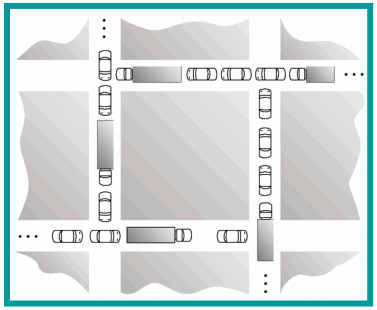
Module 6: Deadlocks

Reading: Chapter 7

Objective:

- **To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks**
- **To present a number of different methods for preventing or avoiding deadlocks in a computer system.**



Resource owned by other waiting process

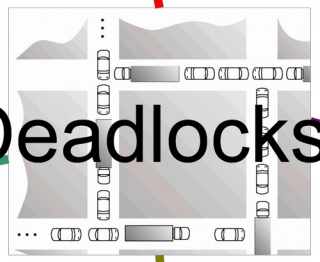
System Model

Process waits on resource

Deadlock Problem

Characterization

Deadlocks



Handling Deadlocks

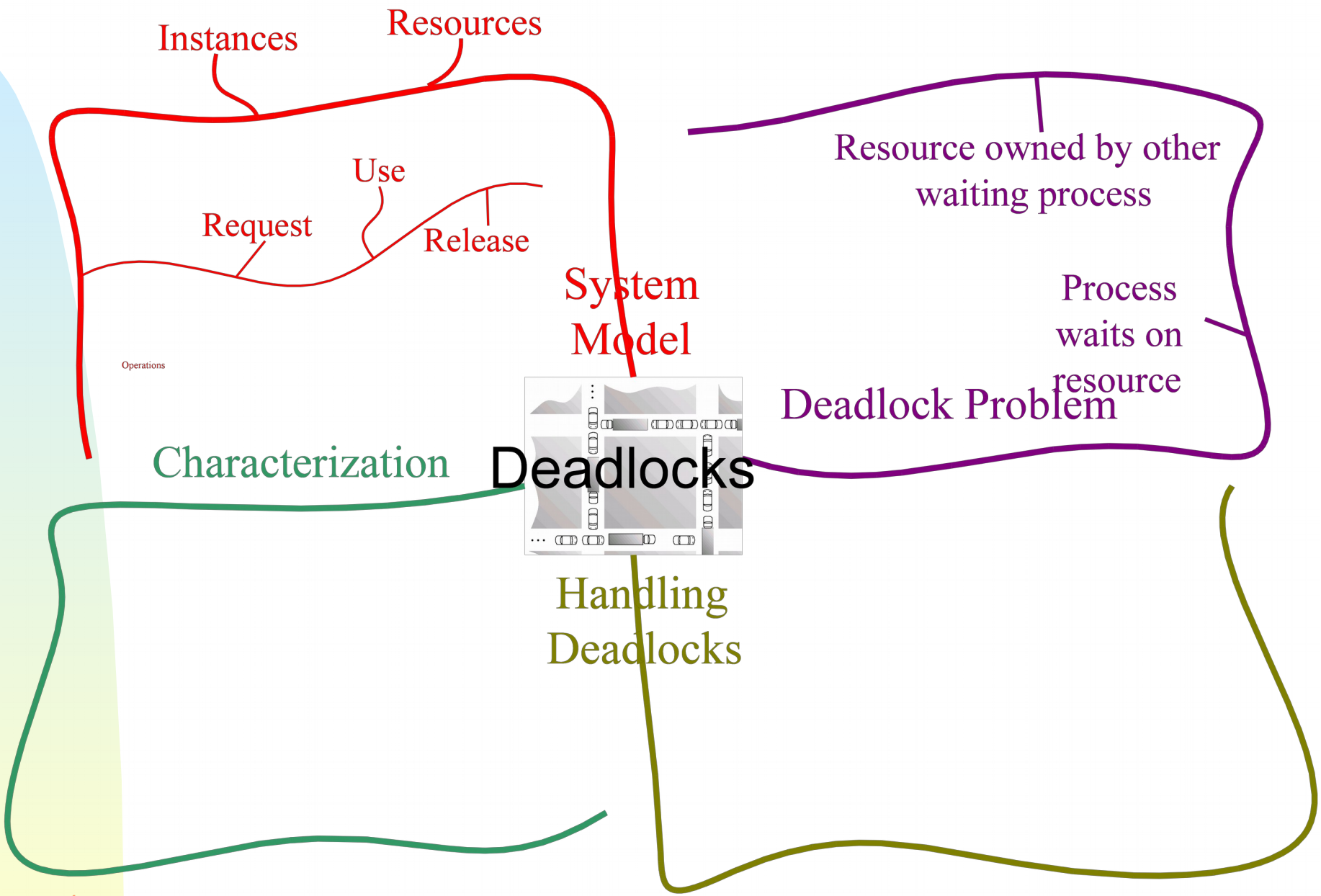
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Recall the example with semaphores
  - semaphores *A* and *B*, initialized to 1

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

- Example – law passed by Kansas Legislature

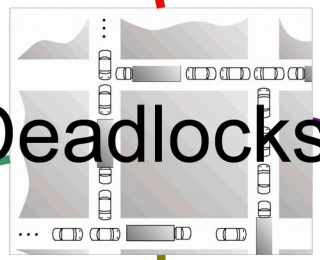
  "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."


