

Modern Operating Systems

Chapter 6 - Deadlock

Zhang Yang
Autumn 2022



Content of this lecture

- 6.1 Resources
- 6.2 Deadlock
- 6.3 Ostrich Algorithm
- 6.4 Deadlock Detection & Deadlock Recovery
- 6.5 Deadlock Avoidance
- 6.6 Deadlock Prevention
- 6.7 Other Issues
- Summary

Resources (1)

- A resource is anything that can be used by a single process at any instant of time.
- **Deadlocks** may occur when ...
 - Processes are granted exclusive access to devices, files, and so forth.
 - We refer to the objects granted as resources
- Resource Types R_1, R_2, \dots, R_m
 - CPU cycles, memory space, I/O devices, record in a database...
- Each resource type R_i has W_i instances.

Resources (2)

- Resources can be either:

- Reusable Resources

- Used by one process at a time and not depleted by that use.
 - Processes obtain resources that they later release for reuse by other processes.
 - Acquire → Use → Release
 - E.g., CPU, memory, disk space, I/O devices, files, databases, semaphores.

- Consumable Resources

- Created (produced) and destroyed (consumed) by a process.
 - Create → Acquire → Use
 - Resource ceases to exist after it has been used, so it is not released.
 - E.g., messages, buffers of information, interrupts.

Resources (3)

- Resources can also be either:

- Preemptible Resources

- Can be taken away from a process with no ill effects.
 - E.g., CPU, memory
 - No deadlock.

- Non-preemptible Resources

- One that cannot be taken away from its current owner without causing the computation to fail.
 - Will cause the process to fail if taken away.
 - E.g., Tape Drives, CD Recorder
 - Deadlock possible.

Resources (4)

- And resources can be either:
 - Shared among several processes
 - E.g., Read-only files
 - Dedicated exclusively to a single process
 - E.g., Printers
- 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.
- Assume that when a process is denied a resource request, it is put to sleep.

Resources (5)

- 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 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();  
2  → { A.Down();  
3  →   B.Down();  
      use both resource  
4  →   B.Up();  
5  →   A.Up(); }
```

```
1  → Process Q();  
6  → { A.Down();  
      use both resource  
      B.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 both

B.Up();

A.Up();

1 → Process Q();

3 → { B.Down();

A.Down();

use both resources

A.Up();

B.Up(); }

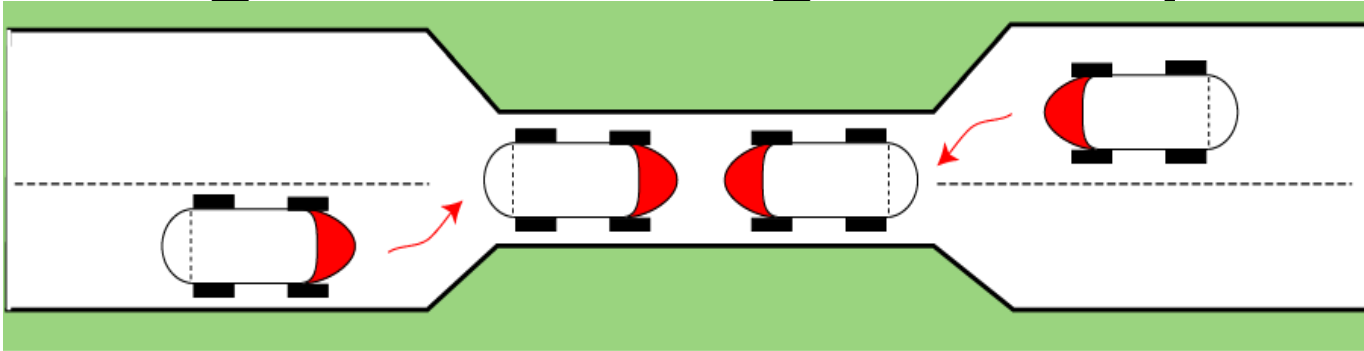
DEADLOCK

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.
- Starvation is possible.
 - Timid driver hesitates to enter bridge.

Introduction to Deadlocks (1)

■ 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.

- Assume that processes have only a single thread.
- Assume that there are no interrupts possible to wake up a blocked process.
- Usually the event is release of a currently held resource.
- The number of processes and the number and kind of resources possessed and requested are unimportant.

Introduction to Deadlocks (2)

- If deadlock, none of the processes can ...
 - Run
 - Release resources
 - Be awakened
- Is deadlock the same as starvation (or indefinitely postponed)?
 - A process is **indefinitely postponed** if it is delayed repeatedly over a *long* period of time while the attention of the system is given to other processes. I.e., logically the process may proceed but the system never gives it the CPU.



Group Discussion

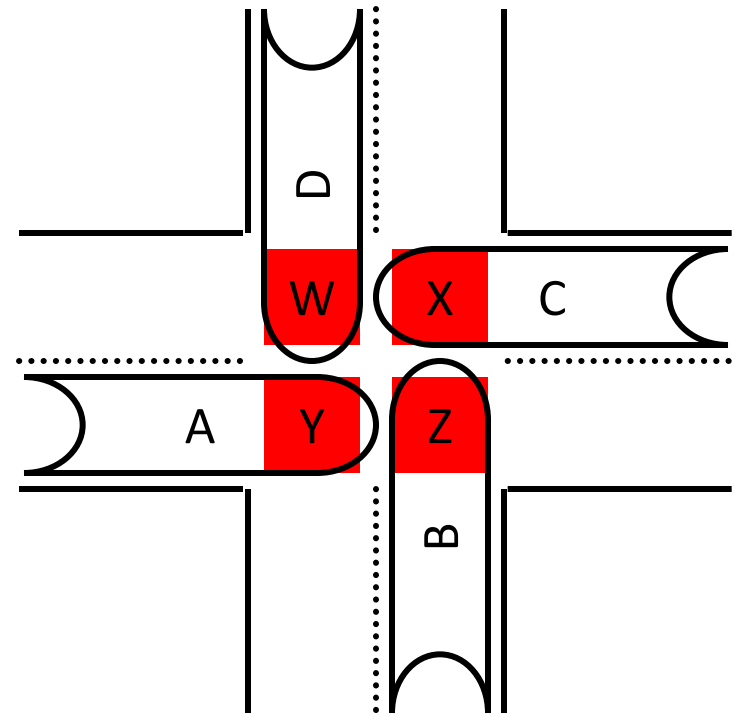
■ Conditions for Deadlock

- What conditions should exist in order to lead to a deadlock?
- Can use real life analogy such as
 - “You take the monitor, I grab the keyboard”

Traffic Jam as Deadlock Example

■ Cars in an Intersection

- Cars A, B, C, D
- Road W, X, Y, Z
- Car A holds road space Y, waiting for space Z
- “Gridlock”



Cars deadlocked
in an intersection

Four Conditions for Deadlock (1)

■ Mutual Exclusion Condition

- Each resource assigned to 1 process or is available

■ Hold and Wait Condition

- Process holding resources can request additional.

■ No Preemption Condition

- Previously granted resources cannot forcibly taken away.
- They must be explicitly released by the process holding them.

