Chapter 6

Deadlocks

Resource (1)

- A **resource** is a commodity needed by a process.
  - printers
  - Disk
  - CPU time
- Resources can be either:
  - **serially reusable:** e.g., CPU, memory, disk space, I/O devices, files.
    acquire □ use □ release
  - **consumable: produced** by a process, needed by a process; e.g., messages, buffers of information, interrupts.
    create □ acquire □ use
    Resource ceases to exist after it has been used, so it is not released.

Resource (2)

- Resources can be either:
  - **shared** among several processes or
  - **dedicated** exclusively to a single process.

Resource (3)

- Resources can also be either:
  - **preemptable**: can be taken away from a process with no ill effects
  - e.g., CPU, Memory
  - **non-preemptable**: will cause the process to **fail** if taken away
  - e.g., CD recorder.
- Generally Deadlocks involve exclusive access to nonpremtable resources.

Resources (4)

- Sequence of events required to use a resource
  1. request the resource
  2. use the resource
  3. release the resource

- If requested resource is not available the requesting process is **blocked**

Resource Access Conrol Using Semaphores

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

```
        Process Q();
5  ⇒   { A.Down();
         B.Down();
           use both resource
         B.Up();
         A.Up(); }
```

Semaphore A(1), B(1);

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

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

3 Semaphore A(0), B(1);

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

But Deadlock can Happen!

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

DEADLOCK

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

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
  - This kind of deadlock is called **Resource Deadlock**
- None of the processes can …
  - run
  - release resources
  - be awakened
- A real life example
  - Kansas 20th century law: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone"

Four Conditions for Deadlock

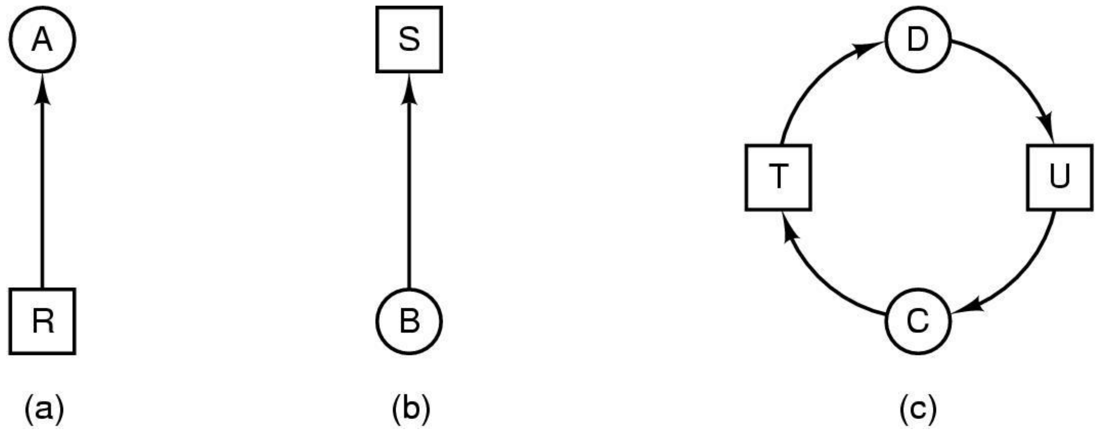
1. Mutual exclusion condition
   - each resource assigned to exactly 1 process or is available
2. Hold and wait condition
   - process holding resources can request additional resources
3. No preemption condition
   - previously granted resources cannot forcibly taken away
4. Circular wait condition
   - must be a circular chain of 2 or more processes
   - each is waiting for resource held by next member of the chain
     - $\boldsymbol{T_1}$ is waiting for a resource that is held by T_2
     - T_2 is waiting for a resource that is held by T_3
     - …
     - T_n is waiting for a resource that is held by $\boldsymbol{T_1}$

Four Conditions for Deadlock

- All four of these conditions must be present for a deadlock to occur.
- If one of them is absent, no deadlock is possible.

