



# Operating Systems

## Deadlocks

Mohamed Zahran (aka Z)

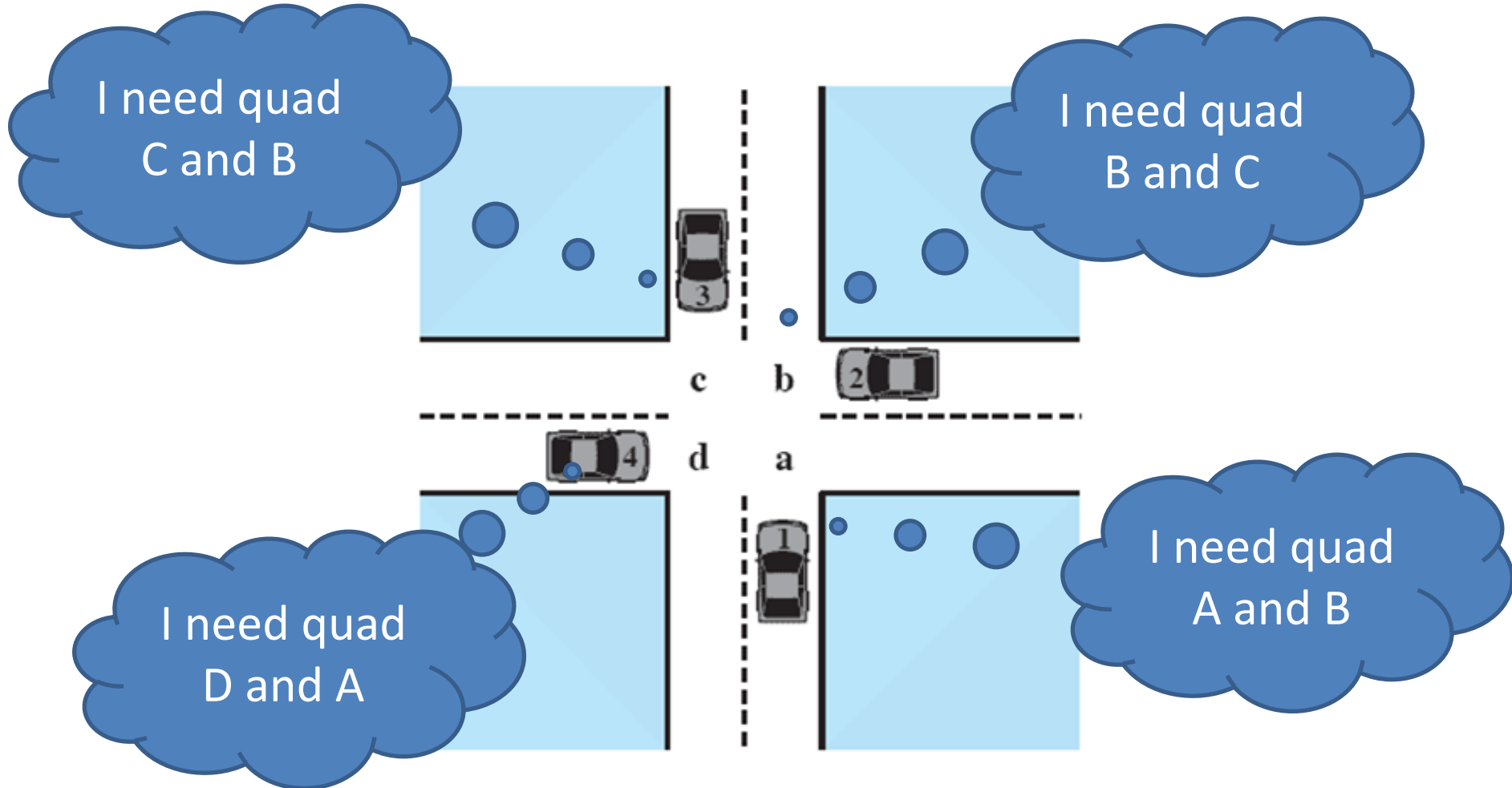
mzahran@cs.nyu.edu

<http://www.mzahran.com>

