Deadlock

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 deadlock

Today’s lecture:
- What is deadlock?
- How can we address deadlock?

Deadlocks

Motivating Examples

- Two producer processes share a buffer but use a different protocol for accessing the buffers
  - Producer1()
    - P(EMPTYBUFFER)
    - P(PRODUCER_MUTEX_LOCK)
  - Producer2()
    - P(PRODUCER_MUTEX_LOCK)
    - P(EMPTYBUFFER)

- A postscript interpreter and a visualization program compete for memory frames
  - PS_Interpreter() { request(memory_frames, 10) request(frame_buffer, 1) }
  - Visualize() { request(frame_buffer, 1) request(memory_frames, 20) }

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
  - \( G = (V, E) \)
  - \( V \) is the set of vertices \( \{P_1, ..., P_n\} \cup \{R_1, ..., R_m\} \)

- \( E \) is the set of edges:
  - (edges from a resource to a process) \( \cup \)
  - (edges from a process to a resource)

A PostScript interpreter that is waiting for the frame buffer lock and a visualization process that is waiting for memory

V = \( \{\text{PS interpreter, visualization}\} \cup \{\text{memory frames, frame buffer lock}\} \)
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?

Deadlock Avoidance
Resource Ordering

- Recall this situation. How can we avoid it?

```
Producer1() {
    P(emptyBuffer)
    P(producerMutexLock)
}
```

```
Producer2() {
    P(producerMutexLock)
    P(emptyBuffer)
}
```

- Eliminate circular waiting by ordering all locks (or semaphores, or resources). All code grabs locks in a predefined order. Problems?
  - Maintaining global order is difficult, especially in a large project.
  - Global order can force a client to grab a lock earlier than it would like, tying up a resource for too long.
  - Deadlock is a global property, but lock manipulation is local.

Deadlock Detection & Recovery
Recovering from deadlock

- Abort all deadlocked processes & reclaim their resources
- Abort one process at a time until all cycles in the 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

Using the Theory
An operational definition of deadlock

- A set of processes are deadlocked if the following conditions hold simultaneously:
  1. Mutual exclusion is required for resource usage (serially useable)
  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:
      1. Generally breaks one of the following conditions:
        - Mutex
        - Hold-and-wait
        - No preemption
        - Circular wait (This is usually the weak link)
      - Deadlock detection and recovery
        1. Admit the possibility of deadlock occurring and periodically check for it
        2. On detecting deadlock, abort
        3. Breaks the no-preemption condition

What does the RAG for a lock look like?
Dealing With Deadlock

**Deadlock avoidance – Banker’s Algorithm**

- 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**
  
  \[ \text{Alloc}_i = \begin{bmatrix} n_{i1} & n_{i2} & \cdots & n_{ir} \end{bmatrix} \]

- **maximum claim matrix**
  
  \[ \text{Max}_i = \begin{bmatrix} \text{Max}_{i1} & \text{Max}_{i2} & \cdots & \text{Max}_{ir} \end{bmatrix} \]

- **available vector**
  
  \[ \text{Avail}_j = \begin{bmatrix} \text{Avail}_{j1} & \text{Avail}_{j2} & \cdots & \text{Avail}_{jr} \end{bmatrix} \]

Where:

- \( R_i \) is the number of units of resource \( j \) held by process \( i \).
- \( n_{ij} \) is the maximum number of units of resource \( j \) that the process \( i \) will ever require simultaneously.
- \( \text{Avail}_{ji} \) is the number of units of resource \( j \) that are unallocated.

**Deadlock detection & recovery**

- What are some problems with the banker’s algorithm?
  - Very slow \( O(n^2m) \)
  - Too slow to run on every allocation. What else can we do?

- **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 and recover from the deadlock.