Bucknell UNIVERSITY

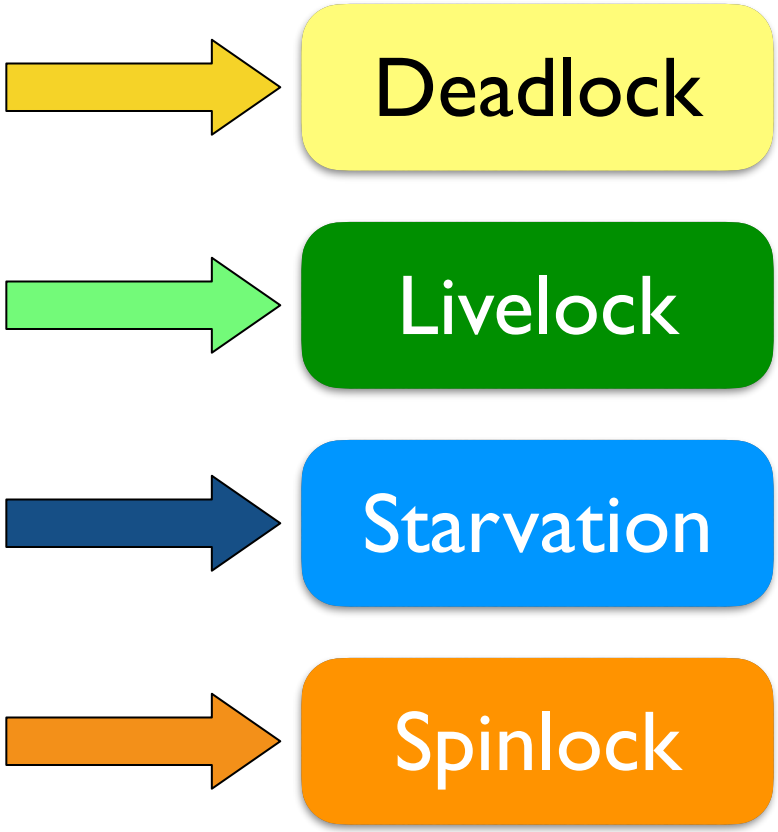
Deadlock

CSCI 315 Operating Systems Design

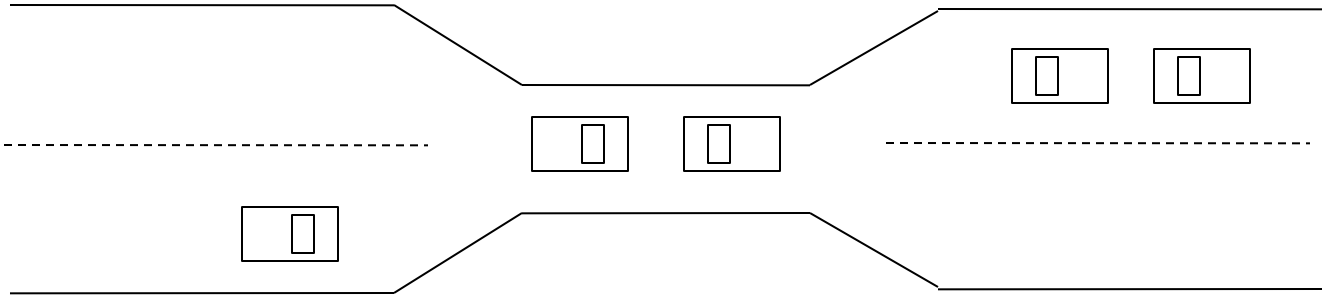
Department of Computer Science

Notice: The slides for this lecture have been largely based on those from an earlier edition of the course text *Operating Systems Concepts, 8th ed.*, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.

Concepts



Deadlock: 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.