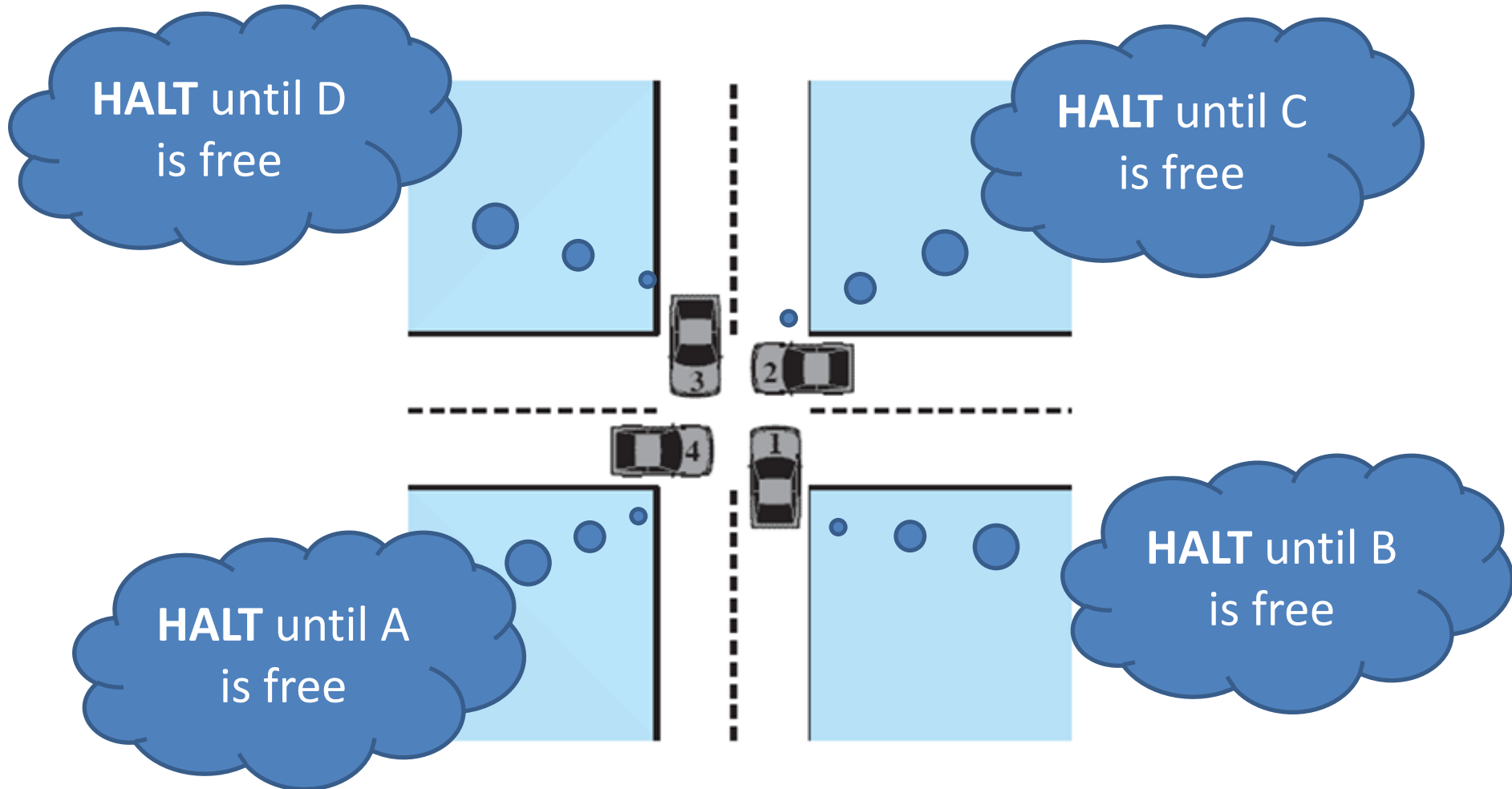




# Potential Deadlock



# Actual Deadlock



# Deadlocks

Occur among **processes** who need to  
acquire **resources** in order to **progress**