Deadlock Modeling (1)

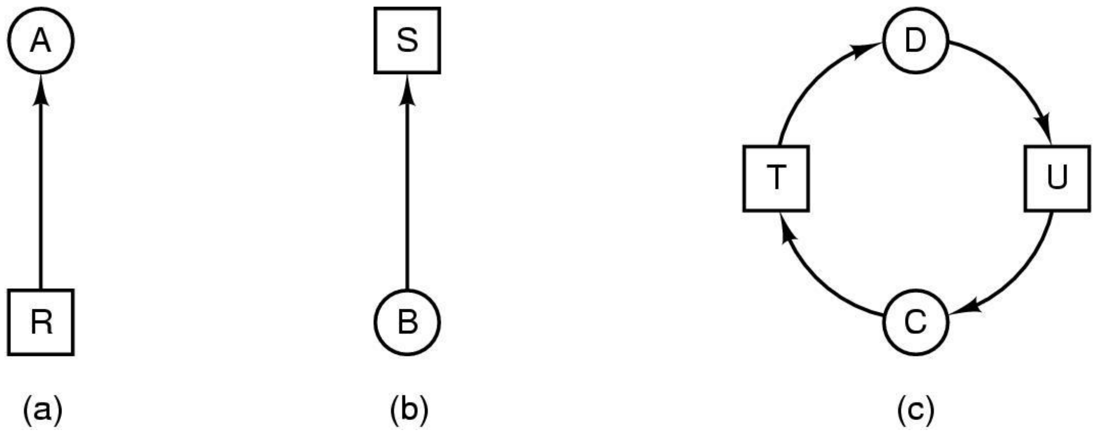
- Modeled with directed graphs
  - 2 types of nodes: process, resource
    - circular: process, rectangle: resource



  - edges
    - R -> A resource R assigned to process A
    - B -> S process B is blocked waiting for resource S
  - process C and D are in deadlock over resources T and U

Deadlock Modeling (2)

- **A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind).**
- **In this example, the cycle is C–T–D–U–C.**

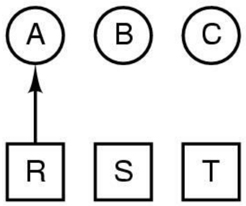


Deadlock Modeling (3)

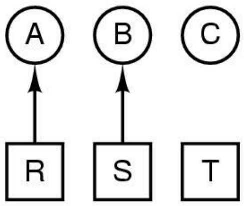
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

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

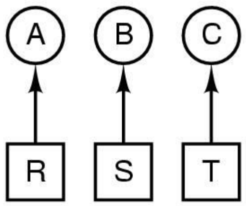
(d)



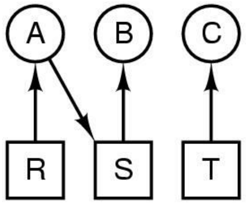
(e)



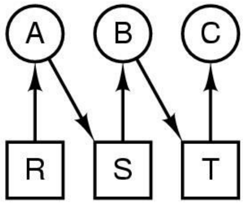
(f)



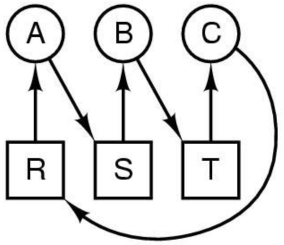
(g)



(h)



(i)



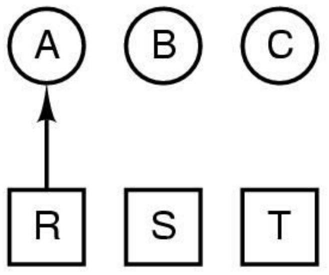
(j)

How deadlock occurs

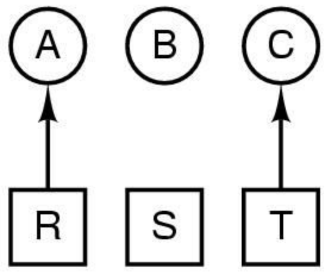
Deadlock Modeling (4)

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

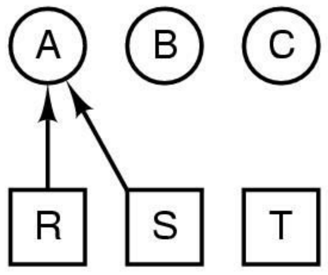
(k)



