CSE 380
Computer Operating Systems

Instructor: Insup Lee and Dianna Xu

University of Pennsylvania, Fall 2003

Lecture Note: Deadlocks

Resource Allocation

- Examples of computer resources
  - printers
  - tape drives
  - semaphores
- Processes need access to resources in specific order
- Undesirable scenario:
  - Suppose a process holds resource A and requests resource B
  - At the same time another process holds B and requests A
  - Both are blocked and remain so, waiting for each other
- Can occur in a multiprogramming environment and also in a distributed system

Deadlock due to Semaphores

```
semaphore:   mutex1 = 1     /* protects resource 1 */
             mutex2 = 1    /* protects resource 2 */
```

Process A code:

```
{
   /* initial compute */
  down (mutex1)
  down (mutex2)
 /* use both resources */
  up (mutex2)
  up (mutex1)
}
```

Process B code:

```
{
   /* initial compute */
  down (mutex2)
  down (mutex1)
 /* use both resources */
  up (mutex2)
  up (mutex1)
}
```

Deadlocks

- System has a set of processes, and a set of resources; each resource can have multiple instances
- Interesting events are:
  - Request for resources
    - A process can request multiple resources in one shot
    - A process can request resources at different times during its execution
  - Granting of a request
    - This is possible only if there are enough free resources
    - A process stays blocked, and waits, until its request is granted
  - Release of resources
    - A process can release some of the resources it is currently holding
- Deadlock situation: There is a set of blocked processes such that there is no way to satisfy their requests (even if the currently unblocked processes release all the resources they currently hold)

Four Conditions for Deadlock

1. Mutual exclusion condition
   - each resource assigned to exactly one process or is available
2. Hold and wait condition
   - process holding resources can request additional resources
3. No preemption condition
   - previously granted resources cannot be taken away
4. Circular wait condition
   - must be a circular chain of 2 or more processes
   - each is waiting for resource held by next member of the chain

Dealing with Deadlocks

1. just ignore the problem altogether
   - The Ostrich Approach
2. detection and recovery
   - Resource Allocation Graphs
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating one of the four necessary conditions

Deadlock Detection

- Goal: How can OS detect when there is a deadlock?
- OS should keep track of
  - Current resource allocation (who has what)
  - Current pending requests (who is waiting for what)
- This info is enough to check if there is a current deadlock (see next few slides)
- What can OS do once a deadlock is detected?
  - Kill a low priority process
  - Revoke currently allocated resources (if that's possible)
  - Inform the users or the administrator

Detecting Deadlocks 1

- Suppose there is only one instance of each resource
- Example 1: Is this a deadlock?
  - P1 has R2 and R3, and is requesting R1
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Example 2: Is this a deadlock?
  - P1 has R2, and is requesting R1 and R3
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Solution: Build a graph , called Resource Allocation Graph (RAG)
  - There is a node for every process and a node for every resource
  - If process P currently has resource R, then put an edge from R to P
  - If process P is requesting resource R, then put an edge from P to R
- There is a deadlock if and only if RAG has a cycle

Detecting Deadlocks 2

- How to detect deadlocks when there are multiple instances of resources
- Example: Is this a deadlock?
  - Suppose there are 2 instances of A and 3 of B
  - Process P currently has 1 instance of A, and is requesting 1 instance of A and 3 instances of B
  - Process Q currently has 1 instance of B, and is requesting 1 instance of A and 1 instance of B

Multiple Resource Case

- Suppose there are n process P1, …. Pn and m resources R1 …. Rm
- To detect deadlocks, we can maintain the following data structures
  - Current allocation matrix C: C[i,j] is the number of instances of resource Rj currently held by process Pi
  - Current request matrix R: R[i,j] is the number of instances of resource Rj currently being requested by process Pi
  - Availability vector A: A[j] is the number of instances of resources Rj currently free.
- Goal of the detection algorithm is to check if there is any sequence in which all current requests can be met
  - Note: If a process Pi's request can be met, then Pi can potentially run to completion, and release all the resources it currently holds. So for detection purpose, Pi's current allocation can be added to A

Example

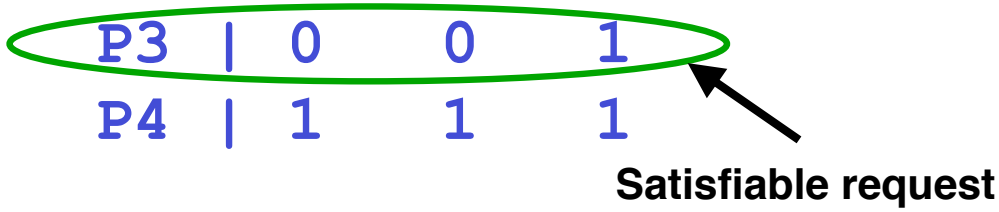
L = {} /* List of processes that can be unblocked */