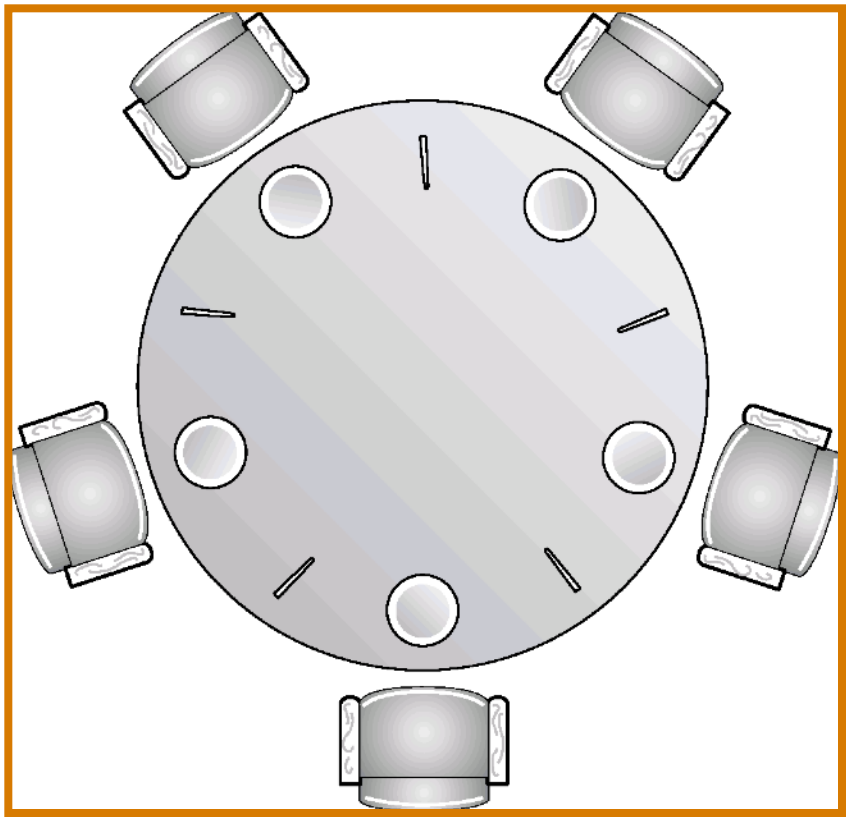
Deadlock: *Dining-Philosophers* Example

All philosophers start out hungry and that they all pick up their left chopstick at the same time.

When a philosopher manages to get a chopstick, it is not released until a second chopstick is acquired and the philosopher has eaten his share.

Question: Why did deadlock happen? Enumerate all the conditions that have to be satisfied for deadlock to occur.

Question: What can be done to guarantee that deadlock won't happen?



A 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.
- Each process uses a resource as follows:
  - request
  - use
  - release

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 $\{P_0, P_1, \ldots, P_0\}$ 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

  $P_2, \ldots, P_{n-1}$ 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.

Resource Allocation Graph

Graph: G=(V,E)

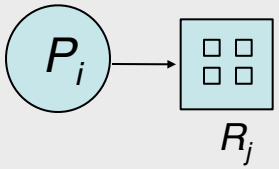
- The nodes in V can be of two types (partitions):
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Request edge – directed edge $P_1 \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph

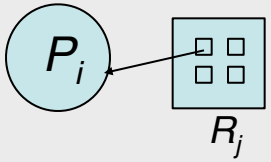
- Process

- Resource Type with 4 instances

- P_i requests instance of R_j



- P_i is holding an instance of R_j

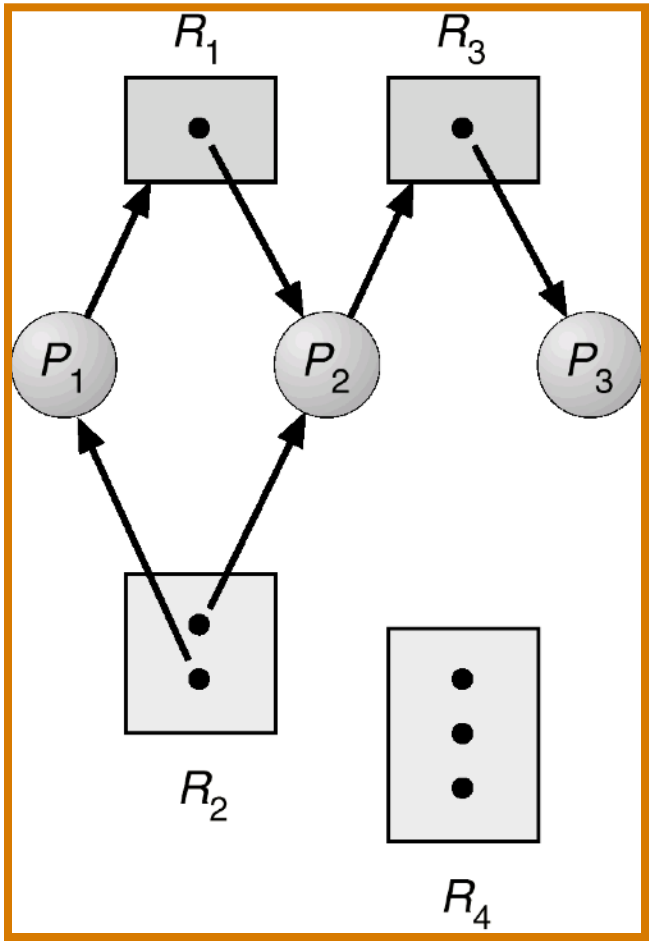


Basic Facts

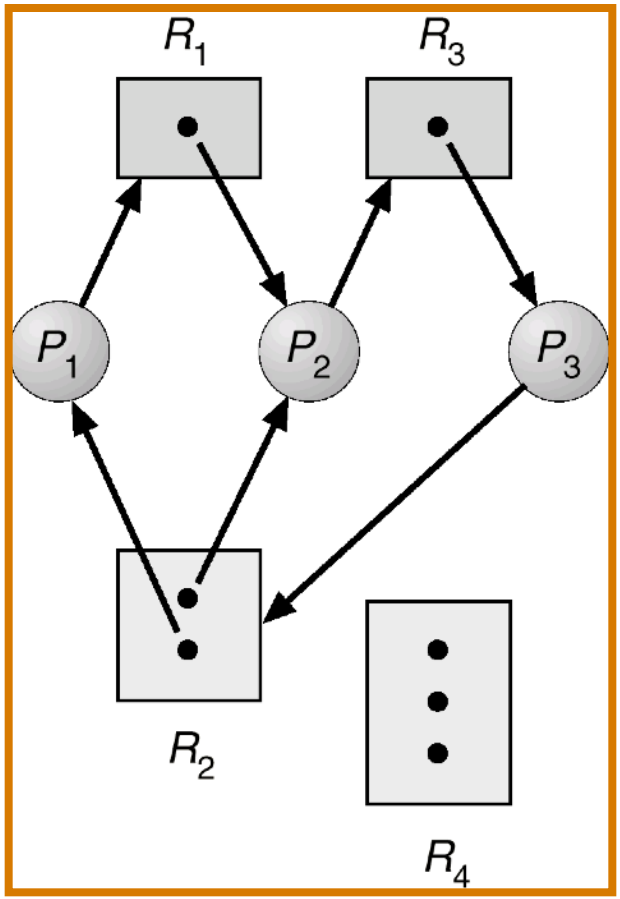
- **If graph contains no cycles**
  ⇒ no deadlock

- **If graph contains a cycle**
  ⇒ there may be deadlock:
  - if only one instance per resource type, then there is deadlock
  - if several instances per resource type, there may be deadlock

