Operating System: Deadlock

Alex Chi

Update: April 2, 2020

Contents

1 Deadlock Characteristics

- mutual exclusion
- hold and wait
- no preemption
- circular wait

2 System Model

- process $\mathrm{P}_1, \ldots \ \mathrm{P}_\mathrm{n}$
- resource types $\mathrm{R}_1, \ldots, \mathrm{R}_\mathrm{m}$
- each resource R_i has W_i instances
- each process utilize a resource
  - request
  - use
  - release

3 Resource Allocation Graph

- a set of vertices V consisting of P and R.
- request / assignment edge ($\mathrm{P} \rightarrow \mathrm{R}$ / $\mathrm{R} \rightarrow \mathrm{P}$)
- a graph with cycle is not necessarily a deadlock
- deadlock condition: do topology sort, there's no idle instance of resource

4 Handle Deadlock

- ensure system never get into deadlock
  - prevention
  - avoidance
- allow system to enter deadlock and recover

    - detection
    - recovery

4.1 Deadlock Prevention

- mutual exclusion
- hold and wait
    - require process allocate all resource at the beginning
    - allow process request when released all resources
- no preemption
- circular wait

4.2 Deadlock Avoidance

- store some prior information
- safe state (using graph)
- ensure system won't get into unsafe state

4.2.1 Safe and Unsafe States

- example
    - available: 3
    - maximum needs, holds, needs
    - P_0 10 5 5
    - P_1 4 2 2
    - P_2 9 2 7
    - P_1 - P_0 - P_2

4.2.2 Avoidance Algorithms

- single instance of a resource type (use graph)
- multiple instances (banker's algorithm)

4.2.3 Resource-Allocation-Graph Algorithm

- claim edge $P_i \rightarrow R_j$ (process will claim the resource)

- only grant request if there won't be cycle

4.2.4 Banker's Algorithm

- n = number of process
- m = number of resources
- Available[m]: R_j available
- Max[n, m]: Max[i, j] = k, P_i will request at most k instances of R_j
- Allocation[n, m]: Allocation[i, j] = k, P_i is allocated k instances of R_j
- Need[n, m]: Need[i, j] = k, P_i may need k more instances of R_j, = Max[i, j] - Allocation[i, j]

4.2.5 Implementation

- Work = Available, Finish[0..n-1] = False
- Find any Finish[i] = False, $Need_i \leq Work$. If not, go to step 4
- Work += Allocation, Finish[i] = True, go to step 2
- Finish[i] == true for all i, safe
- we should know task information beforehand (prior information)
- the assumption is too strong

4.2.6 Resource-Request Algorithm

1. $Request_i \leq Need_i$, step 2, otherwise raise
2. $Request_i \leq Available$, step 3, otherwise raise
3. Pretend to allocate requested resources

4.3 Deadlock Detection

4.3.1 Single Instance of Each Resource Type

- maintain wait-for graph
- periodically check cycle

4.3.2 Data

- Available[m]

- Allocation[n, m]
- Request[n, m]

4.3.3 Algorithm

- similar to previous Banker's algorithm, diff:
- step 1: if $\mathrm{Allocation}_i \neq 0$, finish it
- step 2: find $\mathrm{Request}_i \leq \mathrm{Work}$

4.3.4 Usage

- how often?
- how many process will be involved?

4.3.5 Recovery

- terminate process
  - abort all deadlocked process
  - abort one at a time
    - priority
    - how long a process has computed
    - resource used
    - resource required
    - how many process will be terminated
    - interactive / batch?
- preempt resource