DEADLOCKS

MAN, August 2015

RESOURCE

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

RESOURCE

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

RESOURCE

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

RESOURCE

- 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, Printer.
- Generally Deadlocks involve exclusive access to non-premtable resources

RESOURCES

- Sequence of events required to use a resource
    - request the resource
    - use the resource
    - release the resource
- If requested resource is not available the requesting process is blocked

RESOURCE ACCESS CONTROL USING SEMAPHORES

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

RESOURCE ACCESS CONTROL USING SEMAPHORES

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

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**

INTRODUCTION TO DEADLOCKS

- None of the deadlocked processes can …
  - run
  - release resources
  - be awakened
- Permanent blocking
- A law passed by the Kansas legislature early in the 20th century
  - "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

Mutual exclusion condition

each resource assigned to exactly 1 process or is available

Hold and wait condition

process holding resources can request additional resources

No preemption condition

previously granted resources cannot forcibly taken away

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

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

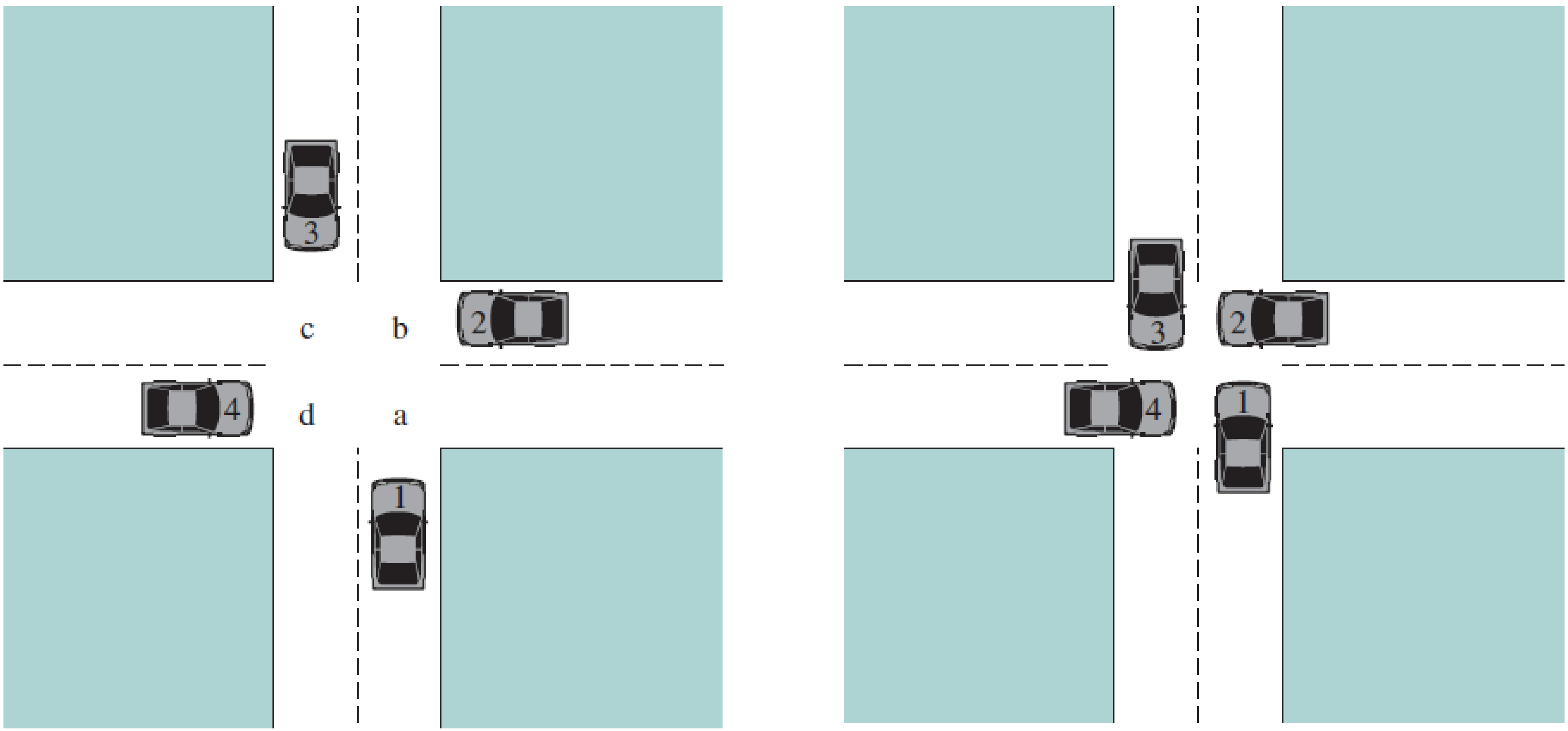
…

T_n is waiting for a resource that is held by T_1

FOUR CONDITIONS FOR DEADLOCK

- The fourth condition is a potential consequence of the first three
  - Given that the first three conditions exist, a sequence of events may occur that lead to an unresolvable circular wait.
  - The unresolvable circular wait is in fact the definition of deadlock.
  - The circular wait listed as condition 4 is unresolvable because the first three conditions hold.
- Thus, the four conditions, taken together, constitute necessary and sufficient conditions for deadlock
- If one of them is absent, no deadlock is possible

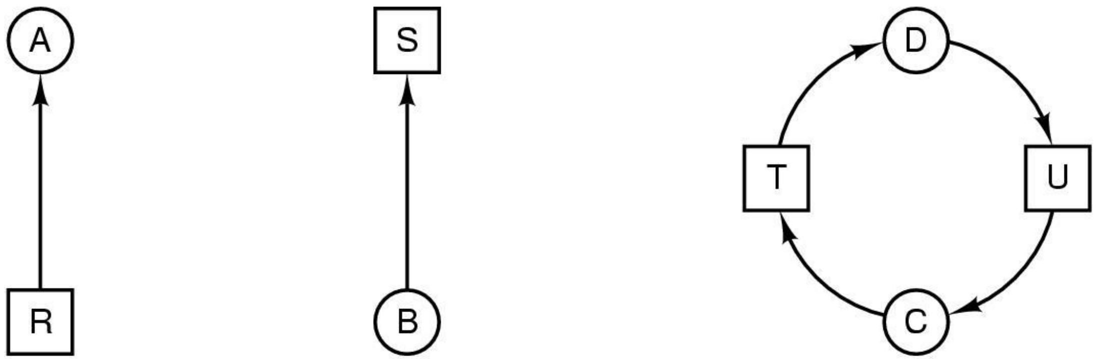
Possibility of Deadlock	Existence of Deadlock
1. Mutual exclusion	1. Mutual exclusion
2. No preemption	2. No preemption
3. Hold and wait	3. Hold and wait
	4. Circular wait



(a) Deadlock possible (b) Deadlock

DEADLOCK MODELING

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

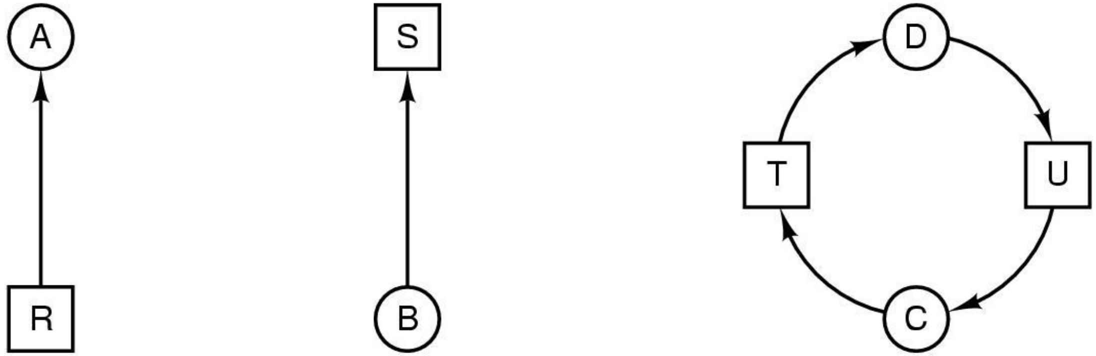


(a) (b) (c)

  - 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

- 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.



