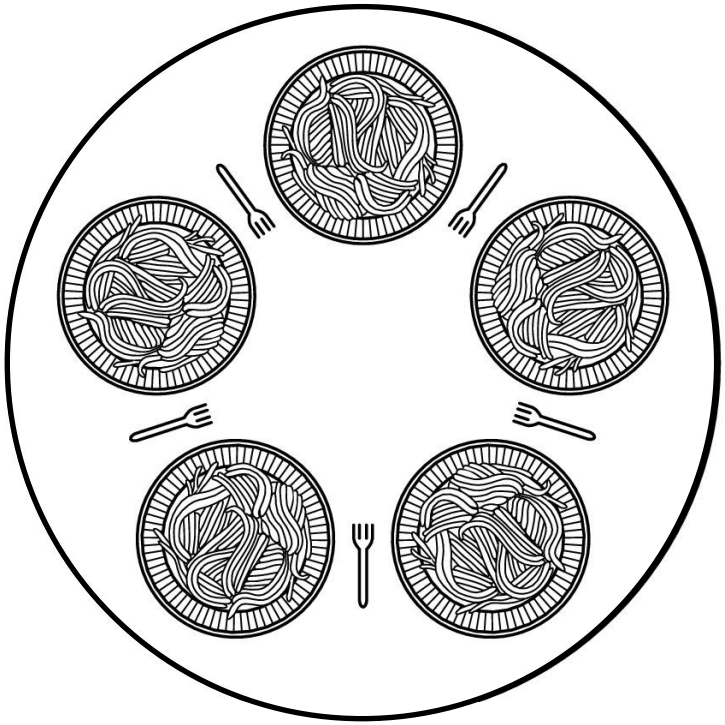
Concurrency and Deadlocks

Learning Outcomes

- Understand what deadlock is and how it can occur when giving mutually exclusive access to multiple resources.
- Understand the related concept of livelock.
- Understand several approaches to mitigating the issue of deadlock in operating systems.
  - Including deadlock *prevention*, *detection and recovery*, and deadlock *avoidance*.

Recap: Dining Philosophers

- Philosophers eat/think.
- Eating needs 2 forks.
- They pick one fork at a time.
- How do we prevent deadlock?



Dining Philosophers

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

A nonsolution to the dining philosophers problem

Resources & Deadlocks

- Suppose a process holds resource A and requests resource B.
  - At same time another process holds B and requests A.
  - Both are now blocked and remain so – *Deadlocked*.

- Deadlocks occur when …
  - processes are granted exclusive access to devices, locks, tables, etc..
  - we refer to these entities generally as resources.

Resources

- Examples of computer resources:
  - printers
  - tape drives
  - tables in a database
  - any value shared in a critical section
- Processes need access to resources in some order.

- Preemptable resources
  - can be taken away from a process with no ill effects.
- Nonpreemptable resources
  - will cause the process to fail if taken away.

Resource Access

- To use a resource:
  1. Request the resource.
  2. Use the resource.
  3. Release the resource.

- Must wait at (2) if request is denied.
  - The requesting process may be blocked.
  - The operation may fail with error code.

Two Resource Access Patterns

```
semaphore res_1, res_2;
void proc_A() {
 down(&res_1);
 down(&res_2);
 use_both_res();
 up(&res_2);
 up(&res_1);
}
void proc_B() {
 down(&res_1);
 down(&res_2);
 use_both_res();
 up(&res_2);
 up(&res_1);
}
```

```
semaphore res_1, res_2;
void proc_A() {
 down(&res_1);
 down(&res_2);
 use_both_res();
 up(&res_2);
 up(&res_1);
}
void proc_B() {
 down(&res_2);
 down(&res_1);
 use_both_res();
 up(&res_1);
 up(&res_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 set can:
  - Run.
  - Release any resources.

Four Conditions for Deadlock

1) Mutual exclusion condition.
   - Resources cannot be shared between processes.

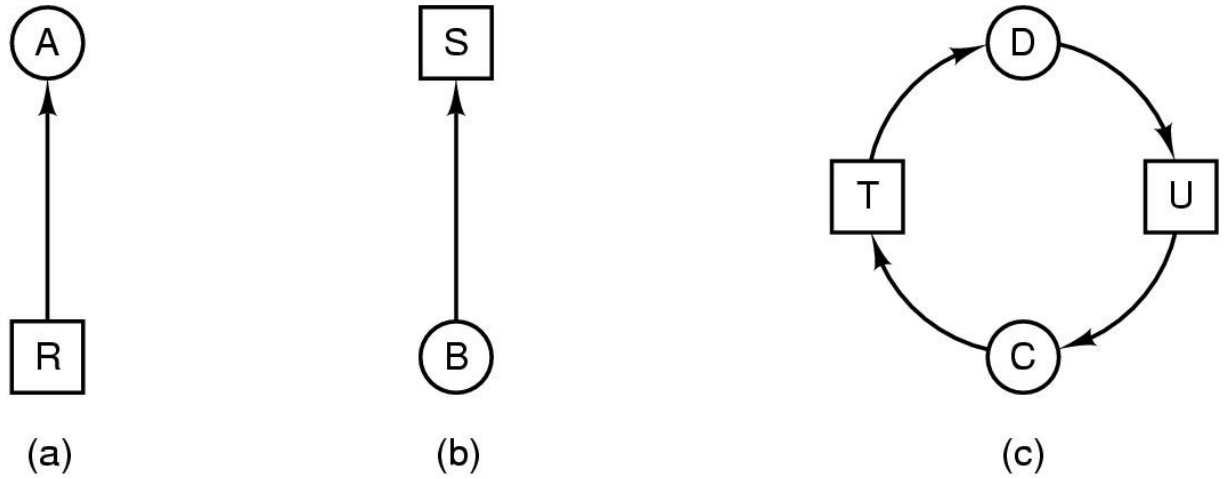
2) Hold while waiting condition.
   - A process can hold one resource while requesting another.

3) Non-preemptable resource condition.
   - Held resources remain held, and cannot be taken away.

4) Circular wait condition.
   - It must be possible to form a cycle of 2 or more waiting processes.
   - Each such process is waiting for a resource held by the next.

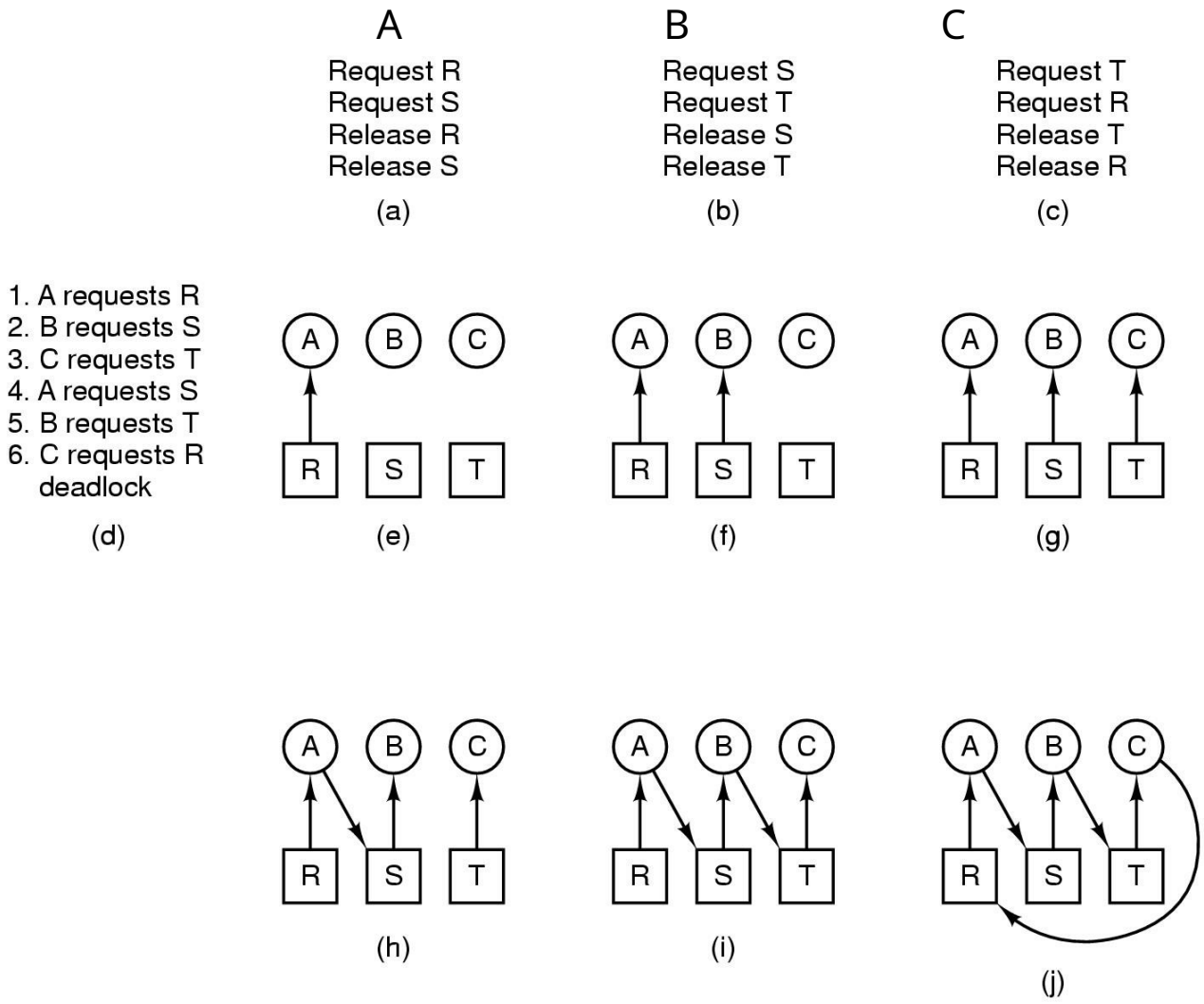
Modelling Deadlocks

- Modeled with directed graphs



- 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

Deadlock Modeling



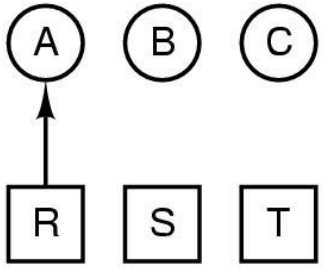
How deadlock occurs

Deadlock Modeling

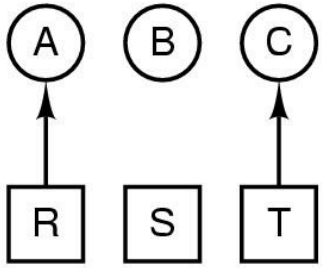
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

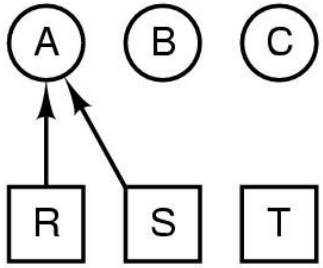
(k)



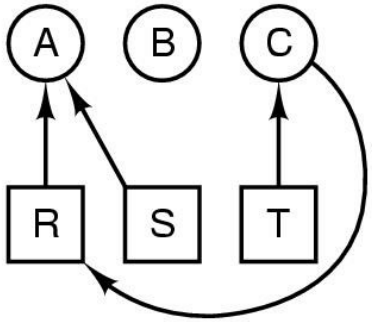
(l)



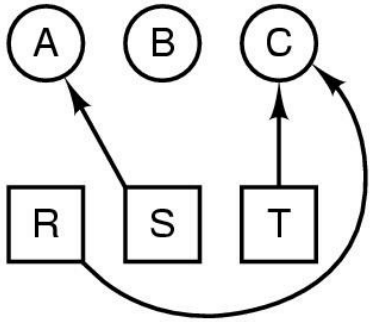
(m)



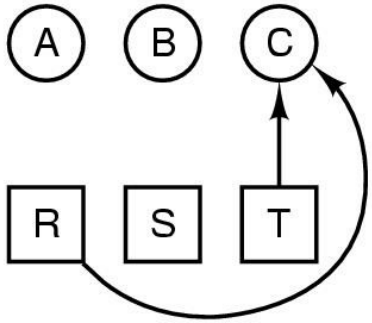
(n)



(o)



(p)



(q)

How deadlock can be avoided

Deadlock

Strategies for dealing with Deadlocks:

1) Ignore the problem altogether.
2) Prevent deadlocks.
    - Negate one of the four necessary conditions.
3) Detect and recover from deadlocks.
4) Dynamic avoidance of deadlocks.
    - Careful checks on resource allocation.

Approach 1: The “Ostrich” Algorithm

- No specific strategy for managing deadlocks.
  - Ignore the problem.
- Reasonable if:
  - deadlocks occur very rarely.
  - the cost of prevention is high.
    - Example of “cost”, only one process runs at a time
- UNIX and Windows arguably take this approach for some of the more complex resource relationships they manage.
- It’s a trade off between:
  - convenience (engineering approach).
  - correctness (mathematical approach).

Approach 2: Deadlock Prevention

- Set resource allocation rules that prevent deadlock.
- Prevent one of the four conditions required for deadlock from being true:
  - Mutual exclusion condition.
  - Hold while waiting condition.
  - Non-preemption condition.
  - Circular wait condition.

Approach 2: Deadlock Prevention

Attack the Mutual Exclusion Condition?

- Not feasible in general
  - Some devices/resource are intrinsically not shareable.

The “Hold and Wait” Condition

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

- Issues
  - may not know required resources at start of run
    - ⇒ not always possible
  - also ties up resources other processes could be using

- Variations:
  - process must give up all resources if it would block holding a resource, then re-request all that are immediately needed
    - prone to livelock

Livelock

- Livelocked processes are not blocked, they change state regularly, but they never make progress.
- Example: Two people passing each other in a corridor that attempt to step out of each other's way in the same direction, indefinitely.
  - Both are actively changing state (state = side of the corridor)
  - Both never pass each other.

Deadlock Example

```
void proc_A() {
  lock_acquire(&res_1);
  lock_acquire(&res_2);
  use_both_res();
  lock_release(&res_2);
  lock_release(&res_1);
}
```

```
void proc_B() {
  lock_acquire(&res_2);
  lock_acquire(&res_1);
  use_both_res();
  lock_release(&res_1);
  lock_release(&res_2);
}
```

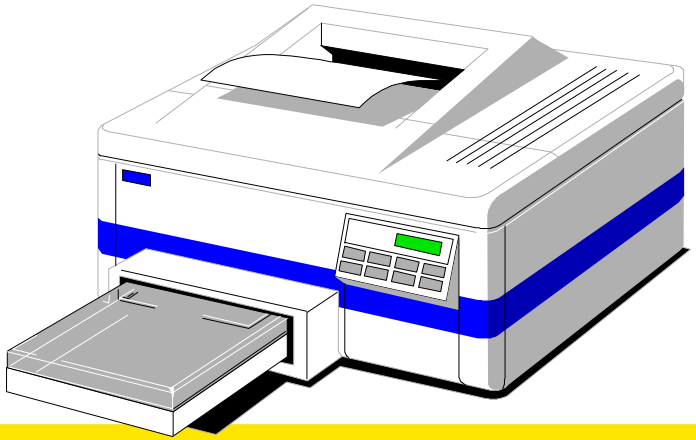
Livelock Example

```
void proc_A() {
 lock_acquire(&res_1);
 while(try_lock(&res_2) == FAIL)
 {
  lock_release(&res_1);
  wait_fixed_time();
  lock_acquire(&res_1);
 }
 use_both_res();
 lock_release(&res_2);
 lock_release(&res_1);
}
```

```
void proc_B() {
 lock_acquire(&res_2);
 while(try_lock(&res_1) == FAIL)
 {
  lock_release(&res_2);
  wait_fixed_time();
  lock_acquire(&res_2);
 }
 use_both_res();
 lock_release(&res_1);
 lock_release(&res_2);
}
```

Attacking the Nonpreemptable condition

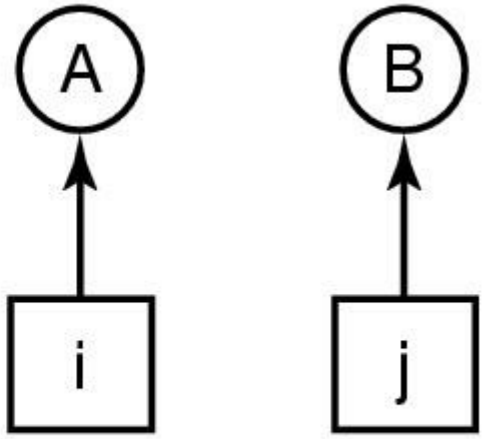
- This is usually not a viable option.
- Consider a process given the printer.
  - Pauses halfway through its job.
  - Now we forcibly take away the printer.
  - !!??



Attacking the Circular Wait Condition

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

(a)

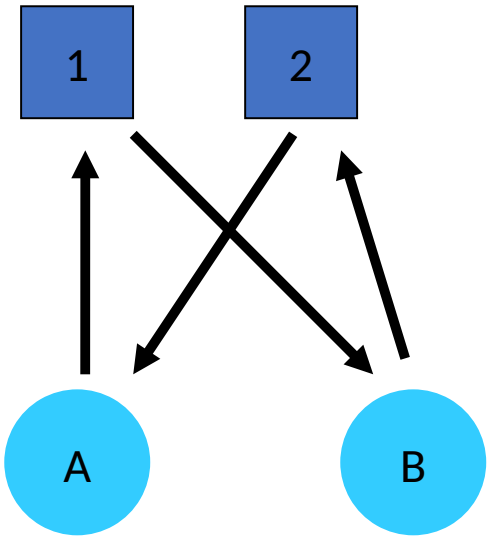


(b)

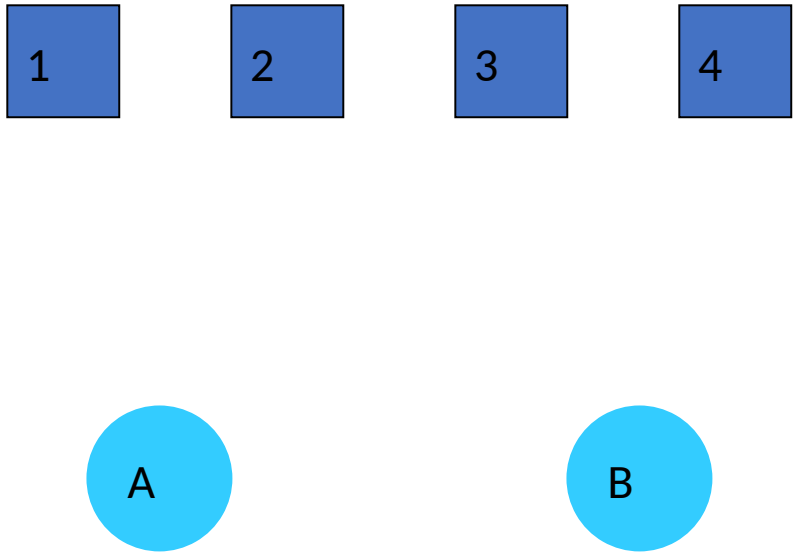
- Numerically ordered resources

Attacking the Circular Wait Condition

- Resources are taken in order.
- The displayed deadlock cannot happen
  - If A requires ***1***, it must acquire it before acquiring ***2***
  - Note: If B has ***1***, all higher numbered resources must be free or held by processes who doesn't need ***1***
- Resource ordering is a common technique in practice!!!!!



Example



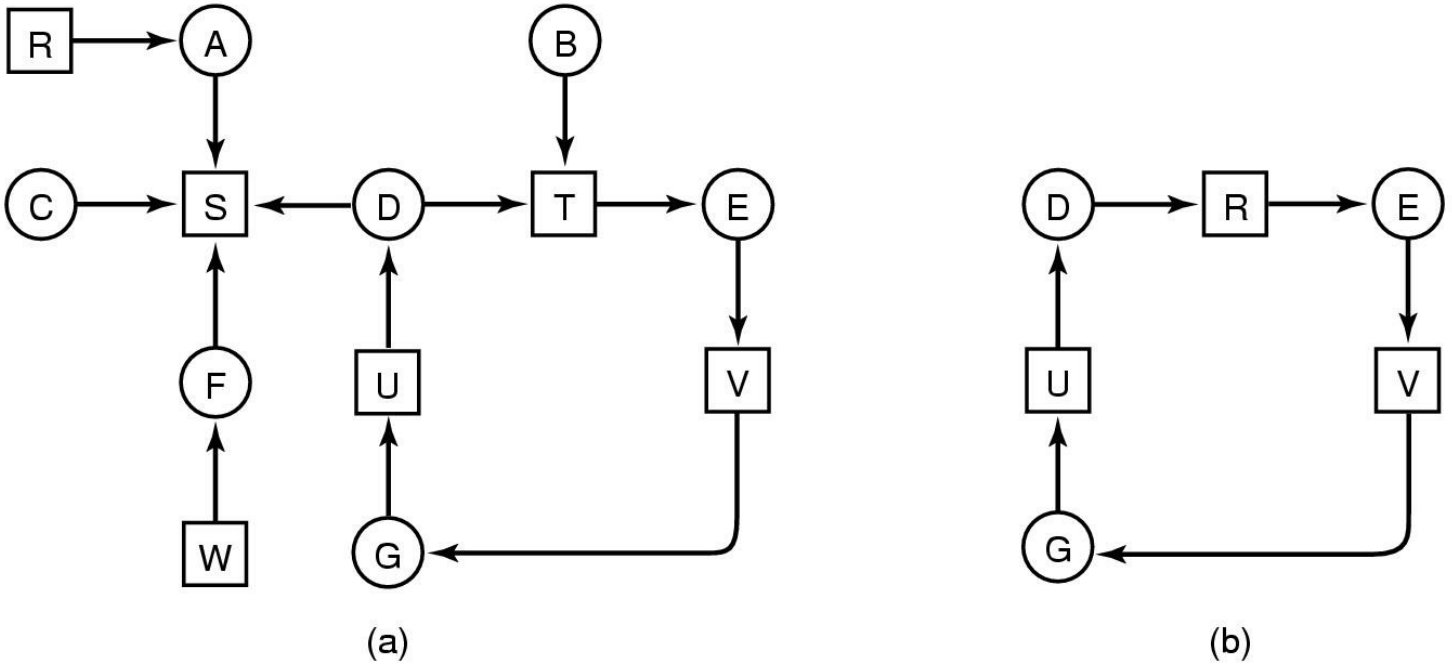
Summary of approaches to deadlock prevention

Condition	Approach
• Mutual Exclusion	• Not feasible
• Hold and Wait	• Request resources initially
• Non-preemptable	• Take resources away
• Circular Wait	• Order resources

Approach 3: Detection & Recovery

- We need a method to determine if a system is deadlocked.
- If and when a deadlock is detected, we need a method of recovery, a way to restore progress to the system.

Approach 3: Detection of Cycles

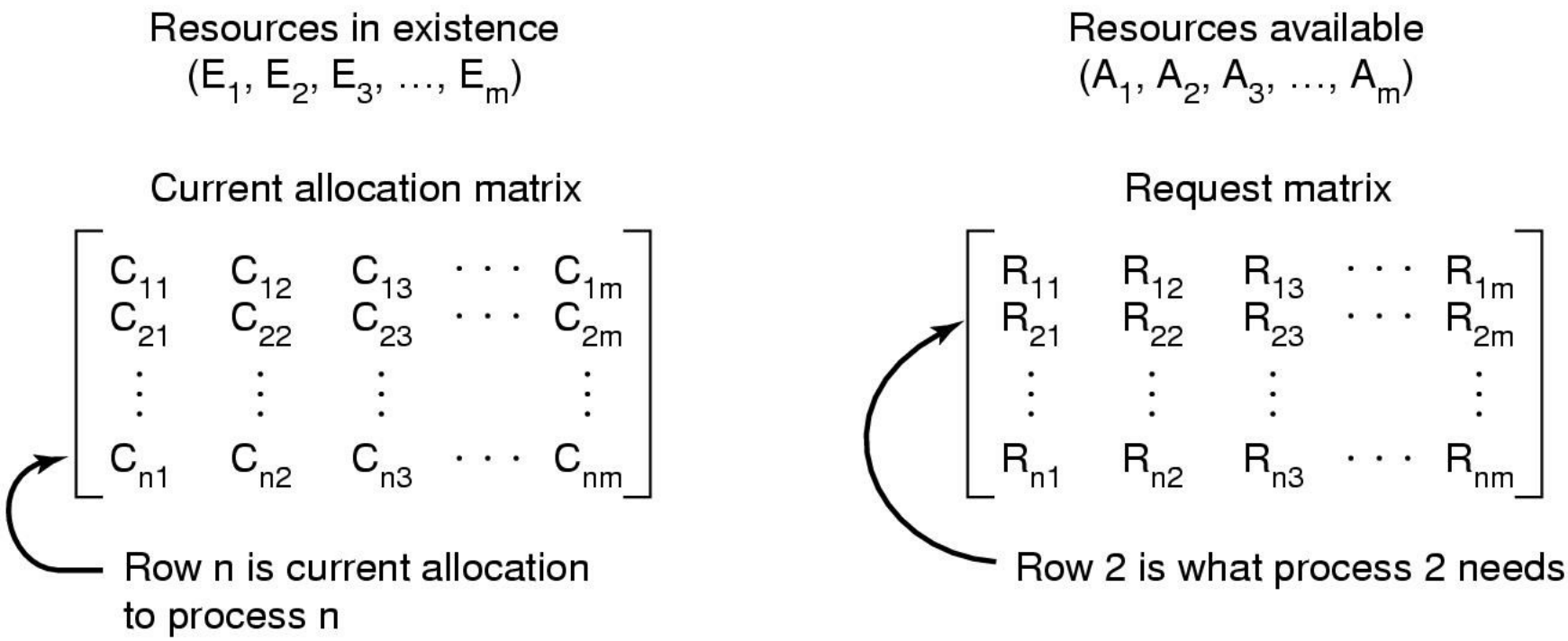


- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

What about resources with multiple units?

- As with producer-consumer and semaphores, sometimes a process is waiting on a “kind” of resource.
- Some examples of multi-unit resources:
  - bytes in RAM.
  - blocks on a hard disk drive.
  - slots in a buffer.
- We can generalise the approach to dealing with resources that consist of more than a single unit.

Detection with Multiple Resources of Each Type



Data structures needed by deadlock detection algorithm

Detection with Multiple Resources of Each Type

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

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

Current allocation matrix

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

Row n is current allocation to process n

Request matrix

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

Row 2 is what process 2 needs

There is a computation that can detect the deadlock condition based on this information.

Recovery from Deadlock

- Having detected a deadlock, how do we continue?
    - A subgroup of processes is deadlocked.

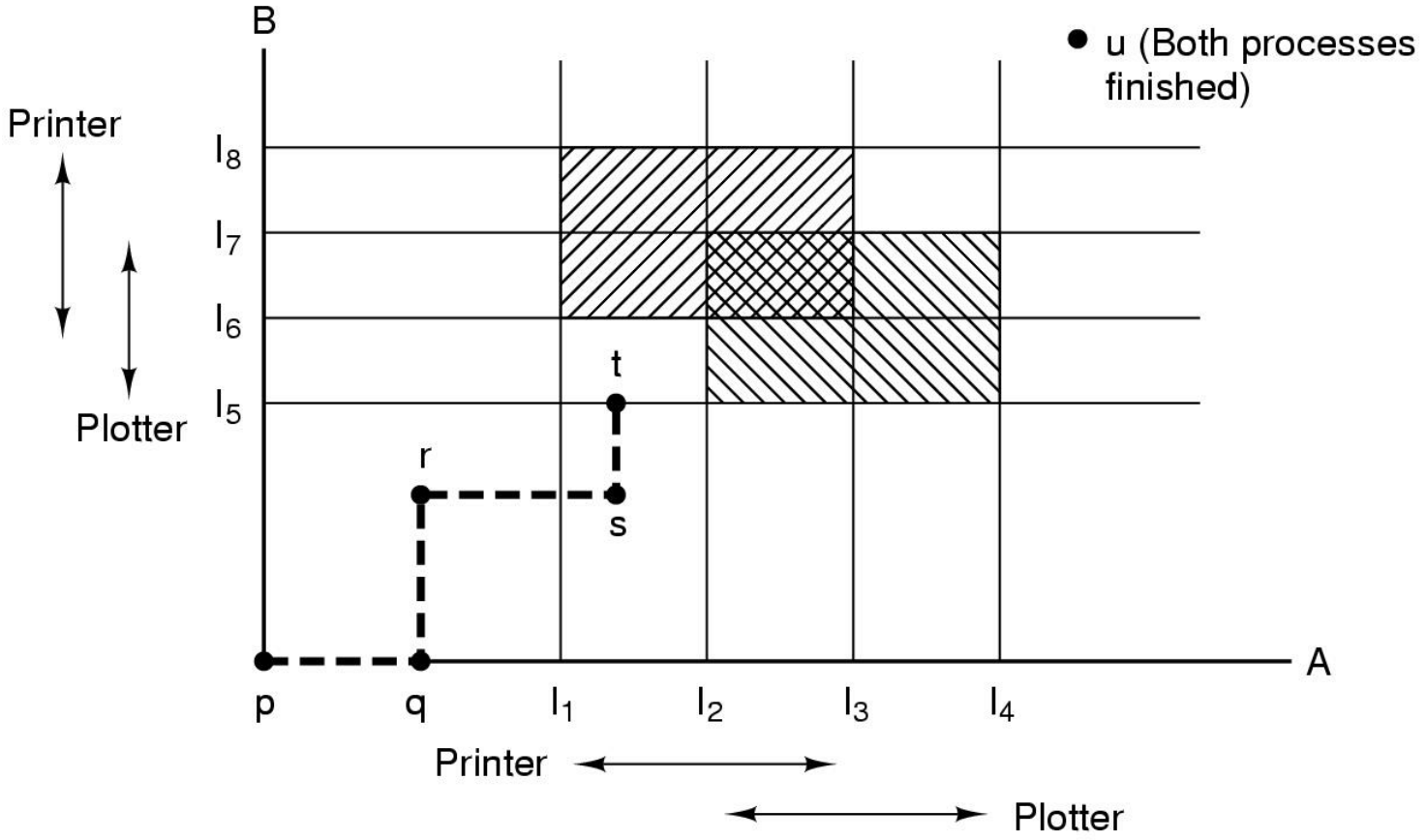
- Recovery is through killing processes:
    - The crudest and simplest way to break a deadlock.
    - Kill one of the processes in the deadlock cycle.
    - The other processes get its resources.
    - If possible, choose a process that can be rerun from the beginning.

Approach 4: Avoidance of Deadlock

- Instead of detecting deadlock, can we simply avoid it?

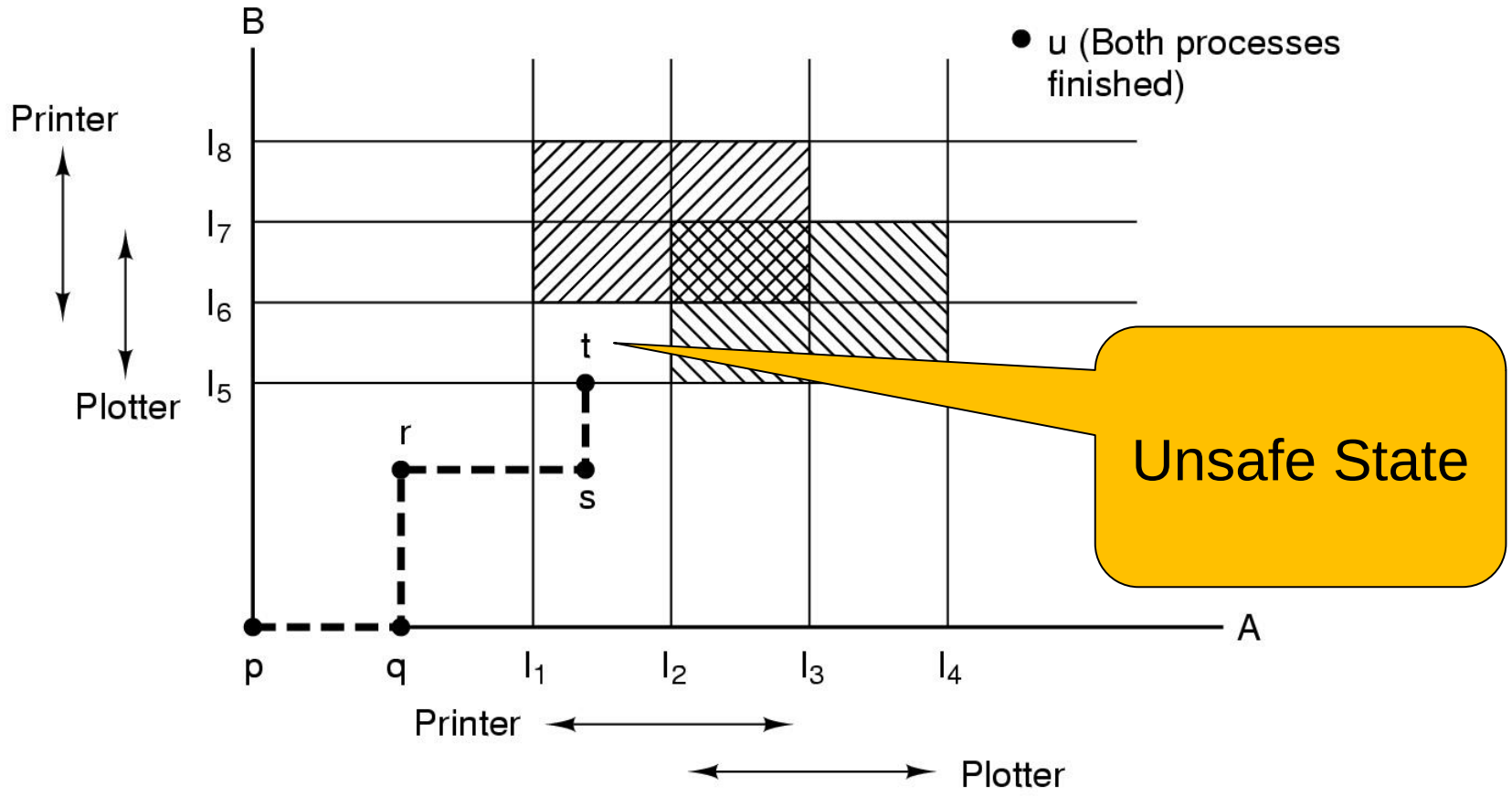
- Answer: Yes.
    - But only if enough information is available in advance.
    - Required information: the maximum amount of each resource that will be required.

Deadlock Avoidance
Resource Trajectories



Two process resource trajectories

Deadlock Avoidance
Resource Trajectories



Two process resource trajectories

Safe and Unsafe States

- A state is *safe* if
  - 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*

Safe and Unsafe States

Note: We have 10 units of the resource

	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

A requests one extra unit resulting in (b)

	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

Safe and Unsafe State

- Unsafe states are not necessarily deadlocked
  - With a lucky sequence, all processes may complete
  - However, we *cannot guarantee* that they will complete (not deadlock)
- Safe states guarantee we will eventually complete all processes
- Deadlock avoidance algorithm
  - Only grant requests that result in safe states

Bankers Algorithm

- Modelled on a Banker with Customers
  - The banker has a limited amount of money to loan customers
    - Limited number of resources
  - Each customer can borrow money up to the customer's credit limit
    - Maximum number of resources required
    - Promises to repay the money once they've done some deal
- Basic Idea
  - Keep the bank in a *safe* state
    - So all customers are *eventually* happy even if they all request to borrow up to their credit limit at the same time.
  - Customers wishing to borrow such that the bank would enter an unsafe state must wait until somebody else repays their loan such that the the transaction becomes safe
    - First re-do the "is this state safe" calculation on the modified state

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

B requests one more, should we grant it?

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
- Problem is structured similar to deadlock detection with multiple resources.
- Example in tutorial

Bankers Algorithm is not commonly used in practice

- It is difficult (sometimes impossible) to know in advance
  - the resources a process will require
  - the number of processes in a dynamic system

Starvation

- *Starvation* of a process is when it never receives the resource it is waiting for, despite the resource (repeatedly) becoming free.
  - The resource is always allocated to some other waiting process.
  - Example: An algorithm to allocate a resource may choose to give the resource to the shortest job first
  - Works great for multiple short jobs in a system
    - Minimises average waiting time to finish a job
  - May cause a long job to wait indefinitely, even though not blocked.

- One solution:
  - First-come, first-serve policy

This Lecture

- Deadlocks
- Livelocks
- Deadlock prevention
- Deadlock detection and recovery
- Deadlock avoidance
- Starvation