CSE 153
Design of Operating Systems

Summer 2022

Lecture 9: Deadlock
Today: Deadlock—the deadly embrace!

- Synchronization— we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - You have likely been intuitively avoiding this situation already

- Consider: processes that use multiple critical sections/need different resources
  - If one process tries to access a resource that a second process holds, and vice-versa, they can never make progress

- We call this situation **deadlock**, and we’ll look at:
  - Definition and conditions necessary for deadlock
  - Representation of deadlock conditions
  - Approaches to dealing with deadlock
Deadlock Definition

- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized
- Definition:
  - Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set

```c
lockA->Acquire();
...
lockB->Acquire();
```

Process 1

```c
lockB->Acquire();
lockA->Acquire();
```

Process 2
Whether deadlock occurs or not depends on the order of operations!
Conditions for Deadlock

- Deadlock can exist if and only if following conditions hold simultaneously:

1. **Mutual exclusion** – At least one resource must be held in non-sharable mode

2. **Hold and wait** – Must be one process holding one resource and waiting for another resource

3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)

4. **Circular wait** – Must exist set of processes \([P_1, \ldots, P_n]\) such that \(P_1\) is waiting for \(P_2\), and \(P_2\) for \(P_3\), and etc.
Each lawyer needs two chopsticks to eat.
Dining Lawyers

Each lawyer needs two chopsticks to eat. Each grabs chopstick on the left first.
Resource Allocation Graph (RAG) Example

Processes: \( P_1, P_2, P_3 \)
Resources: \( R_1, R_2, R_3 \)

Resource Requested:

\[ P_i \rightarrow R_j \]

Resource Allocated:

\[ P_i \leftarrow R_j \]

Is there a cycle?
- If the graph has no cycles, deadlock **cannot exist**
- If the graph has a cycle, deadlock **may exist**

A cycle…and deadlock!
Resoucece Allocation Graph (RAG) Example

A cycle...and deadlock!

Same cycle...but no deadlock. Why?
A Simpler Case

- If:
  - all resources are single unit
  - all processes make single requests

- Then:
  - can represent resource state with **Waits-For Graph (WFG)**

- If the graph has no cycles, deadlock **cannot exist**
- If the graph has a cycle, deadlock **exists**
Waits-For Graph (WFG)

Processes: $P_1, P_2, P_3$

Resource Requested:

$P_i \rightarrow P_j$

Is there a cycle?

- If the graph has no cycles, deadlock cannot exist
- If the graph has a cycle, deadlock exists

A cycle...and deadlock!
Dealing With Deadlock

- Four approaches for dealing with deadlock:
  - Ignore it – how lucky do you feel?
  - Prevention – make it impossible for deadlock to happen
  - Avoidance – control allocation of resources
  - Detection and Recovery – look for a cycle in dependencies
Deadlock Prevention

- Prevention – Ensure at least one of necessary conditions cannot happen
  - Mutual exclusion
    - Make resources sharable (not generally practical)
  - Hold and wait
    - Process cannot hold one resource when requesting another
  - Preemption
    - OS can preempt resource (costly)
  - Circular wait
    - Impose ordering (numbering) on resources and request them in order (popular implementation technique)
Deadlock Prevention

- **One shot allocation**: ask for all your resources in one shot; no more resources can be requested
  - What ingredient does this prevent?
  - Comments?

- Preemption (different types)
  - **Nice**: Give up a resource if what you want is not available
  - **Aggressive**: steal a resource if what you want is not available

- Hierarchical allocation:
  - Assign resources to classes
  - Can only ask for resources from a higher number class than what you hold now
Deadlock Avoidance

- Prevention can be too conservative – can we do better?

- Avoidance
  - Provide info in advance about what resources will be needed by processes
  - System only grants resource requests if it knows deadlock cannot happen
  - Avoids circular dependencies

- Tough:
  - Hard to determine all resources needed in advance
  - Good theoretical problem, not as practical to use
Banker’s Algorithm:

- classic approach to deadlock avoidance for resources with multiple units

1. Assign credit limit to each customer (process)
   - Maximum credit claim must be stated in advance

2. Reject any request that leads to dangerous state
   - Dangerous state: sudden request by any customer for full credit limit could lead to deadlock
   - A recursive reduction procedure recognizes dangerous states

3. In practice, system must keep resource usage below capacity to maintain a resource surplus
   - Rarely used in practice due to low resource utilization
Possible System States

Unsafe
Safe
Deadlock
Banker’s Algorithm (Simplified)

Number of Processes: 2 (P_1 & P_2)
Resources Available: 4
Process Credit Limit: 3
Detection and Recovery

- Without deadlock prevention/avoidance:
  - Deadlock may occur
  - Need to detect deadlock and recover from it

- Need two algorithms:
  1) Determine whether deadlock occurred
  2) Recover from deadlock

- Possible, but expensive (time consuming)
  - Implemented in VMS
  - Run detection algorithm when resource request times out
Deadlock Detection

- **Detection**
  - Traverse resource graph looking for cycles
  - If cycle found, preempt resource (force process to release)

- **Expensive**
  - Many processes/resources to traverse

- **Only invoke detection algorithm depending on:**
  - How often/likely deadlock is
  - How many processes likely to be affected when it occurs
Deadlock Recovery

Once deadlock detected, two recovery options …

1. Abort processes
   - Abort all deadlocked processes
     » Processes need to start over again
   - Abort one process at a time until cycle is eliminated
     » System needs to rerun detection after each abort

2. Preempt resources (force their release)
   - Need to select process and resource to preempt
   - Need to rollback process to previous state
   - Need to prevent starvation
Deadlock Summary

- Deadlock occurs when processes are waiting on each other and cannot make progress
  - Cycles in Resource Allocation Graph (RAG)

- Deadlock requires four conditions
  - Mutual exclusion, hold and wait, no resource preemption, circular wait

- Four approaches to dealing with deadlock:
  - Ignore it – Living life on the edge
  - Prevention – Make one of the four conditions impossible
  - Avoidance – Banker’s Algorithm (control allocation)
  - Detection and Recovery – Look for a cycle, preempt or abort
Remaining slides are FYI – not responsible for them on exam
Other problems with synchronization

- Performance problems
  - Lock contention
    » Only for spinlocks?
    » No; consider blocking locks
  - Lock convoying: several processes need locks in the same order. Slow process gets in first
Advanced Synchronization

- Lock free data structures
  - Can we avoid using locks?

- Transactional memory (e.g., Intel TSX)
  - System support for lock free operation
Lock-Free Data Structures

- Assume compare and swap atomic instruction
  - Limitation: swap a single memory location
  - Only supported on some processor architectures

- Rewrite critical section
  - Create copy of data structure
  - Modify copy
  - Swap in pointer to copy iff no one else has
  - Restart if pointer has changed
consumes_item() {  //consumer code
    do {
        mine = ConsistentCopy(p);
        update_copy();
    } while ((compare&swap(mine, p) != p);
    return item;
}

- No locks! What’s the trick?