Operating Systems: Deadlocks

Shankar

April 14, 2021

Outline

What are Deadlocks

- Deadlock: Set of processes $P_1, \cdots, P_N$ deadlocked iff
  - every P_i is blocked, and
  - every P_i is waiting for an event doable only by some P_j
    // event: release, signal, V, interrupt enable, ...
- Deadlock freedom: desired property of multi-threaded programs
  - ensuring this is hard for a general multi-threaded program
  - but easier for a resource-manager system examined next
- Aside: Livelock is deadlock without blocking
  - processes are in fruitless loops
  - harder to detect (unless loops are very localized)
  - deadlock can be livelock at a lower (spin-lock) level

Resource Manager System

- System = resource manager + user processes
  - processes: request resources, get them, release them
  - RES: set of all resources, initially held by manager
  - $alloc(p)$: resources currently held by (user process) p
  - $avail$: resources currently held by manager
- Function $req(p, res)$: request by p for resources res
  - call only if $res + alloc(p) \subseteq RES$
  - blocking call
  - p gets res at return; happens only if $res \subseteq avail$
- Function $rel(p, res)$: release by p of res
  - call only if $res \subseteq alloc(p)$
  - nonblocking
- System can deadlock without further constraints
- 3 approaches: prevention, avoidance, detection/recovery

Outline

Deadlock Prevention Approach

- Impose further constraints on *req* calls to preclude deadlock
  - no further constraints on *req* returns
- Step 1: identify a necessary condition for deadlock, eg:
  - resource that is non-shareable and non-preemptable
  - process holds a resource and requests more resources
  - cycle of processes: each requesting a resource held by the next
- Step 2: constrain *req* calls to preclude a necessary condition
- Henceforth assume non-shareable/non-preemptable resources

- Examples of deadlock prevention rules
  - $req(p, res)$ can be called only when $alloc(p)$ is empty
  - Impose a total ordering on all resources in RES
    $req(p, res)$ can be called only when $res > max(alloc(p))$

Outline

Deadlock Avoidance Approach

- Deadlock avoidance:
  - impose further constraints on *req* returns to preclude deadlock
    - so $req(p, res)$ return may wait even if $res \subseteq avail$
  - may also involve weak constraints on *req* calls
    - eg, limit on total resources that a process can hold
  - can allow more parallelism than deadlock prevention
  - burden is on manager (unlike deadlock prevention)

- Classical deadlock avoidance solution uses the "Banker's algorithm"

Deadlock Avoidance Solution – 1

- Resources: organized into types $1, \cdots, M$
- $Tot = [Tot_1, \cdots, Tot_M]$ // total # of each resource type

- Processes: $1, \cdots, N$
- Max_i: $[Max_{i,1}, \cdots Max_{i,M}]$ // max total need of process i

- Variables
  - $alloc_i$: $[alloc_{i,1}, \cdots, alloc_{i,M}]$ // resources held by process i
  - $avail$: $[avail_1, \cdots, avail_M]$ // resources held by manager
  - req_i: $[req_{i,1}, \cdots, req_{i,M}]$ // process i's ongoing request
  - $need_i$: $Max_i - alloc_i$ // process i's max possible request

- Assumption: If a process i always gets the resources it asks for, it eventually releases all its resources
  - So if $need_i \leq avail$ and the manager grants only requests of i, then it eventually gets $alloc_i$ back
- A state is safe iff it has a safe sequence
- A safe sequence is a permutation $i_1, \cdots, i_N$ of process ids s.t.
  - $need_{i_1} \leq avail$
  - $need_{i_2} \leq avail + alloc_{i_1}$
  - ...
  - $need_{i_N} \leq avail + alloc_{i_1} + \cdots + alloc_{i_{N-1}}$
- A safe state is not deadlocked and cannot lead to a deadlock

- Banker's Algorithm: determines whether or not a state is safe
  - Variables

    $xavail \leftarrow avail$ // temporary *avail*

    $done[i] \leftarrow$ false, for $i = 1, \cdots, N$ // true iff i accounted for
  - While (there is an i s.t.

    $done[i] =$ false and $need_i \leq xavail$)

    $xavail \leftarrow xavail + alloc_i$

    $done[i] \leftarrow true$
  - Safe iff $done[i] = true$ for every i

- Return $req(p, res)$ only if the resulting state would be safe, ie, apply Banker's algorithm to the current state with
  - *avail* decreased by *res*
  - $alloc_i$ increased by *res*

Banker's Algorithm Example

- 5 processes, 3 resource types
- *Tot*: [10 5 7]
- State

	Max	alloc
P1	7 5 3	0 1 0
P2	3 2 2	2 0 0
P3	9 0 2	3 0 2
P4	2 2 2	2 1 1
P5	4 3 3	0 0 2

- Safe?

Banker's Algorithm Example

- 5 processes, 3 resource types
- *Tot*: [10 5 7]
- State

	Max	alloc	need
P1	7 5 3	0 1 0	7 4 3
P2	3 2 2	2 0 0	1 2 2
P3	9 0 2	3 0 2	6 0 0
P4	2 2 2	2 1 1	0 1 1
P5	4 3 3	0 0 2	4 3 1

- Safe?

Banker's Algorithm Example

- 5 processes, 3 resource types
- *Tot*: [10 5 7]
- State

	Max	alloc	need	avail
P1	7 5 3	0 1 0	7 4 3	3 3 2
P2	3 2 2	2 0 0	1 2 2	
P3	9 0 2	3 0 2	6 0 0	
P4	2 2 2	2 1 1	0 1 1	
P5	4 3 3	0 0 2	4 3 1	

- Safe?

Banker's Algorithm Example

- 5 processes, 3 resource types
- *Tot*: [10 5 7]
- State

	Max	alloc	need
P1	7 5 3	0 1 0	7 4 3
P2	3 2 2	2 0 0	1 2 2
P3	9 0 2	3 0 2	6 0 0
P4	2 2 2	2 1 1	0 1 1
P5	4 3 3	0 0 2	4 3 1

done	avail
	3 3 2
P2	5 3 2
P4	7 4 3
P5	7 4 5
P1	7 5 5
P3	10 5 7

- Safe? Yes. Safe sequence: P2, P4, P1, P3

Outline

Deadlock Detection and Recovery

- Do not constrain *req* calls or returns
- Instead periodically check for deadlock.
  If yes, choose a process i and forceably release $alloc_i$

- Deadlock detection algorithm for M resource types
  // variation of Baker's algorithm
  - Variables
    $xavail \leftarrow avail$ // temporary *avail*
    $done[i] \leftarrow$ false, for $i = 1, \cdots, N$ // true iff i accounted for
  - While (there is an i s.t.
    $done[i] =$ false and $req_i \leq xavail$)
    $xavail \leftarrow xavail + alloc_i$
    $done[i] \leftarrow true$
  - If $done[i] =$ true for every i, then no deadlock.
    Otherwise, processes whose *done* is false are in a deadlock.

Outline

What happens in real-life

- Resources are increasingly shareable
  - disks (vs tapes)
  - demand-paging (vs entire process space in physical memory)
  - virtualization of everything
- Hence livelock (or thrashing) is more common than deadlock
- Hence deadlock prevention/avoidance/detection is rarely used
- Instead, if system “appears” to be in deadlock (or livelock), kill and/or restart processes or entire system