METU - EE 442 - Operating Systems

Deadlocks

Chapter 6

Cüneyt F. Bazlamaçcı (METU-EEE)

The slides include material from Tanenbaum & Bos, Modern Operating Systems, 4th ed., Pearson and Halıcı, U., EE442 Lecture Notes

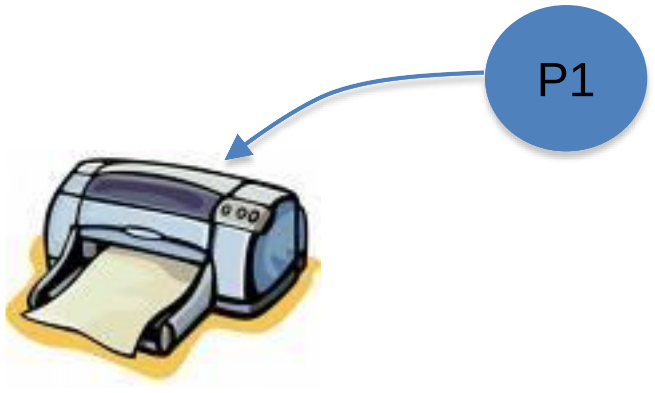
Deadlocks

- In a multiprogramming system, processes request resources.
- If those resources are being used by other processes then the process enters a waiting state.
- However, if other processes are also in a waiting state, we have ***deadlock***.

