Deadlocks

Contents

- A process may also need exclusive access to multiple resources, *e.g.:*
- Two processes each want to record a scanned document on a Blu-ray disc
- Process A requests permission to use the scanner and is granted it;
- Process B requests permission to use the Blu-ray driver and is granted it;
- Process A then asks for the Blu-ray recorder:
    - But request is suspended until B releases it;
- Unfortunately, instead of releasing Blu-ray recorder:
    - Process B asks for the scanner;
- At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

- Deadlocks can also occur in a variety of other situations.. In a database system, for example,

Database system, where program may have to lock several registers *e.g.:*

- Process A locks records R1;
- Process B locks record R2;
- If each process tries to lock each other one's record, we also have a deadlock.

Conclusion:

- Deadlocks can occur on hardware resources or on software resources.
- Deadlocks happen throughout computer science!

Resources

- Examples of computer resources
  - printers
  - tape drives
  - Tables
- Resource can be:
  - Hardware device;
  - Piece of information;
- Processes need access to resources in reasonable order
- Suppose a process holds resource A and requests resource B
  - at same time another process holds B and requests A
  - both are blocked and remain so

Preemptable Resource:

- Can be taken away from the process owning it with no ill effects.

Memory is a preemptable resource, consider a system with: for example,

- A system with 1 GB of user memory, one printer, and two 1-GB processes that each want to print something.
- Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished the computation, it exceeds its time quantum and is swapped out to disk.

Example 2

- Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished the computation, it exceeds its time quantum and is swapped out to disk.

- Process *B* now runs and tries, unsuccessfully as it turns out, to acquire the printer. Potentially, we now have a deadlock situation, because *A* has the printer and *B* has the memory, and neither one can proceed without the resource held by the other.

Nonpreemptable resource:

- Cannot be taken away from its current owner without potentially causing failure

Example: If a process has begun to burn a Blu-ray

- Cannot simply give the Blu-ray drive to another process;
- This would simply result in a garbled Blu-ray;
- Blu-ray recorders are not preemptable at an arbitrary moment.

So, how can we deal with no preemptable resources?

Abstract sequence of events required to use a resource:

- Request the resource.
- Use the resource.
- Release the resource.

- 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.

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 release of a currently held resource
- None of the processes can …
  - run
  - release resources
  - be awakened

Four Conditions for Deadlock

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.