# Resources

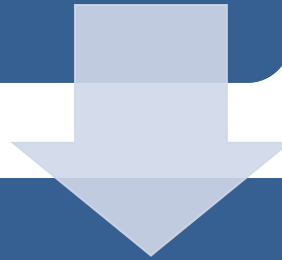
- Anything that must be acquired, used, and released over the course of time.
- Hardware or software resources
- Preemptable and Nonpreemptable resources:
  - **Preemptable:** can be taken away from the process with no ill-effect
  - **Nonpreemptable:** cannot be taken away from the process without causing the computation to fail



# Resource Categories

## Reusable

- can be safely used by only one process at a time and is not depleted by that use
- processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores



## Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, ...
  - I/O buffers

```
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);
}
```



## Deadlock-free code

```
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);  
}
```

## Code with potential deadlock

```
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);  
}
```

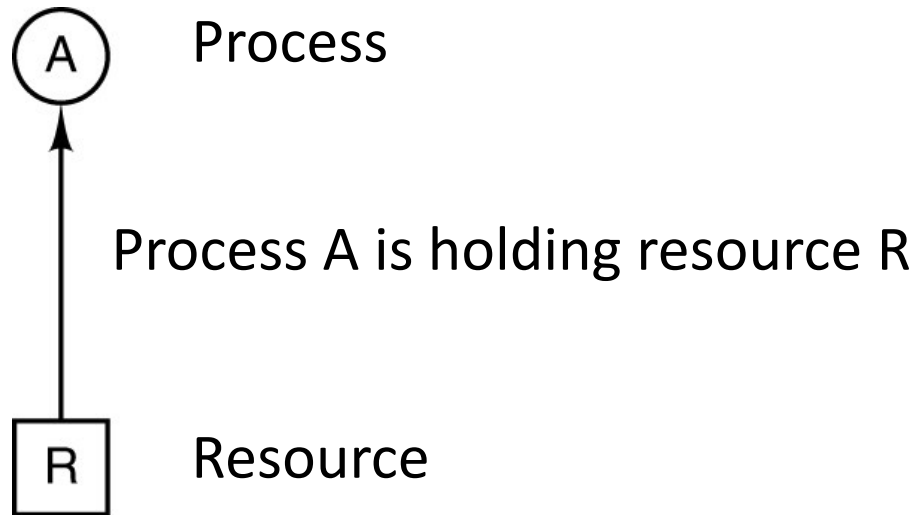
# So ...

- 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.
- Assumptions
  - If a process is denied a resource, it is put to sleep
  - Only single-thread processes
  - No interrupts possible to wake up a blocked process, except the availability of the needed resource

