Deadlocks

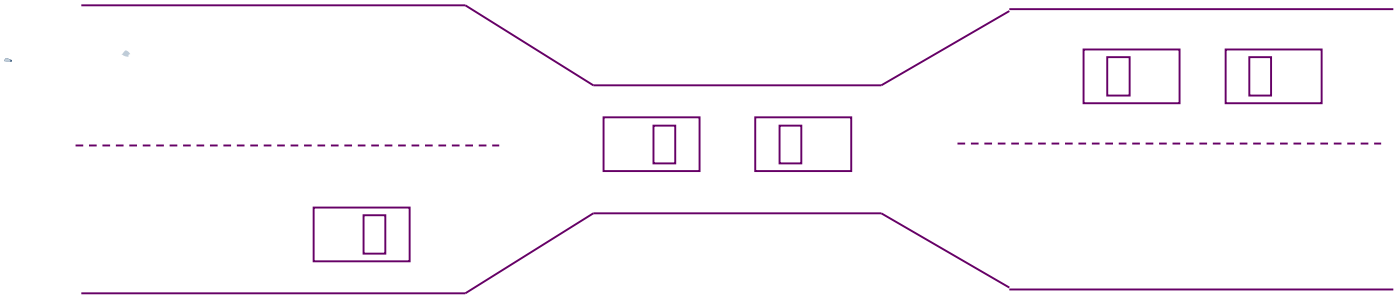
- Outline:
  - Deadlock problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery from Deadlock
  - Combined Approach to Deadlock Handling

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

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

- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by another process in the set.
- Events:
  - Resource acquisition and release.
- Resource types R_1, R_2, . . ., R_n
  - *Physical resources:CPU cycles, memory space, I/O devices*
  - *Logical resources: files, semaphores, and monitors*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - Request: if the request is not granted , then it must wait.
  - Use: The process can operate on the resource.
  - Release: The process releases the resources.
- Multi-threaded programs are good candidates for deadlock because multiple threads compete for shared resources.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use at least one resource. Resources are non sharable.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** Resources can not be preempted; that is 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, …Pn-1,Pn, P_0} 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.

ALL FOUR CONDITIONS MUST HOLD.

Resource-Allocation Graph

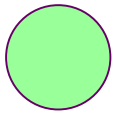
A tool to describe the deadlock.

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 $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

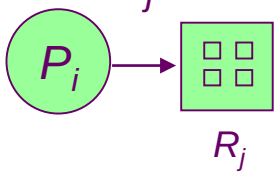
Resource-Allocation Graph (Cont.)

