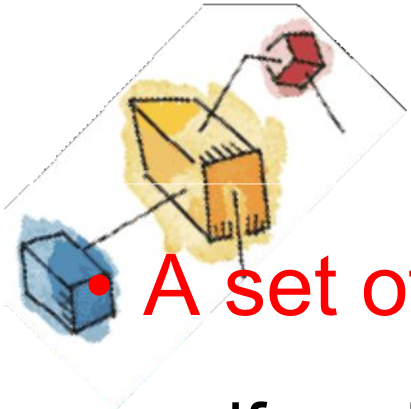
Operating Systems

Deadlocks

The Deadlock Problem



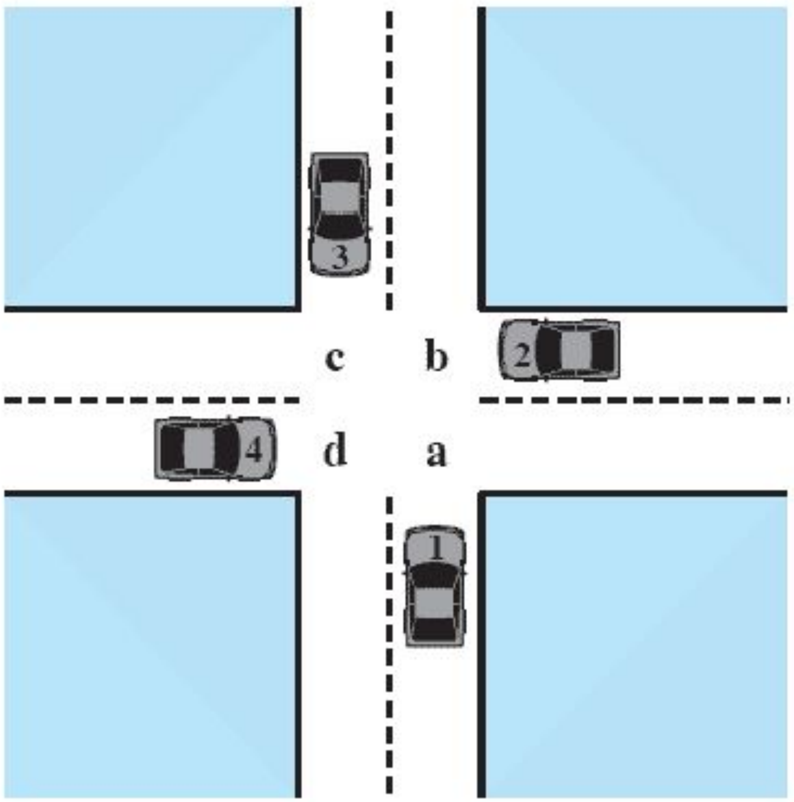
- A set of processes is deadlocked:
  - if each process in the set is waiting for an event that only another process in the set can cause.
  - all processes are waiting, none of them cause wakeup event.
- **Example 1 Example 2**

System has 2 disk drives
P_1 and P_2 each hold one disk drive and each needs another one.

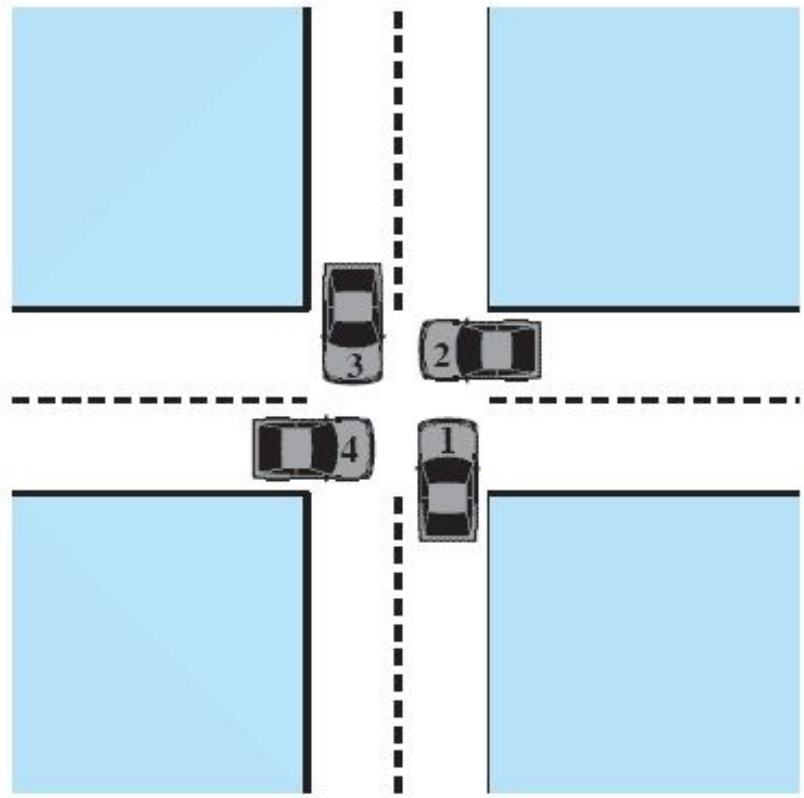
Mutex *A* and *B*, initialized to 0

P_0	P_1
Mutex_lock (A);	Mutex_lock(B);
Mutex_lock (B);	Mutex_lock (A) ;

Deadlock



(a) Deadlock possible



(b) Deadlock

Illustration of Deadlock

Resource-Allocation Graph

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

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource Allocation Graphs

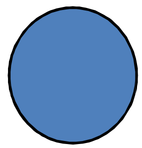
- Directed graph that depicts a state of the system of resources and processes