2. Hold-and-wait condition. Processes currently holding resources that were granted earlier can request new resources.

3. No-preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

4. Circular wait condition. 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.

Deadlock Modeling

Modeled with directed graphs

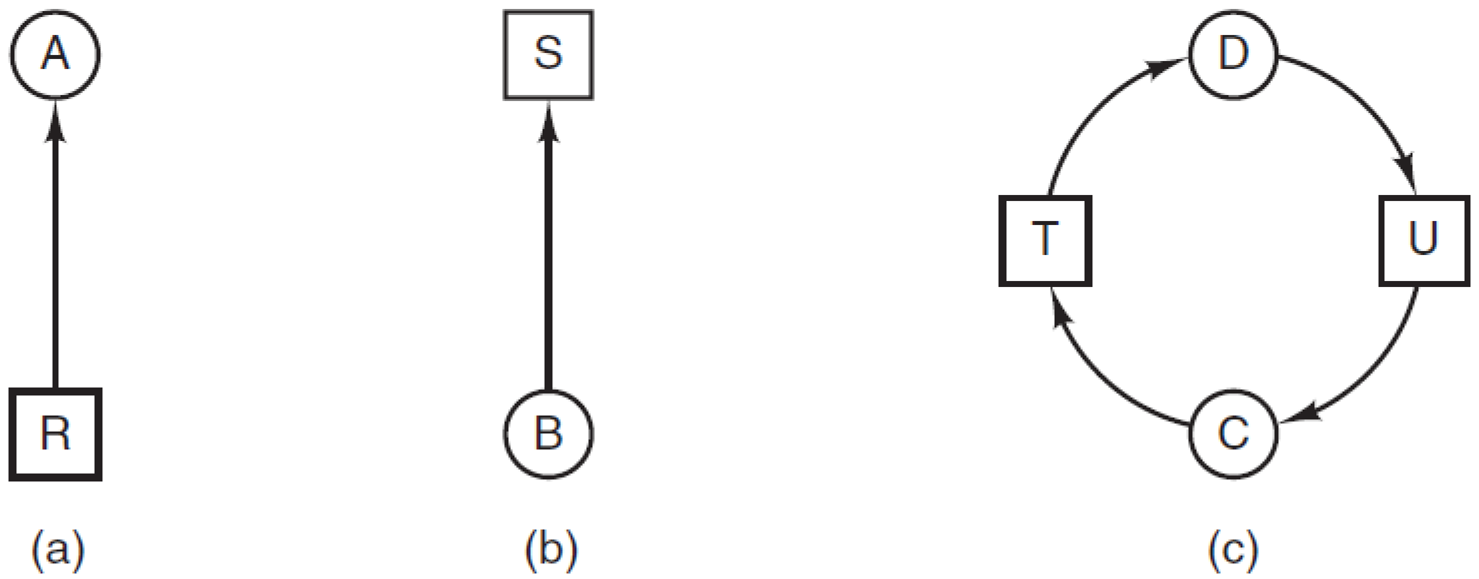


Figure 3-1. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

– resource R assigned to process A

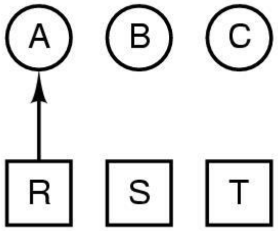
– process B is requesting/waiting for resource S

– process C and D are in deadlock over resources T and U

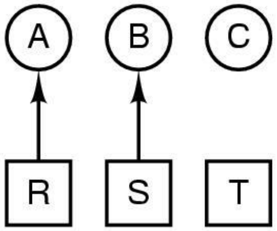
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

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

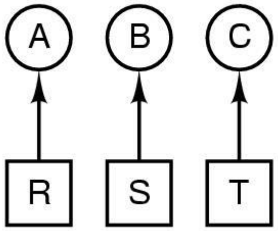
(d)



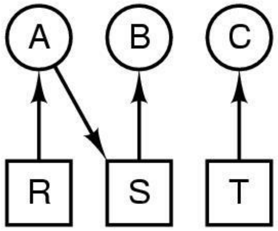
(e)



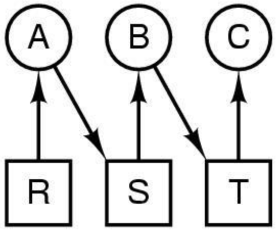
(f)



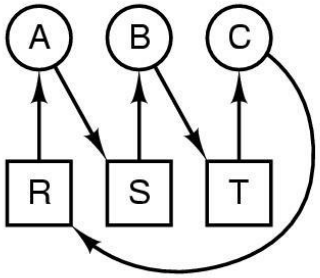
(g)



(h)



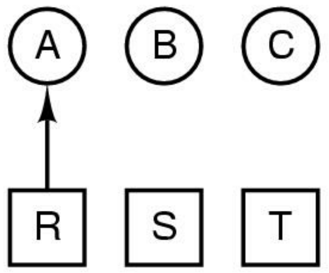
(i)



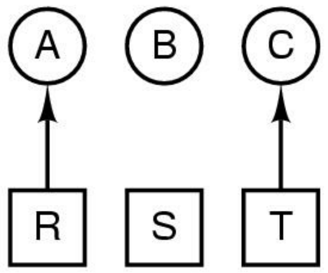
(j)

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

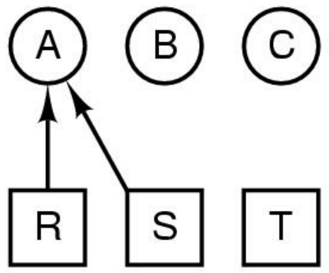
(k)



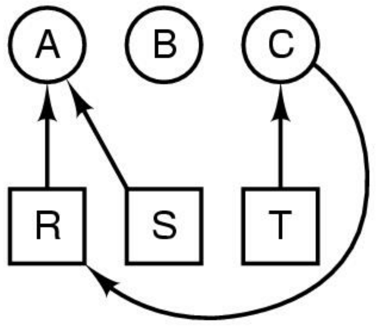
(l)