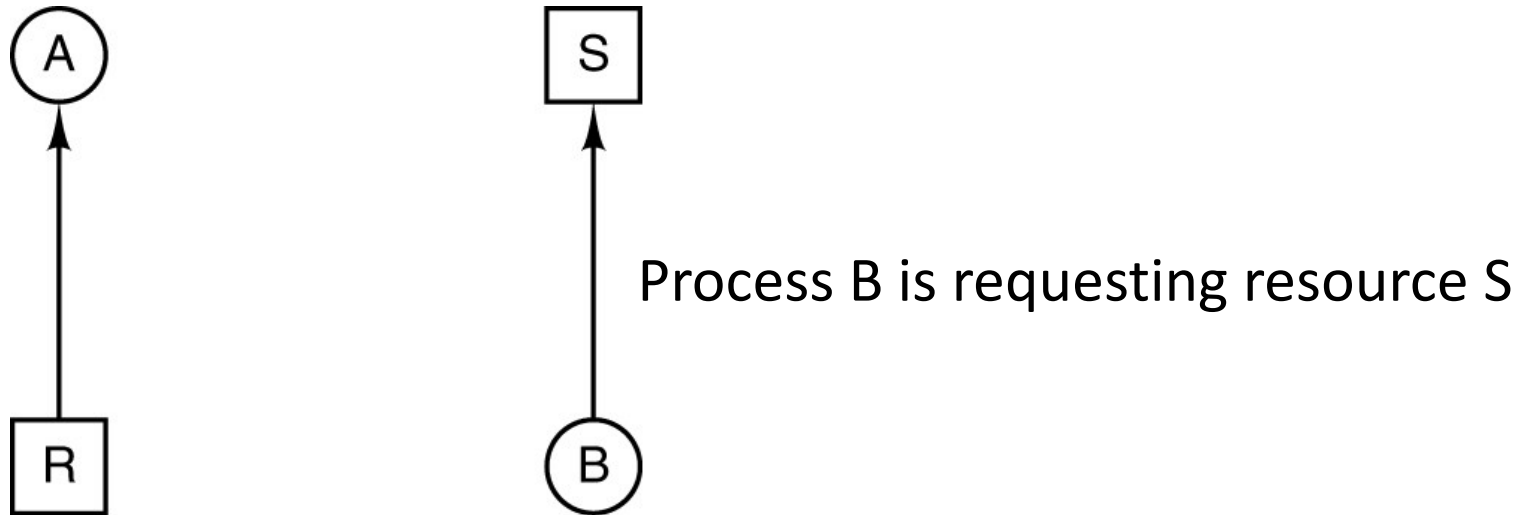
# Conditions for Resource Deadlocks

1. Each resource is either currently assigned to exactly one process or is available.
2. Processes currently holding resources that were granted earlier can request new resources.
3. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. There must be a **circular chain** of two or more processes, each of which is waiting for a resource held by the next member of the chain.

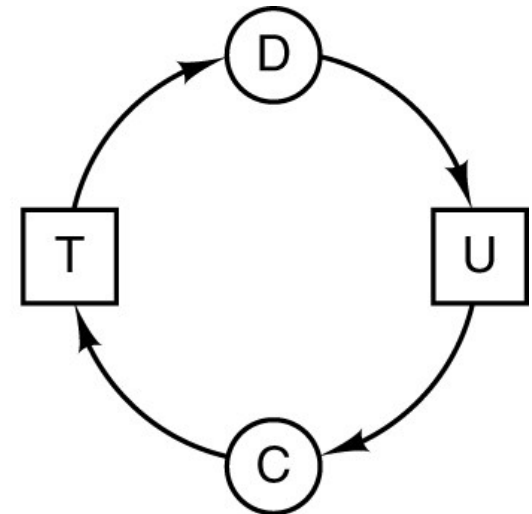
# Resource Allocation Graph



# Resource Allocation Graph



# Resource Allocation Graph



Deadlock!

## Example 1:

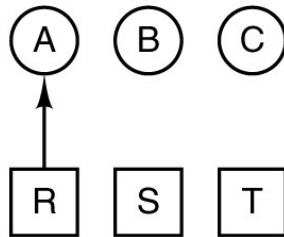
### Actions to be done by each process:

| 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 |

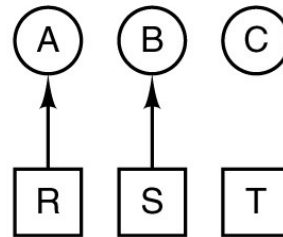
### Actual Scenario

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

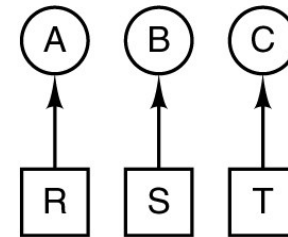
(d)



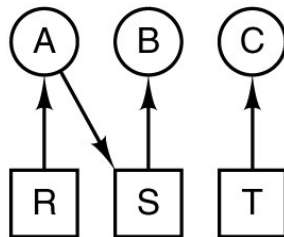
(e)



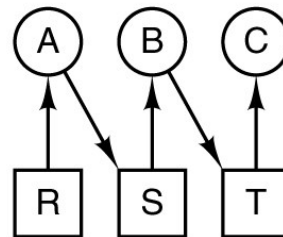
(f)