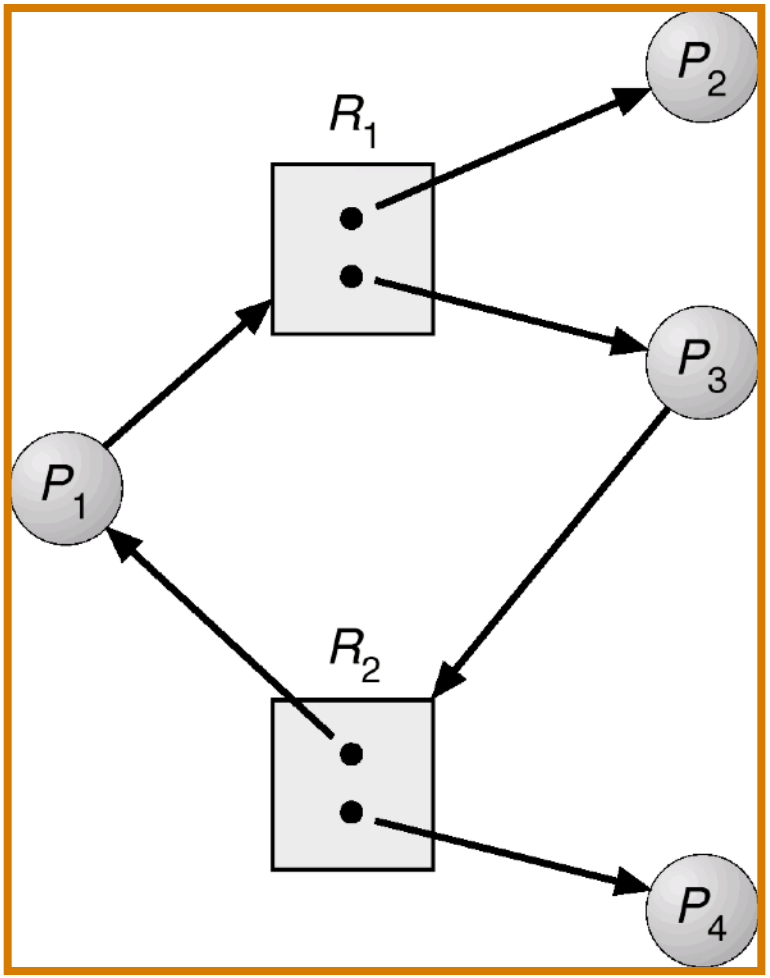
Example of a Resource Allocation Graph



Resource Allocation Graph with Deadlock

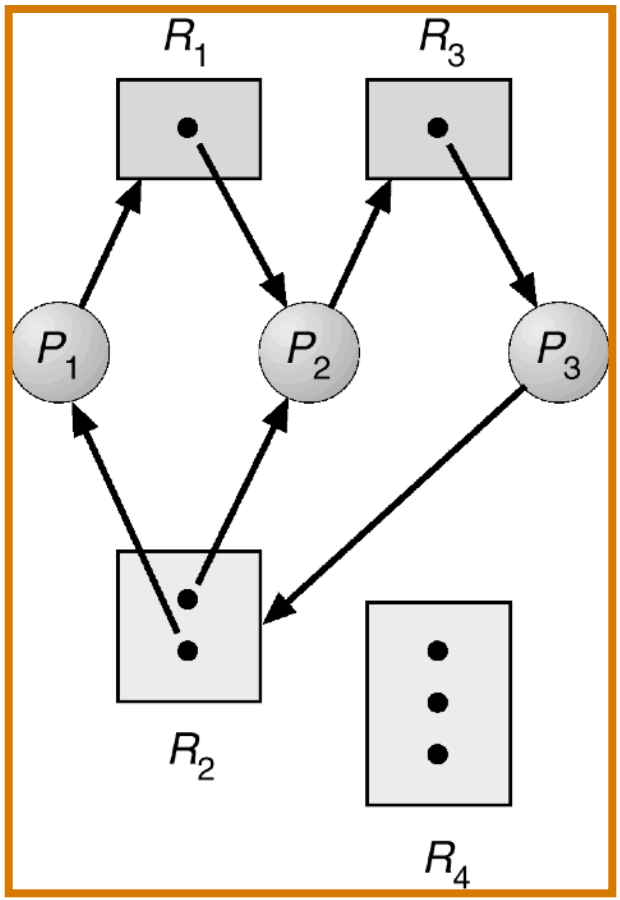


Resource Allocation Graph with Cycle but No Deadlock



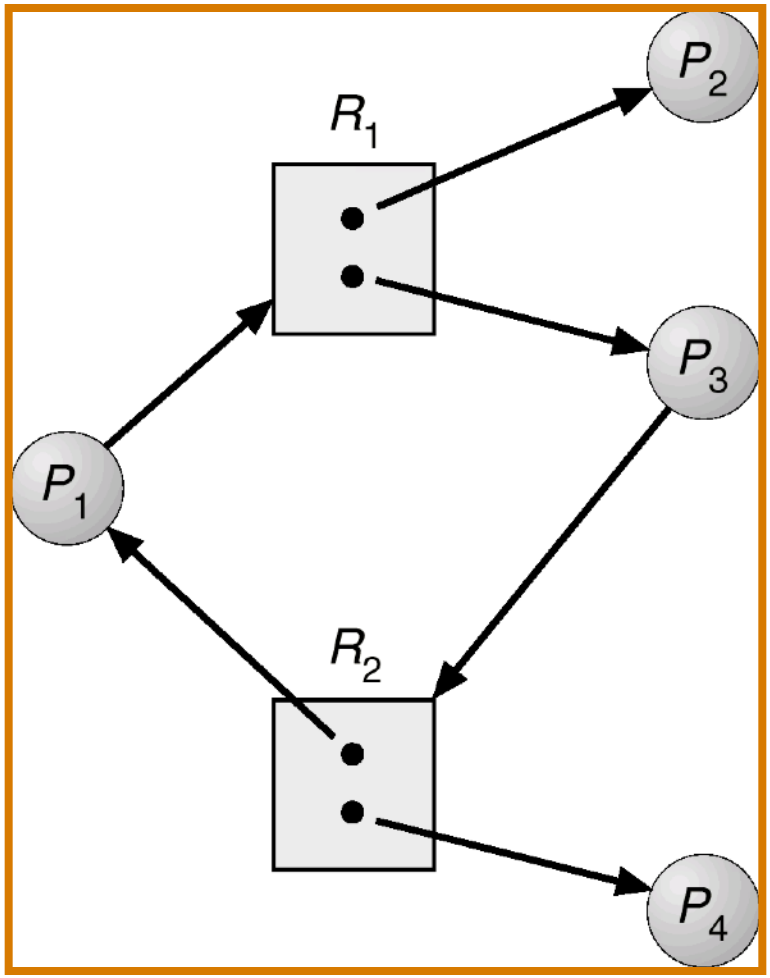
Resource Allocation Graph

Example 1



Resource Allocation Graph

Example 2



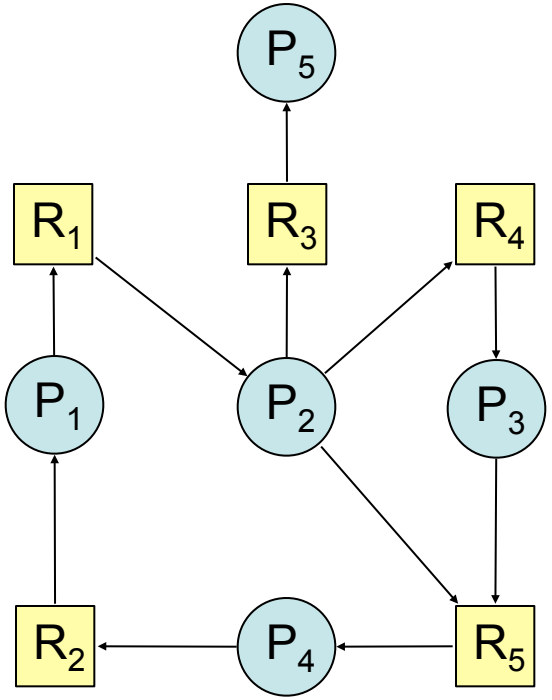
Single Instance of Each Resource Type

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

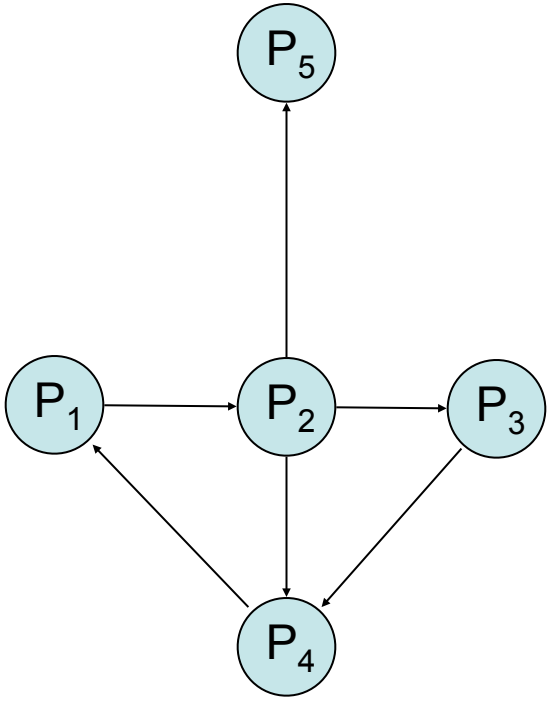
- Periodically invoke an algorithm that searches for a cycle in the graph.

- An algorithm to detect a cycle in a graph requires $O(|V|+|E|)$ time.

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Use only when there is a single instance of each resource type!

Methods for Handling Deadlocks

- Make it impossible for the system ***to go into deadlock***.
- Guarantee through careful decisions we ***avoid letting the system go into deadlock***.
- Relax and let things move along on their own, but ***provide ways to recover from deadlock***.
- Ignore the problem and pretend that deadlocks never happen.

What to do about deadlock

Prevention

Avoidance

Recovery

Deadlock Prevention

Constrain how requests can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.

- **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.
  - Low resource utilization; starvation possible.

Deadlock Prevention

Constrain how requests can be made

- **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.

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

Deadlock Avoidance

The system has additional *a priori* information

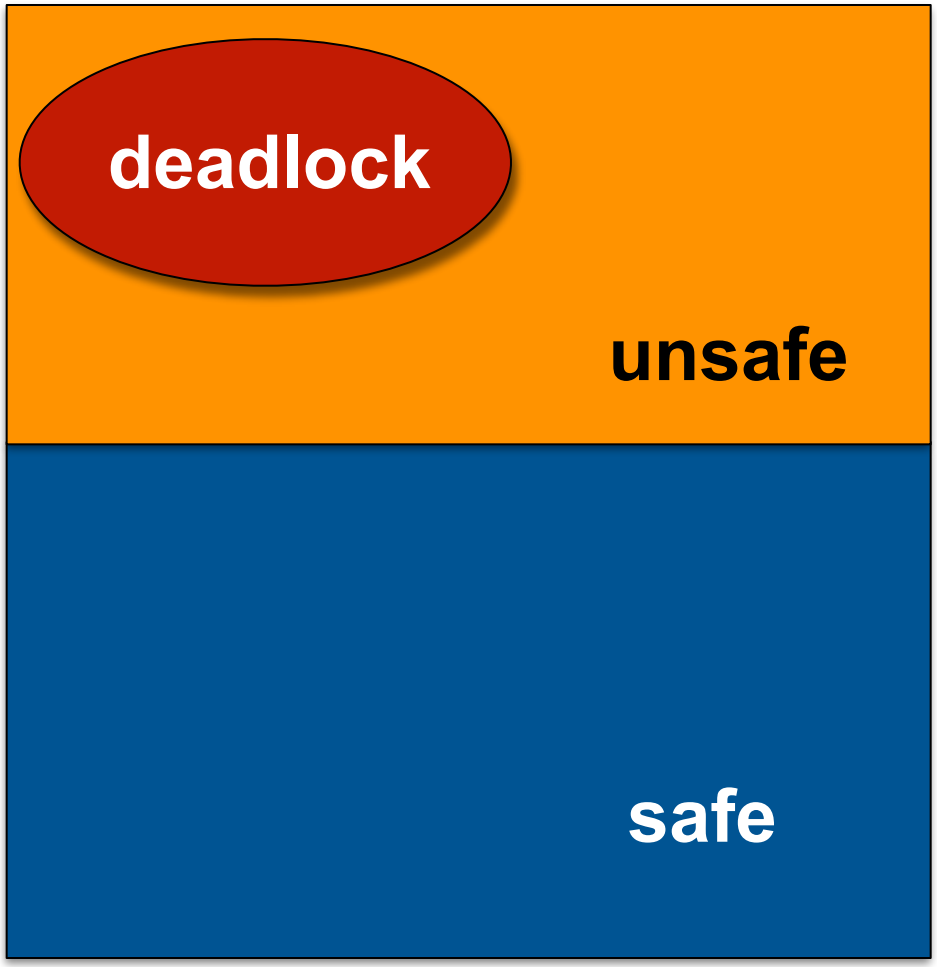
- The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe States

- Sequence **<P_1, P_2, …, P_n>** is ***safe*** if for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus the 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.

- The system is in a ***safe state*** if there exists a safe sequence for all processes.
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a **safe state**.

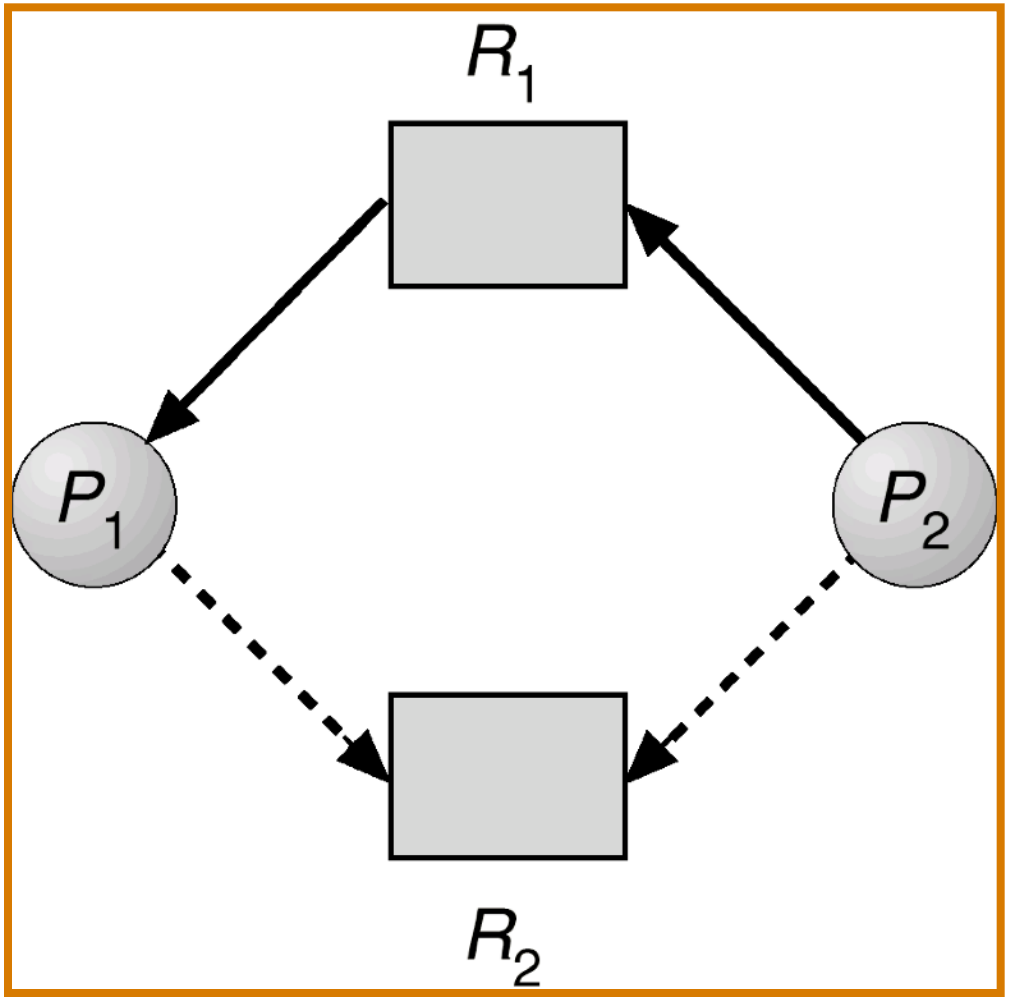
Safe, Unsafe, and Deadlock States



Basic Facts

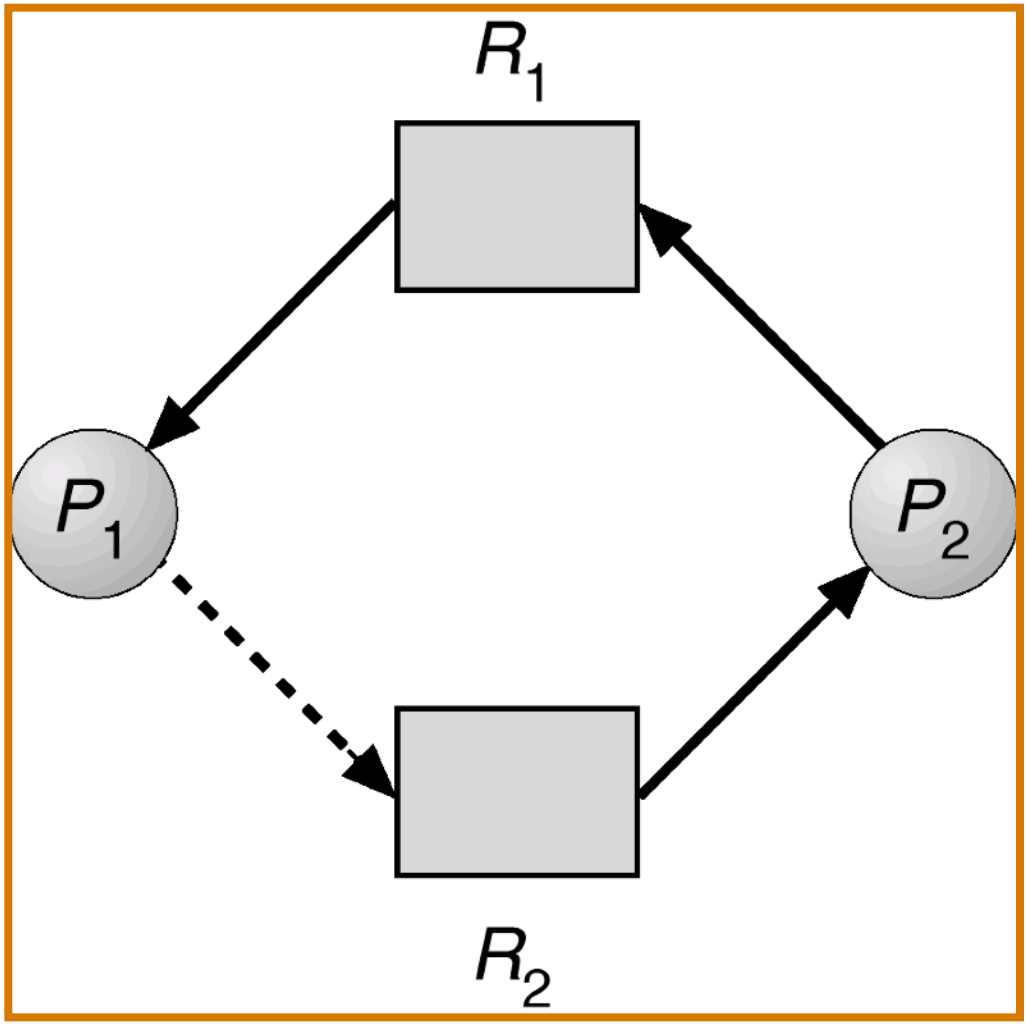
- If a system is in a safe state there can be no deadlock.
- If a system is in unsafe state, there exists the **possibility** of deadlock.
- **Avoidance** strategies ensure that a system will never enter an unsafe state.

