Chapter 4

Deadlocks

Resources

- A resource is anything that can be used by only a single process at any instant of time.

- Examples of computer resources
  - Printer
  - Tape drive
  - A locked record in a database

Resources

Two types:

1. Preemptable resources
   - can be taken away from a process with no ill effects
   - E.g. Process swapping from main memory, buffers, CPU, array processor

2. Non-preemptable resources
   - will cause the process to fail if taken away
   - E.g. Printer, Scanner, tape drive, CD-ROM
     - A process began to burn a CD-ROM
     - Suddenly taking the CD-ROM away from it and giving it to another process will result a garbled CD.

Resources

- 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
  – may fail with error code

Resource Acquisition

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

```
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);
}
```

(b)

Figure 6-1. Using a semaphore to protect resources. (a) One resource. (b) Two resources.

Resource Acquisition

```
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);
        }
```

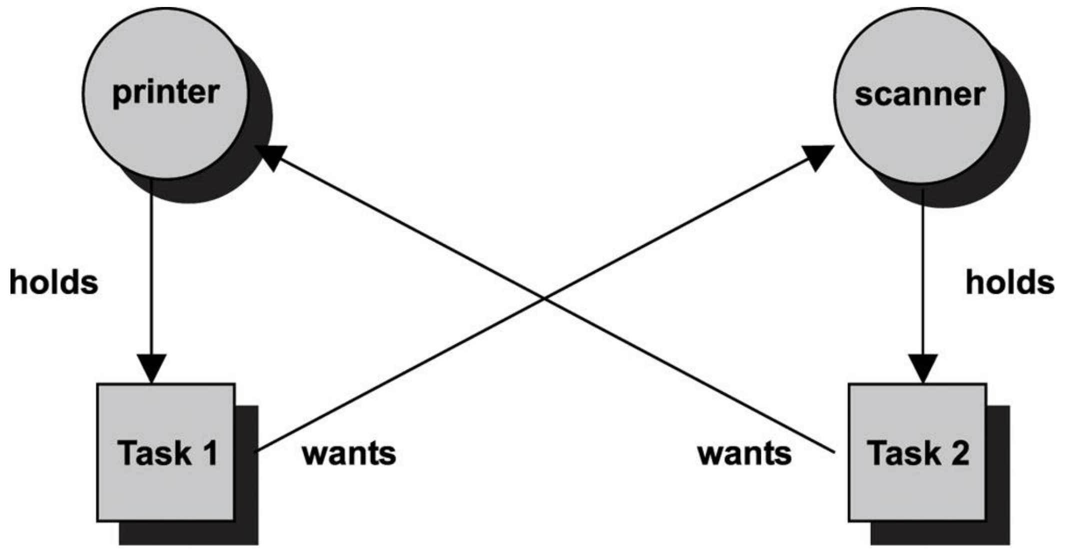
(a)

```
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);
}
```

(b)

Figure 6-2. (a) Deadlock-free code. (b) Code with a potential deadlock.

Introduction to Deadlocks



- Task: two processes **A** and **B**, each wants to record a scanned document on a CD. But they are programmed differently.
- **A** requests to use the **scanner** first, and gets the access. But, **B** requests to use the **CD recorder** first, and gets the access.
- Now **A** asks for the **CD recorder**, but the request is denied until **B** releases it. Unfortunately, instead of releasing the **CD recorder B** asks for the **scanner**.
- Both **A** and **B** are locked and will remain so forever.

Introduction to Deadlocks

You release the lock first
Once I have finished
my task, you can continue.
Why should I?
You release the lock first
and wait until
I complete my task.

Introduction to Deadlocks

- A deadlock is a state in which each member of a group is waiting for some other member to take action
- A deadlock consists of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set.
- None of the processes can …
  - run
  - release resources
  - be awakened

Introduction to Deadlocks

- In general, deadlocks involve nonpreemptable resources.

- If preemtable, deadlocks can be resolved by reallocating the resources from one process to another.

Deadlock in real life

Four Conditions for Deadlock

Deadlock can arise if four conditions hold simultaneously.

1. Mutual exclusion condition
2. Hold and wait condition
3. No preemption condition
4. Circular wait condition

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

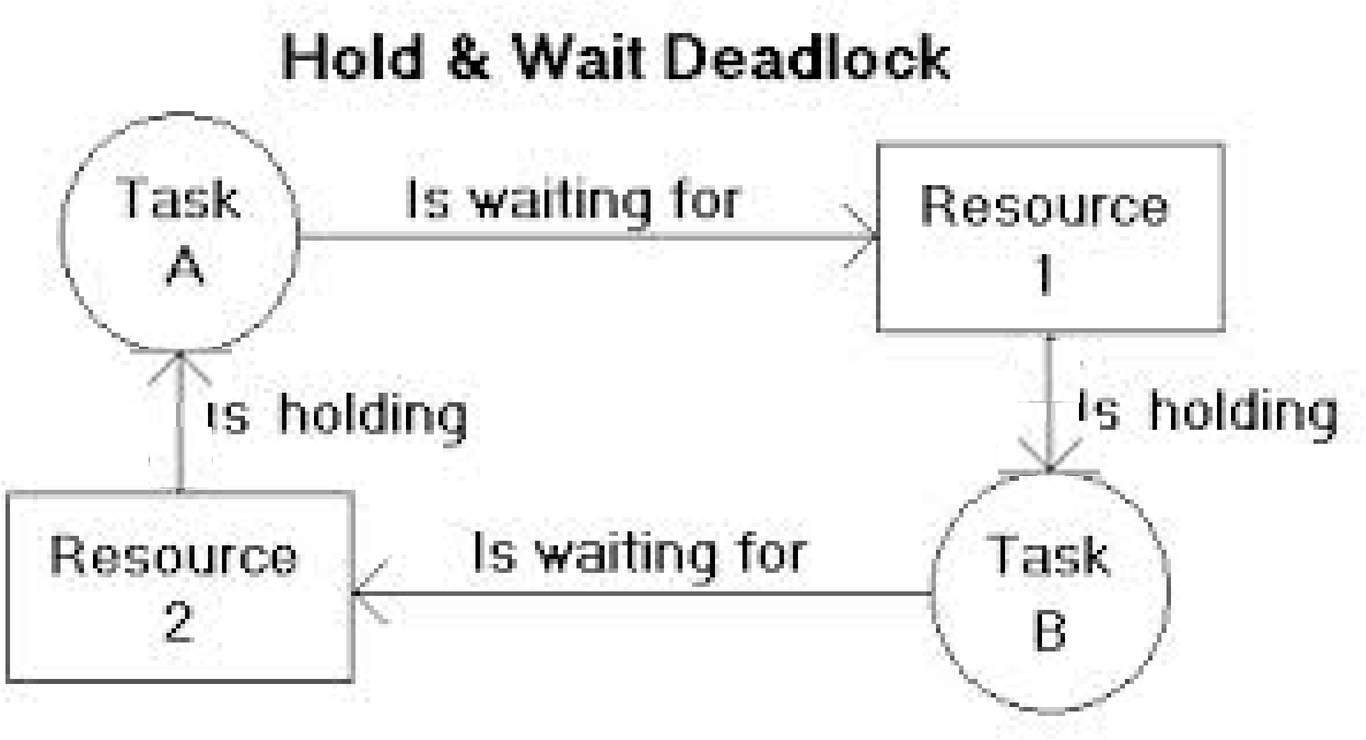
Four Conditions for Deadlock

1. Mutual exclusion condition
   - Only one process at a time can use the resource.
   - If another process requests that resource, the requesting process must be delayed until the resource has been released.

