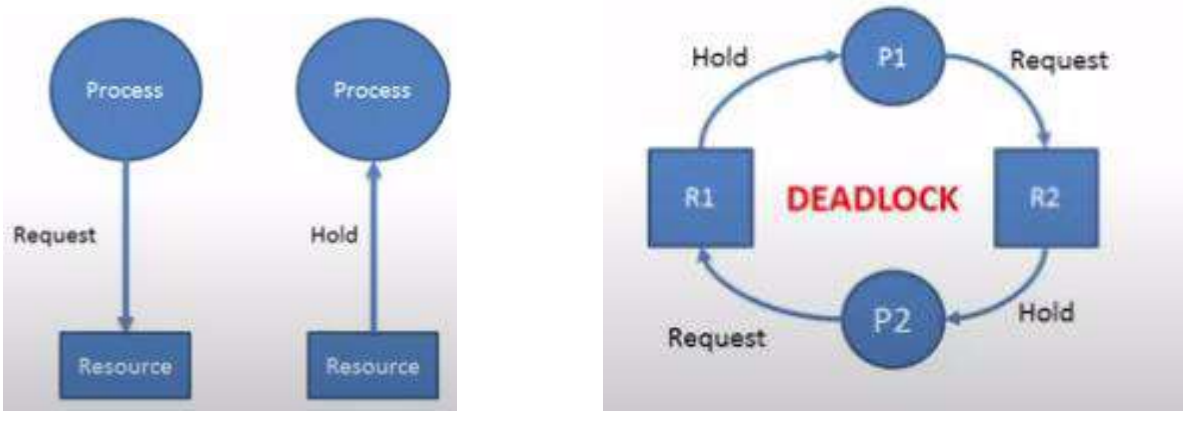
Process Deadlocks

Introduction

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.

Because all the processes are waiting, none of them will ever cause any event that could wake up any of the other members of the set, and all the processes continue to wait forever.

Deadlocks are a set of blocked processes each holding a resource and waiting to acquire resource held by other process.



(a) (b)

Figute: (a) process is requesting resource and a recource is in holding process (b) a deadlock scenario

Example:

- Two processes **A** and **B** each want to record a scanned document on a CD.
- **A** requests permission to use Scanner and is granted.
- **B** is programmed differently and requests the CD recorder first and is also granted.
- 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 Scanner. At this point both processes are blocked and will remain so forever. This situation is called **deadlock**.

Resources

- In short, a resource is anything that must be acquired, used, and released over the course of time.
- The resources include devices, data records, files, and so forth.
- A resource can be a hardware device (e.g., a Blu-ray drive) or a piece of information (e.g., a record in a database).
- A computer will normally have many different resources that a process can acquire.
- A major class of deadlocks involves resources to which some process has been granted exclusive access
- The abstract sequence of events required to use a resource is given below.
    1. Request the resource.
    2. Use the resource.
    3. Release the resource.

Preemptable and Non-preemptable Resources

- Resources come in two types: **preemptable** and **nonpreemptable**.
- A **preemptable resource** is that resource which can be taken away from the process owning it with no ill effects.

  Example: Memory

    - Consider, 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.
    - Process B now runs and tries, unsuccessfully as it turns out, to acquire the printer. Potenially, we now hav e 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.
    - Fortunately, it is possible to preempt (take away) the memory from B by swapping it out and swapping A in. Now A can run, do its printing, and then release the printer. No deadlock occurs.
- A **nonpreemptable resource**, is that resource which cannot be taken away from its current owner process without potentially causing failure.

Example: Blue-Ray recorder, Printer

- If a process has begun to burn a Blu-ray, suddenly taking the Blu-ray recorder away from it and giving it to another process will result in a garbled Blu-ray. Blu-ray recorders are **not preemptable** at an arbitrary moment.

- Whether a resource is preemptible depends on the context
- In general, deadlocks involve nonpreemptable resources.
- Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another.

Deadlock Characterization

- Deadlock characterization describes the distinctive features that are the cause of deadlock occurrence
- The following four conditions must hold for there to be a (resource) 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.

All four of these conditions must be present for a deadlock to occur

Resource – Allocation Graph

— The four conditions for deadlock can be modeled using directed graphs.

— The graphs have two kinds of nodes**: processes, shown as circles**, and **resources, shown as squares**.

— A *directed arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process.* In following figure (a), resource R is currently assigned to process A.

— A *directed arc from a process to a resource means that the process is currently blocked waiting for that resource.* In Figure (b), process B is waiting for resource S.

— In Figure (c) we see a deadlock: process C is waiting for resource T, which is currently held by process D. Process D is not about to release resource T because it is waiting for resource U, held by C. Both processes will wait forever.