Instances
Resources
Use
Request
Release
System Model
Operations
Resource owned by other waiting process
Process waits on resource
Deadlock Problem
Characterization
Deadlocks
Handling Deadlocks

System Model

- 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.
  - 2 printers, three hard drives, etc.
- Each process utilizes a resource as follows (using system calls):
  - request
  - use
  - release
- Deadlock example
  - Three processes each hold a CD drive.
  - Each process requires a 2nd drive.
- Multithreaded programs are good candidates for deadlock
  - Threads share many resources.

Deadlocks



System Model

- Resources
  - Instances
- Operations
  - Request
  - Use
  - Release

Deadlock Problem

- Resource owned by other waiting process
- Process waits on resource

Characterization

- Necessary Conditions
  - Mutual Exclusion
  - Hold and Wait
  - No Pre-emption
  - Circular Wait
- Resource-Allocation Graph
  - Vertex
    - Resources
    - Processes
  - Edge
    - Request
    - Assignment

Handling Deadlocks

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, …, 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.**

- When the first 3 conditions exist, the circular wait is an indication of a deadlock.
- The first 3 conditions do not imply necessarily a deadlock, since the circular wait may not occur.

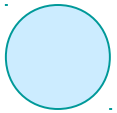
Resource-Allocation Graph

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

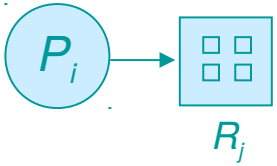
Resource-Allocation Graph (Cont.)

- Process

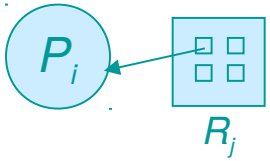


- Resource Type with 4 instances

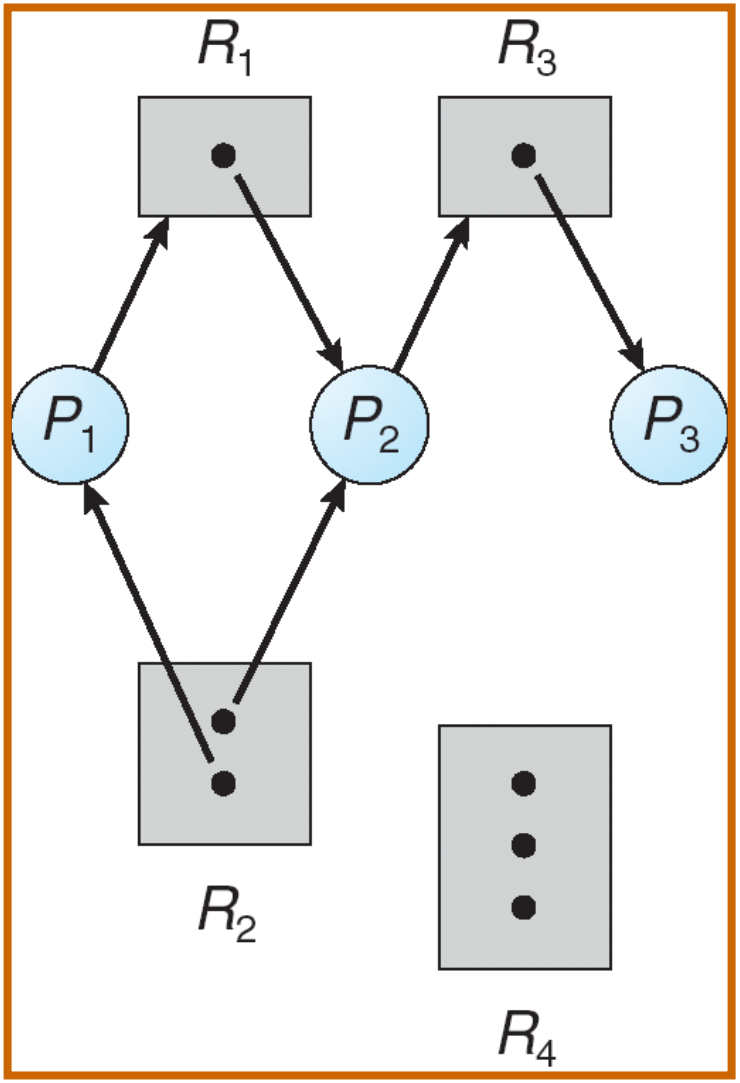
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

