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 (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 occurs)

  ```
  Server Process:
  open("reqFifo",O_RDONLY);
  open("respFifo",O_RDONLY);
  
  Client Process:
  open("respFifo",O_WRONLY);
  open("reqFifo", O_WRONLY);
  ```

  // potential deadlock

  ```
  Process (0):
  do forever {
    Get A stick;
    Get B stick;
    Eat;
    Drop B stick;
    Drop A stick;
  }

  Process (1):
  do forever {
    Get B stick;
    Get A stick;
    Eat;
    Drop B stick;
    Drop A stick;
  }

  // potential deadlock

  Process (1a):
  do forever {
    Get A stick;
    Get B stick;
    Eat;
    Drop A stick;
    Drop B stick;
  }

  // deadlock free
  ```

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??**
**Progress Diagram**

- 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 process 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**

- 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 (based on knowledge of future requests) to avoid deadlock (i.e., find a **safe path**)

- Deadlock Detection
  » Periodically run an algorithm to detect circular waiting
  » After detecting deadlock, run a recovery algorithm to remove the 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 that resources be requested in increasing order

Resource Ordering

- Suppose:
  » Resources R(i), i=1:N, are ordered
    • R(1) < R(2) < ... < R(N) where "<" denotes "precedes"
    • Note: i < j ⇒ R(i) < R(j)

- Simplified Example
  » Processes 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.)

- Each process must 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**: A state in which 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
  » When there is a request, update the system state as if it is granted
  » If the state is safe, grant the request; else block the process until it is safe to grant the 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: 9
    - R2: 3
    - R3: 6
  - Unallocated \([U(j)]\):
    - R1: 1
    - R2: 1
    - R3: 2
  - Max Demand \([D(i,j)]\):
    - R1: 3
    - R2: 1
    - R3: 4
  - Allocation \([A(i,j)]\):
    - P1: 0
    - P2: 0
    - P3: 0
    - P4: 0

- **Example of Banker's Algorithm**
  - Fundamental Question: *Is this a safe state?*
    - Can some process finish? Releases all of its resources
  - Quantity \([Q(j)]\):
    - R1: 9
    - R2: 3
    - R3: 6
  - Unallocated \([U(j)]\):
    - R1: 0
    - R2: 1
    - R3: 1
  - Demand \([D(i,j)]\):
    - R1: 3
    - R2: 1
    - R3: 4
  - Allocation \([A(i,j)]\):
    - P1: 1
    - P2: 6
    - P3: 2
    - P4: 4

- **An Unsafe State**
  - Quantity \([Q(j)]\):
    - R1: 9
    - R2: 3
    - R3: 6
  - Unallocated \([U(j)]\):
    - R1: 0
    - R2: 1
    - R3: 1
  - Demand \([D(i,j)]\):
    - R1: 3
    - R2: 1
    - R3: 4
  - Allocation \([A(i,j)]\):
    - P1: 2
    - P2: 5
    - P3: 2
    - P4: 0

- **P2 Can Complete**
  - Quantity \([Q(j)]\):
    - R1: 9
    - R2: 3
    - R3: 6
  - Unallocated \([U(j)]\):
    - R1: 0
    - R2: 1
    - R3: 1
  - Demand \([D(i,j)]\):
    - R1: 3
    - R2: 1
    - R3: 4
  - Allocation \([A(i,j)]\):
    - P1: 1
    - P2: 5
    - P3: 2
    - P4: 0

- **P1 requests 1 more of R1 and 1 of R3 ... Grant ???**
- **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 Can Complete**
  - 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</td>
<td>R2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Demand [D(i,j)]:</th>
<th>Allocation [A(i,j)]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
</tr>
</tbody>
</table>

Quantity [Q(j)]:

- R1: 9
- R2: 3
- R3: 6

Unallocated [U(j)]:

- R1: 2
- R2: 3
- R3: 4

Demand [D(i,j)]:

- P1: R1: 3, R2: 2, R3: 2
- P2: R1: 0, R2: 0, R3: 0
- P3: R1: 3, R2: 1, R3: 4
- P4: R1: 4, R2: 2, R3: 2

Allocation [A(i,j)]:

- P1: R1: 1, R2: 0, R3: 0
- P2: R1: 0, R2: 0, R3: 0
- P3: R1: 2, R2: 1, R3: 1
- P4: R1: 0, R2: 0, R3: 2

The Banker's Algorithm

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

- \( U'(*) = U(*) - R(*); \) // trial allocation
- \( A'(i,*) = A(i,*) + R(*); \)
  - if (isSafe(U', D, A')) { // allocate
    - \( A(i,*) = A'(i,*); \)
    - \( U(*) = U'(*); \)
  - } else . . . Restore old state and 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,*) \leq Utmp(*)) {
            // simulate allocation
            Utmp(*) = Utmp(*) + A(i,*);
            Rest = Rest - { i }; // simulate release
            } 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