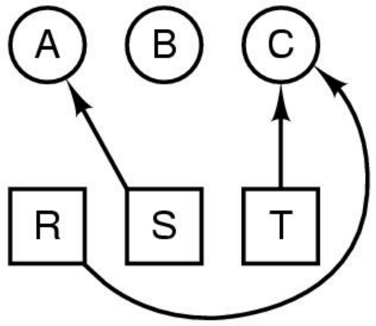
(m)



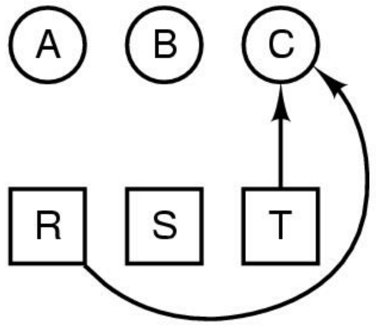
(n)



(o)



(p)



(q)

Figure 3-2. An example of how deadlock occurs and how it can be avoided.

Methods for Handling Deadlocks

Deadlock Prevention

- Ensure that *at least one* of four necessary conditions cannot hold.

Deadlock Avoidance

- Do not allow a resource request → Potential to lead to a deadlock
- Requires advance info of all requests

Deadlock Detection

- Always allow resource requests
- Periodically check for deadlocks
- If a deadlock exists → Recover from it

Ignore

- Makes sense if the likelihood is very low, say once per year
- Cheaper than *prevention*, *avoidance* or *detection*
- Used by most common OS

Deadlock Ignorance (The ostrich algorithm)

- **Ostriches bury their heads in the sand to avoid danger**—similarly, systems using this approach “ignore the problem” and carry on—pretend there is no problem.
- If this happens once in a million operations, the developer might decide: *“Let’s not bother writing complex code to handle this.”*

People react to this strategy in different ways:

Mathematicians (or Theorists):

- Say: “**This is wrong!** Every deadlock should be avoided or fixed!”
- They prefer algorithms to **guarantee safety and correctness**.

Engineers (Practical people):

- Say: “How often does it happen? Is it worth fixing?”
- They prefer to keep the system fast, simple, and cost-effective.
- If a deadlock happens, just restart the process or manually clear it.

When is the Ostrich Algorithm a good idea?

- Deadlocks happen very rarely
- Restarting the system is not a big problem
- You're building a non-critical system (like a media player, personal app, etc.)

