Chapter 6
Deadlocks

Resources

- A **resource** is anything that can be used by a single process at any instant of time, and that must be acquired, used, and released over the course of time.
- Resource types $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices, record in a database…
- Each resource type R_i has W_i instances.
- Preemptable resources
  - can be taken away from a process with no ill effects (e.g. CPU, memory)
- Nonpreemptable resources
  - will cause the process to fail if taken away (e.g. CD recorder)

Resources

- Each process utilizes a resource as follows:
  1. request the resource
  2. use the resource
  3. release the resource

- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code

Resources

- Processes need access to resources in reasonable order
- Suppose a process holds resource A (e.g. scanner) and requests resource B (e.g. CD recorder)
  - At the same time another process holds B and requests A
  - Both are blocked and remain so
- Deadlocks occur when …
  - processes are granted exclusive access to devices

Using Semaphore to Share Resource

```
1 ⇨ Process P();                    1 ⇨ Process Q();
2 ⇨ { A.Down();                     6 ⇨ { A.Down();
3 ⇨    B.Down();                           B.Down();
         use both resource                   use both resource
4 ⇨    B.Up();                             B.Up();
5 ⇨    A.Up(); }                           A.Up(); }
```

1 External Semaphore A(1), B(1);

2 External Semaphore A(0), B(1);

3 External Semaphore A(0), B(0);

4 External Semaphore A(0), B(1);

5 External Semaphore A(1), B(1);

But Deadlock can Happen!

```
1 ⇨ Process P();
2 ⇨ { A.Down();
        B.Down();
          use b
        B.Up()
        A.Up()
```

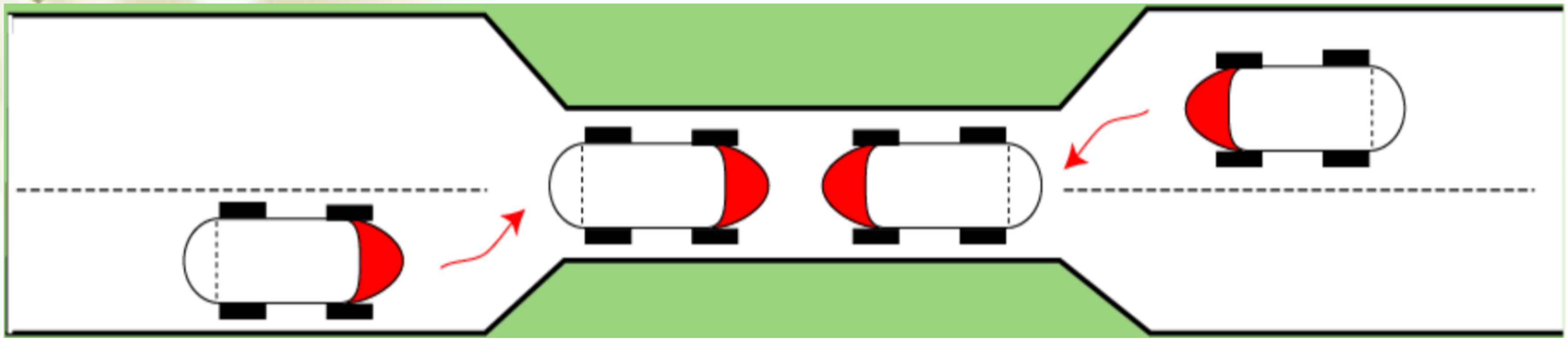
```
1 ⇨ Process Q();
3 ⇨ { B.Down();
        A.Down();
          use both resources
        A.Up();
        B.Up(); }
```

1 External Semaphore A(1), B(1);

2 External Semaphore A(0), B(1);

3 External Semaphore A(0), B(0);

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- Deadlock can occur if two cars enter the bridge from opposite sides
  - 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.

Introduction to Deadlocks

- Formal definition :
  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.
- Usually the event is release of a currently held resource
- None of the processes can …
  - run
  - release resources
  - be awakened

Four Conditions for Deadlock

Deadlock can arise if four conditions hold simultaneously.

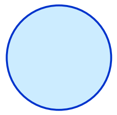
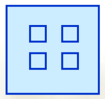
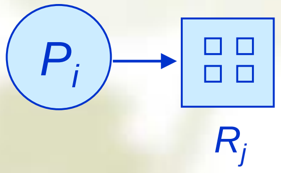
1. Mutual exclusion condition
   - Each resource assigned to 1 process or is available
2. Hold and wait condition
   - Process holding resources can request additional
3. No preemption condition
   - Previously granted resources cannot forcibly taken away
   - They must be explicitly released by the process holding them
4. Circular wait condition
   - Must be a circular chain of 2 or more processes
   - Each is waiting for resource held by next member of the chain

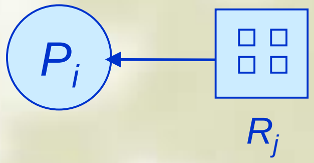
Deadlock Modeling

Modeled with directed graphs (resource-allocation graph)

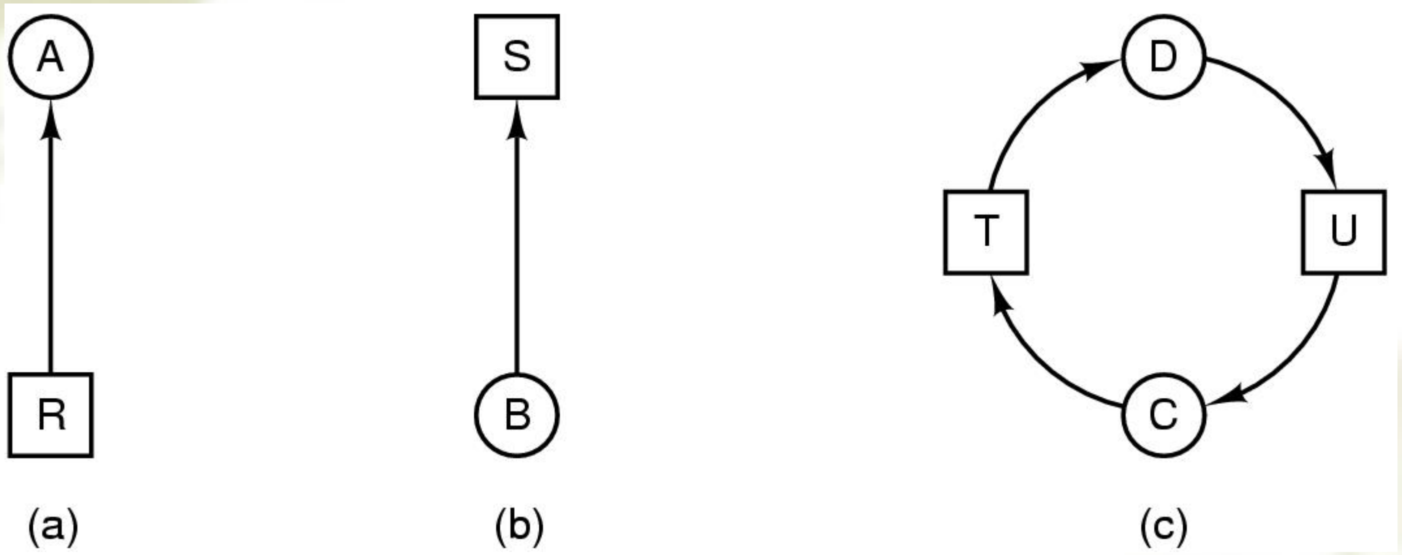
- A set of vertices *V* and a set of edges *E*.
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
  - request edge – directed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