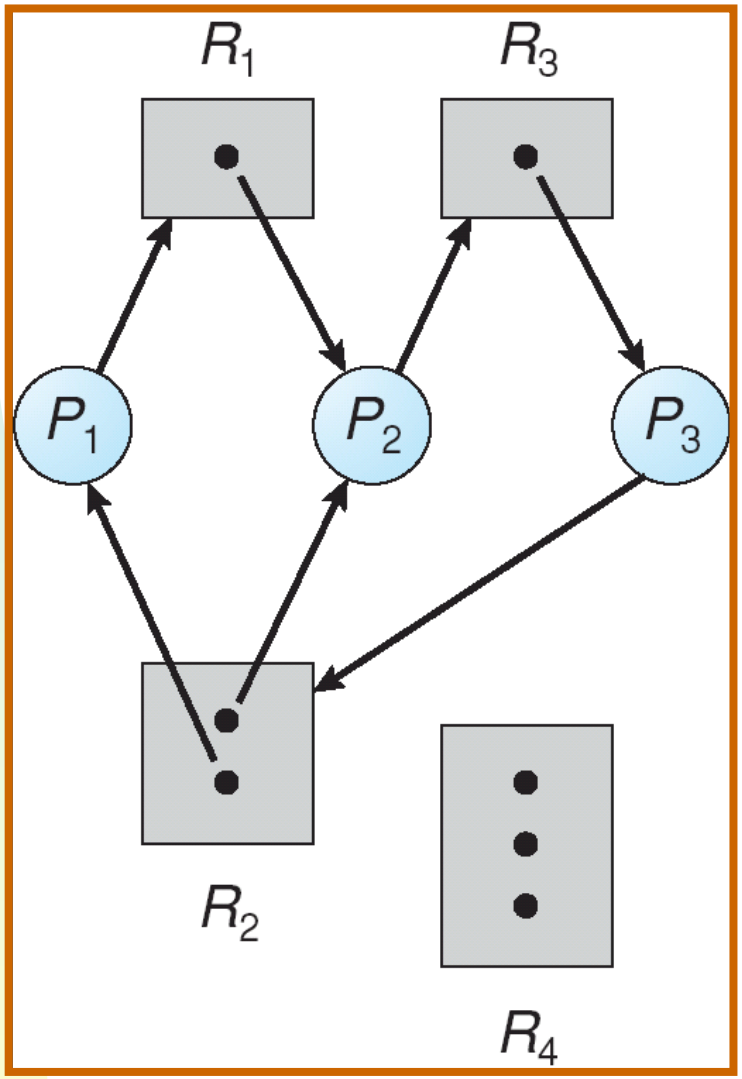


Is there deadlock?

Using the graphs

- Assume that the first three conditions for deadlock exist.
  - Mutual Excl., hold and wait, no pre-emption
- To show a deadlock, must show that no circular wait exists.
  - A process can terminate without any waiting on another, and then the others.
- $<P_3, P_2, P_1>$ is a termination sequence: all processes can terminate in this order.

Resource Allocation Graph With A Deadlock



Circular waits:

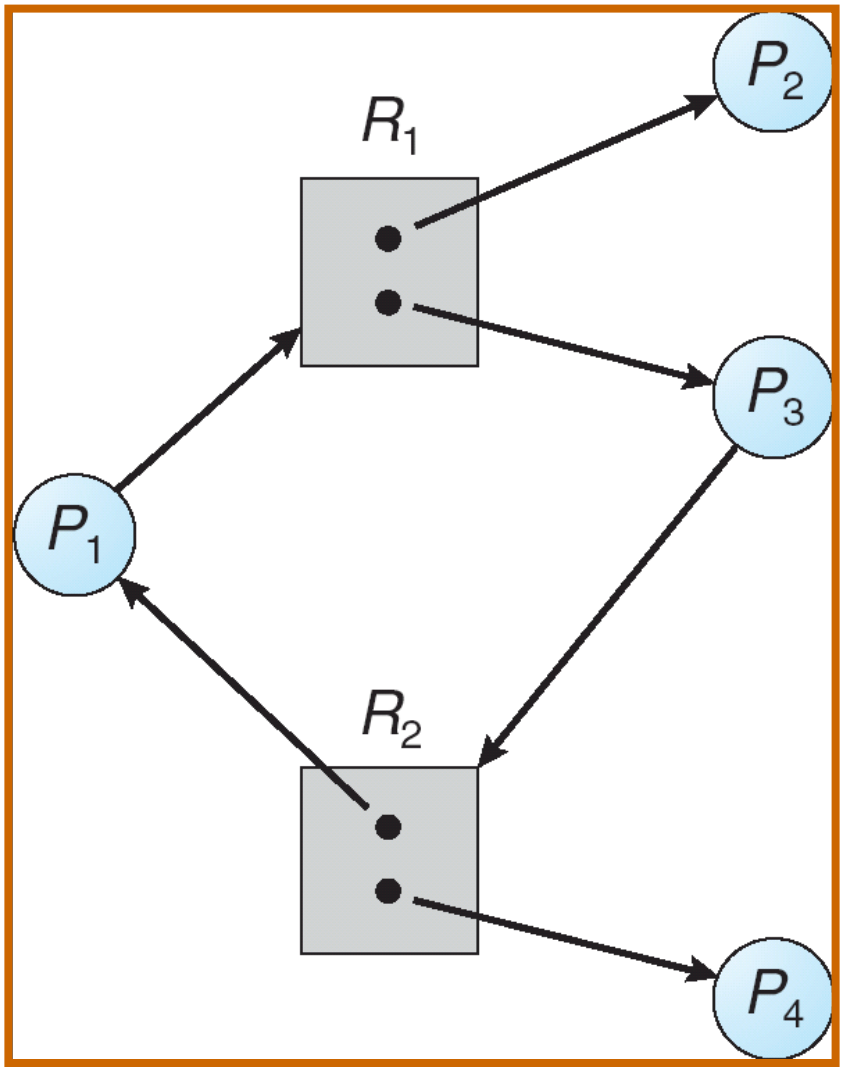
P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

