Deadlocks

Motivating Examples

- Two producer processes share a buffer but use a different protocol for accessing the buffers

```
Producer1()
    P(emptyBuffer)
    P(producerMutexLock)
    : PRODUCTION
```

```
Producer2()
    P(producerMutexLock)
    P(emptyBuffer)
    : PRODUCTION
```

- A postscript interpreter and a visualization program compete for memory frames

```
PSInterpreter()
    request(memory_frames, 10)
    request(frame_buffer, 1)
    <draw file on screen>
```

```
Visualize()
    request(frame_buffer, 1)
    request(memory_frames, 20)
    <update display>
```

Concurrency Issues

- Past lectures:
  - Problem: Safely coordinate access to shared resource
  - Solutions:
    - Use semaphores, monitors, locks, condition variables
    - Coordinate access within shared objects

- What about coordinated access across multiple objects?
  - If you are not careful, it can lead to deadlocks

- Today's lecture:
  - What is a deadlock?
  - How can we address deadlocks?

The TENEX Case

- If a process requests all systems buffers, operator console tries to print an error message
  - To do so
    - lock the console
    - request a buffer

DUH!
**Deadlock**

**Definition**

A set of processes is deadlocked when every process in the set is waiting for an event that can only be generated by some process in the set.

**Starvation vs. deadlock**

- Starvation: threads wait indefinitely (e.g., because some other thread is using a resource).
- Deadlock: circular waiting for resources.

**Deadlock → starvation, but not the other way.**

---

**A Graph Theoretic Model of Deadlock**

The resource allocation graph (RAG)

- Basic components of any resource allocation problem
  - Processes and resources
- Model the state of a computer system as a directed graph
  - \( \mathcal{G} = (V, E) \)
    - \( V \): the set of vertices = \{ \( P_1 \), \ldots, \( P_n \) \} \( \cup \) \{ \( R_1 \), \ldots, \( R_m \) \}
  - \( E \): the set of edges = (edge from a resource to a process) \( \cup \) (edge from a process to a resource)

---

**Resource Allocation Graphs**

**Examples**

- A PostScript interpreter that is waiting for the frame buffer lock and a visualization process that is waiting for memory.
  - \( V = \{ \text{PS interpret, visualization} \} \cup \{ \text{memory frames, frame buffer lock} \} \)

---

**A Graph Theoretic Model of Deadlock**

Resource allocation graphs & deadlock

- **Theorem:** If a resource allocation graph does not contain a cycle then no processes are deadlocked.
  - A cycle in a RAG is a necessary condition for deadlock.
  - Is the existence of a cycle a sufficient condition?
A Graph Theoretic Model of Deadlock
Resource allocation graphs & deadlock

Theorem: If there is only a single unit of all resources then a set of processes are deadlocked iff there is a cycle in the resource allocation graph.

Visualization

Process
Memory Frames
Frame Buffer
PostScript Interpreter

Using the Theory
An operational definition of deadlock

A set of processes are deadlocked iff the following conditions hold simultaneously:
1. Mutual exclusion is required for resource usage (serially usable)
2. A process is in a "hold-and-wait" state
3. Preemption of resource usage is not allowed
4. Circular waiting exists (a cycle exists in the RAG)

Dealing With Deadlock
Deadlock prevention & avoidance

Adopt some resource allocation protocol that ensures deadlock can never occur
- Deadlock prevention/avoidance
  - Guarantee that deadlock will never occur
  - Generally breaks one of the following conditions:
    - Mutex
    - Hold-and-wait
    - No preemption
    - Circular wait
- Deadlock detection and recovery
  - Admit the possibility of deadlock occurring and periodically check for it
  - On detecting deadlock, abort
  - Breaks the no-preemption condition

Using the Theory
An operational definition of deadlock

Dealing With Deadlock
Deadlock avoidance

Examine each resource request and determine whether or not granting the request can lead to deadlock

Define a set of vectors and matrices that characterize the current state of all resources and processes
- resource allocation state matrix
  - \( P \times R \times R \times ... \times R \)
- maximum claim matrix
  - \( M \times R \times R \times ... \times R \)
  - \( M_{ij} = \) the maximum number of units of resource \( j \) that the process \( i \) will ever require simultaneously
- available vector
  - \( \text{Avail} \times R \)
  - \( \text{Avail} = \) the number of units of resource \( j \) that are unallocated
**Deadlock Avoidance**

**State matrices example**

A computer system with 5 processes and 4 resources

<table>
<thead>
<tr>
<th>ALLOCATION</th>
<th>MAXCLAIM</th>
<th>AVAILABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁ R₁ R₂ R₃</td>
<td>R₁ R₂ R₃</td>
<td>R₁ R₂ R₃</td>
</tr>
<tr>
<td>P₂ 0 0 1 2</td>
<td>0 0 1 2</td>
<td>1 5 2 0</td>
</tr>
<tr>
<td>P₃ 1 0 0 0</td>
<td>1 7 5 0</td>
<td></td>
</tr>
<tr>
<td>P₄ 1 3 9 3</td>
<td>2 3 5 6</td>
<td></td>
</tr>
<tr>
<td>P₅ 0 6 3 2</td>
<td>0 6 5 2</td>
<td></td>
</tr>
<tr>
<td>P₆ 0 0 1 4</td>
<td>0 6 5 6</td>
<td></td>
</tr>
</tbody>
</table>

**Deadlock Avoidance**

**Concept**

The OS will examine each resource request and determine whether or not granting the request can lead to deadlock.

- If, after we grant this request, all processes simultaneously make their maximum claim, will the system deadlock?
- Can we satisfy the maximum claims of processes in some order?

