LECTURE 1.8
DEADLOCKS

COP4600

Dr. Matthew Gerber

2/17/2016

Deadlocks (7.1)

- In general, processes use resources as follows:
  - A process *requests* the resource, waiting until the resource is available and, once it is, obtaining exclusive access to it
  - The process then *uses* the resource
  - The process then *releases* the resource, freeing it for the use of other processes
- Consider a set of processes P and a set of resources R
- If *every process in* P *is requesting some resource from* R, and *all resources in* R *are already held by processes in* P, every process in P will wait indefinitely
- This is called a *deadlock*

Conditions for Deadlock (7.2)

- Deadlocks occur if, and only if, **all** of the following conditions are met:
  - Mutual Exclusion
    - Resources are held by processes and cannot be shared
  - Hold and Wait
    - At least one process is waiting on a resource while holding a different one
  - No Resource Preemption
    - A resource cannot be forcibly removed from the process holding it
  - Circular Wait
    - A set of processes P = $\{P_0, P_1, \ldots, P_n\}$ must exist so that:
      - For each i so that $0 \leq i < n$, P_i is waiting on a resource that P_{i+1} holds
      - P_n is waiting on a resource that P_0 holds

Deadlock Handling Approaches (7.3)

- There are four ways of dealing with deadlocks
  - Attempt to *prevent* deadlocks by eliminating one of the four conditions that allows them to occur
  - Attempt to *avoid* deadlocks by monitoring resource allocation with an algorithm
  - Attempt to *recover from* deadlocks by detecting them when they occur and taking action
  - *Ignore* deadlocks and hope they don't happen very often

Deadlock Prevention: Mutual Exclusion (7.4.1)

- This is a dead end
- Resources that are inherently sharable (read-only files, for instance) are already marked as sharable for performance purposes, so they won't deadlock
- Resources that are inherently un-sharable will cause undefined behavior if they're shared
- Obvious example: Mutex locks
  - You can't allow sharing of a resource whose entire purpose is to ensure mutual exclusion!

Deadlock Prevention: Hold-And-Wait (7.4.2)

- This holds a little more promise
- Set up a resource request method so that you cannot request resources when you already have resources
  - We can either do this dynamically upon resource request, or require all resources to be requested before the program can even run
- Either way, it’s not possible for a process to be waiting on a resource when it already has one
- This *will* prevent deadlocks
- It will also play havoc with performance whenever resource requests are necessary
- It also has serious starvation issues

Deadlock Prevention: No Resource Preemption (7.4.3)

- If a process:
  - Is holding resources
  - Requests a new resource
  - Is going to have to wait on the new resource
- Then take away all the resources it currently has and make it wait on all of them at once
- Can also do this via pull rather than push – allow a process to steal a resource from another process if that other process is waiting
- This method doesn’t have most of the hold-and-wait prevention method’s problems, but it’s only a partial solution
  - Fine for the printer, or for the disk, or for anything else with persistent state
  - Still doesn’t do anything about mutex locks or semaphores

The Dining Philosophers Problem

- Imagine a bunch of philosophers sitting around a table
  - …with one fork between each philosopher
- Each philosopher:
  - Thinks until they get hungry
  - Tries to pick up both forks and eat
  - Eats until they're full
  - Puts down their forks
  - Repeats
- To eat, each philosopher needs *both* the fork on their left *and* the fork on their right
- How do we make sure the whole table can't deadlock and starve to death?
- Always picking up the left fork works, up until that last philosopher

The Dining Philosophers Problem

- Imagine a bunch of philosophers sitting around a table
  - …with one fork between each philosopher
- Each philosopher:
  - T
  - T
  - E
  - F
  - F
- To … *nd*
  the
- Ho…
  sta…
- Always picking up the left fork works, up until that last philosopher

NUMBER THE FORKS

Deadlock Prevention: Circular Wait (7.4.4)

- Impose a total order on resources
- Each resource is assigned a number
- A process cannot request a resource if it has any resource with a number higher than that resource
- This mostly prevents circular waits by the simple fact that you can't complete the circle
- If this is implemented in user space, you *can* still set up a circular wait by a sufficiently esoteric race condition
- It also still limits resource allocation

Deadlock Avoidance (7.5)

- There are several algorithms to avoid deadlocks, but they all fundamentally do the same thing:
  - Require each process to declare which resources it will require and in what order
  - Do not allow a process to continue unless the resources it requires in the order it requires them cannot result in a deadlock
- This does prevent deadlocks, by never allowing the conditions necessary to create a circular wait to occur
- Unfortunately, it requires a process to know which resources it will require, and in what order, in the first place

Deadlock Detection (7.6)

- Detecting deadlocks that have already occurred is straightforward
  - Look at the set of waiting processes
  - If there's a circular wait, there's a deadlock
- *This is an* O(n^2) *problem*
  - Specifically, it's detecting a cycle in a graph
- If we allow for counted resources as well as binary resources, it's even more complicated than that
- If we're going to do this, biggest question is how often to do it
  - …and the answer is "it depends on how often we expect deadlocks"

Deadlock Recovery (7.7)

- Once we’ve detected a deadlock, what do we do about it?
- Plan A: Terminate every process involved in the deadlock
  - **The good:** It immediately takes care of the deadlock without further computation
  - **The bad:** We just terminated every process involved in the deadlock!
    - That work still needs to get done
    - They’ll all need to be restarted from the beginning

Deadlock Recovery (7.7)

- Plan B: Terminate processes involved in the deadlock until the deadlock goes away
  - Define a *victim selection* algorithm - a cost function to determine the process out of the set that it will cause the least mischief to terminate
  - Some factors can include:
    - Process priority
    - How close the process is to completion
    - How many resources the process has
    - How many more resources the process needs
    - How likely terminating the process is to [help] solve the problem
    - Whether or not the process is interactive
- Less disruptive than plan A, but requires re-running the deadlock detection algorithm after every process is killed

Deadlock Recovery (7.7)

- Plan C: Preempt specific resources from a process
  - Same sort of victim selection as plan B
  - Simplest version is just terminate the process we're going to preempt them from – but that's equivalent to plan B
  - Requires the victim process to be able to *roll back* to the state before it got the resources it's having taken away
- The least disruptive, but most complicated, of the plans

Practical Examples

- There aren’t any
- Except for database systems, most modern operating systems simply ignore deadlocks
- Three reasons:
  - Proper deadlock handling is really hard
  - They don’t happen very often
  - Unlike other resource constraint issues, user-level programmers *do* have a lot of incentive to avoid them

ONE MORE LOOSE END

Priority Inversion (5.6.4)

- Less severe than deadlock, but still a problem
- Most simply expressed by the following:
  - *If a lower-priority process holds a lock needed by a higher-priority process, that higher-priority process's priority is effectively lowered to that of the lower-priority one.*
- The most obvious way for this to cause mischief is for a medium-priority process to preempt the lower-priority one
- Solution (if you're going to attempt a solution): *priority inheritance*
  - Once a process waits for a lock, the process that holds that lock receives the waiting process's priority

END OF UNIT 1