Definition

- **Example**

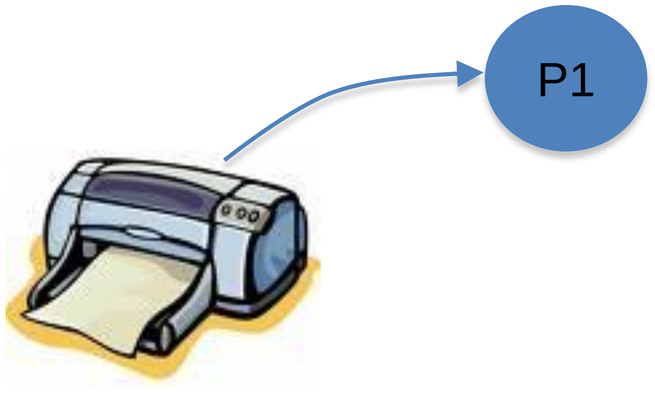
  Process-1 requests the printer



Definition

- **Example**

  Process-1 requests the printer, gets it

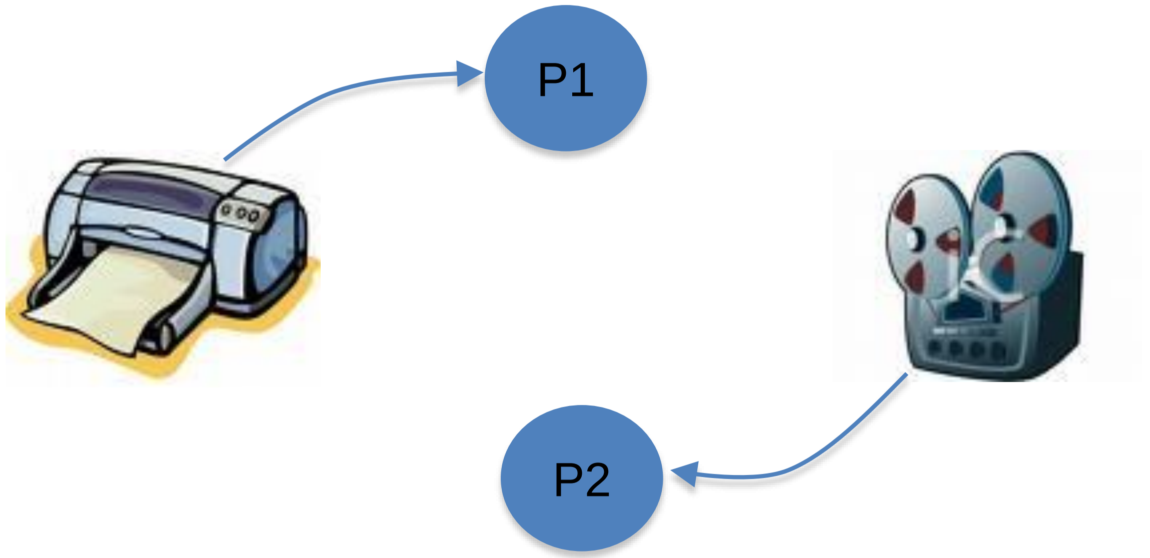


Definition

- **Example**

  Process-1 requests the printer, gets it

  Process-2 requests the tape unit, gets it



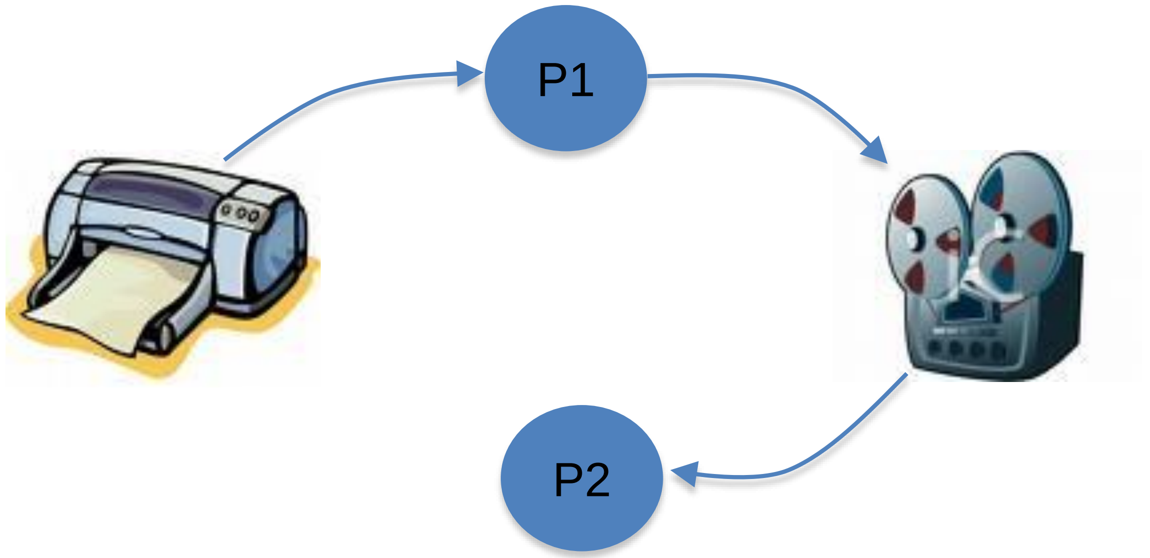
Definition

- **Example**

  Process-1 requests the printer, gets it

  Process-2 requests the tape unit, gets it

  Process-1 requests the tape unit, waits



Definition

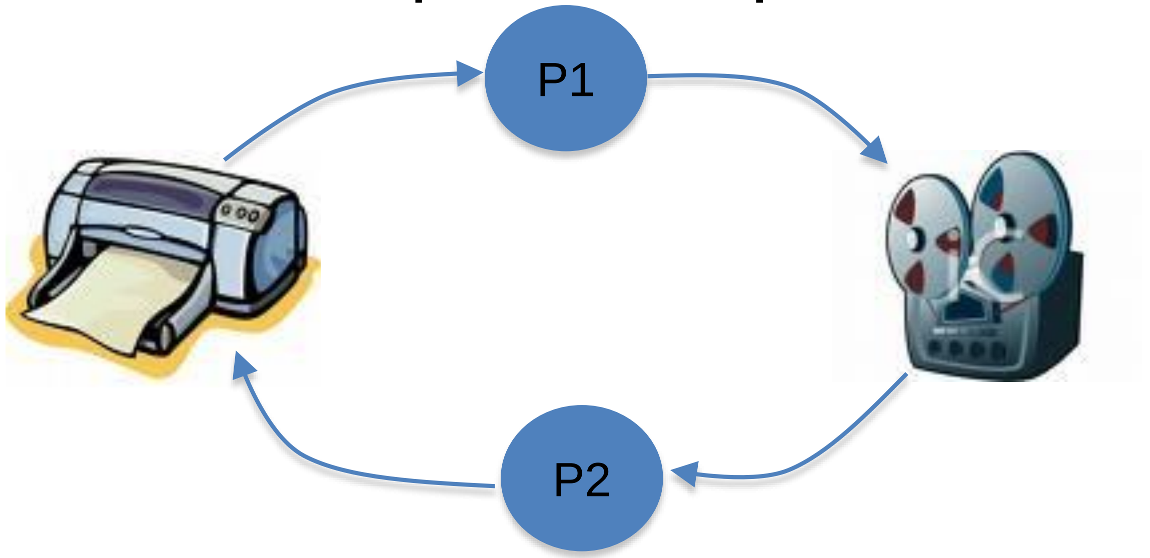
- **Example**

  Process-1 requests the printer, gets it

  Process-2 requests the tape unit, gets it

  Process-1 requests the tape unit, waits

  Process-2 requests the printer, waits



Definition

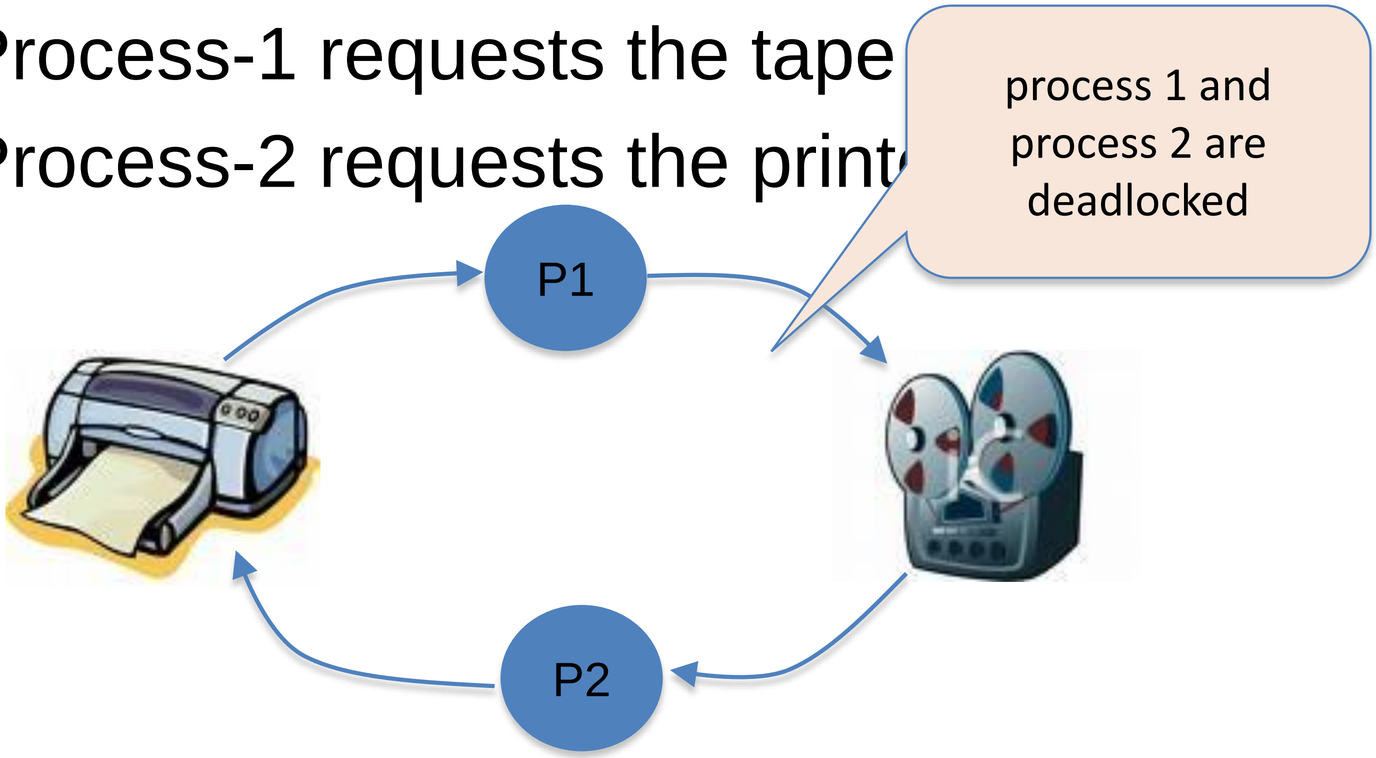
- **Example**

  Process-1 requests the printer, gets it

  Process-2 requests the tape unit, gets it

  Process-1 requests the tape

  Process-2 requests the print



Assumptions

In this chapter, we shall analyze deadlocks with the following assumptions:

- A process must request a resource before using it.
- It must release the resource after using it.

  (request $\rightarrow$ use $\rightarrow$ release)

- A process cannot request a number more than the total number of resources available in the system

Assumptions

- If the resource is not available when it is requested, the requesting process is forced to wait.
- In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available.
- In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.
- The exact nature of requesting a resource is highly system dependent.

  In some systems, a request system call is provided to allow processes to explicitly ask for resources. In others, the only resources that the OS knows about are special files that only one process can open at a time. These are opened by the usual open call.

Resource Acquisition (1)

```
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)

Using a semaphore to protect resources.
(a) One resource. (b) Two resources.

Resource Acquisition (2)

```
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)

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

Definition

- The formal definition of deadlock is as follows:

- **Definition:** A set of processes is in a *deadlock state* if
  - every process in the set is waiting for an event (release)
  - that event can only be caused by some other process in the same set.

- For this model, we assume that processes are single threaded and that no interrupts are possible to wake up a blocked process.

Four deadlock conditions

- A <u>deadlock</u> occurs <u>if and only</u> if the following four conditions hold in a system <u>simultaneously</u>:

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

Four deadlock conditions

1. <u>Mutual Exclusion:</u>

At least one of the resources is non-sharable

- Each resource is either available or assigned to exactly one process.

- if it is requested by a process while it is being used by another one, the requesting process has to wait until the resource is released.

Four deadlock conditions

2. <u>Hold and Wait:</u>

- Processes currently holding resources that were granted earlier can request new resources.

- There should be at least one process that is holding at least one resource and waiting for other resources that are being hold by other processes.

Four deadlock conditions

3. <u>No Preemption:</u>

No resource can be preempted before the holding process completes its task with that resource.

Resources must be explicitly released by the process holding them.

Four deadlock conditions

4. Circular Wait:

There exists a set of processes:

$\{ P_1, P_2, ..., P_n \}$ such that

P_1 is waiting for a resource held by P_2

P_2 is waiting for a resource held by P_3

...

P_{n-1} is waiting for a resource held by P_n

P_n is waiting for a resource held by P_1

- There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.

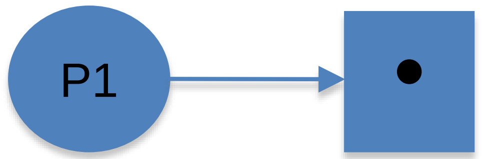
Resource Allocation Graphs

- Resource allocation graphs are drawn in order to see the allocation relations of processes and resources easily.

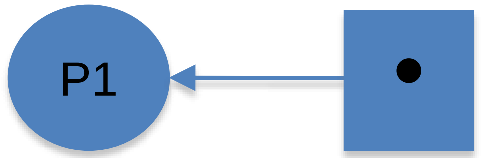
- In these graphs,
  - processes are represented by circles
  - resources are represented by boxes.

- Resource boxes have some number of dots inside
  - indicating available number of that resource, that is number of instances.

Resource Allocation Graphs

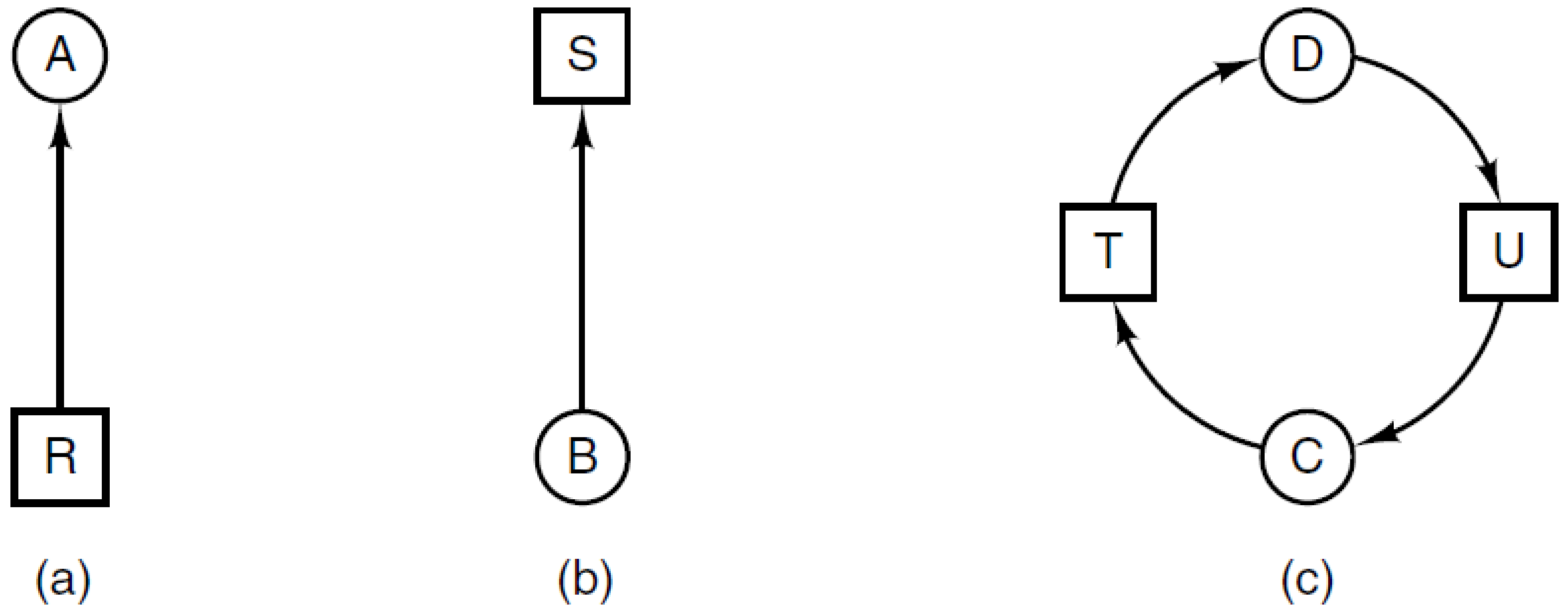


Process p_i is waiting for resource r_j



Resource r_j has been allocated to Process p_i

Deadlock Modeling (1)



Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

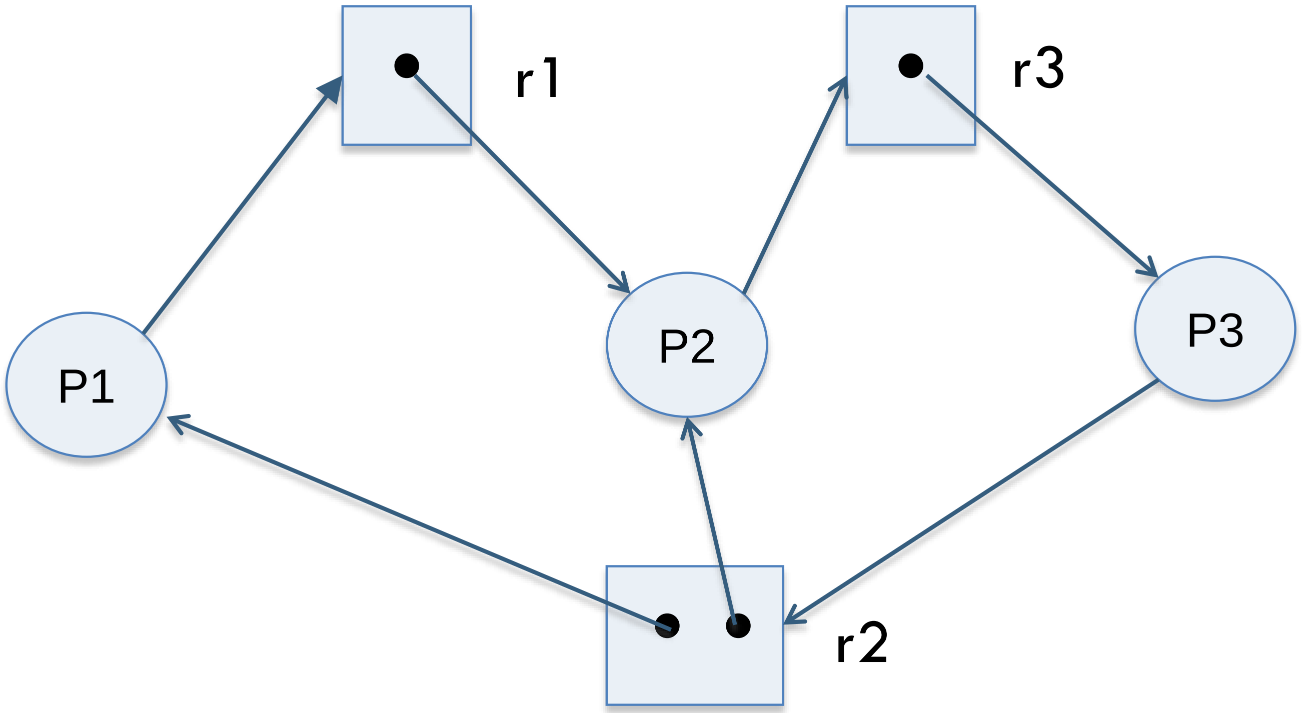
Resource Allocation Graphs

- If the resource allocation graph contains <u>no cycles</u> then there is <u>no deadlock</u> in the system at that instance.

- If the resource allocation graph contains <u>a cycle</u> then a <u>deadlock may exist</u>.

- If there is a cycle, and the cycle involves only resources which have a single instance, then a deadlock has occurred.