Four Conditions for Deadlock

2. Hold and wait condition

Process currently holding resources granted earlier can request additional resources



Four Conditions for Deadlock

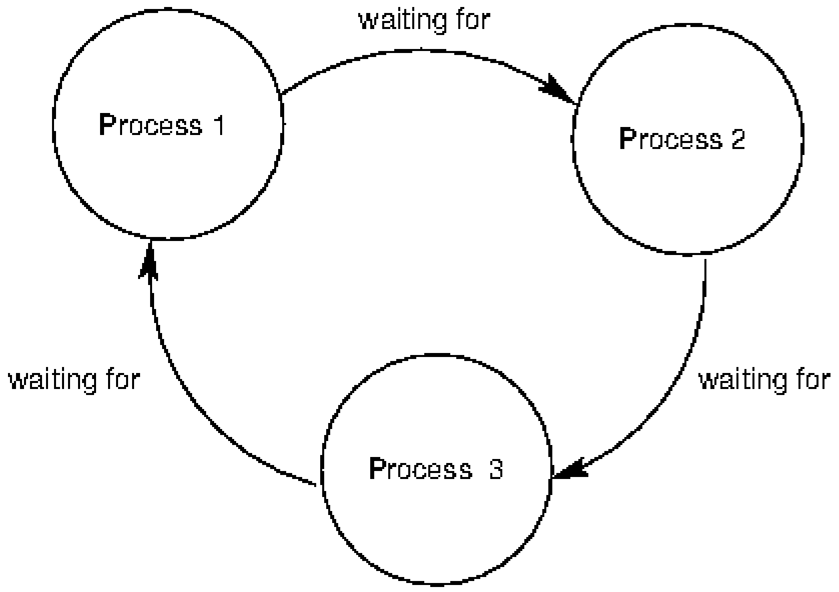
3. No preemption condition

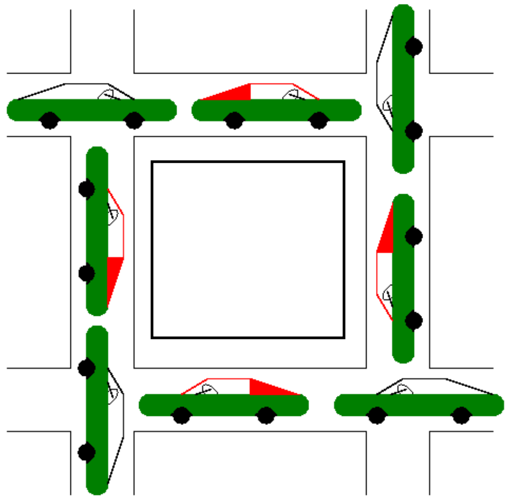
Previously granted resources cannot forcibly taken away. They must be explicitly released by the process holding them.

Four Conditions for Deadlock

4. Circular wait condition

Must be a circular chain of two or more processes, each is waiting for a resource held by next member of the chain

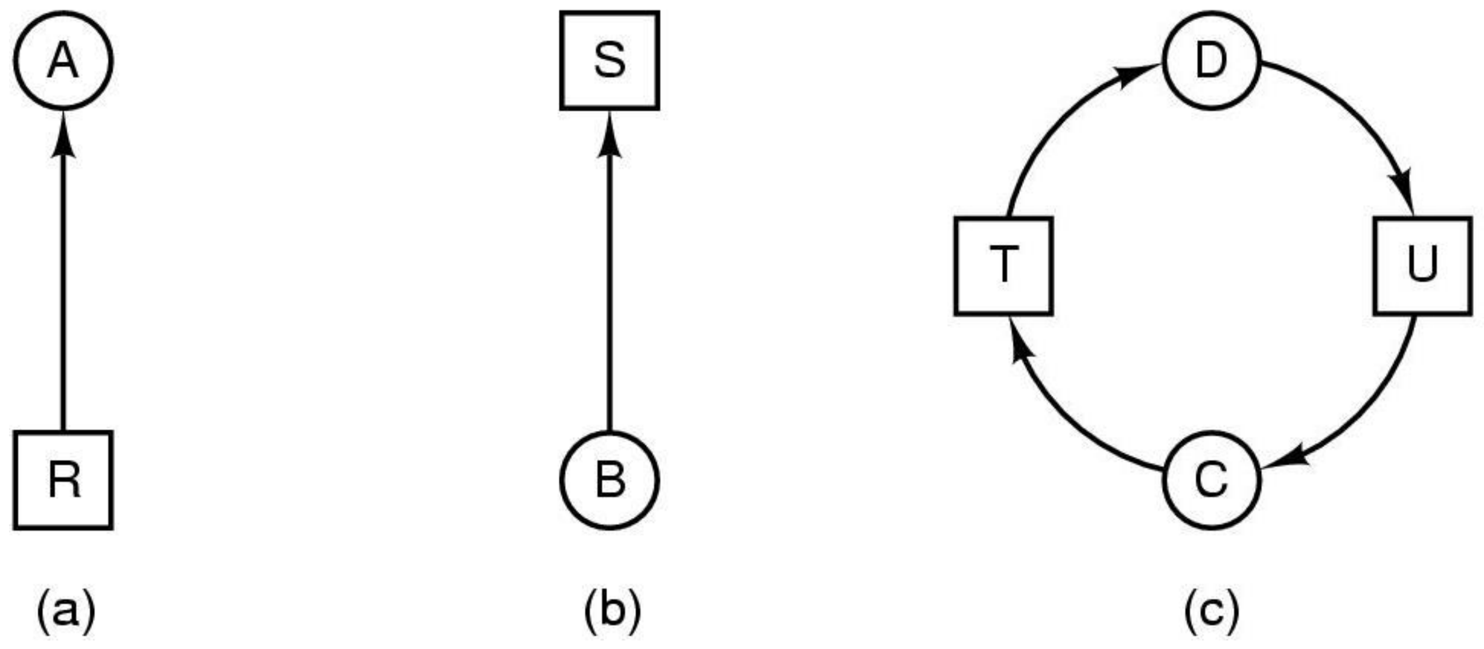




Deadlock Modeling: Resource-Allocation Graphs

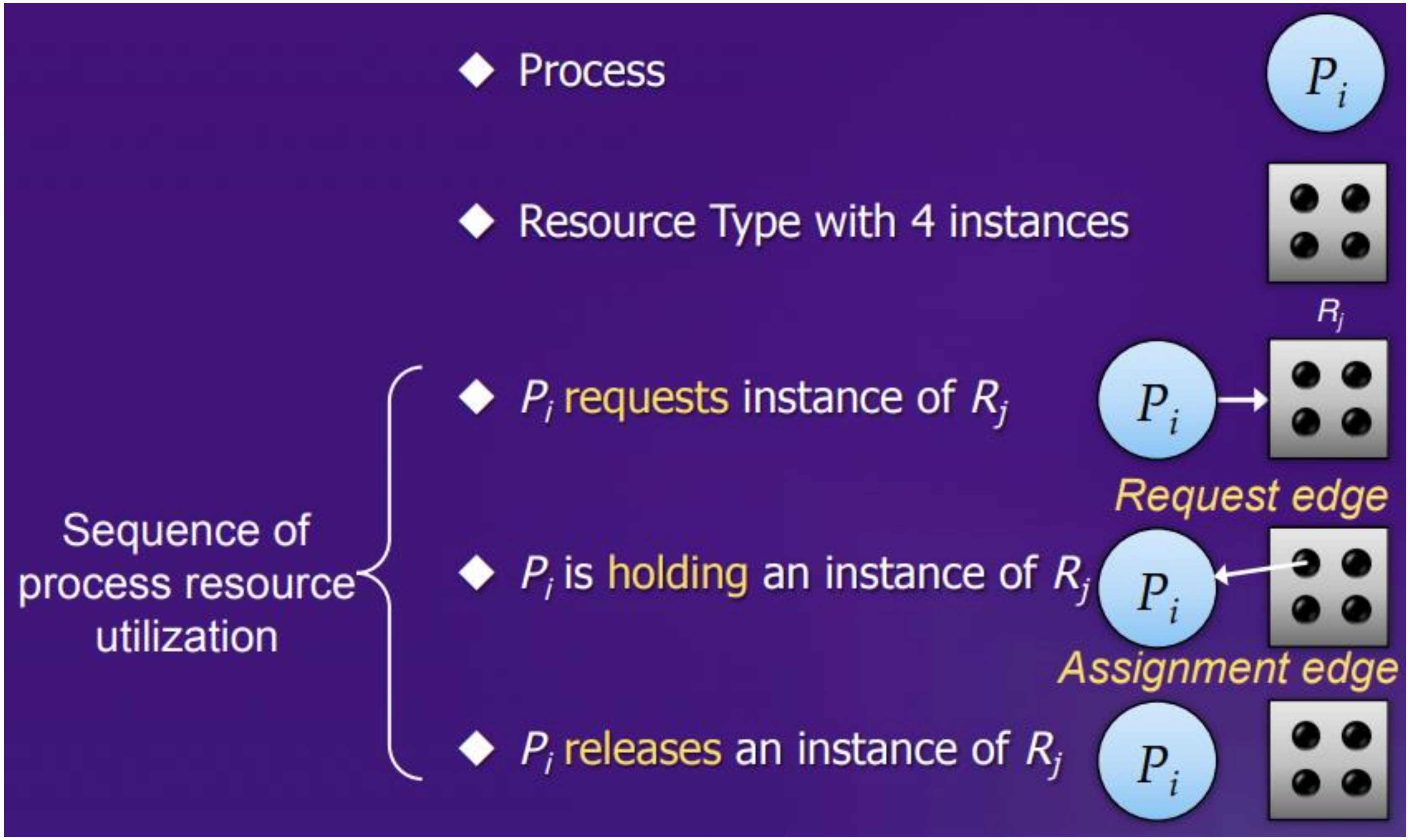
● Process

■ Resource



a) Resource R assigned to process A
b) Process B is waiting for resource S
c) Process C and D are in deadlock over resources T and U