(a) Resouce is requested

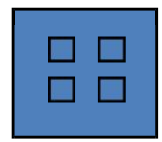
(b) Resource is held

Resource-Allocation Graph

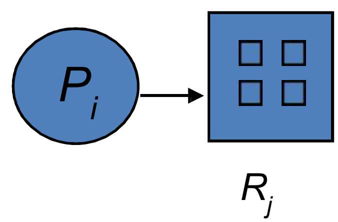
- **Process**



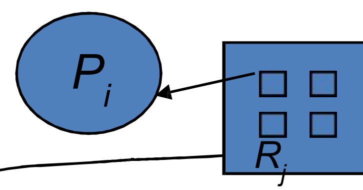
- **Resource Type with 4 instances**



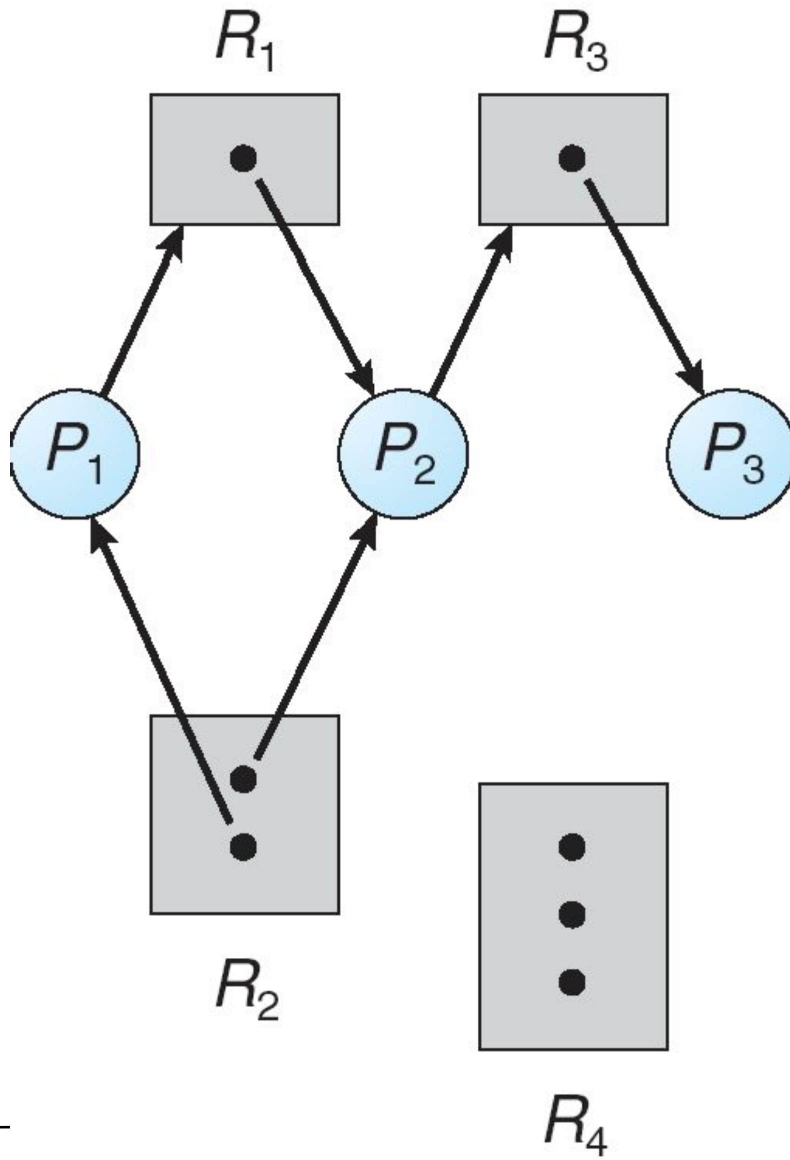
- ***P_i* requests instance of *R_j***



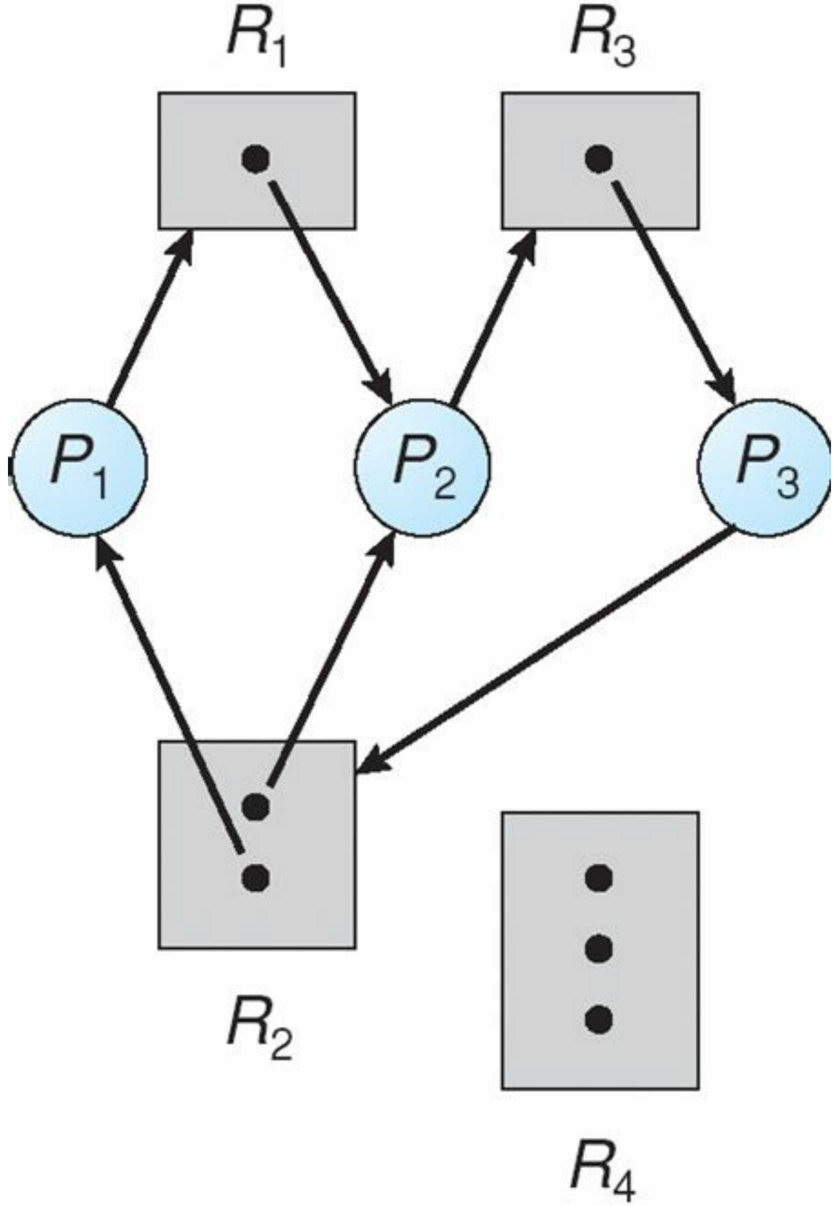
- ***P_i* is holding an instance of *R_j***