Deadlock Detection and Recovery

1. **Deadlock Detection with One Resource of Each Type**

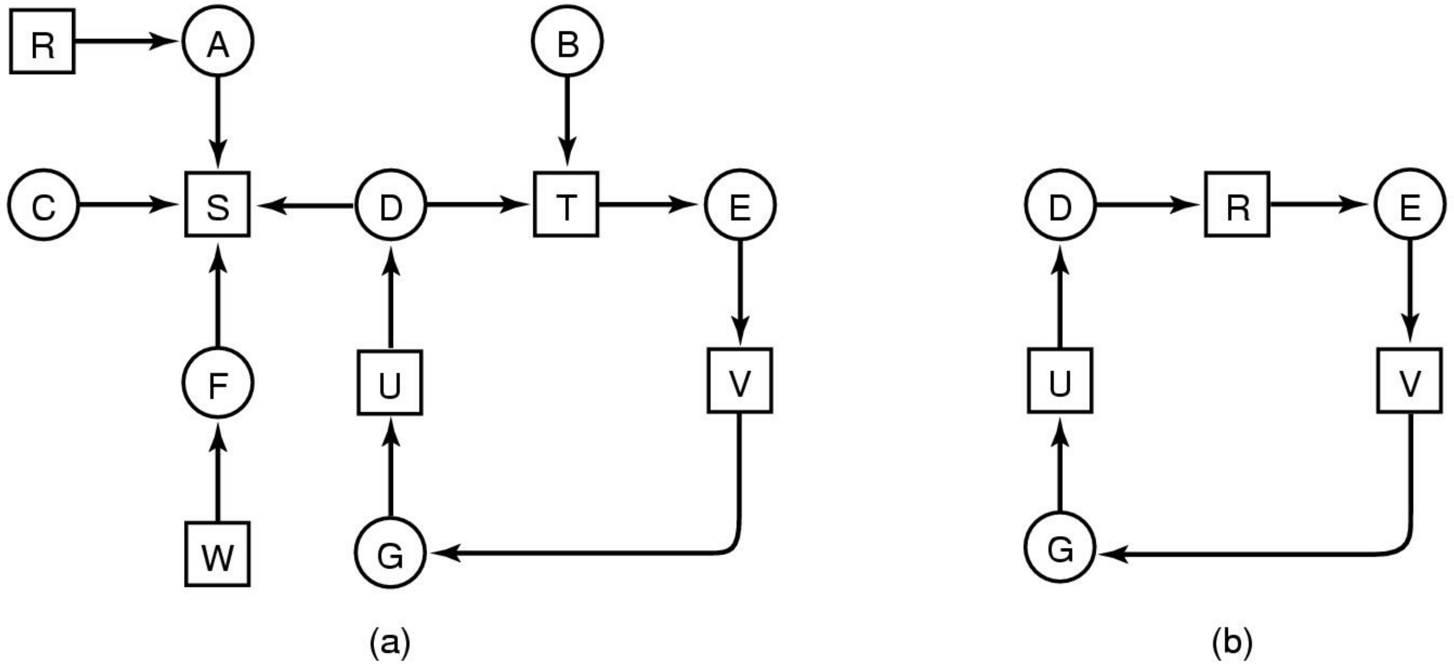


Figure 3-3. (a) A resource graph. (b) A cycle extracted from (a).

- The state of which resources are currently owned and which ones are currently being requested is as follows:
    - Process *A* holds *R* and wants *S*.
    - Process *B* holds nothing but wants *T*.
    - Process *C* holds nothing but wants *S*.
    - Process *D* holds *U* and wants *S* and *T*.
    - Process *E* holds *T* and wants *V*.
    - Process *F* holds *W* and wants *S*.
    - Process *G* holds *V* and wants *U*.
- The cycle is shown in Fig. 3(b). From this cycle, we can see that processes *D*, *E*, and *G* are all deadlocked. Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it.
- A cycle can be found within the graph, denoting deadlock

2. Deadlock Detection with Multiple Resources of Each Type

- When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock among *n* processes, *P*1 through *Pn*.
- Sum of current resource allocation + resources available = resources that exist

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

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

Current allocation matrix

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

Row n is current allocation to process n

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

Request matrix

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

Row 2 is what process 2 needs

Figure 3-4. The four data structures needed by the deadlock detection algorithm.

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

Current allocation matrix

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

Request matrix

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

Figure 3-5. An example for the deadlock detection algorithm.

- The first one cannot be satisfied because there is no Bluray drive available. The second cannot be satisfied either, because there is no scanner free.
- Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving A = (2 2 2 0)

Recovery from Deadlock

1. **Process Termination:**

- Terminate one or more processes involved in the deadlock to break the circular wait.

a. Abort all deadlocked processes

  - Simple and quick.
  - All processes involved in the deadlock are terminated.
  - Disadvantage: All progress is lost. Might corrupt data if not handled carefully.

b. Abort one process at a time until the deadlock cycle is broken

  - Allows more control and less loss of work.
  - Disadvantage: Requires repeated deadlock detection after each termination.

2. Resource Preemption:

- Take resources away from one or more processes and give them to others to resolve deadlock.
- Requires rollback of the process to a safe state.
- Needs to maintain information to restart the process later (checkpointing).

Challenges

- Selecting a victim process to preempt.
- Minimizing cost of preemption (e.g., time, data loss).
- Ensuring the process can be rolled back safely.

3. Rollback:

- Roll back one or more processes to some previously saved safe state.
- Used when processes can be **checkpointed** (saved).
- Checkpointing a process means that its state is written to a file so that it can be restarted later.
- Restart the process if it is found deadlocked

- **Checkpointing (Rollback Mechanism):** To resume a preempted process later, the OS must save its **current state**.
- This saved state is called a **checkpoint**.
- If a process is killed or rolled back, it resumes from this checkpoint.