Resource-Allocation Graph

- Process 
- Resource Type with 4 instances 
- P_i requests an instance of R_j 

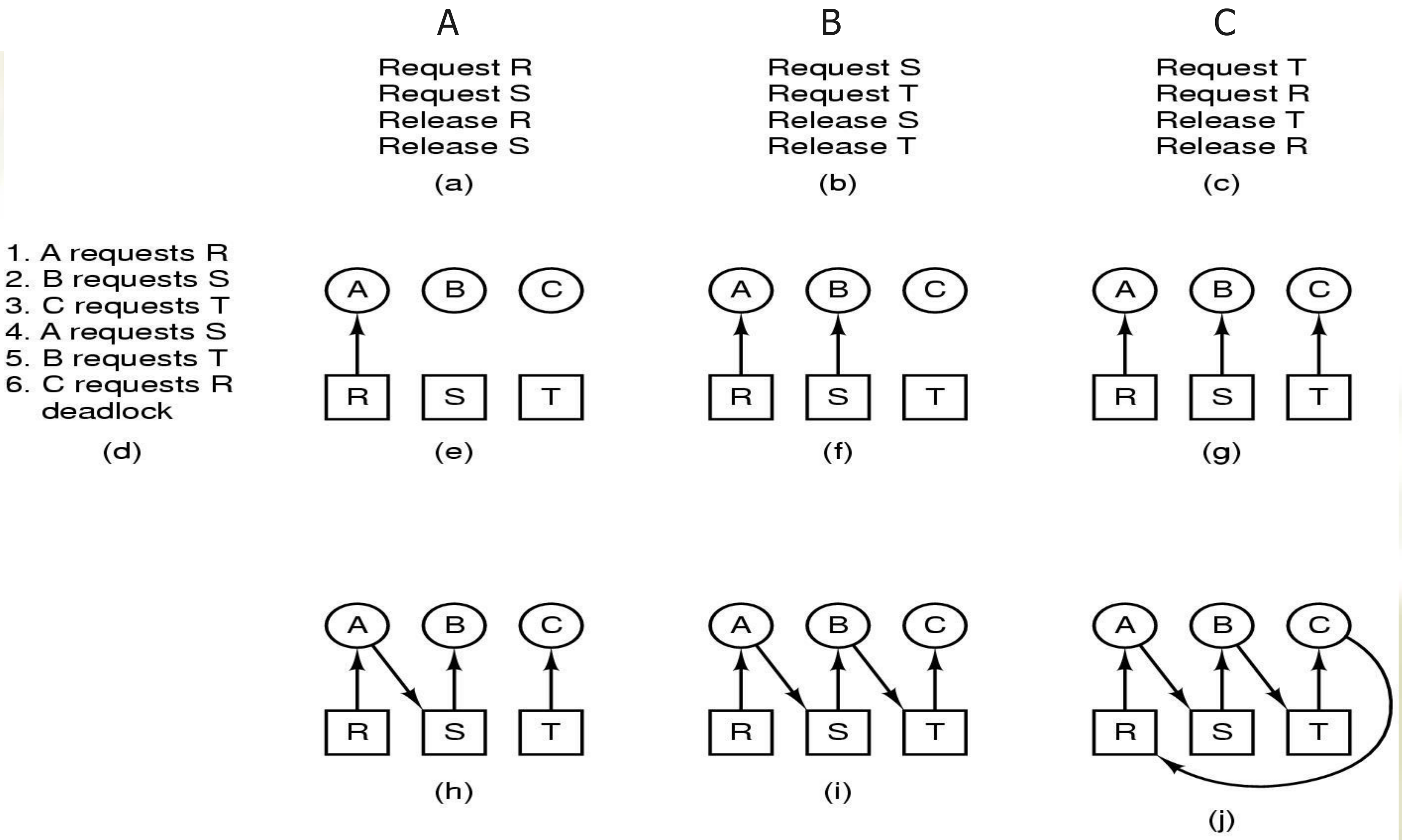
- P_i is holding an instance of R_j 

Resource-Allocation Graph



a) Resource R assigned to process A
b) Process B is requesting/waiting for resource S
c) Process C and D are in deadlock over resources T and U (C-T-D-U-C)

Example of Deadlock Modeling



A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

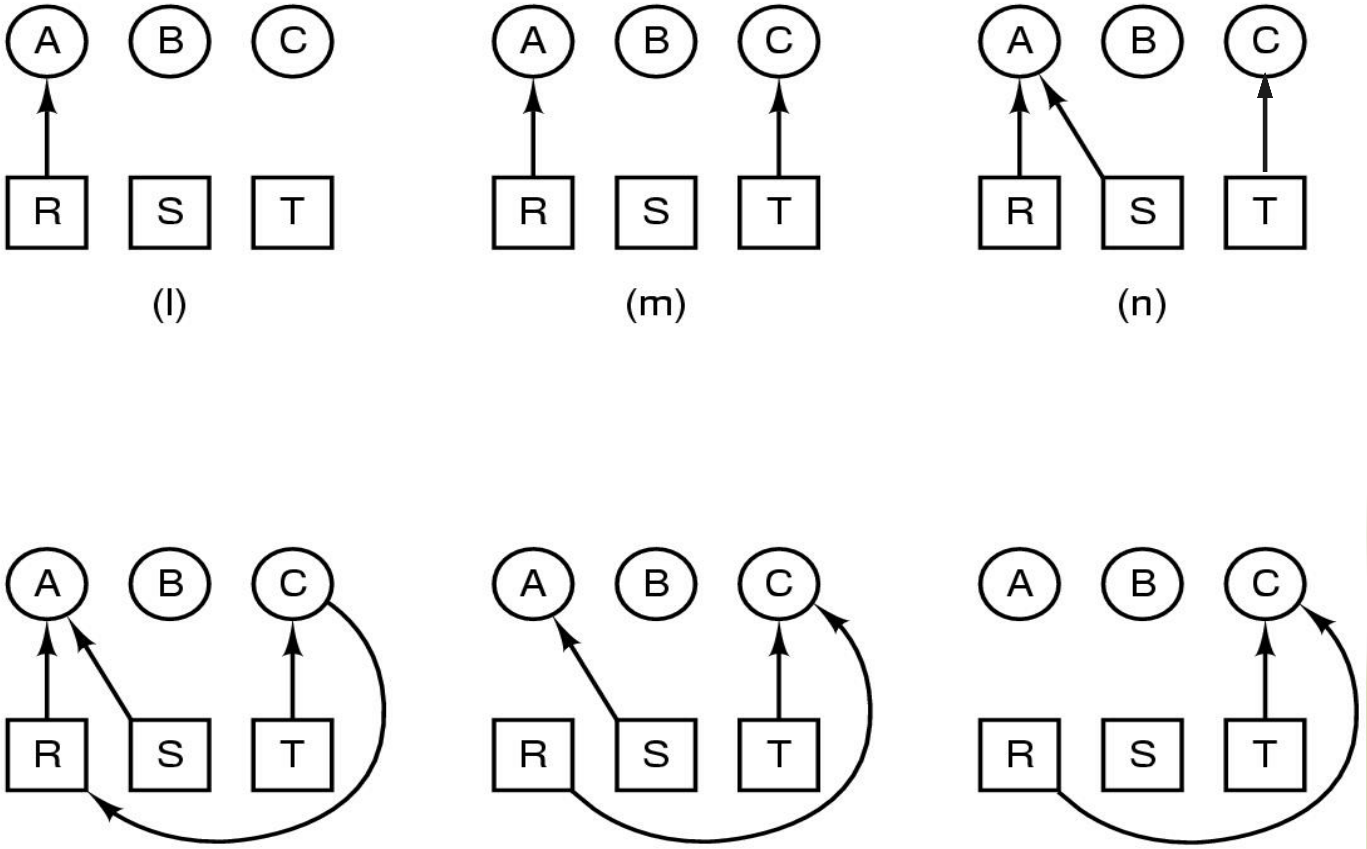
(d)

How deadlock occurs

Example of Deadlock Modeling

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

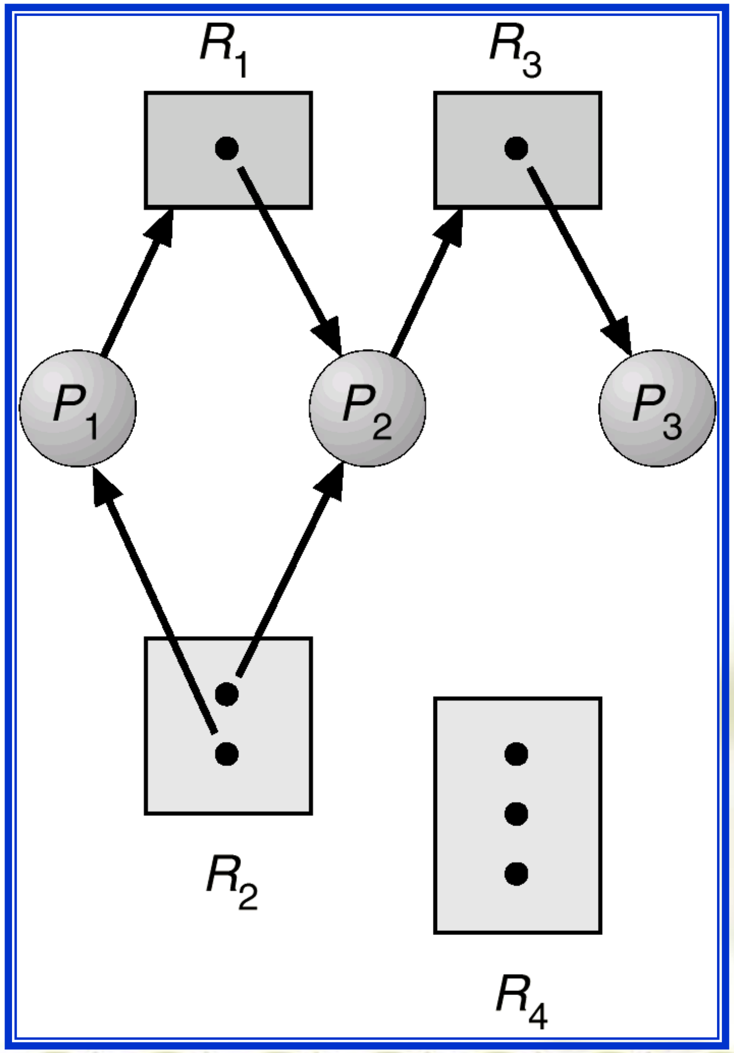
(k)



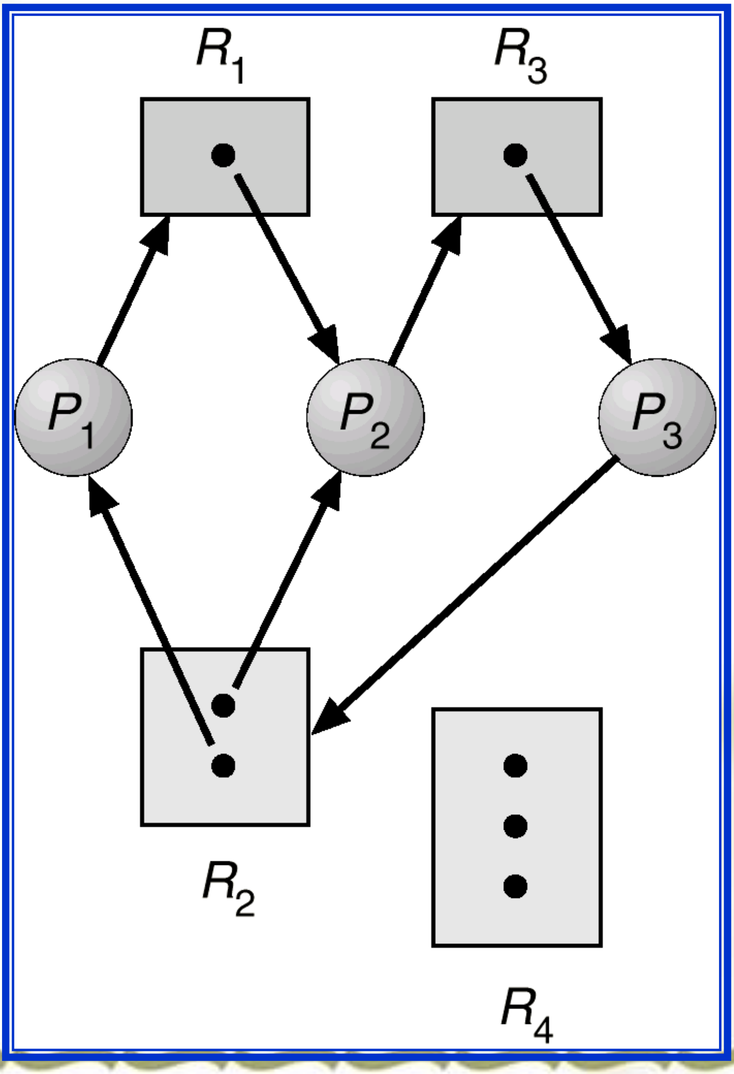
(o) (p) (q)

How deadlock can be avoided

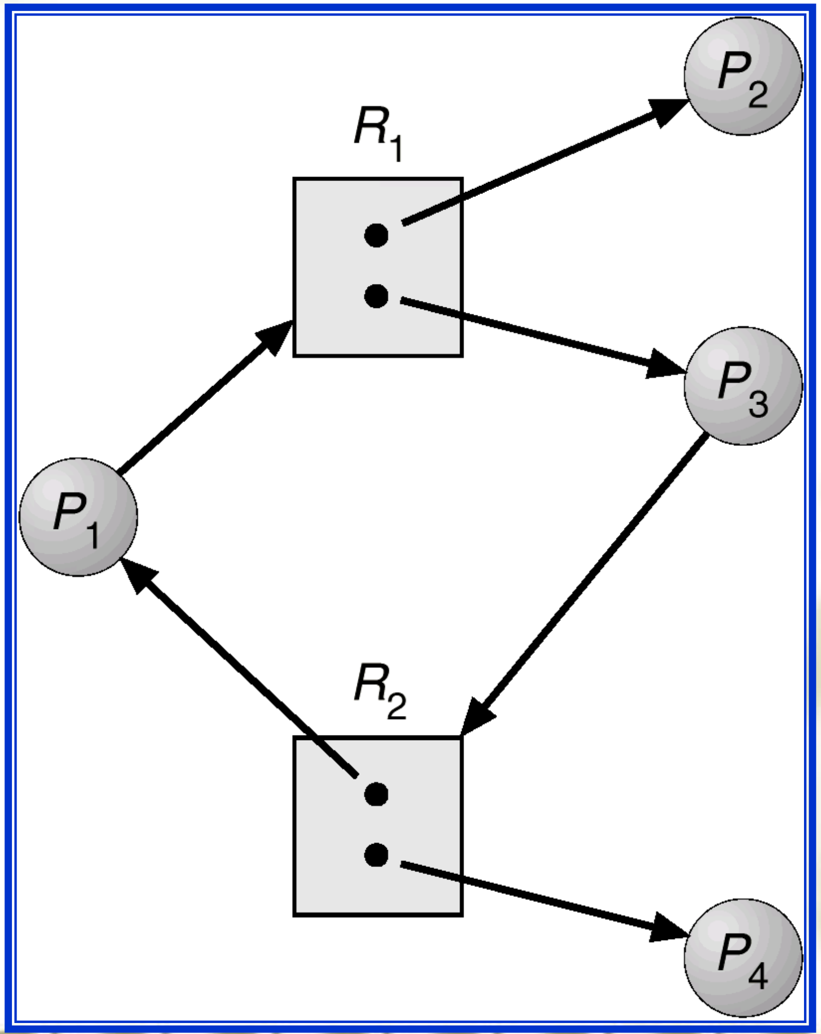
Another Example of a Resource Allocation Graph



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

1. Ignore the problem and pretend that deadlocks never occur in the system
2. Allow the system to enter a deadlock state and then recover.
   - Detection and recovery
3. Ensure that the system will never enter a deadlock state
   - Dynamic avoidance

     (careful resource allocation)
   - Prevention

     (negating one of the four necessary conditions)

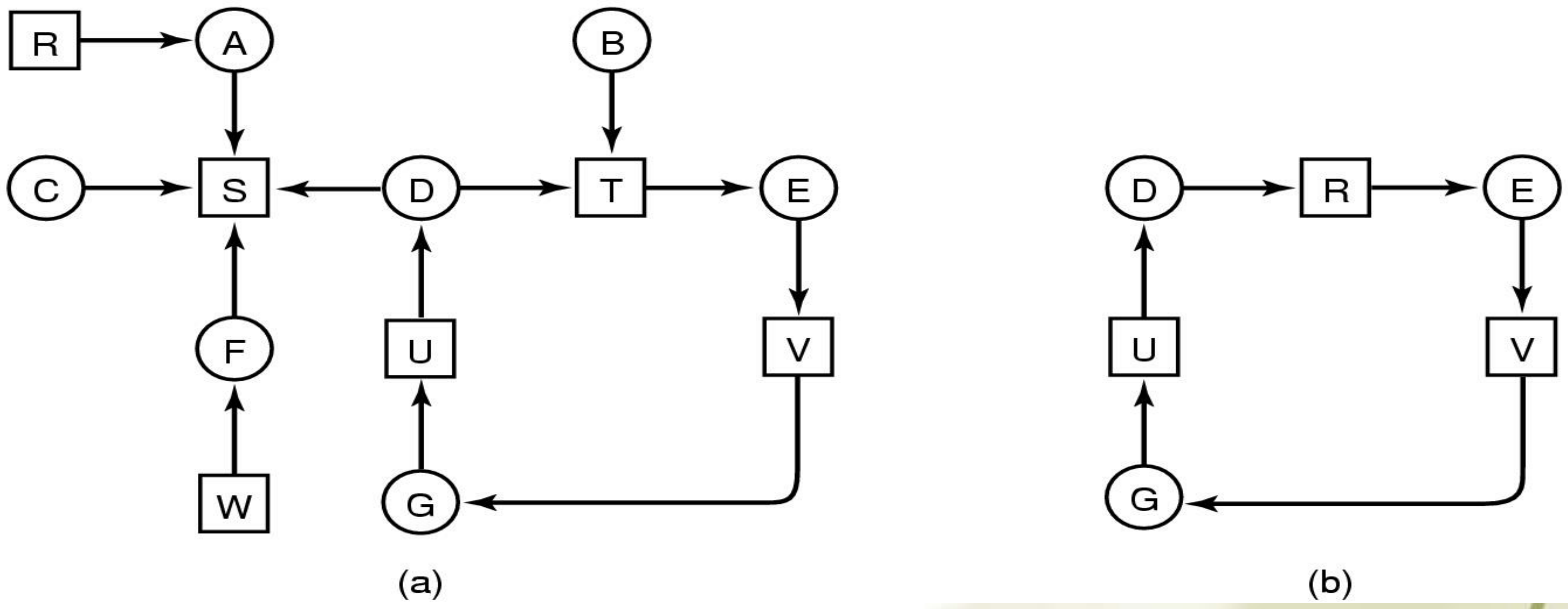
(1) The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
  - convenience
  - correctness

(2) Deadlock Detection

- Allow system to enter deadlock state
- Run the Deadlock Detection algorithm periodically
- After detecting deadlock, run a Recovery algorithm

Detection with One Resource of Each Type



- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

An algorithm for detecting cycles in directed graph

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

- A data structure, L, a list of nodes.
- Arcs will be marked to indicate that they have already been inspected, to prevent repeated inspections.

For each node, *N*, in the graph, performing the following 5 steps with N as the starting node.

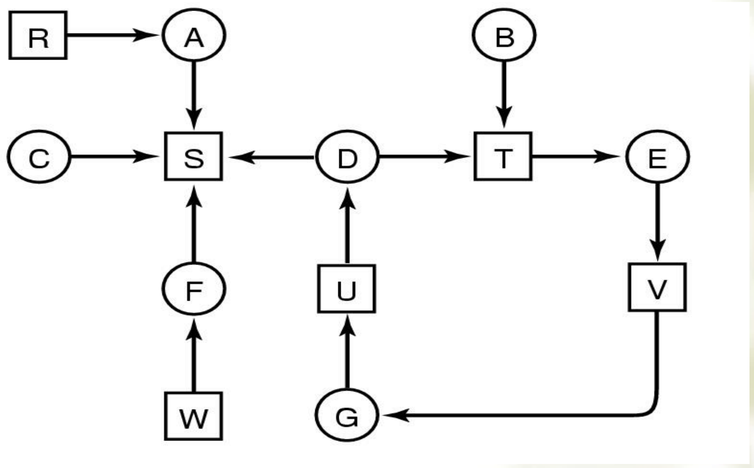
An algorithm for detecting cycles in directed graph

1. Initialize L to the empty list, and designate all the arcs as unmarked.
2. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
3. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 4; if not, go to step 5.
4. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 2.
5. We have now reached a dead end. Remove it and go back to the previous node, make it the current node, and go to step 3. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates.

Example1

Start at R and initialize L to the empty list.

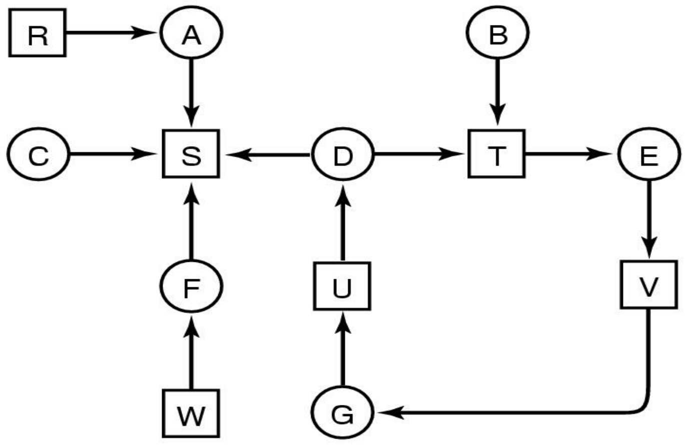
1. From R we go to S, giving L={R, A, S}. S has no outgoing arcs (a dead end).
2. Go backtrack to A.
3. A has no unmarked outgoing arcs, go backtrack to R.
4. R is the initial node, so no cycles.



Example2

Start at B and initialize L to the empty list.

1. From B, we follow outgoing arcs until we get to D, giving L={B, T, E, V, G, U, D}.
2. Pick S randomly and we come to a dead end, then backtrack to D.
3. Now pick T and update L to be {B, T, E, V, G, U, D, T}. T appears two times in L, we discover cycle.



Detection with Multiple Resources of Each Type

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Data structures needed by deadlock detection algorithm