- Process

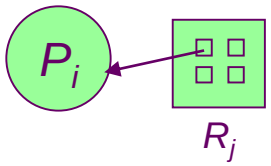


- Resource Type with 4 instances

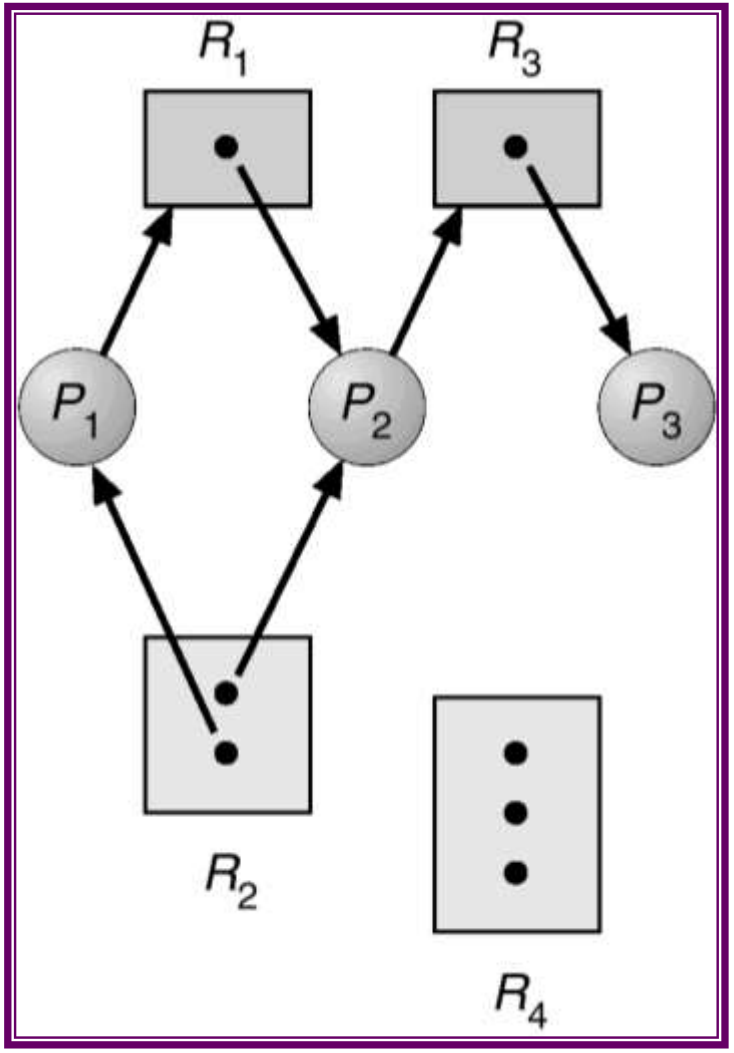
- P_i requests instance of R_j



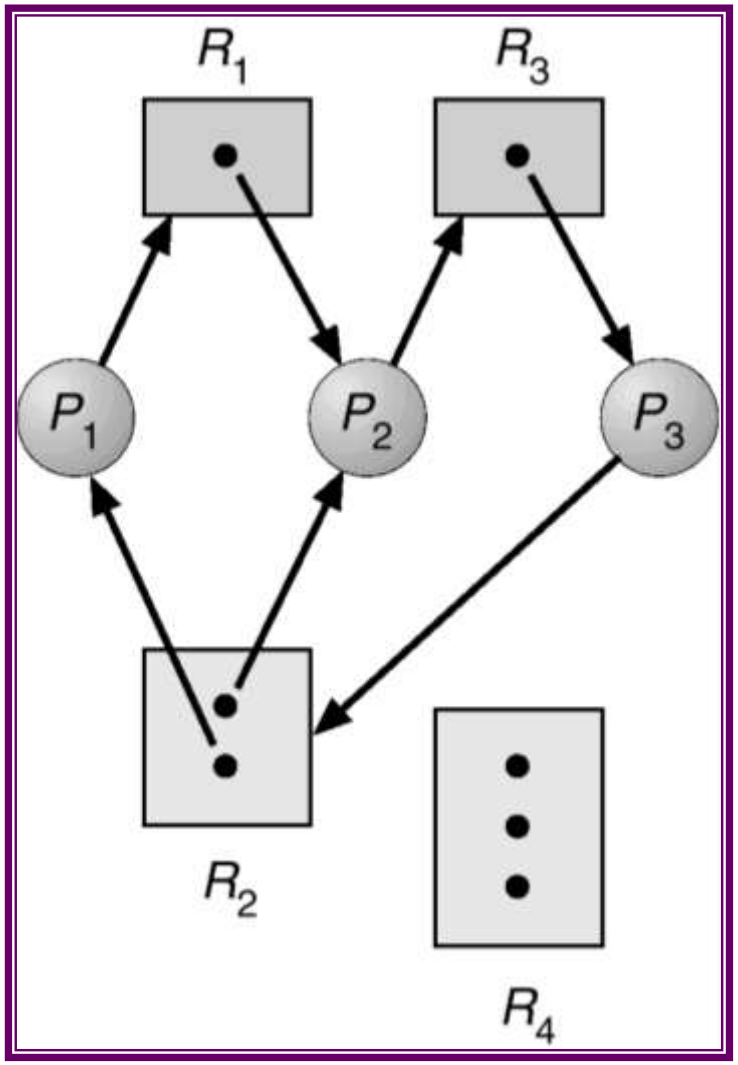
- P_i is holding an instance of R_j



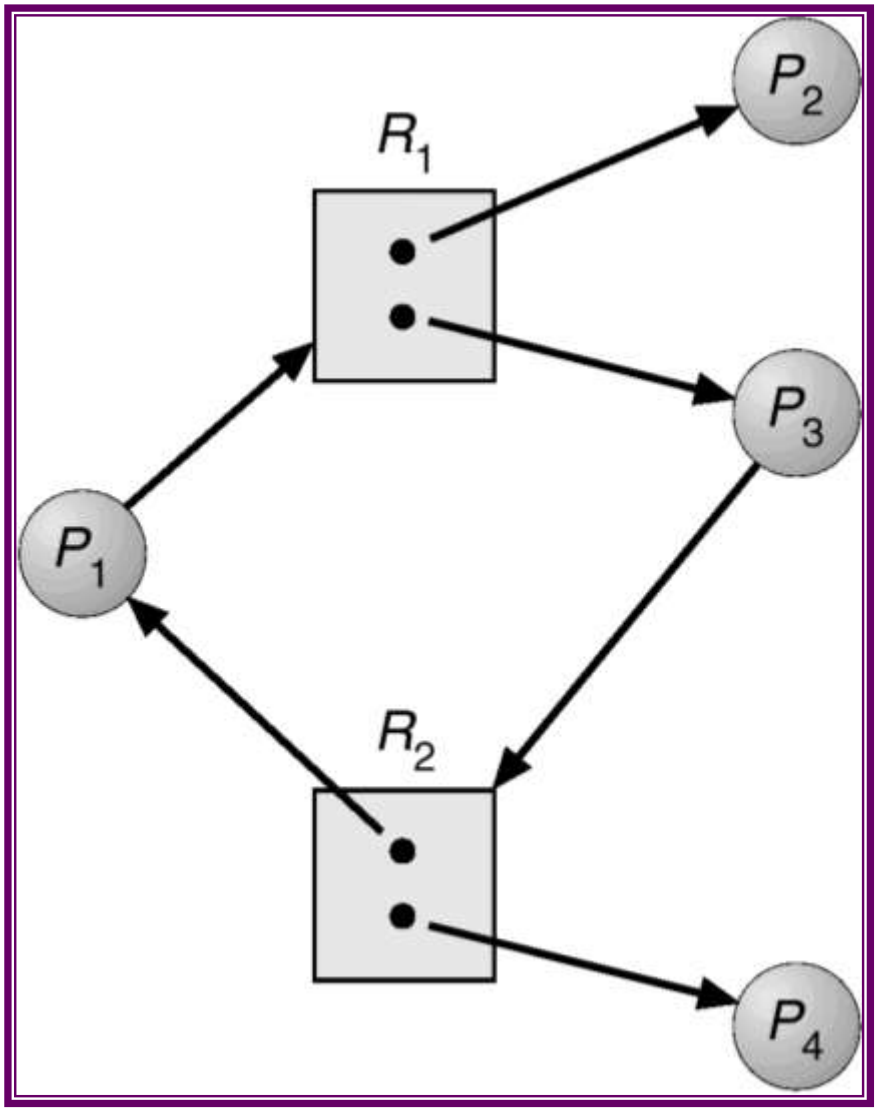
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation 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.

Methods for Handling Deadlocks

- Three ways
  - Ensure that the system will *never* enter a deadlock state.
  - Allow the system to enter a deadlock state and then recover.
  - Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Methods for Handling Deadlocks

- To ensure that deadlocks never occur, the system can use either deadlock prevention and deadlock-avoidance schemes.
  - Deadlock prevention: set of methods for ensuring that at least one of the necessary conditions can not hold.
  - Deadlock avoidance: Requires that operating system be given in advance additional information concerning which resources a process will request and use during its life time.
- Deadlock detection: examines the state of the system to determine whether a deadlock has occurred or not.
- Alternatively, assume that deadlock would not occur. Resort to manual recovery when performance degrades due to deadlock.

Deadlock Prevention

- **Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions can not hold.**

Deadlock Prevention

Restrain the ways request can be made.

- **Mutual Exclusion** – It is not possible to prevent deadlocks through ME, as some resources are intrinsically non-sharable.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Protocol 1: Requires a process to request and be allocated all its resources before it begins execution.
    - System calls requesting resources for a process precede all other system calls.
  - Protocol 2: Allow process to request resources only when the process has none.
    - A process can request some resources and use them. Before, it can request additional resources, it must release all the resources that it is currently allocated.

