Lecture 8: Deadlock
Announcements

- Project 1 due on Friday
  - Good time to reemphasize academic integrity policy

- Next class: Review of material for mid-term
  - Come prepared with questions
  - Attempt HW2 and other practice problems
    » I’ll post more
  - Come to office hours
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
Process 1
lockA->Acquire();
...
lockB->Acquire();

Process 2
lockB->Acquire();
...
lockA->Acquire();
```
Whether deadlock occurs or not depends on the order of operations!
Deadlock and Resources

- There are two kinds of resources: consumable and reusable
  - Consumable resources are generated and destroyed by processes: e.g., a process waiting for a message from another process
  - Reusable resources are allocated and released by processes: e.g., locks on files
- Deadlock with consumable resources is usually treated as a correctness issue (e.g., proofs) or with timeouts
- From here on, we only consider reusable resources
Conditions for Deadlock

- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. **Mutual exclusion** – At least one resource must be held in a non-sharable mode
  2. **Hold and wait** – There 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** – There must exist a set of processes \([P_1, P_2, P_3, \ldots, P_n]\) such that \(P_1\) is waiting for \(P_2\), \(P_2\) for \(P_3\), etc.
Dining Lawyers

Each lawyer needs two chopsticks to eat. Each grabs chopstick on the right first.
Deadlock can be described using a resource allocation graph (RAG)

The RAG consists of a set of vertices $P=\{P_1, P_2, \ldots, P_n\}$ of processes and $R=\{R_1, R_2, \ldots, R_m\}$ of resources

- A directed edge from a process to a resource, $P_i \rightarrow R_j$, means that $P_i$ has requested $R_j$
- A directed edge from a resource to a process, $R_i \rightarrow P_i$, means that $R_j$ has been allocated to $P_i$
- Each resource has a fixed number of units

If the graph has no cycles, deadlock cannot exist

If the graph has a cycle, deadlock may exist
RAG Example

A cycle...and deadlock!

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

- If all resources are single unit and all processes make single requests, then we can represent the resource state with a simpler waits-for graph (WFG).
- The WFG consists of a set of vertices $P = \{P_1, P_2, \ldots, P_n\}$ of processes.
  - A directed edge $P_i \rightarrow P_j$ means that $P_i$ has requested a resource that $P_j$ currently holds.
- If the graph has no cycles, deadlock cannot exist.
- If the graph has a cycle, deadlock exists.
Dealing With Deadlock

- There are 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 that at least one of the 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 an ordering (numbering) on the 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
  - 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 information in advance about what resources will be needed by processes
  - System only grants resource requests if it knows that 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

- The Banker’s Algorithm is the classic approach to deadlock avoidance for resources with multiple units

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

2. Reject any request that leads to a dangerous state
   - A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
   - A recursive reduction procedure recognizes dangerous states

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

- Safe
- Unsafe
- Deadlock
Banker’s Algorithm Simplified

P_1
\[\text{OK}\]
\[3\]

P_2
\[\text{OK}\]
\[3\]

P_1
\[\text{OK}\]
\[3\]

P_2
\[\text{OK}\]
\[3\]

P_1
\[\text{UNSAFE}\]
\[3\]

P_2
\[3\]
Detection and Recovery

- Detection and recovery
  - If we don’t have deadlock prevention or avoidance, then deadlock may occur
  - In this case, we need to detect deadlock and recover from it

- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock

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

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

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

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

Once a deadlock is detected, we have two 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
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
Lock-Free Bounded Buffer

consume_item() {  //consumer code
    do {
        mine = ConsistentCopy(p);
        update_copy();
    } while ((compare&swap(mine, p) != p);
    return item.
}

- No locks! What’s the trick?