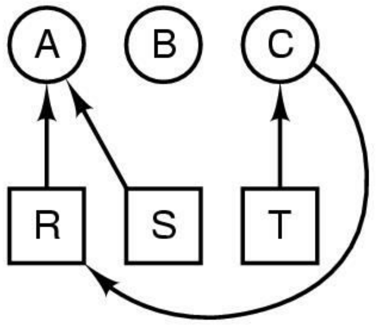
(l)



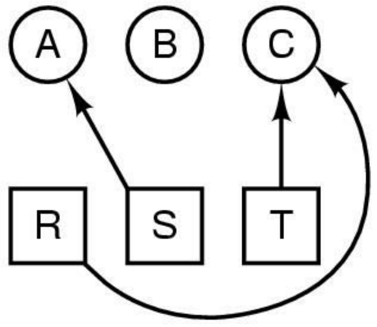
(m)



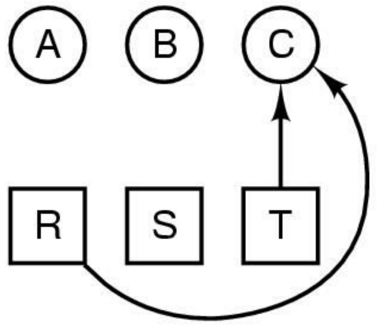
(n)



(o)



(p)



(q)

How deadlock can be avoided

Deadlock Modeling (5)

- resource graphs are a **tool** for finding if a given request/release sequence leads to deadlock.
- We just carry out the requests and releases step by step, and after every step check the graph to see if it contains any **cycles**.
- Although we used resource graphs for the case of a single resource of each type,
- resource graphs can also be generalized to handle multiple resources of the same type

Dealing with Deadlocks

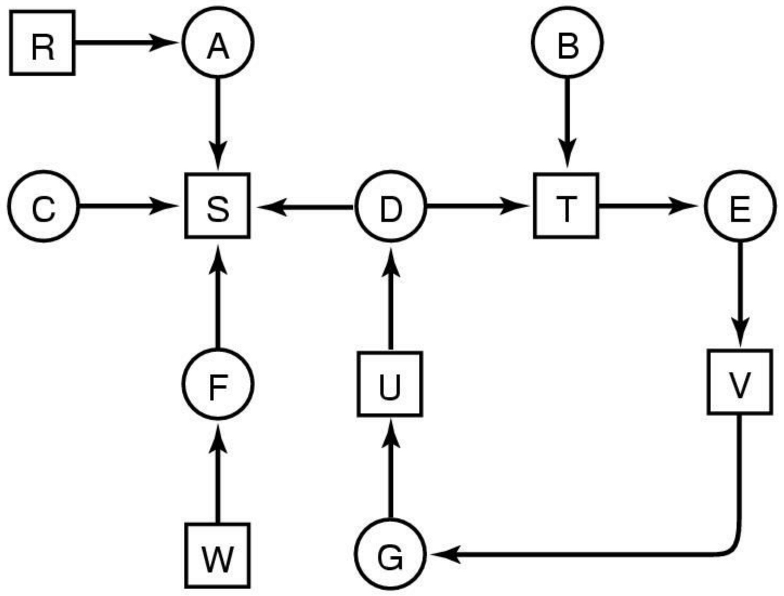
Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. detection and recovery
   - Let deadlocks occur, detect them, and take action.
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating **one** of the four necessary conditions

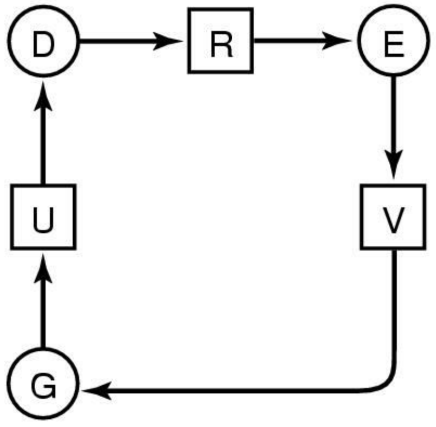
The Ostrich Algorithm

- The simplest approach
  - Don’t do anything, **simply restart** the system (stick your head into the sand, pretend there is no problem at all).
- Rational:
  - Deadlock prevention, avoidance or detection/recovery 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
- It is a trade off between
  - convenience
  - correctness