Deadlock Modeling: Resource-Allocation Graphs



- Process
- Resource Type with 4 instances
- P_i requests instance of R_j
- P_i is holding an instance of R_j
- P_i releases an instance of R_j

Sequence of process resource utilization

Deadlock Modeling: Resource-Allocation Graphs

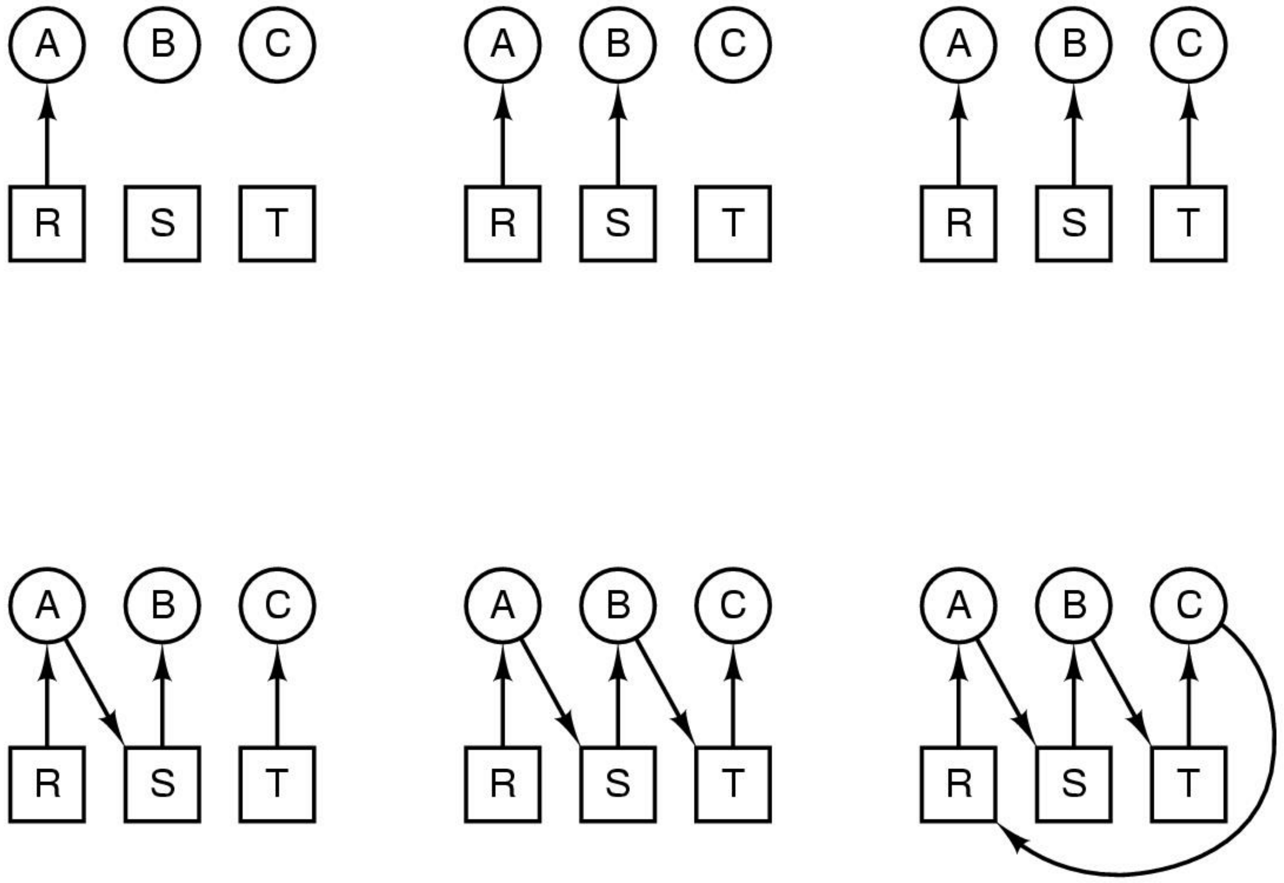
How deadlock occurs?

Exercise:

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

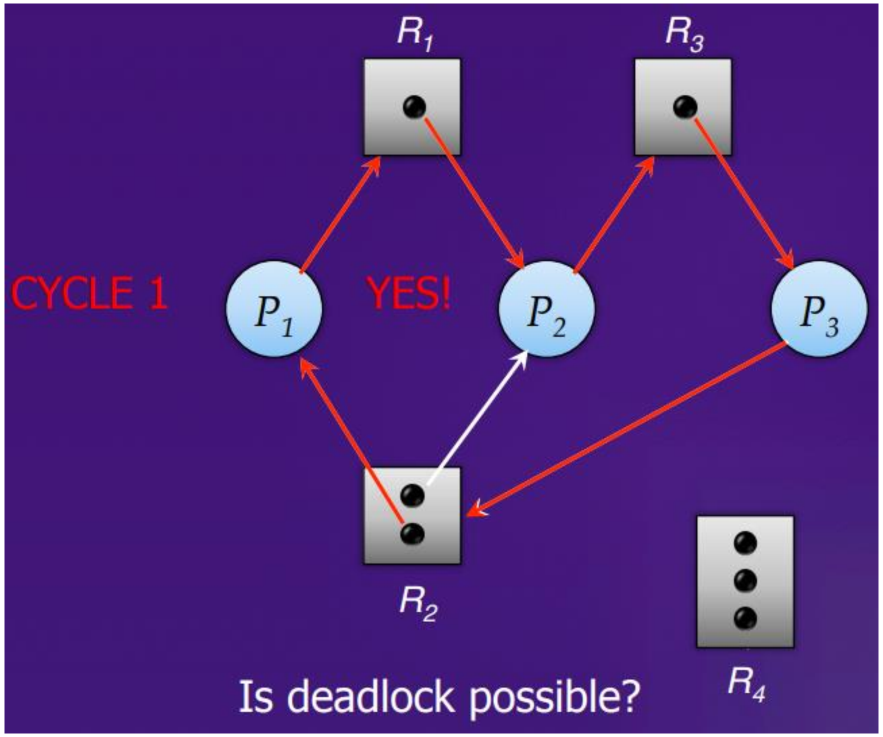
Deadlock Modeling: Resource-Allocation Graphs

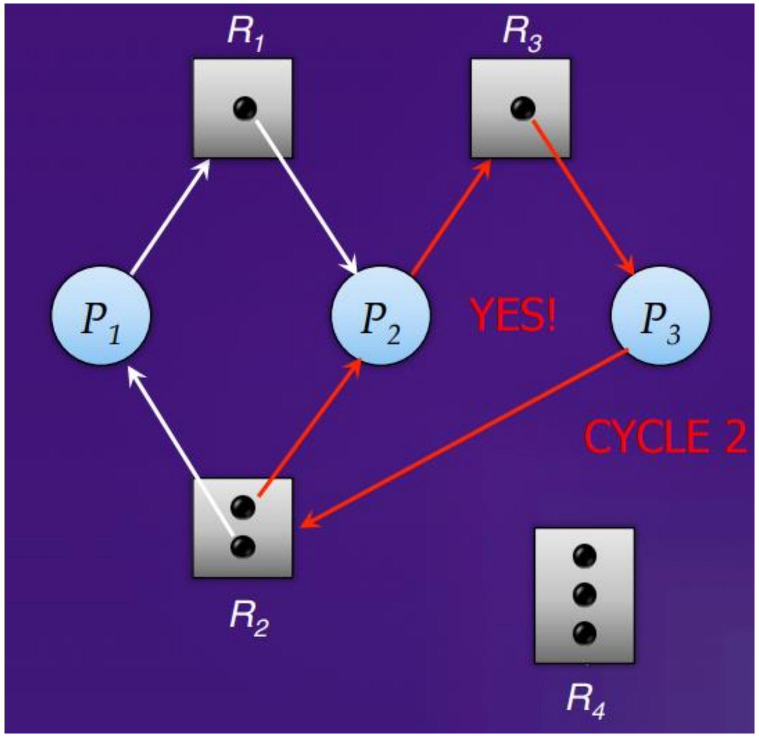
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock



Circular wait → deadlock

Deadlock Modeling: Resource-Allocation Graphs



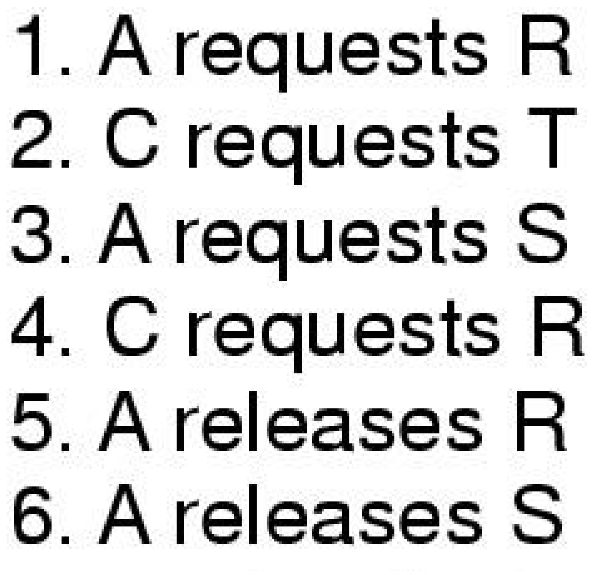


Circular wait → deadlock

Deadlock Modeling: Resource-Allocation Graphs

How deadlock can be avoided

Exercise:

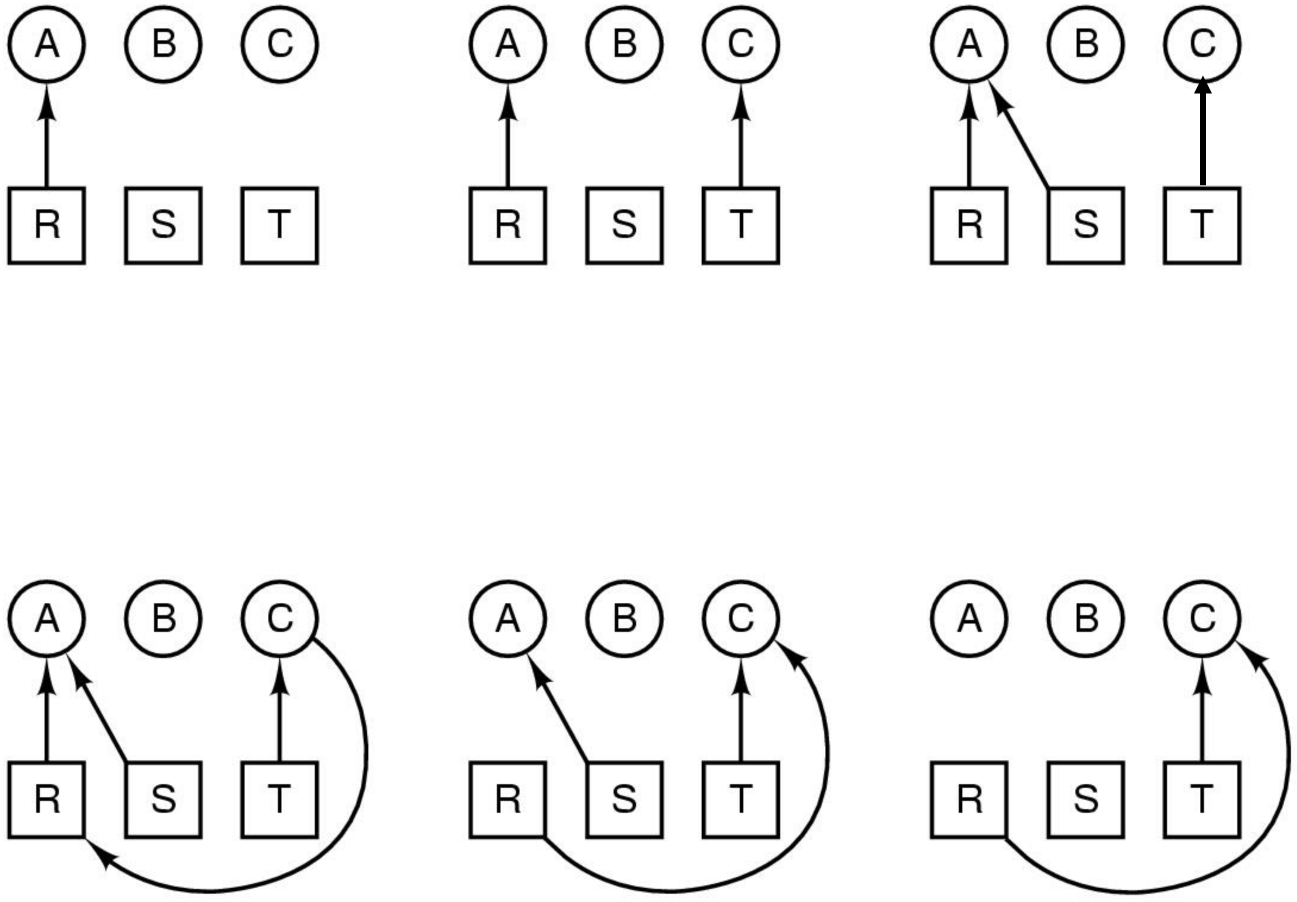


1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

If the OS knows about the deadlock, it could suspend B instead of granting it S.

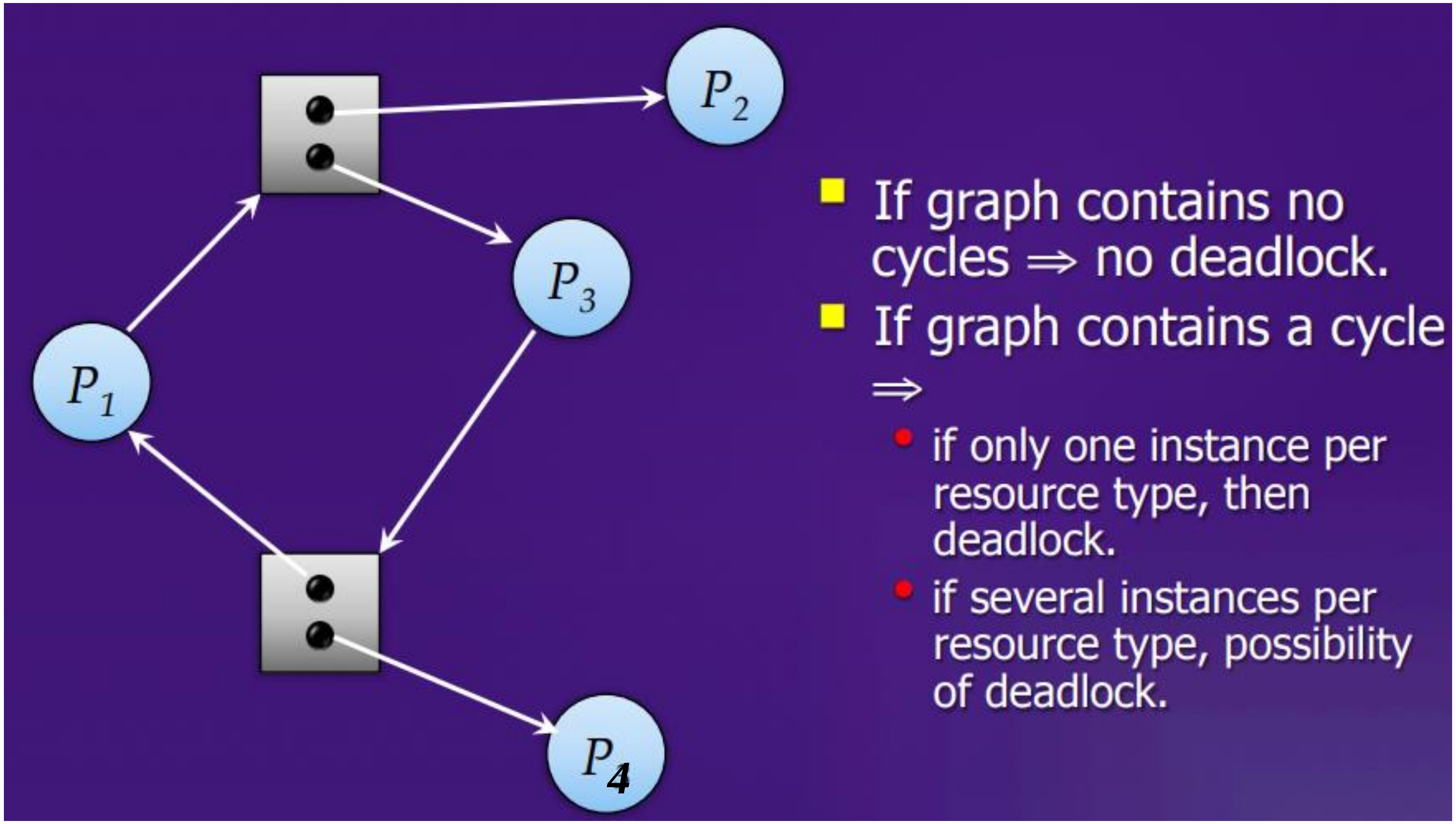
Deadlock Modeling: Resource-Allocation Graphs

