Only single process is allowed to use the resource.
2. **Hold and wait** :Process holding at least one resource and waiting to acquire additional resources currently held by other processes.
3. **No preemption** :No resource can be removed forcibly from a process.
4. **Circular wait:** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

- First three are necessary but not sufficient conditions for a deadlock to exist

Resource-Allocation Graph



Resource-Allocation Graph

- Characterizes allocation of resources to processes.
- Directed graph to describe deadlocks
- 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 (Symbols)

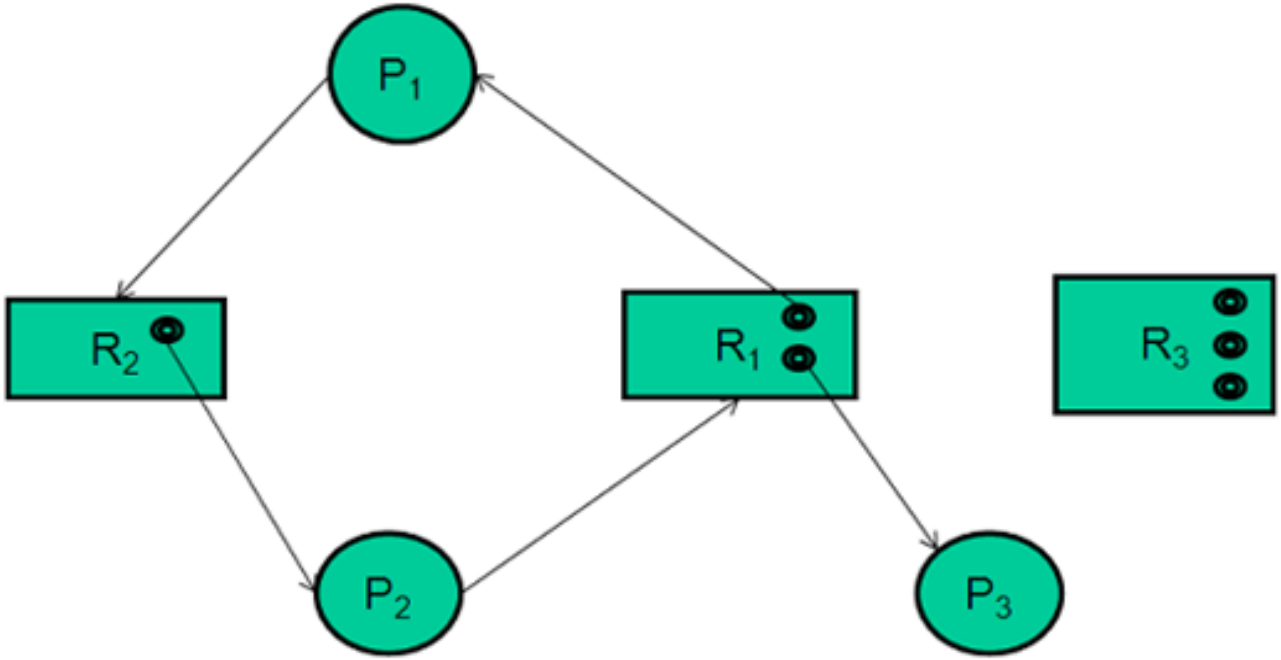
- Process node

- Resource node

- Assignment edge

- Request edge

Example of a Resource allocation graph



Process node

Assignment edge

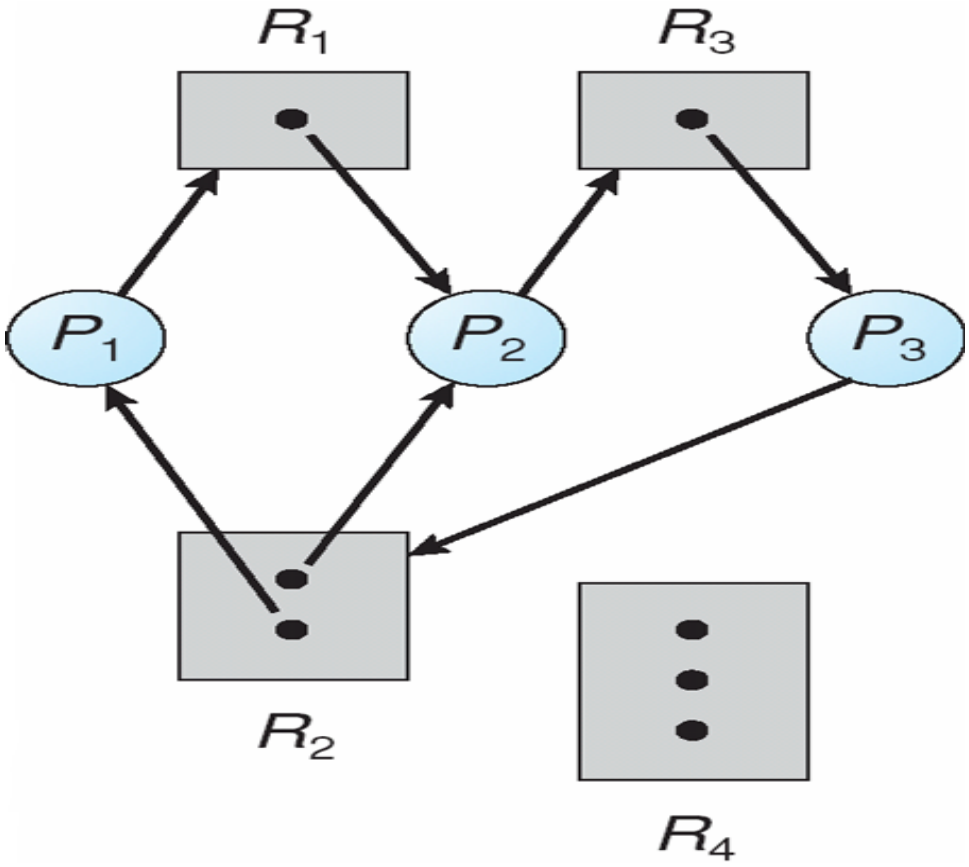
Resource node

Request edge

Case Study

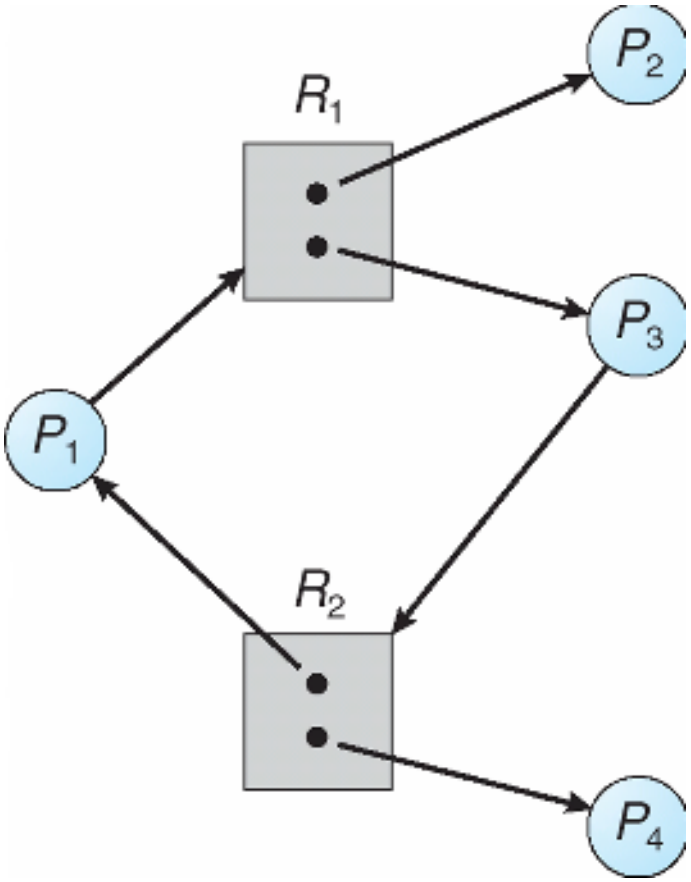
Case Study

And this?



Yes deadlock

… and this?



Cycle but no deadlock

Basic Facts

- Deadlock cycle
- No cycle no deadlock
- Cycle deadlock
  - if only one instance per resource type
  - if several instances per resource type then there is a possibility of a deadlock
- If there is single instance of each resource type then cycle in the RAG is necessary and sufficient condition for the existence of a deadlock
- If each resource type has multiple instances ,then a cycle is a necessary but not sufficient condition for the existence of a deadlock

Dealing with deadlocks

- Four general approaches
  - Deadlock **prevention** – by adopting a policy that eliminates one of the four conditions
  - Deadlock **avoidance** – by making the appropriate dynamic choices based on the current state of resource allocation
  - Deadlock **detection** and **recovery** – attempt to detect presence of deadlock and take actions to recover
  - **Ignore** the problem and pretend that deadlocks never occur in the system
    - Used by most operating systems, including UNIX & windows
    - Remains a programmer's responsibility to write deadlock free code

Deadlock prevention

- Design a system in such a way that the possibility of deadlock is excluded by ensuring that at least one of the necessary conditions cannot hold
- Two main methods

1. Indirect – prevent all three of the necessary conditions occurring at once (Mutual exclusion, Hold-wait,No-preemption)
2. Direct – prevent Circular wait

Deadlock Prevention: Deny Mutual Exclusion

- **Mutual Exclusion** – not required for sharable resources;
- must hold for non-sharable resources.
- A process never needs to wait for a sharable resource