Detection with One Resource of Each Type (1)



(a)

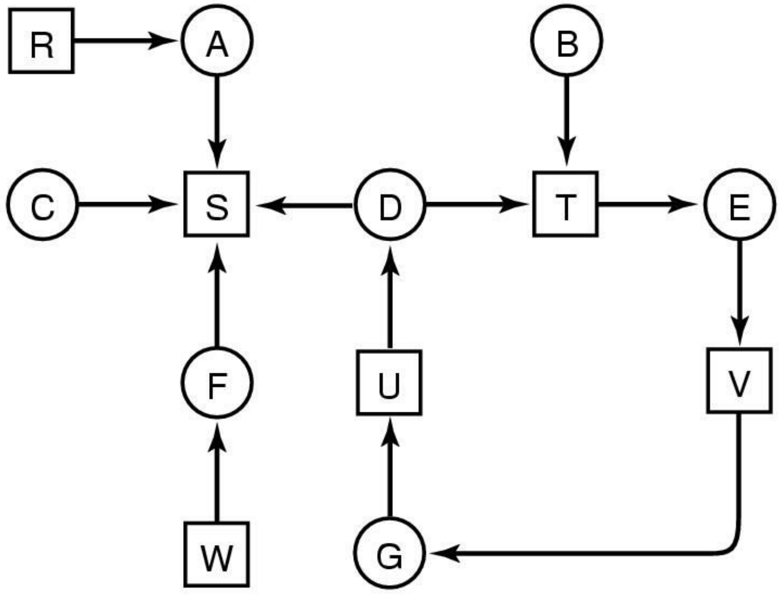


(b)

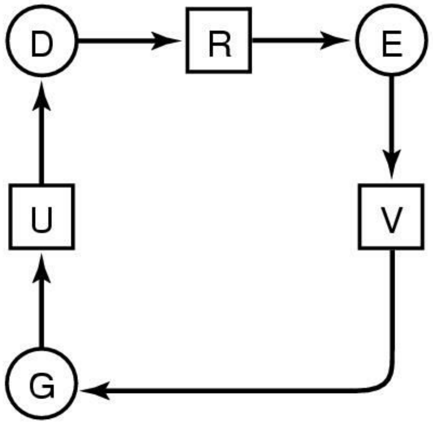
- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

Detection with One Resource of Each Type (1)

1. Process A holds R and wants S.
2. Process B holds nothing but wants T.
3. Process C holds nothing but wants S.
4. Process D holds U and wants S and T.
5. Process E holds T and wants V.
6. Process F holds W and wants S.
7. Process G holds V and wants U.



(a)



(b)

Detection with One Resource of Each Type (1)

Algorithm to detect:
A simple algorithm that inspects a graph and terminates either when it has found a cycle or when it has shown that none exists. It uses

- one dynamic data structure, L,
- a list of nodes, as well as a list of arcs.

During the algorithm, to prevent repeated inspections, arcs will be marked to indicate that they have already been inspected,

Detection with One Resource of Each Type (1)

Algorithm to detect:

The algorithm operates by carrying out the following steps as specified:

1. For each node, N, in the graph, perform the following five steps with N as the starting node.
2. Initialize L to the empty list, and designate all the arcs as unmarked.
3. 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.

   for example, If L = [P1, R1, P2, R2, P1],
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, that is, the one that was current just before this one, make that one the current node, and go to step 3.

Detection with Multiple Resource of Each Type (1)

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

Detection with Multiple Resource of Each Type (2)

- Invariant: $\sum_{1 \le i \le n} C_{ij} + A_j = E_j$
- Algorithm:
  - Look for an unmarked process, P_i, for which the i-th row of R is less than or equal to A.
  - If such a process is found, add the i-th row of C to A, mark the process, go to step 1.
  - If no such process exists, terminate
  - After termination, all the unmarked processes are deadlocked.