— *A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind).* In this example, the cycle is C − T − D −U − C.

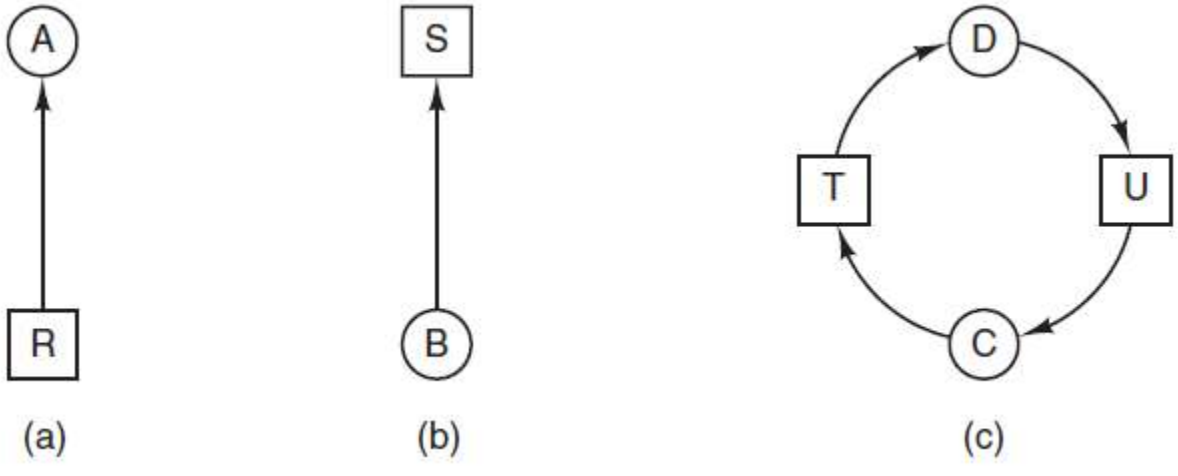


Figure: Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource (c) Deadlock.

Basic Facts in resource allocation graph

- 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

Methods for Handling Deadlocks

In general, following four strategies are used for dealing with deadlocks

1. Just ignore the problem.
   - Maybe if you ignore it, it will ignore you.
2. Detection and recovery.
   - Let them occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions

Ostrich Algorithm

When storm approaches, an ostrich puts it's head in the sand (ground) and pretends (imagines) that there is no problem at all.

- "Stick your head in sand and pretend there is no problem at all"
- Deadlock ignorance strategy
- Ignore the deadlock and pretend that deadlock never occur
- Reasonable if
  - Deadlock occur very rarely
  - Deadlock is difficult to detect
  - Cost of deadlock prevention is high

Deadlock prevention

- If we can ensure that at least one of four conditions of deadlock is never satisfied, then deadlocks will be structurally impossible
- To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be invalidated.
- Deadlock can be prevented by attacking the one of the four conditions that leads to deadlock.
  - **Attacking the Mutual Exclusion Condition**
    - No deadlock if each resource can be assigned to more than one process.

- We cannot assign some resources to more than one process at a time such as CD Recorder, Printer etc...
- So this solution is not feasible.

☞ **Attacking the Hold and Wait Condition**

- If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks
  - ☞ One way to achieve this goal is to require processes to request all their resources before starting execution.
  - ☞ A process is allowed to run if all resources it needed is available. Otherwise nothing will be allocated and it will just wait.
- Problem with this strategy is that a process may not know required resources at start of run.
- Resource will not be used optimally.

☞ **Attacking the No Preemption Condition**

- When a process P0 request some resource R which is held by another process P1 then resource R is forcibly taken away from the process P1 and allocated to P0
- This strategy is tricky at best and impossible at worst.
- Consider a process holds the printer, halfway through its job; taking the printer away from this process without having any ill effect is not possible.
  - ☞ This is not a possible option.

☞ **Attacking the Circular Wait Condition**

- The circular wait can be eliminated in several ways
- One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one.
  - ☞ For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.
- Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in below

  1) Printer 2) Scanner

  3) Plotter 4) Tape Drive

  5) CD Rom

- ☞ Now the rule is that: processes can request resources whenever they want to, but all requests must be made in numerical order.
- ☞ A process need not acquire them all at once.
- ☞ Resource graph can never have cycle.

- Assume A has i and B has j are where i and j are distinct resources and they will have different numbers.
  - — If i > j, then A is not allowed to request j because that is lower than what it already has.
  - — If i < j, then B is not allowed to request i because that is lower than what it already has

  A process may request 1st a printer, then tape drive. But it may not request 1st a plotter, then the printer.

