Chap. 7) Deadlocks

경희대학교 컴퓨터공학과

방 재 훈

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)*

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

Handling Deadlocks

- Deadlock prevention
  - ✓ Restrain how requests are made
  - ✓ Ensure that at least one necessary condition cannot hold
- Deadlock avoidance
  - ✓ Require additional information about how resources are to be requested
  - ✓ Decide to approve or disapprove requests on the fly
- Deadlock detection and recovery
  - ✓ Allow the system to enter a deadlock state and then recover
- Just ignore the problem altogether!

Deadlock Prevention

Restrain the ways request can be made

- Mutual Exclusion
  - ✓ Not required for sharable resources
  - ✓ Must hold for nonsharable resources
  - ✓ Make as few processes as possible actually claim the resource

- 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 (Cont'd)

- 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
  - ✓ $F: R \rightarrow N$, where R = {R_1, R_2, …, R_n} is the set of resource types and N is the set of natural numbers
  - ✓ Whenever a process requests an instance of R_j, it has released any resources R_i such that $F(R_i) >= F(R_j)$

Resources

- Resources
  - ✓ Can only be used by one process at a time
  - ✓ Can be both hardware and software
    - e.g., tape drives, printers, system tables, database entries, memory locations
  - ✓ Request → Allocate → Use → Release
- Preemptable resources
  - ✓ Sharable resources
  - ✓ Can be taken away from a process with no ill effects
- Nonpreemptable resources
  - ✓ Nonsharable resources
  - ✓ Will cause the process to fail if taken away
  - ✓ Generally deadlocks involve nonpreemtable resources

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 utilizes a resource as follows:
  - ✓ request
  - ✓ use
  - ✓ release

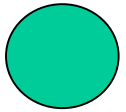
Resource-Allocation Graph

A set of vertices V and a set of edges E

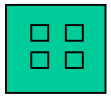
- V is partitioned into two types:
  - ✓ $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - ✓ $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Request edge
  - ✓ directed edge $P_i \rightarrow R_j$
- Assignment edge
  -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

Resource-Allocation Graph (Cont'd)