- But in reality, some resources are intrinsically non-sharable and hence we cannot prevent deadlock by denying mutual exclusion
- If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS.
- Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes.

Deadlock Prevention:Disable hold & wait

- Must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Requires each process to request and be allocated all its resources before it begins execution
  - Allow process to request resources only when the process has none

  Disadvantages:
  - Low resource utilization
  - Starvation is possible
  - Process may not know in advance all of the resources that it will require.

Example

- Process that copies data from DVD drive to a file on disk, sorts the file and prints the results to a printer
- First protocol
  - First request DVD drive, disk file and printer
  - Do the job
  - Obvious demerit?
- Second protocol
  - Request DVD drive and disk file
  - Copy, release
  - Request disk file and printer
  - Print, release
  - Problem: data may have changed by now
- Starvation common to both
  - A process that needs several popular resources may have to wait indefinitely because at least one of the resources it needs is always allocated to some other process

Deadlock Prevention: Enable Pre-emption

- If a process that is holding some resources and 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

Enable preemption

- Alternatively, if a process request some resources
  - Check if they are available.
    - If yes, grant
    - If not, check if they are allocated to some other process that is waiting for additional resources
      - If yes, preempt desired resources from waiting process and allocate to the requesting process
      - If no, requesting process must wait
        - » Some of its resources may be preempted, but only if another process requests them
        - » Process resumes only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting
  - Often applied to resources whose state can be easily saved and restored – CPU registers, memory space

Deadlock Prevention: Disable Circular Wait

- Impose a total ordering of all resource types,.
- Each process requests resources in an increasing order of enumeration.
- Lets say tape drive is 1, disk drive is 5 and printer is 12
- A process can initially request any number of instances of a resource type Ri
- After that process can request instances of resource type Rj only if f(j)>f (i)
- Alternatively, before requesting Rj, release all Ri such that f(i) >= f(j)
- When several instances of same type are needed, a single request for all of them must be issued
- Re-ordering of resources requires re-programming
- Ordering should be as per usage pattern of resources.

