Chapter 3

Deadlocks

DEADLOCKS

Objectives

- ❖ To understand the concept of a deadlock.
- ❖ To study the classical treatment of deadlocks in the context of an OS.
- ❖ Exposure to deadlock treatment in contemporary OS.

DEADLOCKS

The Deadlock Problem (various situations)

Two persons trying to ring each other up.

- Two persons going through a narrow lane in opposite directions.
- Two persons crossing a river in opposite direction using a row of stepping stones.
- Two persons going through a narrow lane in opposite direction; one dead, blocking the lane.
- A kid at a table who would not leave till he has eaten 5 eggs, when there are no eggs there.
- Two kids at a table who would not leave till each has eaten 3 eggs and now none are left.

DEADLOCKS

Entities in a Deadlock

- Processes
- Resources

Assumptions

- Processes progress when given required resources
- There are adequate resources to fulfill the needs of any single process
- Processes release resources after use(in finite time)
- Consider crossing a river using a row of stepping stones …
- Processes = persons
- Resource = a stepping stone
- Holding a resource = standing on a stepping stone
- Clash/contention = 2 or more persons trying to step on same stone.
- Deadlock = contention in opposite direction.

DEADLOCKS

❖ How to handle a deadlock?

➢ Retreat (rollback)

- Persons behind you? all must retreat
  - ❑ (cascading effect)
- Both sides rollback and retry?
  - ❑ Indefinitely?
  - ❑ Livelock !

DEADLOCKS

❖ Avoid/prevent deadlocks. How?

➢Have two rows of stepping stones, one for each direction.

➢Find out if somebody crossing from other side.

If yes, then wait,   If no, then proceed.

Simultaneous checking?

- Both proceed? Deadlock!
- Both wait? (is it dead/live lock ?)

give higher priority to one bank

- - Fairness?
- - Starvation?

atomic indicator lamps

- - press button, get green before crossing
- - go-ahead given to one at a time
- - switch-off after crossing
- - (atomic => indivisible)

System Model

Finite number of resources

Different resource type

1or more, functionally equal, instances in each type e.g.

- memory blocks
- CPUs
- I/O devices

competing processes

- Resource requests (OS Call).

  May request 1 or more types,

  1 or more instances

- resource use
- resource release (OS Call)

if requested resource (s) not available,

requester waits

Deadlocks

Definition :

A set of processes in a deadlock if each waiting for an event that can be caused by other process (es).

- Event : resource release / (other events)

Examples:

- 3 processes , each holding 1 tape-drive, wanting 1 more (total # is 3)
- P_1 holding card-reader, wanting tape driveP_2 holding tape drive, wanting card reader
- 1 tape drive in the system, P_1 holding it,wanting 1 more . (?)
- P_1 holding tape drive, wanting processor.P_2 running on the processor, wanting tape drive. (?)

Characterization

- Necessary conditions:

1. Concurrency

2. Mutual exclusion : (At least some) resources cannot be shared.

   One user at a time.

3. Hold & wait : Processes hold (mutex) resources and wait for additional resources if they are currently held.

4. No preemption : Resource released only voluntarily by process, after use.

5. Circular wait : A cycle formed by waiting processes.

Characterization

Resource Allocation Graph :

- $P = \{P_1, P_2, \ldots, P_m\}$ is set of all processes in the system
- $R = \{R_1, R_2, \ldots, R_n\}$ is set of all resource types
- $R_i = \{r_{i1}, r_{i2}, \ldots, r_{ix}\}$ is set of x equal instances of resource type i
- $V = P \cup R$ is set of vertices in the graph
- $D = \{(P_i, R_j) / P_i \in P \wedge R_j \in R\}$ is the set of current requests for resources
- $H = \{(r_{jk}, P_i) / r_{jk} \in R_j \wedge R_j \in R \wedge P_i \in P\}$ is set of resource allocations (assignments).
- $E = D \cup H$ is set of (directed) is in the graph.

Resource Allocation Graph

- Representation :
  - r_{jk} : a dot
  - R_j : a square bonding all r_{jx} such that $r_{jx} \in R_j$
  - P_i : a circle
  - (P_i, R_j) : directed edge from P_i to R_j square
  - (r_{jk}, P_i) : directed edge from the dot r_{jk} to the circle
- Operations:
  - Resource Request : Add edge(s) (P_i, R_j)
  - Resource Allocation :Instantaneously transform request-edges into assignment-edges
  - Resource Release: Remove assignment-edges.

Example for Resource Allocation Graph

- In a Resource Allocation Graph…
  - No Cycle => No Deadlock
  - Cycle => Possible Deadlock
    - If 1 instance for each Rj, then

      Cycle => Deadlock

Cycle => Deadlock

(1instance of a resource type)

Cycle ≠> Deadlock

(multiple instances in a resource type)

Cycle(s) => Deadlock

(multiple instances in a resource type)

Handling Deadlocks

Three Basic Approaches

1. Scheme/Protocol such that deadlocks cannot occur (PREVENTION)
2. Schemes such that deadlocks are avoided by checking just before resource allocation (AVOIDANCE)
3. Allow deadlocks to occur but recover when they do (DETECTION & RECOVERY)

- Deadlock Prevention :

  Protocol such that 1 or more of the 4 conditions for deadlock is made impossible

  - Mutual exclusion : No go.
  - Hold & wait :
    - Pre-allocation of all resources
    - Release all & place new request
    - ◊ Utilization? ◊ Starvation?

Handling Deadlocks

- No preemption :
  - Preempt all resources if request cannot be satisfied & restart (similar to above)
  - Check status of holding process.

    If waiting, then

    Preempt resources from it

    Else wait.

    (Restart when old + new available)
  - Effects of preemption? (What can be preempted ?)

Handling Deadlocks

CPU ? Memory ? …

- Circular wait :
  - Linear order on resources (or types?)
  - Requests only in increasing order
- OR

Before requesting R_j,

Release all R_x such that $O(R_x) > O(R_j)$

- (Proof? How to order?)

Discuss: Cost/utilization (measure?)

Deadlock Avoidance

- Need info for future!
- Model :

  Each process declares

  (1) resource type and

  (2) # of instances of each   that may be needed before starting
- Allocation State :
  - Available resources type X # instances. (R) $\rightarrow$ (#)
  - Allocated resources type X # instances for each process

    (P, R) $\rightarrow$ (#)
  - Future demands (maximum) (P, R) $\rightarrow$ (#)
- Strategy :   Upon request, before allocation, verify that circular wait cannot arise (i.e. only safe state transitions)

Deadlock Avoidance

- Safe state :

  One in which there exists a safe sequence < P_1, P_2, …, P_n >

- Safe sequence :

  For each P_i in the sequence, the demand of P_i can be satisfied with the

  (i) available resources and

  (ii) those assigned to all P_j, such that j < I

- Observe …

  - Deadlock => system in unsafe state
  - Unsafe state ≠>deadlock;

    => Possibility of deadlock

  - Protocol =>waiting; even for available resources (utilization?)

Deadlock Avoidance Algorithms

- Banker's Algorithm : (Dijkstra & Habermann; many types, many instances)
- Terminology :
  - N : # of processes
  - M : # of resource type
  - available [M] : available (j) is the unallocated # of instances of resource type j
  - max[N][M] : max(i)(j) is the maximum demand of process P_i for resource Rj
  - allocation[N][M] : allocation (i)(j) is # of instances of type R_j allocated to process P_i [allocation(i) refers to an array of size M, giving the resource allocation for process P_i]

Banker’s Algorithm

- need[N][M] : max(i)(j) – allocation(i)(j) this is the balance amount of resources needed by processes. [need(i) refers to size M, giving the balance amount of resources of all types for process P_i]

- request[M] : the array for the resource request of the current process (request(j) instances of type j

- <u>For arrays X[M], Y[M]</u>
- $X \leq Y \Rightarrow X(i) \leq Y(i)$,for all values of i
- $X < Y \Rightarrow X(i) < Y(i)$, for all values of i

Banker's Algorithm

Algorithm (Upon resource-request by a process Pi in request; and for all pending request upon resource release) {requires up to MN^2 operations}

If request > need (i)

Error

If request > available

Wait

(Now Pretend Allocation!)

available $\leftarrow$ available - request

allocation (i) $\leftarrow$ allocation(i) + request

need (i) $\leftarrow$ need(i) – request

If safe( )

Return

Banker's Algorithm

Else

Deallocate resources

Wait

function safe( ) : boolean

int work[N]

boolean finish[N]

int I

1. work ← available

finish(i) ← false for all i

2. find i such that

finish(i) = false and

need(i) ≤ work

(Discuss: can I take any such i ?)

if no such i

goto 4

3. work ← work + allocation(i)

   finish(i) ← true

   goto 2

4. if finish(i) = true for all I

   safe ← true

   else

   safe ← false

Example

1. Let us look at a single resource-type example …Banker has Rs. 50,000

	Max loan demands		Disbursed
A	Rs. 25,000	A	Rs. 20,000
B	Rs. 40,000	B	Rs. 10,000
C	Rs. 20,000	C	Rs. 10,000