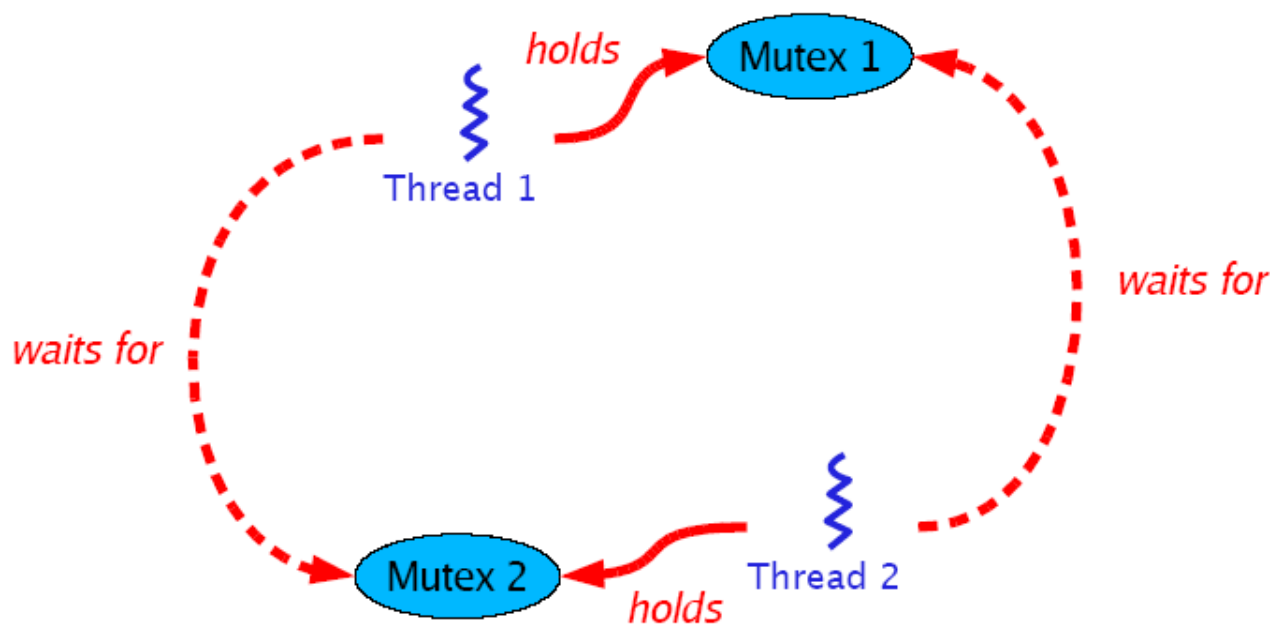
■ Circular Wait Condition

- There exists a set of processes $\{P_1, P_2, \dots, P_N\}$, such that, P_1 is waiting for P_2 , P_2 for P_3 , and P_N for P_1

Four Conditions for Deadlock (2)

■ Circular Wait Condition (ctd.)

□ Example



- Deadlock can arise if four conditions hold simultaneously.

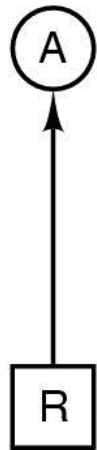


Conditions Analysis

- Deadlock occurs if and only if the circular wait condition is not resolvable.
- The circular wait condition is not resolvable when the first 3 policy conditions hold.
- Therefore, the four conditions taken together constitute the necessary and sufficient conditions for deadlock!

Deadlock Modeling (1)

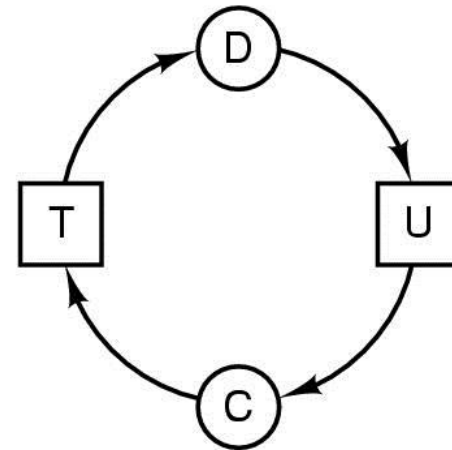
- Modeled with directed graphs



(a)



(b)



(c)

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

Deadlock Modeling (2)

A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

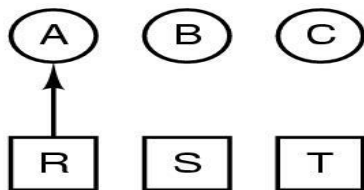
C

Request T
Request R
Release T
Release R

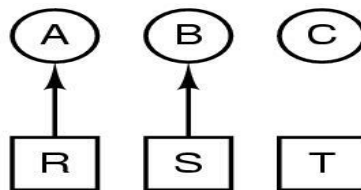
(c)

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

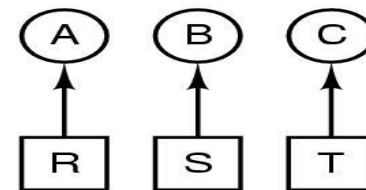
(d)



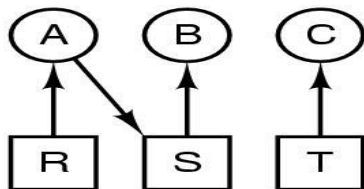
(e)



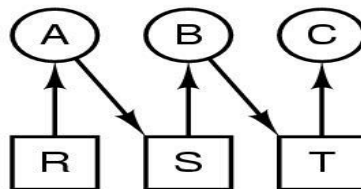
(f)



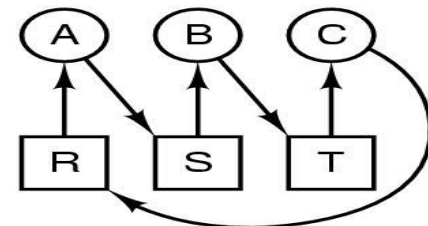
(g)



(h)



(i)

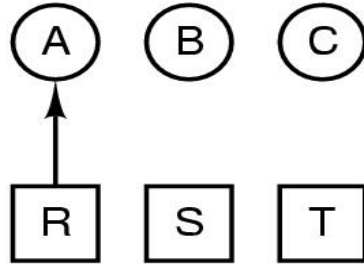


(j)

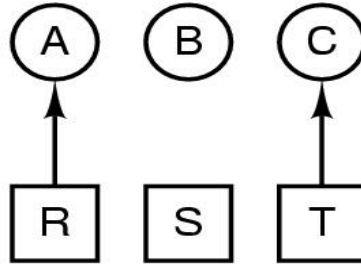
How deadlock occurs

Deadlock Modeling (3)

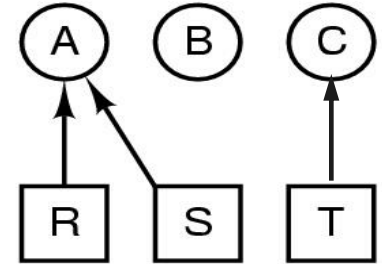
1. A requests R
 2. C requests T
 3. A requests S
 4. C requests R
 5. A releases R
 6. A releases S
- no deadlock



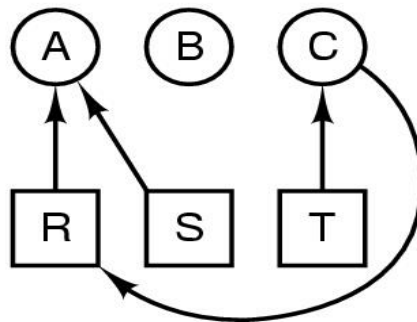
(k)



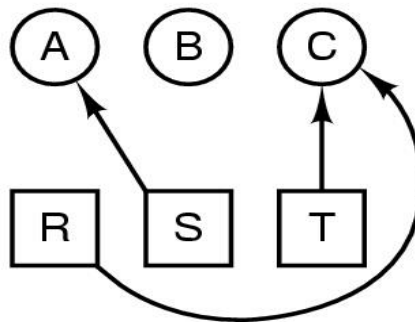
(m)



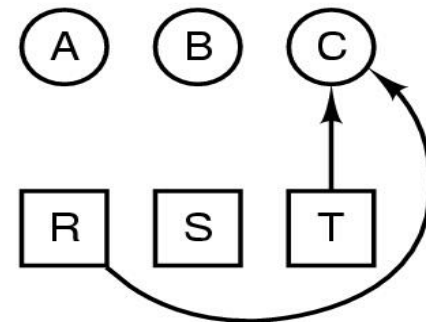
(n)



(o)



(p)



(q)

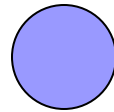
How deadlock can be avoided

Resource-Allocation Graph (1)

- A set of vertices V and a set of edges E .
 - V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, 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 (2)