Deadlock Avoidance

- Deadlock prevention leads to inefficient use of resources & execution of processes.
- Requires that OS be given in advance additional information concerning which resources a process will request and use during its lifetime
- Based on this info, OS decides whether to grant the request or delay it
- System must consider resources currently available, resources currently allocated, future requests and releases

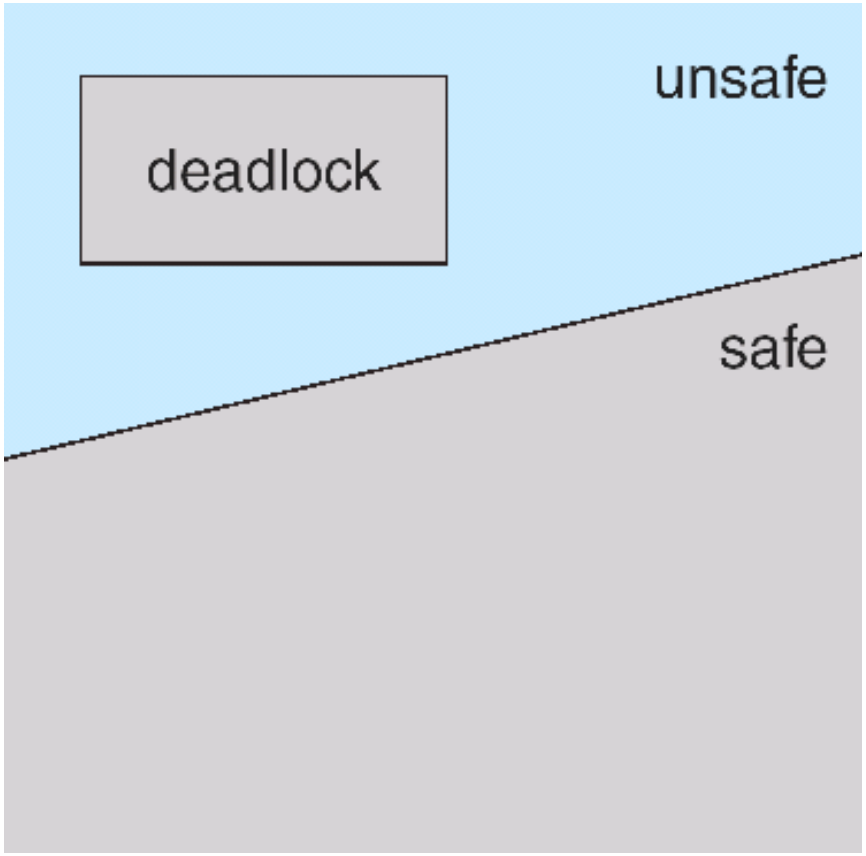
Deadlock Avoidance

- With this knowledge of complete sequence of requests and releases for each process, system can decide for each request whether or not the process should wait in order to avoid a future deadlock
- Variations in algorithms differ in amount and type of information required
- Simplest: each process declares the max number of resources of each type that it may need
- Dynamically examines the resource allocation state to ensure that a circular wait condition can never exist
- State: number of available and allocated resources, maximum demands of 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 of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if 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 $j<i$.
  - 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.

Safe, Unsafe , Deadlock State



- Safe state is not a deadlocked state
- Deadlocked state is an unsafe state
- Not all unsafe states are deadlocks
- Unsafe state may lead to a deadlock
- Better to always be in safe state

Example

- 12 magnetic tape drives, 3 processes
- Snapshot at t0

Process	Maximum needs	Currently holding
P0	10	5
P1	4	2
P2	9	2

- 3 more tape drives available
- System is in safe state with sequence P1, P0, P2

From safe to unsafe state

- Suppose at t1, P2 requests and is allocated 1 more tape drive

Process	Maximum needs	Currently holding
P0	10	5
P1	4	2
P2	9	3

- System is no longer in safe state
  - Only P1 can be allocated now. When done, both P0 and P2 will have to wait forever!
- If only we had made P2 wait until either of the other processes had finished, we could have avoided the deadlock

bottom line of avoidance algorithms

- Grant request only if the system remains in the safe state
  - Even if the resources it is asking for is currently available

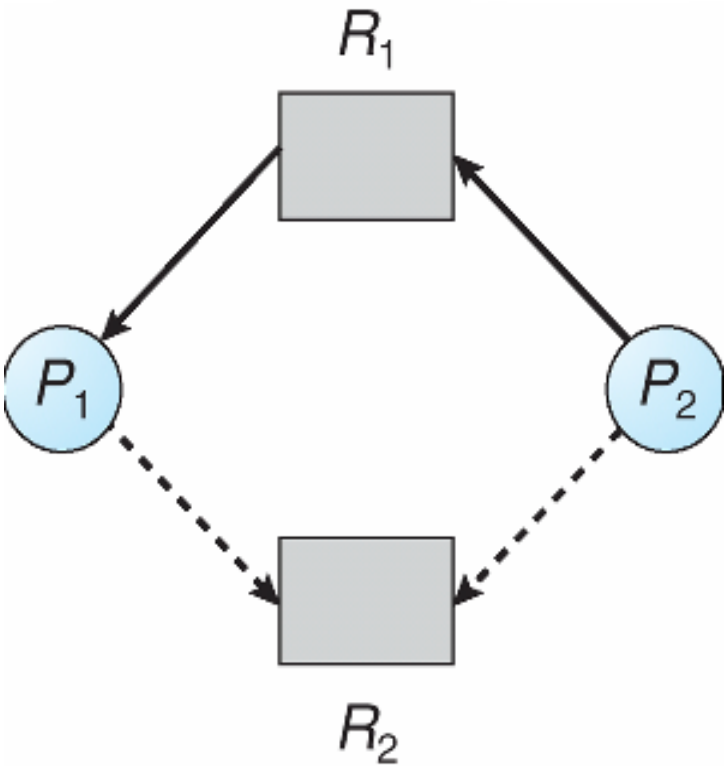
Avoidance algorithms

- When single instance of all resource types use RAG algorithm
  - Uses a variant of the RAG we saw earlier
- When multiple instances of resource types,
  - Use Banker's algorithm