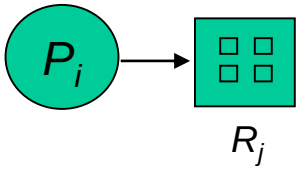
- Process



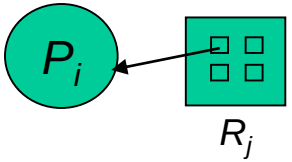
- Resource Type with 4 instances



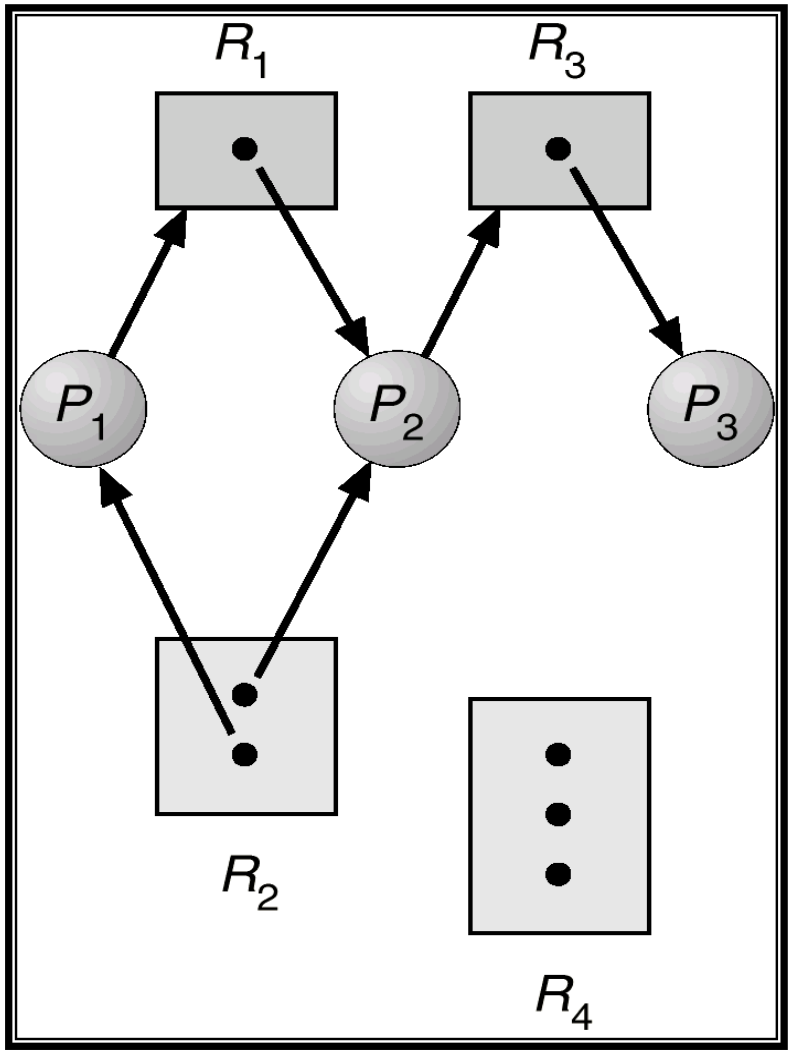
- P_i requests instance of R_j



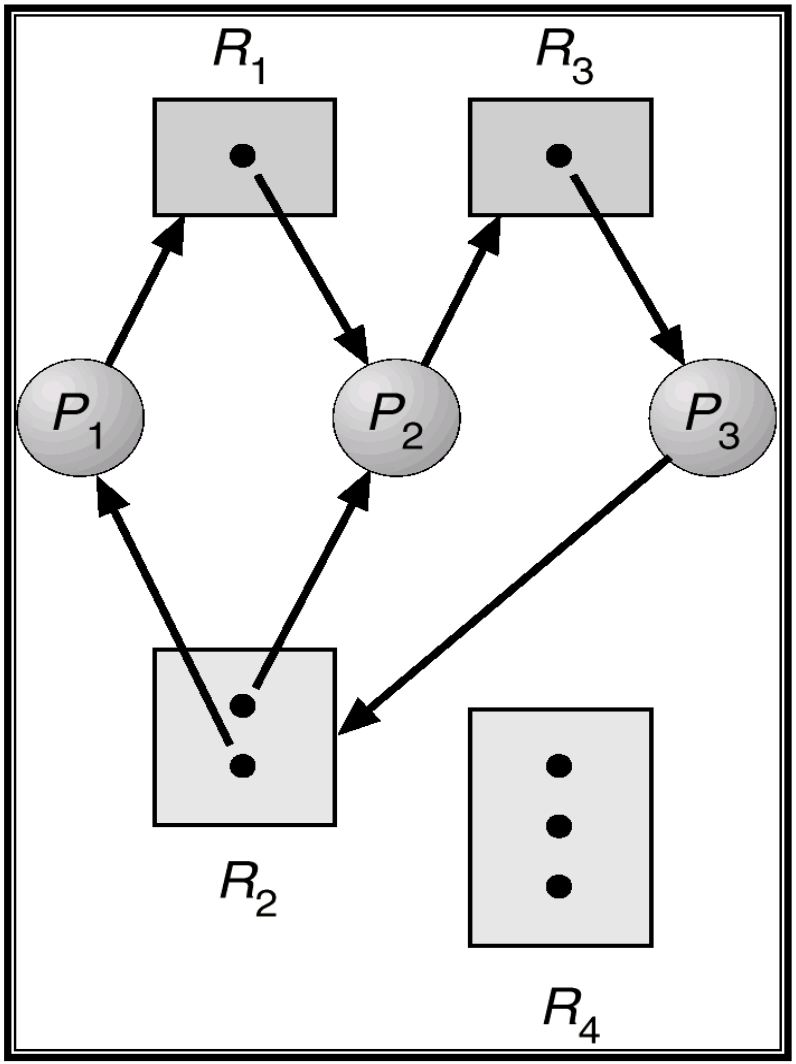