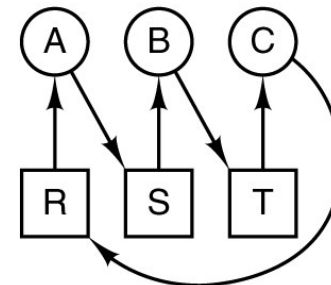
(g)



(h)



(i)



(j)



## Example 2:

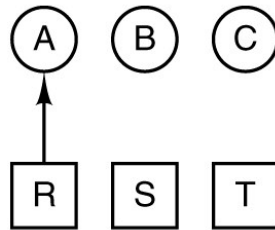
### Actions to be done by each process:

| 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 |

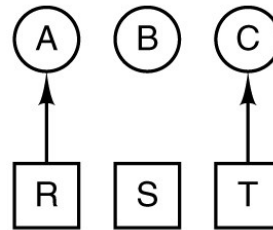
### Actual Scenario

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

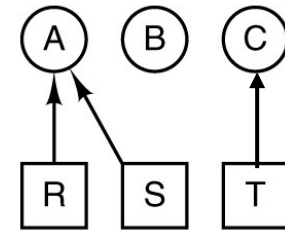
(k)



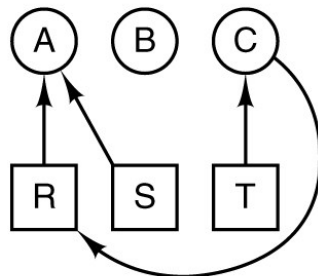
(l)



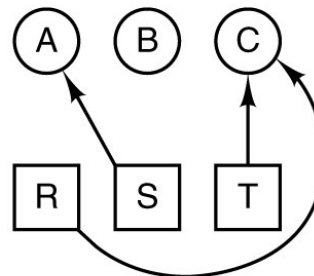
(m)



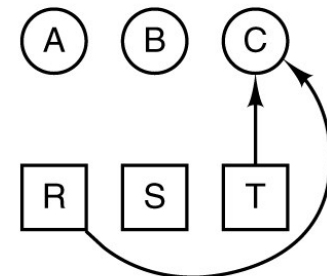
(n)



(o)



(p)



(q)

# How to Deal with Deadlocks

1. Just ignore the problem!
2. Let deadlocks occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four required conditions (slide 11).

Just ignore the problem!

# The Ostrich Algorithm



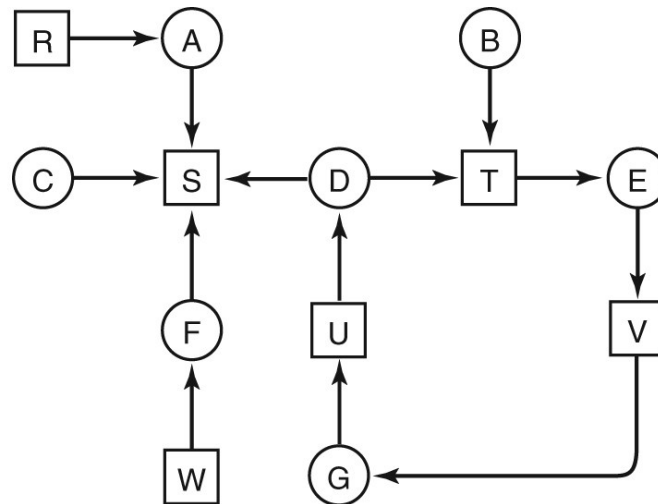
Let deadlocks occur, detect  
them, and take action

# Deadlock Detection and Recovery

- The system does not attempt to prevent deadlocks.
- It tries to detect it when it happens.
- Then it takes some actions to recover.
- Several issues here:
  - Deadlock detection with one resource of each type
  - Deadlock detection with multiple resources of each type
  - Recovery from deadlock

# Deadlock Detection: One Resource of Each Type

- Construct a resource graph
- If it contains one or more cycles, a deadlock exists





# Formal Algorithm to Detect Cycles in the Allocation Graph

For Each node  $N$  in the graph do:

1. Initialize  $L$  to empty list and designate all arcs (i.e. edges) as unmarked
2. Add the current node to end of  $L$ . If the node appears in  $L$  twice then we have a cycle and the algorithm terminates.
3. From the given node pick any unmarked outgoing arc. If none is available go to 5.
4. Mark the arc you picked at 3. Then follow it to the new current node and go to 2.
5. If the node is the initial node, then no cycles and the algorithm terminates. Otherwise, we are in dead end. Remove that node and go back to the previous one. Go to 2.


