Chapter 3: Deadlock

System model

- System consists of resources
- A resource can be a hardware device or a piece of information (e.g., a record in a database, a semaphore etc.)
- Two type of resources:
    - Preemptable
    - Non-preemptable
- A preemptable resource is one that can be taken away from the process owning it with no ill effects.
- Whether a resource is preemptable depends on the context. For example, on a standard PC, memory is preemptable but not on smartphones that do not support swapping or paging.
- A non-preemptable resource is one that cannot be taken away from the process owning it without potentially causing failure. For example, a printer
- Example of a preemptable resource is memory (if OS supports process swapping)
- Non-preemptable resources have to be used in a mutually exclusive manner.

- A process must request a resource before using it and must release the resource after using it.
- In general, the OS follows a protocol to use a non-preemptable resource. Sequence of events in the protocol:
    1. Request
    2. Use
    3. Release

Request

- Before using a resource, a process must request for it.
- Information about all resources are stored in a data structure called resource table.
- When a request is received, the OS checks the status of the resource. If the resource is free, the requesting process will get it. Otherwise, it must wait (be blocked) until it can acquire the resource. The process can be added to a queue of processes waiting for this resource.

Use

- If a process gets the resource, the kernel changes the status of the resource (as well as the state of the process)
- The process can then operate on the resource.

Release

- The process releases the resource through the use of a system call (e.g., close() system call for closing a file, free() for releasing memory etc.)

Deadlock

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.

Necessary conditions for resource deadlocks

Four conditions must hold simultaneously for there to be a deadlock:

1. **Mutual exclusion**
   One or more resource must be held by a process in a non-sharable mode

2. **Hold and wait**
   Processes currently holding resources that were granted earlier are waiting to acquire new resources that are currently being held by other processes

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

4. **Circular wait**
   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. That is, a set of processes {P0, P1, .., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Deadlock Modeling

- Deadlocks can be modeled using directed graphs called resource-allocation graphs (RAG).
- An RAG graph consists of two types of nodes: circles for processes and squares for resources, and a set of edges
- A directed arc from a resource node to a process node means the resource has previously been requested by, granted to, and is currently held by that process.
- A directed arc from a process to a resource means that the process is blocked waiting for the resource.
- If the graph contains no cycle, then no process in the system is deadlocked

- If the graph does contain a cycle, then a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

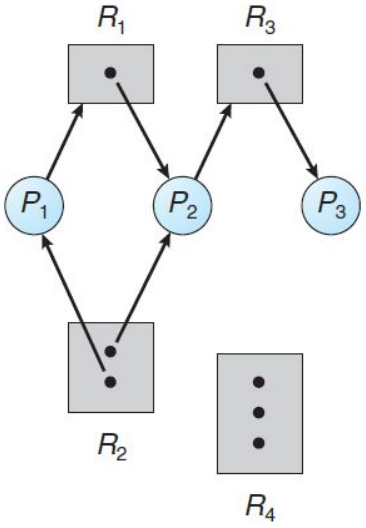


***Fig**: Resource-Allocation Graph. There is no cycle in this RAG, so there is no deadlock in this system*

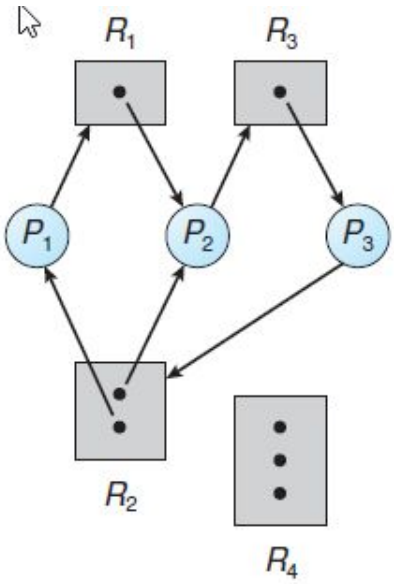


***Fig**: RAG with a deadlock. Here two cycles exist in the system: 1) P2 → R3→ P3 → R2→ P2 and 2) P1 → R1 → P2 → R3 → P3 → R2 → P1. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1. So, none of the resources can be released and no process can proceed in this state. Thus, processes P1, P2 and P3 are deadlocked.*