Summary of approaches to deadlock prevention

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

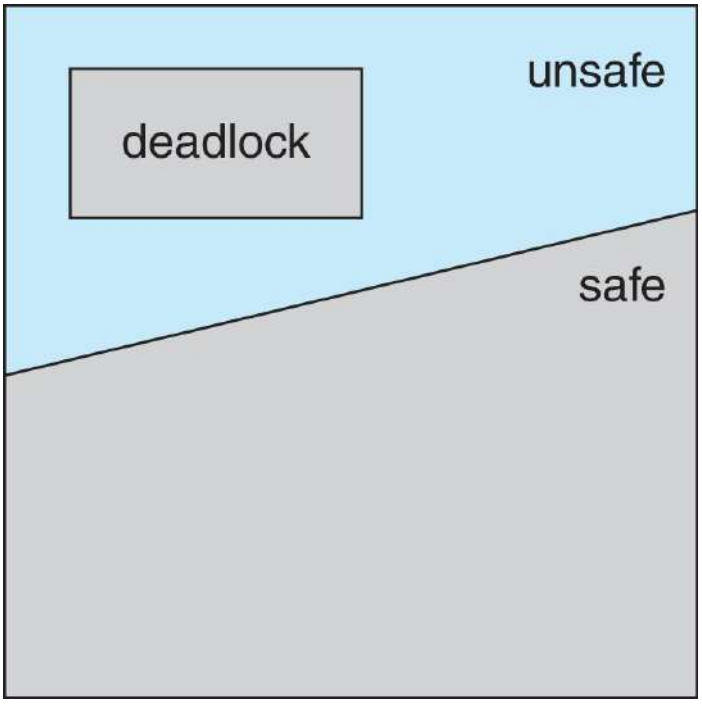
Deadlock Avoidance

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 will dynamically examine the resource allocation operations to ensure that there won't be a circular wait on resources. One of the deadlock avoidance algorithms is **Banker's algorithm**.

- Deadlock can be avoided by allocating resources carefully.
- Carefully analyze each resource request to see if it can be safely granted
- An algorithm is needed which can always avoid deadlock by making right choice all the time (one of such algorithms is Banker's Algorithm)

Safe and unsafe state

- Resource allocation state is defined by the number of available and allocated resources and the maximum demand of the processes. When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**
- 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.*
- From a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.
- If the system is in a safe state, there can be no deadlock. If the system is in an unsafe state, there is the possibility of deadlock.



Example: The state in following figure (a) is safe

☞ In following figure (a) we have a state in which
  - A total of 10 instances of the resource exist.
  - A has three instances of the resource but may need as many as nine eventually.
  - B currently has two and may need four altogether, later.
  - Similarly, C also has two but may need an additional five.
  - Seven resources already allocated, three tare still free.

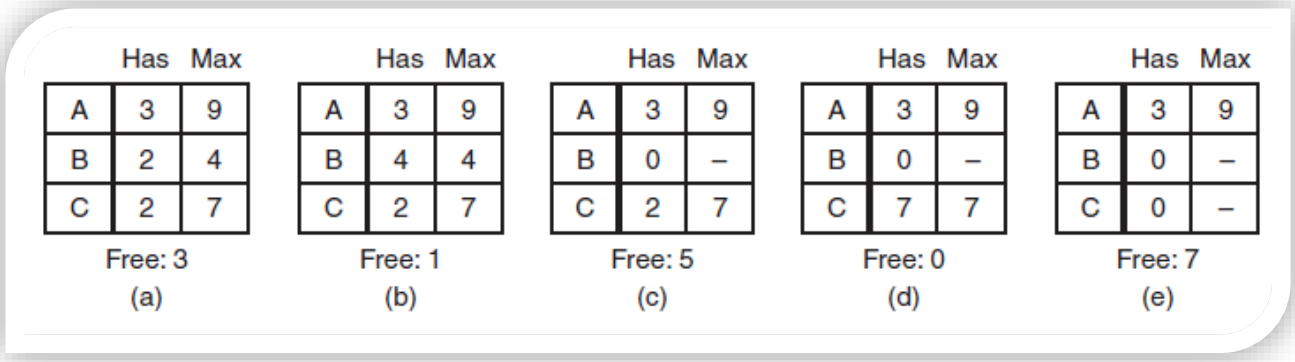


Figure: Demonstration that the state in (a) is safe

- ☑ The state of figure (a) is safe because there exists a sequence of allocations that allows all processes to complete.
  - ☞ The scheduler can simply run B exclusively, until it asks for and gets two more instances of the resource, leading to the state of Fig. (b).
  - ☞ When B completes, we get the state of Fig.(c).
  - ☞ Then the scheduler can run C, leading eventually to Fig. (d).
  - ☞ When C completes, we get Fig (e).
  - ☞ Now A can get the six instances of the resource it needs and also complete.
  - ☞ Thus, the state of fig. (a) is safe because the system, by careful scheduling, can avoid deadlock.