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



How deadlock can be avoided

Deadlock Modeling: Resource-Allocation Graphs



- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

How deadlock can be avoided

Methods for Handling Deadlocks

1. Just ignore the problem altogether
2. Detection and recovery

    - Let them occur, detect them, and take action.

3. Dynamic avoidance

    - By careful resource allocation.

4. Prevention

    - Negating one of the four necessary conditions

1. Ignore the problem

The Ostrich Algorithm

- How often system crashes for other reasons, and how serious a deadlock is?

- If deadlocks occur once every five years, but system crashes due to h/w failures, compiler errors, and OS bugs occur once a week

  → focus on these failures, ignore the deadlock.

Ostrich

- Ostriches are large flightless birds.
- They are the heaviest living birds.
- They lay the largest eggs of any living land animal.
- The ability to run at 70 km/h (43.5 mph), they are the fastest birds on land.

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX and Windows follow this approach
- It is a trade off between
  - convenience
  - correctness

Everything just fine

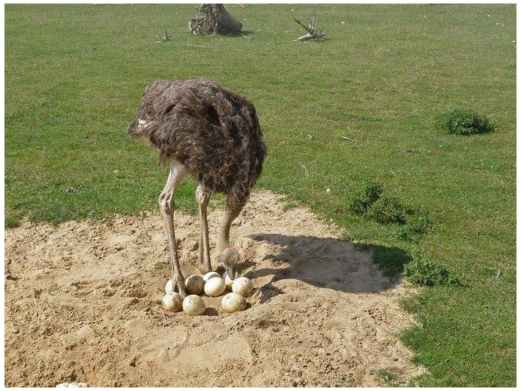
Do ostriches really bury their heads in the sand?

Do ostriches really bury their heads in the sand?

Contrary to the popular myth, ostriches do not bury their head in the sand when scared or frightened.

Ostriches dig shallow holes in the sand to serve as nests for their eggs.

The ostrich will use its beak several times a day to turn the eggs in the nest, creating the illusion of burying its head in the sand.



2. Detection and recovery

Detection and recovery

- Does not attempt to prevent deadlock from occurring.
- Instead, let them occur, tries to detect when this happens, and then takes some action to recover.

Detection with One Resource of Each Type

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*.

Detection with One Resource of Each Type

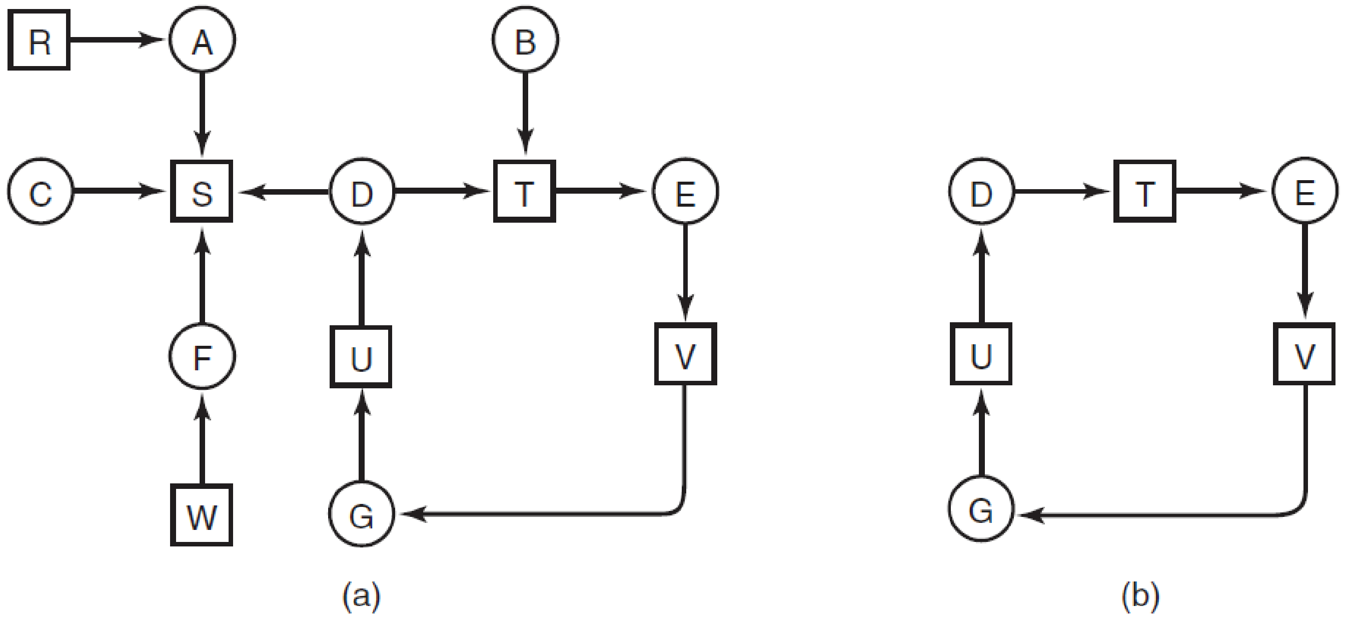


Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

A cycle within the graph → deadlock

Detection with Multiple Resource 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

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

E – total number of resources

Data structures needed by deadlock detection algorithm

Detection with Multiple Resource of Each Type

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

Current allocation matrix

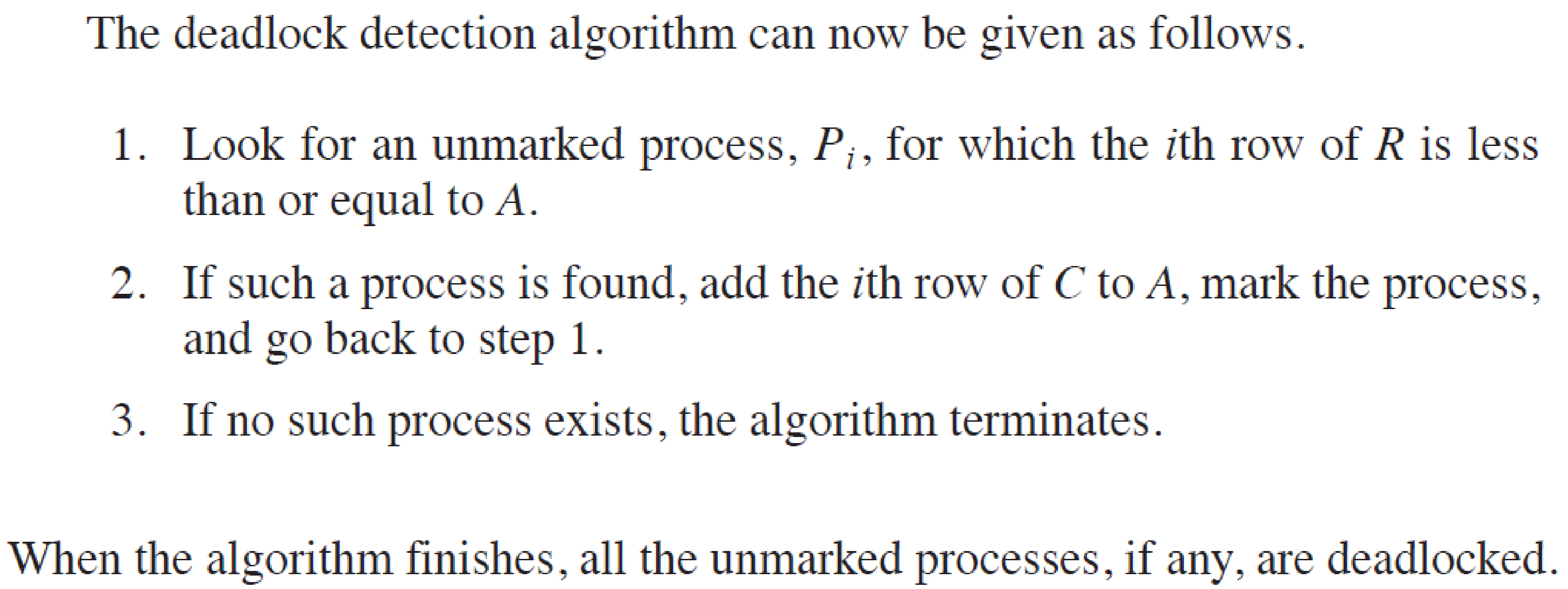
$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

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

Detection with Multiple Resource of Each Type



The deadlock detection algorithm can now be given as follows.

1. Look for an unmarked process, P_i, for which the ith row of R is less than or equal to A.
2. If such a process is found, add the ith row of C to A, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

Detection with Multiple Resource of Each Type

- 1st cannot be satisfied as there is no CD-ROM available
- 2nd cannot be satisfied, no scanner is free
- 3rd can be satisfied, so process 3 runs and returns all of its resources.

$$A = [2,2,2,0]$$

- Now process 2 can run, returns its resources.

$$A = [4,2,2,1]$$

- Now remaining process can run $\rightarrow$ no deadlock

Detection with Multiple Resource of Each Type

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

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

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

An example for the deadlock detection algorithm

Detection with Multiple Resource of Each Type

- 1st cannot be satisfied as there is no CD-ROM available
- 2nd cannot be satisfied, no scanner is free
- 3rd can be satisfied, so process 3 runs and returns all of its resources.

$$A = [2,2,2,0]$$

- Now also process 1 and 2 cannot be satisfied.

- Two processes cannot run → deadlock

Recovery from Deadlock

1. Recovery through preemption
   - take a resource from some other process
   - depends on nature of the resource
     - E.g. CD-ROM cannot be taken away

Recovery from Deadlock

2. Recovery through rollback

- checkpoint a process periodically
  - Write the state of the process to a file, so that it can be restarted from that state.
  - Contains memory image, resource state – which resources are currently assigned to that process.
- use this saved state to restart the process if it is found deadlocked

Recovery from Deadlock

3. Recovery through killing processes

– crudest but simplest way to break a deadlock
– no need to checkpoint (save the state) periodically
– kill one of the processes in the deadlock cycle
– the other processes get its resources
– choose process that can be rerun from the beginning

3. Dynamic Avoidance

Dynamic Avoidance

- System must be able to decide whether granting a resource is safe or not, and only make the allocation if it is safe.
- Needs information available in advance (which process needs which resources, or the maximum number of resources required in each type)
  - E.g. A and B both need printer and plotter

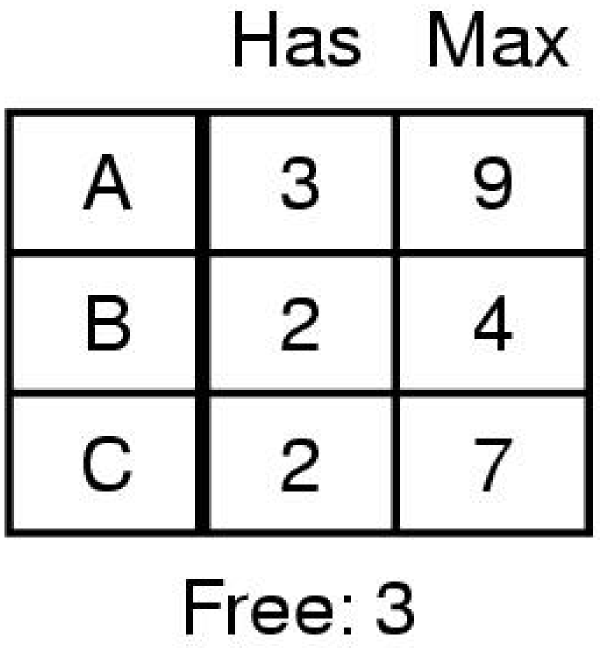
Bankers algorithm

- A well known deadlock avoidance algorithm
- Banker's behavior :
  - Clients are asking for loans up-to an agreed limit
  - Not all clients need their limit simultaneously, they will request time to time
  - All clients must achieve their limits at some point of time but not necessarily simultaneously
  - After fulfilling their needs, the clients will pay-back their loans
  - Banker will check to see,
    - If granting the request leads to an unsafe state. If so, the request is denied.
    - If granting the request leads to a safe state, it is carried out.

Bankers algorithm – safe and unsafe states

A state is said to be **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 resources immediately.

- Processes - A,B,C
- Has – the no. of resources (single type) each process has
- Max – maximum need for each process
- Free – no. of resources available in addition to the resources each process has



	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

Bankers algorithm – safe and unsafe states

	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

Bankers algorithm – 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

Bankers algorithm

- Example:
  - The banker knows that all 4 clients need 22 units together, but he has only total 10 units

Client	Used	Max.
Adam	0	6
Eve	0	5
Joe	0	4
Mary	0	7

Available: 10

State (a)

Client	Used	Max.
Adam	1	6
Eve	1	5
Joe	2	4
Mary	4	7

Available: 2

State (b)

Client	Used	Max.
Adam	1	6
Eve	2	5
Joe	2	4
Mary	4	7

Available: 1

State (c)

- The advanced knowledge required is the maximum number of units of each type of resource that the process (clients) will claim at any one time.
- Any process which requests an allocation beyond its pre-declared maximum - will be aborted.

Banker's Algorithm for Multiple Resources

- E – Total no of resources exists
- P – Possessed resources
- A – Available resources
- N – Need 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

P

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

N

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

$E = P + A$

Banker's Algorithm for Multiple Resources

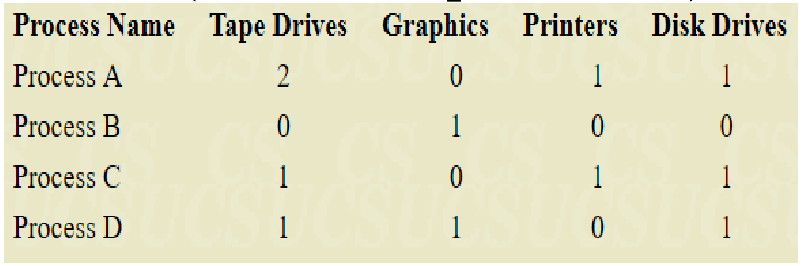
The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, *R*, whose unmet resource needs are all smaller than or equal to *A*. If no such row exists, the system will eventually deadlock since no process can run to completion (assuming processes keep all resources until they exit).
2. Assume the process of the chosen row requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all of its resources to the *A* vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated (in which case the initial state was safe) or no process is left whose resource needs can be met (in which case the system was not safe).

