Chap 5: Deadlock

BY PRATYUSA MUKHERJEE, ASSISTANT PROFESSOR (I)

KIIT DEEMED TO BE UNIVERSITY

Background

- In a multiprogramming environment, several processes may compete for finite number of resources.
- A process requests resources, if they are not available, the process enters waiting state.
- **When a waiting process is never again able to change its state, because the resources it has requested are held by other waiting process, we call this situation a deadlock**.

System Model

- System has finite number of resources to be distributed among number of competing processes.
- Resource types $R_1, R_2, \ldots, R_m$. Example of resources CPU cycles, memory space, I/O devices
- Resources may be partitioned into some number of identical resources called instances. Thus each resource R_i has W_i instances.
- Each process utilizes a resource as follows: **request , use** and **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously. The necessary conditions of deadlock are:

- **Mutual exclusion**: Only one process at a time can use a resource (Non sharable resources)
- **Hold and wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: A resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

Resource-Allocation Graph

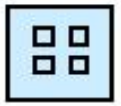
- Deadlocks can be described more precisely in terms of a **directed graph** called system **resource allocation graph**.
- This graph has a set of vertices V and a set of edges E.
- V is partitioned into two types of nodes:
  - P = $\{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system. Represented as circles.
  - R = $\{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system. Represented as boxes.
- Edges are also categorized into two
  - **Request edge**: Its a directed edge Pi → Rj signifying the Pi has requested an instance of Rj and is currently waiting for it to be granted.
  - **Assignment edge**: It is a directed edge Rj → Pi signifying that an instance of Rj has been allocated to process Pi.

- Resource type Rj may have more than one instance, thus they are represented as dots within the rectangle.
- **Request edge** points to only the rectangle corresponding to Rj
- **Assignment edge** must also designate one of the dots in the rectangle.
- When Pi requests an instance of Rj, a request edge is inserted in the resource allocation graph.
- When the request can be fulfilled, the request edge is instantaneously transformed to an assignment edge.
- When the process no longer requires the resource, it releases the resource. Thus, the assignment edge is deleted.

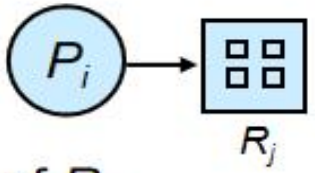
Example illustrated on white board

- Process

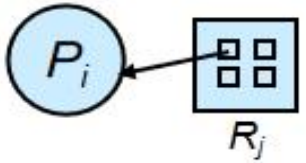
- Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



Utility of Resource Allocation Graph

- If the graph has no cycles, there is no deadlock.
- If the graph has a cycle, deadlock may exist.

Important Points:

- If each resource has only one instance, a cycle implies that deadlock has occurred. Thus in this case, a cycle in the graph is both a necessary and a sufficient condition for deadlock

- If resource has multiple instances, a cycle does not necessarily imply a deadlock. Thus in this case, a cycle in the graph is a necessary but not a sufficient condition for deadlock

Example illustrated on white board

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state by some protocols:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state, detect it and then recover
- Ignore the problem and pretend that deadlocks never occur in the system (used by most operating systems, including UNIX)

Deadlock Prevention vs Avoidance

- Set of methods to ensure that atleast one of the necessary conditions cannot hold.
- These methods prevent deadlock by restricting how the request for resources can be made.

- Here OS is given some additional information in advance concerning which resources a process will request and use in its lifetime.
- Thus now OS can decide for each request, whether or not the process should wait.
- To decide whether the request can be granted or not (immediately or in future), the system must consider the
  - resources currently available
  - currently allocated
  - future requests
  - release of each process.