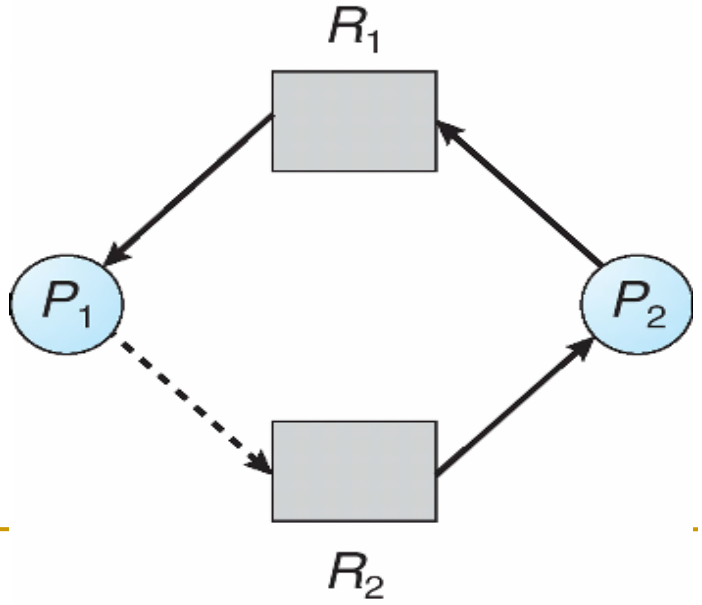
Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j at some time in future
- Similar to request edge in direction but is represented by a dashed line
- when a process requests a resource, Claim edge converts to request edge
- 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 reconverted to a claim edge
- Resources must be claimed *a prior* in the system

Resource-Allocation Graph



- Snapshot at time t
- Suppose P2 requests R2
- Although R2 is currently free it cannot be allocated to P2
  - Cycle!



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.
- Time complexity: For detecting a cycle in a graph it requires an order of $\mathbf{n^2}$ operations, Where n in number of processes in the system.

Banker's Algorithm

- Multiple instances of each resource type.
- Each process must a prior claim maximum use.
- Every process declares it maximum need/requirement.
- Maximum requirement should not exceed the total number of resources in the system.
- When a process requests a resource, system determines whether allocation will keep the system in safe state or not.
- If it is, resources get allocated. Otherwise need to wait until resources get available.
- 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]

Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

Following relationship holds

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j — All resources are either available or allocated.

2. $C_{ij} \leq R_j$, for all i,j — No process can claim more than the total amount of resources in the system.

3. $A_{ij} \leq C_{ij}$, for all i,j — No process is allocated more resources of any type than the process originally claimed to need.

Safety Algorithm

Requires m * n^2 operation to decide whether a state is

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

   Work = *Available*

   Finish [i] = *false* for *i* - 1,3, …, *n*.

2. Find process *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; otherwise process whose index is false *may potentially be in deadlock* in future

Example

- 5 processes P0 through P4
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot of system at t_0

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3			
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Example

- 5 processes P0 through P4
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot of system at t_0 : is it safe?

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
P_1	2	0	0	3	2	2	1	2	2
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1
P_4	0	0	2	4	3	3	4	3	1

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F				
P_1	2	0	0	1	2	2	F				
P_2	3	0	2	6	0	0	F				
P_3	2	1	1	0	1	1	F				
P_4	0	0	2	4	3	1	F				

Work		
3	3	2

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F			
P_1	**2**	**0**	**0**	**1**	**2**	**2**	**F**	**T**			
P_2	3	0	2	6	0	0	F	F			
P_3	2	1	1	0	1	1	F	F			
P_4	0	0	2	4	3	1	F	F			

Work		
3	3	2
5	**3**	**2**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F		
P_1	2	0	0	1	2	2	F	T	T		
P_2	3	0	2	6	0	0	F	F	F		
P_3	**2**	**1**	**1**	**0**	**1**	**1**	**F**	**F**	**T**		
P_4	0	0	2	4	3	1	F	F	F		

Work		
3	3	2
5	3	2
7	**4**	**3**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**7**	**4**	**3**	**F**	**F**	**F**	**T**	
P_1	2	0	0	1	2	2	F	T	T	T	
P_2	3	0	2	6	0	0	F	F	F	F	
P_3	2	1	1	0	1	1	F	F	T	T	
P_4	0	0	2	4	3	1	F	F	F	F	

Work		
3	3	2
5	3	2
7	4	3
7	**5**	**3**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F	T	T
P_1	2	0	0	1	2	2	F	T	T	T	T
P_2	**3**	**0**	**2**	**6**	**0**	**0**	**F**	**F**	**F**	**F**	**T**
P_3	2	1	1	0	1	1	F	F	T	T	T
P_4	0	0	2	4	3	1	F	F	F	F	F

Work		
3	3	2
5	3	2
7	4	3
7	5	3
10	**5**	**5**

Safe State?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C						
P_0	0	1	0	7	4	3	F	F	F	T	T	T
P_1	2	0	0	1	2	2	F	T	T	T	T	T
P_2	3	0	2	6	0	0	F	F	F	F	T	T
P_3	2	1	1	0	1	1	F	F	T	T	T	T
P_4	**0**	**0**	**2**	**4**	**3**	**1**	**F**	**F**	**F**	**F**	**F**	**T**