Banker's Algorithm for Multiple Resources

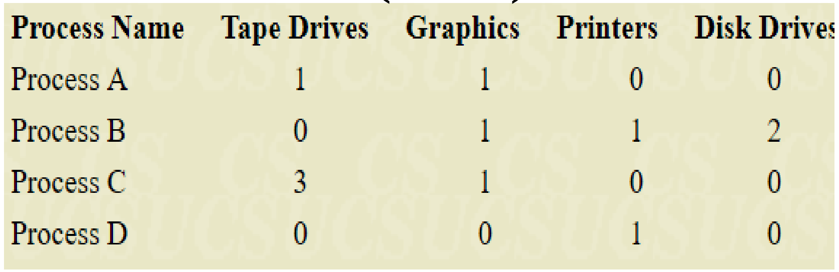
Exercise:

P (allocated/possessed)



Process Name	Tape Drives	Graphics	Printers	Disk Drives
Process A	2	0	1	1
Process B	0	1	0	0
Process C	1	0	1	1
Process D	1	1	0	1

N (need)



Process Name	Tape Drives	Graphics	Printers	Disk Drives
Process A	1	1	0	0
Process B	0	1	1	2
Process C	3	1	0	0
Process D	0	0	1	0

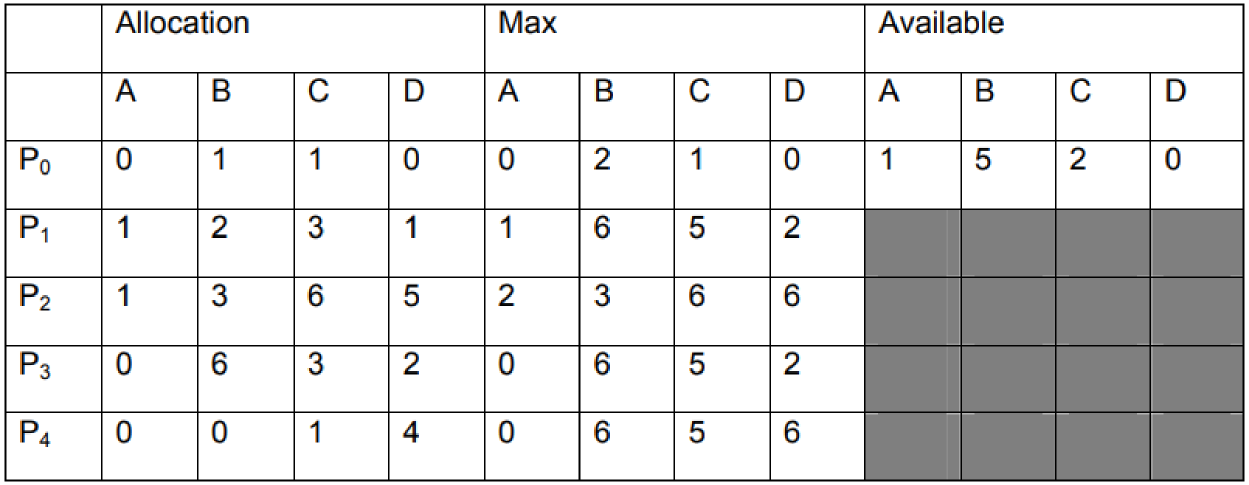
A (available) = [1,0,2,0]

- Find the total no of existing resources (E) = ? E = [5,2,4,3]
- Is this state safe/not?

Safe.
D can complete → A = [2,1,2,1]
A can complete → A = [4,1,3,2]
B can complete → A = [4,2,3,2]
C can complete → A = [5,2,4,3]

Banker's Algorithm for Multiple Resources

Exercise:

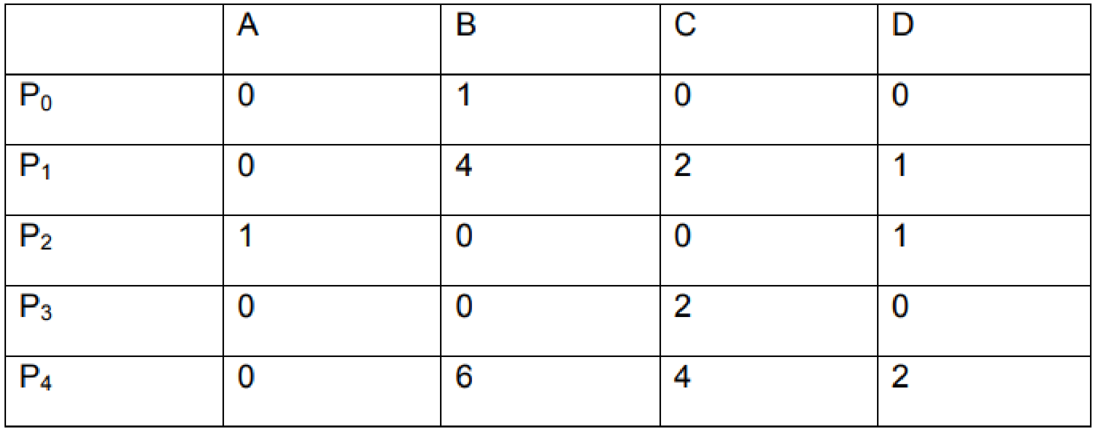


	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Safe.
P0 → A = [1,6,3,0]
P3 → A = [1,12,6,2]
P4 → A = [1,12,7,6]
P1 → A = [2,14,10,7]
P2 → A = [3,17,16,12]
OR

P0 → A = [1,6,3,0]
P3 → A = [1,12,6,2]
P1 → A = [2,14,9,3]
P2 → A = [3,17,15,8]
P4 → A = [3,17,16,12]
…

Create the need matrix (max-allocation)

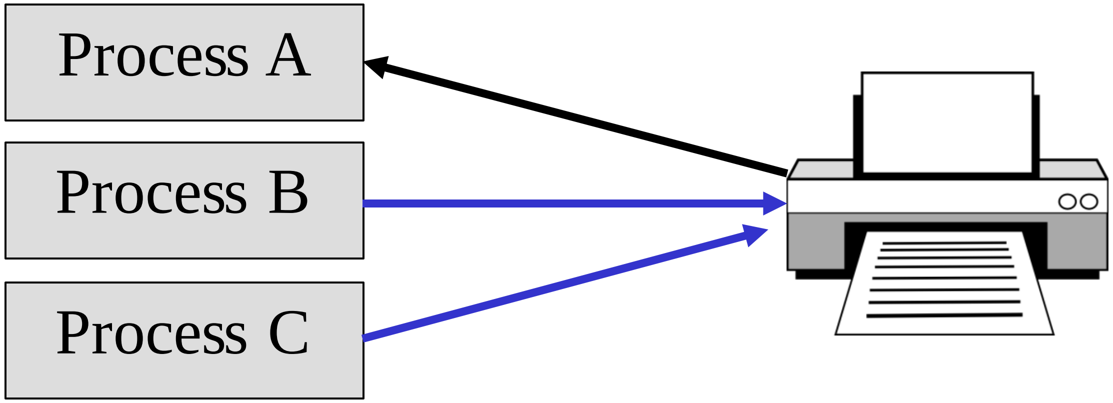


	A	B	C	D
P_0	0	1	0	0
P_1	0	4	2	1
P_2	1	0	0	1
P_3	0	0	2	0
P_4	0	6	4	2

Check LMS for more exercises

4. Deadlock Prevention

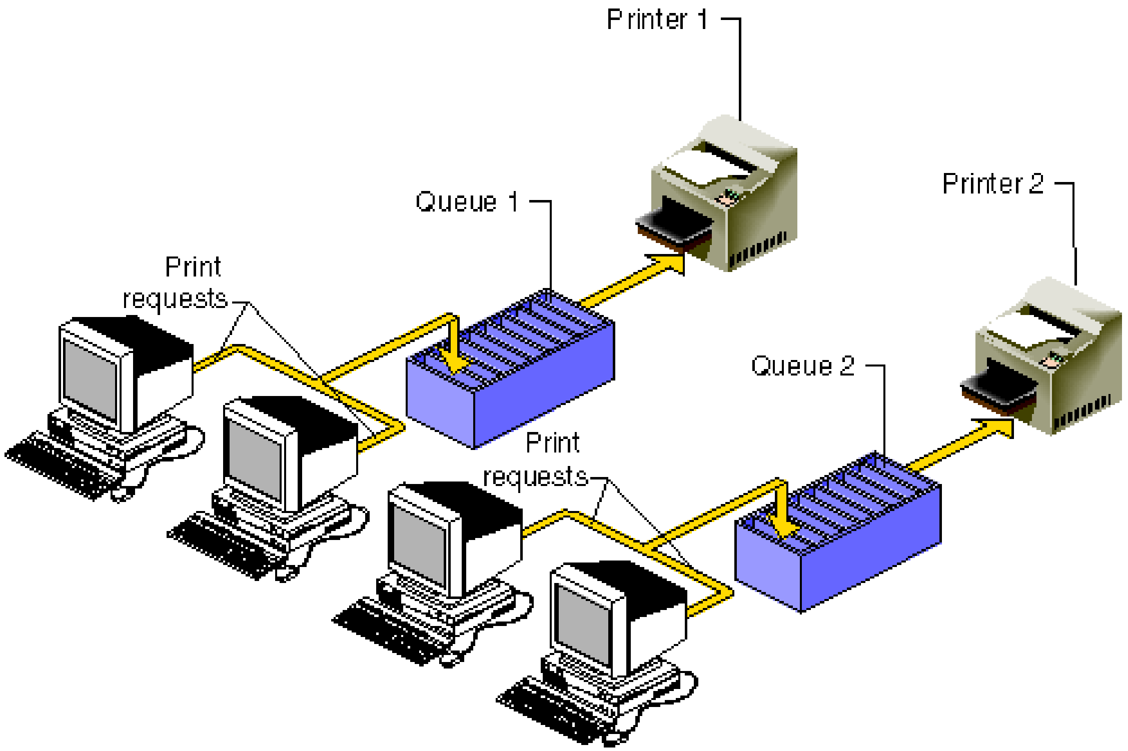
1. Attacking the Mutual Exclusion Condition



When A holds the printer, B and C have to wait.