**Deadlock Avoidance Algorithm**

**State definitions**

- A resource allocation state is safe if the system can allocate resources to each process up to its maximum claim such that the system can not deadlock.

There must be an ordering of the processes P₁, P₂, ..., P₅ such that for all processes Pᵢ:

\[
\text{MAXCLAIM}_i - \text{ALLOCATION}_i \leq \text{AVAIL} + \sum_{j=1}^{i-1}\text{ALLOCATION}_j
\]

The largest request for resources that process Pᵢ can make

The resources available to process Pᵢ after processes P₁ through Pᵢ₋₁ terminate

This ordering of processes is called a safe sequence.

If a safe sequence exists then there exists a process (Pᵢ) that can execute to completion in the current state, and Pᵢ can execute to completion at worst after processes P₁ through Pᵢ₋₁ complete.

**Example**

A computer system with 5 processes and 4 resources

<table>
<thead>
<tr>
<th>ALLOCATION</th>
<th>MAXCLAIM</th>
<th>AVAILABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁ R₁ R₂ R₃</td>
<td>R₁ R₂ R₃</td>
<td>R₁ R₂ R₃</td>
</tr>
<tr>
<td>P₂ 0 0 1 2</td>
<td>0 0 1 2</td>
<td>1 5 2 0</td>
</tr>
<tr>
<td>P₃ 1 0 0 0</td>
<td>1 7 5 0</td>
<td></td>
</tr>
<tr>
<td>P₄ 1 3 5 3</td>
<td>2 3 5 6</td>
<td></td>
</tr>
<tr>
<td>P₅ 0 6 3 2</td>
<td>0 6 5 2</td>
<td></td>
</tr>
<tr>
<td>P₆ 0 0 1 4</td>
<td>0 6 5 6</td>
<td></td>
</tr>
</tbody>
</table>

Is this system in a safe state?

Does there exist a safe sequence?

\[
\text{MAXCLAIM}_i - \text{ALLOCATION}_i \leq \text{AVAIL} + \sum_{j=1}^{i-1}\text{ALLOCATION}_j
\]
Deadlock Avoidance Example

Safe sequence computation

1. Compute the largest possible resource request a process can make

<table>
<thead>
<tr>
<th>P_i</th>
<th>MAX_REQUEST</th>
<th>ALLOCATION</th>
<th>MAX_CLAIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>R, R, R, R_1</td>
<td>0, 0, 1, 2</td>
<td>0, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td>P_1</td>
<td>1, 7, 5, 0</td>
<td>1, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td>P_2</td>
<td>2, 3, 5, 6</td>
<td>1, 3, 5, 3</td>
<td></td>
</tr>
<tr>
<td>P_3</td>
<td>0, 6, 5, 2</td>
<td>0, 6, 3, 2</td>
<td></td>
</tr>
<tr>
<td>P_4</td>
<td>0, 6, 5, 6</td>
<td>0, 0, 2, 0</td>
<td></td>
</tr>
<tr>
<td>P_5</td>
<td>0, 6, 4, 2</td>
<td>0, 0, 0, 0</td>
<td></td>
</tr>
</tbody>
</table>

\[ R_1 = R_2 = R_3 = R_4 = \text{MAX_CLAIM} \]

2. Attempt to build a safe sequence

- Does there exist a process \( P_i \) such that MAX_REQUEST_{P_i} ≤ AVAILABLE
- If no, then there is no safe sequence, the state is unsafe
- If yes, add \( P_i \) to the sequence
- Set AVAILABLE = AVAILABLE + ALLOCATION_{P_i}

If \( P_2 \) wants to change its allocation to \( <0, 4, 2, 0> \):
- Is the resulting allocation state safe?
Dealing With Deadlock

Deadlock prevention and avoidance:
- Develop and use resource allocation mechanisms and protocols that prohibit deadlock

Deadlock detection and recovery:
- Let the system deadlock and then deal with it
  Detect that a set of processes are deadlocked
  Recover from the deadlock

Deadlock Detection & Recovery

Detecting deadlock

- Run Banker’s algorithm and see if a safe sequence exists
- Replace \( \text{MAX\_REQUEST} \) with a "REQUEST" matrix
- If a safe sequence does not exist then the system is deadlocked

Recovering from deadlock

- Abort all deadlocked processes & reclaim their resources
- Abort one process at a time until all cycles in the \( \text{RAG} \) are eliminated
- Where to start?
  - Select low priority process
  - Processes with most allocation of resources
- Caveat: ensure that system is in consistent state (e.g., transactions)
- Optimization:
  - Checkpoint processes periodically; rollback processes to checkpointed state

<table>
<thead>
<tr>
<th>ALLOCATION</th>
<th>MAX_CLAIM</th>
<th>AVAILABLE</th>
<th>REQUEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>( 0, 0, 0, 0, 1 )</td>
<td>( 0, 0, 1, 2 )</td>
<td>( 1, 5, 2, 0 )</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>( 1, 0, 0, 0, 0 )</td>
<td>( 7, 1, 5, 0 )</td>
<td>( 0, 2, 1, 0 )</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>( 1, 3, 5, 3, 0 )</td>
<td>( 2, 3, 5, 6 )</td>
<td>( 1, 0, 0, 0 )</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>( 0, 6, 3, 2, 0 )</td>
<td>( 0, 6, 5, 2 )</td>
<td>( 0, 0, 2, 0 )</td>
</tr>
<tr>
<td>( P_5 )</td>
<td>( 0, 0, 1, 4, 0 )</td>
<td>( 0, 6, 5, 6 )</td>
<td>( 0, 2, 2, 2 )</td>
</tr>
</tbody>
</table>