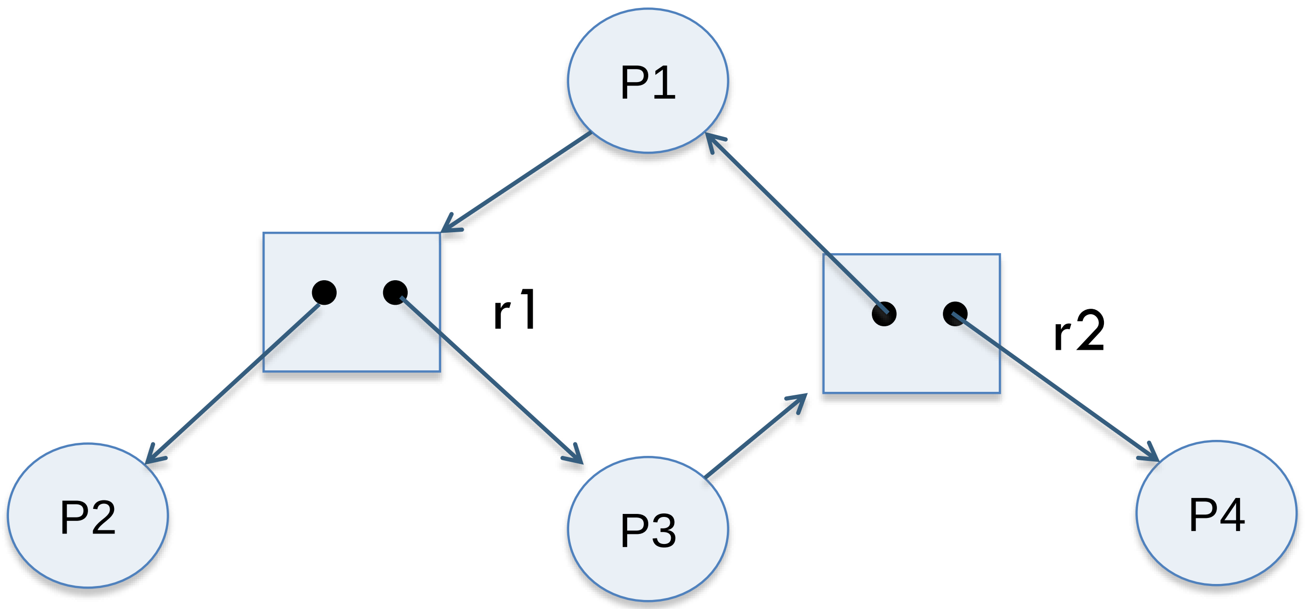
Resource Allocation Graphs



There are two cycles, so a deadlock may exists.

Actually p1, p2 and p3 are deadlocked

Resource Allocation Graphs



There is a cycle, however there is no deadlock.

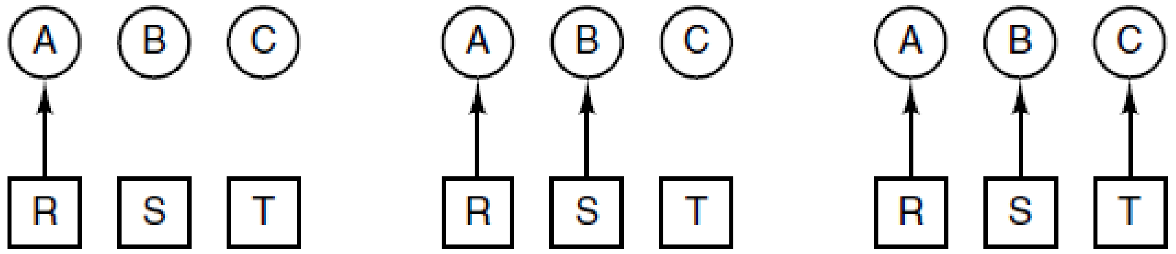
If p4 releases r2, r2 may be allocated to p3, which breaks the cycle.

Deadlock Modeling (2)

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

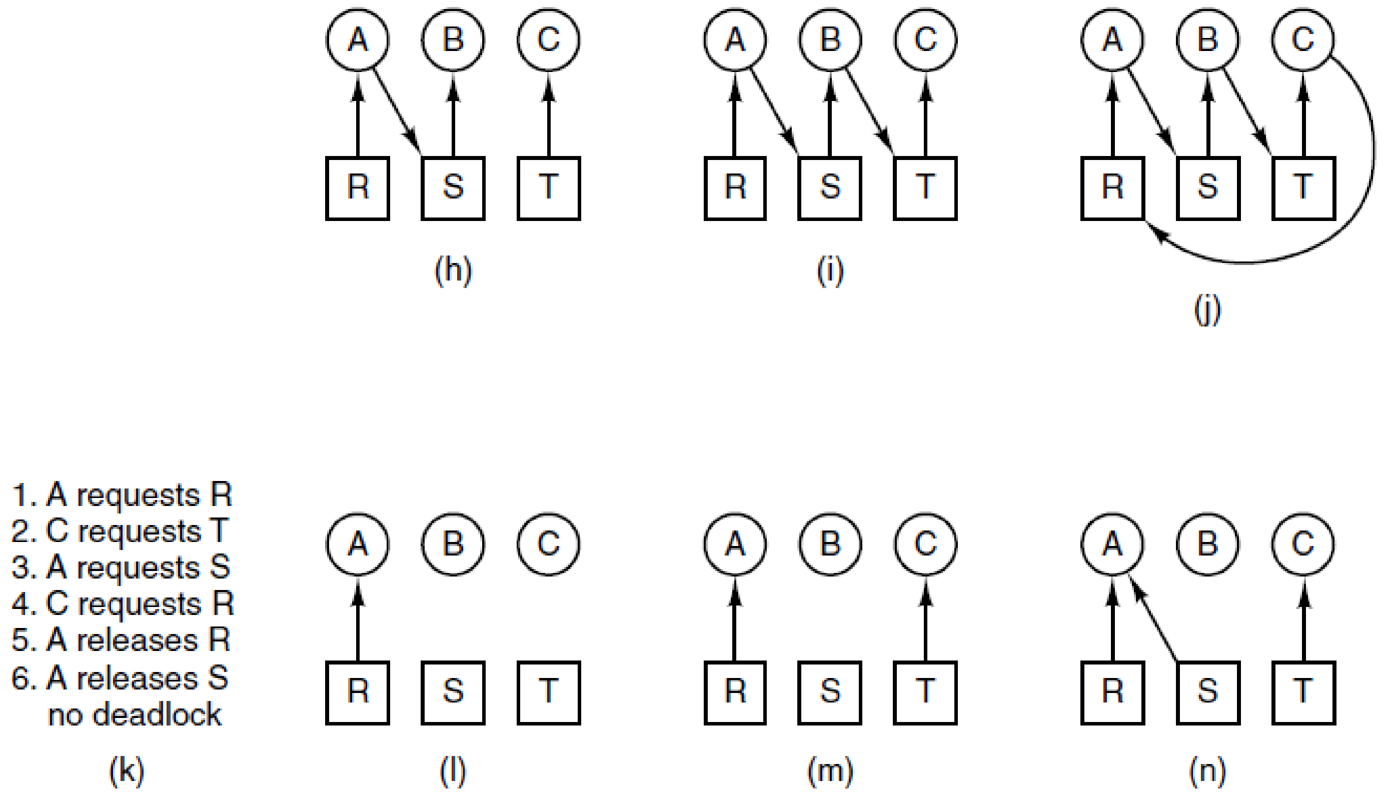
(d)



(e) (f) (g)

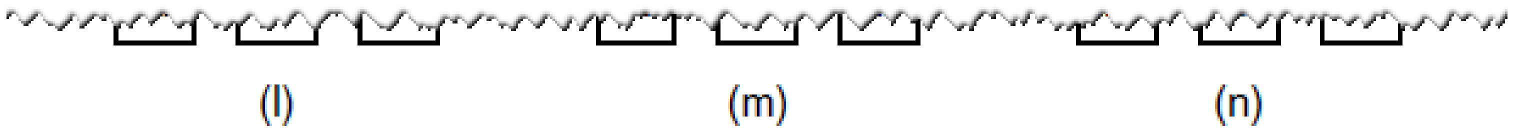
An example of how deadlock occurs and how it can be avoided.