No proc. can terminate

No possible way out.

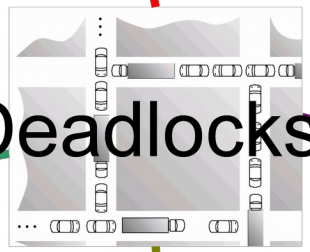
Graph with a circular wait, but not deadlock (why?)



Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.
    - Must determine if a process can terminate and if so, will it allow other processes to terminate.

Deadlocks



System Model

Instances

Resources

Use

Request

Release

Operations

Deadlock Problem

Resource owned by other waiting process

Process waits on resource

Characterization

Mutual Exclusion

Necessary Conditions

Hold and Wait

No Pre - emption

Circular Wait

Resource - Allocation Graph

Resources

Processes

Vertex

Request

Assignment

Edge

Handling Deadlocks

Avoid

Prevent

No deadlocks

Recovery

Detection

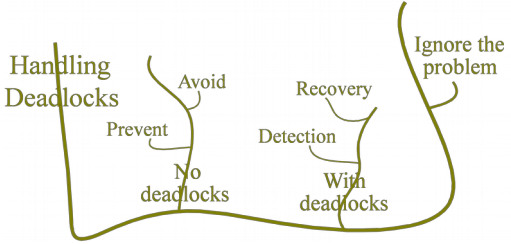
With deadlocks

Ignore the problem

Methods for handling deadlocks

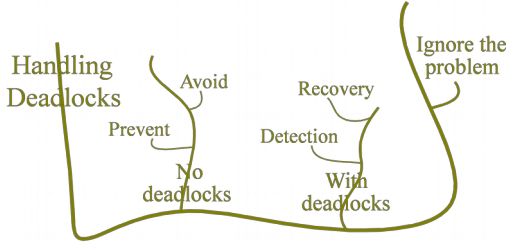
- Do not enter deadlock state
  - Deadlock prevention
    - Disallow 1 of the 4 necessary conditions of deadlock occurrence.
    - Difficult and restraining – can lead to low device utilization and low throughput.
  - Deadlock avoidance
    - Do not grant a resource request if this allocation might lead to deadlock.
- Allow deadlocks
  - Periodically check for the presence of deadlock and then recover from it.
- Ignore the problem:
  - Ignore the problem and pretend that deadlocks never occur in the system.
  - Used by most operating systems, including UNIX/Windows.

Deadlock Prevention: prevent at least one of the four conditions that lead to deadlock



- Mutual exclusion: **reduce as much as possible the use of shared resources and critical sections (almost impossible).**
- Hold and wait: **a process that requests new resources cannot hold up other processes (ask for all resources at once).**
- No pre-emption: **If a process asks for resources and cannot obtain them, it is suspended and its resources already held are released.**
- Circular wait: **define an request ordering for resources, a process must ask for resources in this order (e.g. always ask for the printer before the tape drive) (see section 7.4.4 in the textbook for more details).**

Deadlock Avoidance



Requires that the system has some additional *a priori* information available.

