운영체제의 기초: Deadlock

2023년 4월 20일

홍 성 수
sshong@redwood.snu.ac.kr
SNU RTOSLab 지도교수
서울대학교 전기정보공학부 교수

Seoul National University RTOS Lab

Agenda

I. Introduction
II. Deadlock Prevention
III. Deadlock Detection and Recovery

I. Introduction

Overview (1)

- Deadlock is one area where there is a strong theory but it is almost completely ignored in practice
  - Reason
    - Solutions are expensive and/or require predicting the future

- Definition of deadlock
  - A situation where each of a collection of processes is waiting for something from other processes in the collection
  - Since all are waiting, none can provide any of the things being waited for

Overview (2)

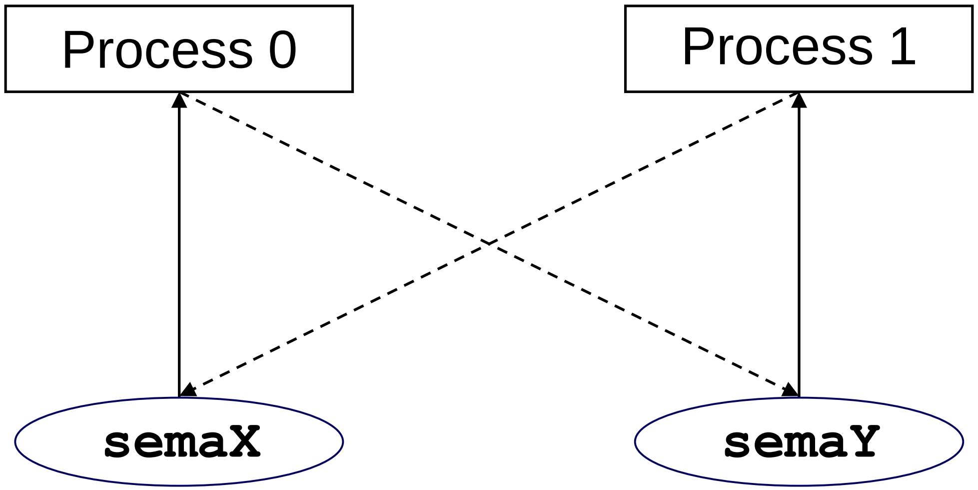
- Deadlock example with semaphores

Process 0:

```
P(semaX);
P(semaY);
```

Process 1:

```
P(semaY);
P(semaX);
```



Overview (3)

- The previous example was relatively simple-minded
  - Things may be much more complicated
    - In general, don't know in advance how many resources a process will need. Only if we could predict the future ...
    - Deadlock can occur over separate resources, as in the semaphore example, or over pieces of a single resource, as in memory, or even over totally separate classes of resources (tape drives and memory)
    - Deadlock can occur over anything involving, for example, messages in a pipe system
    - Hard for OS to control

Deadlock Handling (1)

- Solutions to deadlock problem fall into two general categories
  1. Prevention
     - Organize the system so that it is impossible for deadlock ever to occur
     - May lead to less efficient resource utilization in order to guarantee no deadlocks
  2. Detection and recovery
     - Determine when the system is deadlocked, and then take drastic action
     - Requires termination of one or more processes in order to release their resources

Deadlock Handling (2)

- Four necessary conditions for deadlock
  - Mutual exclusion (limited access)
    - Resources cannot be shared
  - No preemption
    - Once given, a resource cannot be taken away
  - Hold and wait (multiple independent requests)
    - Processes don’t ask for resources all at once
  - Circular wait
    - There is a circularity in the graph of who has what and who wants what

II. Deadlock Prevention

Deadlock Prevention (1)

- Avoiding one of four necessary conditions
  - No mutual exclusion
    - Don't allow exclusive access
    - This is probably not reasonable for many applications
  - No preemption
    - Allow preemption (E.g., Preempt your disk space?)