- Database systems (with transaction rollback)

Deadlock Prevention

- Resource allocation rules prevent deadlock by prevent one of the four conditions required for deadlock from occurring
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular Wait

1. Mutual Exclusion

- Allow everybody to use the resources immediately they require.
- **Not always possible**, since some resources (like printers) must be used exclusively.
- By spooling printer output, several processes can generate output at the same time.
- **Spooling** stands for *Simultaneous Peripheral Operations On-Line*.
- Instead of each process directly using the printer, they write their output to a disk file (spool queue).
- A background process called the **printer daemon** reads the spool and sends jobs to the printer— which **never causes deadlock**.

2. Hold and Wait Condition

Approach 1: Request All Resources at Once

- Require each process to request all of its required resources simultaneously before it begins execution.
- If all resources are available: Allocate them, and the process proceeds.
- Advance Knowledge required: A process **may not know** in advance what resources it will need during its execution.

Approach 2: Release and Re-request (Variation)

- If a process is holding resources and needs additional ones that are not available, it must release all currently held resources, then request all needed resources again in a single request.
- Prevents a process from holding some resources while waiting for others, eliminating hold and wait.

- Processes may **lose progress** if they are forced to release resources and start over.
- May result in starvation.

3. No Preemption

- Allow resources to be preempted.
- If a process that currently holds some resources requests an additional resource that cannot be immediately allocated, then all the resources currently held by that process are forcibly released (preempted).
- Not practical for resources that can't be safely preempted (e.g., printers).

4. Circular Wait

- To prevent circular wait, the system imposes a total ordering (an ordered list) on all resource types.
- This means:
- Every resource type is assigned a unique number (or rank).
- Each process must request resources in increasing order of these numbers.

Suppose: R1 = 1 (lowest order), R2 = 2, R3 = 3 (highest order).

P1 holds R1 (1) and waits for R2 (2) → Allowed (ascending order).

P2 holds R2 (2) and waits for R3 (3) → Allowed (ascending order).

P3 holds R3 (3) and waits for R1 (1) → **Not allowed** (descending order).

Deadlock avoidance

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

– YES, but only if enough information is available in advance.

  - Maximum number of each resource required

Resource Trajectories

- The main algorithms for deadlock avoidance are based on the concept of safe states.
- Graphical approach does not translate directly into a usable algorithm, it gives a good intuitive feel for the nature of the problem.