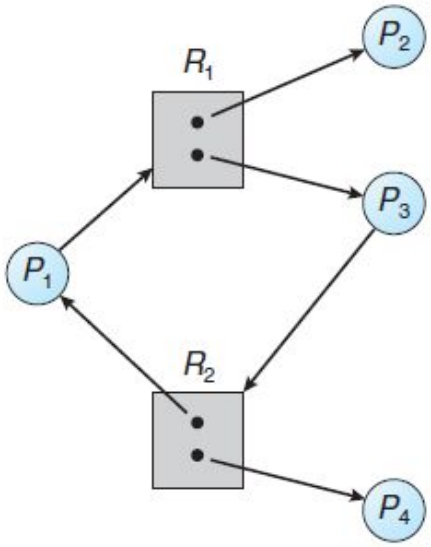


***Fig**: RAG with a cycle but no deadlock. Here, process P4 requires only R2, so once P4 no longer needs R2, it can release the instance of R2 it is currently holding. Thus, an instance of R2 will be available in some time in future. This instance can then be allocated to process P3, thereby breaking the cycle.*

Deadlock handling

1. Ignore deadlocks
2. Ensure deadlock never occurs using either
    a. Prevention, or
    b. Avoidance
3. Allow deadlock to happen and then recover. This requires using both:
    a. Detection, and
    b. Recovery

Ignore deadlocks

- Most OSs do this
- The ostrich algorithm: Stick your head in the sand and pretend there is no problem at all

Deadlock prevention

Ensure that one of the four conditions for deadlock is never satisfied.

A. Preventing mutual exclusion condition

If no resources were ever assigned exclusively to a single process, we would never have deadlocks. However, it is practically not possible.

Non-shareable resources require mutual exclusion. Otherwise, no process would be able to get the desired output.
So, mutual exclusion cannot be prevented, but the idea here is to recognize and use shareable resources as much as possible
For example, using read-only files, which do not require mutual exclusion, instead of opening a file in read-write mode when writing to the file is not needed.

B. Preventing hold and wait condition

If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks.
For this, two protocols can be used.

1. The first protocol:
   Instead of requesting a resource on a need basis, each process should request all resources at one time and in advance. No process will be allowed to execute until it gets all of its declared resources.
   Problems:
   a. A process may be idle for a long time, waiting for all the resources
   b. Reduced resource utilization because resources are not used throughout the execution of the process
   c. Some frequently-used resources will always be held up with a process, leading to starvation of other processes.
   d. Many processes do not know how many resources they need in advance.
2. The second protocol:
   A requesting process should not be holding any resource. If a process needs another resource, it must first release the resources it is currently holding.
   In other words, a requesting process is required to first temporarily release all the resources it currently holds and then try to get everything it needs all at once.

C. Preventing no-preemption condition

If a process that is holding some resource requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted / implicitly released.
The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
Problem: when a process releases its resources, it may lose all of its work. So, this method should be used with care and with the following guidelines:

1. The preemption is based on the execution of the processes.
   The process which is about to complete its execution should be allowed to continue execution and the resources of the ones which just started should be preempted.
2. The preemption is based on the priority level of the processes.
   The process with lower priority should release its resources, and let the one with higher priority complete first.

This protocol is often applied to resources whose state can be easily saved and restored, e.g., CPU registers, memory space etc.
It cannot be generally applied to resources like printers.

D. Preventing circular wait condition

Since circular wait condition is the consequence of the other three conditions, it can be prevented if any of these three conditions are prevented.
The circular wait can be prevented independently also through resource-ranking or ordering, where a global numbering / unique integer number is provided to each resource as an identification. Two approaches can be applied:

1. All resource requests must be made in numerical order (increasing sequence)
2. No process must request a resource lower than what it is already holding

Problem:
Difficult to implement practically because it may be difficult to find an ordering that satisfies everyone.

Deadlock avoidance

- System maintains a set of data using which it takes a decision whether to entertain a new request or not, to be in 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.

Example:

Suppose we have 10 instances of a resource and 3 processes using a certain number of instances of the resource. Process A is allocated 3 instances of the resource but may need as many as 9 instances. Similarly, processes B and C are allocated 2 instances of the resource and may need as many as 4 and 7 instances respectively.

Process	**Allocation**	**Max**
A	3	9
B	2	4
C	2	7

In this situation,
Total allocated = 3 + 2 + 2 = 7
Available = Total available - total allocated = 10 - 7 = 3

Currently 7 instances of the resource are being allocated to the processes. Thus, 3 instances of the resource are available.

If there exists a sequence of allocations that allows all processes to complete, then this state is safe.

To determine if the state is safe, we first check how many more instances are required by each process to complete its execution. That is, we compute Need = Max - Allocation.

