Deadlock

Michael E. Locasto
Seeds of the (Meta)-Problem

Safe concurrent access requires mutual exclusion.

Processes typically need more than a single resource.

Key issue: imposing logical order on a sequence of locking operations.
Resource Deadlock Definition

Two or more processes are blocked waiting for access to the same set of resources in such a way that none can make progress.

“A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.” –MOS, pg 437
Conditions Necessary For Deadlock

**Mutual Exclusion:** Each resource is only available via a protected critical section (to a single task)

**Hold & Wait:** Tasks holding resources can request others

**Circular Waiting:** Must have two or more tasks depending on each other

**No Preemption:** Tasks must release resources
It arises from our need to address the problems of concurrency, and our naïve solutions to address synchronized access to critical sections (and the resources they refer to).

**DEADLOCK IS A MANUFACTURED PROBLEM**
The book uses examples of resources such as physical devices: printers, tape drives, CD/optical drives, etc. More typically, the “resources” that cause deadlock are the locks and semaphores protecting entry to the critical regions permitting access to these devices.

RESOURCES ARE USUALLY LOCKS IN THIS CONTEXT
Resource Graphs: A Modeling Approach to Understanding Deadlock

Processes are circles

Resources are squares

Directed Graphs depict dependencies

Circle -> Square: Blocked Waiting / Request

Square -> Circle: Currently Held

Cycles indicate deadlock

An approach to preventing deadlock:

(1) carry out a sequence of resource requests

(2) update the resource graph with the corresponding edge

(3) check if graph contains a cycle
DETECT DEADLOCK
Detecting Cycles

Can be done in any number of ways.

For example, if you have a linked list, send a pointer iterating over the list and compare with the current node under consideration; if equal, you have a loop.

Detecting cycles in graphs is typically done through some form of depth-first search.

The algorithm in the book describes a depth-first approach to exploring the graph to see if any outgoing edges lead back to the “root” node.

Assume a list of nodes
For each node N
Let root=N
Add root to temp list L
if(root is already in L) cycle();
No: back to previous node.
# Approaches to Handling Deadlock

<table>
<thead>
<tr>
<th>Avoidance</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore it. (test this later)</td>
<td>Let deadlock happen; detect this condition and attempt to recover</td>
</tr>
<tr>
<td>Implement a dynamic avoidance (e.g., speculative execution, resource</td>
<td>Prevent it from happening in the first place (usually by scrupulous</td>
</tr>
<tr>
<td>graphs) approach to inform resource allocation sequencing</td>
<td>coding); attempt to make one of the four deadlock conditions impossible</td>
</tr>
<tr>
<td></td>
<td>or untrue</td>
</tr>
</tbody>
</table>
RECOVER FROM DEADLOCK
Recovery Tactics

Preemption: Suspend & steal. Depends on resource semantics.

Rollback: checkpoint & restart. Select a task to rollback, thereby releasing its held resource(s)

Terminate: (1) break cycle (2) make extra resources available
Peer into the future and sidestep. Impossible.

DEADLOCK AVOIDANCE
Banker’s Algorithm Caveats

The art of abstraction hinders the application to reality.

Need to know maximum resource request.

Physical devices break.

Number of new tasks hard to predict.
Banker’s Algorithm

Dijkstra 1965
Safe states: there exists an ordering of resource allocations such that all tasks run to completion even if maximal demand made on the system instantly
Unsafe!=deadlock...just risky
Giving out credit: but delivered credit is unevenly distributed, and remaining credit may not allow the system to reach a safe path (i.e., allow some sequence of tasks to complete)
Break at least 1 of those 4 conditions

DEADLOCK PREVENTION
Program Structure

Introduce sequential processing (to avoid mutual exclusion).

Tough to break hold & wait (prediction again)

No preemption: tough to kill or preempt b/c semantics & data correctness may rely on this

Circular wait: sequentially enumerate resources; impose (code) rule of sequential access
Causing deadlock with pthreads.

CODE: LET’S MAKE THIS HAPPEN
Discussion Questions

Can we make the scheduler aware of potential deadlocks?

Can deadlocks be a security mechanism?

Project: write a wrapper for pthread that outputs a GraphViz .dot specification to depict a resource graph (of locks)