- P_i is holding an instance of R_j



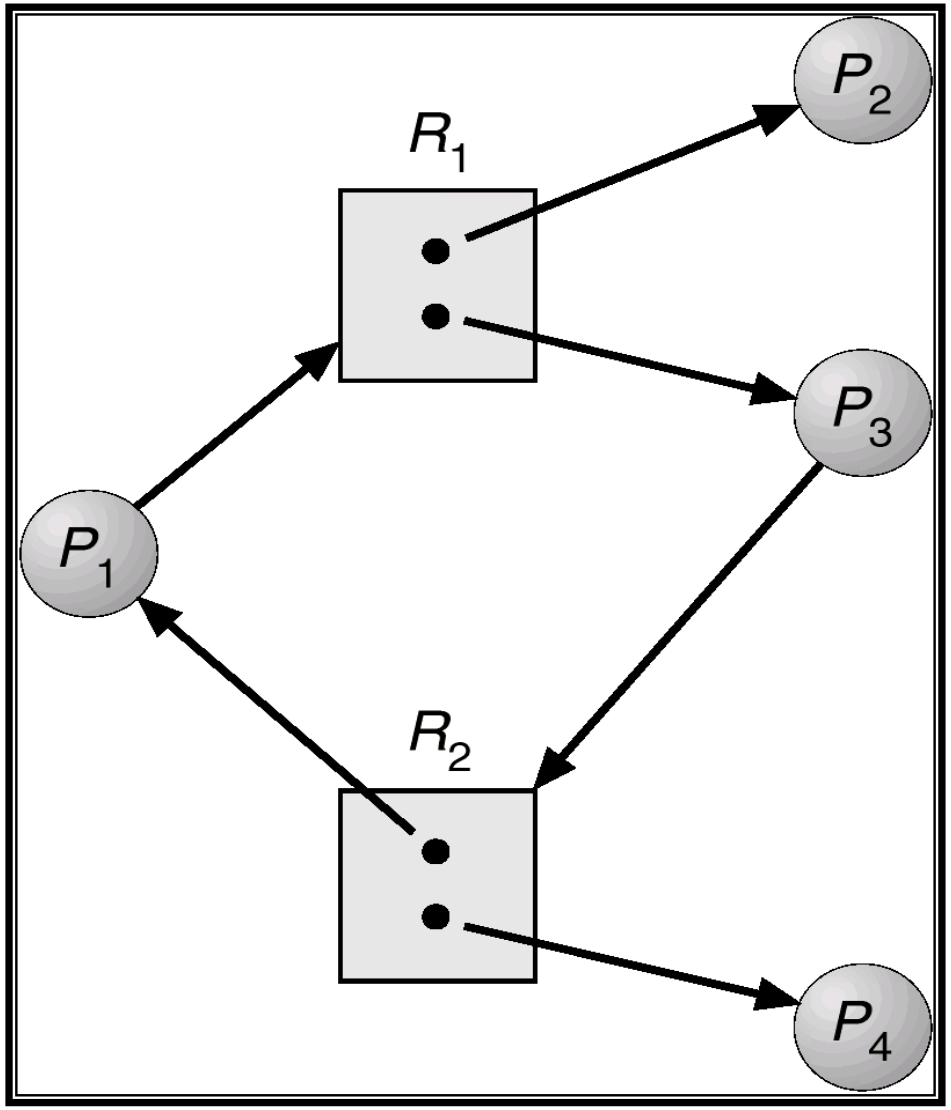
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles ⇒ 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

Deadlock Avoidance

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

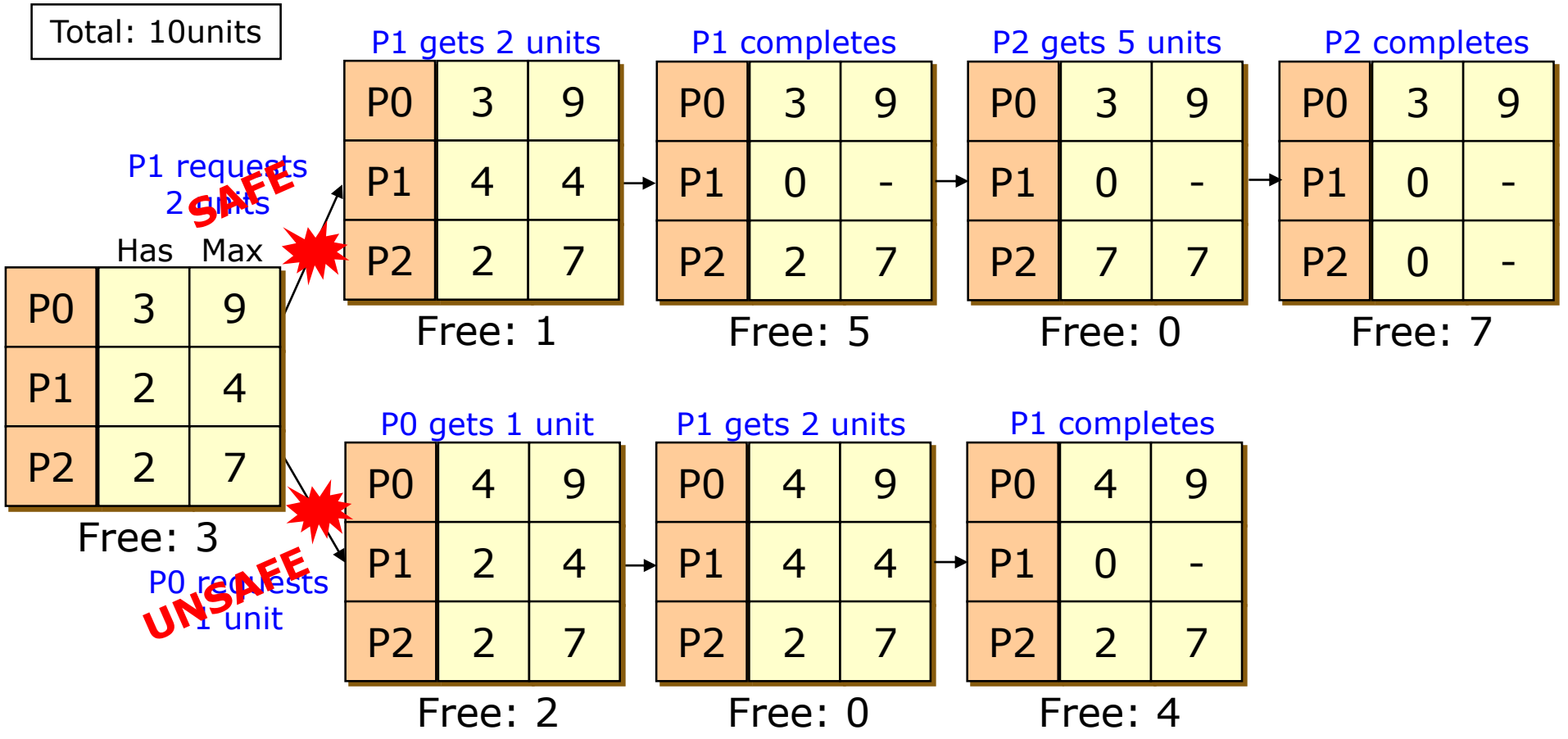
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* 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 $<P_1, P_2, \ldots, P_n>$ 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 vs. Unsafe State

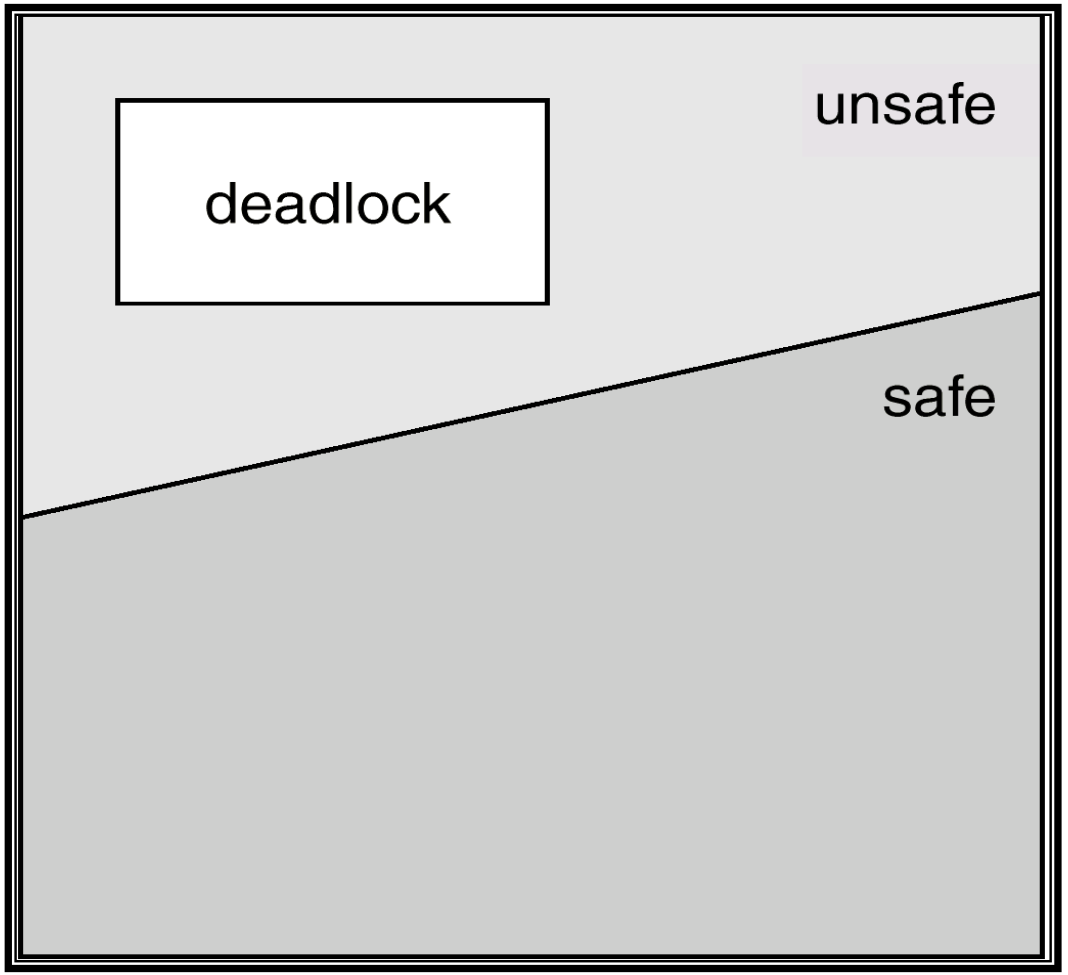
- Examples



Basic Facts

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

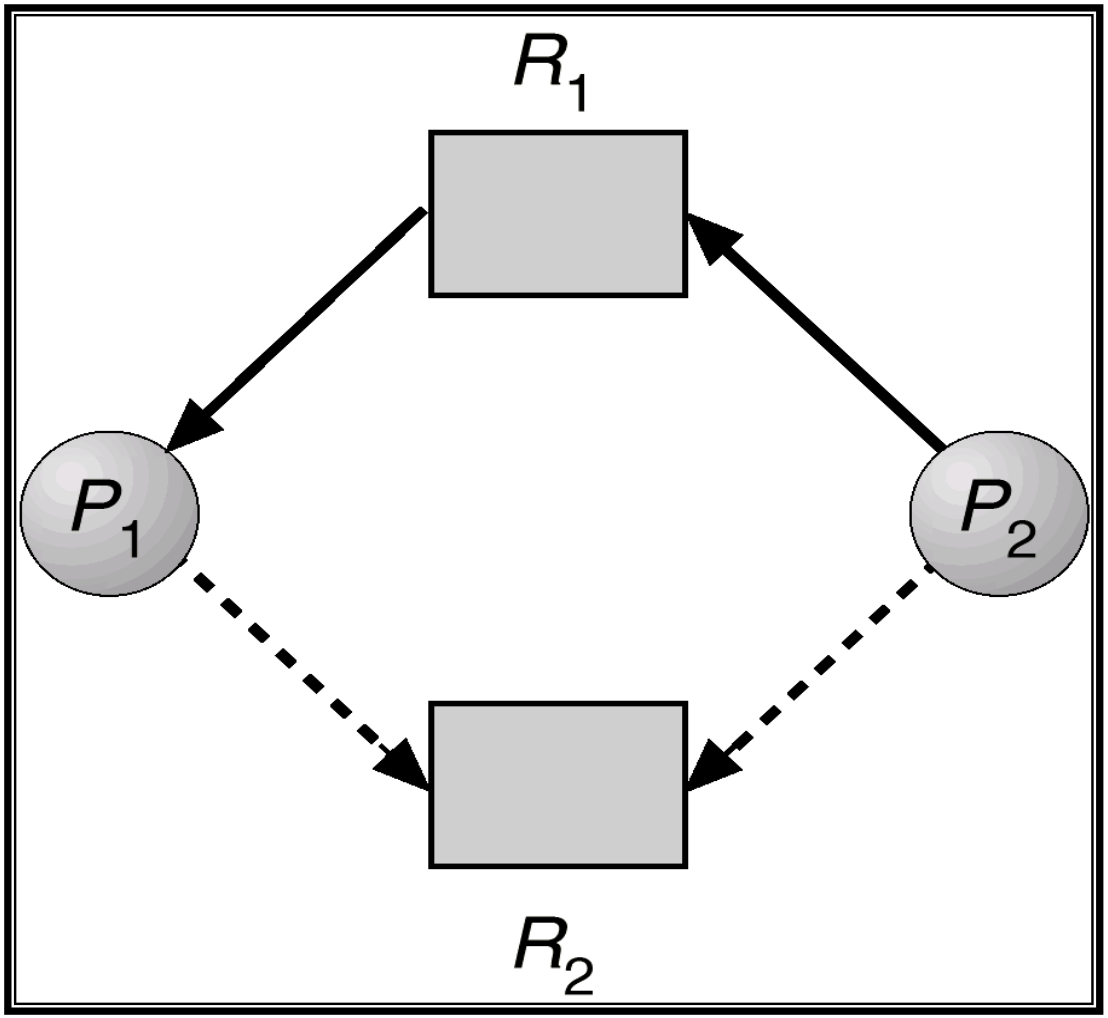
Safe, Unsafe , Deadlock State



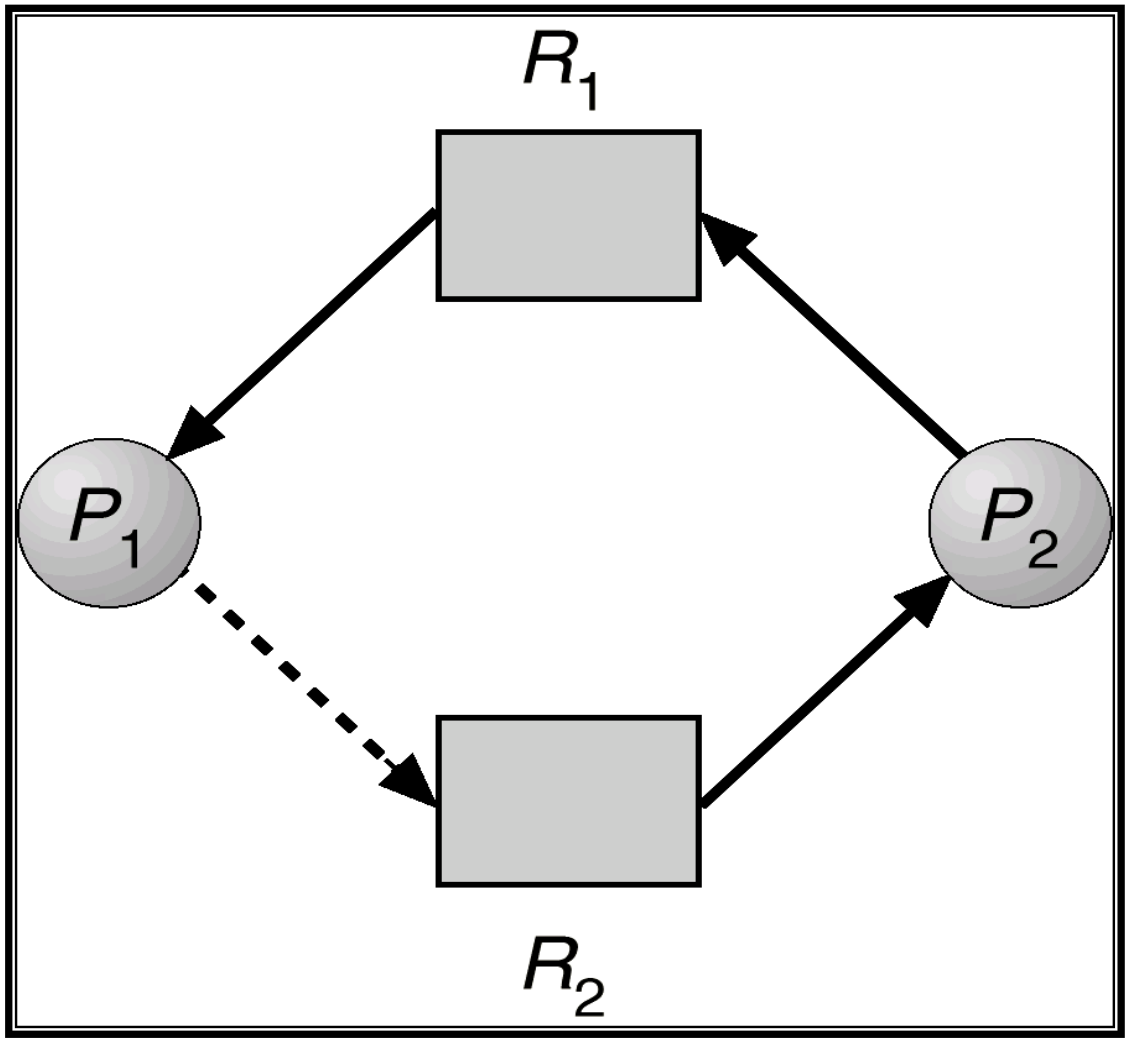
Resource-Allocation Graph Algorithm

- *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

Resource-Allocation Graph For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



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 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]

Safety Algorithm

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

   Work := *Available*

   Finish [*i*] := *false* for *i* = 1,2, …, *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

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 Process P_i

- $Request_i$: 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*

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'd)

- 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 (Redrawn)

Total Resources

A	B	C
10	5	7

Available Resources

A	B	C
3	3	2

Allocated Resources

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max Resources

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Needed Resources

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Currently safe: <P1, P3, P4, P0, P2> is a safe sequence

Example P_1 Request (1,0,2) (Cont'd)

- 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?

Banker's Algorithm

- Notes
  - ✓ Safety checking algorithm requires $O(mn^2)$ operations, where
    - m is the number of resource types
    - n is the number of processes
  - ✓ Processes rarely know in advance what their maximum resource needs will be
  - ✓ The number of processes is not fixed, but dynamically varying as new users log in and out
  - ✓ Resources that were thought to be available can suddenly vanish
    - e.g., tape drives or disk drives
  - ✓ In practice, few, if any, existing systems use the banker's algorithm for avoiding deadlocks

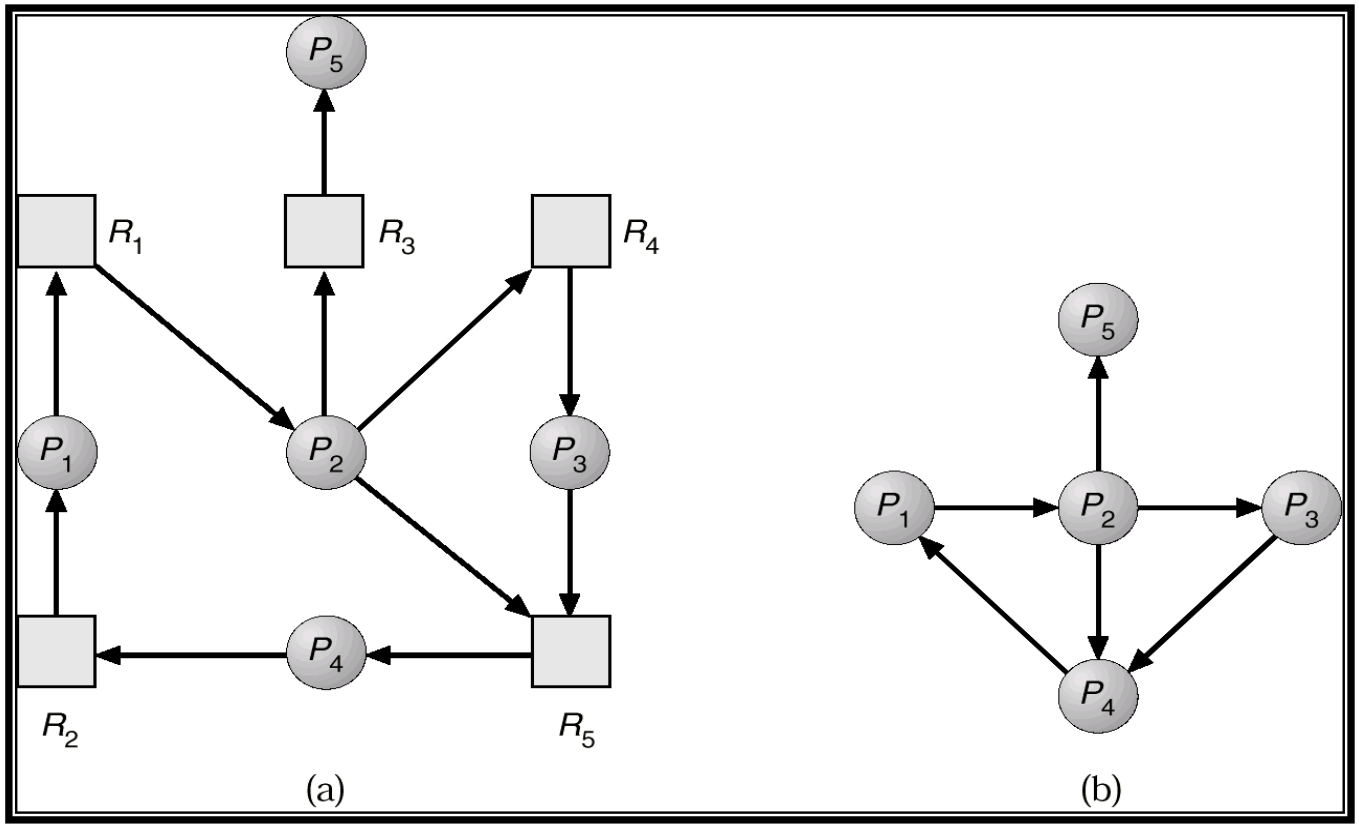
Deadlock Detection

- 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 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'd)

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 (Redrawn)

Total Resources

A	B	C
7	2	6

Available Resources

A	B	C
0	0	0

Allocated Resources

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2

Requested Resources

	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	0	0	2

What if this becomes 1?

Currently not in a deadlock: <P0, P2, P3, P1, P4> will work

Example (Cont'd)

- 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
- Starvation
  - ✓ Same process may always be picked as victim, include number of rollback in cost factor

Combined Approach to Deadlock Handling

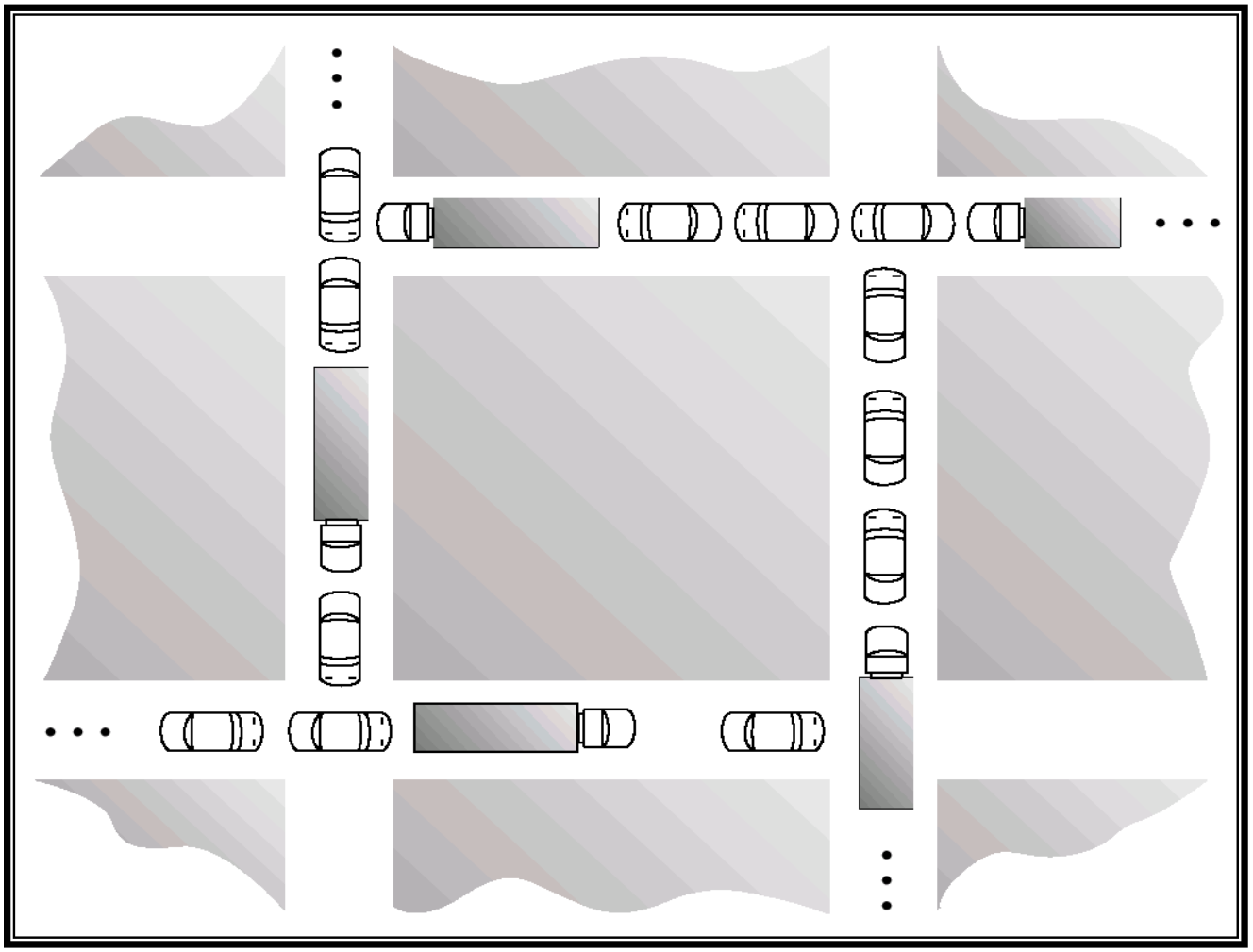
- Combine the three basic approaches
  - ✓ Prevention
  - ✓ Avoidance
  - ✓ Detection

  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

Traffic Deadlock for Exercise 8.4



Deadlock Ignorance

- The Ostrich algorithm
  - ✓ Just put your head in the sand and pretend there is no problem at all
  - ✓ Reasonable if
    - Deadlocks occur very rarely
    - Cost of prevention is high
  - ✓ UNIX and Windows take this approach
  - ✓ It is a trade-off between
    - Convenience
    - Correctness

Revisited: Handling Deadlocks

- Deadlock prevention
  - ✓ Restrain how requests are made
  - ✓ Ensure that at least one necessary condition cannot hold
- Deadlock avoidance
  - ✓ Require additional information about how resources are to be requested
  - ✓ Decide to approve or disapprove requests on the fly
- Deadlock detection and recovery
  - ✓ Allow the system to enter a deadlock state and then recover
- Just ignore the problem altogether!

Revisited: Dining Philosopher

- A simple solution

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
   while (1)  {
      think ();
      wait (chopstick[i]);
      wait (chopstick[(i+1) % N];
      eat ();
      signal (chopstick[i]);
      signal (chopstick[(i+1) % N];
   }
}
```

$\Rightarrow$ Problem: causes deadlock

Revisited: Dining Philosopher (Cont'd)

- Deadlock-free version: starvation?

```
#define N        5
#define L(i)     ((i+N-1)%N)
#define R(i)     ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
 }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```


Summary

- Deadlock concept
  - ✓ In a set of blocked processes each process is holding a resource and waiting to acquire a resource held by another process
  - ✓ Necessary conditions for deadlock
    - Mutual exclusion
    - Hold and wait
    - No preemption
    - Circular wait
- Mechanisms to handle deadlocks
  - ✓ Deadlock prevention
    - Ensure that at least one necessary condition cannot hold
  - ✓ Deadlock avoidance
    - Decide to approve or disapprove requests on the fly
  - ✓ Deadlock detection and recovery
    - Allow the system to enter a deadlock state and then recover
  - ✓ Deadlock ignorance