Deadlock

1 Introduction

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Deadlock can arise if the 4 necessary and sufficient conditions hold simultaneously.

- **Mutual Exclusion:** Only one process at a given time can use the resource. If another process requests that resource, the requesting process must be delayed until the release of resource.
- **Hold and Wait:** The processes must not release the resources they have already been allocated while waiting for other (requested) resources. If the process had to release its resources when a new resource was requested, deadlock could not occur.
- **No Preemption:** The processes must not have resources taken away while that resource is being used.
- **Circular Wait:** A circular chain of processes, with each process holding resources which are currently being requested by the next process in the chain.

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work.

2 Resource Allocation Graph

Deadlocks can be described in terms of a directed graph called a **system resource-allocation graph**. $P_i \to R_j$ (**request edge**) signifies P_i has requested an instance of resource type R_j and is currently waiting for that resource. $R_j \to P_i$ (**assignment edge**) signifies that an instance of resource type R_j has been allocated to process P_i.

The OS adopts 2 strategies to allocate resources to user programs once they have been de-allocated by another process.

- **Resource Partitioning:** OS decides beforehand what resources should be allocated to which program. It creates partitions and places resources in them and allocated a partition with all its resources to the process. It is easy to implement and has less overhead but lacks flexibility. If it contains more resources, they are wasted and if it has less resources, the partition is useless.
- **Pool-based:** There is a common pool of resources. The OS checks the allocation status in the resource table and allocates it if free. Resources are not wasted but there is the overhead of allocating and de-allocating resources on every request and release.

3 Deadlock Prevention

Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

- **Mutual Exclusion:** Not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.
- **Hold and Wait:** Allocate all required resources to the process before the start of its execution but it will lead to low device utilization. The process will make a new request for resources and can lead to **starvation**.
- **No Preemption:** Preempt resources from the process when resources required by other high priority processes.
- **Circular Wait:** Each resource will be assigned with a numerical number. A process can request the resources increasing/decreasing. order of numbering.

4 Banker's Algorithm for Deadlock Avoidance

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not.

- **Available:** 1-d array of size 'm'. Available[j]=k means there are k instances of R_j.
- **Max:** 2-d array of size 'n*m' Max[i,j]=k means P_i may request at most k instances of R_j.
- **Allocation:** 2-d array of size 'n*m' Allocation[i,j]=k means P_i currently allocated k instances of R_j.
- **Need:** 2-d array of size 'n*m' Need[i,j]=k means P_i currently needs k instances of R_j.

Safety Algorithm

1. Let $Work$ and $Finish$ be vectors of length 'm' and 'n' respectively.Initialize:
   $Work = Available$
   $Finish_i = false$ for i=1, 2, 3, 4....n
2. Find an i such that both
   a) $Finish_i = false$
   b) $Need_i <= Work$
   if no such i exists goto step (4)
3. $Work = Work + Allocation_i$
   $Finish_i = true$
   goto step (2)
4. if $Finish_i = true$ for all i then the system is in a safe state

Request-Resource Algorithm

1. If $Request_i <= Need_i$
   Goto step (2) ; otherwise, raise error since the process exceeded its maximum claim.
2. If $Request_i <= Available$
   Goto step (3); otherwise, P_i must wait since resources are unavailable.
3. Have system pretend to have allocated requested resources to P_i by modifying state as:
   $Available = Available – Request_i$
   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Need_i – Request_i$

5 Deadlock Detection and Recovery

If resources have a single instance, presence of cycle in RAG is sufficient condition for deadlock. If there are multiple instances of resources, it is not sufficient condition.

A traditional OS such as Windows doesn't deal with deadlock recovery as it is time and space consuming. The recovery methods include **killing process** till a deadlock exists. Another method is **preempt allocated resources** till deadlock exists. This can lead to **starvation**.