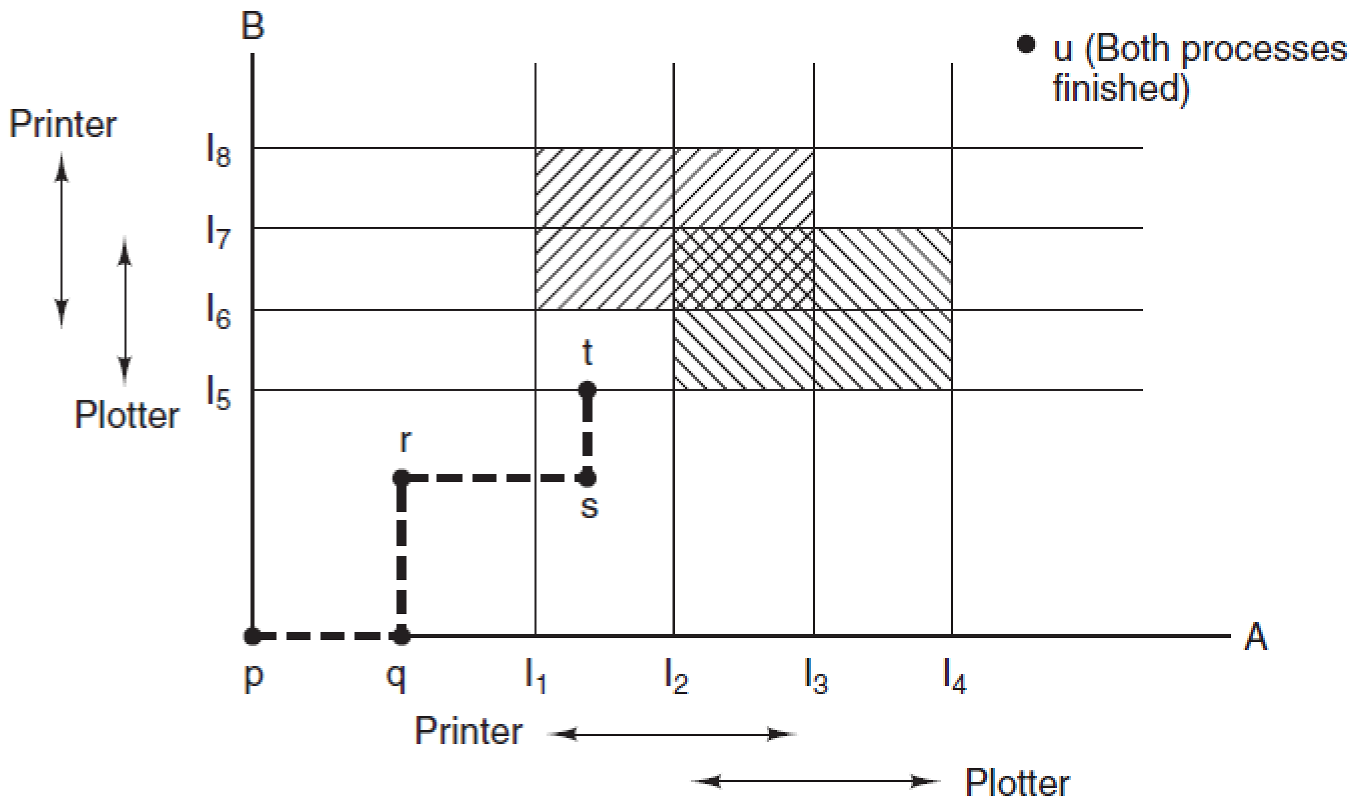


Figure 3-6. Two process resource trajectories.

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.
- Total Resources = 10
  - Process A: Allocated 3 resources, needs 9 more
  - Process B: Allocated 2 resources, needs 4 more
  - Process C: Allocated 2 resources, needs 7 more

Total Allocated = 7, so Available (Free) Resources = 10 − 7 = 3, fig(a).

	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)

Figure 3-7 . Demonstration that the state in (a) is safe.

Safe state

Available (3), Suppose Process B finishes first,

Process B needs 4 → can finish

    Frees 1 → New available = 1 + 4 = 5

Process C needs 7 →can finish

    Frees 0 → New available = 7

Process A needs 9→ can finish

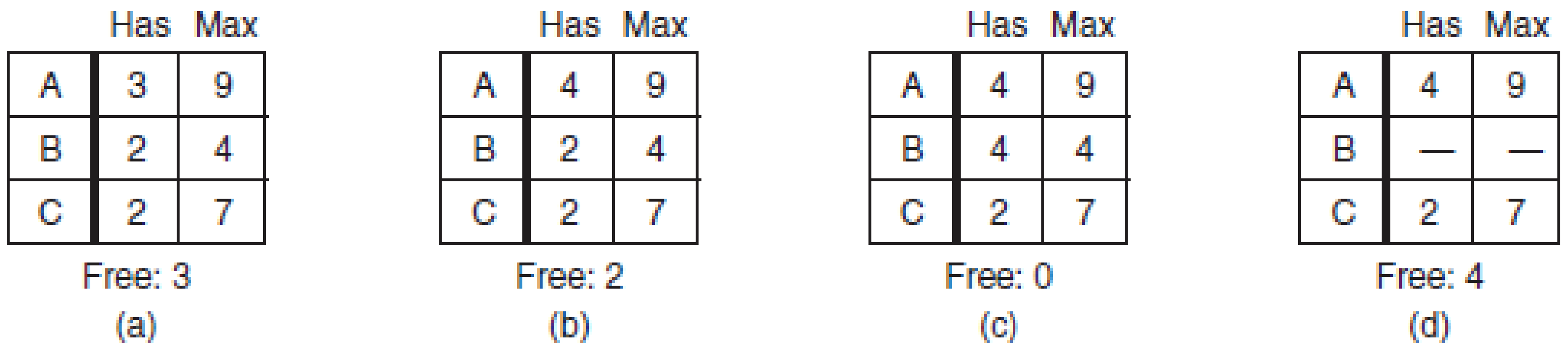
    Frees 1 → Final available = 7 + 3 = 10

