Deadlocks

Prof. James L. Frankel
Harvard University

Version of 4:38 PM 4-Oct-2022

Introduction to Deadlocks

- Computer resources
    - Files
    - Database records
    - Fields in Internal Tables
    - Printers
    - Tape drives
- Processes need access to resources in reasonable order
- Example of a deadlock
    - Process 1 holds resource A and requests resource B
    - At the same time, process 2 holds resource B and requests A
    - Both processes are blocked and neither can make progress

Resources (1 of 2)

- Deadlocks can occur through a chain of exclusive access grants and requests
- Preemptable resources
  - Can be taken away from a process with no ill effects
- Nonpreemptable resources
  - Will cause the process to fail if taken away

Resources (2 of 2)

- Sequence of events required to use a resource
  1. Request the resource
  2. Use the resource
  3. Release the resource

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

Resource Acquisition: Deadlock-free

```
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 Acquisition: Potential deadlock

```
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_2);
  down(&resource_1);
  use_both_resources();
  up(&resource_1);
  up(&resource_2);
}
```

Introduction to Deadlocks

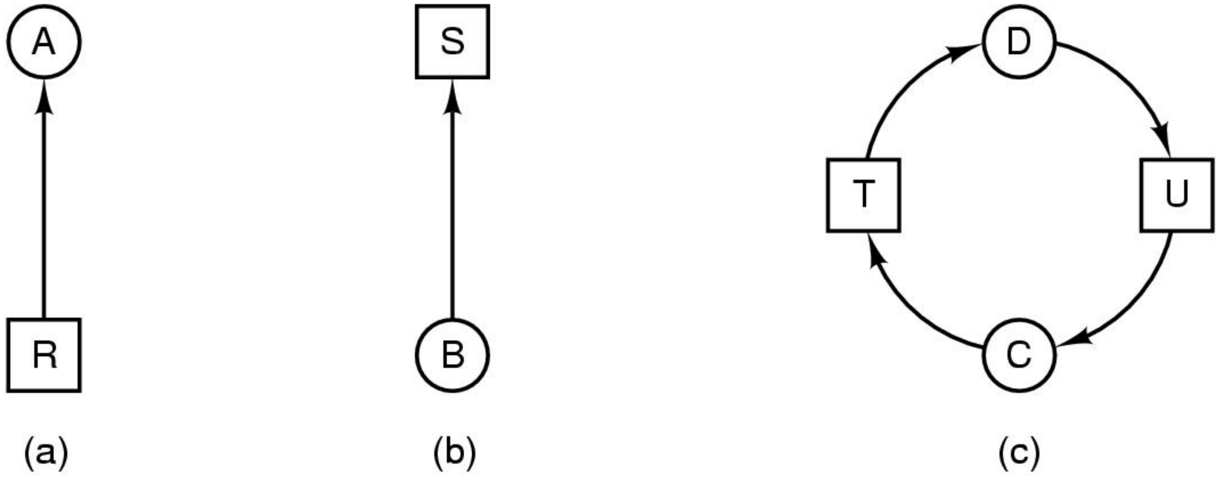
- Formal definition:
  A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Usually the event is the release of a currently held resource
- None of the processes in the deadlock chain are able to
  - Run
  - Release resources
  - Be awakened

Four Conditions for Deadlock

1. Mutual exclusion
    - Each resource is assigned to a single process or is available
2. Hold and wait
    - Processes can hold resources then request more resources
3. No preemption
    - Previously granted resources cannot be forcibly taken away
4. Circular wait
    - Must be a circular chain of two or more processes
    - Each process is waiting for resource held by the next member of the chain

Deadlock Modeling (1 of 3)

- Modeled with directed graphs called **Resource Allocation Graphs**
- Squares are resources and circles are processes



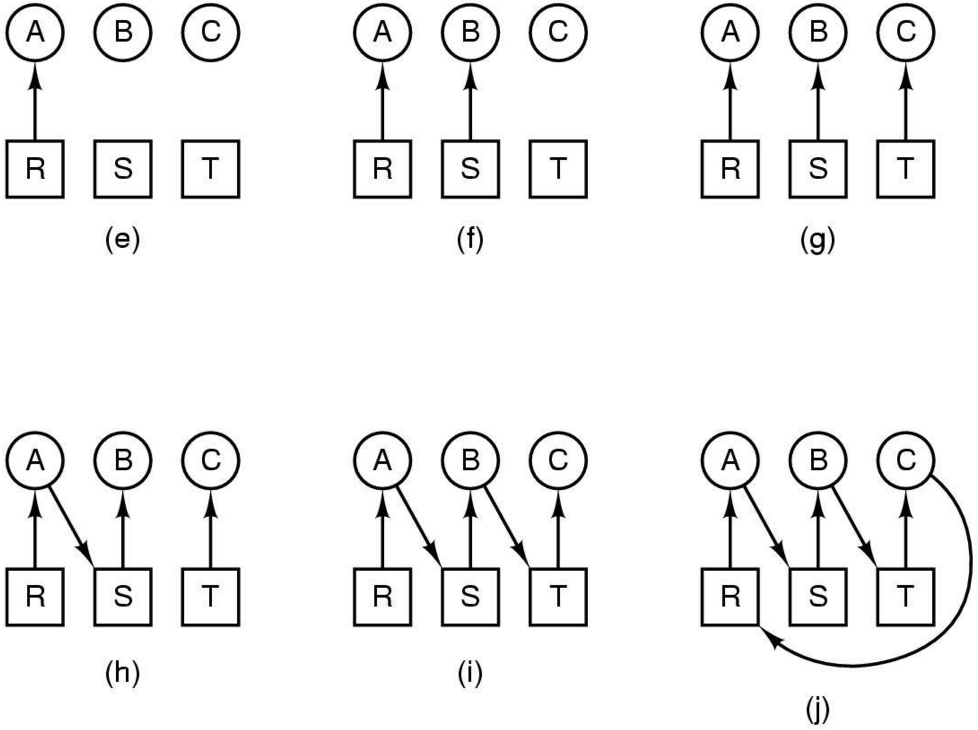
  - a: resource R is being **held by** process A
