Lecture Topics: 11/12

- Slides from 11/8 set on synchronization
- Deadlock
  - 4 necessary conditions
  - Deadlock prevention
  - Deadlock avoidance
  - Deadlock detection
  - Deadlock recovery

Deadlock

- Process A wants what process B has
- Process B wants what process A has
- Neither can make progress without acquiring the other's resource
- Neither will relinquish its own resource
- No progress possible!

- We saw this with the dining philosophers problem

System Model

- We have *processes* and *resources*
- For a process to utilize a resource, it must follow these steps:
  - Request the resource
    - If the resource is unavailable, block
  - Use the resource
  - Release the resource

Necessary Conditions

- Mutual Exclusion
  - The resource can't be shared
- Hold and Wait
  - Some process holds one resource while waiting for another
- No Preemption
  - Once a process has a resource, it cannot be forced to give it up
- Circular Wait
  - A waits for B, B for C, C for D, D for A

Dealing with Deadlock

- Deadlock Prevention
  - Use a protocol to ensure statically that deadlock is impossible
- Deadlock Avoidance
  - Use a protocol to ensure dynamically that deadlock is impossible
- Deadlock Detection and Recovery
  - Allow deadlock to occur, but notice when it does and try to recover
- Ignore the Problem

Deadlock Prevention

- There are four necessary conditions for deadlock
- Take any one of them away and deadlock is impossible
- Let's attack deadlock by
  - examining each of the conditions
  - considering what would happen if we threw it out

Mutual Exclusion

- This condition says, "There exist resources in the system that can be used by only one process at a time."
- Examples include printer, write access to a file or record, entry into a section of code
- Best not to get rid of this condition
  - some resources are intrinsically nonsharable

Hold and Wait (1/2)

- This condition says, "Some process holds one resource while waiting for another."
- To attack the hold and wait condition:
  - Force a process to acquire all the resources it needs before it does anything; if it can't get them all, get none
- Each philosopher tries to get both chopsticks, but if only one is available, put it down and try again later

Hold and Wait (2/2)

- Resource Utilization may be low
  - If you need resources P and Q for a long time and R only at the end, you still have to hold R's lock the whole time
- Starvation prone
  - If a process needs several popular resources, it may have to wait indefinitely before they are all available at the same time

No Preemption (1/2)

- This condition says, "Once a process has a resource, it will not be forced to give it up."
- To attack the no preemption condition:
  - If a process asks for a resource not currently available, block it but also take away all of its other resources
  - Add the preempted resources to the list of resource the blocked process is waiting for

No Preemption (2/2)

- If philosopher 2 has chopstick 2 and blocks trying to get chopstick 3, take chopstick 2 away
- This strategy works for some resources:
  - CPU state (contents of registers can be spilled to memory)
  - memory (can be spilled to disk)
- But not for others:
  - The printer (at least tricky)

Circular Wait (1/2)

- This condition says, "A is blocked waiting for B, B for C, C for D, and D for A"
- Note that the number of processes is actually arbitrary
- To attack the circular wait condition:
  - Assign each resource a priority
  - Make processes acquire resources in priority order

Circular Wait (2/2)

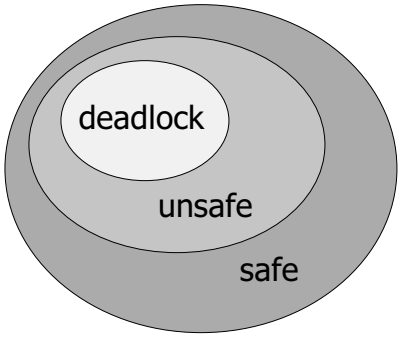
- Dictate that philosophers must pick up low-numbered chopsticks before high-numbered
  - Philosophers 0, 1, 2, and 3 operate as before (right then left)
  - Philosopher 4 switches (left then right)
- This is the most common form of deadlock prevention
- The only problem: sometimes forced to relinquish a resource

Deadlock Avoidance

- Deadlock prevention is sometimes too strict
  - low device utilization
  - reduced system throughput
- If the OS had more information, it could do more sophisticated things to avoid deadlock and keep the system in a safe state

The Banker's Algorithm

- Idea: know what each process *might* ask for
- Only make allocations that leave the system in a *safe* state
- Inefficient



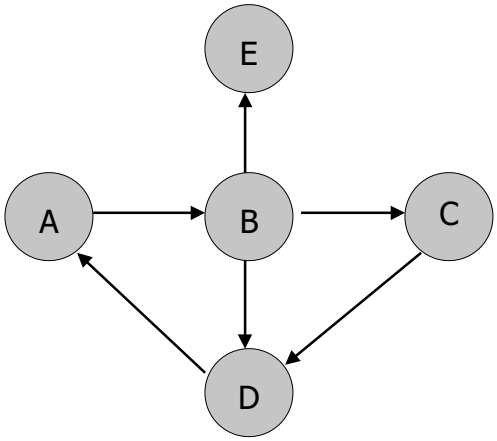
Resource allocation state space

Deadlock Detection

- Build a *wait-for* graph and periodically look for cycles, to find the circular wait condition
- The wait-for graph contains:
  - nodes, corresponding to processes
  - directed edges, corresponding to a resource held by one process and desired by the other

Wait-For Graphs

- A waits for B, B for D, and D for A
  - deadlock!
- How do you write a program to detect this?



Deadlock Recovery

- Once you've discovered deadlock, what do you do about it?
- One option: abort one of the processes to recover from circular wait
  - That process will likely have to start over from scratch (unless it *checkpointed*)
  - Which process should you choose? (Not your supervisor's!)

Resource Preemption

- Another solution is to take a resource away from a process
  - Again, which process should you choose?
  - How can you *roll back* the process to its state before it had the coveted resource?
  - Make sure you don't keep on preempting from the same process: avoid starvation

Ignoring Deadlock

- Perhaps not as silly as it sounds
- Use resource prioritization for heavily used resources, otherwise, don't worry about it
- The mechanisms outlined previously for handling deadlock may be very expensive; if the alternative is to have a forced reboot once a year, that might be acceptable