Allocation Matrix C

	R1	R2	R3
P1	1	1	1
P2	2	1	2
P3	1	1	0
P4	1	1	1

Request Matrix R

	R1	R2	R3
P1	3	2	1
P2	2	2	1
P3	0	0	1
P4	1	1	1



Satisfiable request

A = (0, 0, 1) /* available resources */

Request by process i can be satisfied if the row R[i] is smaller than or equal to the vector A

After first iteration

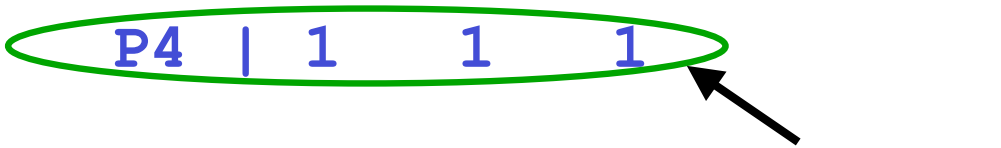
L = { P3 }

Allocation Matrix

	R1	R2	R3
P1	1	1	1
P2	2	1	2
P3			
P4	1	1	1

Request Matrix

	R1	R2	R3
P1	3	2	1
P2	2	2	1
P3			
P4	1	1	1



Satisfiable request

A = (1, 1, 1)

Note: P3's allocation has been added to A

After Second iteration

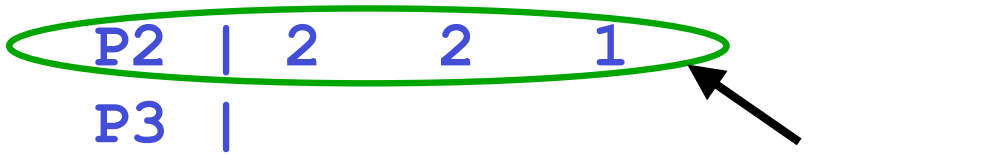
L = { P3, P4}

Allocation Matrix

	R1	R2	R3
P1	1	1	1
P2	2	1	2
P3			
P4			

Request Matrix

	R1	R2	R3
P1	3	2	1
P2	2	2	1
P3			
P4			



Satisfiable request

A = (2, 2, 2).

Deadlock Detection Algorithm

```
L = EmptyList; /* processes not deadlocked */
repeat
   s = length(L);
   for (i=1; i<=n; i++){

     if (!member(i,L) && R[i] <= A) {
        /* request of process i can be met */

       A = A + C[i];
      /* reclaim resources held by process i */

       insert(i,L);
     }
    }
until (s == length(L));
/* if L does not change, then done */

if (s<n) printf("Deadlock exists");
```

Note: Running time of this algorithm is $O(n^2 m)$, where m: length of a row

Dead Lock Recovery

- Preemption
  - Take away a resource temporarily from current owner
  - Frequently impossible
- Rollback
  - Checkpointing periodically to save states
  - Reset to earlier state before acquiring resource
- Killing
  - Crude but simple
  - Keep killing processes in a cycle until cycle is broken

Deadlock Prevention

- Can OS ensure that deadlocks never happen?
- There are four necessary conditions for deadlocks to occur, can any of these conditions be negated ?
- Mutual Exclusion
  - spooling
- Hold and Wait
  - processes to request all resources at once, grant is all are available, deny if any is not
- No Preemption
- Circular Wait
  - hierarchical allocation

Hierarchical Allocation
Avoiding the circular wait

- Resources are grouped into levels (i.e. prioritize resources numerically)
- A process may only request resources at levels higher than any resource currently held by that process.
- Resources may be released in any order.
- Example:
  - Resources: Directory blocks and file blocks
  - Constraint: Can't request a file block if you are holding onto a directory block

Properties

- When all requests are at the same level, this method is equivalent to one-shot allocation.
  - A process has to request all the resources in one shot
- Global numbering prevents cycles

- Resources at lower levels are blocked for longer periods, but those at higher levels are shared well.
- This method works well when the resources are semaphores.

```
semaphore S1,S2,S3
(with priorities in increasing order)
P(S1);…; P(S2);…; P(S3) allowed
P(S2);…; P(S3);…; P(S1) not allowed
```

Avoidance

- Motivation: "Is there an algorithm that can always avoid deadlock by conservatively making the right/safe choice all the time?"
- Deadlock is the result of granting a resource.
- Banker's algorithm
  - Deny potentially unsafe requests

Banker's Algorithm

- Suppose there are n process P1, …. Pn and m resources R1 …. Rm
- Suppose every process has declared in advance, its claim---the maximum number of resources it will ever need
  - Sum of claims of all processes may even exceed total number of resources
- To avoid deadlocks, OS maintains the allocation state
  - Current allocation matrix C: C[i,j] is the number of instances of resource Rj currently held by process Pi
  - Claims matrix M: M[i,j] is the maximum number of instances of resource Rj that process Pi will ever request
  - Availability vector A: A[j] is the number of instances of resources Rj currently free.
- Suppose process Pi requests certain number resources. Let Req be the request vector (Req[j] is number of requested instances of Rj)
  - Valid request if Req <= M[i]-C[i] (i.e. it should be in accordance with claim)
  - If Req <= A, then it is possible for OS to grant the request
  - Avoidance strategy: Deny the request if the resulting state will be unsafe

Safe state

- An allocation state is **safe** if there is an ordering of processes, a **safe sequence**, such that:
  - the first process can finish for sure: there are enough unallocated resources to satisfy all of its claim.
  - If the first process releases its currently held resources, the second process can finish for sure (even if it asks all its claim), and so on.
- The state is safe because OS can definitely avoid deadlock by blocking any new processes, or any new requests, until all the current processes have finished in the safe order.

Example

(One resource class only)

process	holding	max claims
A	4	6
B	4	11
C	2	7

unallocated: 2

safe sequence: A,C,B

If C should have a claim of 9 instead of 7, there is no safe sequence.

Example

process	holding	max claims
A	4	6
B	4	11
C	2	9

unallocated: 2

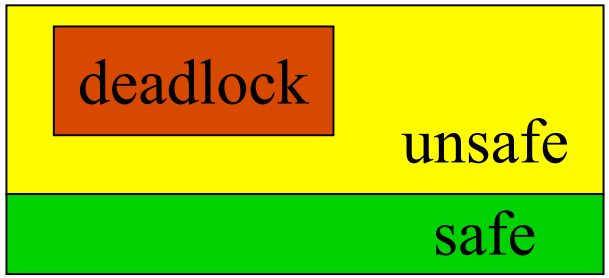
deadlock-free sequence: A,C,B

if C makes only 6 requests

- However, this sequence is not safe: if C should have 7 instead of 6 requests, deadlock exists.

Banker's algorithm

- Maintain claims M, current allocation C and current availability A
- Suppose process Pi requests Req such that Req <= A and Req+C[i] <= M[i]
- Consider the state resulting from granting this request (i.e. by adding Req to C[i] and subtracting Req from A). Check if the new state is a safe state. If so, grant the request, else deny it.
- It ensures that allocation state is aways safe



The Banker's Algorithm is conservative:
it cautiously avoids entering an unsafe state even if this unsafe state has no deadlock.

Checking Safety

- How do we check if an allocation state is safe?
  - Current allocation matrix C
  - Maximum claims matrix M
  - Availability vector A
- Same as running the deadlock detection algorithm assuming that every process has requested maximum possible resources
  - Choose Requests Matrix R to be M – C, and see if the state is deadlocked (is there an order in which all of these requests can be satisfied).

Example

	Allocation			Claims			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

this is a safe state:

safe sequence <P1, P3, P4, P2, P0>

Suppose that P1 requests (1,0,2). To decide whether or not to grant this request, add this request to P1's allocation and subtract it from A.

Example

	Allocation			Claims			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

This is still safe:

safe seq <P1, P3, P4, P0, P2>

In this new state,

P4 requests (3,3,0) not enough available resources

P0 requests (0,2,0) let's check resulting state

Example

	Allocation			Claims			Available		
	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	2	1	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

This is unsafe state (why?)

So P0's request will be denied

Starvation

- When multiple processes requests the same resource, allocation policy needs to be established
- Any policy that attempts to optimize may potentially lead to starvation
- FCFS is fair and commonly used

Dead Locks Happen

- Dead lock avoidance and prevention is often impossible in real systems
- Thorough detection of all possible scenarios too expensive
- All operating systems have potential dead locks
- Engineering philosophy:

  The price of infrequent crashes in exchange for performance and user convenience is worth it