But, state in following figure (b) is unsafe.

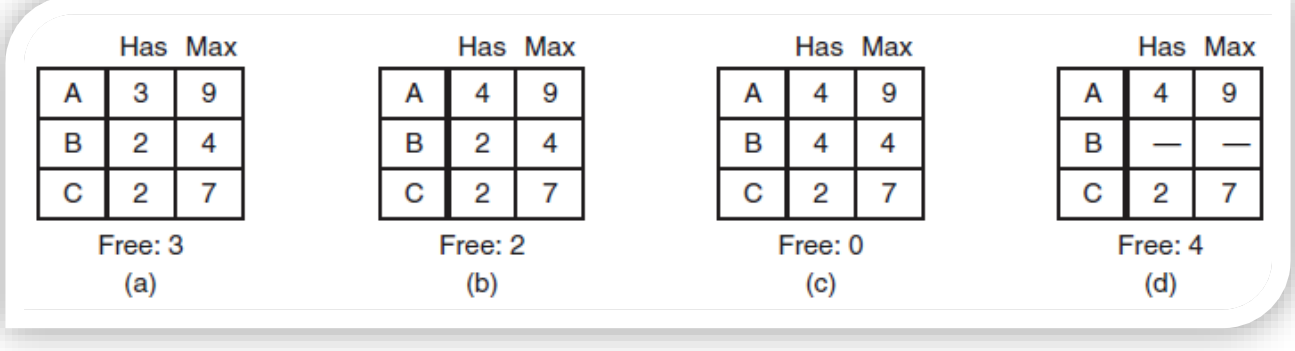


Figure : Demonstration that the state in (b) is not safe.

- ☞ Now suppose we have the initial state shown in Fig.(a), but this time A requests and gets another resource, giving Fig.(b).
- ☞ Can we find a sequence that is guaranteed to work?
- ☞ Let us try.
  - ➢ The scheduler could run B until it asked for all its resources, as shown in Fig.(c).
  - ➢ Eventually, B completes and we get the state of Fig (d).
  - ➢ At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five.
- ☞ There is no sequence that guarantees completion.
- ☞ Thus, the allocation decision that moved the system from above Fig. (a) to (b) went from a safe to an unsafe state.

The Banker's algorithm

- The Banker's Algorithm is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. (Years ago, banks did not lend money unless they knew they could be repaid.)
- The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Dijkstra.
- Resource allocation state is defined by the number of available and allocated resources and the maximum demand of the processes. When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

  The Banker algorithm does the simple task –

  - ☞ If ***granting the request leads to an unsafe state the request is denied.***
  - ☞ If ***granting the request leads to safe state the request is carried out.***

  Basic Facts:

  - ☑ If a system is in safe state ⇒ no deadlocks.
  - ☑ If a system is in unsafe state ⇒ possibility of deadlock.
  - ☑ Avoidance ⇒ ensure that a system will never enter an unsafe state.

The Banker's Algorithm for a Single Resource

- Resource of only type is considered (eg : tape drive)
- Resource allocation state is defined by the number of available and allocated resources and the maximum demand of the processes. When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

  The Banker algorithm does the simple task –

  - ☞ If ***granting the request leads to an unsafe state the request is denied.***
  - ☞ If ***granting the request leads to safe state the request is carried out.***

Example: (Safe State)

- Available units = 10
- In following figure (a) we see four customers (processes), A, B, C, and D, each of whom has been granted a certain number of credit units (e.g., 1 unit is 1K dollars).
- The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units (resources say tape drives) rather than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

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

Free: 10

(a)

- The customers go about their respective businesses, making loan requests from time to time (i.e., asking for resources). At a certain moment, the situation is as shown in following figure (b).

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

Free: 2

(b)

- This state is safe because with two units left, the banker can delay any requests except C's, thus letting C finish and release all four of his resources. With four units in hand, the banker can let either D or B have the necessary units, and so on.

Example: (Unsafe State)

- Consider what would happen if a request from B for one more unit were granted in above Fig(b). We would have situation as in following Fig(c), which is unsafe.

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

Free: 1

(c)

- If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a **deadlock**.

  NOTE: *An unsafe state does not imply the existence or even eventual existence of a deadlock. What an unsafe state does imply is that some unfortunate sequence of events might lead a deadlock.*

Problem 01: State whether the given processes are in deadlock or not. Given that resource instance is 10.