Work		
3	3	2
5	3	2
7	4	3
7	5	3
10	5	5
10	**5**	**7**

Safe Sequence: <P1, P3, P0, P2, P4>

Resource-Request Algorithm

$Request_i \rightarrow$ request vector (P_i); e.g. $Request_i[j] = k$

1. If $Request_i \leq Need_i$ go to step 2; Else *raise error condition* $\rightarrow$ process exceeds its maximum claim
2. If $Request_i \leq Available$, go to step 3; Else P_i *must wait*, since resources are not available
3. Tentatively allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

Check the safety of state -

- *If safe* $\Rightarrow$ *the resources are allocated to* P_i
- *If unsafe* $\Rightarrow$ P_i *must wait, and the tentative resource allocation is cancelled*

Example: P_1 Request (1,0,2)

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

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

$Request_1 = (1, 0, 2)$

- Request <need that is, (1,0,2) < (1,2,2) □->true
- $Request_1$ <= Available
  - (1, 0, 2) <= (3, 3, 2)
- Pretend to grant
  - $Allocation_1$, $Need_1$, Available will change (2, 3, 0)

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
P_1	**3**	**0**	**2**	**3**	**2**	**2**	**0**	**2**	**0**
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1
P_4	0	0	2	4	3	3	4	3	1

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F				
P_1	3	0	2	0	2	0	F				
P_2	3	0	2	6	0	0	F				
P_3	2	1	1	0	1	1	F				
P_4	0	0	2	4	3	1	F				

Work		
2	3	0

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F			
P_1	**3**	**0**	**2**	**0**	**2**	**0**	**F**	**T**			
P_2	3	0	2	6	0	0	F	F			
P_3	2	1	1	0	1	1	F	F			
P_4	0	0	2	4	3	1	F	F			

Work		
2	3	0
5	**3**	**2**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F		
P_1	3	0	2	0	2	0	F	T	T		
P_2	3	0	2	6	0	0	F	F	F		
P_3	**2**	**1**	**1**	**0**	**1**	**1**	**F**	**F**	**T**		
P_4	0	0	2	4	3	1	F	F	F		

Work		
2	3	0
5	3	2
7	**4**	**3**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**7**	**4**	**3**	**F**	**F**	**F**	**T**	
P_1	3	0	2	0	2	0	F	T	T	T	
P_2	3	0	2	6	0	0	F	F	F	F	
P_3	2	1	1	0	1	1	F	F	T	T	
P_4	0	0	2	4	3	1	F	F	F	F	

Work		
2	3	0
5	3	2
7	4	3
7	**5**	**3**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	7	4	3	F	F	F	T	T
P_1	3	0	2	0	2	0	F	T	T	T	T
P_2	**3**	**0**	**2**	**6**	**0**	**0**	**F**	**F**	**F**	**F**	**T**
P_3	2	1	1	0	1	1	F	F	T	T	T
P_4	0	0	2	4	3	1	F	F	F	F	F

Work		
2	3	0
5	3	2
7	4	3
7	5	3
10	**5**	**5**

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C						
P_0	0	1	0	7	4	3	F	F	F	T	T	T
P_1	3	0	2	0	2	0	F	T	T	T	T	T
P_2	3	0	2	6	0	0	F	F	F	F	T	T
P_3	2	1	1	0	1	1	F	F	T	T	T	T
P_4	**0**	**0**	**2**	**4**	**3**	**1**	**F**	**F**	**F**	**F**	**F**	**T**

Work		
2	3	0
5	3	2
7	4	3
7	5	3
10	5	5
10	**5**	**7**

Safe Sequence: <P1, P3, P0, P2, P4>

$Request_4 = (3, 3, 0)$

- $Request_4$ <= Available?
  - (3, 3, 0) <= (2, 3, 0)?
  - No. Thus request from P4 cannot be granted

$Request_0 = (0, 2, 0)$

- $Request_0$ <= Available
  - (0, 2, 0) <= (2, 3, 0)
- Pretend to grant
  - $Allocation_0$, $Need_0$, Available will change (2, 1, 0)

Process	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	**0**	**3**	**0**	**7**	**5**	**3**	**7**	**2**	**3**
P_1	3	0	2	3	2	2	0	2	0
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1
P_4	0	0	2	4	3	3	4	3	1

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	3	0	7	2	3	F				
P_1	3	0	2	0	2	0	F				
P_2	3	0	2	6	0	0	F				
P_3	2	1	1	0	1	1	F				
P_4	0	0	2	4	3	1	F				

Work		
2	1	0

Is the state safe?

Process	Allocation			Need			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	3	0	7	2	3	F				
P_1	3	0	2	0	2	0	F				
P_2	3	0	2	6	0	0	F				
P_3	2	1	1	0	1	1	F				
P_4	0	0	2	4	3	1	F				

Work		
2	1	0

Example : Is the below system in safe state?

Process	Allocation				Max			
	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2
P_1	1	0	0	0	1	7	5	0
P_2	1	3	5	4	2	3	5	6
P_3	0	6	3	2	0	6	5	2
P_4	0	0	1	4	0	6	5	6