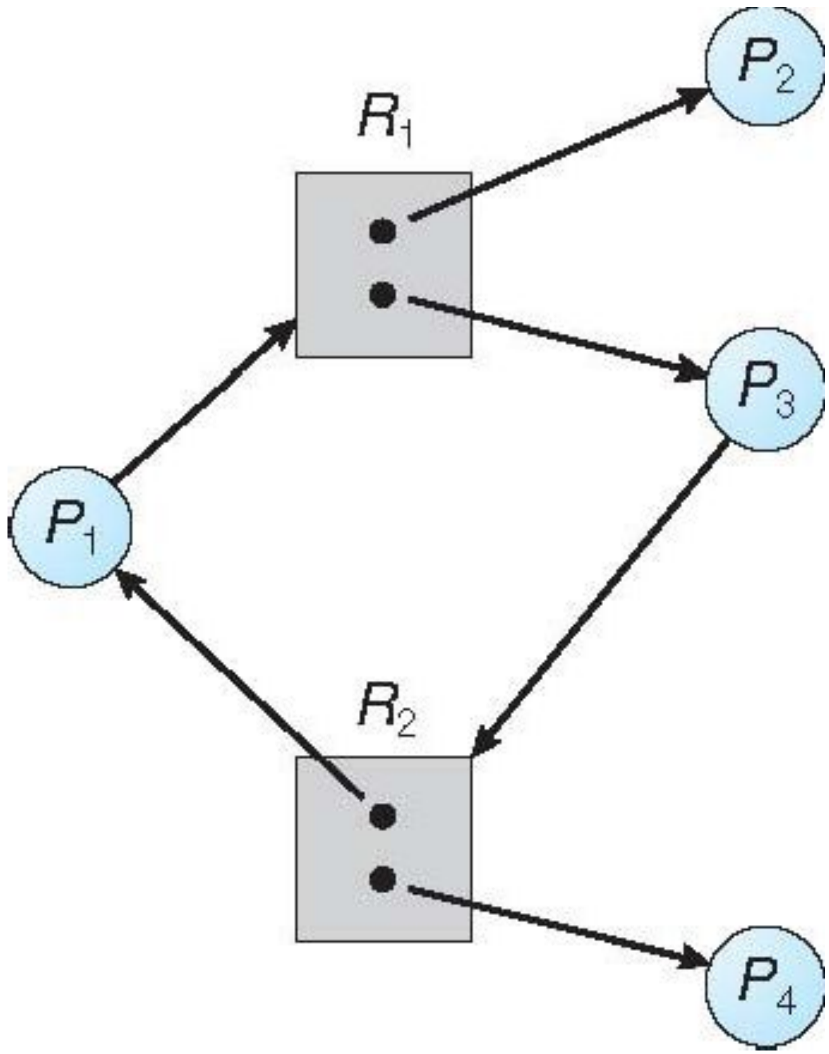
Example of a Resource Allocation Graph

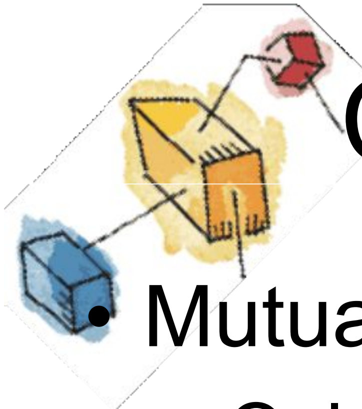


Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock

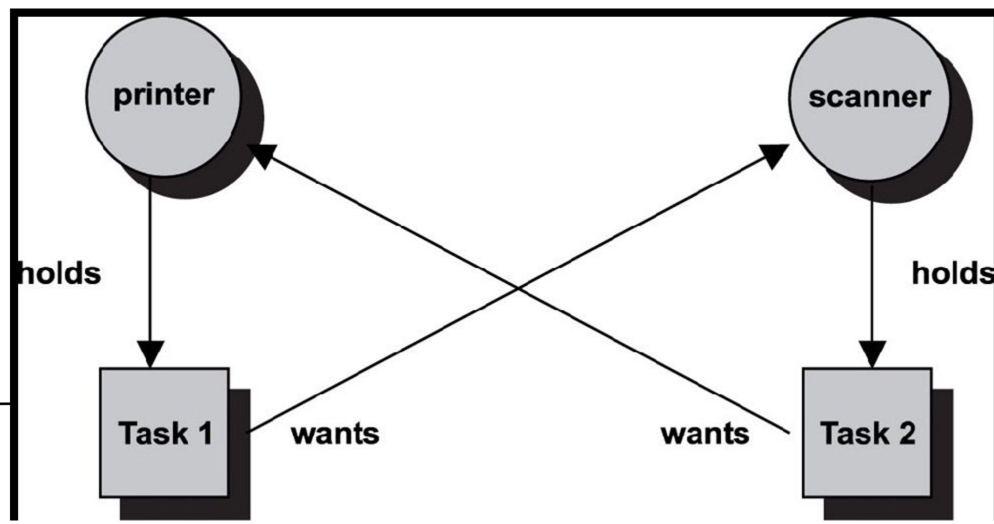


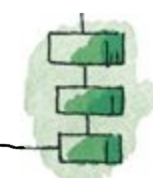


Conditions for Deadlock

- Mutual exclusion
  – Only one process may use a resource at a time

- Hold-and-wait
  – A process may hold allocated resources while awaiting assignment of others