# Deadlock Detection: Multiple Resources of Each Type

n processes and m resource types

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

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

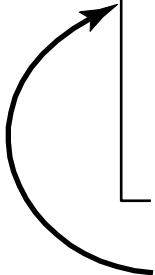
Current allocation matrix



|          |          |          |         |          |
|----------|----------|----------|---------|----------|
| $C_{11}$ | $C_{12}$ | $C_{13}$ | $\dots$ | $C_{1m}$ |
| $C_{21}$ | $C_{22}$ | $C_{23}$ | $\dots$ | $C_{2m}$ |
| $\vdots$ | $\vdots$ | $\vdots$ |         | $\vdots$ |
| $C_{n1}$ | $C_{n2}$ | $C_{n3}$ | $\dots$ | $C_{nm}$ |

Row n is current allocation  
to process n

Request matrix



|          |          |          |         |          |
|----------|----------|----------|---------|----------|
| $R_{11}$ | $R_{12}$ | $R_{13}$ | $\dots$ | $R_{1m}$ |
| $R_{21}$ | $R_{22}$ | $R_{23}$ | $\dots$ | $R_{2m}$ |
| $\vdots$ | $\vdots$ | $\vdots$ |         | $\vdots$ |
| $R_{n1}$ | $R_{n2}$ | $R_{n3}$ | $\dots$ | $R_{nm}$ |

Row 2 is what process 2 needs

A Process is said to be **marked** if they are able to complete and hence not deadlocked

# Deadlock Detection: Multiple Resources of Each Type

n processes and m resource types

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

Steps of the deadlock detection algorithm:

1. Look for an unmarked process, process i, for which the  $i^{\text{th}}$  row of  $R \leq A$
2. If such process is found, Add  $i^{\text{th}}$  row of  $C$  to  $A$ , mark the process, and go to step 1.
3. If no such process exists and there are unmarked processes  $\rightarrow$  deadlock

# Example

- Resources in Existence: **[2 1 3]**

Means we have three resource types. We have 2 instances of the first resource, one instance of the second, and 3 instances of the third.

- We have **three processes**

Availability **[1 0 2]** → At that moment, we have one 1 instance of the first resource available, none of the second resource, and 2 of the third.

Current **allocation matrix:**

|   |   |   |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

**Request matrix:**

|   |   |   |
|---|---|---|
| 1 | 0 | 3 |
| 0 | 0 | 1 |
| 2 | 0 | 0 |

Do we have a deadlock?

# Example

- Resources in Existence: **[2 1 3]**

Means we have three resource types. We have 2 instances of the first resource, one instance of the second, and 3 instances of the third.

- We have **three processes**

Availability **[1 0 2]** → At that moment, we have one 1 instance of the first resource available, none of the second resource, and 2 of the third.

Current **allocation matrix:**

|          |          |          |
|----------|----------|----------|
| <b>1</b> | <b>1</b> | <b>0</b> |
| <b>0</b> | <b>0</b> | <b>1</b> |
| <b>0</b> | <b>0</b> | <b>0</b> |

**Request matrix:**

|          |          |          |
|----------|----------|----------|
| <b>1</b> | <b>0</b> | <b>3</b> |
| <b>0</b> | <b>0</b> | <b>1</b> |
| <b>2</b> | <b>0</b> | <b>0</b> |

# Example

- Resources in Existence: [2 1 3]

Means we have three resource types. We have 2 instances of the first resource, one instance of the second, and 3 instances of the third.

- We have three processes

Availability [1 0 2] → At that moment, we have one 1 instance of the first resource available, none of the second resource, and 2 of the third.

Current allocation matrix:

|   |   |   |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

Request matrix:

|   |   |   |
|---|---|---|
| 1 | 0 | 3 |
| 0 | 0 | 1 |
| 2 | 0 | 0 |

- All processes are unmarked.
- Let's pick a process whose request row is less than the availability vector.  
→ Process 2 is a good candidate (second row of request matrix  $\leq$  availability vector)

Then:

- Add the row #2 of the allocation matrix to the availability row.
- Mark process 2

# Example

- Resources in Existence:  $[2 \ 1 \ 3]$

Means we have three resource types. We have 2 instances of the first resource, one instance of the second, and 3 instances of the third.

- We have **three processes**

Availability  $[1 \ 0 \ 3]$  → At that moment, we have one 1 instance of the first resource available, none of the second resource, and 2 of the third.

Current **allocation matrix**:

|   |   |   |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

**Request matrix**:

|   |   |   |
|---|---|---|
| 1 | 0 | 3 |
| 0 | 0 | 1 |
| 2 | 0 | 0 |

- Marked process {2}
- Let's pick a process whose request row is less than the availability vector.  
→ Process 1 is a good candidate (first row of request matrix  $\leq$  availability vector)

Then:

- Add the row #1 of the allocation matrix to the availability row.
- Mark process 1



# Example

- Resources in Existence: **[2 1 3]**

Means we have three resource types. We have 2 instances of the first resource, one instance of the second, and 3 instances of the third.

- We have **three processes**

Availability **[2 1 3]** → At that moment, we have one 1 instance of the first resource available, none of the second resource, and 2 of the third.

Current **allocation matrix**:

|   |   |   |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

**Request matrix**:

|   |   |   |
|---|---|---|
| 1 | 0 | 3 |
| 0 | 0 | 1 |
| 2 | 0 | 0 |

- Marked process {1, 2}
- Let's pick a process whose request row is less than the availability vector.  
→ Process 3 is left (third row of request matrix  $\leq$  availability vector)

Then:

- Add the row #3 of the allocation matrix to the availability row.
- Mark process 3

# Example

- Resources in Existence: **[2 1 3]**

Means we have three resource types. We have 2 instances of the first resource, one instance of the second, and 3 instances of the third.

- We have **three processes**

Availability **[2 1 3]** → At that moment, we have one 1 instance of the first resource available, none of the second resource, and 2 of the third.

Current **allocation matrix**:

|   |   |   |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |

**Request matrix**:

|   |   |   |
|---|---|---|
| 1 | 0 | 3 |
| 0 | 0 | 1 |
| 2 | 0 | 0 |

Marked process {1, 2, 3}

**All processes marked, no deadlock!**

# When to Check for Deadlocks?

- Check every time a resource request is made.
- Check every  $k$  minutes.
- When CPU utilization has dropped below a threshold.

# Recovery from Deadlock

- We have detected a deadlock ... What next?
- We have some options:
  - Recovery through preemption
  - Recovery through rollback
  - Recovery through killing processes

# Recovery from Deadlock: Through Preemption

- Temporary take a resource away from its owner and give it to another process.
- Highly dependent on the nature of the resource.

# Recovery from Deadlock: Through Rollback

- Have processes **checkpointed** periodically
- Checkpoint of a process: its **state** is written to a file so that it can be restarted later
- In case of deadlock, a process that owns a needed resource is rolled back to the point before it acquired that resource

# Recovery from Deadlock: Through Killing Processes

- Kill a process in the cycle.
- Can be repeated (i.e. kill other processes) till deadlock is resolved.
- The victim can also be a process NOT in the cycle.



Dynamic avoidance by careful  
resource allocation

# Deadlock Avoidance


- In most systems, resources are requested one at a time.
- Resource is granted only if it is **safe** to do so.

# Safe and Unsafe States

- A **state** is said to be safe if there is one scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
- An **unsafe** state is NOT a deadlock state, but a potential deadlock.

# Safe and Unsafe States

Maximum the process will need (e.g. A will need 6 more).



|   | Has | Max |
|---|-----|-----|
| A | 3   | 9   |
| B | 2   | 4   |
| C | 2   | 7   |

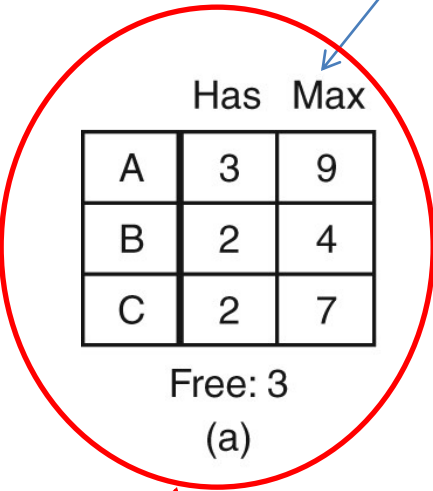
Free: 3

(a)

**Assume a total of 10 instances of the resources available  
Therefore “Free: x” means we have x instances available.**

# Safe and Unsafe States

Maximum the process will need (e.g. A will need 6 more).



|   | 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)

**Assume a total of 10 instances of the resources available**  
**Therefore “Free: x” means we have x instances available.**

**This state is **safe** because there exists a sequence of allocations that allows all processes to complete.**

# 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)

How about this state?

Unsafe  
(as shown by this sequence)

The difference between a safe and unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

# The Banker's Algorithm

- Dijkstra 1965
- Checks if granting the request leads to an unsafe state.
- If it does, the request is denied.

# The Banker's Algorithm: The Main Idea

- The algorithm checks to see if it has enough resources to satisfy some customers.
- If so, the process closest to the limit is assumed to be done and resources are back, and so on.
- If all loans (resources) can eventually be repaid, the state is safe.



# The Banker's Algorithm: Example (single resource type)

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

Free: 10

(a)

Safe

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

Free: 2

(b)

Safe

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

Free: 1

(c)

Unsafe

# The Banker's Algorithm: Example (multiple resources)

|   | Process | Tape drives | Plotters | Printers | 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 | Printers | 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

# The Banker's Algorithm

- Very nice theoretically
- Practically useless!
  - Processes rarely know in advance what their maximum resource needs will be.
  - The number of processes is not fixed.
  - Resources can suddenly vanish.

Prevention, by structurally  
negating one of the four  
required conditions (slide 11).

# Deadlock Prevention

If we can ensure that at least one of the four conditions of the deadlock is never satisfied, then deadlocks will be structurally impossible.

1. Each resource is either currently assigned to exactly one process or is available.
2. Processes currently holding resources that were granted earlier can request new resources.
3. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. There must be a **circular chain** of two or more processes, each of which is waiting for a resource held by the next member of the chain.

# Deadlock Prevention: Attacking the Mutual Exclusion

- Can be done for some resources (e.g the printer) but not all.
- Spooling: saves the data with the OS till the resource becomes available
  - e.g. A file to be printed is stored with the OS till the printer becomes available.

# Deadlock Prevention:

## Attacking the Hold and Wait Condition

- Prevent processes holding resources from waiting for more resources.
- This requires all processes to request all their resources before starting execution.
- A different strategy: require a process requesting a resource to first temporarily release all the resources it currently holds. Then tries to get everything it needs all at once

# Deadlock Prevention:

## Attacking No Preemption Condition

- Virtualizing some resources can be a good strategy (e.g. virtualizing a printer)
- Not all resources can be virtualized (e.g. records in a database)



# Deadlock Prevention:

## The circular Wait Condition

- **Method 1:** Have a rule saying that a process is entitled to only a single resource at a moment.
- **Method 2:**
  - Provide a global numbering of all resources.
  - A process can request resources whenever they want to, but all requests must be done in numerical order.
  - With this rule, resource allocation graph can never have cycles.

# Deadlock Prevention: Summary

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

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

# Conclusions

- Deadlocks can occur on hardware/software resources
- OS need to be able to:
  - Detect deadlocks
  - Deal with them when detected
  - Try to avoid them if possible