Process	Allocated	Maximum
A	**3**	**9**
B	**2**	**4**
C	**2**	**7**

Solution:

Calculating needed resources, using

Needed = Maximum - Allocated

We get,

Process	Allocated	Maximum	Needed
A	**3**	**9**	**6**
B	**2**	**4**	**2**
C	**2**	**7**	**5**

Here currently,

Total allocation = 3+2+2 = 7

So, free = total available – current allocation = 10 – 7 = 3

- With current free resources process B can be executed, since need of B ≤ Free i.e 2 ≤ 3 . So , B executes. After execution of B it release the resources allocated by it. Total free resource becomes,

  free = current free + Allocation by B

  = (1+2+2) = 5

- Now, with current free resources process C can be executed, since need of C ≤ Free i.e 5 ≤ 5 . So, C executes. After execution of C it release the resources allocated by it. Total free resource becomes,

  free = current free + Allocation by C

  = (0+5+2) = 7

- With current free resources process A can be executed, since need of A ≤ Free i.e 6 ≤ 7. So , A executes. After execution of A it releases the resources allocated by it. Total free resource becomes

  free = current free + Allocation by A

  = (1+6+3) = 10

Here all the process runs successfully hence they are is safe state and occurs no deadlock. Safe sequence is: **B→C→A**

The Banker's Algorithm for a Multiple Resource

The banker's algorithm can be generalized to handle multiple resources (the resources of multiple types say printers, tape drives, plotters, Blue –ray)

Algorithm

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A (Available resources). 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).

NOTE: If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the pool of available resources either gets larger, or at worst, stays the same.

Example:

- In following figure left matrix shows how many of each resource are currently assigned to each of the five processes (A, B, C, D and E).
- The matrix on the right shows how many resources each process still needs in order to complete.

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)

- As in the single-resource case, processes must state their total resource needs before executing, so that the system can compute the right-hand matrix at each instant.
- The three vectors (E, P and A) at the right of the figure show the existing, possessed and available resources.
  - E represents the number of existing resources
  - P represents the number of possessed resources
  - A, represents number of the available resources
  - ☑ E = (6 3 4 2) means the system has 6 tape drives, 3 plotters, 4 printers and 2 blue –rays (existing)
  - ☑ P = (5 3 2 2) means 5 tape drives, 3 plotters, 2 printers and 2 blue-rays are currently used by processes (possessed).
  - ☑ A = (1 0 2 0) means 1 tape drive and 2 printers are free now (available)
- The available resource vector is just the difference between what the system has and what is currently in use

- The current state in above figure is safe.
  - Suppose that process B now makes a request for the printer. This request can be granted because the resulting state is still safe (process D can finish, and then processes A or E, followed by the rest).
  - Now imagine that after giving B one of the two remaining printers, E wants the last printer. Granting that request would reduce the vector of available resources to (1 0 0 0), which leads to deadlock, so E's request must be deferred for a while.

Limitations:

- Although in theory the algorithm is wonderful, in practice it is essentially useless because processes rarely know in advance what their maximum resource needs will be.
- In addition, the number of processes is not fixed, but dynamically varying as new users log in and out.
- Furthermore, resources that were thought to be available can suddenly vanish (tape drives can break).

Problem 01: A system has four process P1, P2, P3 and P4 and three resources R1, R2 and R3 with existing resources E = (15, 9, 5). After allocating resources to all the processes available resources becomes A = (3, 2, 0). State whether the process is safe or not using banker's algorithm. If safe, write the safe sequence.

Process	Allocation			Maximum			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	0	1	3	2	2	0	2	1
P2	5	4	1	6	8	2	1	4	1
P3	2	2	0	3	2	4	1	0	4
P4	2	1	3	4	2	3	2	1	0

NOTE: If Need is not given, it can be calculated using Need = Maximum – Allocation

Solution: We have A = (3, 2, 0)

Step 1: With current available resources A = (3, 2, 0) P4 can be executed. Since need of P4 ≤ A i.e. (2, 1, 0) ≤ (3, 2, 0) so P4 executes after complete execution of P4 it releases the resources which is allocated by it. Now total current available resources A becomes,

A = previous free + Allocation by P4

A = (3, 2, 0) + (2, 1, 3) = (5, 3, 3)

Step 2: With current available resources A = (5, 3, 3) P1 can be executed. Since need of P1 ≤ A i.e. (0, 2, 1) ≤ (5, 3, 3) so P1 executes. After complete execution of P1 it releases the resources which is allocated by it. Now total current available resources A becomes,

A = previous free + Allocation by P1

A = (5,3,3) + (3,0,1) = (8,3,4)

Step 3: With current available resources A = (8, 3, 4) P3 can be executed. Since need of P3 ≤ A i.e. (1,0,4) ≤ (8,3,4) so P3 executes After complete execution of P3 it releases the resources which is allocated by it. Now total current available resources A becomes,

A = previous free + Allocation by P3

A = (8,3,4) + (2,2,0) = (10,5,4)