Deadlock Prevention

- **Hold and Wait** – Example
  - Consider a process that copies data from tape drive to a disk file and then prints the results to the printer.
  - Protocol1: If all the resources are requested at the beginning of a process, the process must request the tape drive, disk file, and printer.
    - It will the printer during entire execution, even though it needs at the end.
  - Protocol 2: It copies data from tape drive to disk file and releases them. It gain requests disk file and printer.
- Disadvantages:
  - Resource utilization is very low
    - Resource may be allocated but unused for a long time.
  - Starvation is possible
    - A process that needs several resources may have to wait indefinitely, at least one of the resources that it needs is always allocated to another process.
    - Inefficient
    - Delays process initiation
    - Future resource requirements must be known

Deadlock Prevention (Cont.)

- **No Preemption** –
  - Protocol: 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.
  - Possible for CPU registers and memory space whose state can be restored later, but not possible for printers and tape drives.

Deadlock Prevention (Cont.)

- **Circular Wait –**
  - F:R→N is a one-to-one function, where N is a set of natural numbers.
  - **Protocol 1:** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
    - A process initially can request any number of instances of type Ri. Then the process can request instances of a resource type Rj if and only if F(Rj) >F(Ri).
  - **Protocol 2**: Whenever a process requests an instance of resource type Rj, it has released any other resources Ri such that F(Ri) >=F(Rj).
  - Ordering

Deadlock prevention

- **Circular Wait**:
  - "**witness**" system call is implemented in BSD UNIX for lock order verifier
  - "**witness**" uses mutual-exclusion locks to protect critical regions and maintains the relationship of lock orders in the system
  - Suppose thread_one aquires locks in the order first_mutex and second_mutex. Then **witness** records that first_mutex should be obtained before second_mutex. If thread_two later aquires locks out of order, **witness** generates a warning message.

```
thread_one
{
        lock(first_mutex);
        lock(second_mutex);
.
.
}
```

```
thread_two:
{
        lock(second_mutex);
        lock (first_mutex);
.
.
}
```

Deadlock Avoidance

- Deadlock prevention algorithms
  - Low device utilization, and reduced system throughput.
- Alternative method is get additional information about the processes
  - Resources currently available, resources currently allocated to a process, and future requests and releases of each process.
- Various algorithms differ about amount and type of information required.

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.