Detection with Multiple Resource of Each Type (2)

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

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

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

Detection with Multiple Resource of Each Type (2)

Process Checked	Can Run?	After Running → Available (A)
Process 1	❌ No (no CD rom)	—
Process 2	❌ No (no scanner)	—
Process 3	✅ Yes	(2 2 2 0)
Process 2 (again)	✅ Yes	(4 2 2 1)
Process 1 (finally)	✅ Yes	(All processes finished)

Recovery from Deadlock (1)

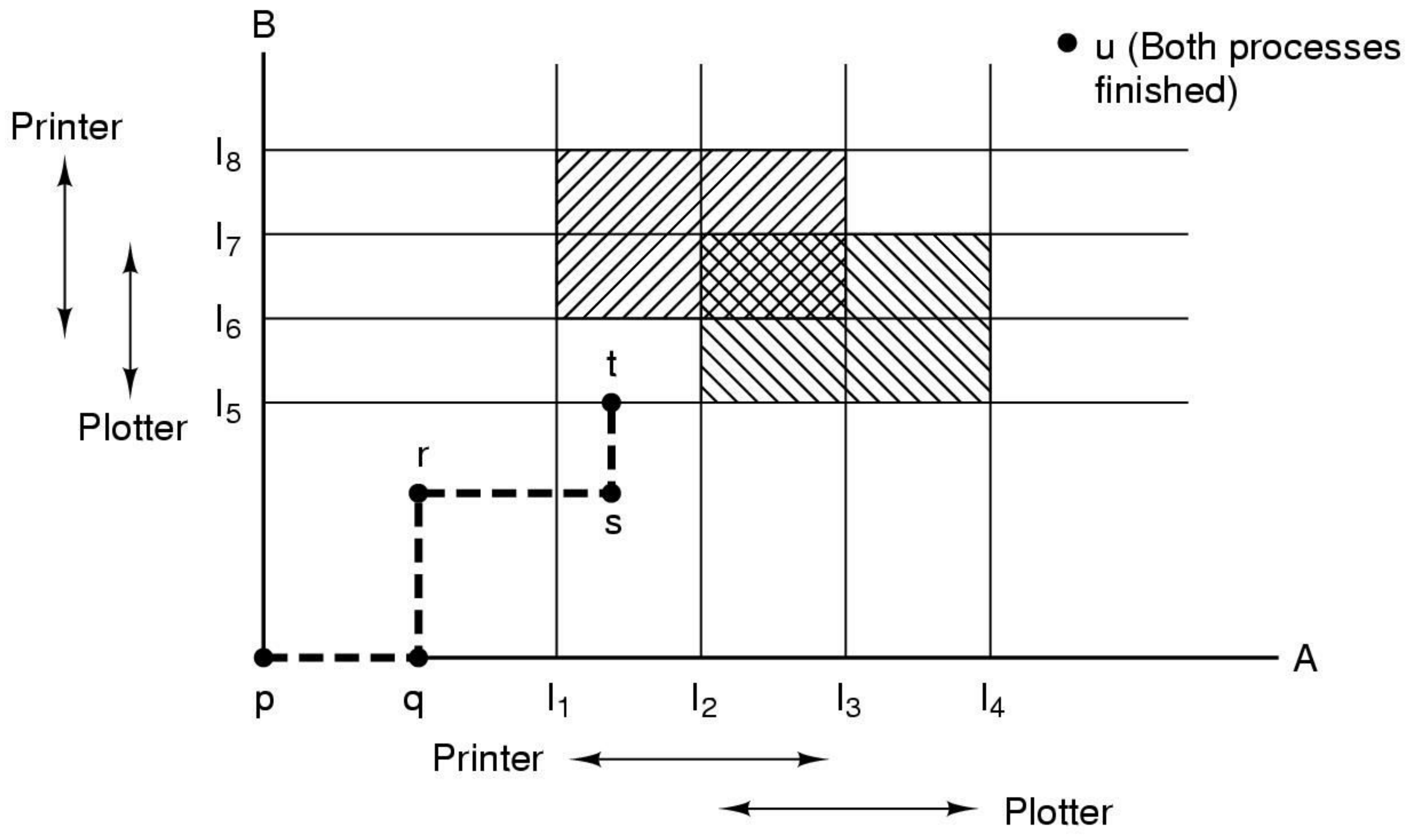
- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked

Recovery from Deadlock (2)

- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning
  - Alternatively, one of the processes not deadlocked may be killed!!!

Deadlock Avoidance

Resource Trajectories



Two process resource trajectories

Safe and Unsafe States (1)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

Demonstration that the state in (a) is safe

Safe and Unsafe States (2)

	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 state in b is not safe

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)

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

Free: 1

(c)

- Three resource allocation states
  - safe
  - safe
  - 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

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

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

Example of banker's algorithm with multiple resources

Banker's Algorithm for Multiple Resources: Algorithmic Steps

- Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If none exists then DEADLOCK
- Mark the process as terminated and add all its resources to A
- Repeat steps 1 and 2 until either all processes are terminated in which case the initial state was safe, or until a deadlock occurs in which case it was not

Banker's Algorithm for Multiple Resources

Data Structures

The algorithm requires the operating system to maintain several data structures, where 'n' is the number of processes and 'm' is the number of resource types:

- Available: A vector of length 'm' showing the number of available instances of each resource type.
- Max: An `n x m` matrix defining the maximum demand of each process for each resource type.
- Allocation: An `n x m` matrix showing the number of resources of each type currently allocated to each process.
- Need: An `n x m` matrix indicating the remaining resource need for each process (`Need = Max - Allocation`).

Banker's Algorithm for Multiple Resources

The Algorithms: The Banker's Algorithm uses two main algorithms to operate:

1. Safety Algorithm

This algorithm determines if the current system state is safe or unsafe.

- Initialization:
    1. `Work = Available`
    2. `Finish[i] = false` for all processes `i`.
- Steps:
    1. Find an index `i` such that `Finish[i] == false` and the process's `Need[i]` is less than or equal to `Work`.
    2. If no such process is found, go to step 4.
    3. If a process is found:
        - `Work = Work + Allocation[i]` (Resources are released back to the system).
        - `Finish[i] = true` (Process is marked as finished).
        - Go back to step 1 and repeat for remaining processes.
    4. If `Finish[i] == true` for all processes, the system is in a safe state.

Banker's Algorithm for Multiple Resources

Resource Request Algorithm

This algorithm decides whether to grant a resource request made by a process.

- When a process `Pi` requests resources:
    1. Check if the request is valid (i.e., less than or equal to its `Need[i]`). If not, raise an error.
    2. Check if the requested resources are available (`Request[i] <= Available`). If not, the process must wait.
    3. If both conditions are met, the system temporarily allocates the resources and runs the Safety Algorithm to check if the new state is safe.
    4. If the new state is safe, the request is granted. If unsafe, the allocation is rolled back, and the process waits.

Banker's Algorithm for Multiple Resources: Disadvantages

- Theoretically wonderful but in practice essentially useless.
- Why?
  - Processes rarely know in advance what there maximum resource needs will be
  - The number of processes is not fixed, but dynamically varying.
  - Available resources may vanish (Broken tape drives)!!!

Deadlock Prevention

Attacking the Mutual Exclusion Condition

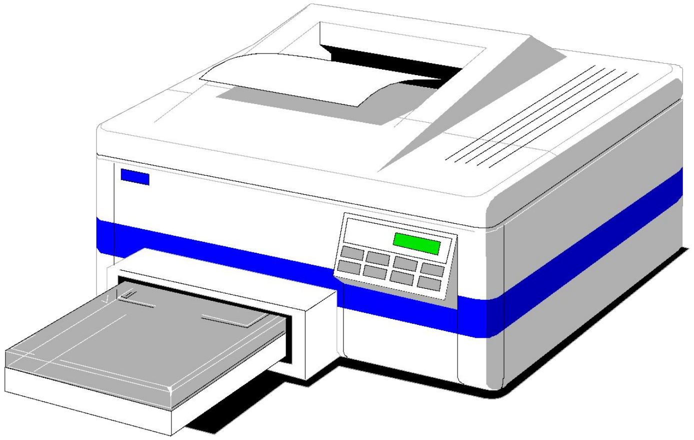
- Some devices (such as printer) can be spooled
  - only the printer daemon uses printer resource
  - thus deadlock for printer eliminated
- Not all devices can be spooled
- Principle:
  - avoid assigning resource when not absolutely necessary
  - as few processes as possible actually claim the resource

Attacking the Hold and Wait Condition

- Require processes to request 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 must give up all resources
  - then request all immediately needed

Attacking the No Preemption Condition

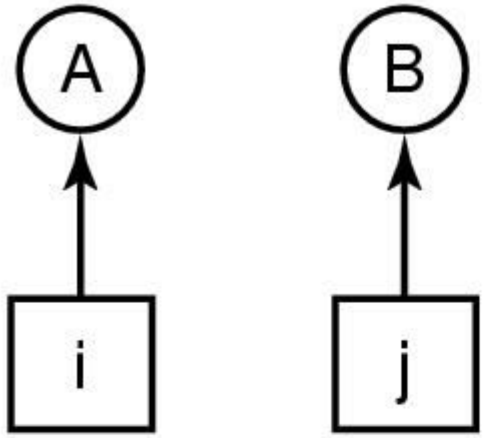
- This is not a viable option
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



Attacking the Circular Wait Condition (1)

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

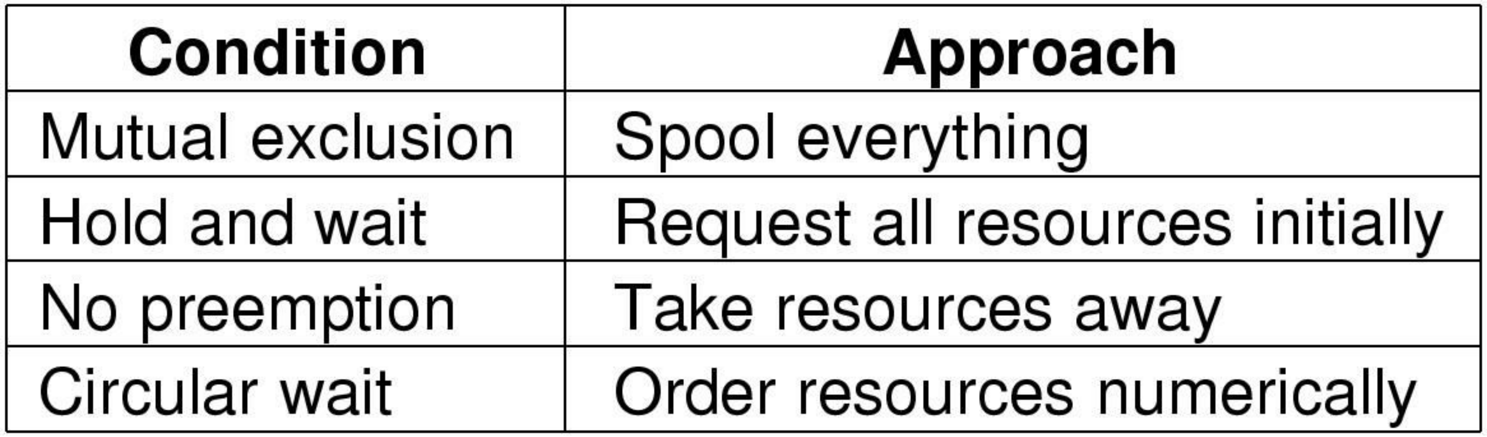
(a)



(b)

- Normally ordered resources
- A resource graph

Attacking the Circular Wait Condition (1)



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

Summary of approaches to deadlock prevention

Other Issues

Two-Phase Locking

- mainly for DBMS
- 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

Nonresource Deadlocks

- Possible for two processes to deadlock
  - each is waiting for the other to do some task
- Can happen with semaphores
  - each process required to do a *down()* on two semaphores (*mutex* and another)
  - if done in wrong order, deadlock results

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-serve policy