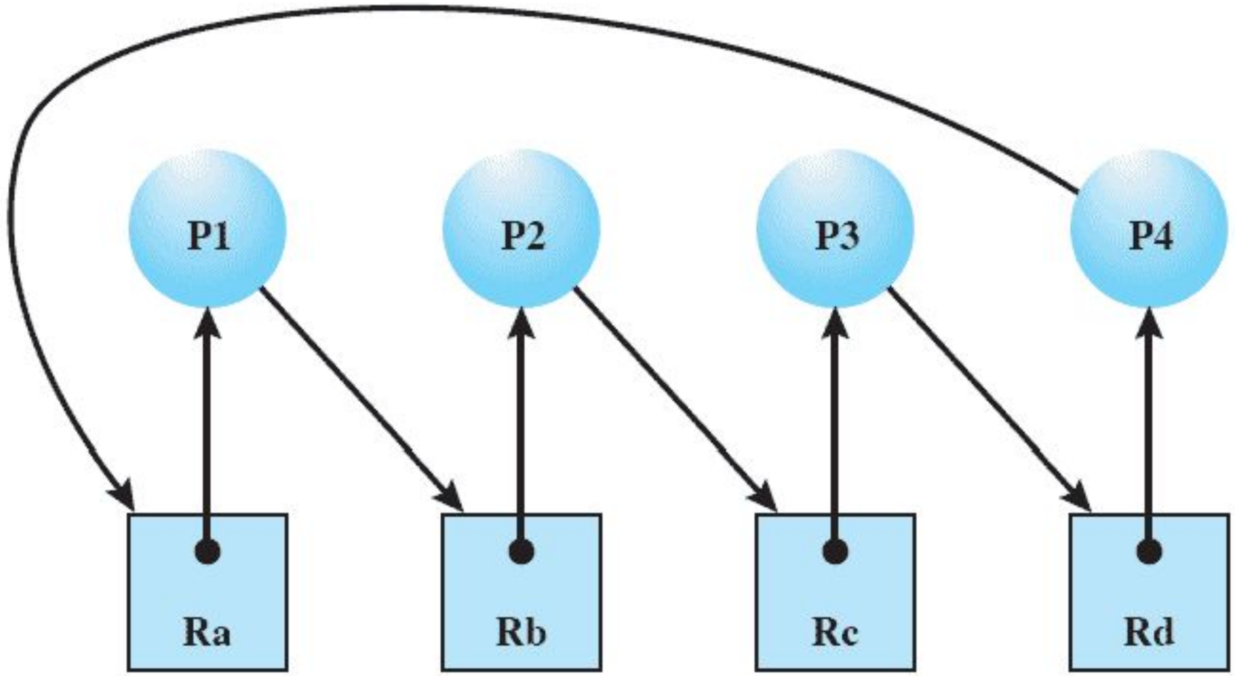




Conditions for Deadlock

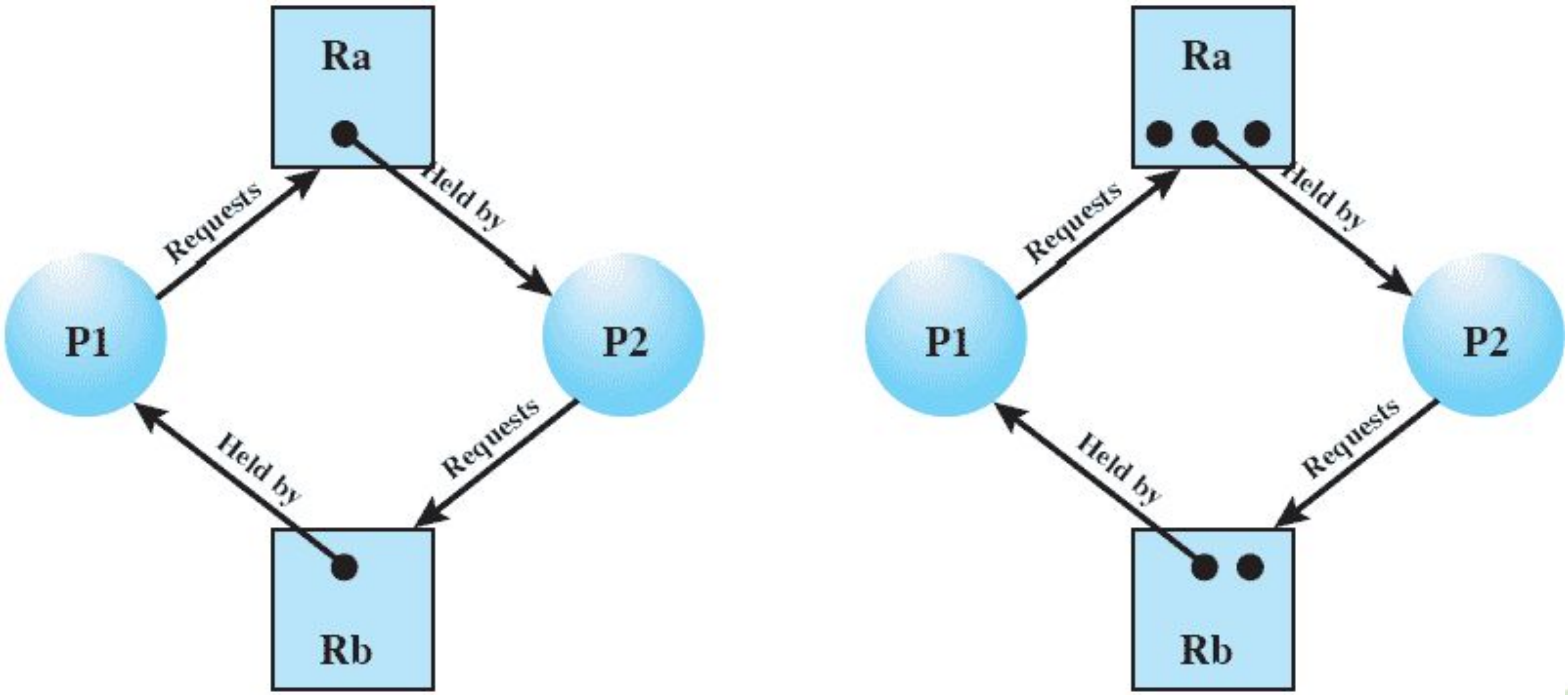
- No preemption
  - No resource can be forcibly removed form a process holding it.
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

Resource Allocation
(



Circular Wait

- Deadlock Must Occurs:
  - all four of these conditions are present.



(c) Circular wait

(d) 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

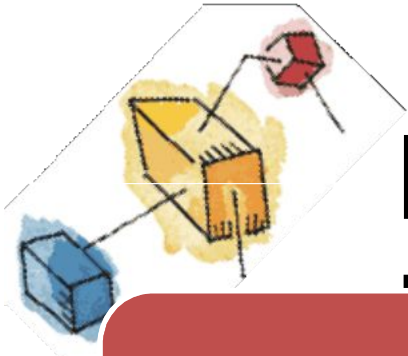
- Ensure that the system will ***never*** enter a deadlock state
  - deadlock prevention, deadlock avoidance
- Allow the system to enter a deadlock state and then recover
  - detect a deadlock and recover
- Ignore the problem and pretend that deadlocks never occur in the system;
  - used by most operating systems, including Linux and windows

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion**
  - Must hold for non-sharable resources
    - Printer can not be simultaneously shared by several processes

- **Hold and Wait** – must guarantee
  - a requested process must not hold any other resources
  - Maintain **protocols** (next slides)

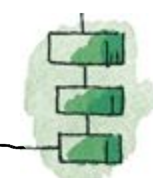


Hold & Wait

1. Process must be allocated all its requested resources before it begins execution,

- OR

2. Allow process to request resources only when the process has none



Example Problem

Protocols

- Copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.

1

- **Request: DVD** drive, disk file, printer
  - Holding the printer for entire execution although it needs only at the end

2

- **Request 1: Copy file from DVD** drive to disk then release.
- **Request 2: Copy file** Disk to Printer and then release

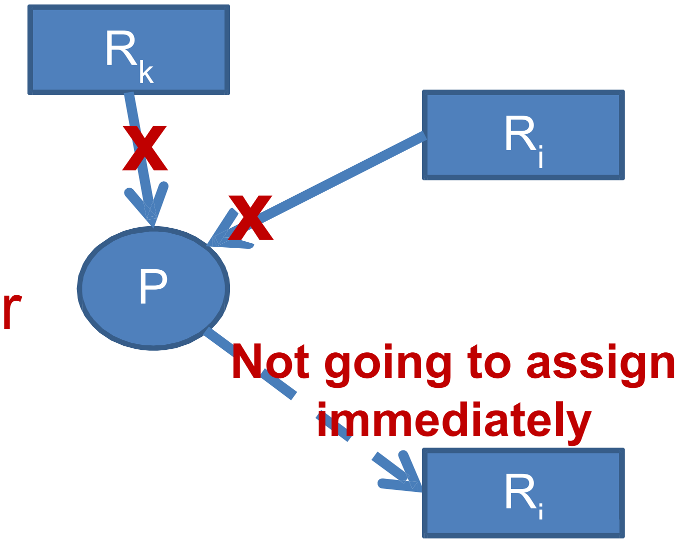
Disadvantages: i) low resource utilization (allocated but not used)

ii) Starvation: may have to wait for popular resources

Deadlock Prevention (Cont.)

- **Preemption**

P starts execution after gaining all R_i R_k R_j



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

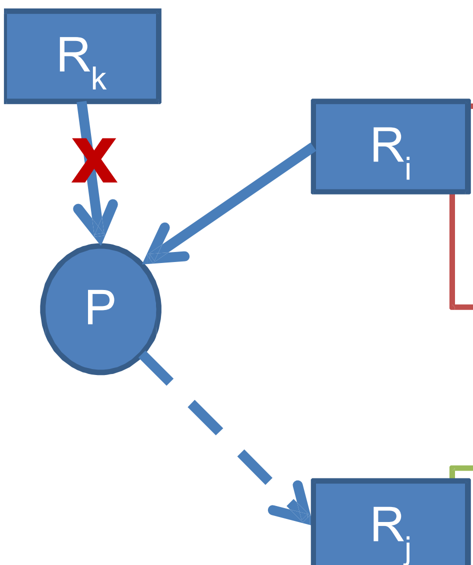
Deadlock Prevention (Cont.)

- **Circular Wait** – Resource will be allocated in increasing order.

F: R N→

F(tape drive) =1
F(disk drive) =5
F(printer) =12

Ri < Rj < Rk



P requests R_j only when $F(R_i)>F(R_i)$

P requests R_j only when it released $F(R_k)>=F(R_j)$

Proof: Circular wait will not occur if the two conditions hold

- <u>Proof by Contradiction</u>

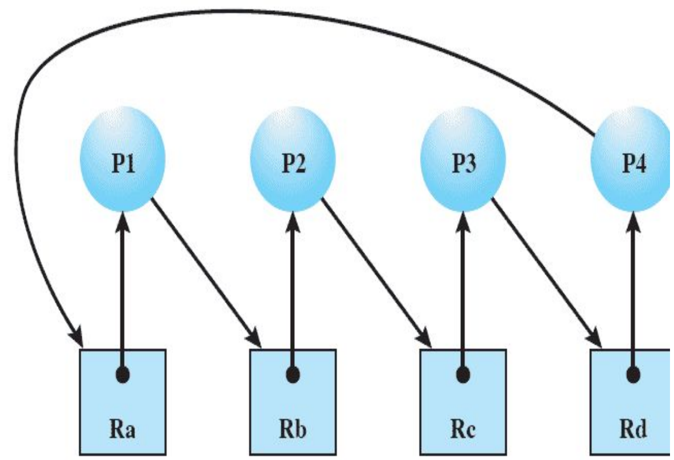
- Lets circular wait exits for $\{P_0, P_1, P_2 \ldots P_n\}$
  – P_i waits for R_i, and R_i holds by process P_{i+1} and it continues till P_n waiting for P_0's resource R_0.