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

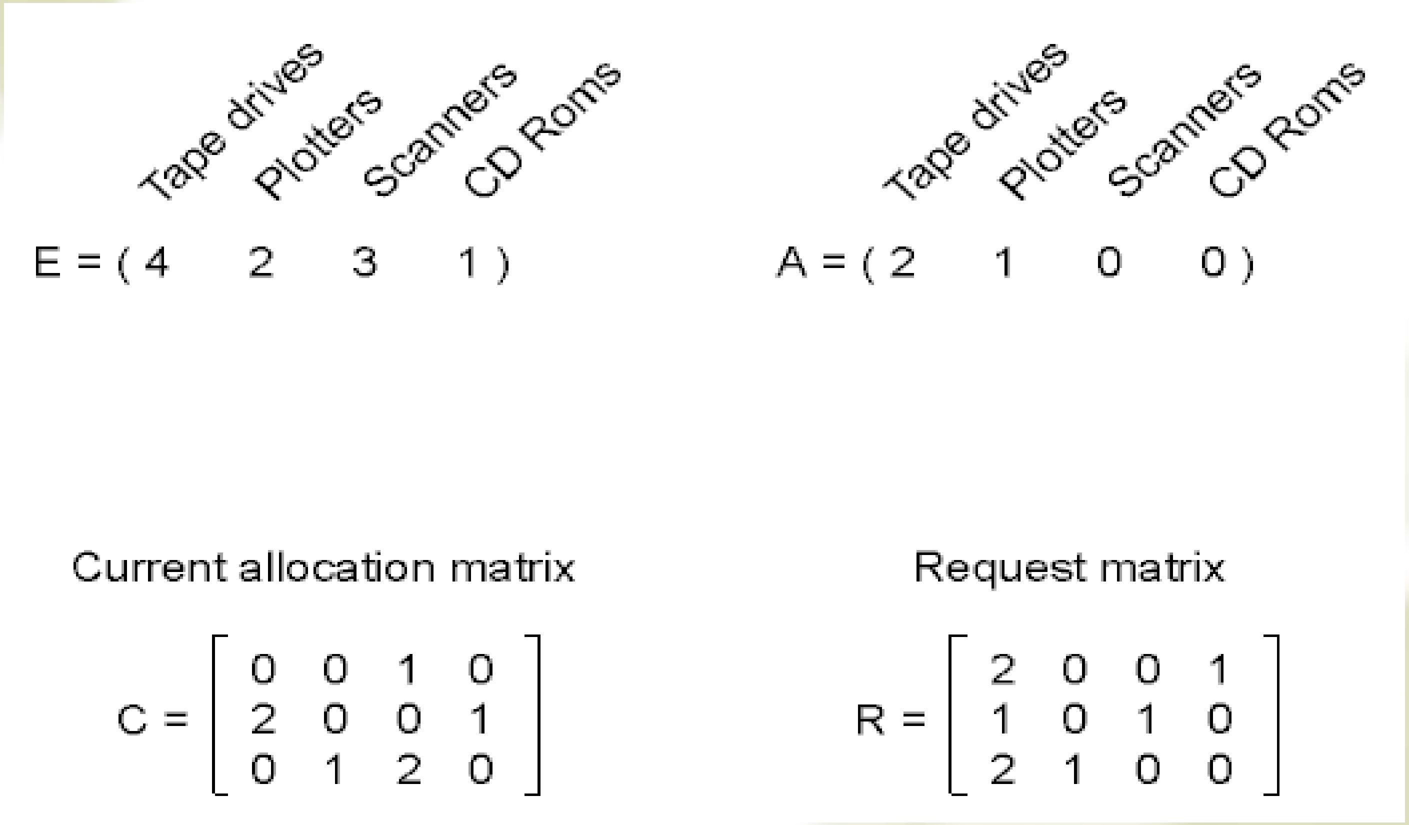
Deadlock Detection Algorithm

- Based on comparing vectors.

  (A<B hold if and only if $A_i < B_i$ for 1<=i<=m)
- Each process is initially unmarked.

1. Look for an unmark process P_i, for which the i-th row of R is less than or equal to A. (P_i's request can be satisfied)
2. If such a process is found, add the i-th row of C to A, mark the process and go back to step 1. (When P_i completes, its resources will become available).
3. If no such process exist, the algorithm terminates. The unmarked process, if any, are deadlock.

Detection with Multiple Resources of Each Type (Example)



$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (2 & 1 & 0 & 0) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

An example for the deadlock detection algorithm

Recovery from Deadlock

- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
    - write the state of a process to a file
      - the memory image, the resource state, ...
  - use this saved state
  - restart the process if it is found deadlocked

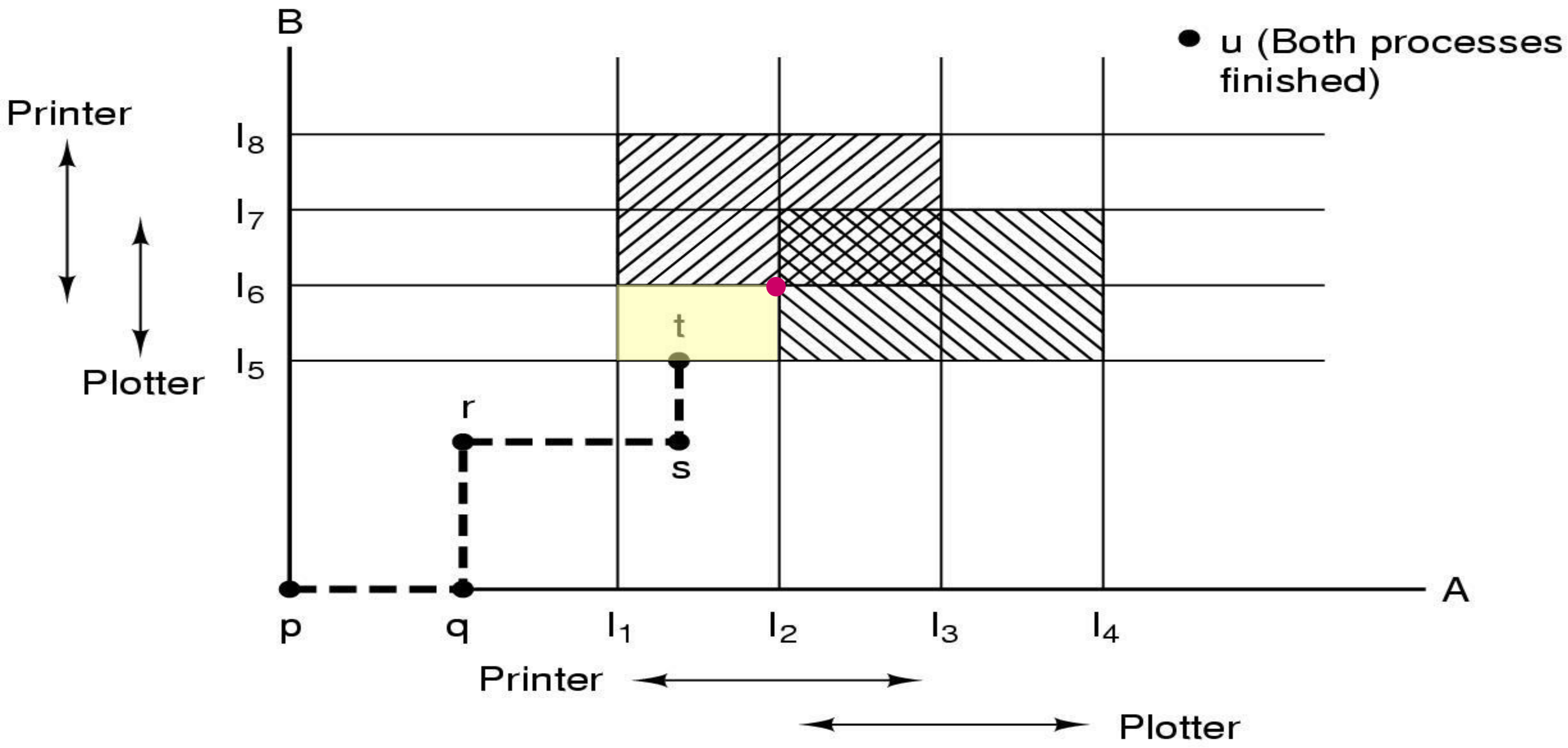
Recovery from Deadlock

- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle, then the other processes get its resources
  - kill process that can be rerun from the beginning with no ill effects

(3) Deadlock Avoidance

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- 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 are based on the safe states.

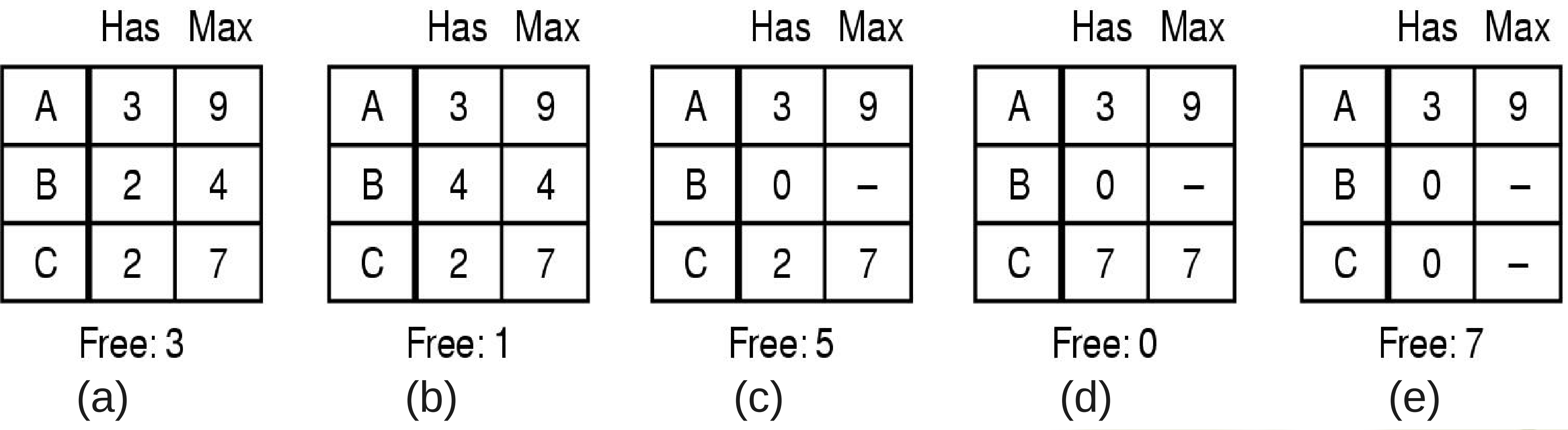


Two process resource trajectories

Safe and Unsafe States

- A state is safe if it is not deadlock and there is a way of satisfying all pending requests by running the process in some order.
- System is in safe state if there exists a safe sequence of all processes.
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