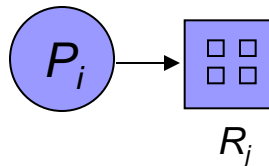
- Process



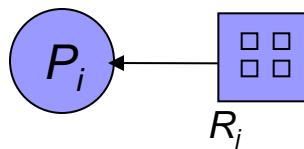
- Resource Type with 4 instances



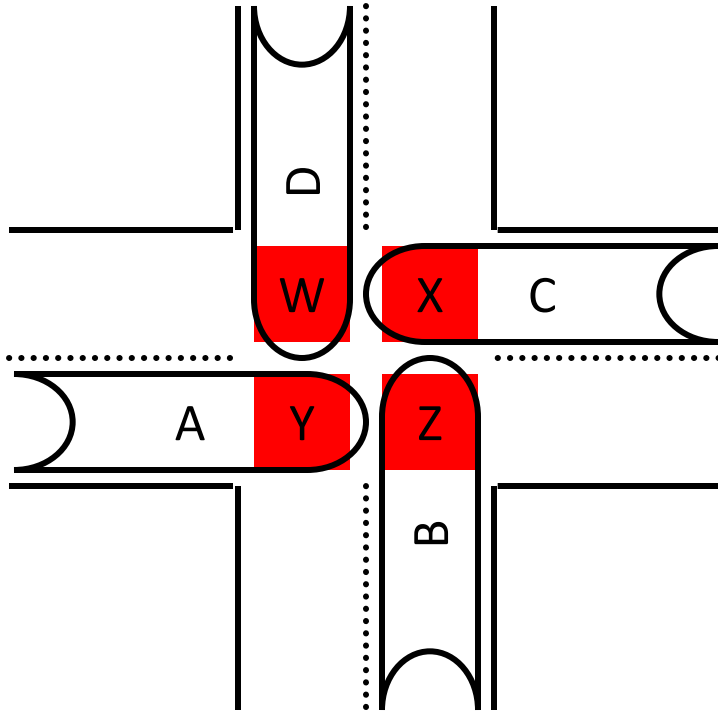
- P_i requests instance of R_j



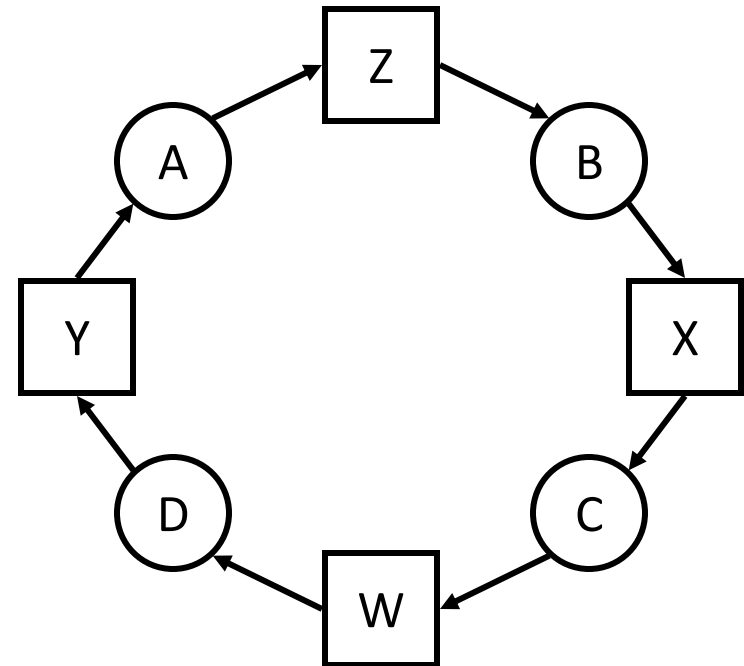
- P_i is holding an instance of R_j



Resource Allocation Graph of Traffic Jam

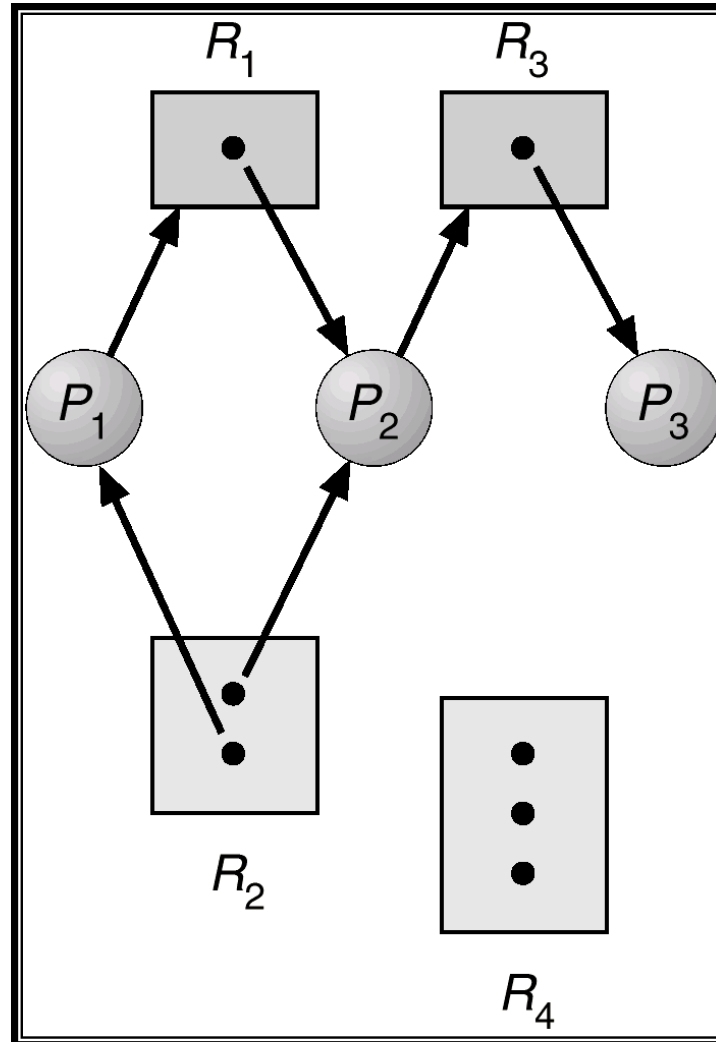


Cars deadlocked
in an intersection

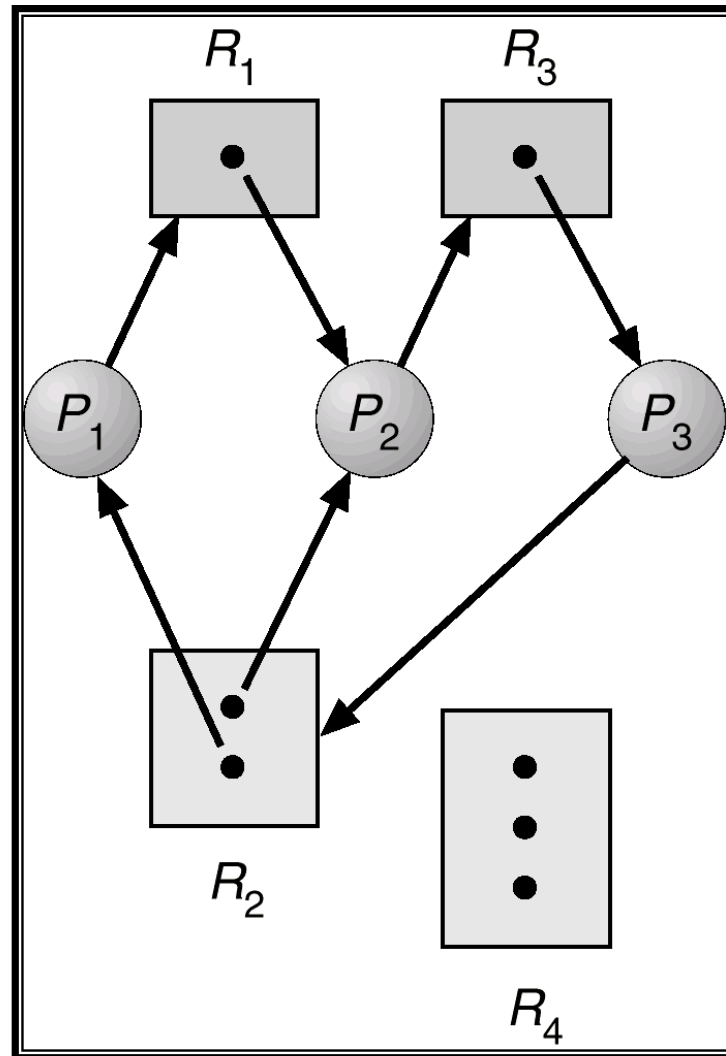


Resource Allocation
Graph

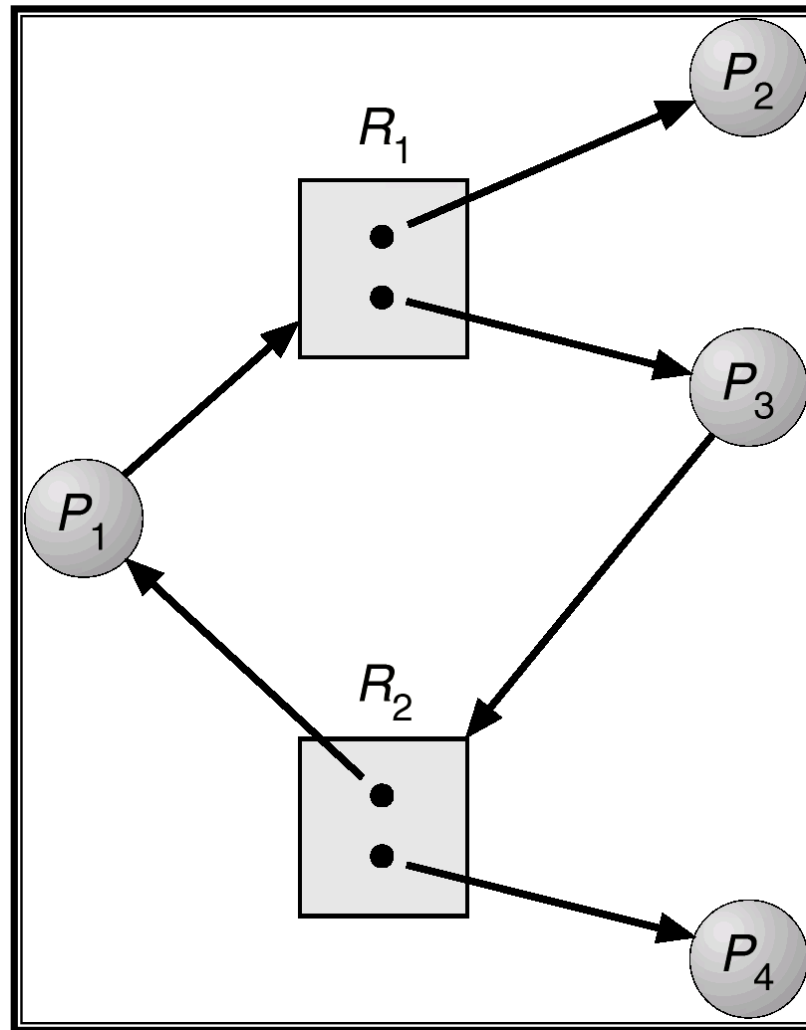
Example of a Resource Allocation Graph




Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock





Basic Facts about Resource Allocation Graph

- 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

- Ignore the problem and pretend that deadlocks never occur in the system.
- Allow the system to enter a deadlock state and then recover.
 - Deadlock detection & recovery
- Ensure that the system will never enter a deadlock state.
 - Dynamic avoidance (careful resource allocation)
 - Prevention (negating one of the four necessary conditions)

The Ostrich Algorithm (1)

- Don't do anything, simply restart the system (stick your head into the sand, pretend there is no problem at all).
- Consider
 - ☐ How often will a deadlock happen?
 - ☐ How severe will it be if it does happen?
 - ☐ How hard would it be to avoid/prevent/detect?



The Ostrich Algorithm (2)

- Rational: make the common path faster and more reliable.
 - Deadlock detection/recovery, avoidance or prevention algorithms are expensive.
 - If deadlock occurs only rarely, it is not worth the overhead to implement any of these algorithms.
- UNIX and Windows takes this approach for some of the more complex resource relationships to manage.
- It's a trade off between
 - Convenience (engineering approach)
 - Correctness (mathematical approach)



Deadlock Detection and Recovery

- Allow system to enter deadlock state.
- Run the Deadlock Detection algorithm periodically.
- After detecting deadlock, run a Recovery algorithm.
- Despite drawbacks, database systems do this.



Deadlock Detection

- How often should the detection algorithm be run?
 - Check every time a resource request is made.
 - Possibly expensive in terms of CPU time.
 - Check every k minutes, or perhaps only when the CPU utilization has dropped below some threshold.



Single Resource per Type (1)

- Check to see if a deadlock has occurred!
- Recall
 - If there is only one instance of each resource, it is possible to detect deadlock by constructing a resource allocation/request graph and checking for cycles.
- Graph theorists have developed a number of algorithms to detect cycles in a graph.

Single Resource per Type (2)

■ Deadlock Detection Algorithm

- 1. For **each node** N in the graph, perform the following 5 steps with N as the starting node.
- 2. Initialize L to the empty list, and designate all the arcs as unmarked. (L is the list of nodes)