Deadlock Prevention (2)

- Avoiding one of four necessary conditions
  - No hold and wait
    - Make process ask for everything at once
    - Either get them all or wait for them all
    - Must be able to wait on many things without locking anything
    - Painful for process
      - May be difficult to predict, so must make very wasteful use of resources
      - Tricky to implement
      - This requires the process to predict the future

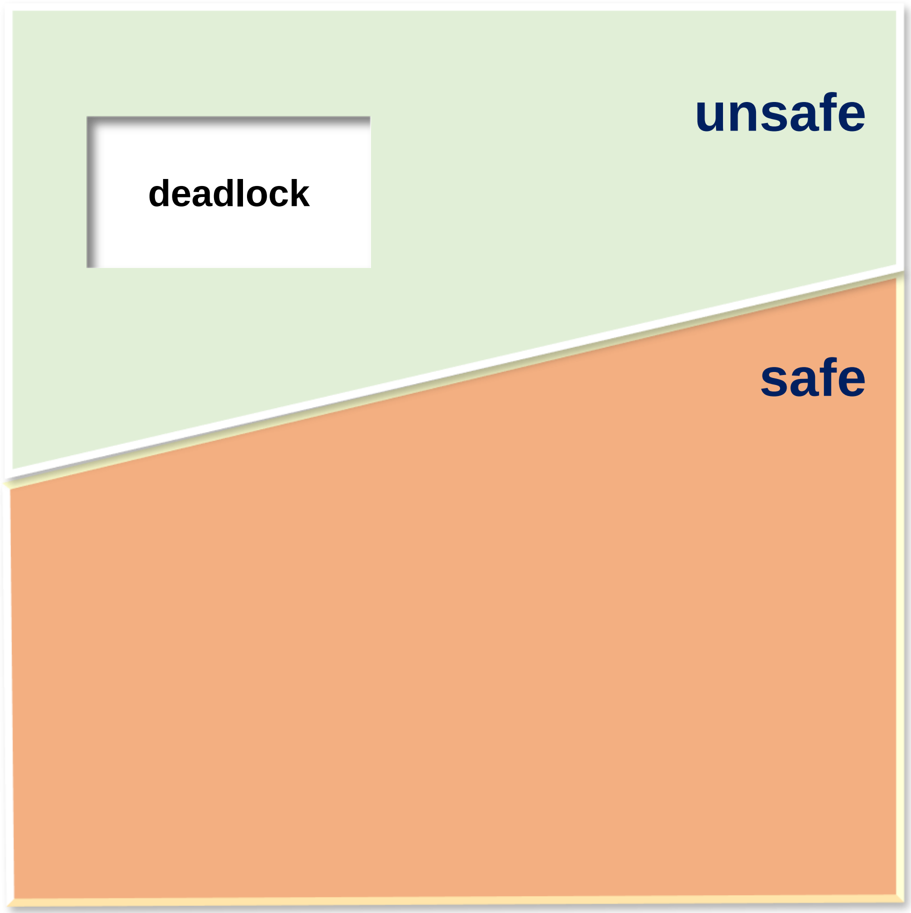
Deadlock Prevention (3)

- Avoiding one of four necessary conditions
  - No circular waiting
    - Create enough resources so that there's always plenty for all
    - Don't allow waiting
      - This punts the problem back to the user (E.g., Phone company)
    - Make ordered or hierarchical requests
      - E.g., ask for all S's, then all T's etc.
    - All processes must follow the same ordering scheme
    - Of course, for this you have to know in advance what is needed

Banker's Algorithm (1)

- Safe state
  - The system can allocate resources to each process up to its maximum in some order and still avoid a deadlock
  - A safe sequence must exist from a safe state
- Unsafe state
  - May lead to a deadlock

Banker's Algorithm (2)



Banker's Algorithm (3)

- Example: A system with 12 magnetic drives

Process	Max Needs	Current Allocations
P_0	10	5
P_1	4	2
P_2	9	2

- Detecting safe/unsafe state
  - Safe sequence: <P_1,P_0,P_2>
  - Transition to an unsafe one: < P'_2>
    - By making an additional request

Process	Max Needs	New Allocations
P'_2	9	3

Banker's Algorithm (4)

- Key idea
  - A new process must declare the maximum resource needs
  - When a process requests resources, the algorithm checks if the allocation will leave the system in a safe state
  - Grant the resources, if so
  - Otherwise, have it wait until some other process releases enough resources

Banker's Algorithm (5)

- Notations
  - Available[1:m]
    - The number of available resources of each type
  - Max[1:n,1:m]
    - The maximum demand of each process
  - Allocation[1:n,1:m]
    - The number of resources of each type currently allocated to each process
  - Need[1:n,1:m]
    - The remaining resource need of each process
    - Max[i, j] = Allocation[i, j] + Need[i, j]

Banker's Algorithm (6)

- Notations
  - For two vectors X and Y:
    - $X \le Y$ iff $\forall i : 1 \le i \le n : X[i] \le Y[i]$
    - $X < Y$ iff $X \le Y$ and $X \ne Y$

Banker's Algorithm (7)

- Safety check

  Step 0: Work[1:m] and Finish[1:n] are two vectors

  Step 1: Work = Available and Finish[i] = false for i = 1,2,...,n

  Step 2: Find an i such that both

  Finish[i] = false and Need[i] $\leq$ Work

  If no such i exists, go to Step 4

  Step 3: Work = Work + Allocation[i]

  Finish[i] = true

  Go to Step 2

  Step 4: If Finish[i] = true for all i, then the system is in a safe state

Banker's Algorithm (8)

- Handling resource request for process P_i

Step 0: Request[1:n, 1:m] is the resource request of each process

Step 1: If Request[i] ≤ Need[i], go to step 2

Otherwise, raise an error condition

Step 2: If Request[i] ≤ Available, go to step 3

Otherwise, P_i must wait for the resource

Banker's Algorithm (9)

- Handling resource request for process P_i

Step 3: Grant the resource request as below

Available = Available – Request[i];

Allocation[I] = Allocation[i] + Request[i];

Need[i] = Need[i] – Request[i];

Step 4: If the resulting resource allocation is safe, the transaction is completed and P_i if allocated; Otherwise, P_i must wait and old resource allocation state is restored

Banker's Algorithm (10)

- Example
  - A state snapshot
    - Safe sequence <P_1,P_3,P_4,P_2,P_0>

Processes	Allocations	Max Needs	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

  - NewRequest[1] = (1,0,2):
    - Determine if the new state is safe

III. Deadlock Detection and Recovery

Deadlock Detection (1)

- Limitations in deadlock handling mechanisms
  - Prevention of deadlock is expensive and/or inefficient
  - Detection is also expensive and recovery is seldom possible
  - (What if process has things in a weird state?)
    - Particularly, in a mission critical system such as a vehicle

Deadlock Detection (2)

- Detection of deadlock could be complicated
  - Single instance of each resource type
    - Existence of cycle is a necessary and sufficient condition for a deadlock
  - Multiple instances of a resource type
    - Use a deadlock detection algorithm similar to the banker's algorithm

Deadlock Detection Algorithm (1)

Step 0: Work[1:m] and Finish[1:n] are two vectors

Step 1: Work = Available

For i = 1,2,...,n, Finish[i] = $\begin{cases} \text{false, if Allocation[i]} \neq 0 \\ \text{true, otherwise} \end{cases}$

Step 2: Find an i such that both

Finish[i] = false

Request[i] ≤ Work

If no such exists, go to Step 4

Step 3: Work = Work + Allocation[i]; Finish[i] = true

Go to Step 2

Step 4: If Finish[i] = false for some i, then the system is in a deadlock state (Such i (i.e., P_i) is a deadlocked process)

Deadlock Detection Algorithm (2)

- Example: <P_0, P_2, P_3, P_1, P_4> results in Finish[i]=true

Processes	Allocations	Requests	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

  - What will happen if P_2 makes an additional request for a instance of type C?

Deadlock Detection Algorithm (3)

- Example: Deadlock involving P_1, P_2, P_3, P_4

Processes	Allocations	Requests	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Deadlock Recovery

- Process termination
  - Abort all deadlocked processes
  - Abort processes one at a time until the deadlock cycle is eliminated
- Resource preemption
  - Select a victim
  - Rollback
  - Starvation