Process	**Allocation**	**Max**	**Need**
A	3	9	9 - 3 = 6
B	2	4	2
C	2	7	5

Since only 3 instances are available, the only process that can be executed to completion even if it requests its maximum number of requests is process B, which needs only 2 more instances. If we allocate 2 instances to B, it will lead to the following state:

Process	**Allocation**	**Max**	**Need**
A	3	9	6
B	4	4	0
C	2	7	5

Now, Available = Previously available - just allocated = 3 - 2 = 1

When B completes, the resource held by B will be released. Hence, the state will be in the following state:

Process	**Allocation**	**Max**	**Need**
A	3	9	6
B	0	-	-
C	2	7	5

Available = Previously available + Number of instances released by the completed process
= 1 + 4 = 5

Similarly, now the scheduler can run process C because $Need_c$ <= $Available_c$, leading to the following state:

Process	Allocation	Max	Need
A	3	9	6
B	0	-	-
C	7	7	0

Available = 5 - 5 = 0

When C completes, the system will be in the following state:

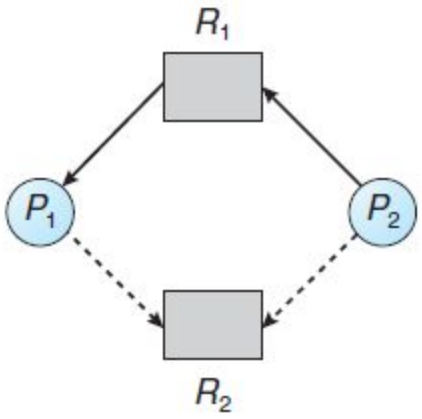
Process	Allocation	Max	Need
A	3	9	6
B	0	-	-
C	0	-	-

Available = 0 + 7 = 7

Now, A can execute too completion because it requires 6 more instances and we have 7. Thus, running these 3 processes in the sequence B, C, A will guarantee that all processes will finish without a deadlock. Thus, the system is in safe state.

Resource-Allocation Graph Algorithm

A variant of resource-allocation graph (RAG) can be used to avoid deadlocks in a system with only one instance of each resource type. This graph is known as a wait-for graph. Unlike RAG, it contains claim edges from process Pi to resource Rj, which indicate that a Pi may request a Rj at some time in future and are represented by dashed lines. When Pi requests resource Rj, the claim edge is converted to a request edge. When Rj is released by Pi, the assignment edge is reconverted to a claim edge.



When a process requests a resource, a cycle detection algorithm is applied to check if converting the claim edge to a request edge will result in circular wait. If it leads to a cycle, the request will be rejected.

Banker's algorithm

RAG-based cycle detection algorithm cannot be applied when there are multiple instances of resources.
Banker's algorithm takes analogy of a bank, where customers requests to withdraw cash (loan) and the banker cannot give more cash that what a customer has requested for and the total available cash.
This algorithm has two parts:

1. Safety test algorithm that checks if the current state is safe or not.
2. Resource request handling algorithm that verifies whether the requested resources when allocated to the process, affect the safe state. If it does, the request is denied.

To implement Banker's algorithm, we need the following data structures for n processes and m resources:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.

Max: An n x m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process P_i may request at most k instances of resource type R_j .

Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process P_i is currently allocated k instances of resource type R_j.

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P_i may need k more instances of resource type R_j to complete its task
Need[i][j] = Max[i][j] - Allocation[i][j]

Let X and Y be vectors of length n.
We say that X ≤ Y if and only if X[i] ≤ Y[i] for all i = 1, 2, ..., n.

For example, if X = (1, 7, 3, 2) and Y = (0, 3, 2, 1), then Y ≤ X.
In addition, Y < X if Y ≤ X and Y ≠ X.

Safety test algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.
2. Find an index i such that both
    a. Finish[i] == false
    b. $Need_i \leq Work$
    If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
    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 handling algorithm

Let $Request_i$ be the request vector for process P_i. If Requesti [j] == k, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq$ Available, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
    $Available = Available – Request_i$;
    $Allocation_i = Allocation_i + Request_i$;
    $Need_i = Need_i – Request_i$;

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $Request_i$, and the old resource-allocation state is restored.

Example: Safety test algorithm

Consider a system with the following information. Determine whether the system is in safe state.

Total resources:

R1	R2	R3
15	8	8

Process	Allocation			Max		
	R1	**R2**	**R3**	**R1**	**R2**	**R3**
P1	2	1	0	5	6	3
P2	3	2	3	8	5	6
P3	3	0	2	4	8	2
P4	3	2	0	7	4	3
P5	1	0	1	4	3	3

Solution

	R1	**R2**	**R3**
Total allocated resources =	12	5	6

The total number of available resource instances is calculated by

$Available_i = Total\ instance_i - \sum_{all\ processes} Allocation_i$

Hence, the available resources are:

	R1	R2	R3
[15 8 8] - [12 5 6] =	[3	3	2]

Next, we find the Need matrix by subtracting Allocation from Max

Process	Need		
	R1	**R2**	**R3**
P1	5 - 2 = 3	6 - 1 = 5	3 - 0 = 3
P2	5	3	3
P3	1	8	0
P4	4	2	3
P5	3	3	2

To execute the safety test algorithm to find whether the state is safe, we first initialize a vector, Finish, to false.

	Finish
P1	False
P2	False
P3	False
P4	False
P5	False

The sequence of the processes should be such that each process satisfies the criteria Need ≤ Available

Here, only process P5 satisfies this criteria. So, if the resources are allocated to P5, it will be able to execute. After it finishes its execution, it must release all the resources it is holding, so that the next process can avail them as per its need. Thus, after P5 finishes its execution, total number of available resources will be
Available = Available + Resources held by P5 = [3 3 2] + [1 0 1] = [4 3 3]

Repeating this procedure until all processes complete their execution, we get the following result:

Process found	**Current availability**	**Finish**
P5	[3 3 2] + [1 0 1] = [4 3 3]	Finish[P5] = True
P4	[4 3 3] + [3 2 0] = [7 5 3]	Finish[P4] = True
P1	[7 5 3] + [2 1 0] = [9 6 3]	Finish[P1] = True
P2	[9 6 3] + [3 2 3] = [12 8 6]	Finish[P2] = True
P3	[12 8 6] + [3 0 2] = [15 8 8]	Finish[P3] = True

From the table, it can be observed that Finish is true for all processes. Therefore, the system is in safe state.

Example: Resource-request algorithm

If P4 requests two more instances of R1 and two instances of R3 in the previous, will the system still be in safe state?

Solution

Here, Req4 = [2 0 2]
Req4 ≤ Need4 and Req4 ≤ Available, so the request is addressable but we first need to check if the allocation of these resources lead to an unsafe state or not.

Let's pretend that we allocate these resources, then the state of the system would be

Process	Allocation			Max			Need		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	1	0	5	6	3	3	5	3
P2	3	2	3	8	5	6	5	3	3
P3	3	0	2	4	8	2	1	8	0
P4	5	2	2	7	4	3	2	2	1
P5	1	0	1	4	3	3	3	3	2

Current availability = Total resources - total allocated = [15 8 8] - [14 5 8] = [1 3 0]

Now, there is no process whose Need ≤ Available. So, no process can be started. Thus, if the Req4 is entertained, the state would be unsafe. So, the OS should not grant the resources requested by P4, until the system is in a safe state.

Problems with deadlock avoidance algorithm

It is not practically possible to implement Banker's algorithm because

1. It is almost impossible to have the knowledge of maximum demand of each process in advance. However, with some analysis on a stable system, the maximum demand of processes for resources can be estimated.
2. In a multiprogramming system, the number of processes is not fixed.
3. It is important that the resources must be available when a process requests for them, otherwise the algorithm cannot be implemented

Deadlock detection

Lets the deadlocks to occur, tries to detect when this happens, and then takes some action to recover from deadlocks.

Deadlock detection with one resource of each type

This can be done using a wait-for graph, which is a variant of RAG.
A wait-for graph is from the RAG by removing the resource nodes and collapsing the appropriate edges.

If the wait-for graph contains a cycle, there is certainly a deadlock in the system.

Deadlock detection with multiple resources of each type

An algorithm similar to Banker's algorithm can be used to detect deadlock in the system with multiple resources of each type. This algorithm employs several time-varying data structures similar to those of Banker's algorithm:
Available: A vector of length m indicates the number of available resources of each type
Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process
Request: An n x m matrix indicates the current request of each process. If Request[i][j] equals k, then process P_i is requesting k more instances of resource type R_j

1. Let Work and Finish be vectors of length m and n, respectively.
   Initialize Work = Available.
   For i = 0, 1, ..., n–1, if $Allocation_i \neq 0$, then Finish[i] = false. Otherwise, Finish[i] = true.
2. Find an index i such that both
   a. a. Finish[i] == false
   b. b. $Request_i \leq Work$
   c. If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[i] = true
   Go to step 2.
4. If Finish[i] == false for some i, $0 \leq i < n$, then the system is in a deadlocked state.
   Moreover, if Finish[i] == false, then process P_i is deadlocked.

Example:

Consider a system with the following information:

Total resources:

R1	R2	R3
5	6	4

Process	Allocation			Request		
	R1	**R2**	**R3**	**R1**	**R2**	**R3**
P1	1	0	2	1	0	0
P2	1	1	0	4	0	2
P3	1	1	0	0	0	2
P4	0	2	1	2	1	0
P5	1	2	0	3	1	4
Total allocated resources =	**4**	**6**	**3**			

Total number of available resource instances:
Available = Total resources - Total allocated = [5 6 4] - [4 6 3] = [1 0 1]

Executing the detection algorithm
We first initialize a vector of length n, Finish, to false

	Finish
P1	False
P2	False
P3	False
P4	False
P5	False

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied, i.e. Request ≤ Available

Here, request of P1 = [1 0 0] and Available = [1 0 1]
Thus, P1's request can be satisfied. After P1 finishes its execution, it must release all the resources it is holding so Available will then become [1 0 1] + [1 0 2] = [2 0 3]

Repeating this procedure until all processes complete their execution, we get the following result:

Process found	Current availability	Finish
P1	[1 0 1] + [1 0 2] = [2 0 3]	Finish[P1] = True
P3	[2 0 3] + [1 1 0] = [3 1 3]	Finish[P3] = True
P4	[3 1 3] + [0 2 1] = [3 3 4]	Finish[P4] = True
P5	[3 3 4] + [1 2 0] = [4 5 4]	Finish[P5] = True
P2	[4 5 4] + [1 1 0] = [5 6 4]	Finish[P2] = True

Here, all the elements of Finish are true. Therefore, the system is not in deadlocked state. If the processes are executed in the order <P1, P3, P4, P5, P2>, then there will be no deadlock.

Another example:

Consider a system with the following information:

Total resources:
R1 R2 R3
5 6 4

Process	Allocation			Request		
	R1	**R2**	**R3**	**R1**	**R2**	**R3**
P1	1	0	2	1	0	0
P2	1	1	0	4	0	2
P3	1	1	0	0	0	2
P4	0	2	1	2	2	0
P5	1	2	0	3	1	4
Total allocated resources =	**4**	**6**	**3**			

Executing the deadlock detection algorithm, we obtain the following order of processes:

Process found	Current availability	Finish
P1	[1 0 1] + [1 0 2] = [2 0 3]	Finish[P1] = True

P3	[2 0 3] + [1 1 0] = [3 1 3]	Finish[P3] = True

After P3, no resource request can be satisfied. Therefore, the system is in deadlocked state due to processes P4, P5 and P2.

Detection algorithm usage

If deadlocks occur frequently, then the detection algorithm should be invoked frequently.

- One possibility is to check every time a resource request is made. But it is potentially expensive in terms of CPU time.
- A less expensive alternative is to check every k minutes, or perhaps only when the CPU utilisation has dropped below some threshold.

Deadlock recovery

A. Recovery through resource preemption
   - Temporarily take a resource away from its current owner and give it to another process
   - If preemption is required to deal with deadlocks, these three issues need to be addressed:
     1. Selecting a victim
        - We must determine the order of preemption to minimize cost.
        - If the number of resources is less compared to the requirement of other process to break the deadlock, or if the process is not holding the desired type of resource, then those resource need to be preempted.
        - A process whose execution has just started and requires many resources to complete will be the right victim for preemption.

     2. Rollback
        If a resource is preempted from a process, it cannot continue with its normal execution, so we must rollback the process to some safe state, and restart it from that state.

     3. Starvation
        - We must avoid the situation where the same process is always picked as a victim, leading to starvation.
        - We must ensure that a process can be picked as a victim only a (small) finite number of times.

B. Recovery through process termination (killing process)
   1. Abort all deadlocked process
      This method is a costly solution since these processes may lose their work.

   2. Abort one process at a time until the deadlock cycle is eliminated.

- After each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- We must choose the viction carefully. Where possible, it is best to kill a process that can rerun from the beginning with no ill effects. For example, a compilation process is better to be aborted than a process doing some database transaction. Other factors to determine the victim process:
    - Priority of the process
    - Execution span of the process (how long it can be executed and how much longer it will compute before execution).
    - Type of the process (batch/ interaction)
    - Number and type of resources a deadlocked process is holding / needed by the process in order to complete.
    - etc.