- 3. Add the current node to the end of the 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.

Single Resource per Type (3)

■ Deadlock Detection Algorithm (ctd.)

- 4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
- 5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
- 6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, make that one the current node, and go to step 3.

Single Resource per Type (4)

■ Example

- Process A holds R and wants S.
- Process B holds nothing but wants T.
- Process C holds nothing but wants S.
- Process D holds U and wants S and T.
- Process E holds T and wants V.
- Process F holds W and wants S.
- Process G holds V and wants U.

Is this system deadlocked, and if so, which processes are involved?

Single Resource per Type (5)

- Example (ctd.)

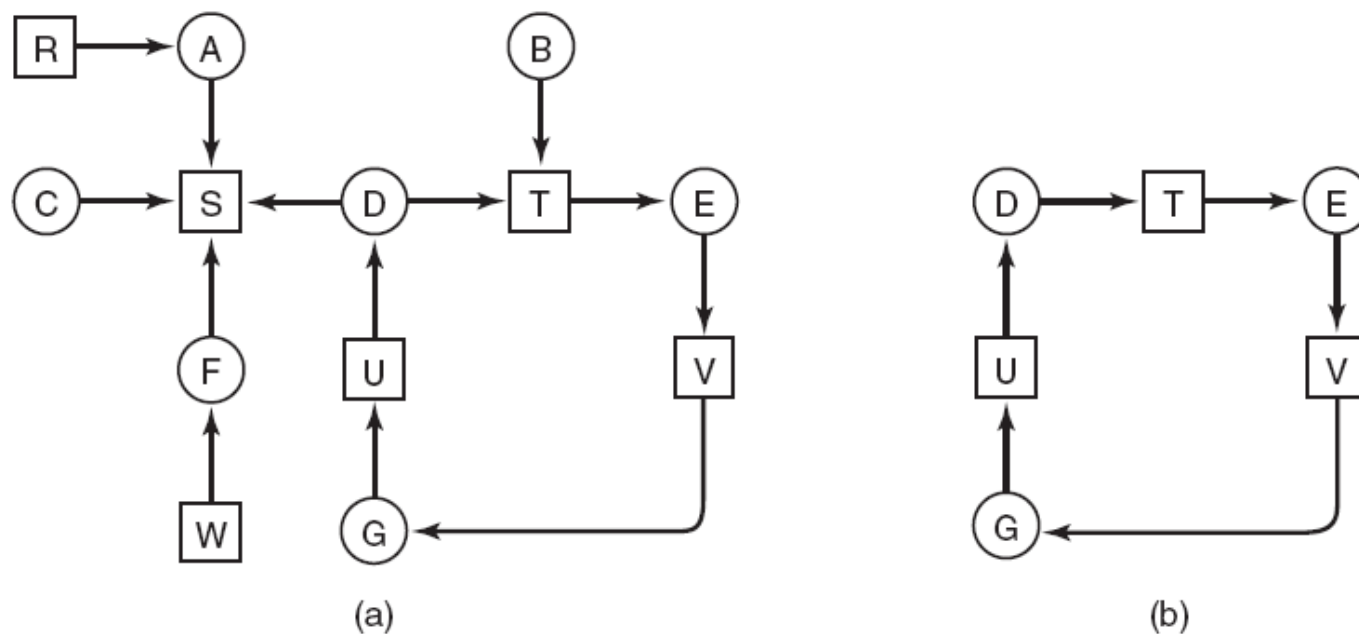


Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

Note the resource ownership and requests.

A cycle can be found within the graph, denoting deadlock.

Multiple Resources of Each Type (1)

■ Matrix-based Algorithm

- N processes, P_1 to P_n
- The number of resource classes: m
 - E_i : resources of class i ($1 \leq i \leq m$)
- E : the existing resource vector
- A : the available resource vector
- C : the current allocation matrix
- R : the request matrix

Multiple Resources of Each Type (2)

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

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

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

Figure 6-6 Data structures needed by deadlock detection algorithm

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

Multiple Resources of Each Type (3)

■ Deadlock Detection Algorithm (ctd.)

- Based on comparing vectors.

- $(A \leq B \text{ if and only if } A_i \leq B_i \text{ for } 0 \leq i \leq m)$

- 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.

Multiple Resources of Each Type (4)

■ Example 1

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Tape drives
Plotters
Scanners
Blu-rays

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Tape drives
Plotters
Scanners
Blu-rays

Current allocation matrix

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

Request matrix

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

Figure 6-7 An example for the deadlock detection algorithm

Multiple Resources of Each Type (5)

■ Example 2

□ Suppose, P3 needs 1 Blu-ray as well as 2 Tapes and 1 Plotter.

$$E = (4 \quad 2 \quad 3 \quad 1)$$

$$A = (2 \quad 1 \quad 0 \quad 0)$$

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

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

Deadlock!

Recovery from Deadlock (1)

- 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 process's state (memory image & resource state) to a file.
 - Use this saved state.
 - Restart the process if it is found deadlocked.

Recovery from Deadlock (2)

- Recovery through killing processes.
 - Crudest but simplest way to break a deadlock.
 - How to choose which process to kill?
 - Kill one of the processes in the deadlock cycle, then the other processes get its resources.
 - Kill the process not in the deadlock cycle, but hold the resources that some process in the cycle needs.
 - Kill process that can be rerun from the beginning with no ill effects.

Deadlock Avoidance (1)

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- This technique requires that the system would not allocate resources if deadlock *could* happen.
- Very conservative since just because deadlock *could* happen doesn't mean that deadlock *will* happen.
- 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.

Deadlock Avoidance (2)

■ Resource Trajectories

- Implication: At t point, to avoid deadlock, B should be suspended until A has requested and released the plotter.

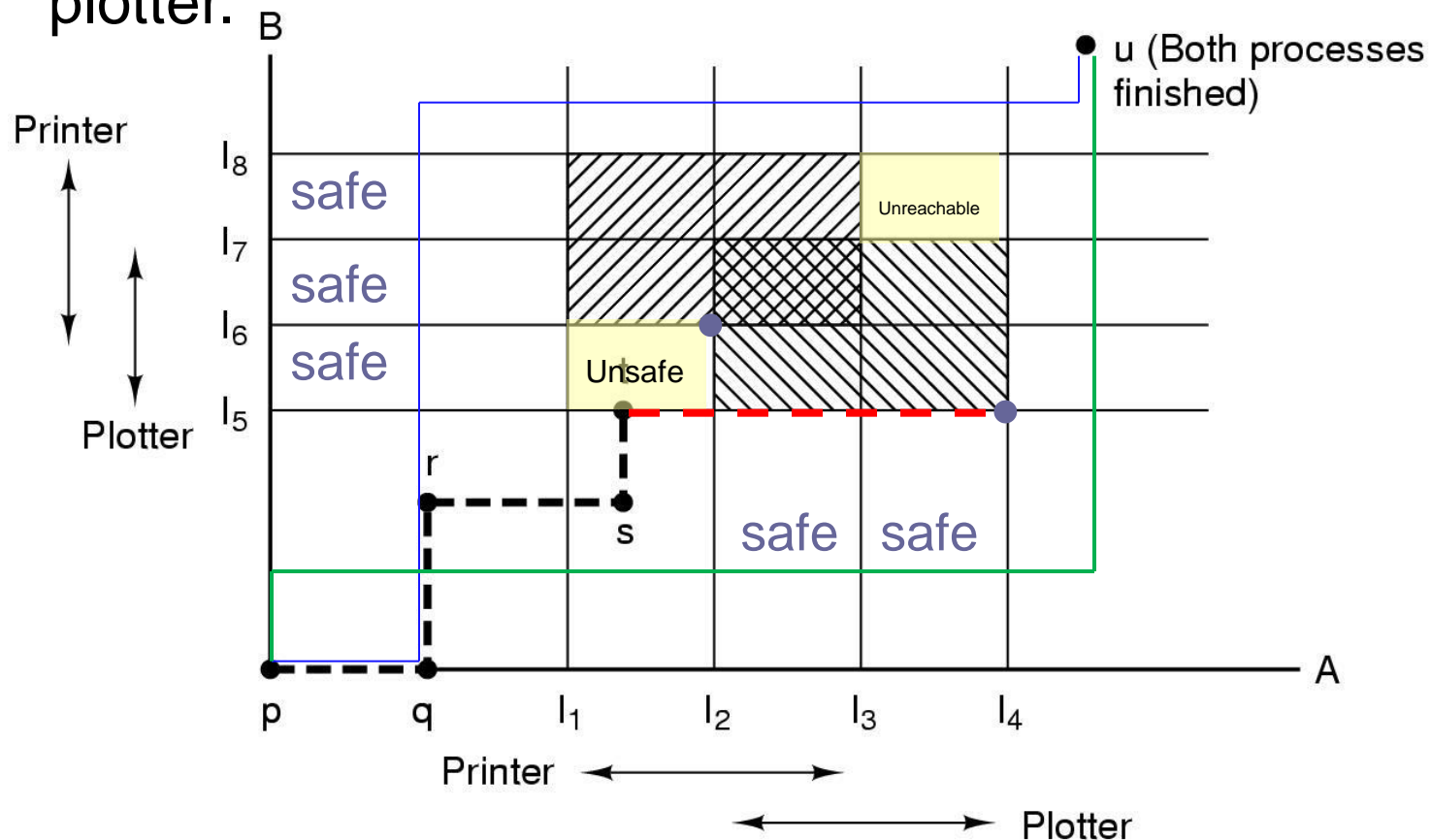


Figure 6-8 Two process resource trajectories

Safe and Unsafe State (1)

- The Current State

- Which processes hold which resources.

- Safe State

- The system is not deadlocked.
 - There exists a scheduling order that results in every process running to completion, even if they all request their maximum resources immediately.

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

Safe and Unsafe State (2)

■ Example : Safe State

□ Note: we have 10 units of the resource

| Has Max | | | Has Max | | | Has Max | | | Has Max | | | Has Max | | |
|---------|---|---|---------|---|---|---------|---|---|---------|---|---|---------|---|---|
| A | 3 | 9 | A | 3 | 9 | A | 3 | 9 | A | 3 | 9 | A | 3 | 9 |
| B | 2 | 4 | B | 4 | 4 | B | 0 | — | B | 0 | — | B | 0 | — |
| C | 2 | 7 | C | 2 | 7 | C | 2 | 7 | C | 7 | 7 | C | 0 | — |
| Free: 3 | | | Free: 1 | | | Free: 5 | | | Free: 0 | | | Free: 7 | | |
| (a) | | | (b) | | | (c) | | | (d) | | | (e) | | |

Figure 6-9 Demonstration that the state in (a) is safe
(there exists a safe sequence BCA)

Safe and Unsafe State (3)

■ Example : Unsafe State

□ A requests one extra unit resulting in (b)

| Has Max | | | 6 2 5 | Has Max | | | | Has Max | | | | Has Max | | | |
|---------|---|---|-------------|---------|---|---|--|---------|---|---|--|---------|---|---|--|
| A | 3 | 9 | | A | 4 | 9 | | A | 4 | 9 | | A | 4 | 9 | |
| B | 2 | 4 | | B | 2 | 4 | | B | 4 | 4 | | B | Đ | Đ | |
| C | 2 | 7 | | C | 2 | 7 | | C | 2 | 7 | | C | 2 | 7 | |
| Free: 3 | | | | Free: 2 | | | | Free: 0 | | | | Free: 4 | | | |
| (a) | | | | (b) | | | | (c) | | | | (d) | | | |

Figure 6-10 Demonstration that the state in b is not safe

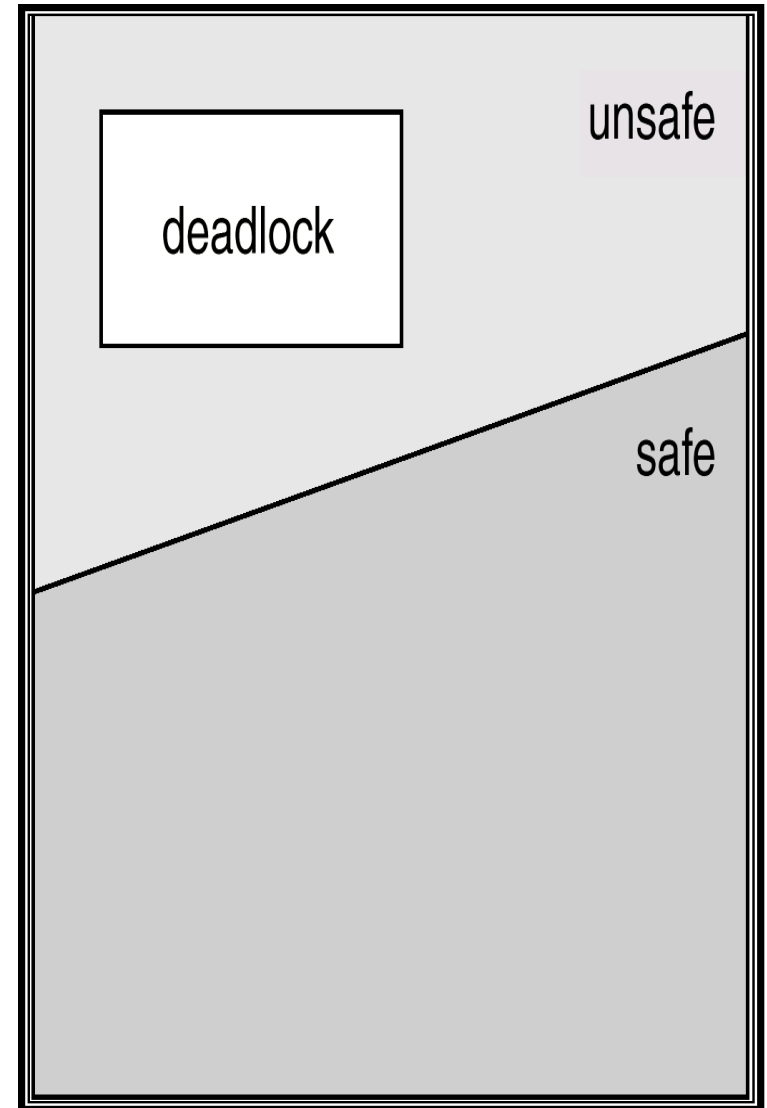
Safe and Unsafe State (4)