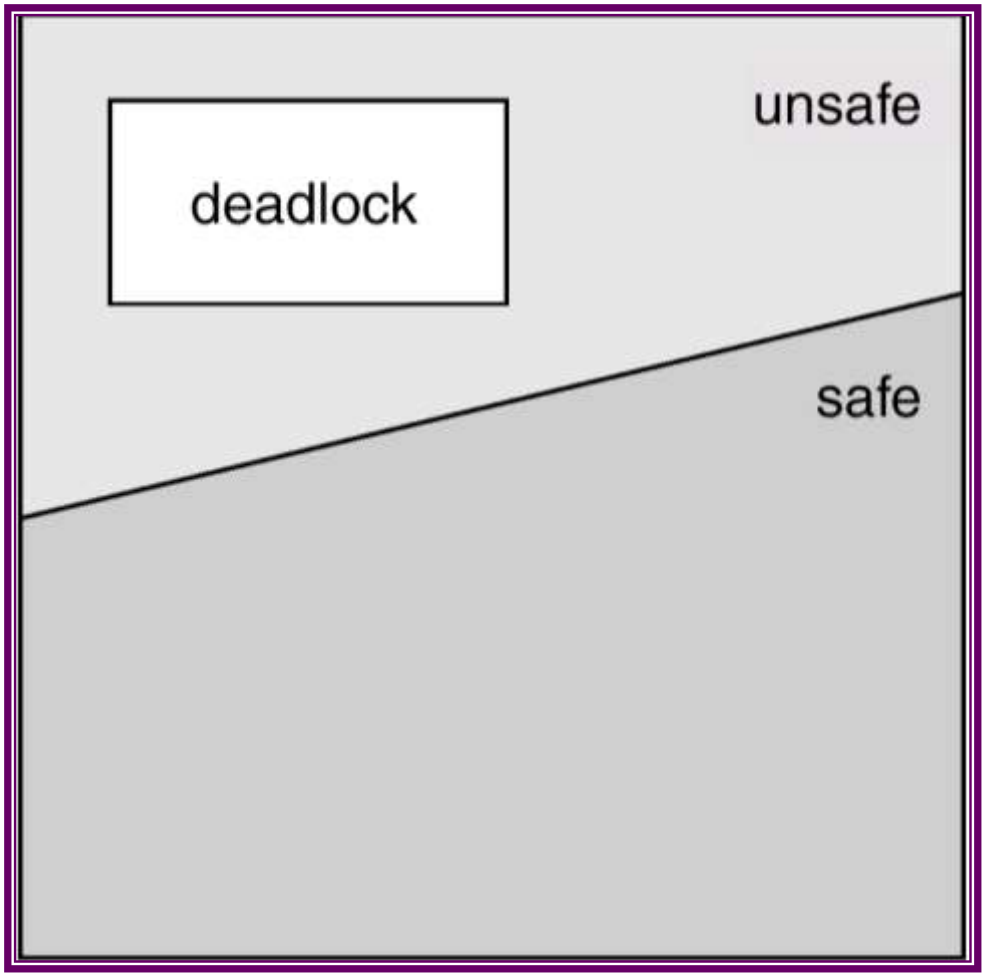
Safe State

- 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 safe sequence of all processes.
- Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that *Pi* 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.

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



Example

- Consider a system with 12 magnetic tape drives and three processes P0,P1,P2.
- P0 requires 10 tape drives, P1 may need 4, and P2 may need up to 9 tape drives.
- Suppose at time t0, P0 is holding 5 tape drives, P1 holding 2, and P2 is holding 2 tape drives.
- There are 3 free tape drives

	Maximum needs	allocated	current needs
P0	10	5	5
P1	4	2	2
P2	9	2	7

- At t0, system is in safe condition.
- <P1,P0,P2> satisfies satisfy safety condition.
- Suppose, at t1, P2 requests and is allocated 1 more tape drive.
  - The system is no longer in safe state.
- The mistake was in granting the request of P2 for one more tape drive.
- The main idea of avoidance algorithm is to ensure the system is always in a safe state.

Two algorithms for Deadlock Avoidance

- Resource-Allocation Graph algoritm
- Bankers algoritm

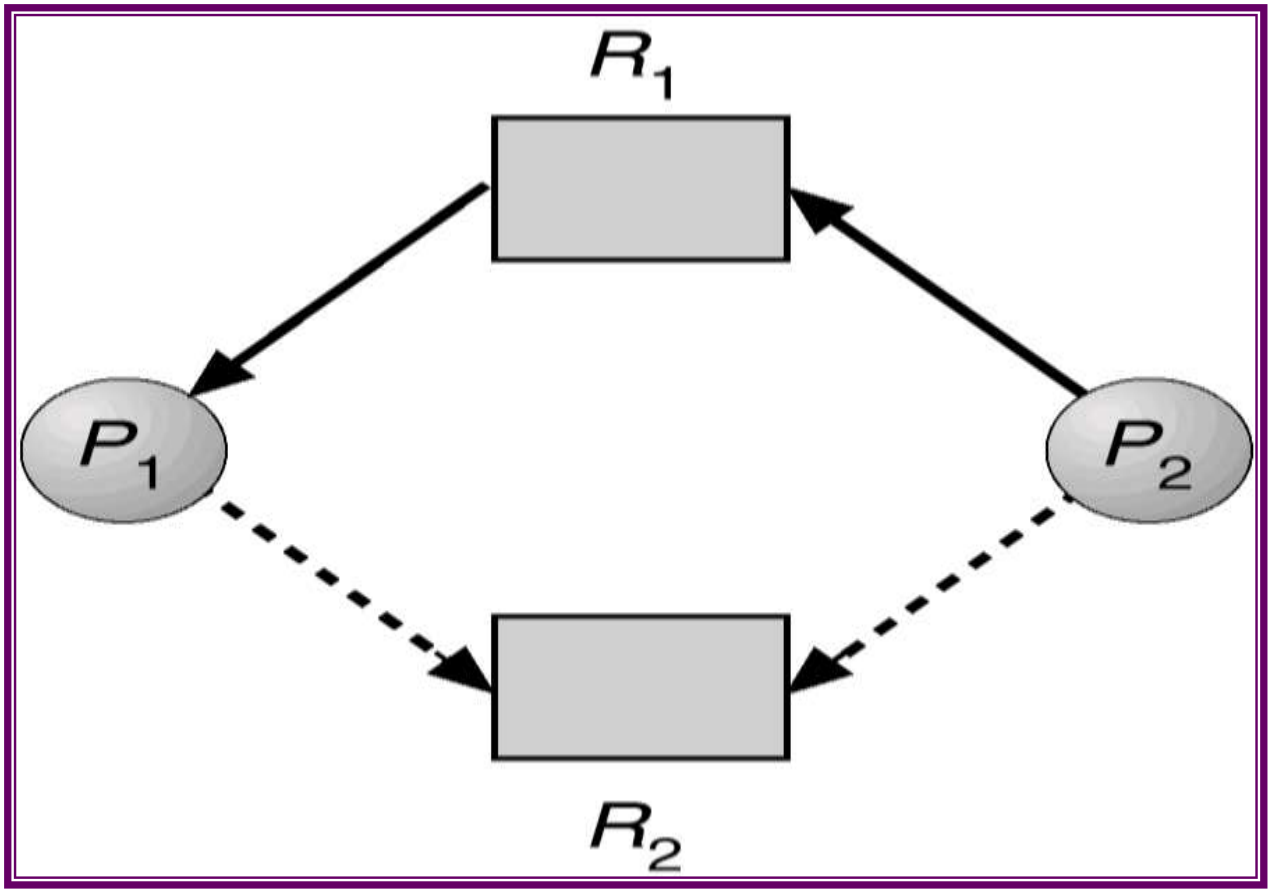
Resource-Allocation Graph Algorithm

- If we have a resource-allocation (RA) system with one instance of each resource type, we can use this approach.
- *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.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.
- Before the process starts executing, all its claim edges must appear in the RA graph.
- Suppose a process Pi requests a resource Rj. The request can be granted if converting the request edge Pi → Rj to an assignment edge Rj →Pi does not result in the formation of a cycle in the RA graph.

Resource-Allocation Graph Algorithm

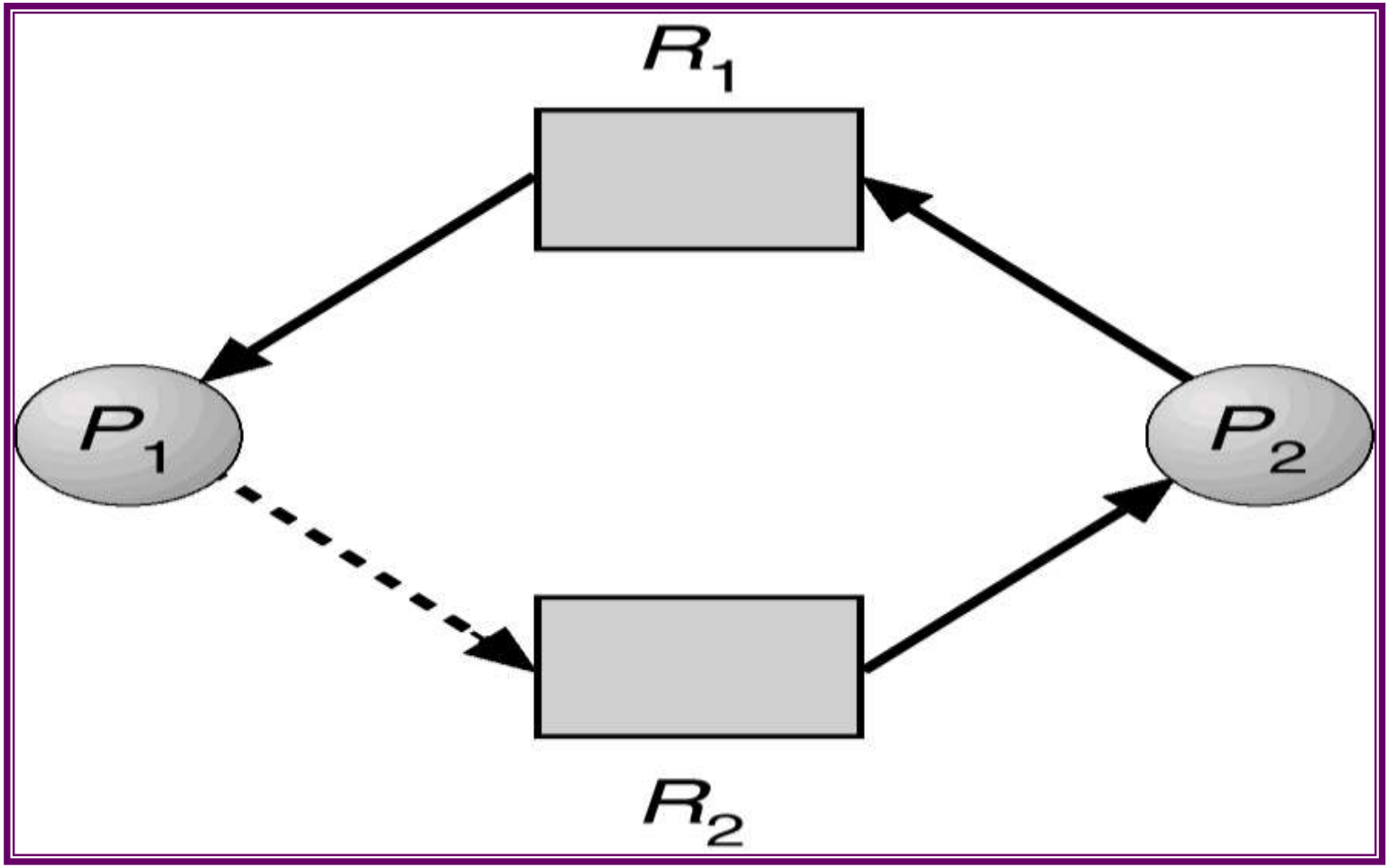
- Cycle detection algorithm is used to detect a cycle.
- If there are n processes detecting a cycle requires an order of n*n operations.

Resource-Allocation Graph For Deadlock Avoidance



Suppose P2 requests R2

Unsafe State In Resource-Allocation Graph



Cycle in RA graph! So system is in an unsafe state. So we can not allocate R2 to P2.

Banker's Algorithm

- Bank never allocates its available cash such that it no longer satisfy the needs of all customers.
- 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.

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

- Notation:
  - Let X and Y be vectors of length n.
  - We say X <= Y if and only if X[i] <= Y[i] for all i=1,2,…,n.
  - For example, if X=(1,7,3,2) and Y=(0,3,2,1) then Y<= X.
  - Y < X if Y<=X and Y # X.

Resource-Request Algorithm for Process P_i

Request = 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*

Safety Algorithm

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 and *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.

- Safety algorithm may require an order of $m*n^2$ operations.

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:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
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 (Cont.)

- The content of the matrix. Need is defined to be Max – Allocation.

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

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

Example P_1 Request (1,0,2) (Cont.)

- 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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

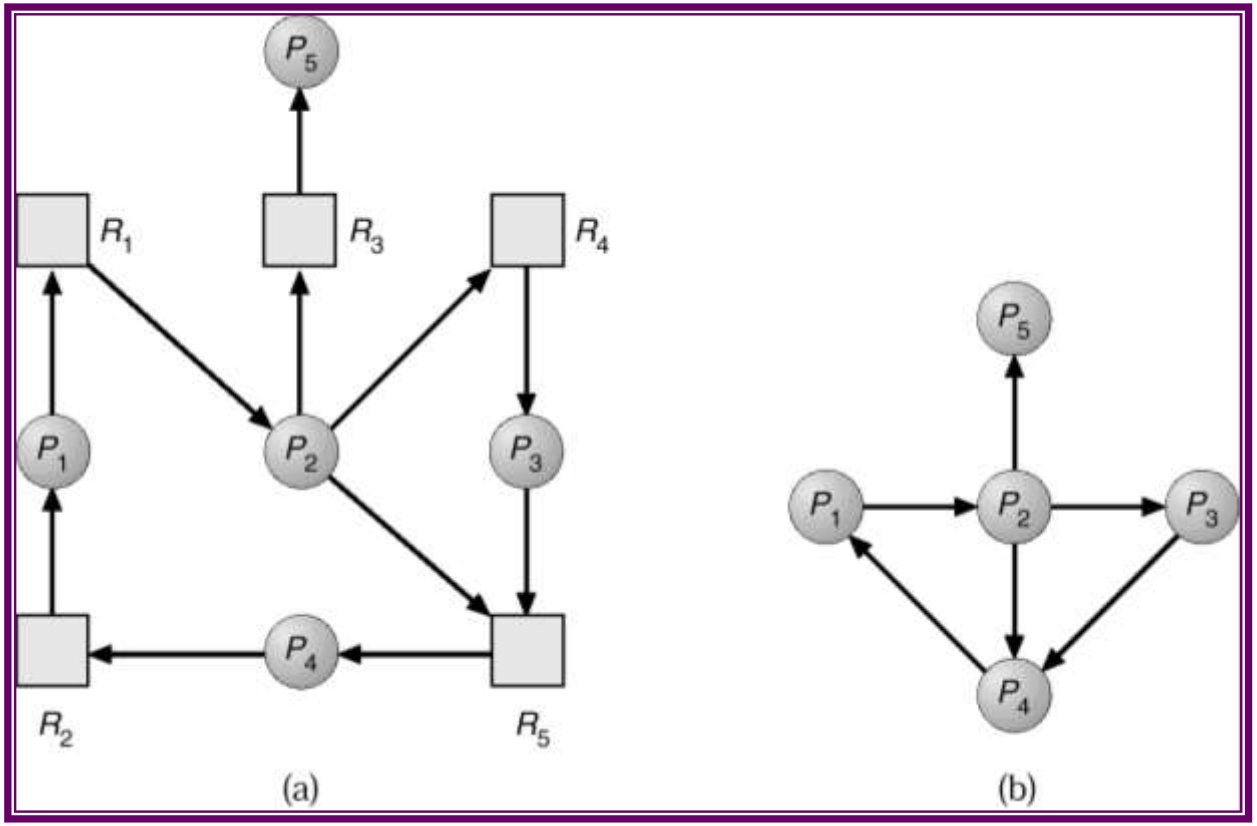
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Resource allocation graph algorithm
  - Detection algorithm (similar to Banker's algoritm)
- Recovery scheme

Single Instance of Each Resource Graph based approach

- 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.
- 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

- ***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 *i* = 1,2, …, *n*, 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.

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	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	

- Sequence <P_0, P_2, P_3, P_1, P_4> 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 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- 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

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, 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.
- Invoking for every request is expensive
  - Invoke once per hour; when CPU utilization drops below 40 %.

Recovery from Deadlock: Process Termination

- Several alternatives exist.
  - Inform the operator
  - Automatic: system will recover from the deadlock automatically
  - Two options exist for automatically breaking a deadlock
    - Process termination
    - Resource preemption

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
  - More expensive
- Abort one process at a time until the deadlock cycle is eliminated.
  - Overhead: after the abort of each process detection algorithm have to be invoked.
- Aborting a process is not easy.
  - Printing or updating a file
- In which order should we choose to abort? (should incur minimum cost)
  - 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?

Recovery from Deadlock: Resource Preemption

- We preempt some resources from one process and give it to other processes. Until the deadlock cycle is broken.
- For preemption three issues are to be addressed
  - Selecting a victim – minimize cost.
    - Number of resources the deadlocked process is holding, time consumed by it for execution.
  - Rollback – return to some safe state, restart process for that state.
    - Rollback as far as necessary to break the deadlock.
  - Starvation – same process may always be picked as victim,
    - Solution: include number of rollbacks in cost factor.

Deadlock Handling: summary

- Three basic approaches
  - **Prevention**: Use some protocol to prevent deadlocks, ensuring the system will never enter a deadlock state.
    - Low resource utilization
  - **Avoidance:** Use a priori information; do not allow the system to enter unsafe state.
    - Needs a priori information about future requests.
  - **Detection**: Allow the system to enter deadlock state: then recover.
    - Select the victim based on the cost factors.