CS 31: Intro to Systems
Deadlock (and other scary thread interactions)

Vasanta Chaganti & Kevin Webb
Swarthmore College
December 12, 2023
“Deadly Embrace”

- *The Structure of the THE-Multiprogramming System* (Edsger Dijkstra, 1968)

- Also introduced semaphores

- Deadlock is as old as synchronization
What is Deadlock?

• Deadlock is a problem that can arise:
  – When processes compete for access to limited resources
  – When threads are incorrectly synchronized

• Definition:
  – Deadlock exists among a set of threads if every thread is waiting for an event that can be caused only by another thread in the set.
What is Deadlock?

• Set of threads are permanently blocked
  – Unblocking of one relies on progress of another
  – But none can make progress!

• Example
  – Threads A and B
  – Resources X and Y
  – A holding X, waiting for Y
  – B holding Y, waiting for X
  – Each is waiting for the other; will wait forever
Traffic Jam as Example of Deadlock

- Cars A, B, C, D
- Road W, X, Y, Z
- Car A holds road space Y, waiting for space Z
- “Gridlock”

Cars deadlocked in an intersection
Traffic Jam as Example of Deadlock

Cars deadlocked in an intersection

Resource Allocation Graph
Four Conditions for Deadlock

1. Mutual Exclusion
   – Only one thread may use a resource at a time.

2. Hold-and-Wait
   – Thread holds resource while waiting for another.

3. No Preemption
   – Can’t take a resource away from a thread.

4. Circular Wait
   – The waiting threads form a cycle.
Four Conditions for Deadlock

1. Mutual Exclusion
   – Only one thread may use a resource at a time.

2. Hold-and-Wait
   – Thread holds resource while waiting for another.

3. No Preemption
   – Can’t take a resource away from a thread.

4. Circular Wait
   – The waiting threads form a cycle.
Examples of Deadlock

• Memory (a reusable resource)
  – total memory = 200KB
  – $T_1$ requests 80KB
  – $T_2$ requests 70KB
  – $T_1$ requests 60KB (wait)
  – $T_2$ requests 80KB (wait)

• Messages (a consumable resource)
  – $T_1$: receive $M_2$ from $P_2$
  – $T_2$: receive $M_1$ from $P_1$
struct account {
    mutex lock;
    int balance;
}

Transfer(from_acct, to_acct, amt) {
    lock(from_acct.lock);
    lock(to_acct.lock)

    from_acct.balance -= amt;
    to_acct.balance += amt;

    unlock(to_acct.lock);
    unlock(from_acct.lock);
}
If multiple threads are executing this code, is there a race? Could a deadlock occur?

```c
struct account {
    mutex lock;
    int balance;
};

Transfer(from_acct, to_acct, amt) {
    lock(from_acct.lock);
    lock(to_acct.lock);

    from_acct.balance -= amt;
    to_acct.balance += amt;

    unlock(to_acct.lock);
    unlock(from_acct.lock);
}
```

If there’s potential for a race/deadlock, what execution ordering will trigger it?

<table>
<thead>
<tr>
<th>Clicker Choice</th>
<th>Potential Race?</th>
<th>Potential Deadlock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>C</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>D</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Common Deadlock

Thread 0
Transfer(acctA, acctB, 20);
Transfer(...)
   lock(acctA.lock);
   lock(acctB.lock);

Thread 1
Transfer(acctB, acctA, 40);
Transfer(...)
   lock(acctB.lock);
   lock(acctA.lock);
Common Deadlock

Thread 0
Transfer(acctA, acctB, 20);
Transfer(...) {
  lock(acctA.lock);
    \textcolor{red}{T_0 \text{ gets to here}}
  lock(acctB.lock);
}

Thread 1
Transfer(acctA, acctB, 40);
Transfer(...) {
  lock(acctB.lock);
    \textcolor{red}{T_1 \text{ gets to here}}
  lock(acctA.lock);
}

\textcolor{red}{T_0 \text{ holds A’s lock, will make no progress until it can get B’s.}}
\textcolor{red}{T_1 \text{ holds B’s lock, will make no progress until it can get A’s.}}
How to Attack the Deadlock Problem

• What should your OS do to help you?

• Deadlock Prevention
  – Make deadlock impossible by removing a condition

• Deadlock Avoidance (“Banker’s Algorithm”)
  – Avoid getting into situations that lead to deadlock

• Deadlock Detection
  – Don’t try to stop deadlocks
  – Rather, if they happen, detect and resolve
Which type of deadlock-handling scheme would you expect to see in a modern OS (Linux/Windows/OS X) ?

A. Deadlock prevention

B. Deadlock avoidance

C. Deadlock detection/recovery

D. Something else

“Ostrich Algorithm”
How to Attack the Deadlock Problem

• Deadlock Prevention
  – Make deadlock impossible by removing a condition

• Deadlock Avoidance
  – Avoid getting into situations that lead to deadlock

• Deadlock Detection
  – Don’t try to stop deadlocks
  – Rather, if they happen, detect and resolve

• These all have major drawbacks...
Other Thread Complications

• Deadlock is not the only problem

• Performance: too much locking?

• Priority inversion

• ...
Priority Inversion

• Problem: Low priority thread holds lock, high priority thread waiting for lock.
  – What needs to happen: boost low priority thread so that it can finish, release the lock
  – What sometimes happens in practice: low priority thread not scheduled, can’t release lock

• Example: Mars Pathfinder (1997)
Mars Rover

• Three periodic tasks:
  1. Low priority: collect meteorological data
  2. Medium priority: communicate with NASA
  3. High priority: data storage/movement

• Tasks 1 and 3 require exclusive access to a hardware bus to move data.
  – Bus protected by a mutex.
Mars Rover

• Failsafe timer (watchdog): if high priority task doesn’t complete in time, reboot system

• Observation: uh-oh, this thing seems to be rebooting a lot, we’re losing data...

JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren’t important, using the rationale "it was probably caused by a hardware glitch".
What Happened: Priority Inversion

Low priority task, running happily.

Time
What Happened: Priority Inversion

Low priority task acquires mutex lock.
What Happened: Priority Inversion

Time

H

M

L

Blocked

Medium task starts up, takes CPU.
What Happened: Priority Inversion

High priority task tries to acquire mutex, can’t because it’s already held.

[Diagram showing time line with H, M, and L priorities and blocked periods]
What Happened: Priority Inversion

High priority task tries to acquire mutex, can’t because it’s already held.

Low priority task can’t give up the lock because it can’t run - medium task takes priority over it.
What Happened: Priority Inversion

High priority is taking too long.
Reboot!
Solution: Priority Inheritance

High priority task tries to acquire mutex, can’t because it’s already held.

Give to blue red’s (higher) priority!
Solution: Priority Inheritance

High priority finishes in time.

Release lock, revert to low priority.

Blocked

Blocked

Blocked

Deadlock Summary

• Deadlock occurs when threads are waiting on each other and cannot make progress.

• Deadlock requires four conditions:
  – Mutual exclusion, hold and wait, no resource preemption, circular wait

• Approaches to dealing with deadlock:
  – Ignore it – Living life on the edge (most common!)
  – Prevention – Make one of the four conditions impossible
  – Avoidance – Banker’s Algorithm (control allocation)
  – Detection and Recovery – Look for a cycle, preempt/abort