■ Unsafe States

- Being in an unsafe state does not guarantee deadlock.
- An unsafe state merely means that the OS cannot guarantee deadlock will not happen.
 - In other words, it is out of the hands of the OS.
- From a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

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
 - Dynamically examines the resource-allocation state
 - Ensure that a system will never enter an unsafe state.



Banker's Algorithm for a Single Resource (1)

■ The Banker's Algorithm

- Consider each request as it occurs, and see if granting it leads to a safe state.
- If it does, the request is granted; otherwise, it is postponed until later.

- *Safe state* - There is a lending sequence such that all customers can take out a loan.
- *Unsafe state* - There is a possibility of deadlock.

Banker's Algorithm for a Single Resource (2)

■ The Banker's Algorithm (ctd.)

- Example: Suppose we start with the following situation
 - Our banker has 10 credits to lend, but a possible liability of 22. His job is to keep enough in reserve so that ultimately each customer can be satisfied over time: That is, that each customer will be able to access his full credit line, just not all at the same time.

| Customer | Credit Used | Credit Line |
|-----------------|-------------|-------------|
| Andy | 0 | 6 |
| Bob | 0 | 5 |
| Mary | 0 | 4 |
| Sue | 0 | 7 |
| Funds Available | 10 | |
| Max Commitment | | 22 |

Banker's Algorithm for a Single Resource (3)

■ The Banker's Algorithm (ctd.)

□ Example (ctd.)

- Suppose, after a while, the bank's credit line book shows
- The question then is: Does a way exist such that each customer can be satisfied? Can each be allowed their maximum credit line in some sequence?

| Customer | Credit Used | Credit Line |
|-----------------|-------------|-------------|
| Andy | 1 | 6 |
| Bob | 1 | 5 |
| Mary | 2 | 4 |
| Sue | 4 | 7 |
| Funds Available | 2 | |
| Max Commitment | | 22 |

Safe

Banker's Algorithm for a Single Resource (4)

- The Banker's Algorithm (ctd.)

- Example (ctd.)

- Suppose, however, that the banker proceeded to award Bob one more credit after the credit book arrived at the state immediately above:

| Customer | Credit Used | Credit Line |
|-----------------|-------------|-------------|
| Andy | 1 | 6 |
| Bob | 2 | 5 |
| Mary | 2 | 4 |
| Sue | 4 | 7 |
| Funds Available | 1 | |
| Max Commitment | | 22 |

Unsafe

Banker's Algorithm for Multiple Resources (1)

■ The 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 \Rightarrow not safe. (If several processes are eligible to be chosen, it does not matter which one is selected.)
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 \Rightarrow safe state
or not marked \Rightarrow not safe state.

Banker's Algorithm for Multiple Resources (2)

■ Example

- Figure 6-12 is safe state
- Suppose that process B now makes a request for the printer. Can this request be granted?
- After giving B one printer, E wants the last printer. Can this request be granted?

| | Process | Tape drives | Plotters | Printers | Blu-rays |
|---|---------|-------------|----------|----------|----------|
| 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 | Printers | Blu-rays |
|---|---------|-------------|----------|----------|----------|
| 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 assigned

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

R

Figure 6-12 Example of banker's algorithm with multiple resources

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 | | | |

If P1 requests max resources, can complete

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 | 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 | | | |

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.



Banker's Problem 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).



Deadlock Prevention

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

Attacking the Mutual Exclusion Condition (1)

- Some devices (such as printer) can be spooled.
- Example: By spooling printer output and allocating the actual printer to only the printer daemon.
 - Printer daemon doesn't request other resources, so no deadlock for the printer possible.
 - Question – Should the daemon start printing before all the output is spooled?
 - 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 (2)

■ Principle

- Avoid assigning resource when not absolutely necessary.
- As few processes as possible actually claim the resource.

■ Problem

- Not all devices can be spooled.

■ Not a general solution, but useful nevertheless.



Attacking the Hold and Wait Condition (1)

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- Approach 1: Require all processes to request all their resources before starting execution.
 - If everything is available, the process will be allocated whatever it needs and can run to completion.
 - If one or more resources are busy, the process will just wait.
 - Still done sometimes in the mainframe world.

Attacking the Hold and Wait Condition (2)

■ Problems of Approach 1

- May not know required resources at start of run.
 - Of course, if we could do that, we could use the Banker's Algorithm.
- Resources will not be used efficiently.
 - Read data from tape, then analyze it for a week, then write back to the tape.
 - The tape would be unusable by other processes that entire time.

Attacking the Hold and Wait Condition (3)

- Approach 2: Process must give up all resources, then request all immediately needed.
 - Might work for some programs, but many could not possibly (or efficiently) do this.
- Problem of Approach 2
 - Doesn't work if some resources must be held.

Attacking the No Preemption Condition (1)

■ Difficult to achieve in practice

- Cannot take a printer away from a process in the middle of printing.
- Cannot take a semaphore away from a process arbitrarily.
 - Might be in the middle of updating a shared area.
- Cannot take open streams, pipes and sockets away.
 - Process would need to be written very carefully, probably using *signals*.
 - Very undesirable if possible at all.

Attacking the No Preemption Condition (2)

■ Occasionally possible

- Processes resident in main memory.
- Some deadlock occurs such as failure to allocate a page.
- One or more processes can be *swapped* out to disc to release their pages and allow remaining processes to continue.
 - As long as they release any other resources they also hold on the way out.
- Put back on the scheduling queues to be re-admitted to memory later.

Attacking the No Preemption Condition (3)

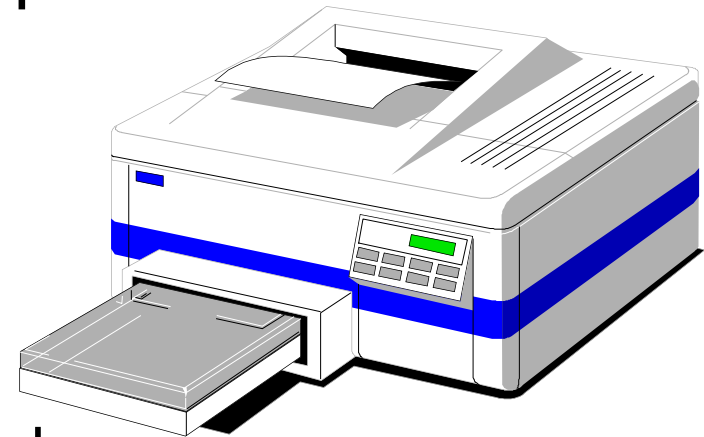
■ Examples

□ Consider a process given the printer

- Halfway through its job.
- Now forcibly take away printer.
- !!??

□ Virtualization of Resources

- Spooling printer output to the disk.
- Allow only the printer daemon access to the real printer.



Attacking the No Preemption Condition (4)

■ Approaches

- Virtualization of Resources

- Some resources can't be virtualized

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.
- The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

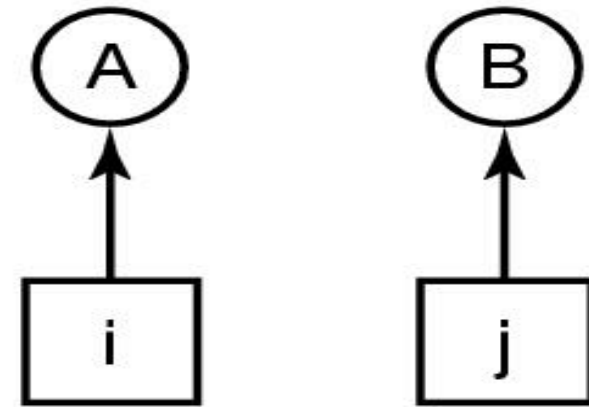
Attacking the Circular Wait Condition (1)

- 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
 - Req(resource1), req(resource2)..
 - Why no circular wait?
 - Figure (next page)
- Approach 3: Variation of Approach 2
 - No process request a resource lower than what it is already holding.

