**OPERATING SYSTEMS LEC 05**

**Deadlock**

* Several process may compete for finite resources; some may wait forever due to resources being held for other processes.
* Processes are in deadlock state if every process in set is waiting for an event that can be caused only by another process in the set.
* Can be resolved by pre-empting resources and rollback.

**System Model**

* Resources are partitioned into several types, each consisting of identical instances
  + Identical: allocation of any instance of the type will satisfy processes’ request
  + Resources can be physical (CPU cycles, printers) or logical resources (files, semaphores, monitors).
  + Pre-emptible resource is one that can be taken away from process with no ill effect to it. Eg. Memory
  + Non-preemptible resource cannot be taken from its user, will make user fail
    - Eg. Printers. Potential deadlocks involve this resource type.
* Each process uses a resource as follows:
  + Request the resource – wait if its being used
  + Use the resource – eg. Process can print on printer
  + Release the resource

**Necessary conditions for deadlock**

Four conditions must hold for deadlock to occur:

1. Mutual exclusion condition – only one process at a time can use resource
2. Hold and wait condition – process holding at least one resource is waiting to acquire held resources from other processes
3. No Preemption condition – Resource can be released only voluntarily by process holding it after it has completed its task.
4. Circular wait condition – A set of waiting processes in which the next processes is waiting for the previous resources, creating a circle loop. Where the end process is waiting for the first.

These are not completely independent. Circular-wait condition implies hold and wait condition

**Deadlock Modelling**

* Deadlocks can be explained with a graph G(V, E)
* V is partition into:
  + Set of processes in system. ‘P’
  + Set of all resource types in system. ‘R’
* Request edge – directional edge Pi -> Rj
  + Process requests instance of resource
* Assignment edge – directed edge Rj -> Pi
  + Resource has been allocated to process

A close up of a map

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Methods for Handling deadlock**

* Use protocol to ensure system never reaches deadlock
  + Using deadlock prevention and/or avoidance
* Allow system to enter deadlock state and recover
  + Needs deadlock prevention and recovery
* Ignore problem and pretend deadlocks never occur in system
  + Used by most OS, including UNIX. **Ostrich** algorithm

**Deadlock Prevention**

Restrain the way resources request can be made.

* Use method to ensure that any one of the deadlock conditions cannot hold
* Deny mutual exclusion
  + Not required for sharable resources
    - read-only files cannot be in deadlock
  + Must hold for non-shareable resources
    - printer cannot be shared by several resources
  + In general, not possible to prevent deadlock by denying mutual condition since some resources are non-sharable
* Deny hold and wait
  + When a process requests a resource, must not hold other resources
  + Options:
    - Each process granted all resources before it starts
    - Allows process to request resources when it has none
  + Problem:
    - Resource allocation low
    - Possible starvation – waiting for popular resource can wait forever
* Prevent no pre-emption (allow pre-emption)
  + When a process holding some resources requests another resource that cannot be immediately allocated, it must release all resources currently held
    - Pre-empted resources are added to process’ list of requested resources
    - Process is restarted when it regains old resources and obtains new one it is requesting
  + Problem is it can be applied easily to resources who state can be saves easily such as memory but not easily for others (printer)
* Deny circular wait
  + All resource types are ordered
  + Each process must request increasing order of resources
  + Protocol:
    - Each processes requests resources in increasing order
    - Initially a process can request for any resource
    - After that it can only request resources with larger number
  + Problem: may be impossible to find resource ordering that satisfies everything

**Deadlock Avoidance**

* System must have additional priority information for which resources a process will request and use during its lifetime
  + System can decide for each request whether or not process should wait
  + Simplest and most useful model makes each process declare maximum number of resources of each type it may need
* Algorithm dynamically examines resource-allocation state to ensure there can never be a circular-wait condition
* A resource-allocation state is defined by:
  + Number of available and allocated resources and
  + Maximum demands of the processes

**Safe State**

* When process request available resource, system checks if allocation keeps system in safe state
* Will be in safe state if there exists a safe sequence of all processes
* A sequence is safe if for each process, the resources requested by a process can be allocated from the currently available resources and resources held by the previous processes
  + Process will wait until previous resources have finished
  + When previous processes finished, next process obtains all needed resources and then executed and retunes resources and terminates.
  + When a process terminates, the next process can obtain its needed resources
* Basic facts
  + If system is safe – no deadlocks
  + Unsafe state – possibility of deadlocks
  + Avoidance ensures system can never enter unsafe state
  + A process requesting for available resources may have to wait
    - Resource allocation is lower than without deadlock avoidance

**Resource-Allocation Graph Algorithm**

* Claim edge P->R indicates the process P may request resource R
* Claim edge converts to request edge when process requests a resource
* Assignment edge converts to claim edge when resource is released by process
* Resources must be claimed a priori in the system
* Need a cycle detection algorithm – O(n2)
* Cannot be used for systems compromising resource types with multiple instances

**Banker’s Algorithm**

* Algorithm for system compromising resource types with multiple instances
* Similar to bank – don’t allocate available cash if it can no longer satisfy needs of all customers
* Each process must claim maximum number of instances of each resource type that it may need.
* When a process requests resources it may have to wait until some other processes releases enough resources
* When process gets all its resources it must return them in a finite amount of time

**Detection Algorithm Usage**

* Based on how often we believe deadlocks occur and how many processes will be affected by deadlock when it occurs
* Invoke every time a process makes a request would be expensive but can identify the processes that cause the deadlock; good for debugging
* Invoke at less frequent intervals
  + Eg. When CPU utilisation falls below 40% or every hour

**Deadlock Recovery**

* Terminate processes
  + Abort all deadlocked processes
  + Kill one process at a time until deadlock cycle eliminated
  + In which order should we choose process to abort?
    - The process with lowest priority
    - How long the process has computed, and how much longer to completion
    - Resources the process has used
    - Resources the process needs to completed
    - How many processes will need to be terminated
    - Is process interactive or batch?
  + Problem: aborting process can lead to incorrect file
    - Eg. Aborting process in middle of updating file
* Pre-empt a resource from a process
  + Select victim process to minimise cost
  + Roll back process to a safe start and restart from there
    - Easiest way: destroy process and restart
    - Use checkpoints during execution
  + Some process may always be picked as victim (Starvation)
    - Include number of rollbacks in cost factor to ensure no starvation occurs

**Combined Approach to deadlock handling**

* Combine three approaches
  + Prevention
  + Avoidance
  + Detection
* Partition resourced into hierarchal classes and use appropriate techniques for handling of deadlocks within the classes.