  - b: process B is **requesting/waiting** for resource S
  - c: processes C and D are in a deadlock over resources T and U

Deadlock Modeling (2 of 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)

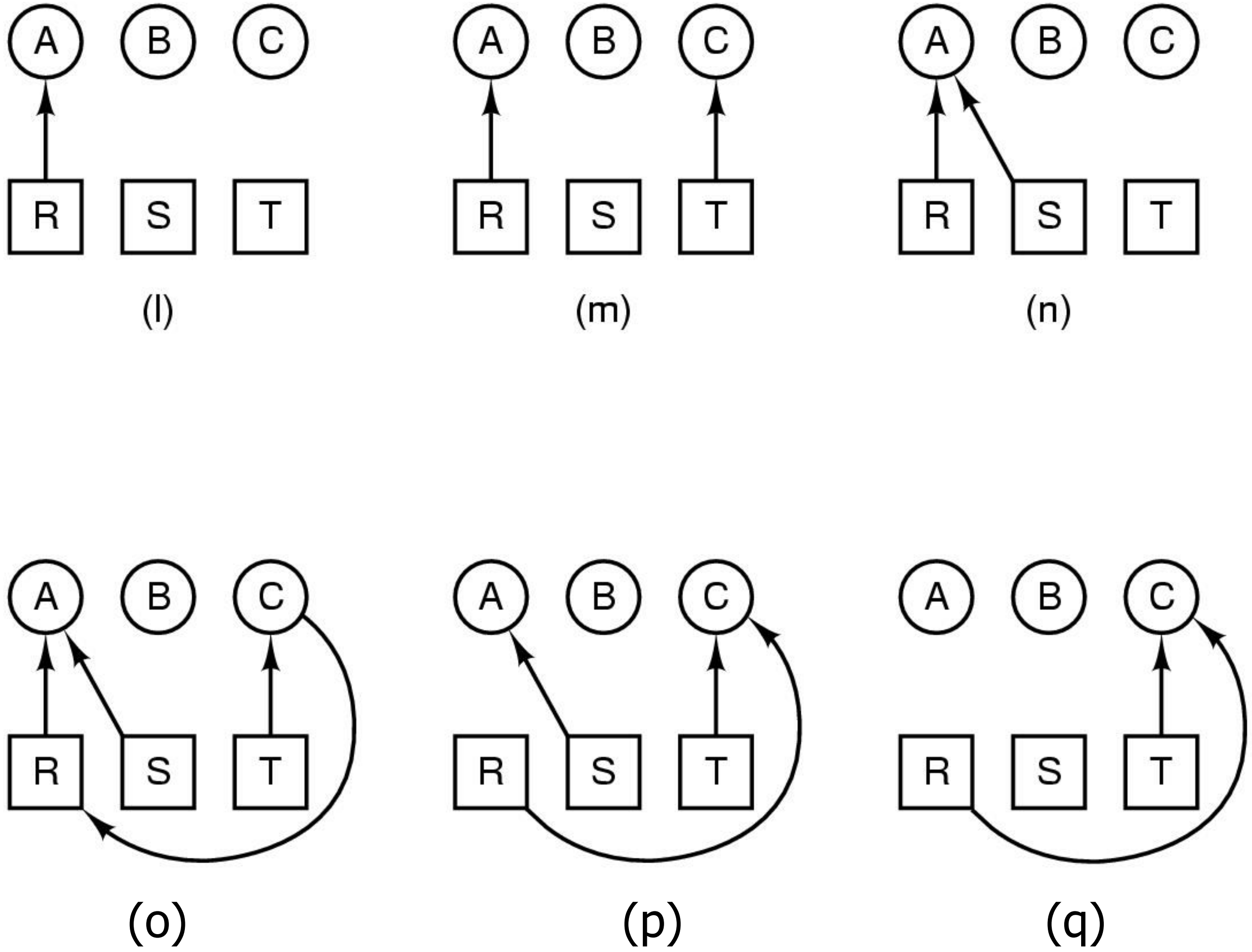
This ordering results in a deadlock

Deadlock Modeling (3 of 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)



This ordering avoids a deadlock

Deadlock Strategies

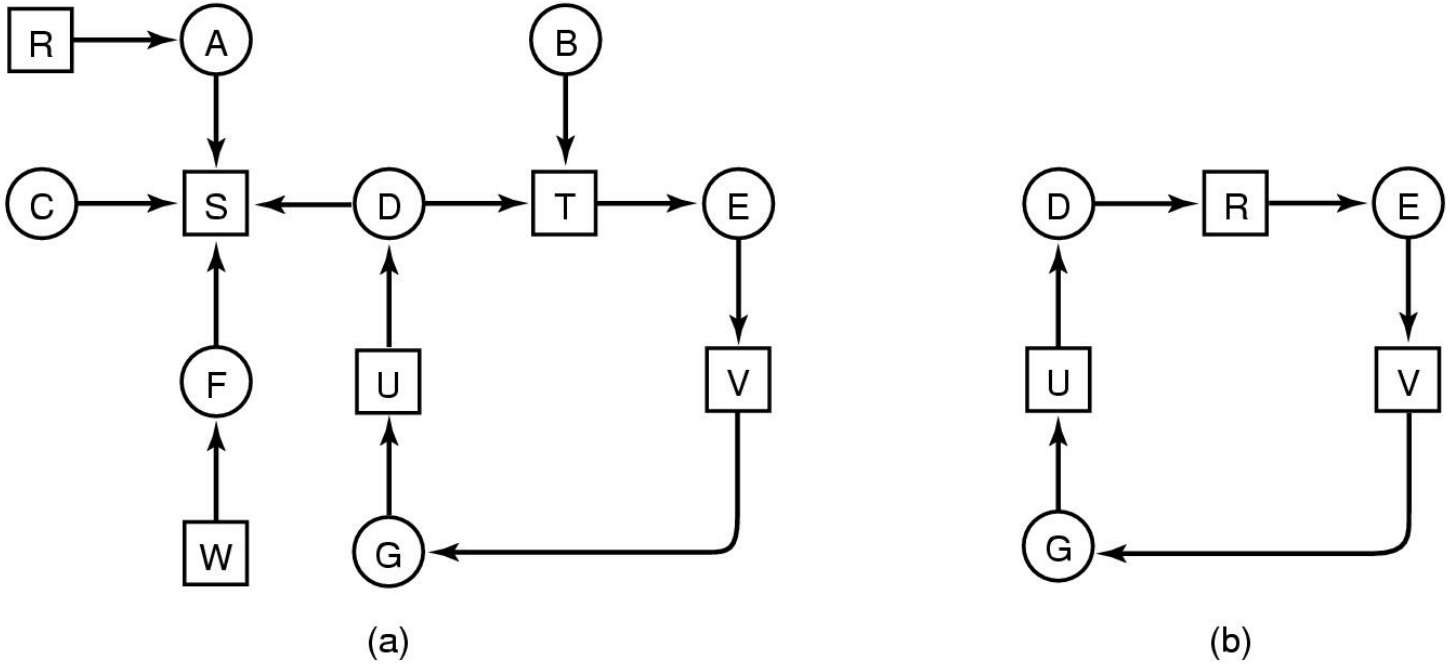
Four approaches to deal with deadlocks

1. *Ignore* the problem – follow the so-called “Ostrich Algorithm”
2. *Detect and recover* from a deadlock
3. Dynamically *avoid* deadlocks
    - Carefully allocate resources
4. *Prevent* deadlocks from occurring
    - Negate at least one of the four necessary conditions

Ignore the Problem: The Ostrich Algorithm

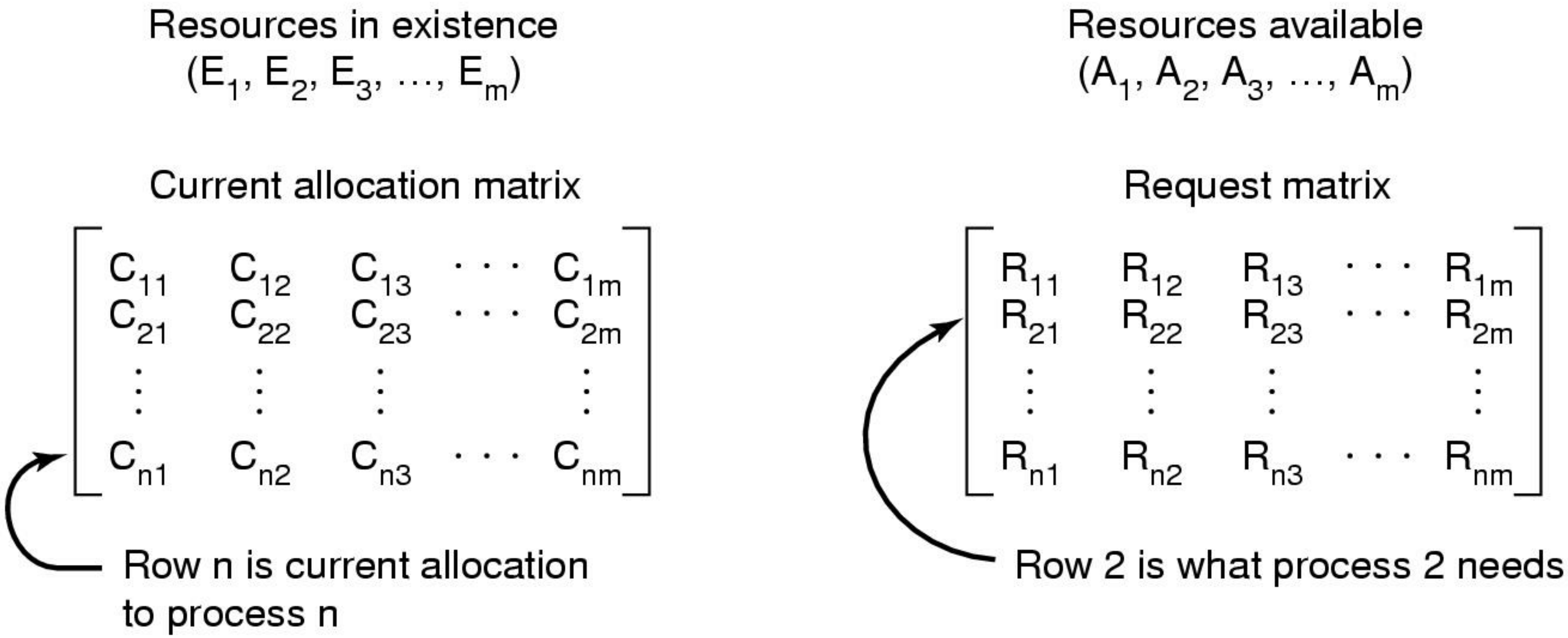
- Pretend there is no problem
- Reasonable in some circumstances
  - When deadlocks occur very rarely
  - When cost of prevention is high
- Some aspects of UNIX and Windows OSes take this approach
- Trade off between
  - Convenience
  - Correctness

Detection with One Resource of Each Type