Deadlock Prevention

Mutual Exclusion (ME):

- It mandates that atleast one resource is non sharable. Thus to deny it, go for sharable resources
- Sharable resources dont need ME thus are not involved in deadlock. For eg: read only files. A process never waits for sharable resources.
- But we cant prevent deadlock by denying ME as some resources are intrinsically non sharable. For eg: Mutex

Hold and Wait:

- OS must guarantee that whenever a process requests a resource, it does not hold any other resources at that time.
- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
- Low resource utilization; starvation possible

Example discussed verbally

No Preemption:

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular Wait:

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Example discussed verbally

Deadlock Avoidance

- Deadlock prevention ensures that atleast one of the necessary conditions for deadlock cannot occur. However,this may lead to low device utilization and reduced system throughput.
- An alternative method to avoid deadlock is to gather additional information about how resources are to be required.
- In order to do so, the system considers the resources currently available, currently allocated to each process and the future requests and releases of each process. Requires that the system has some additional ***a priori*** information available.
- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the **resource-allocation state** to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the **number of available and allocated resources, and the maximum demands of the processes**

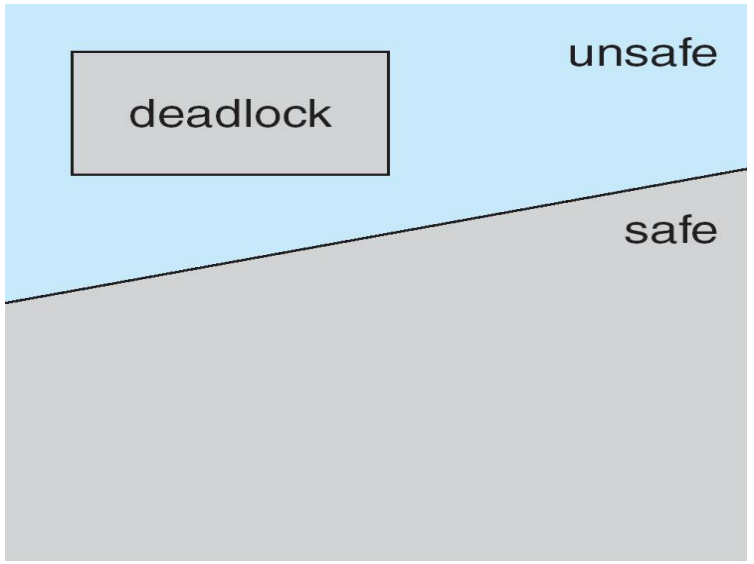
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence <P1, P2, …, Pn> of ALL the processes in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the Pj, with j < I
- That is:
  - If Pi resource needs are not immediately available, then Pi can wait until all Pj have finished
  - When Pj is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate
  - When Pi terminates, Pi +1 can obtain its needed resources, and so on
  - If no such sequnec exists, system is said to be **unsafe.**

further illustrated on white board

Important 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.



Avoidance Algorithms

Single instance of a resource type

- Use a resource-allocation graph

Multiple instances of a resource type

- Use the banker's algorithm

Resource-Allocation Graph for Deadlock Avoidance

- In addition to Request and Assignment edge, **Claim edge** Pi → Rj indicates that process Pj may request resource Rj in future; thus it is represented by a **dashed line**
- Claim edge converts to Request Edge as soon as process requests a resource.
- Request edge converted to an Assignment Edge whenever the resource is allocated to the process.
- When a resource is released by a process, Assignment Edge reconverts to a Claim Edge
- Resources must be claimed a priori in the system i.e before Pi starts execution all its claim edges must be added to the graph.

further illustrated on white board

Important Points for RAG based DA

- Suppose that process Pi requests a resource Rj
- The request can be granted only if converting the Request Edge to an Assignment Edge does not result in the formation of a cycle in the resource allocation graph.
- If no cycle exists, allocation of the resource will leave the system in safe state.
- If a cycle if found, then alloaction will put the system in unsafe state. Thus Process Pi will have to wait for the request to be satisfied. **(Can you guess whether unsafe state will lead to a DL or not??)**

further illustrated on white board

n^2 opeartion to detect existence of cycle

Banker's Algorithm

- Deadlock Avoidance Algorithm for resources with multiple instances
- Each process must a priori claim maximum number of instances of each resource that it may need whenever it enters the system.
- When a process requests a resource, system must determine whether allocation of this resource will leave the system in a safe state.
- If it will, resources are allocated otherwise it may have to wait until some other process releases enough resources to satisfy him.
- When a process gets all its resources and finishes its task, it must return them in a finite amount of time.

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 x 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

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

Initialize:

(a) **Work = Available**

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

Step 2: Find an i such that both:

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

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

If no such i exists, go to step 4

Step 3: $\mathbf{Work = Work + Allocation_i}$
Finish[i] = true
go to step 2

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

Resource-Request Algorithm for Process Pi

$Request_i$ = request vector for process Pi. If Requesti [j] = k then process Pi wants k instances of resource type Rj

1. If **$Request_i \leq Need_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If **$Request_i \leq Available$**, go to step 3. Otherwise Pi must wait, since resources are not available

3. Pretend to allocate requested resources to Pi by modifying the state as follows:

$Available = Available - Request_i$;

$Allocation_i = Allocation_i + Request_i$;

$Need_i = Need_i - Request_i$;

After this, If safe $\Rightarrow$ the resources are actually allocated to Pi

If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored

example of Banker's Algorithm illustrated on white board

Deadlock Detection

- If a system does not employ DP or DA then DL may occur. In such situations, the system must provide
  - An algo. to examine the state of the system to determine if DL has occurred or not
  - An algo. to recover from DL if it has occurred.
- Detection and Recovery scheme needs overhead tht includes run time costs to maintain necessary info and execute detection algo. plus the potential losses inherent in recovering from a deadlock.

Single Instance of Each Resource Type

- Maintain **wait-for graph**
  - Nodes are processes
  - Draw Pi → Pj , if Pi is waiting for Pj to release some R.

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

example illustrated on white board

Several Instances of a Resource Type

- **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] = k, then process Pi is requesting k more instances of resource type Rj.

Detection Algorithm

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

Initialize:

(a) Work = Available

(b) For i = 1,2, …, n, if $Allocation_i \neq 0$, then Finish[i] = false; otherwise, Finish[i] = true.

Step 2: Find an index i such that both

(a) Finish[i] == false

(b) $Request_i \leq Work$

If no such i exists, go to step 4

Detection Algorithm (contd)

Step 3: Work = Work + $Allocation_i$
Finish[i] = true
go to step 2

Step 4: If Finish[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish[i] == false, then Pi is deadlocked.

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

further illustrated on white board

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by DL?
- **If DL occurs frequently, DD algo must be invoked frequently** as resources allocated to deadlocked processes will be idle until DL is broken. Also no. of processes involved in DL cycle may grow.
- **We can also invoke DD algo every time a request for allocation cannot be granted immediately.** We can identify the deadlocked set of processes and also find which specific process "caused" DL. But this is expensive.
- **Better is to invoke DD algo at defined intervals of time** (Rg after every 2 hrs or when CPU utilization is less than 40%)
- **If detection algorithm is invoked arbitrarily**, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Deadlock Recovery by Process Termination

- System basically reclaims all its allocated resources to these terminated processes.
- Abort all deadlocked processes (expensive)
- Abort one process at a time until the deadlock cycle is eliminated (high oerhead as DD algo must be invoked after each abortion to check if DL still persists)

In which order should we choose to abort?

- Priority of the process
- How long process has computed, and how much longer to completion
- Resources the process has used
- Resources process needs to complete
- How many processes will need to be terminated
- Is process interactive or batch?

Deadlock Recovery by Resource Preemption

- We preempt some R from Ps and give these to other Ps until DL is broken.
- **Selecting a victim** – Which R and which P are to be preempted? the order must be such that its cost minimal.
- **Rollback** – If we preempt a R from a P, what to do with that processs?? It definitely cant continue so better to return/roll back to some safe state and restart process for that state (need to know that state of all running processes)
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor