COMP 530: Operating Systems

Deadlock

Don Porter
Portions courtesy Emmett Witchel

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?

Deadlock: Motivating Examples

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

```
Producer1() {
    Lock(emptyBuffer)
    Lock(producerMutexLock)
}
Producer2() {
    Lock(producerMutexLock)
    Lock(emptyBuffer)
}
```

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

```
PS_Interpreter() {
    request(memory_frames, 10)  // request file
    request(frame_buffer, 1)    // request frame
    draw file on screen
}
Visualization() {
    request(frame_buffer, 1)    // display data
    request(memory_frames, 20)  // update display
}
```

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 \( \Rightarrow \) starvation, but not the other way

Resource Allocation Graph

- Basic components of any resource allocation problem
  - Processes and resources
- Model the state of a computer system as a directed graph
  - \( G = (V, E) \)
  - \( V \) = the set of vertices = \( \{P_1, ..., P_n\} \cup \{R_1, ..., R_m\} \)
  - \( E \) = the set of edges =
    - (edges from a resource to a process)
    - (edges from a process to a resource)
**Resource Allocation Graph & 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?

**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)

**Deadlock Prevention and/or Recovery**

- 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 “This is usually the weak link”
  - Deadlock detection and recovery
    - Admit the possibility of deadlock occurring and periodically check for it
    - On detecting deadlock, abort
      - Breaks the no-preemption condition
      - And non-trivial to restore all invariants

**Deadlock Avoidance: Resource Ordering**

- Recall this situation. How can we avoid it?

  ```
  Producer1() {
      Lock(emptyBuffer)
      Lock(producerMutexLock):
  }
  Producer2() {
      Lock(producerMutexLock)
      Lock(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.

**Lock Ordering**

- A program code convention
- Developers get together, have lunch, plan the order of locks
- In general, nothing at compile time or run-time prevents you from violating this convention
  - Research topics on making this better:
    - Finding locking bugs
    - Automatically locking things properly
    - Transactional memory
How to order?
- What if I lock each entry in a linked list. What is a sensible ordering?
  - Lock each item in list order
  - What if the list changes order?
  - Uh-oh! This is a hard problem
- Lock-ordering usually reflects static assumptions about the structure of the data
  - When you can’t make these assumptions, ordering gets hard

Linux solution
- In general, locks for dynamic data structures are ordered by kernel virtual address
  - I.e., grab locks in increasing virtual address order
- A few places where traversal path is used instead

Lock ordering in practice

From Linux: fs/dcache.c

```c
void d_cache_alien(int dentry, struct inode *inode, struct hlist_node *node, void (*spun_lock)(struct dentry *dentry); void (*unsn_lock)(struct dentry *dentry); void (*unsn_lock)(struct dentry *dentry); spin_lock(&inode->dentry_rwlock); dentry->dentry_rwlock.node = node;
```

Deadlock Avoidance: Banker’s Algorithm
- Examine each resource request and determine whether 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
  - $A_{i}$ = the number of units of resource $j$ held by process $i$
- maximum claim matrix
  - $M_{i,j}$ = the maximum number of units of resource $j$ that the process $i$ will ever require simultaneously
- available vector
  - $A_{i}$ = the number of units of resource $j$ that are unallocated

Common in Databases; Hard in General-Purpose Apps
Dealing with Deadlock

• 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
    ➢ Recover from the deadlock

Summary and Editorial

• Deadlock is one difficult issue with concurrency
• Lock ordering is most common solution
  – But can be hard:
    • Different traversal paths in a data structure
    • Complicated relationship between structures
  – Requires thinking through the relationships in advance
• Other solutions possible
  – Detect deadlocks, abort some programs, put things back together (common in databases)
    • Transactional Memory
  – Banker’s algorithm

Current Reality

<table>
<thead>
<tr>
<th>Performance</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fine-Grained Locking</td>
<td>Unsavory trade-off between complexity and performance scalability</td>
</tr>
<tr>
<td>Coarse-Grained Locking</td>
<td></td>
</tr>
</tbody>
</table>