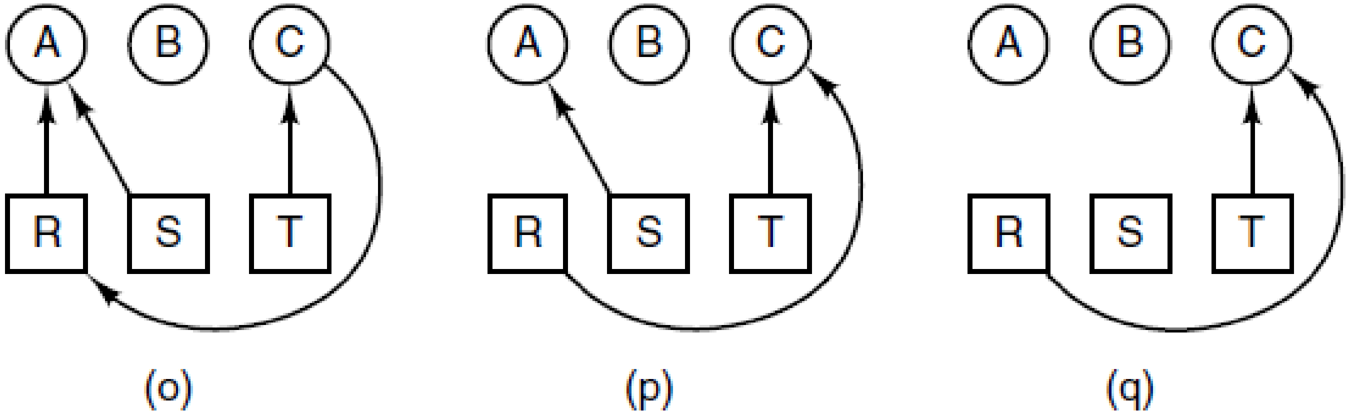
Deadlock Modeling (3)



An example of how deadlock occurs and how it can be avoided.

Deadlock Modeling (4)





An example of how deadlock occurs and how it can be avoided.

Deadlock Modeling (5)

Strategies are used for dealing with deadlocks:

1. Prevention, by structurally negating one of the four required conditions
2. Dynamic avoidance by careful resource allocation.
3. Detection and recovery. Let deadlocks occur, detect them, and take action.
4. Ignore the problem, maybe it will go away (the ostrich alg.).

Methods for handling deadlocks

- Deadlock prevention
  - Preventing deadlocks by constraining how requests for resources can be made in the system and how they are handled (system design).
  - The goal is to ensure that at least one of the necessary conditions for deadlock can never hold.
- Deadlock avoidance
  - The system dynamically considers every request and decides whether it is safe to grant it at this point,
  - The system requires additional apriori information regarding the overall potential use of each resource for each process.
  - Allows more concurrency.
- Deadlock detection and recovery.

Deadlock Prevention

Assure that at least one of conditions is never satisfied

- Mutual exclusion
- Hold and wait
- No Preemption
- Circular wait

Deadlock Prevention

- To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be discarded.

  The methods for preventing those conditions to hold are as follows:

- <u>Mutual Exclusion</u>:
  - In general, we do not have systems with all resources being sharable.
  - Some resources such as printers may be non-sharable.
  - So it may not be possible to prevent deadlocks by denying mutual exclusion

Deadlock Prevention

- Hold and Wait:
  - One protocol to ensure that hold-and-wait condition never occurs says
    - "each process must request and get all of its resources before it begins execution".
  - Another protocol is
    - "Each process can request resources only when it does not occupy any resources."

  The second protocol is better. However, both protocols cause low resource utilization and starvation.
  - Many resources are allocated but most of them are unused for a long period of time.
  - A process that requests several commonly used resources causes many others to wait indefinitely.

Deadlock Prevention

- <u>No Preemption</u>:
  - One protocol is
    - “If a process that is holding some resources requests another resource and that resource cannot be allocated to it, then it must release all resources that are currently allocated to it.”
  - Another protocol is
    - “When a process requests some resources,
      - if they are available, allocate them.
      - If a resource it requested is not available, then
        - » check whether it is being used or it is allocated to some other process waiting for other resources.
        - » If it is used, the requesting process must wait.
        - » If not, then the OS preempts it from the waiting process and allocates it to the requesting process.
  - This protocol can be applied to resources whose states can easily be saved and restored (registers, memory space).

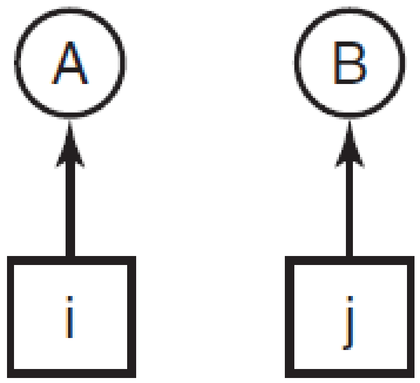
Deadlock Prevention

- Circular Wait:
  - One protocol to ensure that the circular wait condition never holds is
    - “Impose a linear ordering of all resource types.”
    - Then, each process can only request resources in an increasing order of priority.
    - For example, set priorities for r1 = 1, r2 = 2, r3 = 3, and r4 = 4. With these priorities, if process P wants to use r1 and r3, it should first request r1, then r3.
  - Another protocol is
    - “Whenever a process requests a resource r_j, it must have released all resources r_k with priority(r_k) $\geq$ priority (r_j).

Attacking Circular Wait Condition

1. Imagesetter
2. Printer
3. Plotter
4. Tape drive
5. CD-ROM drive

(a)



(b)

(a) Numerically ordered resources.
(b) A resource graph

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

Summary of approaches to deadlock prevention.

Deadlock avoidance

- Given some additional information on how each process will request resources, it is possible to construct an algorithm that will avoid deadlock states.

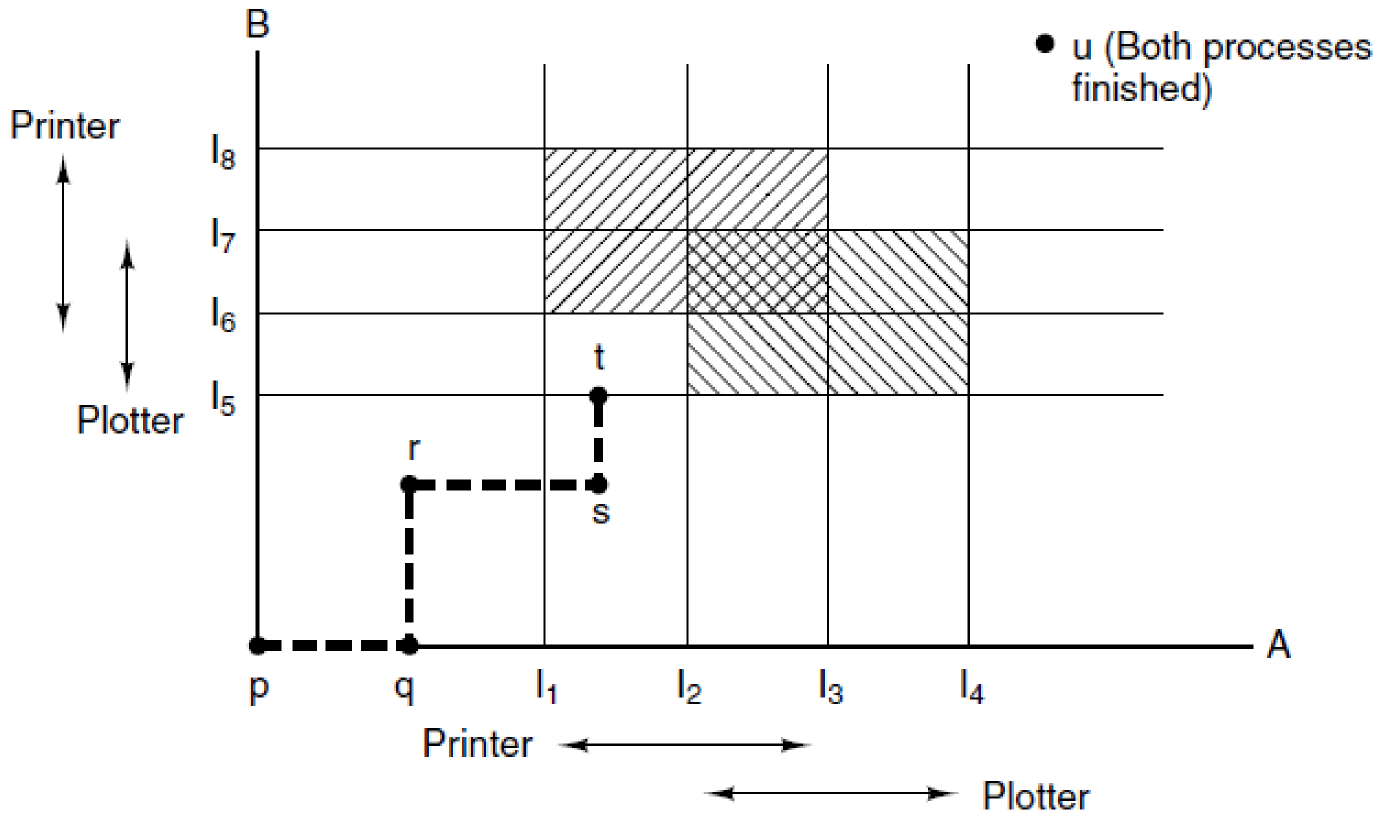
- The algorithm can dynamically examine the resource allocation operations to ensure that there won't be a circular wait on resources.

- When a process requests a resource that is already available, the system must decide whether that resource can immediately be allocated or not.

- The resource is immediately allocated only if it leaves the system in a *safe state*.

- A state is ***safe*** if the system can allocate resources to each process in some order avoiding a deadlock. On other words, a safe state is one in which there exists a sequence of events that guarantee that all processes can finish.

Deadlock Avoidance Resource Trajectories



Two process resource trajectories.

Safe and Unsafe States (1)

	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)

	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.

Note that an unsafe state is not a deadlocked state.

The difference between a safe state and an 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.

Deadlock avoidance

- Consider a system with 12 tape drives.
- Assume there are three processes: p1, p2, p3.
- Assume we know the maximum number of tape drives that each process may request:

  p1 : 10,   p2 : 4,   p3 : 9
- Suppose at time t_{now}, 9 tape drives are allocated as follows :

  p1 : 5,   p2 : 2,   p3 : 2
- So, we have three more tape drives which are free.

Deadlock avoidance

Max:	p1 : 10,	p2 : 4,	p3 : 9
Allocation:	p1 : 5,	p2 : 2,	p3 : 2
Available:	3		

- This system is in a safe state because we sequence processes as <p2,p1,p3>:
  - p2 can get two more tape drives and it finishes its job, and returns four tape drives to the system. Then the system will have 5 free tape drives.
  - Allocate all of them to p1, it gets 10 tape drives and finishes its job. p1 then returns all 10 drives to the system.
  - Then p3 can get 7 more tape drives and it does its job.

Deadlock avoidance

- It is possible to go from a safe state to an unsafe state:
- Consider the above example.
- At time t_{now+1}, p3 requests one more tape drive and gets it. Now, the system is in an unsafe state because

Max:	p1 : 10,	p2 : 4,	p3 : 9
Allocation:	p1 : 5,	p2 : 2,	p3 : 3
Available:	2		

- There are two free tape drives, so only p2 can be allocated all its tape drives. When it finishes and returns all 4 tape drives, the system will have four free tape drives.
  - p1 is allocated 5, may request 5 more → has to wait
  - p3 is allocated 3, may request 6 more → has to wait
- We allocated p3 one more tape drive at t_{now+1} and this caused an unsafe state.

Banker's Algorithm for 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 (2)

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, system will eventually deadlock.
2. Assume the process of row chosen requests all resources needed and finishes. Mark that process as terminated, add its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated (safe state) or no process is left whose resource needs can be met (deadlock)

Banker's Algorithm

- It is a deadlock avoidance algorithm.

- The following data structures are used in the algorithm:

  m: number of resources

  n : number of processes

  Available[m]: One dimensional array of size m.

  - It indicates the number of available resources of each type.
  - if Available[i]=k then there are k instances of resource r_i.

  Max[n,m]: Two dimensional array of size n*m.

  - It defines the maximum demand of each process from each resource type.
  - if Max[i,j]=k then process p_i may request at most k instances of resource type r_j.

Banker's Algorithm

Allocation[n,m]: Two dimensional array of size n*m.

- It defines the number of resources of each type currently allocated to each process.

Need[n,m] : Two dimensional array of size n*m.

- It indicates the remaining need of each process, of each resource type.
- If Need[i,j]=k, process p_i may need k more instances of resource type r_j.
- Note that Need[i,j] = Max[i,j] - Allocation[i,j].

Request[n,m] : Two dimensional array of size n*m.

- It indicates the pending requests of each process, of each resource type.

Banker's Algorithm

- Now, take each row vector in Allocation and Need as Allocation(i) and Need(i). (Allocation(i) specifies the resources currently allocated to process p_i.)

- Define the ≤ relation between two vectors X and Y, of equal size n as follows:

$$X \le Y \quad \Leftrightarrow \quad X[i] \le Y[i]\ ,\ i = 1,2,\ \ldots,\ n$$
$$X\ !\!\le Y \quad \Leftrightarrow \quad X[i] > Y[i] \text{ for some } i$$

Banker's Algorithm

The algorithm is as follows:

1. Process p_i makes requests for resources. Let Request(i) be the corresponding request vector.

   So, if p_i wants k instances of resource type r_j, then Request(i)[j] = k.
2. If Request(i) !≤ Need(i), there is an error.
3. Otherwise, if Request(i) !≤ Available, then p_i must wait.
4. Otherwise, Modify the data structures as follows :

   Available = Available - Request(i)

   Allocation(i) = Allocation(i) + Request(i)

   Need(i) = Need(i) - Request(i)
5. Check whether the resulting state is safe. (Use the safety algorithm presented below.)
6. If the state is safe, do the allocation. Otherwise, p_i must wait for Request(i).

Banker's Algorithm

Safety Algorithm to perform Step 5:

Let Work and Finish be vectors of length m and n, respectively.

```
1. Initialize Work = Available, Finish [j] = false, for all j.
2. Find an i such that
         Finish [ i ] = false and Need(i) ≤ Work
     If no such i is found, go to step 4.
3. If an i is found, then for that i, do :
     Work = Work + Allocation(i)
     Finish [i] = true
     Go to step 2.
4. If Finish [j] = true for all j, then the system is in a safe
     state.
```

Banker's Algorithm

- **Example**: *2 processes, 3 resources*

available

1	4	1

max

1	3	1
1	4	1

alloc

0	0	0
0	0	0

need

1	3	1
1	4	1

request

1	2	0
0	2	1

Request(1) is to be processed. If it is satisfied data would become:

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

work

0	2	1

finish

F
F

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

work

0	2	1

finish

F
F

i=1,
Need(1) = [0 1 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(1)
Finish (1) = true

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

work

0	2	1

finish

F
F

i=1,
Need(1) = [0 1 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(1)
Finish (1) = true

work

1	4	1

finish

T
F

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

work

0	2	1

finish

F
F

i=1,
Need(1) = [0 1 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(1)
Finish (1) = true

work

1	4	1

finish

T
F

i = 2 :
Need(2) = [1 4 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(2) = [1 4 1]
Finish (2)= true

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

work

0	2	1

finish

F
F

i=1,
Need(1) = [0 1 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(1)
Finish (1) = true

work

1	4	1

finish

T
F

i = 2 :
Need(2) = [1 4 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(2) = [1 4 1]
Finish (2)= true

work

1	4	1

finish

T
T

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

work

0	2	1

finish

F
F

i=1,
Need(1) = [0 1 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(1)
Finish (1) = true

work

1	4	1

finish

T
F

i = 2 :
Need(2) = [1 4 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(2) = [1 4 1]
Finish (2)= true

work

1	4	1

finish

T
T

Go to step 4
All true: System is in a safe state, so do the allocation.

Banker's Algorithm

available

0	2	1

alloc

1	2	0
0	0	0

need

0	1	1
1	4	1

Now, apply the safety algorithm:

work

0	2	1

finish

F
F

i=1,
Need(1) = [0 1 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(1)
Finish (1) = true

work

1	4	1

finish

T
F

i = 2 :
Need(2) = [1 4 1] $\leq$ Work ?   Yes.
Work = Work + Allocation(2) = [1 4 1]
Finish (2)= true

work

1	4	1

finish

T
T

Go to step 4.
All true: System is in a safe state, so do the allocation.

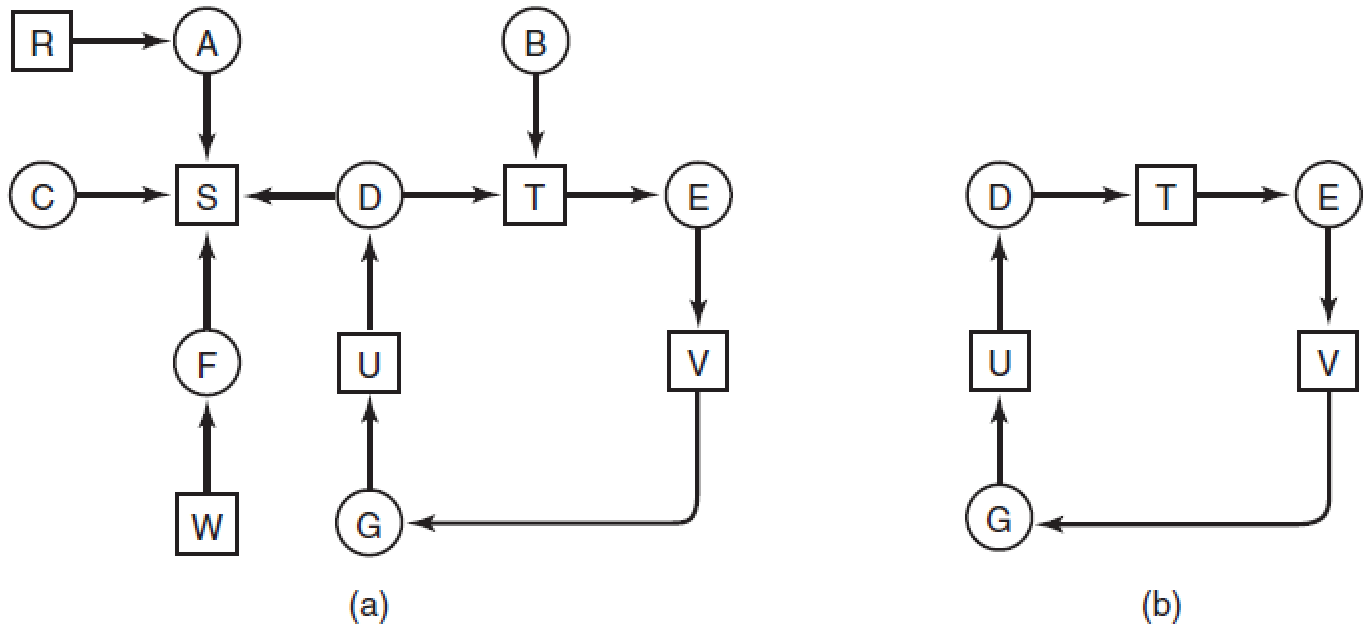
If the algorithm is repeated for Request(2), the system will end up in an unsafe state.

Deadlock Detection with One Resource of Each Type (1)

Example of a system – is it deadlocked?

1. Process A holds R, wants S
2. Process B holds nothing, wants T
3. Process C holds nothing, wants S
4. Process D holds U, wants S and T
5. Process E holds T, wants V
6. Process F holds W, wants S
7. Process G holds V, wants U

Deadlock Detection with One Resource of Each Type (2)



(a) A resource graph. (b) A cycle extracted from (a).

Algorithm to Detect Deadlocks (1)

1. For each node *N* in the graph, perform following five steps with *N* as starting node.
2. Initialize *L* to empty list, and designate all arcs as unmarked.
3. Add current node to end of L, check to see if node now appears in L two times. If so, graph contains a cycle (listed in L) and algorithm terminates

Algorithm to Detect Deadlocks (2)

4. From given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick unmarked outgoing arc at random, mark it. Then follow to new current node and go to step 3.
6. If this is initial node, graph does not contain cycles, algorithm terminates. Otherwise, dead end. Remove it and go back to the previous node.

Deadlock Detection with Multiple Resources of Each Type (1)

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

The four data structures needed
by the deadlock detection algorithm.

Deadlock Detection with Multiple Resources of Each Type (2)

Deadlock detection algorithm:

1. Look for 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 back to step 1.
3. If no such process exists, algorithm terminates.

Deadlock Detection with Multiple Resources of Each Type (3)

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

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

Current allocation matrix

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

Request matrix

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

An example for the deadlock detection algorithm.

Deadlock Detection Algorithm

- Data Structure is as:

Available [m]

Allocation [n,m] as in Banker's Algorithm

Request [n,m] indicates the current requests of each process.

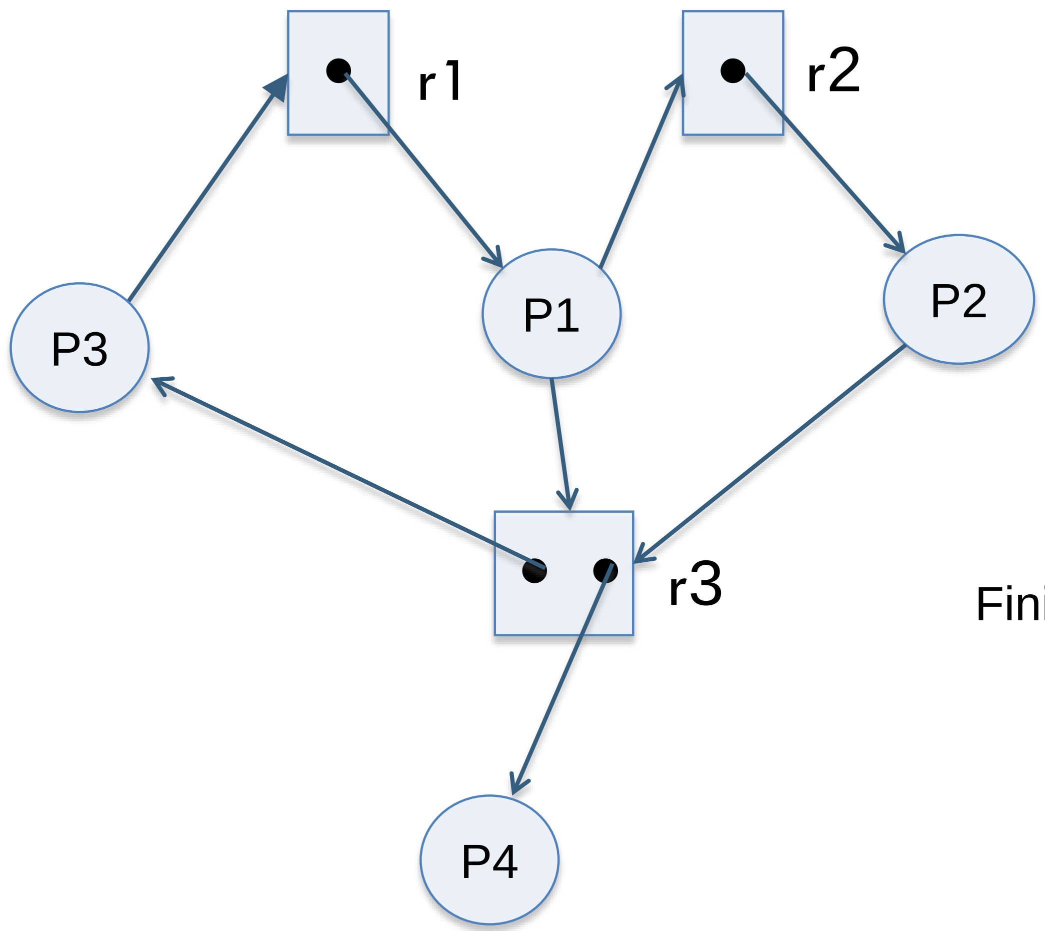
Let Work and Finish be vectors of length m and n, as in the safety algorithm.

Deadlock Detection Algorithm

The algorithm is as follows:

```
1. Initialize Work = Available
   for i = 1 to n do
         If Allocation(i) = 0 then Finish[i] = true else Finish[i] = false
2. Search an i such that Finish[i] = false and Request(i) ≤ Work
   If no such i can be found, go to step 4.
3. For i found in step 2 do:
      Work = Work + Allocation(i)
      Finish[i] = true
   Go to step 2.
4. If Finish[i] ≠ true for a some  i then the system is in deadlock state else
    there is no deadlock
```

Deadlock Detection Algorithm



Available

0	0	0

Allocation

1	0	0
0	1	0
0	0	1
0	0	1

Finish

F
F
F
F

Request

0	1	1
0	0	1
1	0	0
0	0	0

Deadlock Detection Algorithm

Available

0	0	0

Allocation

1	0	0
0	1	0
0	0	1
0	0	1

Request

0	1	1
0	0	1
1	0	0
0	0	0

Finish

F
F
F
F

work

0	0	0

Request(4) ≤ Work → i = 4:

Work = Work + Allocation(4) = [0 0 0] + [0 0 1] = [0 0 1] ;
Finish[4] = True

Deadlock Detection Algorithm

Work

0	0	1

Allocation

1	0	0
0	1	0
0	0	1
0	0	1

Request

0	1	1
0	0	1
1	0	0
0	0	0

Finish

F
F
F
T

Request(2) ≤ Work → i = 2:

Work = Work + Allocation(2) = [0 0 1] + [0 1 0] = [0 1 1] ;
Finish[2] = True

Deadlock Detection Algorithm

Work

0	1	1

Allocation

1	0	0
0	1	0
0	0	1
0	0	1

Request

0	1	1
0	0	1
1	0	0
0	0	0

Finish

F
T
F
T

Request(1) $\leq$ Work$\longrightarrow$ i = 1:

Work = Work + Allocation(1) = [0 1 1] + [1 0 0] = [1 1 1] ;
Finish[1] = True

Deadlock Detection Algorithm

Work

1	1	1

Allocation

1	0	0
0	1	0
0	0	1
0	0	1

Request

0	1	1
0	0	1
1	0	0
0	0	0

Finish

T
T
F
T

Request(3) $\leq$ Work $\rightarrow$ i = 3:

Work = Work + Allocation(3) = [1 1 1] + [0 0 1] = [1 1 2] ;
Finish[3] = True

Deadlock Detection Algorithm

Work

1	1	2

Allocation

1	0	0
0	1	0
0	0	1
0	0	1

Request

0	1	1
0	0	1
1	0	0
0	0	0

Finish

T
T
T
T

Since Finish[i] = true for all i, there is no deadlock in the system .

Recovery from Deadlock

Possible Methods of recovery (though none are “attractive”):

1. Preemption
2. Rollback
3. Killing processes

Recovery From Deadlock

- If the system is in a deadlock state, some methods for recovering it from the deadlock state should be applied.

- There are various ways for recovery:
  - Allocate one resource to several processes, by violating mutual exclusion.
  - Preempt some resources from some of the deadlocked processes.
  - Abort one or more processes in order to break the deadlock.

Recovery From Deadlock

- If preemption is used:
  - Select a victim. (Which resource(s) is/are to be preempted from which process?)
  - Rollback: If we preempt a resource from a process, roll the process back to some safe state and make it continue.

- Here OS may probably encounters starvation problem.
  - How can we guarantee that resources will not always be preempted from the same process?

Recovery From Deadlock

- In selecting a victim, important parameters may be:
  - Process priorities
  - How long the process has occupied the resource?
  - How long will it occupy the resource to finish its job?
  - What type and what amount of resources did the process use?
  - How many more resources does the process need to finish its job?
  - How many processes will be rolled back? (in case where more than one victim may be selected)

Recovery From Deadlock

- For rollback, the simplest solution is a total rollback.
- A better solution is to roll the victim process back only as far as it’s necessary to break the deadlock.
- However, the OS needs to keep more information about process states to use the second solution.
- To avoid starvation, ensure that a process can be picked as a victim for only a small number of times.
- So, it is a wise idea to include the number of rollbacks as a parameter.

Communication Deadlocks

Another kind of deadlock can occur in communication systems (e.g., networks), in which two or more processes communicate by sending messages.

A common arrangement is that process *A* sends a request message to process *B*, and then blocks until *B* sends back a reply message.

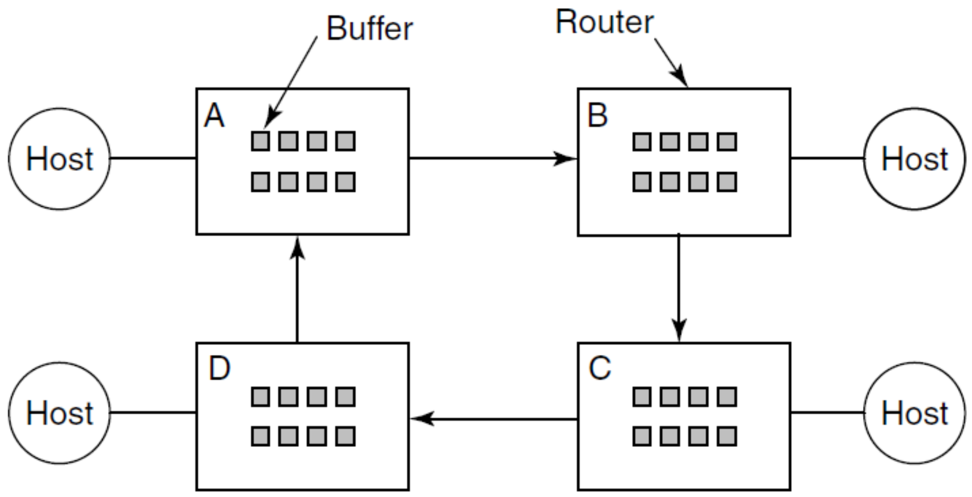
Suppose that the request message gets lost. *A* is blocked waiting for the reply. *B* is blocked waiting for a request asking it to do something.

Communication deadlocks cannot be prevented

- by ordering the resources
  (since there are no resources) or
- avoided by careful scheduling
  (since there are no moments when a request could be postponed).

Fortunately, there is another technique that can usually be employed to break communication deadlocks: timeouts.

Communication Deadlocks



A resource deadlock in a network

When a packet comes into a router from one of its hosts, it is put into a buffer for subsequent transmission to another router and then to another until it gets to the destination.

These buffers are resources and there are a finite number of them.

Suppose that all the packets at router *A* need to go to *B* and all the packets at *B* need to go to *C* and all the packets at *C* need to go to *D* and all the packets at *D* need to go to *A*.

No packet can move because there is no buffer at the other end. and we have a classical resource deadlock, albeit in the middle of a communications system.

Livelock

```
void process_A(void) {
        acquire_lock(&resource_1);
        while (try_lock(&resource_2) == FAIL) {
                release_lock(&resource_1);
                wait_fixed_time();
                acquire_lock(&resource_1);
        }
        use_both_resources( );
        release_lock(&resource_2);
        release_lock(&resource_1);
}
```

```
void process_B(void) {
        acquire_lock(&resource_2);
        while (try_lock(&resource_1) == FAIL) {
                release_lock(&resource_2);
                wait_fixed_time();
                acquire_lock(&resource_2);
        }
        use_both_resources( );
        release_lock(&resource_1);
        release_lock(&resource_2);
}
```

Figure 6-16. Busy waiting that can lead to livelock.

Livelock

Livelock and deadlock can occur in surprising ways.

In some systems, the total number of processes allowed is determined by the number of entries in the process table.

Thus, process-table slots are finite resources.

If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

End

Chapter 3