Step 4: With current available resources A = (10, 5, 4) P2 can be executed. Since need of P2 ≤ A i.e. (1,4,1) ≤ (10,5,4) so P2 executes After complete execution of P2 it releases the resources which is allocated by it. Now total current available resources A becomes,

A = previous free + Allocation by P2

A = (10,5,4) + (5,4,1) = (15,9,5)

Here all the process runs hence they are in safe state and occurs **no deadlock.** Safe sequence is: **P4→P1→P3→P2**

Deadlock Detection

- System allows deadlock to occur and tries to detect when it happens, and then takes some action recover after the fact.
- When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to

Deadlock Detection with One Resource of Each Type

- There is only one resource of each type.
- Such a system might have one scanner, one Blu-ray recorder, one plotter, and one tape drive, but no more than one of each class of resource.
- For a system with one instance of each resource, we can detect deadlock by constructing a resource allocation graph
  - ☞ If the graph contains one or more cycles, a deadlock exists. Any process that is part of the cycle is deadlocked.
  - ☞ If no cycle exists, the system is not deadlocked
- Example:

  Consider a system with seven processes, A through G, and six resources, R through W. The state of which resources are currently owned and which ones are currently being requested is as follows:

  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.

  The question is: ‘‘Is this system deadlocked, and if so, which processes are involved?’’

Solution:

From given information, we can construct resource allocation graph as follows:

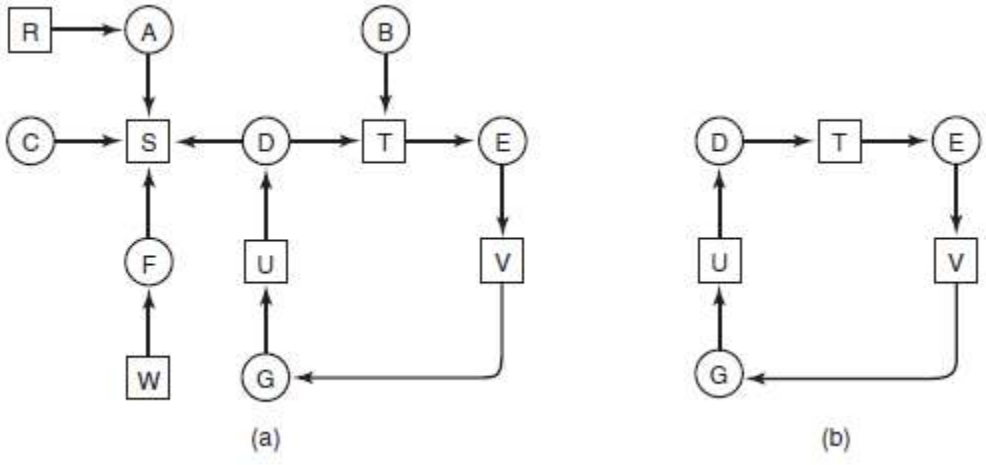


Figure: (a) A resource graph. (b) A cycle extracted from (a).

- 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 take by other two processes in turn.

Many algorithms for detecting cycles in directed graphs are known. One of them is given below:

1. For each node, N, in the graph, perform the following five steps with N as the starting node.
2. Initialize L (a dynamic data structure) to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

In above example, the algorithm is traced as follows

- We are starting from node D.
- Empty list L = ()
- Add current node so Empty list = (D).
- From this node there is one outgoing arc to T so add T to list.
- So list become L = (D, T).
- Continue this step….so we get list as below
  L = (D, T, E)…… L = (**D**, T, E, V, G, U, **D**)
- In the above step in list the node **D appears twice**, **so deadlock**.

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. A matrix-based algorithm for detecting deadlock among n processes, P_1 through P_n is presented here.
- Let the number of resource classes be m, with E1 resources of class 1, E_2 resources of class 2, and generally, E_i resources of class i ($1 \leq i \leq m$).
  - E is the **existing resource vector**. It gives the total number of instances of each resource in existence.
  - For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives
- At any instant, some of the resources are assigned and are not available. Let A be the **available resource vector**, with A_i giving the number of instances of resource i that are currently available (i.e., unassigned).
  - If both of our two tape drives are assigned, A_1 will be 0.
- The two arrays, C and R are taken. Where
  - C = **the current allocation matrix**
  - R = **the request matrix.**
  - The i^{th} row of C tells how many instances of each resource class P_i currently holds. Thus,
  - C_{ij} is the number of instances of resource j that are held by process i.
  - Similarly, R_{ij} is the number of instances of resource j that P_i wants.

- These four data structures are shown in following figure

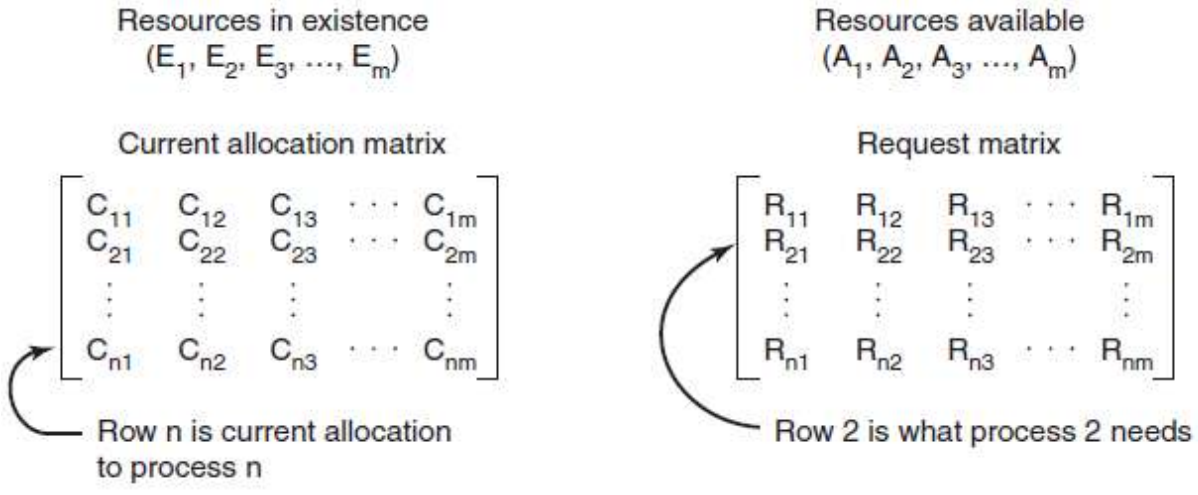


Figure: The four data structures needed by the deadlock detection algorithm.

- An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

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

  In other words, if we add up all the instances of the resource j that have been allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist.

- The deadlock detection algorithm is based on comparing vectors.
  - Mathematically, A ≤ B holds if and only if Ai ≤ Bi for 1 ≤ i ≤ m.

Main Idea:

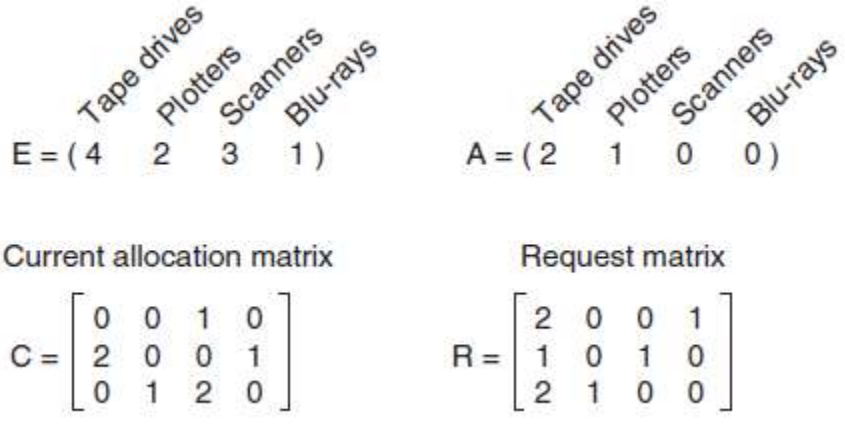
- ☑ Each process is initially said to be unmarked.
- ☑ As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked.
- ☑ When the algorithm terminates, any unmarked processes are known to be deadlocked.

(This algorithm assumes a worst-case scenario: all processes keep all acquired resources until they exit.)

The deadlock detection algorithm can now be given as follows.

1. **Look for an 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, and go back to step 1.**
4. **If no such process exists, the algorithm terminates.**

Example:



Let us consider three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanners, and Blu-ray drives.

- Process 1 has one scanner. Process 2 has two tape drives and a Blu-ray drive. Process 3 has a plotter and two scanners.
- Each process needs additional resources, as shown by the R matrix.

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied.

— The first one cannot be satisfied because there is no Blu-ray 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 = previous free + Allocation by P3

A = (2 1 0 0) + (0 1 2 0)

A = (2 2 2 0)

— At this point process 2 can run and return its resources, giving

A = previous free + Allocation by P2

A = (2 2 2 0) + (2 0 0 1)

A = (4 2 2 1)

— Now the remaining process can run. There **is no deadlock** in the system.

- Suppose that process 3 needs a Blu-ray drive as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system will eventually be **deadlocked**. Even if we give process 3 its two tape drives and one plotter, the system deadlocks when it requests the Blu-ray drive.

