Today: Deadlock

Remember example from last week?

```
my_work(id_t id) { /* id can be 0 or 1 */
    flag[id] = true; /* indicate entering CS */
    while (!flag[1-id]) ; /* entry section */
    /* critical section, access protected resource */
    flag[id] = false; /* exit section */
    ... /* remainder section */
}
```

- What went wrong here?

Example of Deadlock

- Suppose processes $P$ and $Q$ need resources $A$ and $B$:

```
Process P ...
Get A ...
Get B ...
Release A ...
Release B ...
Process Q ...
Get B ...
Get A ...
Release B ...
Release A ...
```

Example: dining philosophers:

- A philosopher needs two forks to eat.
- Idea for protocol:
  - When philosopher gets hungry grab right fork, then grab left fork.
- Is this a good solution?

Not just an OS Problem!

- Law passed by Kansas Legislature in early 20th Century:
  - "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start upon again until the other has gone."
Deadlock Defined
- The permanent blocking of a set of processes that either:
  - Compete for system resources, or
  - Communicate with each other
- Each process in the set is blocked, waiting for an event which can only be caused by another process in the set
  - Resources are finite
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes

Types of Resources
- Reusable
  - Can be used by one process at a time, released and used by another process
    - printers, memory, processors, files
    - Locks, semaphores, monitors
- Consumable
  - Dynamically created and destroyed
  - Can only be allocated once
    - e.g. interrupts, signals, messages

Deadlock continued …
- What conditions must hold for a deadlock to occur?
  - Necessary conditions
  - Sufficient conditions

Conditions for Deadlock
1. Mutual Exclusion
   - Only one process may use a resource at a time
2. Hold and wait
   - A process may hold allocated resources while awaiting assignment of others
3. No preemption
   - No resource can be forcibly removed from a process holding it
   - These are necessary conditions

One more condition...
4. Circular wait
   - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
   - Together, these four conditions are necessary and sufficient for deadlock
   - Circular wait implies hold and wait, but the first results from a sequence of events, while the second is a policy decision

Deadlock Prevention
- Ensure one of the four conditions doesn’t occur
  - Break mutual exclusion –
  - not much help here, as it is often required for correctness
Preventing Hold-and-Wait
- Break “hold and wait” - processes must request all resources at once, and will block until entire request can be granted simultaneously
- May wait a long time for all resources to be available at the same time
- May hold resources for a long time without using them (blocking other processes)
- May not know all resource requirements in advance
- An alternative is to release all currently-held resources when a new one is needed, then make a request for the entire set of resources.

Preventing No-Preemption
- Break “no preemption” - forcibly remove a resource from one process and assign it to another
- Need to save the state of the process losing the resource so it can recover later
- May need to rollback to an earlier state
- Name some resources that this works for…
- Name some resources for which this is hard…
- Impossible for consumable resources

Preventing Circular-wait
- Break “circular wait” - assign a linear ordering to resource types and require that a process holding a resource of one type, R, can only request resources that follow R in the ordering
- e.g. R_i precedes R_j if i < j
- For deadlock to occur, need P to hold R_i and request R_j, while Q holds R_j and requests R_i
- This implies that i < j (for P’s request order) and j < i (for Q’s request order), which is impossible.
- Hard to come up with total order when there are lots of resource types

Deadlock Avoidance
- All prevention strategies are unsatisfactory in some situations
- Avoidance allows the first three conditions, but orders events to ensure circular wait does not occur
  - How is this different from preventing circular wait?
  - Requires knowledge of future resource requests to decide what order to choose

Two Avoidance Strategies
1. Do not start a process if its maximum resource requirements, together with the maximum needs of all processes already running, exceed the total system resources
   - Pessimistic, assumes all processes will need all their resources at the same time
2. Do not grant an individual resource request if it might lead to deadlock
Safe States

- A state is **safe** if there is at least one sequence of process executions that does not lead to deadlock, *even if every process requests their maximum allocation immediately*.
- Example: 3 processes, 1 resource type, 10 instances.

<table>
<thead>
<tr>
<th>PID</th>
<th>Alloc</th>
<th>Max Claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0: Available = 3 A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>T1: Available = 1 B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>T2: Available = 5 C</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>T3: Available = 0 D</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T4: Available = 7 E</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Unsafe States & Algorithm

- An **unsafe** state is one which is not safe.
- Is this the same as a deadlocked state?

Deadlock avoidance algorithm

- For every resource request:
  - Update state assuming request is granted.
  - Check if new state is safe.
  - If so, continue.
  - If not, restore the old state and block the process until it is safe to grant the request.
- This is the banker's algorithm.
- Processes must declare maximum needs.
- See text for details of the algorithm.

Restrictions on Avoidance

- Maximum resource requirements for each process must be known in advance.
- Processes must be independent:
  - If order of execution is constrained by synchronization requirements, system is not free to choose a safe sequence.
- There must be a fixed number of resources to allocate:
  - Tough luck if a printer goes offline!

Deadlock Detection & Recovery

- Prevention and avoidance is awkward and costly.
  - Need to be cautious, thus low utilization.
- Instead, allow deadlocks to occur, but detect when this happens and find a way to break it.
  - Check for circular wait condition periodically.
- When should the system check for deadlocks?

Deadlock Detection & Recovery

- How can you detect a deadlock?

Draw resource alloc graph

- Check for cycles in resource allocation graph.
Deadlock Detection

- Finding circular waits is equivalent to finding a cycle in the resource allocation graph
- Nodes are processes (drawn as circles) and resources (drawn as squares)
- Arcs from a resource to a process represent allocations
- Arcs from a process to a resource represent ungranted requests
- Any algorithm for finding a cycle in a directed graph will do
- Note that with multiple instances of a type of resource, cycles may exist without deadlock

Deadlock Recovery

- Basic idea is to break the cycle
- Drastic - kill all deadlocked processes
- Painful - back up and restart deadlocked processes (hopefully, non-determinism will keep deadlock from repeating)
- Better - selectively kill deadlocked processes until cycle is broken
  - Re-run detection alg. after each kill
- Tricky - selectively preempt resources until cycle is broken
  - Processes must be rolled back

Reality Check

- No single strategy for dealing with deadlock is appropriate for all resources in all situations
- All strategies are costly in terms of computation overhead, or restricting use of resources
- Most operating systems employ the “Ostrich Algorithm”
  - Ignore the problem and hope it doesn’t happen often

Why does the Ostrich Alg work?

- Recall causes of deadlock:
  - Resources are finite
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes
- Prevention/Avoidance/Detection mostly deal with last 2 points
- Modern operating systems virtualize most physical resources, eliminating the first problem
  - Some logical resources can’t be virtualized (there has to be exactly one), such as bank accounts or the process table
  - These are protected by synchronization objects, which are now the only resources that we can deadlock on