Safe sequence = $B \rightarrow C \rightarrow A$

Unsafe State

Available (3)

- None of the processes (A, B, or C) can complete with 3 available resources.
- Unsafe state (also a deadlock here), Unsafe (System is at risk of or already in deadlock)



	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)

Figure 3-8. Demonstration that the state in (b) is not safe.

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)

Figure 3-9. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

- The banker's algorithm was first published by Dijkstra in 1965.
- **Banker's Algorithm** is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.
- In Fig. 6-11(a) we see four customers, *A*, *B*, *C*, and *D*, each of whom has been granted a certain number of credit units.
- The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units rather than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

The Banker’s Algorithm for Multiple Resources

Process	Tape drives	Plotters	Printers	Blu-rays
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	Blu-rays
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 assigned

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

Figure 3-10. The banker’s algorithm with multiple resources.

- The three vectors at the right of the figure show the existing resources, *E*, the possessed resources, *P*, and the available resources, *A*, respectively. From *E* we see that the system has six tape drives, three plotters, four printers, and two Blu-ray drives.
- Five tape drives, three plotters, two printers, and two Blu-ray drives are currently assigned.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$

Finish*[*i*] = *true

go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

Resource-Request Algorithm for Process P_i

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$. If $\boldsymbol{Request_i\,[j] = k}$ then process $\boldsymbol{P_i}$ wants $\boldsymbol{k}$ instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \le Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\boldsymbol{Request_i \le Available}$, go to step 3. Otherwise $\boldsymbol{P_i}$ must wait, since resources are not available
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

$$\boldsymbol{Available = Available - Request_i;}$$

$$\boldsymbol{Allocation_i = Allocation_i + Request_i;}$$

$$\boldsymbol{Need_i = Need_i - Request_i;}$$

- If safe $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$
- If unsafe $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored

Example :1

- 5 processes P_0 through P_4;

  3 resource types: *A* (10 instances), *B* (5instances), and *C* (7 instances)

- Snapshot at time T_0:

	Allocation	**Max**	**Available**
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\mathbf{P_1}$, $\mathbf{P_3}$, $\mathbf{P_4}$, $\mathbf{P_0}$, $\mathbf{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\mathbf{P_4}$ be granted?
- Can request for (0,2,0) by $\mathbf{P_0}$ be granted?

Example :2 Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1.What is the reference of the need matrix?

2.Determine if the system is safe or not.

3.What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Ans. 1: Context of the need matrix is as follows:

Need [i] = Max [i] – Allocation/Has [i]

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3

Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2

Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0

Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1

Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Ans. 2: Apply the Banker's Algorithm: Available Resources of A, B and C are 3, 3, and 2. Now we check if each type of resource request is available for each process.

Step 1: For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2: For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) => 5, 3, 2

Similarly, we examine another process P3.

Step 3: For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4: For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5: For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

New available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Step 6: For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7: For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence : P2, P4, P5, P1 and P3.

Ans. 3: For granting the Request (1, 0, 2), first we have to check that **Request <= Available**, that is (1, 0, 2) <= (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