Resource-Allocation Graph for Deadlock Avoidance



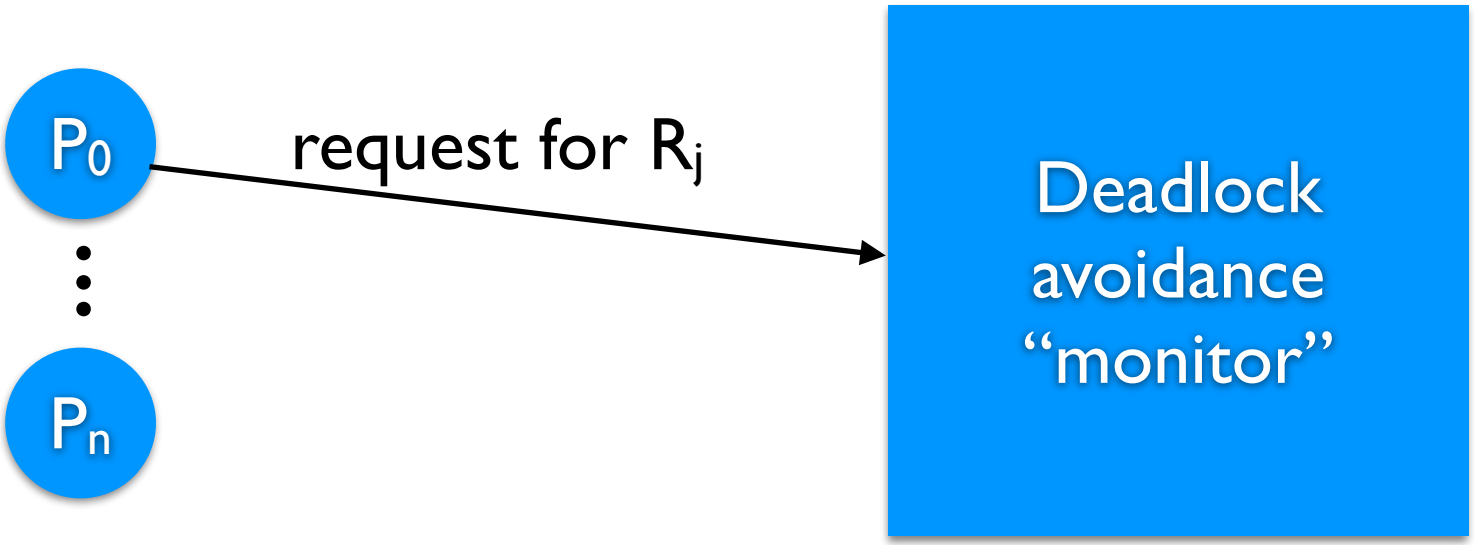
Arrows like this indicate that the process may possibly make this request in the future:

Unsafe State In Resource-Allocation



Resource-Allocation Graph Algorithm

Goal: *avoid* an unsafe state.



When processes start, they "declare" the resources they need

Applicable only when there is a single instance of each resource type

Resource-Allocation Graph Algorithm

Goal: *avoid* an unsafe state.

- *Claim edge* $P_i \rightarrow R_j$ indicates 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.

The Banker's Algorithm

- Applicable when there are multiple instances of each resource type.
- In a bank, the cash must never be allocated in a way such that it cannot satisfy the need of **all its customers**.
- Each process must state a priori the maximum number of instances of each kind of resource that it will ever need.
- 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.

The Banker's Algorithm: Data Structures

Let ***n* = number of processes or threads**,
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.

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

The Banker's Algorithm(s)

Safety	Is the system in a safe state?
Resource-Request	If a request is granted, will the system be in a safe state?
Detection	Is the system in a deadlock state?

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 an *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4.

$O(mn^2)$ operations

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 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:

   $Available = Available = Request_i$

   $Allocation_i = Allocation_i + Request_i$

   $Need_i = Need_i - Request_i$;

   $O(mn^2)$ operations

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

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 <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?

When Deadlock Happens

- Another way to deal with deadlock is not to use either prevention or avoidance. The system may enter a deadlock state; the OS will deal with that [when | if] it happens.

- What is needed in such a system:
  - a **detection algorithm** to determine when deadlock states are entered, and
  - a **recovery scheme** to get the system back on a safe state.

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 $\boldsymbol{Allocation_i \neq 0}$, 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.

$O(mn^2)$ operations

3. ***Work* = *Work* + $\boldsymbol{Allocation_i}$**
   Finish*[*i*] = *true
   Go to step 2.

4. If ***Finish*[*i*] == false**, for some *i*, $0 \leq i \leq n$, then the system is in deadlock.
   Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked.

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 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of the system?
  - Can reclaim resources held by process P_0, but have insufficient resources to fulfill the requests of other processes.
  - 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.

Recovery from Deadlock

- 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.
- **Starvation** – same process may always be picked as victim; include number of rollbacks in cost factor.

Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - **prevention**
  - **avoidance**
  - **detection**

  allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.