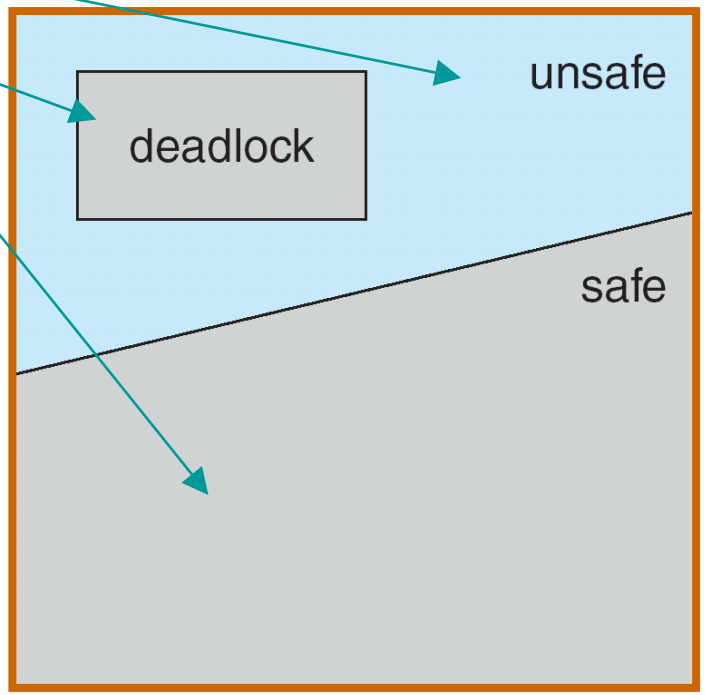
- **Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.**
- **The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.**
- **Resource-allocation *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.**
- **What is a *safe* state?!**
- **System is in *safe* state if there exists a *safe sequence* of all processes.**

Basic Facts

- If a system is in safe state ⇒ no deadlock possible.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.



Safe Sequence

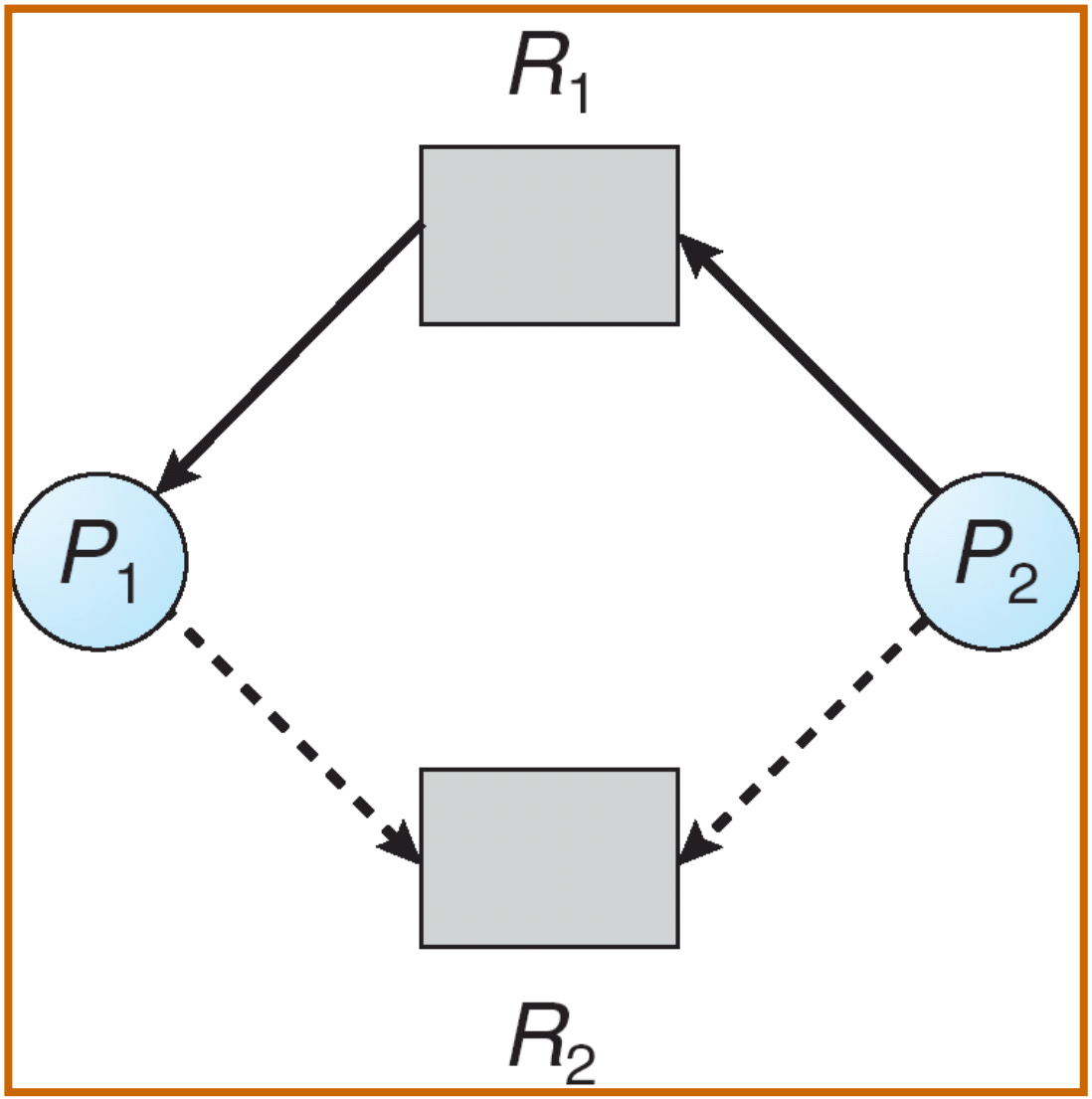
- Safe sequence is a sequence witnessing that we can run all processes to completion
- P_1 is the process that can be run to completion using only the available resources
- P_2 is the process that can be completed when P1 completes and releases its resources…
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe 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.