Another example:

Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

— Sequence <P_0, P_2, P_3, P_1, P_4> can run without deadlock

— But there is deadlock if request of P_2 is (0 0 1)

Recovery From Deadlock

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. Some way is needed to recover and get the system going again. Once the deadlock has been detected, it can be recovered through various ways.

a) Recovery through Preemption

- In this method, a resource can be temporarily taken away from its current owner and given to another process.
- The ability to take a resource away from a process, have another process use it and then give it back without the process noticing it, is highly dependent on the nature of the resource.
- Recovering this way is frequently difficult or impossible.

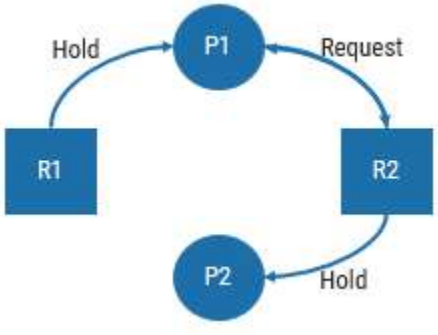


Figure: P1 has R1 and P2 has R2. If P1 requests R2 then deadlock occurs. It is recovered by temporarily taking away R2 from P2 and giving to P1

b) Recovery through Rollback

- PCB and resources state are periodically save at "checkpoint"
- When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to an earlier checkpoint when it did not have the resource.
- The resource can now be assigned to one of the deadlocked processes.
- All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again).
- In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes.
- If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

NOTE: Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, in other words, which resources are currently assigned to the process

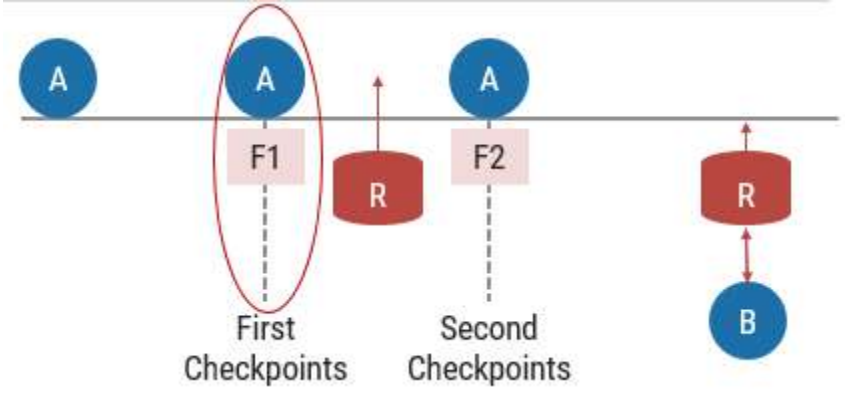


Figure: State of process A is saved at file F1 at some point of time (first checkpoint) then A has acquired resource R. At another time, state of A is saved on File F2 (second checkpoint). After that Process B has requested the resource R which caused deadlock. Now the Process A is rolled back to the first checked point.

c) Recovery through Killing Processes

- The crudest but simplest way to break a deadlock is to kill one or more processes.
  - One way is to kill all the processes involved in deadlock
  - Another way is to kill process one by one
  - After killing each process check for deadlock
    - If deadlock recovered then stop killing more processes
    - Otherwise kill another process

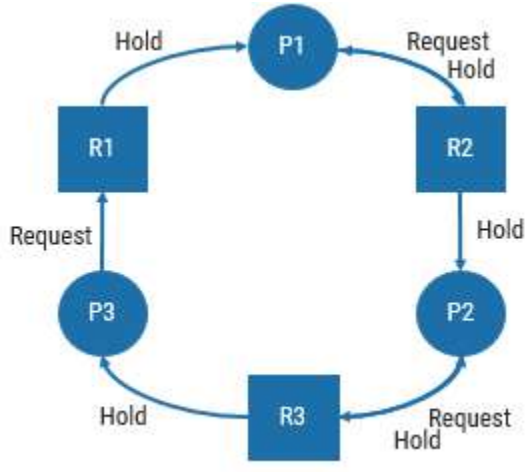


Figure: To recover from this deadlock, first kill P3 then P2

Deadlock vs Starvation

S.N.	Deadlock	Starvation
1.	All processes keep waiting for each other to complete and none get executed	High priority processes keep executing and low priority processes are blocked
2.	Resources are blocked by the processes	Resources are continuously utilized by high priority processes
3.	Necessary conditions Mutual Exclusion, Hold and Wait, No preemption, Circular Wait	Priorities are assigned to the processes
4.	Also known as Circular wait	Also known as lived lock
5.	It can be prevented by avoiding the necessary conditions for deadlock	It can be prevented by aging