Hence, P_{i+1} holds resource R_i and requested R_{i+1}

- $F(R_i)<F(R_{i+1})$ for all i
- But it means
- $F(R_0)<F(R_1)<F(R_2)<F(R_3)< \ldots\ldots < F(R_n)<F(R_0)$

By transitivity $F(R_0)<F(R_0)$ which is impossible

No Circular Wait- Proved



Problem in Deadlock Prevention Protocols

- Low device utilization
- Reduce system throughput

  – Deadlock avoidance

Deadlock Avoidance

- A decision is made dynamically whether
  – the current resource allocation request (if granted), leads to a deadlock ??
- Requires:
  – knowledge of future process requests

Deadlock Avoidance Algorithm

- Dynamically examines the resource allocation state – and ensure, there can never be a circular-wait condition
  - – the state is safe.

Resource Allocation State:

– # of available & allocated resources and maximum demand

Safe State

- Allocate resources to each process (up to its maximum demand) in some order and avoid deadlock.

- Holds a safe sequence $<P_1, P_2, \ldots, P_n>$

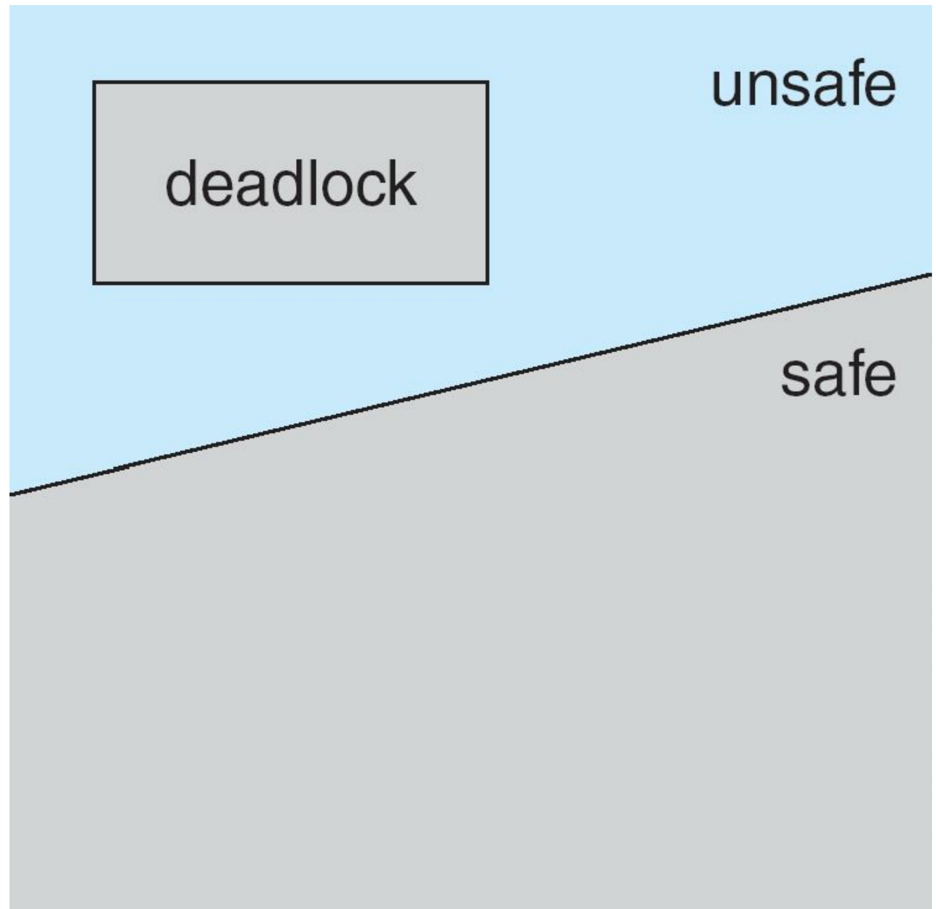
Safe, Unsafe, Deadlock State

A safe is not a deadlock state

A deadlock state is in unsafe state

Not all unsafe states are in deadlock.

However, unsafe state may lead to a deadlock.



Safe, Unsafe, Deadlock State

	Maximum Need	Current Need
P_0	10	5
P_1	4	2
P_2	9	2

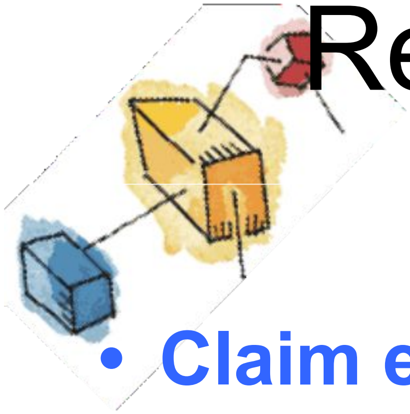
- Consider the system has twelve magnetic tape drives.
- What will happen if P2 requests and is allocated one more tape drive?

Avoidance algorithms

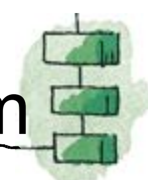
- Single instance of a resource type
  - –Use a **resource-allocation graph**

- Multiple instances of a resource type
  - – Use the **banker’s algorithm**

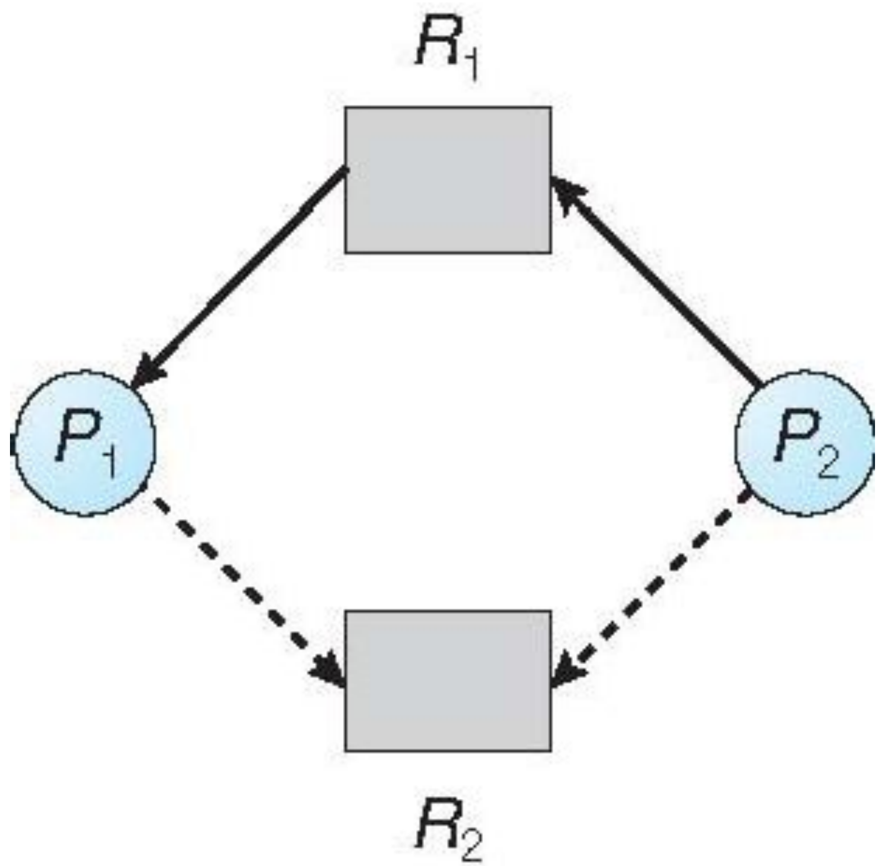
Resource-Allocation Graph Scheme



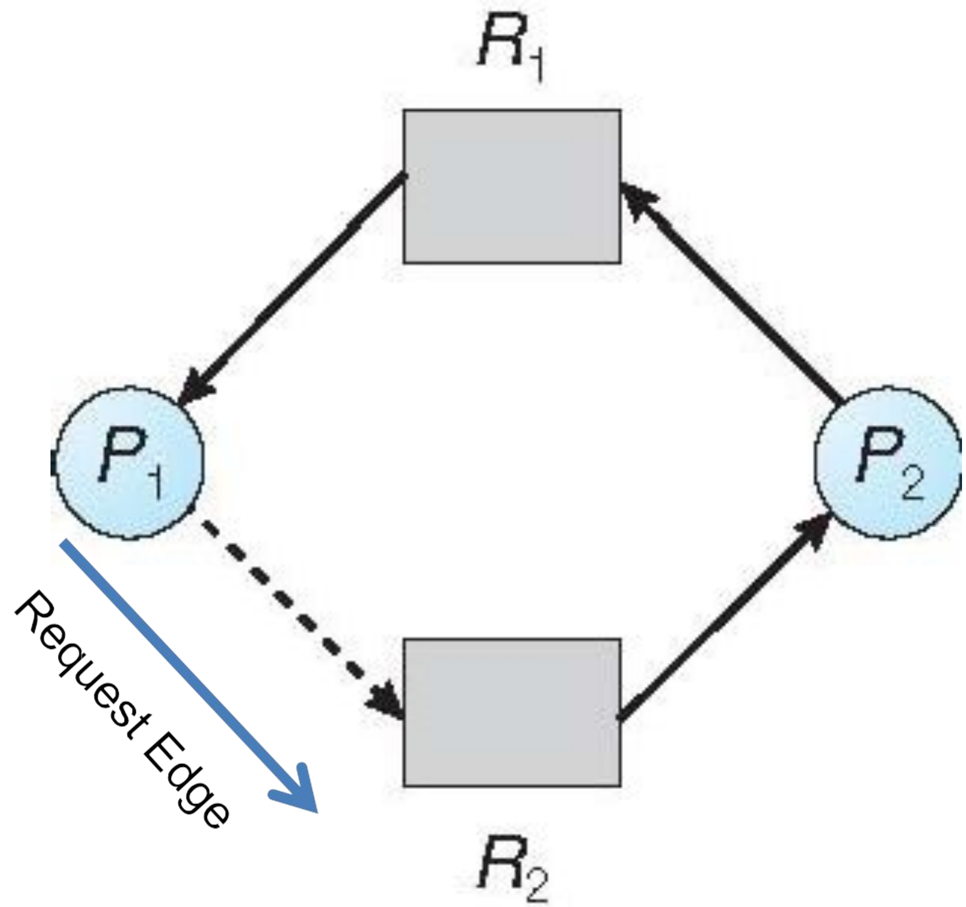
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; [represented by a dashed line]
  - Claim edge converts to request edge when a process requests a resource
  - Request edge converted to an assignment edge when the resource is allocated to the process
  - When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



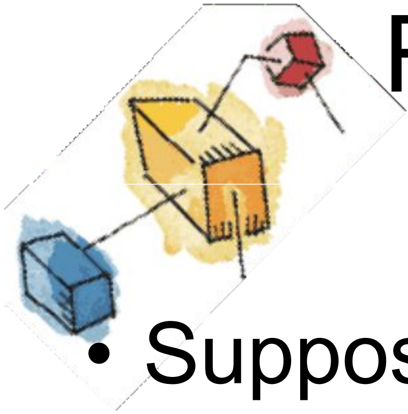
Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



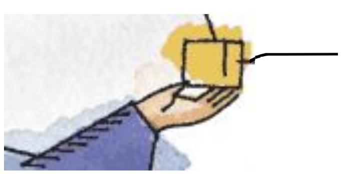
May Have Deadlock (in future)



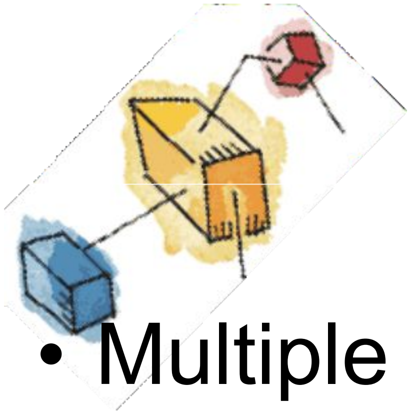
Resource-Allocation Graph Algorithm

- Suppose 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.