Spooling

- send (data that is intended for printing or processing on a peripheral device) to an intermediate store.
- multiple processes can produce outputs simultaneously



1. 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 resources when not absolutely necessary

2.Attacking the Hold and Wait Condition

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

Problems:

- May not know required resources at the start of run
  - if they know, the bankers algorithm could be used to avoid deadlock while they are running
- If one or more resources are busy, nothing will be allocated and the process will just wait
- Resources may not be optimally used.
  - E.g. A process reads data from input tape, analyse it for an hour, write the output to a tape, and plot the results. If all the resources are requested in advance, the resources (tape drive, and the plotter) will be wasted for an hour.

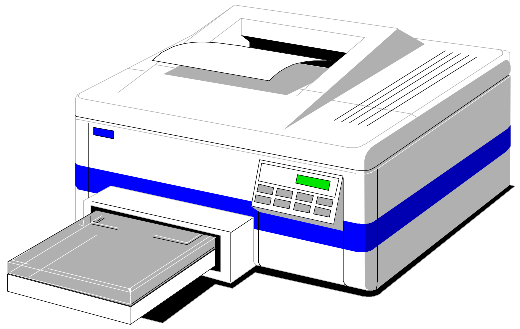
2.Attacking the Hold and Wait Condition

2. Request for a resource only when it is not holing any other resource

    – first temporarily release all the resources it currently holds.
    – Then it tries to get everything it needs all at once.

3. 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
    - !!??



- A possible scenario
  - Process A needs more memory, CPU preempts process B from memory and allocates this memory to process A.

4. Attacking the Circular Wait Condition

- Can be eliminated in several ways

1. A process can have only one resource at a time. If it needs the second one, release the first one.

   Eg. for not possible – A process needs to copy a huge file from the disk to the printer – it needs both resources at the same time.

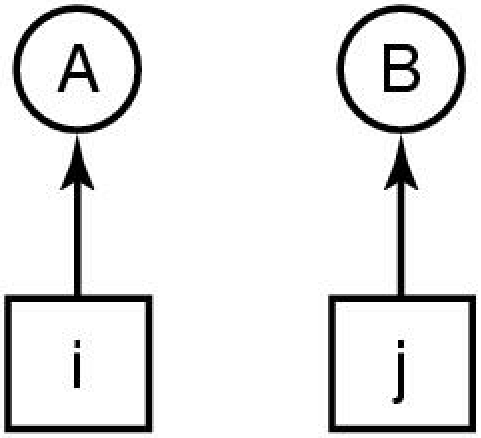
4. Attacking the Circular Wait Condition

2. Global numbering for the resources

- Process can request resources whenever they want to,
- But all the requests must be made in numerical order.

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

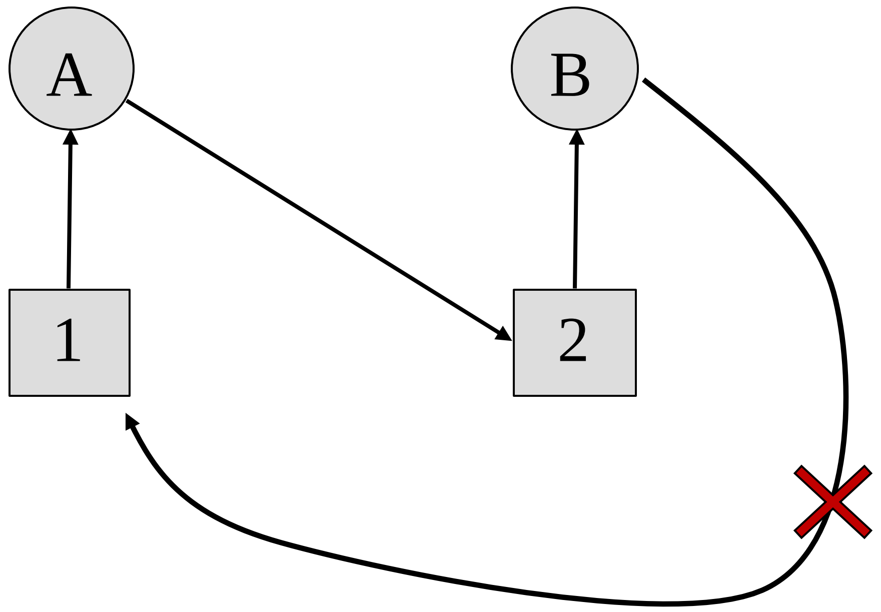
(a)



(b)

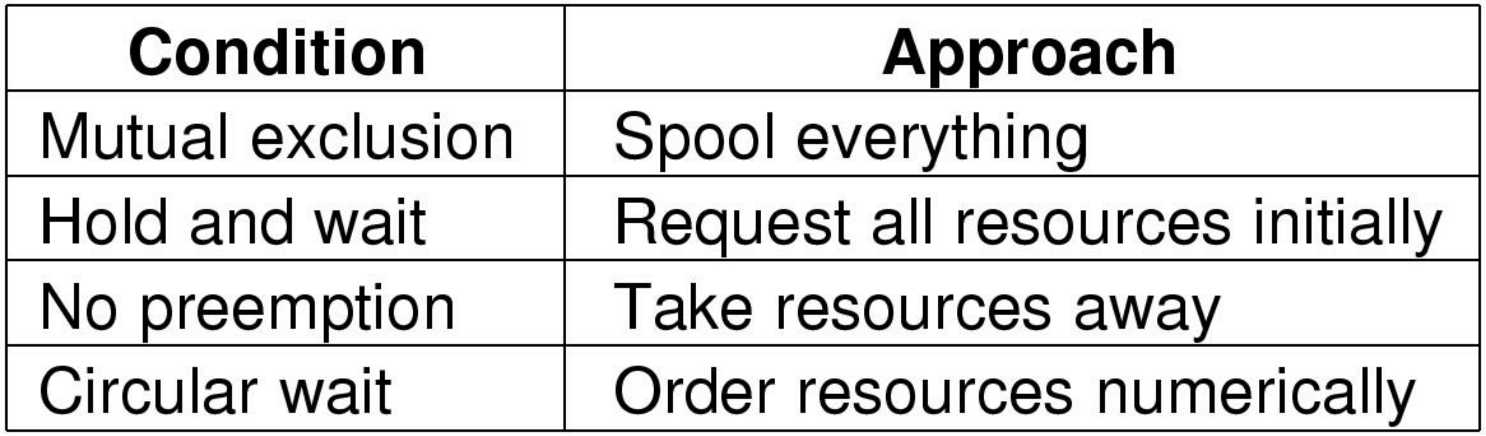
A process may request first a plotter and a tape drive,
but it cannot request first a plotter and then a scanner

4. Attacking the Circular Wait Condition



A process cannot request first i and then j, if i>j.

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

Starvation

- Indefinite postponement of a process because it requires some resource before it can run, but the resource, though available for allocation, is never allocated to this process.

- Example: Allocation of CPU in Priority Scheduling: the low priority process may never get the chance to run

Running Java Thread
Starving Thread
Higher Priority Threads waiting...

Deadlock vs. Starvation

- Starvation occurs when a process waits for a resource that becomes available continuously, but it is not allocated to that process.
- In starvation there are chances that the victim process will get the requested resources.
- But in deadlocked scenario processes are permanently blocked because required resource never become available.
- In starvation there is a progress (apart from the victim), but in deadlock no progress.