CS 355  
Operating Systems  
Deadlocks

Resources
- Processes need access to resources in specific order
- Deadlocks occur when...
  - processes are granted exclusive access to devices
  - we refer to these devices generally as resources
- Preemptable resources
  - can be taken away from a process with no ill effects
- Nonpreemptable resources
  - will cause the process to fail if taken away

Resources (2)
- Sequence of events required to use a resource
  1. request the resource
  2. use the resource
  3. release the resource
- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code

Introduction to Deadlocks
- Formal definition:
  A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- Usually the event is release of a currently held resource
- None of the processes can...
  - run
  - release resources
  - be awakened

Deadlock due to Semaphores

```c
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)
}
```

Deadlock
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 is either assigned to a 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

Deadlock Modeling

- Modeled with directed graphs

![Directed Graph Example]

Deadlock Modeling

- Resource Allocation Graphs

![Resource Allocation Graphs Example]

How deadlock occurs

Dealing with Deadlocks

1. Just ignore the problem altogether

2. Detection and recovery
   - Resource Allocation Graphs

3. Dynamic avoidance
   - careful resource allocation

4. Prevention
   - negating one of the four necessary conditions

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high

- UNIX and Windows takes this approach
- It is a trade off between
  - Convenience/speed
  - correctness
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
- 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

- 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

Resource Allocation Graph

- Build a 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

Detection with One Instance of Each Resource

- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

Multiple Resource Instances

- 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 P_1, ..., P_n, and m resources R_1, ..., R_m
- To detect deadlocks, we can maintain the following data structures
  - Existence vector E: E[j] is the total number of instances of resource R_j in existence
  - Current allocation matrix C: C[i,j] is the number of instances of resource R_j currently held by process P_i
  - Current request matrix R: R[i,j] is the number of instances of resource R_j currently being requested by process P_i
  - Availability vector A: A[j] is the number of instances of resource R_j 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 P_i request can be met, then P_i can potentially run to completion, and release all the resources it currently holds. So for detection purpose, P_i’s current allocation can be added to A
Data Structures for Multiple Resource Instances

Resources in existence (E₁, E₂, E₃, ..., Eₙ)

Current allocation matrix:

| C₁₁ | C₁₂ | C₁₃ | ... | C₁ₙ |
| C₂₁ | C₂₂ | C₂₃ | ... | C₂ₙ |
| ... | ... | ... | ... | ... |
| Cₙ₁ | Cₙ₂ | Cₙ₃ | ... | Cₙₙ |

Resources available (A₁, A₂, A₃, ..., Aₙ)

Request matrix:

| R₁₁ | R₁₂ | R₁₃ | ... | R₁ₙ |
| R₂₁ | R₂₂ | R₂₃ | ... | R₂ₙ |
| ... | ... | ... | ... | ... |
| Rₙ₁ | Rₙ₂ | Rₙ₃ | ... | Rₙₙ |

Flow n is current allocation to process n

\[ \sum_{j=1}^{n} C_{ij} + A_j = E_j \]

Example

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

Allocation Matrix | Request Matrix
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>R₂</td>
</tr>
<tr>
<td>P₁</td>
<td>1</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
</tr>
<tr>
<td>P₃</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>1</td>
</tr>
</tbody>
</table>

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 | Request Matrix
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>R₂</td>
</tr>
<tr>
<td>P₁</td>
<td>1</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
</tr>
<tr>
<td>P₃</td>
<td></td>
</tr>
<tr>
<td>P₄</td>
<td>1</td>
</tr>
</tbody>
</table>

A = (1, 1, 1)

Note: P₃’s allocation has been added to A

After Second Iteration

L = { P₃, P₄ }

Allocation Matrix | Request Matrix
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>R₂</td>
</tr>
<tr>
<td>P₁</td>
<td>1</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
</tr>
<tr>
<td>P₃</td>
<td></td>
</tr>
<tr>
<td>P₄</td>
<td>1</td>
</tr>
</tbody>
</table>

A = (2, 2, 2).

Deadlock Detection Algorithm

L = EmptyList; // processes not deadlocked
repeat
s = length(L); // if length(L) is 0, then end
for (i=0; i<n; i++)
  if ![member(i,L) && R[i] <= A]
    A = A + C[i]; // reclaim resources held by process i
    insert(i,L);
until (s == length(L)); // if L does not change, then done
printf("Deadlock exists");

Complexity of this algorithm?

Recovery from Deadlock

- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource

- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked
Recovery from Deadlock

• Recovery through killing processes
  – crudest but simplest way to break a deadlock
  – kill one of the processes in the deadlock cycle
  – the other processes get its resources
  – choose process that can be rerun from the beginning

Deadlock Prevention

• We saw how to detect deadlocks in the last class. Can OS ensure that deadlocks never happen?
• There are four necessary conditions for deadlocks to occur, can these conditions be negated?
  – Hierarchical allocation strategy
• Banker’s algorithm for avoidance
  – Deny potentially unsafe requests

Attacking the Mutual Exclusion

• Some devices (such as printer) can be spooled
  – only the printer daemon uses printer resource
  – thus deadlock for printer eliminated

• Not all devices can be spooled
• Principle:
  – avoid assigning resource when not absolutely necessary
  – as few processes as possible actually claim the resource

Attacking the Hold and Wait

• Require processes to request resources before starting
  – a process never has to wait for what it needs

• Problems
  – may not know required resources at start of run
  – also ties up resources other processes could be using

• Variation:
  – process must first give up all current resources
  – then request all needed at once

Attacking the No Preemption

• This is not a viable option
• Consider a process given the printer
  – halfway through its job
  – now forcibly take away printer
  – !??

Hierarchical Allocation
Avoiding the circular wait

• Resources are grouped into levels (i.e. prioritize resources)

• 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:
  – Can first request a scanner then a printer, but can not request a scanner if you are holding onto a printer
Attacking the Circular Wait

- Ordered resources and DAG

Properties
- This ensures that DAG never have cycles. Why?
- 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
- Resources at lower levels are blocked for longer periods, but those at higher levels are shared well.

Did We Solve Anything?

<table>
<thead>
<tr>
<th>Condition</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual exclusion</td>
<td>Spool everything</td>
</tr>
<tr>
<td>Hold and wait</td>
<td>Request all resources initially</td>
</tr>
<tr>
<td>No preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular wait</td>
<td>Order resources numerically</td>
</tr>
</tbody>
</table>

Summary of approaches to deadlock prevention

Avoidance
- Motivation: “Is there an algorithm that can always avoid deadlock by conservatively making the right/safe choice all the time?”
- Answer is a qualified “yes, if we have certain information in advance”
- Main ideas are based on the concept of safe states.

Resource Trajectories

Safe state
- An allocation state is safe if there is an ordering of processes, a safe sequence, such that:
  - a 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 next 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.
Banker’s Algorithm

- Suppose there are \( n \) processes \( P_1, \ldots, P_n \) and \( m \) resources \( R_1, \ldots, R_m \).
- Suppose every process has declared in advance, its **claim** --- the maximum number of resources it will ever need.

- To avoid deadlocks, OS maintains the allocation state:
  - Current allocation matrix \( C \): \( C[i,j] \) is the number of instances of resource \( R_j \) currently held by process \( P_i \).
  - Claims matrix \( M \): \( M[i,j] \) is the maximum number of instances of resource \( R_j \) that process \( P_i \) will ever request.
  - Availability vector \( A[j] \) is the number of instances of resources \( R_j \) currently free.

**Banker’s Algorithm for Multiple Resources**

- Suppose process \( P_i \) requests \( R \) such that \( R \leq A \) and \( R + C[i] \leq M[i] \).
- Consider the state resulting from granting this request (i.e. by adding \( R \) to \( C[i] \) and subtracting \( R \) 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 always safe.

**Example**

(One resource class only)

<table>
<thead>
<tr>
<th>Process</th>
<th>Holding</th>
<th>Max Claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

unallocated: 2
safe sequence: A, C, B

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

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

unallocated: 2

deadlock-free sequence: A, C, B

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

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 when 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

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Claims</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 0 1 0</td>
<td>7 5 3</td>
<td>3 3 2</td>
</tr>
<tr>
<td>P1 2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>P2 3 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>P3 2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P4 0 0 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Claims</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 0 1 0</td>
<td>7 5 3</td>
<td>2 3 0</td>
</tr>
<tr>
<td>P1 3 0 2</td>
<td>3 2 2</td>
<td>3 2 2</td>
</tr>
<tr>
<td>P2 3 0 2</td>
<td>9 0 2</td>
<td>9 0 2</td>
</tr>
<tr>
<td>P3 2 1 1</td>
<td>2 2 2</td>
<td>2 2 2</td>
</tr>
<tr>
<td>P4 0 0 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>

In this new state, P4 requests (3,3,0), P0 requests (0,2,0)