If no cycle exists, the allocation of the resource will leave the system in a safe state

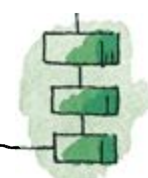


If cycle found, the allocation of the resource will leave the system in an unsafe state. Thus wait for the request to satisfy.



Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time



Data Structures for the Banker's Algorithm

- Let *n* = number of processes, and *m* = number of resource types.
- **Available**: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available
- **Max**: *n x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j
- **Allocation**: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_j
- **Need**: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task

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

Safety Algorithm

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

   Work = *Available*

   Finish [i] = *false* for i = 0, 1, …, n- 1

2. Find an i such that both:

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

   (b) $Need_i \leq Work$

   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

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*
	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)

- 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	

- 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 not be granted(un-available resource)
- Can request for (0,2,0) by P_0 be granted?
  - Can not be granted (unsafe state)

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:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- ***Go for safety algorithm***
- *If safe $\Rightarrow$ the resources are allocated to Pi*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation*

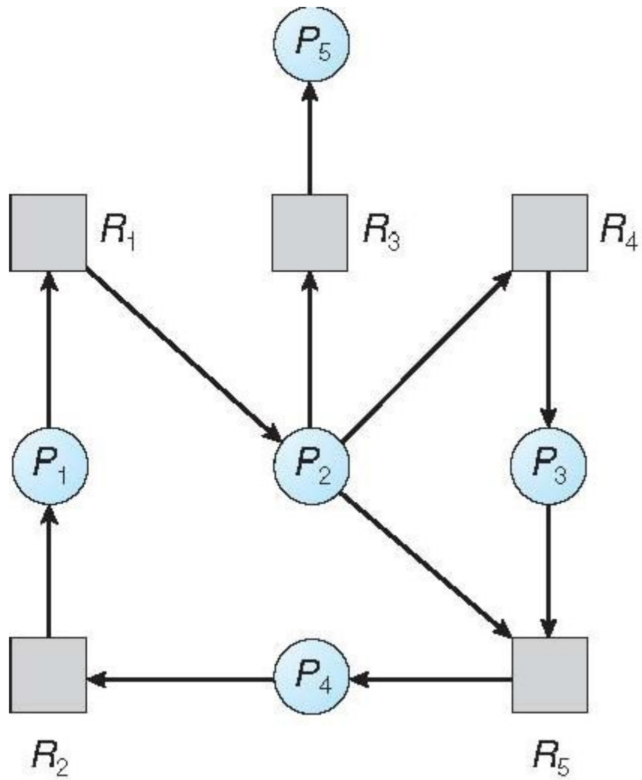
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

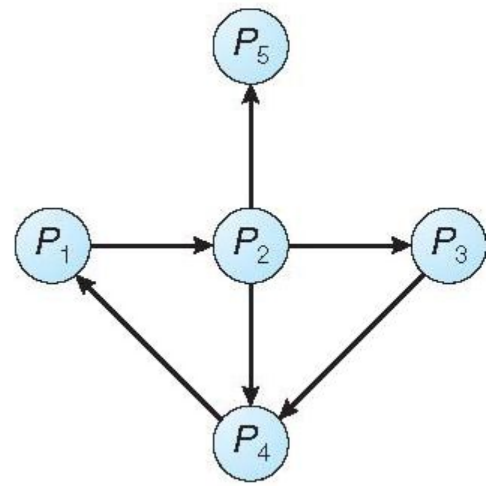
- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm
  - searches for a cycle in the graph.
  - If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation Graph



(b)

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 $i = 1,2, \ldots, n$, 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, $1 \le i \le n$, then the system is in deadlock state. Moreover, if $Finish[i]$ == *false*, then P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ 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	2
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.

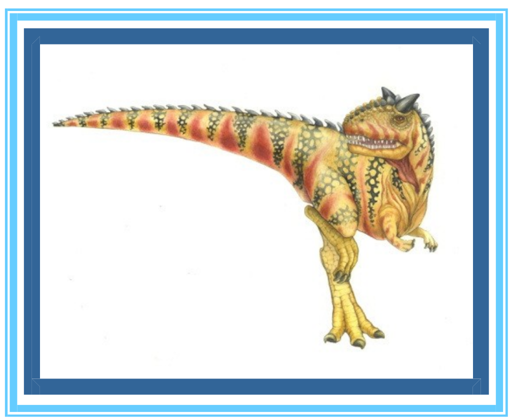
Recovery from Deadlock: Process Termination

- 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
- Problem:
  - starvation
  - same process may always be picked as victim, include number of rollback in cost factor.

Chapter 8
Reading 8.1-8.6



Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

Determination of an Unsafe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) **Initial state**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) **P1 requests one unit each of R1 and R3**

Deadlock Avoidance

```
struct state {
       int resource[m];
       int available[m];
       int claim[n][m];
       int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                         /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Deadlock Avoidance

```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible) {
      <find a process Pk in rest such that
         claim [k,*] – alloc [k,*] <= currentavail;>
      if (found) {                          /* simulate execution of Pk */
         currentavail = currentavail + alloc [k,*];
         rest = rest - {Pk};
      }
      else possible = false;
   }
   return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

Deadlock Detection

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Figure 6.10 Example for Deadlock Detection