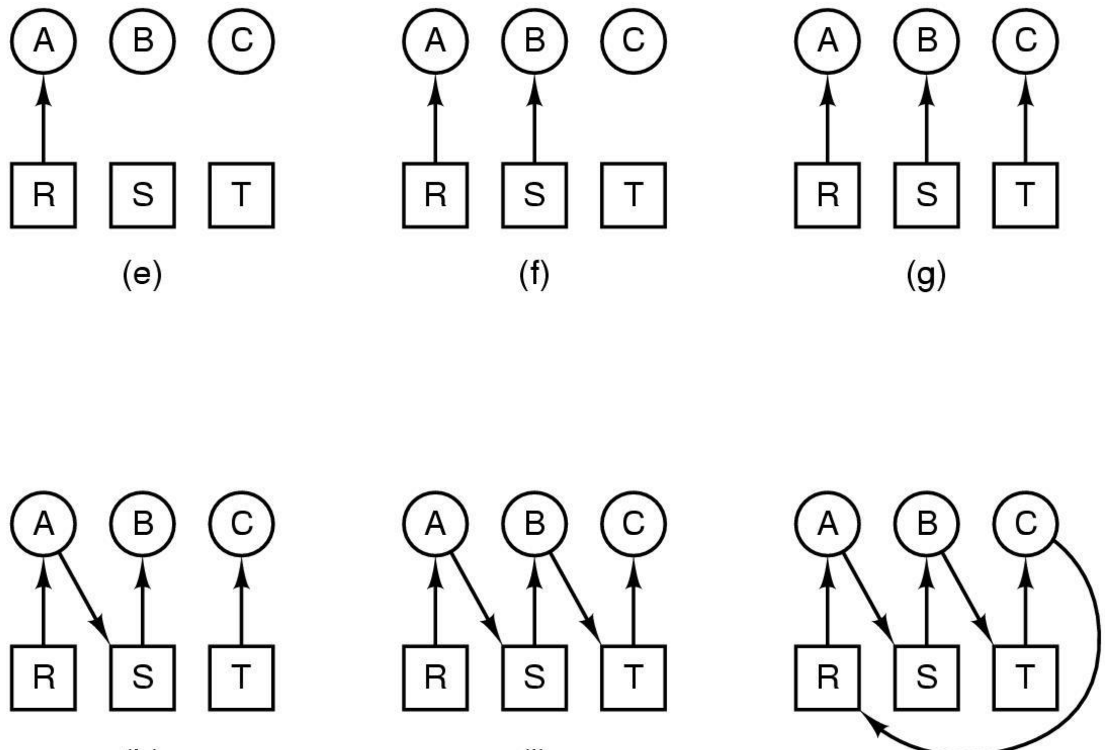
(a) (b) (c)

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)



(e) (f) (g)

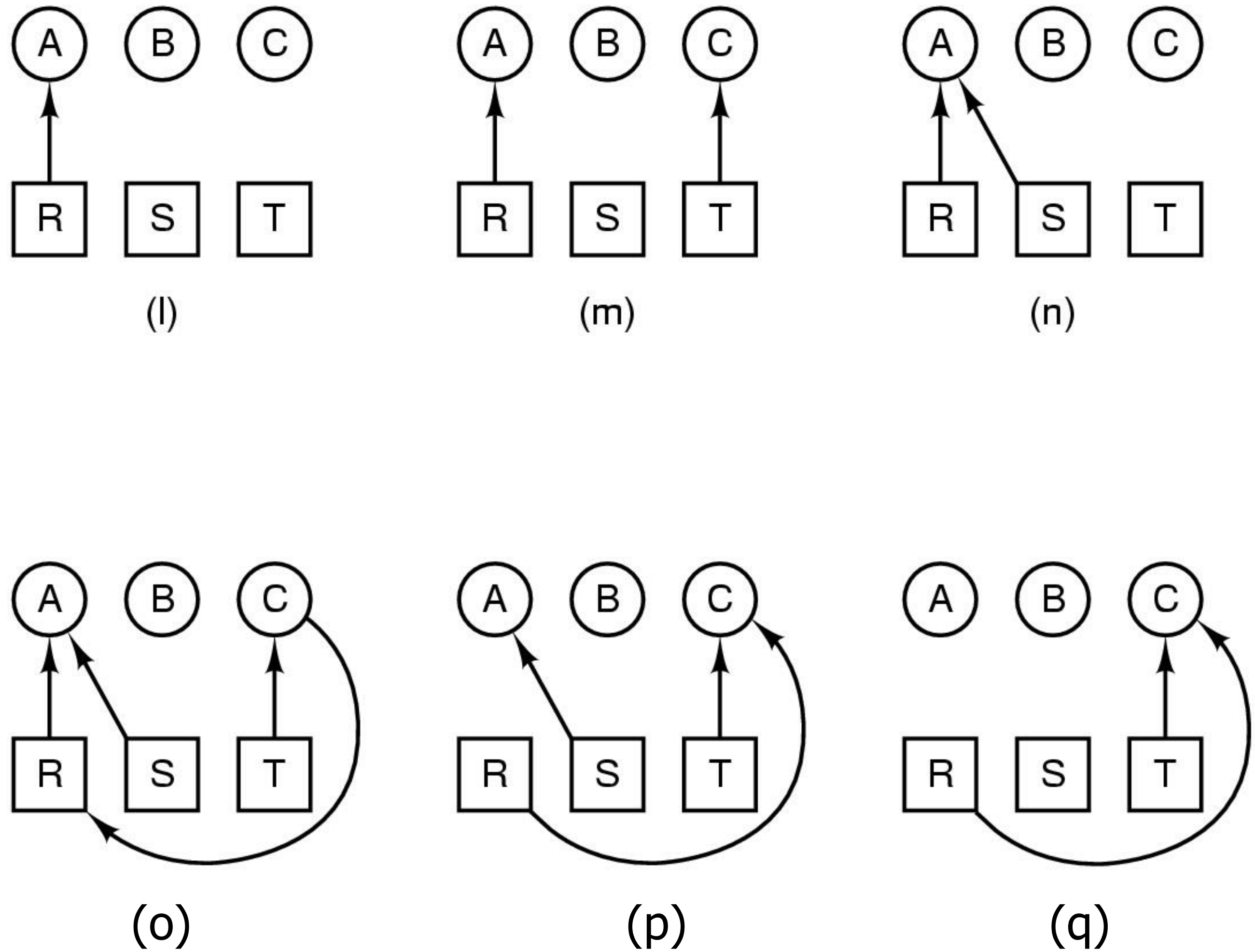
(h) (i) (j)

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)



How deadlock can be avoided

DEADLOCK MODELING

- 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

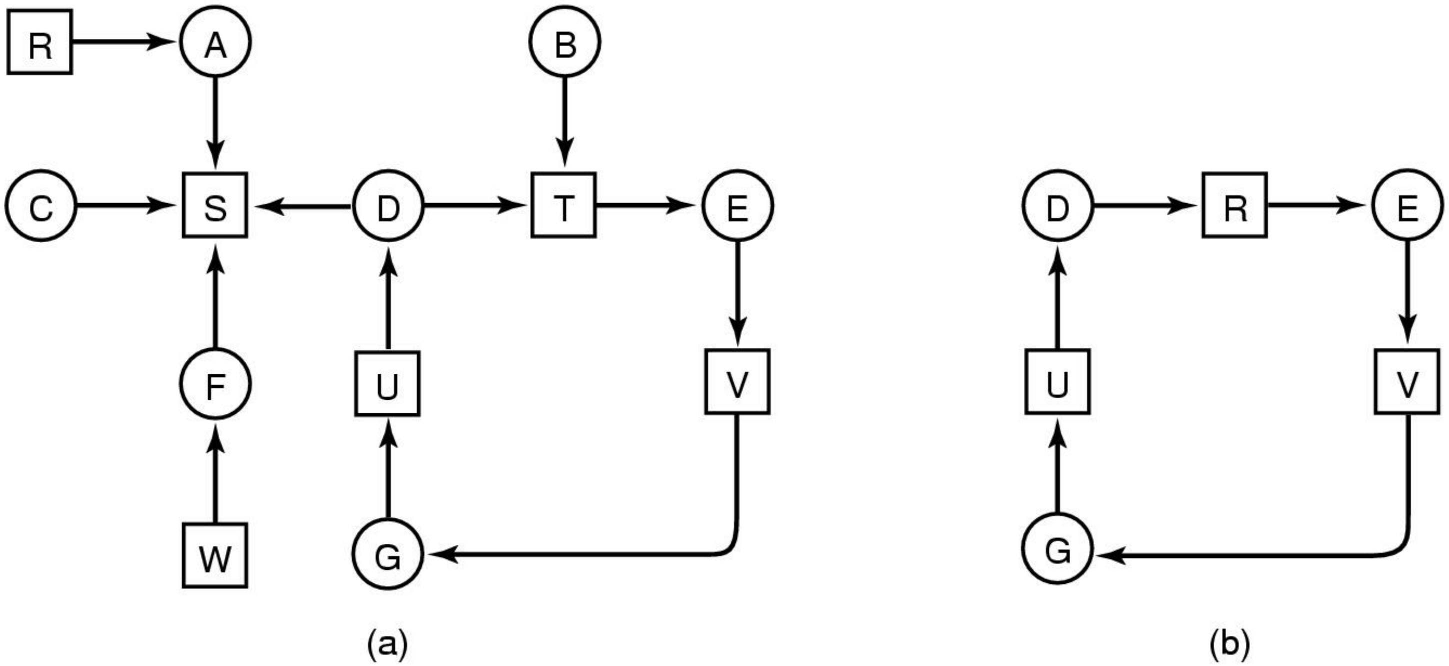
- Strategies for dealing with Deadlocks
  - Just ignore the problem altogether
  - detection and recovery
    - Let deadlocks occur, detect them, and take action.
  - Dynamic avoidance
    - careful resource allocation
  - Prevention
    - negating one of the four necessary conditions

THE OSTRICH ALGORITHM

- The simplest approach
- Don’t do anything, **simply restart** the system
- 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, Linux and Windows takes this approach
- It is a trade off between
  - convenience
  - correctness

DETECTION WITH ONE RESOURCE OF EACH TYPE

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



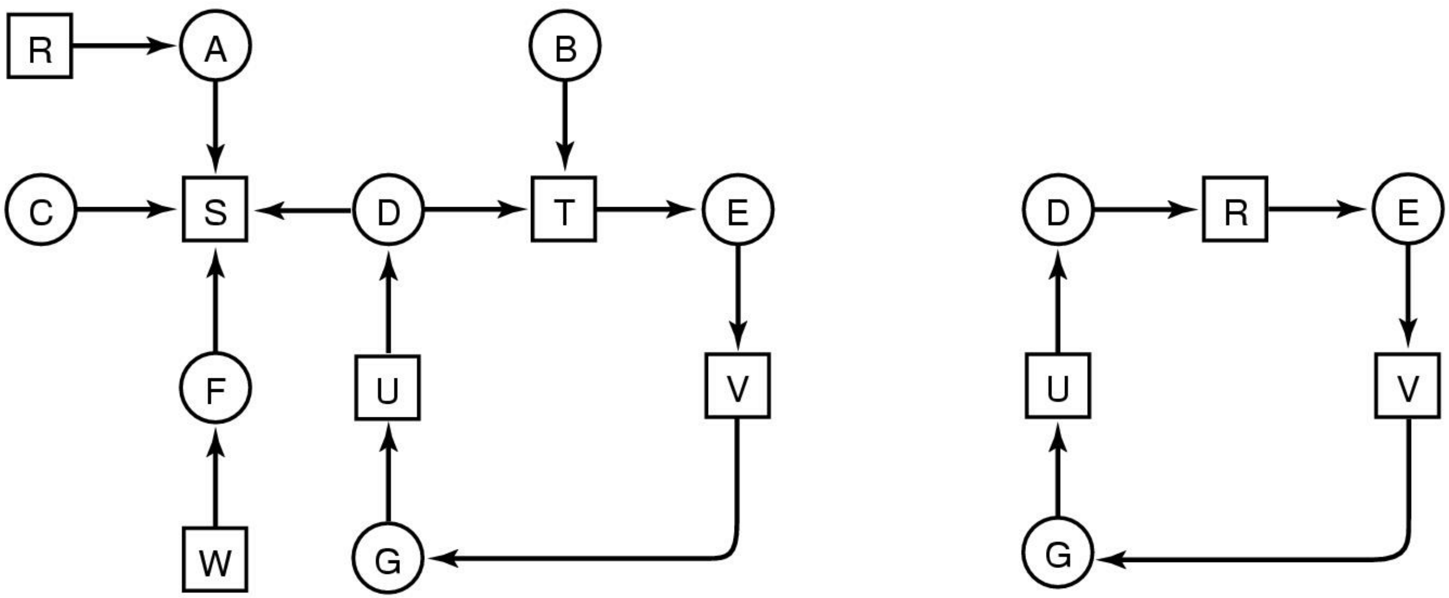
DETECTION WITH ONE RESOURCE OF EACH TYPE

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.
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 ONE RESOURCE OF EACH TYPE

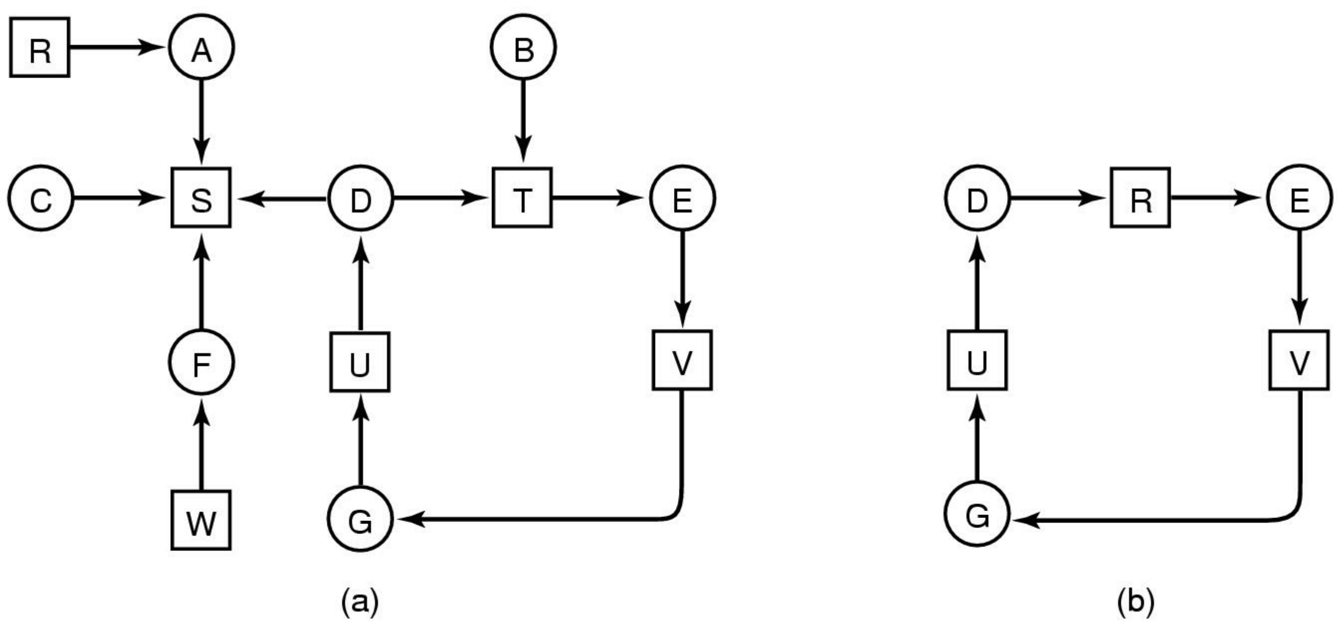
- The order of processing the nodes is arbitrary
- let us just inspect them from left to right, top to bottom
  - starting at R, then successively A, B, C, S, D, T, E, F…
  - If we hit a cycle, the algorithm stops.



(a) (b)

DETECTION WITH ONE RESOURCE OF EACH TYPE

- From B we continue to follow outgoing arcs until we get to D, at which time L = [B, T, E, V, G, U, D].
  - Now we must make a random choice
  - If we pick S we come to a dead end and backtrack to D
  - The second time we pick T
  - Update L to be [B, T, E, V, G, U, D, T]



DETECTION WITH MULTIPLE RESOURCE OF EACH TYPE

Data structures needed by deadlock detection algorithm

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

DETECTION WITH MULTIPLE RESOURCE OF EACH TYPE

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

DETECTION WITH MULTIPLE RESOURCE OF EACH TYPE

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

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

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

WHEN TO RUN DETECTION ALGORITHM?

- Every time a resource request is made
  - Can detect deadlock as early as possible
  - Expensive in terms of CPU time
- When CPU utilization drops below a threshold
  - If enough processes are deadlocked there will be few runnable processes

RECOVERY FROM DEADLOCK

- Recovery through preemption
  - Take a resource temporarily from the current owner process
    - Manual intervention may be needed
  - Depends on nature of the resource
- Recovery through rollback
  - Checkpoint a process periodically
  - use this saved state
  - Rollback a process to an earlier time if needed

RECOVERY FROM DEADLOCK

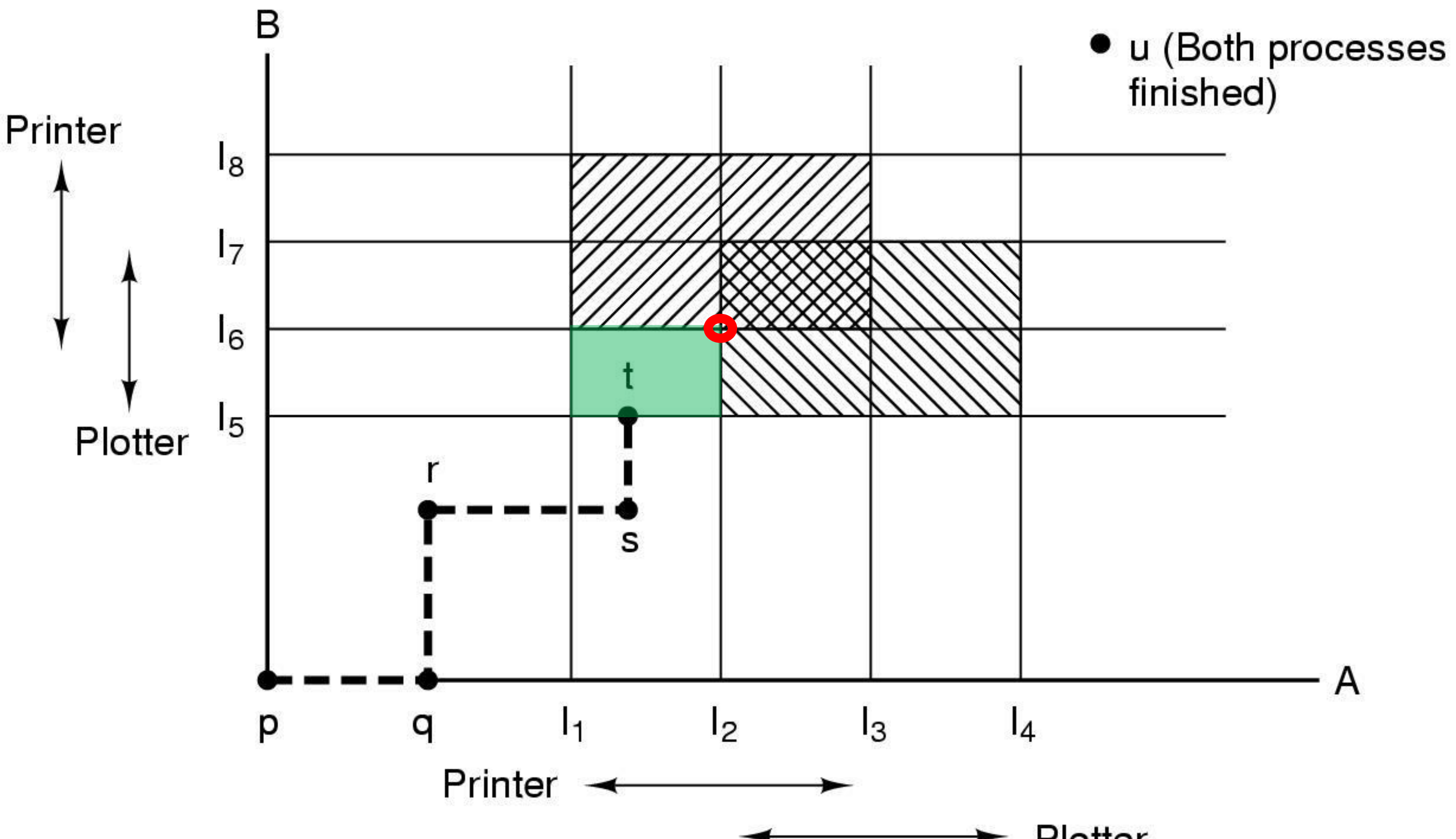
- Recovery through killing processes
  - Crudest but simplest way to break a deadlock
  - Kill one of the processes in the deadlock cycle
  - The other processes get its resources
  - Alternatively, one of the processes not deadlocked may be killed
  - Choose process for killing that can be rerun from the beginning with no ill effect

DEADLOCK AVOIDANCE

- So far we assumed that a process asks for all the resources it needs at once
- In most systems, resources are requested one at a time
- To avoid deadlock the system must be able to decide
  - Whether granting a request is safe or not
  - Only make the allocation when it is safe
- But we need some information **in advance**

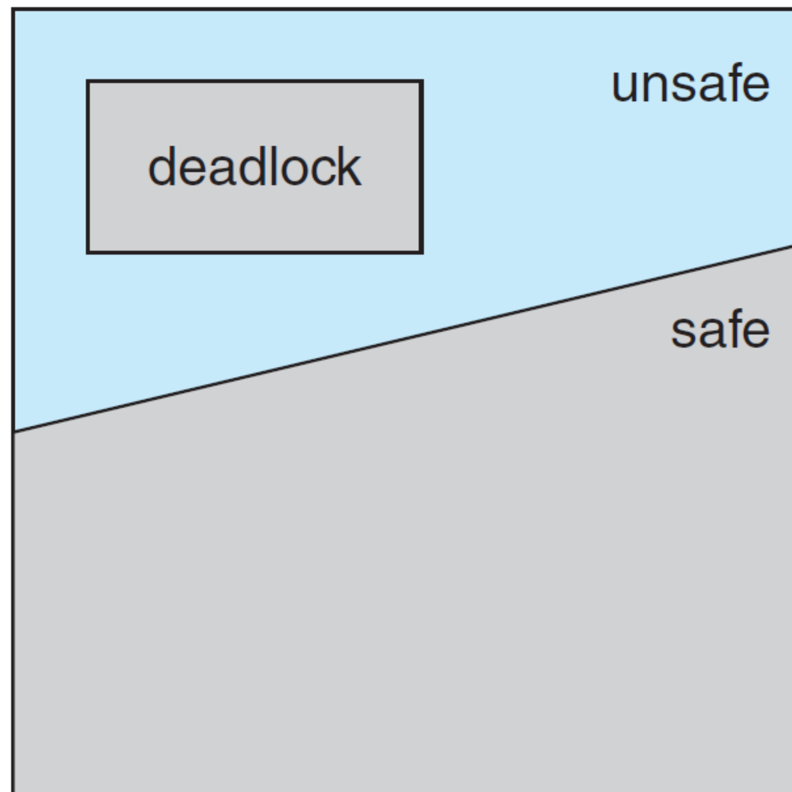
CONCEPT OF SAFETY

Two process resource trajectories



SAFE AND UNSAFE STATES

- A state is safe if
  - there is some scheduling order in which
    - every process can complete even if
      - all of them suddenly request their maximum number of resources immediately
- An unsafe state is not a deadlocked state
- Safe Vs. Unsafe
  - A safe state guarantees that all processes will finish



SAFE AND UNSAFE STATES

- One Resource Case
- Total 10 instances available

(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

SAFE AND UNSAFE STATES

	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
- The allocation decision that moved the system from (a) to (b) went from a safe state to unsafe state
- **NOTE:** An unsafe state is not a deadlocked state
- Safe Vs Unsafe
  - A safe state guarantees that all processes will finish

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
  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

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 STEPS:

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A
2. Mark the process as terminated and add all its resources to A
3. Repeat steps 1 and 2 until
   - either all processes are terminated in which case the initial state was safe
   - or no process is left whose needs can be met in which case the initial state was unsafe

BANKER'S ALGORITHM: DISADVANTAGES

- Theoretically wonderful but in practice essentially useless
- Why?
  - Processes rarely know in advance what there maximum resource needs will be