Deadlock (CSE 422S)

Ken Wong
Washington University

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

Two Dining Philosophers

0

\[\text{Process (0): do forever \{ \text{Get A stick; Get B stick; Eat; Drop B stick; Drop A stick; } \} } \]

1

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

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

\[\text{// potential deadlock } \]

\[\text{// deadlock free } \]

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 ???

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 cannot be taken away from the owning process without adversely affecting its computation
  - e.g., a write-once device that has already started writing

Two Dining Philosophers

Process (0):

\[\text{do forever \{ Get A stick; Get B stick; Eat; Drop B stick; Drop A stick; } \} \]

Process (1):

\[\text{do forever \{ Get B stick; Get A stick; Eat; Drop B stick; Drop A stick; } \} \]

Process (1a):

\[\text{do forever \{ Get A stick; Get B stick; Eat; Drop A stick; Drop B stick; } \} \]

// potential deadlock

// deadlock free

Progress Diagram

Progress (Philosopher 0)

Contention for B

Drop A

Drop B

Has B

Get B

Has A

Get A

Contention for A

Deadlock Region

Progress (Philosopher 1)

Get B

Get A

Drop B

Drop A

Has A

Has B

// deadlock free

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**

- Can we avoid deadlock by reordering the resource grants?

**Approaches for Handling Deadlock**

- Ignore Deadlock
  - If infrequent enough

- 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 (Simplified)

- **Suppose:** Resources R(i), i=1:N, are ordered
  - \( R(1) \prec R(2) \prec \ldots \prec R(N) \) where “\( \prec \)” denotes “precedes”
  - Note: \( i < j \Rightarrow R(i) \prec R(j) \)
- **Simplification:** Processes acquire-release resources in pairs
  - Acquire \( R(j) \), Acquire \( R(k) \), Use, Release \( R(k) \), Release \( R(j) \) where \( R(j) \prec R(k) \)
- **Rule**
  - All requests must be made in resource order
  - This rule works even when processes can acquire more than two resources

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 \([Q(j)]\): Unallocated \([U(j)]\):
  - Max Demand \([D(i,j)]\): Allocation \([A(i,j)]\):

<table>
<thead>
<tr>
<th>Process</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

- **Ex. 1:** P1 requests 1 more unit of R1 and 1 unit of R3
An Unsafe State After Ex. 1

- P1, P2 and P3 still need 1 unit of R1, but none left

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
<th>Unallocated [U(j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

P2 Can Complete

- P2 returns resources and demand goes to 0

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
<th>Unallocated [U(j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Example of Banker's Algorithm

- Is this a safe state?
  » Can some process finish? Releases its resources

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
<th>Unallocated [U(j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Example of Banker's Algorithm

- Safe Order: P2, P1, P3, P4

<table>
<thead>
<tr>
<th>Quantity [Q(j)]</th>
<th>Demand [D(i,j)]</th>
<th>Allocation [A(i,j)]</th>
<th>Unallocated [U(j)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R1</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
The Banker's Algorithm

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

  $U'(\ast) = U(\ast) - R(\ast)$; // trial allocation
  $A'(i,\ast) = A(i,\ast) + R(\ast)$;
  if (isSafe($U', D, A'$)) { // allocate
    $A(i,\ast) = A'(i,\ast)$;
    $U(\ast) = U'(\ast)$;
  } 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(\ast) = U(\ast);
    Rest = . . . Set of all processes . . .
    safePath = True;
    while (safePath) {
        if (there is a process i such that $D(i,\ast) - A(i,\ast) \leq Utmp(\ast)$) {
            // simulate allocation
            Utmp(\ast) = Utmp(\ast) + A(i,\ast);
            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 $D(\ast,\ast)$
  - There must be a fixed amount of resources $Q(\ast,\ast)$ to allocate
  - Execution order of the process under consideration must not be constrained by synchronization requirements of other processes
    - Ability to finish is completely determined by resource demands