Available			
1	5	2	0

Deadlock Detection

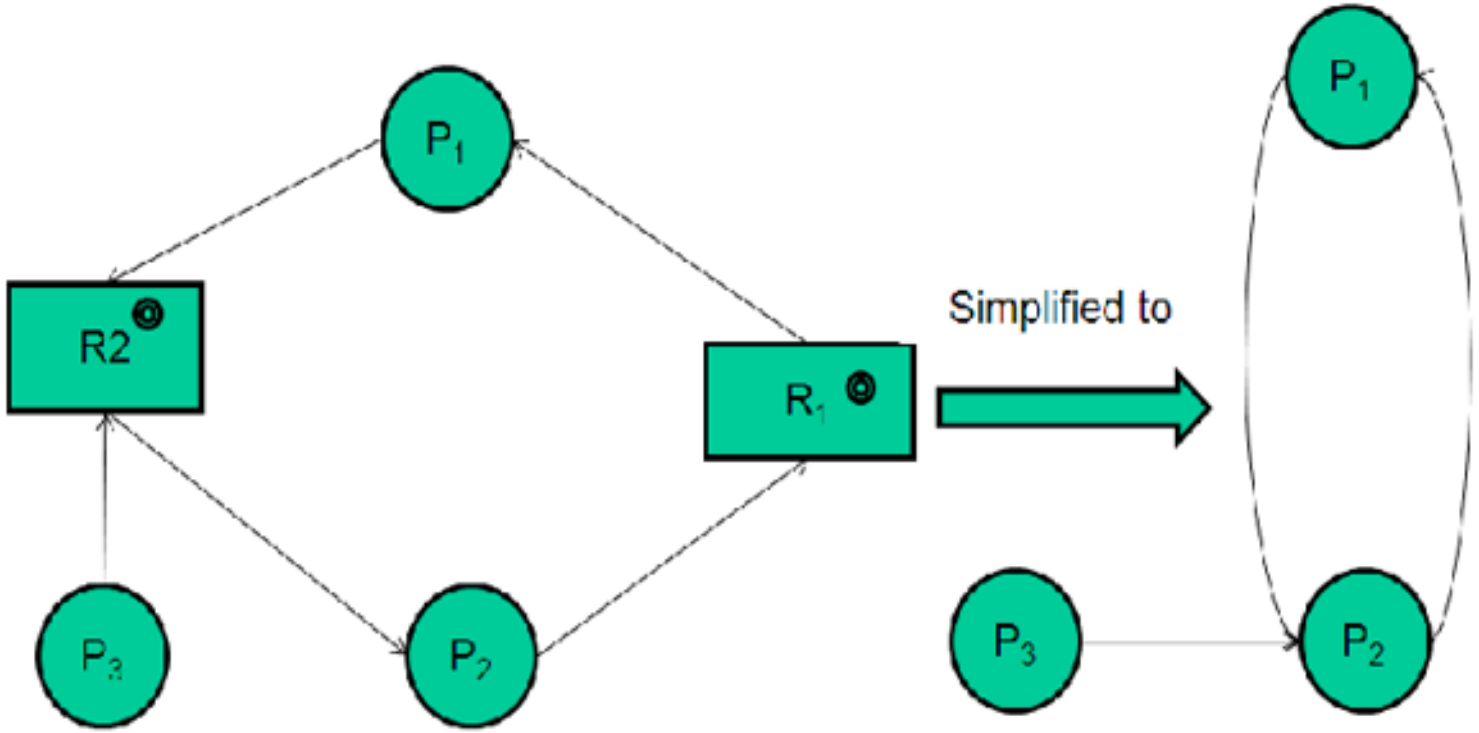
- When neither prevention nor avoidance is employed, a deadlock situation may arise
- System can provide an algorithm to detect an algorithm and an algorithm to recover from it if it has occurred
- Requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock

Case 1: Single instance of each resource type

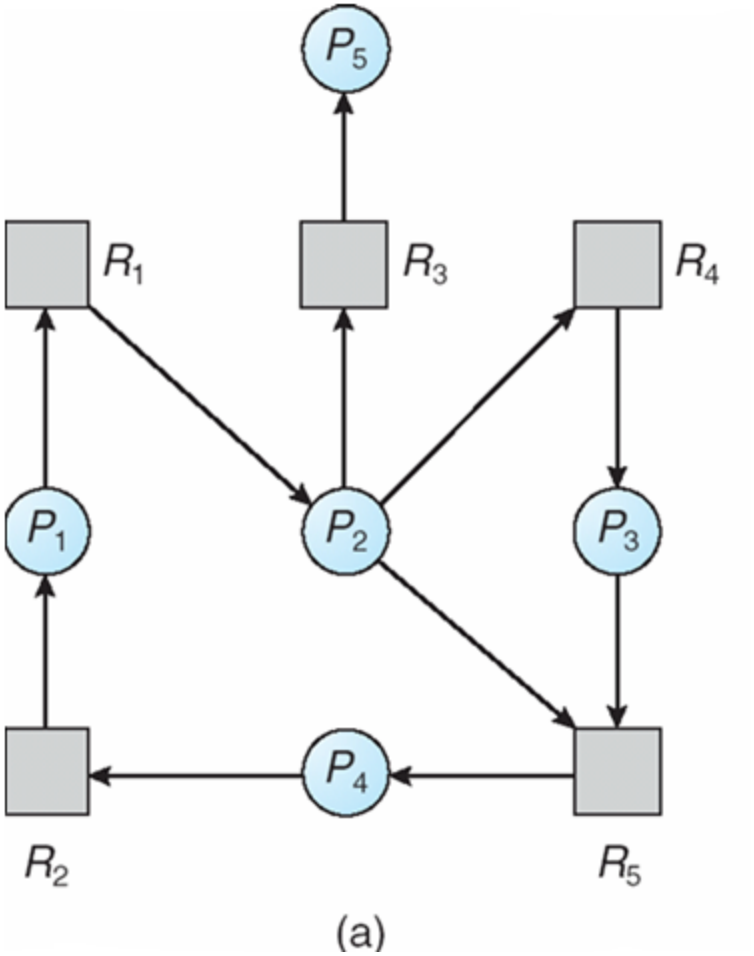
- Use a variant of RAG called WFG
  - by removing the resource nodes and collapsing the appropriate edges
- An edge from P_i to P_j in a wait-for graph implies that
- process P_i is waiting for process P_j to release a resource that P_i needs
  - if the corresponding RAG contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some R_q
- Periodically invoke an algorithm that searches for a cycle in the WFG
  - If there is a cycle, there exists a deadlock.
  - O(n^2) operations

Resource-Allocation Graph and Wait-for Graph

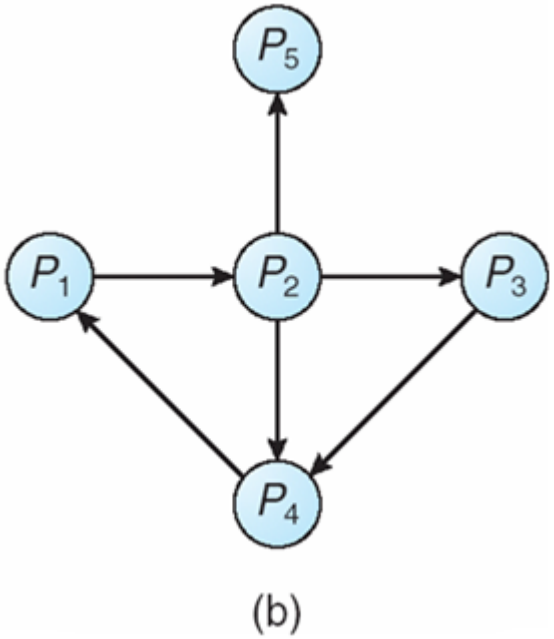
Def: Remove the resource edges from RAG and collapse the appropriate edges. WFG is constructed only when cycle is both necessary & sufficient condition for deadlock



Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Case 2: Several instances of each resource type

- WFG is no longer applicable for this case
  - Why?
- Detection algorithm
  - Uses several time varying data structures similar to those of Banker's
  - Available (m), Allocation (nXm), Request(nXm)
  - Same definition of <= notation
  - Same definition of $Allocation_i$ and $Request_i$
- Crux of the algorithm
  - Simply investigates every possible allocation sequence for the processes that remain to be completed

Case 2: Several instances of each resource type

Similar to the *Banker's algorithm* safety test with the following difference in semantics;

- Replacing $Need_i \rightarrow Request_i$; where $Request_i$ is the actual vector of resources, process *i* is currently waiting to acquire
- May be slightly optimized by initializing *Finish* [i] to *true* for every process *i* where $Allocation_i$ is zero
- Optimistic and *only care if there is a deadlock now*. If process will need more resources in future → deadlock, discovered in future
- Processes *in the end* remaining *with false entry* are the ones *involved in deadlock* at this time

Deadlock Detection Algorithm

Requires $m * n^2$ operation to detect a deadlock

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

   Work = *Available*

   If $Allocation_i \neq 0$ for $i = 1,2, \ldots, n$ then

   Finish [i] = **false**, else *Finish* [i] = **true**

2. Find process *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 $1 \leq i \leq n$, $\rightarrow$ deadlocked;

   If *Finish* [i] == **false** then process P_i is deadlocked

Contrast with Banker's Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize *Work* = *Available*. For $i = 0, 1, ..., n-1$, 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, $0 < i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

Example

- 5 processes P0 through P4
- 3 resource types: A (7 instances), B (2 instances), and C (6 instances)
- Snapshot of system at t_0 : is there a deadlock?

Process	Allocation			Request		
	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

Available		
0	0	0

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F				
P_1	2	0	0	2	0	2	F				
P_2	3	0	3	0	0	0	F				
P_3	2	1	1	1	0	0	F				
P_4	0	0	2	0	0	2	F				

Work		
0	0	0

Please note that all are initially ‘F’ here by coincidence

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**0**	**0**	**0**	**F**	**T**			
P_1	2	0	0	2	0	2	F	F			
P_2	3	0	3	0	0	0	F	F			
P_3	2	1	1	1	0	0	F	F			
P_4	0	0	2	0	0	2	F	F			

Work		
0	0	0
0	**1**	**0**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T	T		
P_1	2	0	0	2	0	2	F	F	F		
P_2	**3**	**0**	**3**	**0**	**0**	**0**	**F**	**F**	**T**		
P_3	2	1	1	1	0	0	F	F	F		
P_4	0	0	2	0	0	2	F	F	F		

Work		
0	0	0
0	1	0
3	**1**	**3**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T	T	T	
P_1	**2**	**0**	**0**	**2**	**0**	**2**	**F**	**F**	**F**	**T**	
P_2	3	0	3	0	0	0	F	F	T	T	
P_3	2	1	1	1	0	0	F	F	F	F	
P_4	0	0	2	0	0	2	F	F	F	F	

Work		
0	0	0
0	1	0
3	1	3
5	**1**	**3**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T	T	T	T
P_1	2	0	0	2	0	2	F	F	F	T	T
P_2	3	0	3	0	0	0	F	F	T	T	T
P_3	**2**	**1**	**1**	**1**	**0**	**0**	**F**	**F**	**F**	**F**	**T**
P_4	0	0	2	0	0	2	F	F	F	F	F

Work		
0	0	0
0	1	0
3	1	3
5	1	3
7	**2**	**4**

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish	
	A	B	C	A	B	C						
P_0	0	1	0	0	0	0	F	T	T	T	T	T
P_1	2	0	0	2	0	2	F	F	F	T	T	T
P_2	3	0	3	0	0	0	F	F	T	T	T	T
P_3	2	1	1	1	0	0	F	F	F	F	T	T
P_4	**0**	**0**	**2**	**0**	**0**	**2**	**F**	**F**	**F**	**F**	**F**	**T**

Work		
0	0	0
0	1	0
3	1	3
5	1	3
7	2	4
7	2	6

Thus, no deadlock: Sequence $<P_0, P_2, P_3, P_1, P_4>$

P2 now makes an additional request for 1 C

- Request matrix is modified. Is there deadlock now?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F				
P_1	2	0	0	2	0	2	F				
P_2	3	0	3	0	0	**1**	F				
P_3	2	1	1	1	0	0	F				
P_4	0	0	2	0	0	2	F				

Work		
0	0	0

Is there a deadlock?

Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	**0**	**1**	**0**	**0**	**0**	**0**	**F**	**T**			
P_1	2	0	0	2	0	2	F	F			
P_2	3	0	3	0	0	**1**	F	F			
P_3	2	1	1	1	0	0	F	F			
P_4	0	0	2	0	0	2	F	F			

Work		
0	0	0
0	**1**	**0**

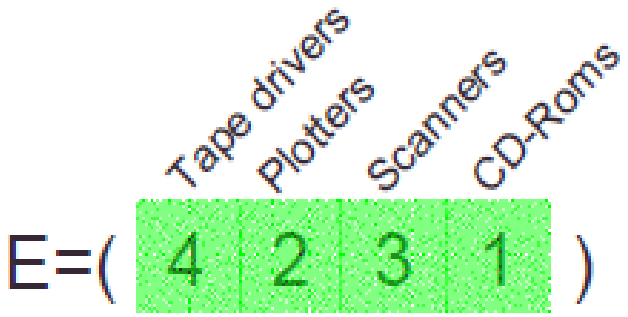
Is there a deadlock?

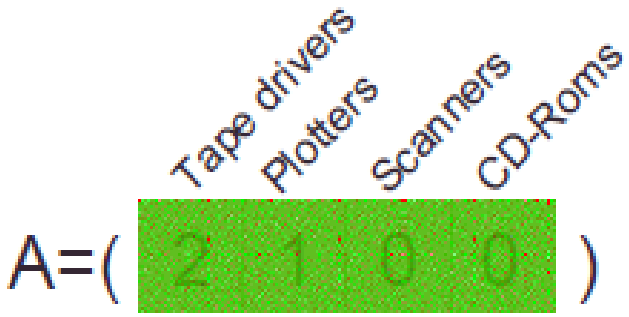
Process	Allocation			Request			Finish	Finish	Finish	Finish	Finish
	A	B	C	A	B	C					
P_0	0	1	0	0	0	0	F	T			
P_1	2	0	0	2	0	2	F	F			
P_2	3	0	3	0	0	**1**	F	F			
P_3	2	1	1	1	0	0	F	F			
P_4	0	0	2	0	0	2	F	F			

Work		
0	0	0
0	1	0

P1, P2, P3 and P4 are in a deadlock now!

Example: Multiple resources of each type





Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Example: Multiple resources of each type

Tape drivers Plotters Scanners CD-Roms

$E = (\ 4\ \ 2\ \ 3\ \ 1\ )$

Tape drivers Plotters Scanners CD-Roms

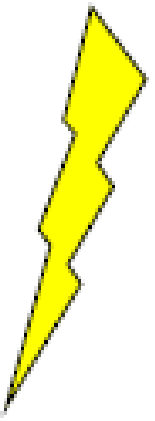
$A = (\ 2\ \ 0\ \ 0\ \ 0\ )$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$



Detection-Algorithm Usage

- When, and how often, to invoke algorithm depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by deadlock when it happens?
- If deadlock occurs frequently, then the detection algorithm should be invoked frequently.
- We could invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
- By this we can identify deadlock causing process & processes involved in deadlock also.
- But this incurs overhead in computation time.
- So can call algorithms after 1 hour / CPU utilization drops below 40%.
- 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.

Contents

- Uniprocessor Scheduling: Types of Scheduling: Preemptive, Non-preemptive, Long-term, Medium-term, Short-term scheduling
- Scheduling Algorithms: FCFS, SJF, RR, Priority
- Multiprocessor Scheduling: Granularity
- Design Issues, Process Scheduling
- Deadlock: Principles of deadlock, Deadlock Avoidance
- Deadlock Detection, Deadlock Prevention
- Deadlock Recovery

Deadlock Recovery

- Once detected, several alternatives are available for recovery inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually
- Abort one or more processes to break the circular wait
- Preempt some resources from one or more of the deadlocked processes

Process Termination / Abort

- Two alternatives
- In both, system reclaims all resources held by the terminated process
- **Abort all deadlocked processes**
  - Everything the processes had done so far has gone down the drain!
- **Abort one process at a time until the deadlock cycle is eliminated**
  - Whose turn is next?
    - Policy decision similar to scheduling decisions
  - Considerable overhead of detecting deadlock after each termination!
- Aborting comes with several issues
  - What if in the middle of updating a file or printing to a printer

Whom to abort next?

- Abort those processes whose termination will incur the minimum *cost*
- *Cost* depends upon
  - What is the priority of the process?
  - How long the process has computed and how much longer the process will compute before completing its designated task?
  - How many and what type of resources the process has used (e. g., preemptable, non-preemptable)?
  - How many more resources the process needs in order to complete?
  - How many processes will need to be terminated?
  - Whether the process is interactive or batch?

Resource Preemption

- We successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- Three issues need to be addressed:
  - Selecting a victim
  - Rollback
  - Starvation

Selecting a victim

- Which resources and of which processes next in order to minimize cost

Rollback

- What to do with the process from whom resource has been preempted? It cannot continue normal execution
- Rollback to some safe state and restart it from that state
  - What is safe state, also system will require to keep states of all processes
  - Thus some systems prefer total rollback

Starvation

- How can we guarantee that resources will not always be preempted from the same process
- If decision to pick is primarily based on cost factors, same unfortunate fellow may get picked up every time!
- We thus need an upper bound (small and finite) on how many times you can be chosen as a victim
  - Include the number of rollbacks in the cost factor

Summary of Deadlock prevention, Avoidance,Detection

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses