Deadlock (CSE 422S)

Ken Wong
Washington University

kenw@wustl.edu
www.arl.wustl.edu/~kenw

Definitions

- **Resource**
  - Some reusable object
  - e.g., memory, I/O device, variable

- **Preemptable Resource**
  - It can be taken away from the owning process with no ill effects
  - e.g., memory

- **Nonpreemptable Resource**
  - It can not be taken away from the owning process without adversely affecting its computation
  - e.g., a write-once device that has already started writing

Deadlock Example

- **FIFO (Named Pipe) Communication**
  - A **named pipe (FIFO)** is a special file that is similar to a pipe except that it is accessed as part of the file system
    - Can be opened by multiple processes for reading or writing
    - Kernel passes data between processes without writing to file system
    - See fifo(4)
  - By default, the open call **blocks** until there is a reading process and a writing process (a **rendezvous**)

<table>
<thead>
<tr>
<th>Occurs</th>
<th>Server Process:</th>
<th>Client Process:</th>
</tr>
</thead>
<tbody>
<tr>
<td>open(&quot;reqFifo&quot;,O_RDONLY);</td>
<td>open(&quot;respFifo&quot;,O_WRONLY);</td>
<td></td>
</tr>
<tr>
<td>open(&quot;respFifo&quot;,O_RDONLY);</td>
<td>open(&quot;reqFifo&quot;,O_WRONLY);</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Two Dining Philosophers

- **Deadlock**
  - A set of processes S is deadlocked if each member of S is permanently waiting for a resource held by some other member of S

- What can be done about deadlocks???
Deadlock is possible but depends on timing.

**Deadlock**

- **Necessary Conditions**
  - **Mutual Exclusion**: Only one process at a time can use a resource
  - **Hold and Wait**: A process holding one resource is waiting to acquire additional resources held by other processes
  - **No Preemption**: No resource can be forcibly taken from a processing holding it

- **Sufficient Condition**
  - **Circular Waiting**: There exists a permanent, circular sequence of processes such that each process is waiting for a resource held by the preceding process

**Resource Allocation Graph**

- **Resource-Process Arc** (solid)
  - Resource has been requested, granted and is being held
- **Process-Resource Arc** (dashed)
  - Process is blocked waiting for resource

**Interpretation**

- No Cycle ➔ There is no deadlock
- Cycle ➔ There is deadlock

**Deadlock Example**

- Each process needs 2 resources before proceeding
- Can we avoid it?
  - This sequence avoids deadlock. Can we generalize?
- Can we avoid deadlock by reordering the resource grants?
Approaches for Handling Deadlock

- **Ignore Deadlock**
  » If infrequent enough and result is not serious

- **Deadlock Prevention**
  » Prevent one of the necessary/sufficient conditions

- **Deadlock Avoidance**
  » Allow the 3 necessary conditions
  » Dynamically make choices to avoid deadlock
    » decide based on knowledge of future requests
    » i.e., find a safe path

- **Deadlock Detection**
  » Periodically run algorithm to detect circular waiting
  » After detecting deadlock,
    » run a recovery algorithm to remove deadlock

Deadlock Prevention

- **Mutual Exclusion**
  » Not required for sharable resources, but must hold for non-sharable ones

- **Never Hold and Wait**
  » Block a process until it can acquire all of its requested resources at once

- **Allow Preemption**
  » Process must release all resources when it is denied a resource request

- **Prevent Circular Waiting**
  » Define a total ordering when allocating resource types
  » Require resources be requested in increasing order

Resource Ordering

- **Suppose:**
  » Resources $R(i), i=1:N$, are ordered
  » $R(1) \prec R(2) \prec \ldots \prec R(N)$
  » where \( \prec \) denotes "precedes"
  » Note: \( i < j \Rightarrow R(i) \prec R(j) \)

- **Simplified Example**
  » Process acquire-release resources in pairs
  » $Acq$ $R(j)$, $Acq$ $R(k)$, $Use$, $Rel$ $R(k)$, $Rel$ $R(j)$

- **Rule**
  » All requests must be made in resource order
  » This rule works even when processes can acquire more than two resources
    » Challenge: Prove this result.

Deadlock Avoidance (Banker's Alg.)

- **Processes state resource demands a priori**
  » Bank lends out fixed amount of money
  » Customer gets fixed line of credit; borrows and pays back part of loan over time

- **Safe State:**
  » there is at least one resource request sequence in which all processes can run to completion

- **Unsafe State:**
  » There is only a potential for deadlock

- **Always ensure the system is in a safe state**
  » Request: Update system state as if it is granted
  » If state is safe, grant the request; else block the process until it is safe to grant request

- **When a process gets all of its resources, it must return them in finite time**
Example (Banker's Algorithm)

- Current State
  - Quantity [Q(j)]:
    - R1 3 6
    - R2 1 1
    - R3 2 2
  - Unallocated [U(j)]:
    - R1 1 1
    - R2 2
    - R3 1
  - Max Demand [D(i,j)]:
    - R1 6
    - R2 1
    - R3 4
  - Allocation [A(i,j)]:
    - P1 1 0 0
    - P2 6 1 1
    - P3 2 1 1
    - P4 0 0 2

- P1 requests 1 more of R1 and 1 of R3 ... Grant?

Example of Banker's Algorithm

- Fundamental Question: Is this a safe state?
  - Can some process finish? Releases all of its resources

- P2 Can Complete

- An Unsafe State

- P1, P2 and P3 still need 1 unit of R1, but none left
- Only potential for deadlock . . . not certainty
  - e.g., Some process could release 1 unit of R1
  - Reach unsafe state if we grant P1's request

P2 returns resources and demand goes to 0
Example of Banker’s Algorithm

- **Safe Order:** P2, P1, P3, P4

<table>
<thead>
<tr>
<th>Quantity [Q(j)]:</th>
<th>Unallocated [U(j)]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 9</td>
<td>R1 2</td>
</tr>
<tr>
<td>R2 3</td>
<td>R2 3</td>
</tr>
<tr>
<td>R3 6</td>
<td>R3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Demand [D(i,j)]:</th>
<th>Allocation [A(i,j)]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 3 0 2</td>
<td>P1 1 0 0</td>
</tr>
<tr>
<td>P2 0 0 0</td>
<td>P2 0 0 0</td>
</tr>
<tr>
<td>P3 3 1 4</td>
<td>P3 2 0 1</td>
</tr>
<tr>
<td>P4 4 2 2</td>
<td>P4 0 0 2</td>
</tr>
</tbody>
</table>

The Banker’s Algorithm

- Process i requests R(*) resources of each type
  - Requirement: A(i,*) + R(*) ≤ D(i,*)

  - U’(*) = U(*) - R(*); // trial allocation
  - A’(i,*) = A(i,*) + R(*);
  - if (isSafe(U', D, A')) { // allocate
    - A(i,*) = A’(i,*);
    - U(*) = U’(*);
  } else . . . Restore old state; suspend process . . .

  - isSafe(U', D, A’)
    - True if there is a possible sequence (path) of process completions that includes all processes

The Test for Safety

```java
boolean isSafe(U, D, A) {
  Utmp(*) = U(*);
  Rest = . . . Set of all processes . . .
  safePath = True;
  while (safePath) {
    if (there is a process i such that
        D(i,*) - A(i,*) ≤ Utmp(*)) {
      // simulate allocation
      Utmp(*) = Utmp(*) + A(i,*);
      Rest = Rest - { i };
      } else safePath = False;
    }
    return (isEmpty(Rest));
}
```

Pros and Cons of Deadlock Avoidance

- **Pros**
  - Less restrictive than deadlock prevention
  - Not necessary to preempt/rollback processes

- **Cons**
  - Must state in advance the max resource demand D(*,*)
  - There must be a fixed amount of resources Q(*,*) to allocate
  - Ability to finish is completely determined by resource demands
    - Execution order of the process under consideration must not be constrained by synchronization requirements of other processes