Safe and Unsafe States



(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(c)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(d)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(e)

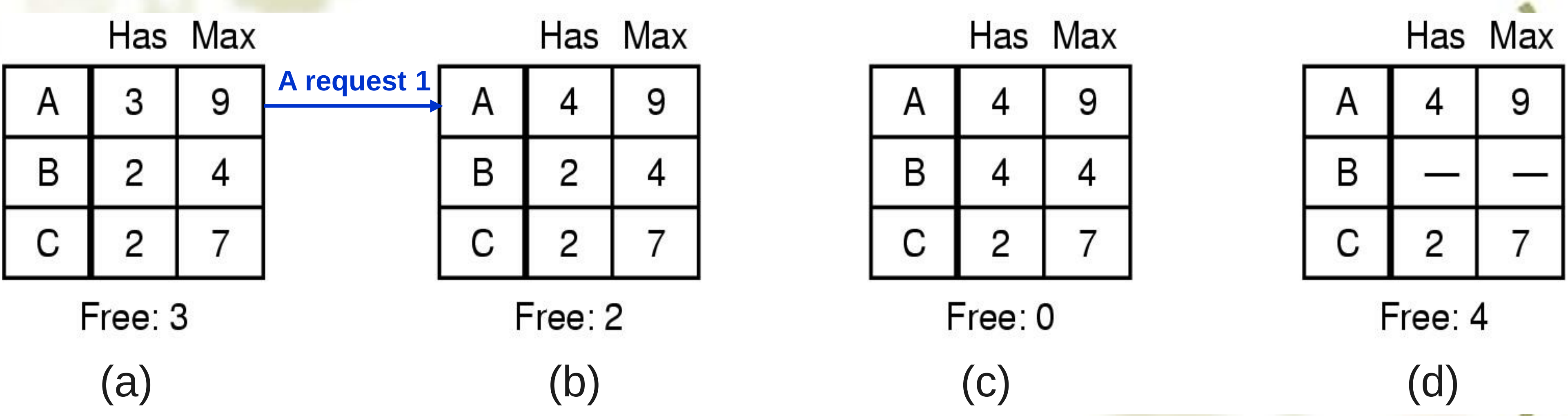
	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

Demonstration that the state in (a) is safe

(there exists a safe sequence BCA)

Safe and Unsafe States



A request 1

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

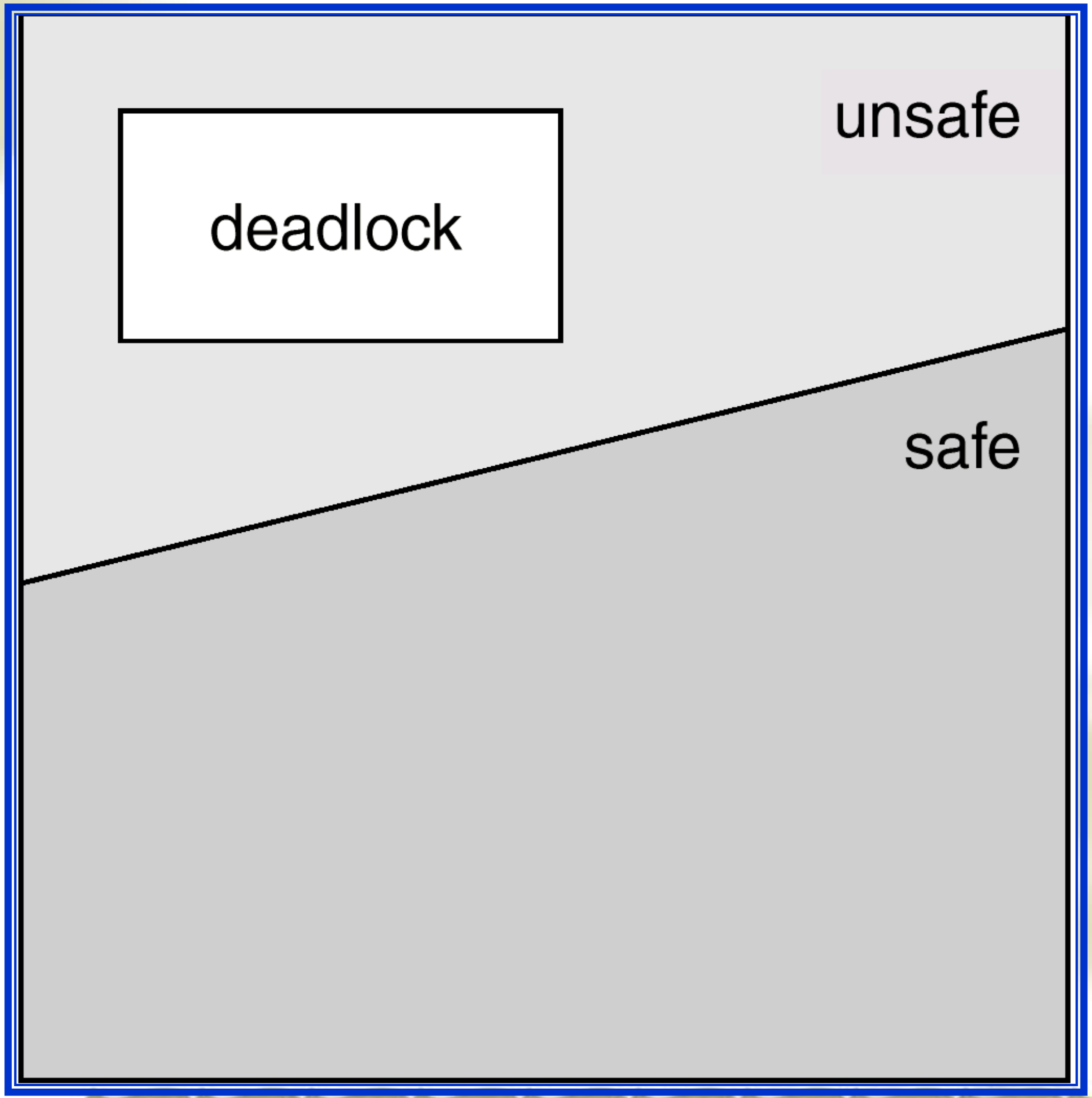
(d)

Demonstration that the sate in b is not safe

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 spaces



Deadlock-avoidance algorithm

- Banker Algorithm (Dijkstra, 1965)
  - Each customer tells banker the maximum number of resources it needs
  - Customer borrows resources from banker
  - Customer returns resources to banker
  - Customer eventually pays back loan
  - Banker only lends resources if the system will be in a *safe* state after the loan
- *Safe state* - there is a lending sequence such that all customers can take out a loan
- *Unsafe state* - there is a possibility of deadlock

The Banker's Algorithm for a Single Resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10
(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2
(b)

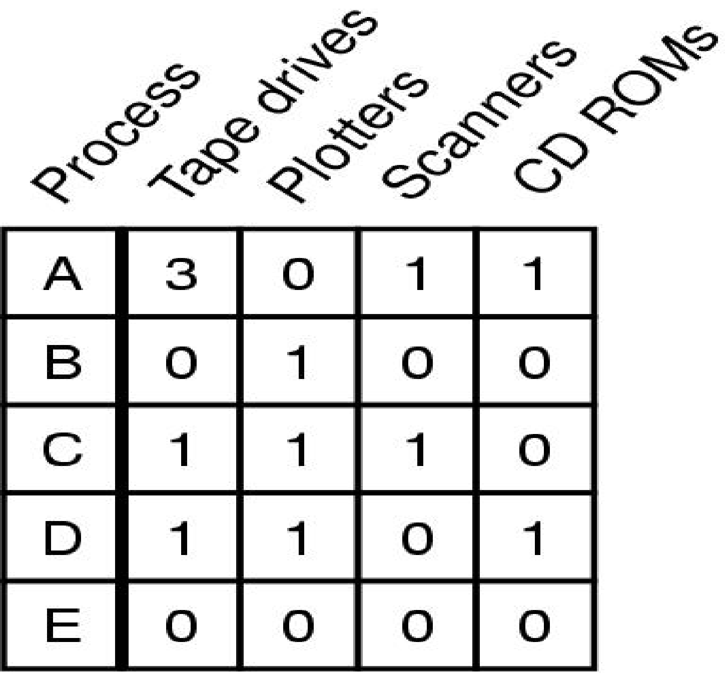
B request 1

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1
(c)

- Three resource allocation states
  - (a) safe
  - (b) safe
  - (c) unsafe

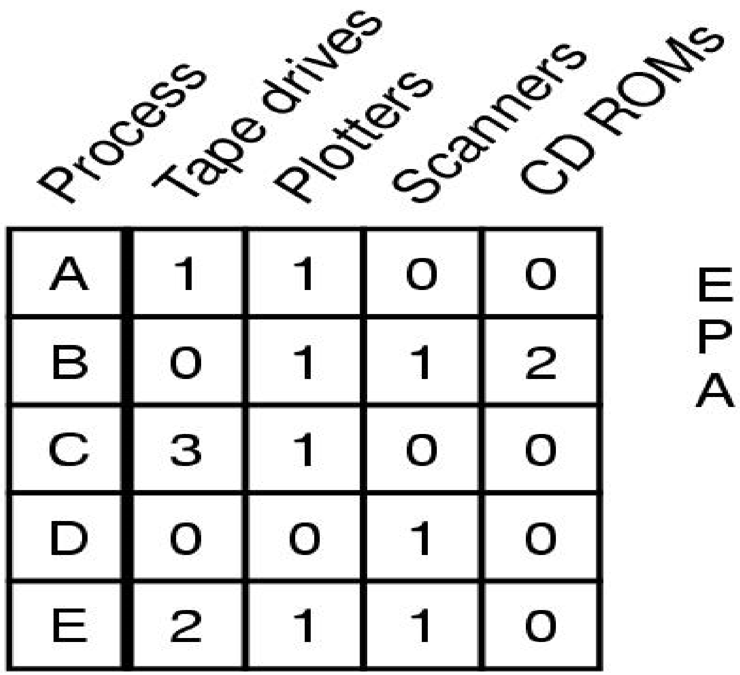
Banker's Algorithm for Multiple Resources



Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

C



Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

R

E = (6342)
P = (5322)
A = (1020)

Example of banker's algorithm with multiple resources

Banker's Algorithm

Check to see if a state is safe:

1. Some look for a new row in R which is smaller than A. If no such row exists the system will eventually deadlock ==> not safe.
2. If such a row exists, the process may finish. Mark that process (row) as terminate and add all of its resources to A.
3. Repeat Steps 1 and 2 until all rows are marked == > safe state

or not marked == > not safe.

How to Compute Safety

Given:

n kinds of resources

p processes

Set P of processes

struct {resource needs[n], owns[n]} ToDo[p]

resource available[n]

```
while there exists a p in P such that
    for all i (ToDo[p].needs[i] < available[i] )
      { do for all i
            available[i] += ToDo[p].owns[i];
            P = P - p;
      }
```

If P is empty then system is safe.

Is Allocation (1 0 2) to P1 Safe?

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	2	0	0	3	2	2	1	2	2	3	3	2
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

And Run Safety Test: Group Discussion

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

If P1 requests max resources, can complete

Allocate to P1, Then

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	2	2	3	2	2	0	0	0	2	1	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Release - P1 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	5	3	2
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Now P3 can acquire max resources and release

Release – P3 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	7	4	3
P2	3	0	0	9	0	2	6	0	2			
P3	0	0	0	2	2	2	2	2	2			
P4	0	0	2	4	3	3	4	3	1			

Now P4 can acquire max resources and release

Release - P4 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	7	4	5
P2	3	0	0	9	0	2	6	0	2			
P3	0	0	0	2	2	2	2	2	2			
P4	0	0	0	4	3	3	4	3	3			

Now P2 can acquire max resources and release

Release - P2 Finishes

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	10	4	5
P2	0	0	0	9	0	2	9	0	2			
P3	0	0	0	2	2	2	2	2	2			
P4	0	0	0	4	3	3	4	3	3			

Now P0 can acquire max resources and release

So P1 Allocation (1 0 2) Is Safe

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

There exists a safe sequence: P1, P3, P4, P2, P0

Is allocation (0 2 0) to P0 Safe?

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Try to Allocate 2 B to P0

Run Safety Test

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	7	2	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	1	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

No Processes may get max resources and release

So Unsafe State - Do Not Enter

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Return to Safe State and do not allocate resource

P0 Suspended Pending Request

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	3	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

When enough resources become available, P0 can awake

Results

- P0’s request for 2 Bs cannot be granted because that would prevent any other process from completing if they need their maximum claim.

Just Because It’s Unsafe---

- P0 could have been allocated 2 Bs and a deadlock might not have occurred if:
    - P1 say didn’t use its maximum resources but finished using the resources it had

Allocate to P0(0 2 0)

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	7	2	3	10	5	5
P1	3	0	2	3	2	2	0	2	0	2	1	0
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

P1 can finish its work using the resources it just had

If P1 Doesn’t Need Max---

Process	Alloc			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	7	2	3	10	5	5
P1	0	0	0	3	2	2	3	2	2	5	1	2
P2	3	0	0	9	0	2	6	0	2			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

Then P1 would have finished…

Banker Solution Issues

- Trade-off: The banker’s algorithm is conservative --- it reduces parallelism for safety sake
- Not very practicable
  - Processes rarely know advance what their maximum resource needs will be
  - The number of processes is not fixed
  - Resources that were thought to be available can suddenly vanish (tape drive can break)

(4) Deadlock Prevention

- Restrain the ways request can be made.
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait

Attacking the Mutual Exclusion Condition

- Some devices (such as printer) can be spooled
  - only the printer daemon uses printer resource
  - printer daemon doesn’t request other resources
  - thus deadlock for printer eliminated
- Spooling space is limited, so deadlock is still possible with this decision
  - Two processes each fill up half the available space and can’t continue, so there is deadlock on the disk

Attacking the Mutual Exclusion Condition

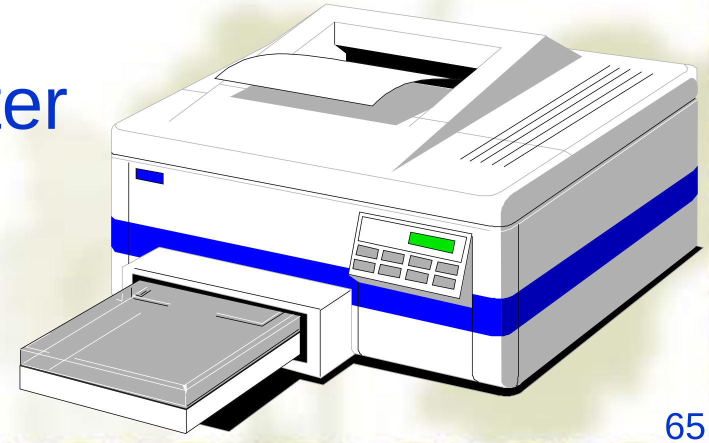
- Principle
  - avoid assigning resource if not absolutely necessary
  - as few processes as possible actually claim the resource
- Problem
  - Not all devices can be spooled

Attacking the Hold and Wait Condition

- Require processes to request all resources before starting
  - a process never has to wait for what it needs
- Problems
  - may not know required resources at start of run
  - also ties up resources other processes could be using
- Variation:
  - process requesting a resource must give up all the resources it currently holds, then request all immediately needed

Attacking the No Preemption Condition

- This condition can be violated by forceful preemption
- Difficult to achieve in practice
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



Attacking the No Preemption Condition

- Virtualization of Resources
  - Spooling printer output to the disk
  - Allow only the printer daemon access to the real printer.
- Problem
  - Not all resources can be virtualized

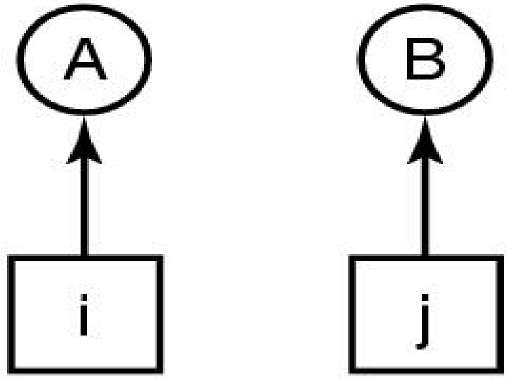
Attacking the Circular Wait Condition

- Approach 1: Request one resource at a time. Release the current resource when request the next one
- Approach 2: Global ordering of resources
  - Requests have to made in increasing order
- Approach 3: Variation of Approach 2
  - No process request a resource lower than what it is already holding.

Attacking the Circular Wait Condition

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

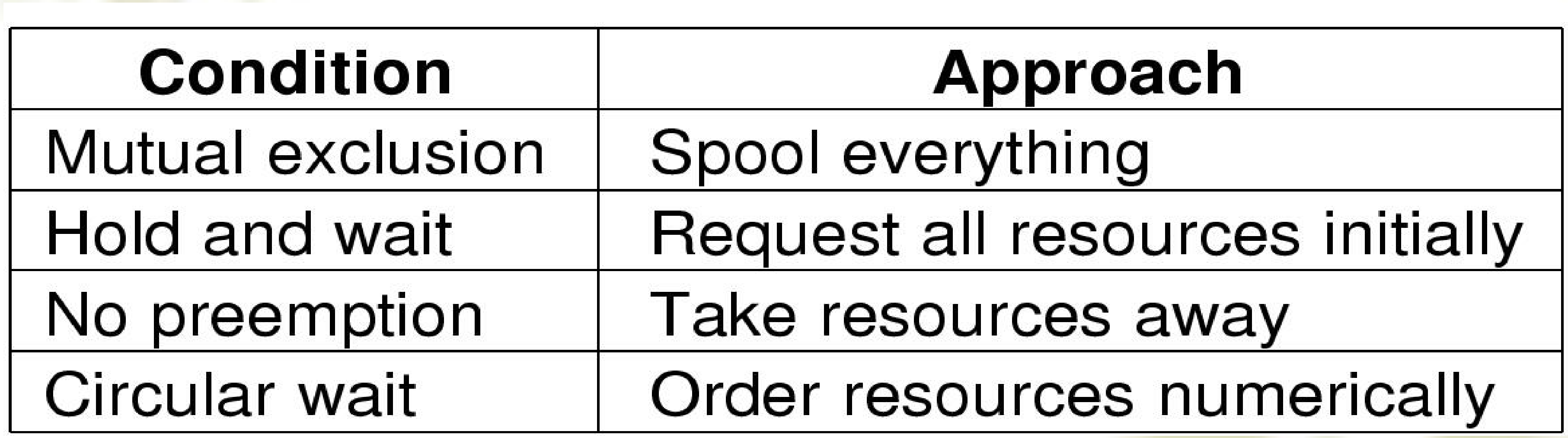
(a) Normally ordered resources



(b) A resource graph

- Problems
  - Finding a suitable numbering to satisfy everybody could be difficult/impossible
  - Increases burden on programmers to know the numbering

Summary of approaches to deadlock prevention



Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Other Issues

- Two-Phase Locking
- Communication Deadlocks
- Livelock
- Starvation

Two-Phase Locking

- Phase One
  - process tries to lock all records it needs, one at a time
  - if needed record found locked, start over
- If phase one succeeds, it starts second phase,
  - performing updates
  - releasing locks
- Note similarity to requesting all resources at once
- Algorithm works where programmer can arrange
  - program can be stopped, restarted

Communication Deadlocks

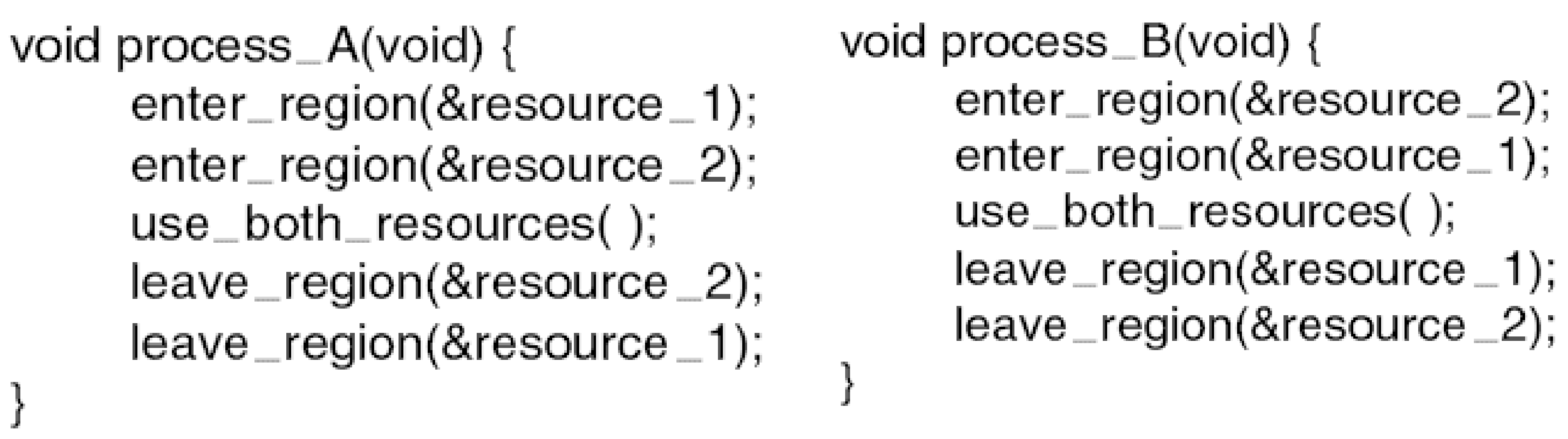
- Resource deadlocks are common, but not the only kind of deadlock.
- Communication deadlocks
  - A set of processes, each blocked waiting for an event only the other one can cause.
  - Example
    - Process A sends a request message to process B, and then blocks until B sends back a reply message.
    - If the request message gets lost, B is blocked waiting for a request asking it to do something.
- Solution: timeout

Livelock

- Deadlock: Processes are blocked
- Livelock: Processes run but make no progress
- Polling (busy waiting), used to enter a critical region or access a resource, can lead to livelock

Livelock

- Example



```
void process_A(void) {
        enter_region(&resource_1);
        enter_region(&resource_2);
        use_both_resources( );
        leave_region(&resource_2);
        leave_region(&resource_1);
}
```

```
void process_B(void) {
        enter_region(&resource_2);
        enter_region(&resource_1);
        use_both_resources( );
        leave_region(&resource_1);
        leave_region(&resource_2);
}
```

- When A acquires resource_1 and B acquires resource_2, no process can make further progress, and neither process blocks. →livelock

Starvation

- Algorithm to allocate a resource
  - may be to give to shortest job first
- Works great for multiple short jobs in a system
- May cause long job to be postponed indefinitely
  - even though not blocked
- Solution:
  - First-come, first-served policy

Summary

- In general, deadlock detection or avoidance is expensive
- Must evaluate cost of deadlock against detection or avoidance costs
- Deadlock avoidance and recovery may cause indefinite postponement
- Unix and Windows use Ostrich Algorithm

Homework

- 23、26、31、34、41