Resource-Allocation Graph Algorithm

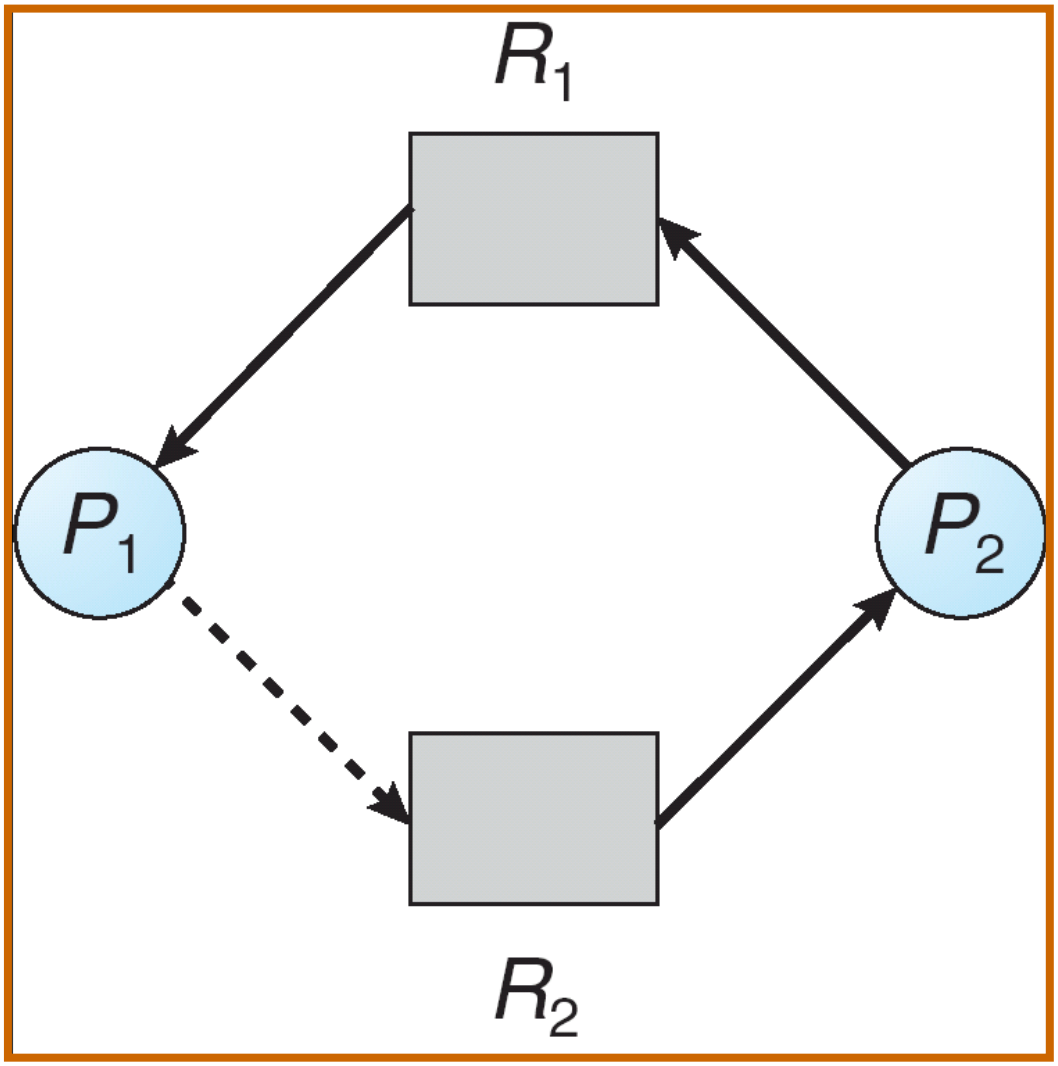
First consider the simpler case of one-instance resources

- Note: in such case, a cycle in the Resource-Allocation graph implies deadlock
- Lets introduce *claim edges*:
  - *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j; represented by a dashed line.
  - Claim edge is converted 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.
- The algorithm: if satisfying request creates a cycle in the modified R-A graph (including claim edges), reject the request

Resource-Allocation Graph with Claim Edges



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- For the case when resources have multiple instances.
- Again, each process must a priori claim the resources it is going to use
  - Must also specify quantity
- When a process requests a resource it may have to wait, even if the resource is currently available
  - The banker's algorithm decides whether to grant the resource
- Assumes that when a process gets all resources it wants, within finite amount of time it will finish and release them

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*].**

Banker's Algorithm for Process P_i

Describes how to respond to a resource request by 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 the resulting state is safe ⇒ the resources are allocated to Pi.***
- ***If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored***

Safety Algorithm

- But how do we test whether a state is safe?
- Remember, there must be a safe sequence in a safe state.
- So, we will try to find a safe sequence.
  - Find a process that can be satisfied with the current resources, give it the resources and let it finish. (include the need ressources).
  - This would be our P1.
  - When P1 finishes, release its resources, and repeat
- How long should repeat?
  - Until all processes are finished – then the state was indeed safe.
  - Or until the we cannot find a process to finish – then the state was unsafe.
- Remember, this is all as-if.

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, 2, 3, …, *n-1.*

2. Find and *i* such that both:

   (a) ***Finish* [*i*] = *false*** ***// not yet finished process***

   (b) $Need_i \leq Work$ ***// that can be finished with available resources***

   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 initial state was safe,

   Otherwise, it was unsafe

Safety Algorithm

- Great! Now I understand how it works.
- Too bad I don't understand why is works.
  - We have tried only one way to find a safe sequence
  - If we failed, a different choice of the satisfiable processes might have lead to a safe sequence, but we just made bad choices leading us to a dead end
- That is actually not true
  - If we found a satisfiable process, let it run to completion and then released its resources, we are better off (more resources available) then how we started
  - So, it does not harm us to choose any satisfiable process

Example of Banker's Algorithm

- 5 processes P_0 through P_4; 3 resource types *A* (10 instances), *B* (5 instances), and *C* (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*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		? ? ?
P_4	0 0 2	4 3 3		? ? ?

- Is the system safe?

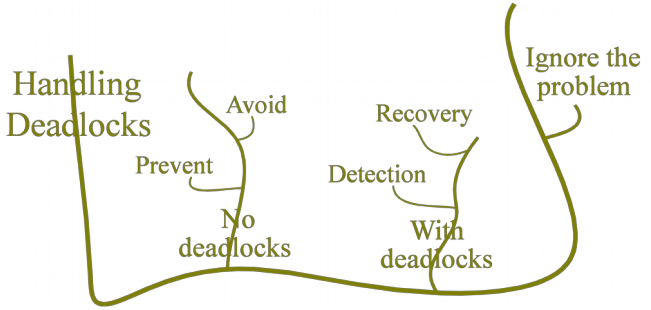
Example: P_1 requests (1,0,2)

- Check that Request ≤ Need[1](i.e, (1,0,2) ≤ (1,2,2)) ⇒ true.
- Check that Request ≤ Available (i.e, (1,0,2) ≤ (3,3,2)) ⇒ true.

	Allocation	*Need*	*Available*
	A B C	*A B CA 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 <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Deadlock Detection



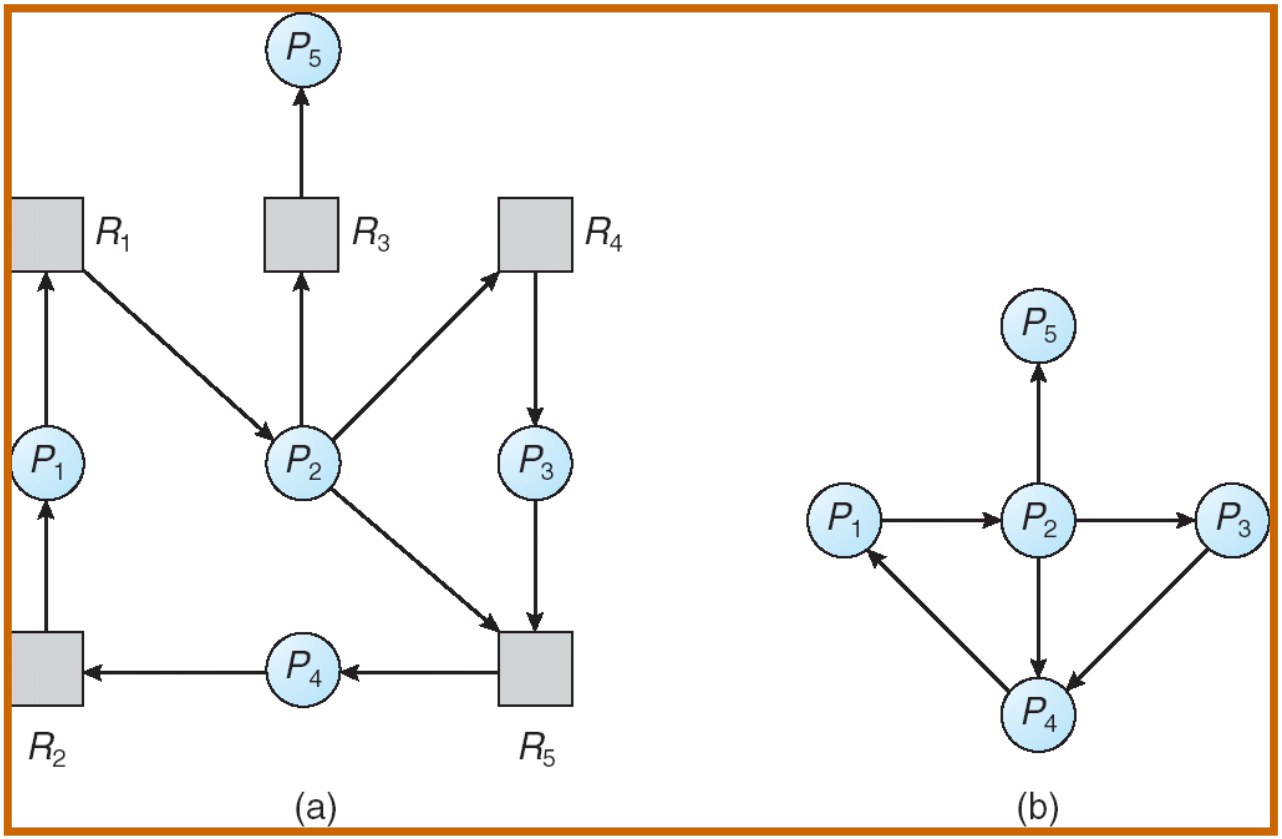
- Resource access are granted to processes whenever possible. The OS needs:
  - an algorithm to check if deadlock is present
  - an algorithm to recover from deadlock
- The deadlock check can be performed at every resource request
  - Such frequent checks consume CPU time

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.

- Periodically invoke an algorithm that searches for a cycle in the graph.
  - Requires an order of n^2 operations, where n is the number of vertices in the graph.
  - Don't want to run it too often

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

Idea:

- **Reuse the ideas from the Safety algorithm**
- **Essentially, check whether it is possible from the current state to finish all processes**
- **In other words, try to construct a safe sequence**
  - **if succeeded, not it deadlock (at least not yet)**
  - **if failed (some processes have not been finished), these processes are deadlocked**
  - **The processes that do not hold resources are considered “finished” (they are not blocking others, so they cannot participate in a deadlock cycle)**

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available* // *as-if, remember*
   (b) For *i* = 1,2, …, *n*,
   if $Allocation_i \neq 0$ or $Request_i \neq 0$
   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 ≤ *i* ≤ *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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in *Finish*[i] = true for all i.

Example (Cont.)

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

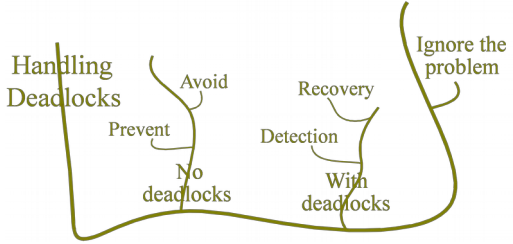
	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- 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 be affected by the deadlock when detected?
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph
  - Difficult to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination



- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
  - Need to create and maintain checkpoints for rolling back to safe points
- Starvation – same process may always be picked as victim,
  - Include number of rollback in cost factor.

Instances

Resources

Use

Request

Release

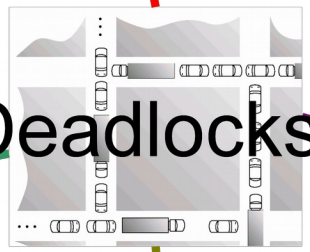
System Model

Operations

Resource owned by other waiting process

Process waits on resource

Deadlock Problem



Deadlocks

Characterization

Mutual Exclusion

Necessary Conditions

Hold and Wait

No Pre - emption

Circular Wait

Resource - Allocation Graph

Vertex

Ressources

Processes

Edge

Request

Assignment

Handling Deadlocks

Avoid

Prevent

No deadlocks

Recovery

Detection

With deadlocks

Ignore the problem