Attacking the Circular Wait Condition (2)

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

(a)



(b)

- (a) Normally ordered resources
- (b) A resource graph
 - Number all resources and require all requests to be made in numerical order.



Attacking the Circular Wait Condition (3)

■ 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 |



Group Discussion

- How to Prevent Deadlock in Dining Philosopher Problem?
 - Attacking the mutual exclusion condition?
 - Attacking the hold and wait condition?
 - Attacking the no preemption condition?
 - Attacking the circular wait condition?



Integrated Deadlock Strategy

- We can combine the previous approaches into the following way:
 - Group resources into a number of different classes and order them. Eg.
 - Swappable space (secondary memory)
 - Process resources (I/O devices, files...)
 - Main memory...
 - Use prevention of circular wait to prevent deadlock between resource classes.
 - Use the most appropriate approach for each class for deadlocks within each class.



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.
- ☐ No real work done in phase one.

■ 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 (1)

■ Resource Deadlocks

- A process needs multiple resources for an activity.
- Deadlock occurs if each process in a set request resources held by another process in the same set, and it must receive all the requested resources to move further.

■ Communication Deadlocks

- Processes wait to communicate with other processes in a set.
- Each process in the set is waiting on another process's message, and no process in the set initiates a message until it receives a message for which it is waiting.

- One technique to break communication deadlock is timeout.

Communication Deadlocks (2)

- Not all deadlocks occurring in communication systems or networks are communication deadlocks.

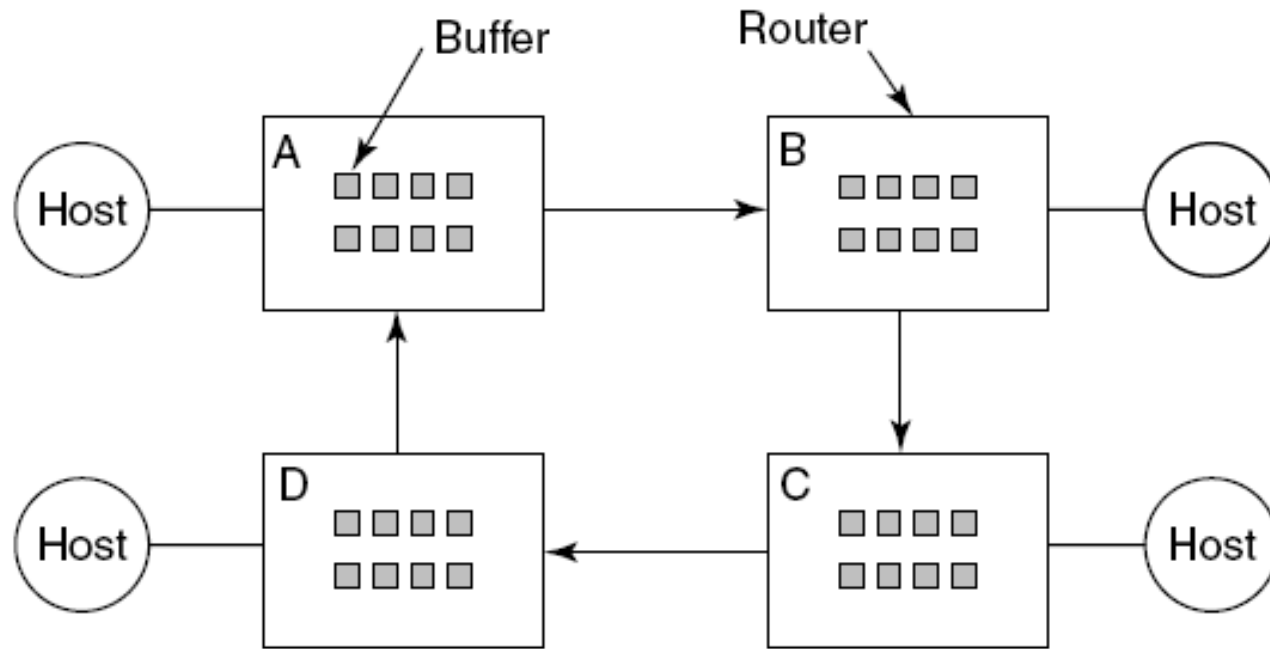


Figure 6-15. A resource deadlock in a network.

Livelock (1)

- Deadlock
 - Processes are blocked.
- Livelock
 - Processes run but make no progress.
- Both deadlock and livelock lead to starvation.
- But starvation may have other causes:
 - MLQ scheduling where one queue is never empty.
 - Memory requests: unbounded stream of 100MB requests may starve a 200MB request

Livelock (2)

■ Livelock Example: Figure 6-16

```
void process_A(void) {  
    acquire_lock(&resource_1);  
    while (try_lock(&resource_2) == FAIL) {  
        release_lock(&resource_1);  
        wait_fixed_time();  
        acquire_lock(&resource_1);  
    }  
    use_both_resources( );  
    release_lock(&resource_2);  
    release_lock(&resource_1);  
}
```

```
void process_A(void) {  
    acquire_lock(&resource_2);  
    while (try_lock(&resource_1) == FAIL) {  
        release_lock(&resource_2);  
        wait_fixed_time();  
        acquire_lock(&resource_2);  
    }  
    use_both_resources( );  
    release_lock(&resource_1);  
    release_lock(&resource_2);  
}
```

Starvation (1)

- 饥饿(**starvation**): 当等待时间给进程推进和响应带来明显影响时,称发生了进程饥饿。饥饿到一定程度的进程所赋予的使命即使完成也不再具有实际意义时称该进程被饿死(**starved to death**)。
- 没有时间上界的等待
 - 排队等待
 - 忙式等待
- 忙式等待条件下发生的饥饿,称为活锁(**live lock**).

Starvation (2)

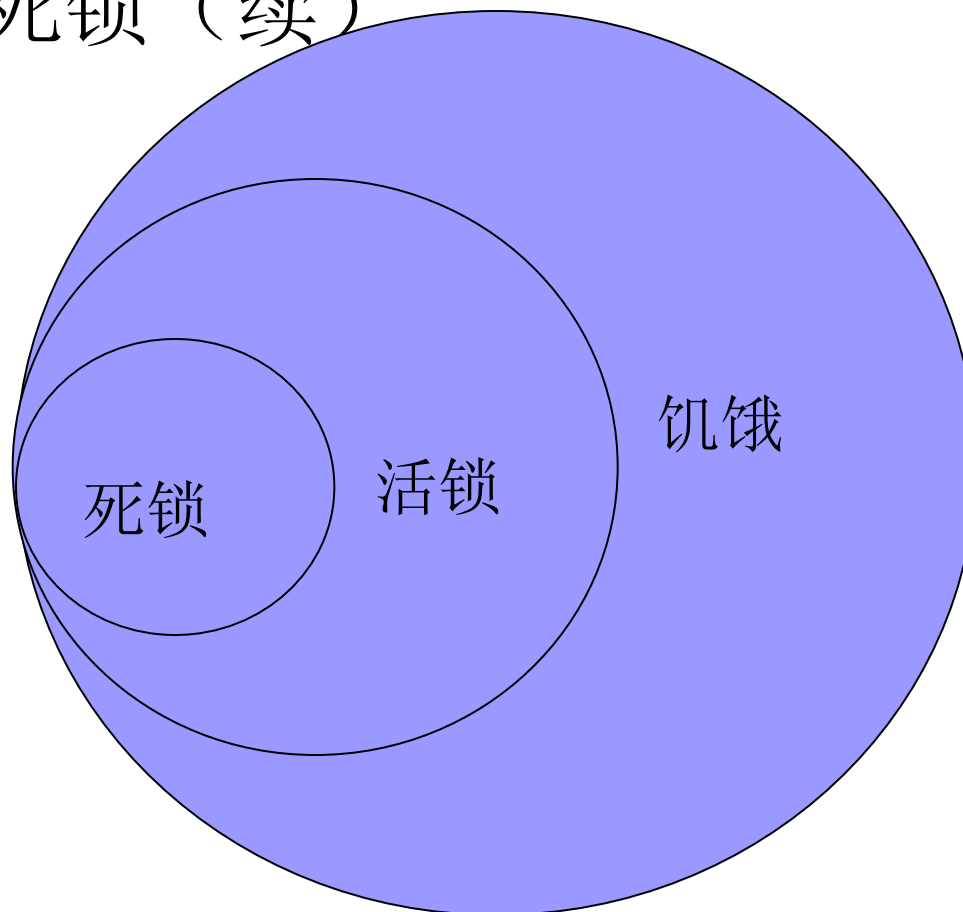
■ 饥饿 vs 死锁

- 死锁进程处于等待状态，忙式等待的进程并非处于等待状态，但却可能被饿死
- 死锁进程等待永远不会释放的资源，饿死进程等待可能被释放，但却不会分给自己的资源，其等待时间没有上界
- 死锁一定发生了循环等待，饿死不然；
- 死锁至少涉及两个进程，饿死进程可能只有一个