- Resource graph denotes resource ownership and requests
- If a **cycle** can be found within the graph, then a deadlock has been identified

Detection w/Multiple Resources of Each Type



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

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

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

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

$$\Sigma(i=1 \text{ to } n)\ C_{ij} + A_j = E_j$$

Detection w/Multiple Resources of Each Type

Deadlock Detection Algorithm

Assume a worst case scenario: that processes keep all acquired resources until they exit

1. Start with all processes unmarked
2. Look for an unmarked process, P_i, for which the i-th row of R is less than or equal to A (for all elements)
3. If such a process is found, add the i-th row of C to A, mark process P_i and go back to step 2
4. If no process P_i exists, the algorithm terminates
5. When the algorithm terminates, all unmarked processes (if any exist) are deadlocked

Detection w/Multiple Resources of Each Type

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array} \qquad A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \end{array}$$

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

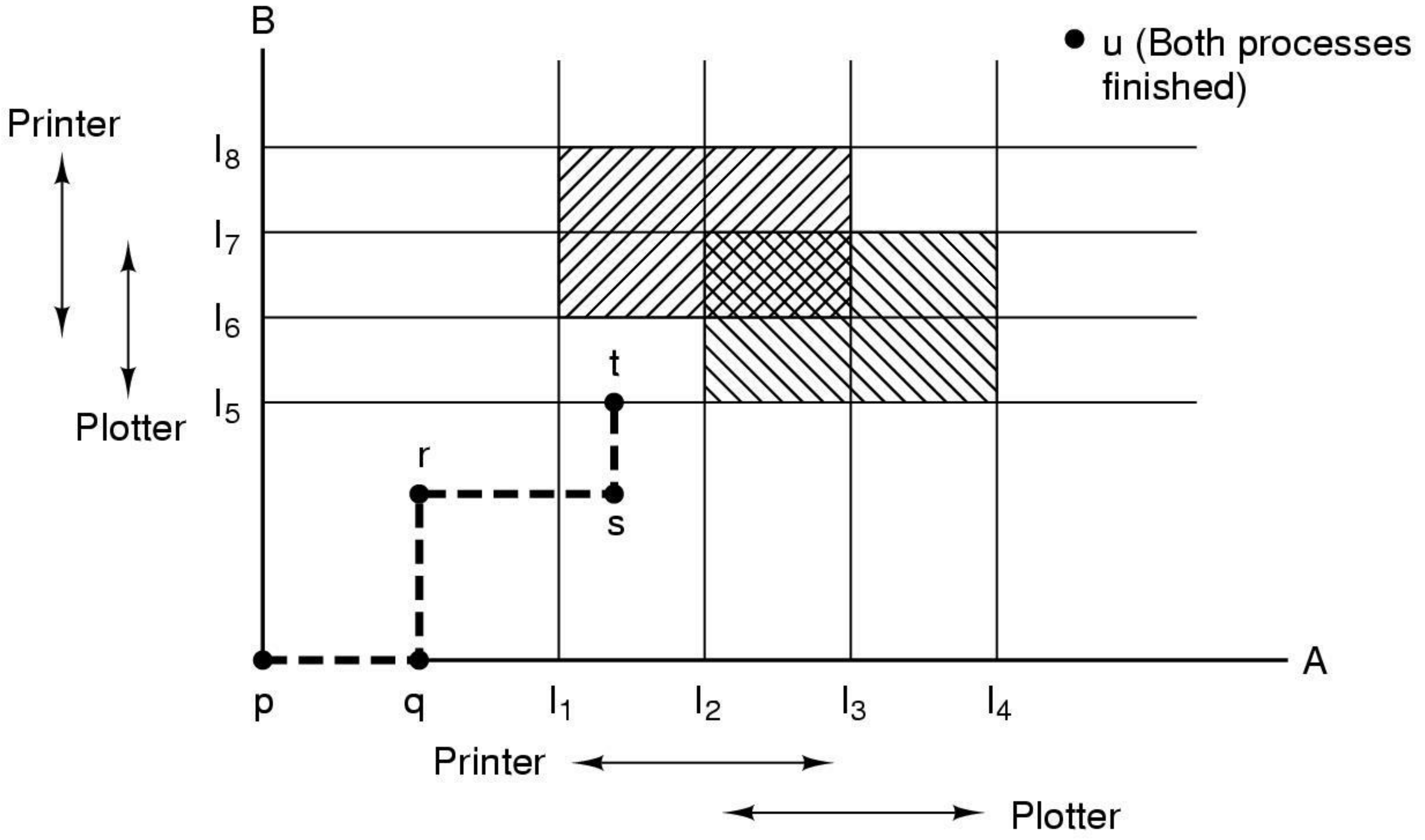
Recovery from Deadlock (1 of 2)

- Recovery through *preemption*
  - Take a resource from some process to break the deadlock
  - Whether this is possible depends on the nature of the resource
- Recovery through *rollback*
  - Checkpoint processes on a periodic basis
  - If a deadlock occurs, roll back some process to a saved state where it did not yet acquire a needed resource

Recovery from Deadlock (2 of 2)

- Recovery through *killing* processes
  - Crudest but simplest way to break a deadlock
  - Kill one of the processes in the deadlock cycle
  - Other processes get its resources
  - Choose a process that can be rerun with no ill effects

Deadlock Avoidance: Resource Trajectories



Resource trajectories of two processes

The rectangle bounded by I_5 & I_6, I_1 & I_2 is **unsafe**

Safe States

- A state is **safe** if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of remaining resources immediately

Unsafe States

- An **unsafe** state does not mean that a system is currently deadlocked
- A system can continue to run in a unsafe state, but it *may* eventually lead to a deadlock
- If a system is in a safe state, it is guaranteed that the system will allow all processes to eventually complete successfully – that is, no deadlock *can* occur from a safe state

Safe and Unsafe States (1 of 2)

	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)

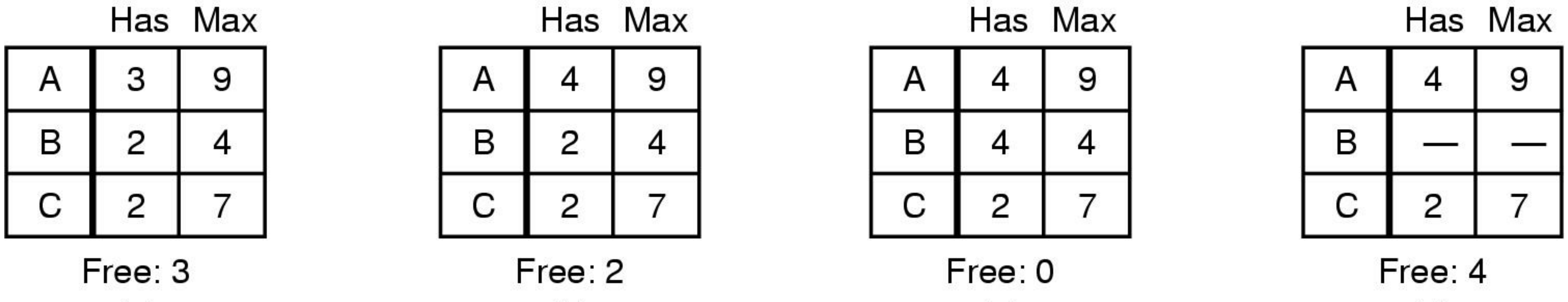
	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

Demonstration that the state in (a) is safe

Safe and Unsafe States (2 of 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)

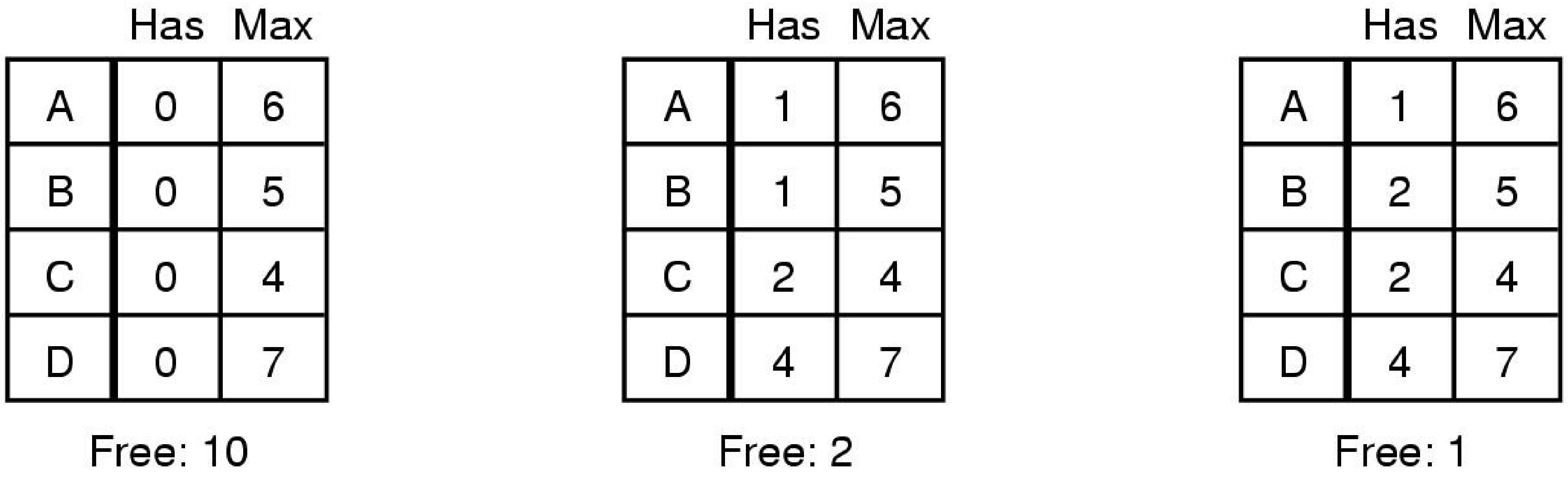
	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

Demonstration that the state in (b) is not safe

The Banker's Algorithm for a Single Resource



(a)

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

Free: 10

(b)

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

Free: 2

(c)

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

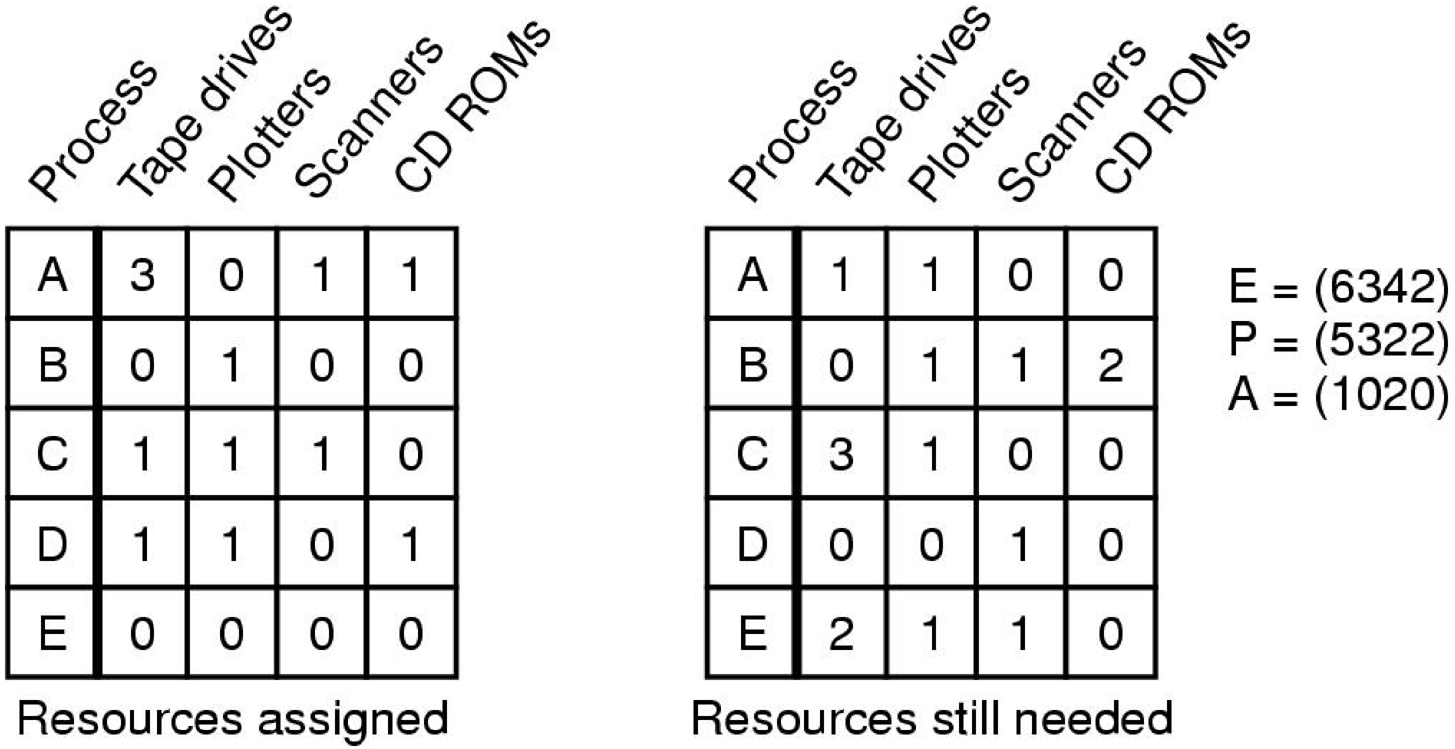
Free: 1

- Example of resource allocation states
  - (a) is safe
  - (b) is safe
  - (c) is unsafe

The Banker's Algorithm

- Small-town banker’s actions
  - Grant lines of credit to customers
  - If granting a loan request leads to an unsafe state, the request is denied
  - If granting a loan request leads to a safe state, the request is carried out
  - A state is safe if the banker has enough resources to satisfy some customer
    - If so, then those funds are assumed to be repaid
    - Next, the customer now closest to the limit is checked and the algorithm repeats
    - If all loans can eventually be repaid, then the state is safe

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

The Banker's Algorithm for Multiple Resources

- Look for any row, R, whose unmet resource needs are <= A. If none is found, then system will eventually deadlock.
- Assume the process for row R requests and releases all its resources. Mark that process as terminated and add its resources to the A vector.
- Repeat the two steps above until all processes terminate – then the initial state was safe – or no eligible row is found – then the initial state was unsafe.

Shortcomings of the Banker's Algorithm

- Processes rarely know what their maximum resource needs are
- The number of processes changes dynamically
- Resources can become unavailable – a resource can break
- Processes may have to wait too long for their needed resources to be released

Deadlock Prevention: Attacking the Mutual Exclusion Condition

- Some devices (such as printer) can be spooled
    - Only the printer daemon uses printer resource
    - Deadlock for printer eliminated via spooling
- Not all devices can be spooled
- Principle
    - Virtualize the resource
    - Avoid assigning resource when not absolutely necessary
    - As few processes as possible actually claim the resource

Deadlock Prevention: Attacking the Hold and Wait Condition

- Require processes to request all resources before starting execution
  - A process never has to wait for what it needs

- Problems
  - May not know required resources at start of execution
  - Ties up resources that other processes could be using

- Variation
  - Each process must give up all resources before requesting any additional resources
  - Then, the process can request all currently needed resources

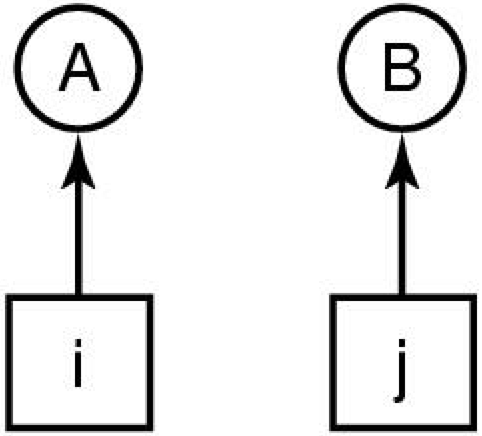
Deadlock Prevention: Attacking the No Preemption Condition

- This is not a viable option
- Consider a process given the printer
    - Halfway through its job
    - Forcibly take away printer
- But, spooling/virtualization effectively allows preemption

Deadlock Prevention: Attacking the Circular Wait Condition

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

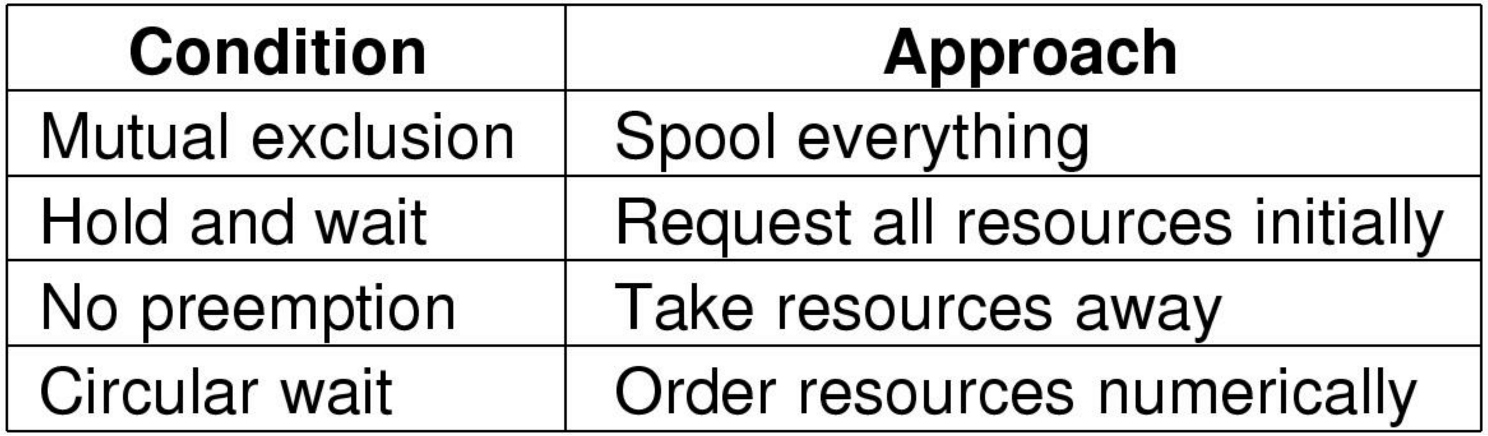
(a)



(b)

- Numerically ordered resources
- A resource graph
- Requests must be made in numerical order

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 Approaches: 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 – locks are acquired
- When Phase One succeeds, start second phase
  - Read data, performing updates
  - Release locks
- Similar to requesting all resources at once

Nonresource Deadlocks

- Possible for two processes to deadlock without a resource
  - Each process is waiting for the other to do some task; for example, for communication to occur
    - Timeouts may help to resolve this deadlock
- Can happen with semaphores
  - Each process required to do a *down()* on two semaphores (*mutex* and another)
  - If done in wrong order, deadlock results
- Livelock
  - Busy waiting rather than deadlocking, but otherwise equivalent

Starvation

- If algorithm is to allocate a resource to the shortest job first, this can cause indefinite starvation
- Works great for multiple short jobs in a system
- May cause a long job to be postponed indefinitely even though it is not blocked
- Solution
    - First-come, first-served policy