Example :3

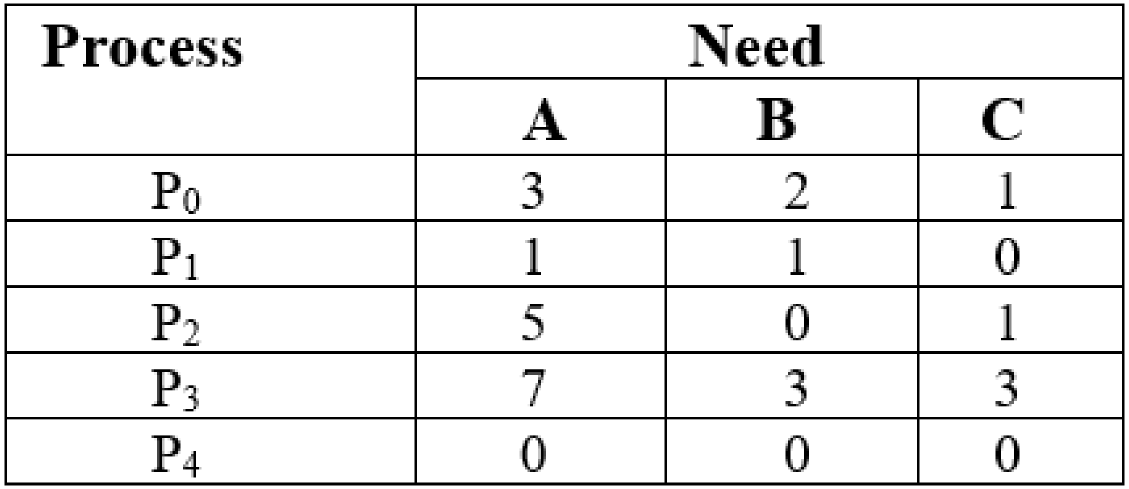
Consider the following snapshot of a system:

Processes	**Allocation** **A B C**	**Max** **A B C**	**Available** **A B C**
P_0	**1 1 2**	**4 3 3**	**2 1 0**
P_1	**2 1 2**	**3 2 2**	
P_2	**4 0 1**	**9 0 2**	
P_3	**0 2 0**	**7 5 3**	
P_4	**1 1 2**	**1 1 2**	

1. Calculate the content of the need matrix?
2. Is the system in a safe state?
3. Determine the total amount of resources of each type?

Ans. 1. Content of the need matrix can be calculated by using the below formula

Need = Max – Allocation



Process	**Need**		
	A	**B**	**C**
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

2. Now, we check for a safe state

Safe sequence:

1. For process P_0, Need = (3, 2, 1) and

Available = (2, 1, 0)

Need ≤ Available = False

So, the system will move for the next process.

2. For Process P_1, Need = (1, 1, 0)

Available = (2, 1, 0)

Need ≤ Available = True

Request of P_1 is granted.

∴ Available = Available +Allocation

= (2, 1, 0) + (2, 1, 2)

= (4, 2, 2) (New Available)

3. For Process P_2, Need = (5, 0, 1)

Available = (4, 2, 2)

Need $\leq$ Available = False

So, the system will move to the next process.

4. For Process $P_{3,}$ Need = (7, 3, 3)

Available = (4, 2, 2)

Need $\leq$ Available = False

So, the system will move to the next process.

5. For Process P_4, Need = (0, 0, 0)

Available = (4, 2, 2)

Need $\leq$ Available = True

Request of P_4 is granted.

$\therefore$ Available = Available + Allocation

= (4, 2, 2) + (1, 1, 2)

= (5, 3, 4) now, (New Available)

6. Now again check for Process $P_{2,}$ Need = (5, 0, 1)

Available = (5, 3, 4)

Need $\leq$ Available = True

Request of P_2 is granted.

$\therefore$ Available = Available + Allocation

= (5, 3, 4) + (4, 0, 1)

= (9, 3, 5) now, (New Available)

7. Now again check for Process P_3, Need = (7, 3, 3)

Available = (9, 3, 5)

Need $\leq$ Available = True

Request of P_3 is granted.

$\therefore$ Available = Available +Allocation

= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)

8. Now again check for Process $P_{0,}$ = Need (3, 2, 1)

= Available (9, 5, 5)

Need $\leq$ Available = True

So, the request will be granted to P_0. Safe sequence: < $P_{1,}$ $P_{4,}$ $P_{2,}$ $P_{3,}$ $P_{0>}$

The system allocates all the needed resources to each process. So, we can say that system is in a safe state.

3. The total amount of resources = sum of columns of allocation + Available

= [8 5 7] + [2 1 0] = [10 6 7]