- 死锁、活锁、饥饿是包含与被包含的关系。死锁是活锁的特例，而活锁又是饥饿的特例。

Starvation (3)

■ 饥饿 vs 死锁（续）



Deadlock Summary (1)

| Approach | Resource Allocation Policy | Different Schemes | Major Advantages | Major Disadvantages |
|------------|--|---|---|---|
| Prevention | Conservative; undercommits resources | Requesting all resources at once | <ul style="list-style-type: none"> • Works well for processes that perform a single burst of activity • No preemption necessary | <ul style="list-style-type: none"> • Inefficient • Delays process initiation • Future resource requirements must be known by processes |
| | | Preemption | <ul style="list-style-type: none"> • Convenient when applied to resources whose state can be saved and restored easily | <ul style="list-style-type: none"> • Preempts more often than necessary |
| | | Resource ordering | <ul style="list-style-type: none"> • Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design | <ul style="list-style-type: none"> • Disallows incremental resource requests |
| Avoidance | Midway between that of detection and prevention | Manipulate to find at least one safe path | <ul style="list-style-type: none"> • No preemption necessary | <ul style="list-style-type: none"> • Future resource requirements must be known by OS • Processes can be blocked for long periods |
| Detection | Very liberal; requested resources are granted where possible | Invoke periodically to test for deadlock | <ul style="list-style-type: none"> • Never delays process initiation • Facilitates online handling | <ul style="list-style-type: none"> • Inherent preemption losses |

Deadlock Summary (2)

| 方法 | 资源分配策略 | 各种可能模式 | 主要优点 | 主要缺点 |
|------------------|----------------------------|--|--|--|
| 检测 Detection | 宽松的；只要允许，就分配资源 | 定期检查死锁是否已经发生 | 不延长进程初始化时间；允许对死锁进行现场处理 | 通过剥夺解除死锁，造成损失 |
| 避免 Avoidance | 是“预防”和“检测”的折衷（在运行时判断是否可能死锁 | 寻找可能的安全的运行顺序 | 不必进行剥夺 | 必须知道将来的资源需求；进程可能会长时间阻塞 |
| 预防 Prevention | 保守的；宁可资源闲置 | Spooling 一次请求所有资源 资源剥夺 资源按序申请 | 适用于作突发式处理的进程；不必剥夺 适用于状态可以保存和恢复的资源 可以在编译时（而不必在运行时）就进行检查 | 效率低；进程初始化时间延长 剥夺次数过多；多次对资源重新启动 不便灵活申请新资源 |

Process Management (1)

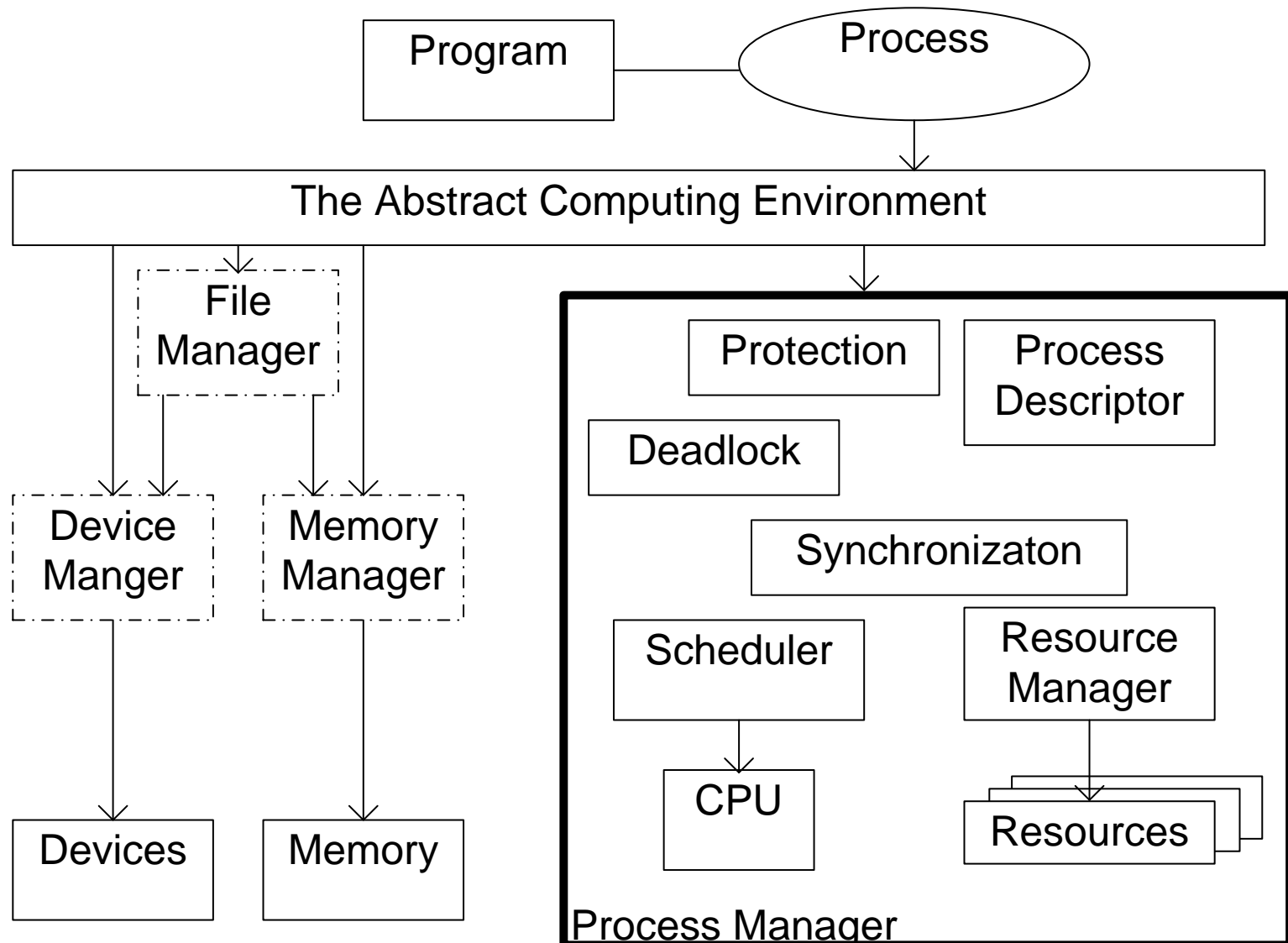
■ Process components:

- A program to define the behavior of the process.
- The data operated on by the process and the results it produces.
- A set of resources to provide an environment for the execution.
- A status record to keep track of the progress and control of the process during execution.

■ Process manager functions:

- Implements CPU sharing (called *scheduling*).
- Must allocate resources to processes in conformance with certain policies.
- Implements process synchronization and inter-process communication.
- Implements deadlock strategies and protection mechanisms.

Process Management (2)



Homework

- P465: 2, 9,14,31,34

- 补充题

□ A system has five processes and four allocatable resources. The current allocation and additional needs are as follows:

| Process | Allocation | | | | Need | | | | Available | | | |
|---------|------------|---|---|---|------|---|---|---|-----------|---|---|---|
| | A | B | C | D | A | B | C | D | A | B | C | D |
| P1 | 0 | 0 | 3 | 2 | 0 | 0 | 1 | 2 | 1 | 6 | 2 | 2 |
| P2 | 1 | 0 | 0 | 0 | 1 | 7 | 5 | 0 | | | | |
| P3 | 1 | 3 | 5 | 4 | 2 | 3 | 5 | 6 | | | | |
| P4 | 0 | 3 | 3 | 2 | 0 | 6 | 5 | 2 | | | | |
| P5 | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 | | | | |

Please answer the following questions:

- (1) Is this state safe? Why?
- (2) The request (1,2,2,2) of P3 can be granted or not? Why?