Deadlock

CSE 410, Spring 2004
Computer Systems

http://www.cs.washington.edu/education/courses/410/04sp/
Readings and References

• Reading
  » Chapter 8, *Operating System Concepts*, Silberschatz, Galvin, and Gagne

• Other References
**Deadlock**

- Circular waiting for resources
  - Task A wants what task B has
  - Task B wants what task A has
- No progress possible!
  - Neither can make progress without the other’s resource
  - Neither will relinquish its own resource

```javascript
... lockOne->Acquire();
...
lockTwo->Acquire();
... lockTwo->Acquire();
lockOne->Acquire();
DEADLOCK!
```
Simple Traffic Gridlock Example
System Model

• There are *tasks* and *resources*

• A task follows these steps to utilize a resource
  » Acquire the resource
    • If the resource is unavailable, block
  » Use the resource
  » Release the resource
Necessary Conditions for Deadlock

- **Mutual Exclusion**
  - The resource can’t be shared

- **Hold and Wait**
  - Task holds one resource while waiting for another

- **No Preemption**
  - If a task 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

\[
\begin{align*}
A & \rightarrow B \\
C & \rightarrow A \\
B & \rightarrow C \\
D & \rightarrow D
\end{align*}
\]
Is Gridlock an Example of Deadlock?

- Mutual Exclusion
  - space-time can only hold one car at a time
- Hold and wait
  - I’m here, and I want to turn left, so watch out
- No preemption
  - cannons are not allowed in cars at this time
- Circular wait
  - blue waiting for red’s space and vice versa
Dealing with Deadlock

- **Deadlock Prevention**
  - Ensure statically that deadlock is impossible

- **Deadlock Avoidance**
  - 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**
  - Let the operator untangle it, that's what they're paid for
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
Condition: Mutual Exclusion

• Usually can't eliminate this condition
  » some resources are intrinsically non-sharable
• Examples include printer, write access to a file or record, entry into a section of code
• However, you can often mitigate this by adding a layer of abstraction
  » For example, use a print spooler, not direct connection to the printer.
Condition: Hold and Wait

• Eliminate partial acquisition of resources
• Task must acquire all the resources it needs before it does anything
  » if it can’t get them all, then it gets none
• Issue: Resource utilization may be low
  » If you need P for a long time and Q only at the end, you still have to hold Q’s lock the whole time
• Issue: Starvation prone
  » May have to wait indefinitely before popular resources are all available at the same time
Condition: No Preemption

• Allow preemption
  » If a process asks for a resource not currently available, block it and take away all of its other resources
  » Add the preempted resources to the list of resources the process is waiting for

• 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:
  » printer - rip off the existing printout and tape it on later?
Condition: Circular Wait

- To attack the circular wait condition:
  - Assign each resource a priority
  - Make processes acquire resources in priority order
- Two processes need the printer and the scanner, both must acquire the printer (higher priority) before the scanner
- This is a common form of deadlock prevention
- The only problem: sometimes forced to relinquish a resource that you thought you had locked up
Deadlock Avoidance

• Deadlock prevention is often 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
  » “If” is a little word, but it packs a big punch
  » predicting all needed resources \textit{a priori} is hard
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 tasks
  - directed edges, corresponding to a resource held by one task and desired by the other.

**Diagram:**
- A waits for B
- B waits for D
- D waits for A
- E

Deadlock!
Deadlock Recovery

• Once you’ve discovered deadlock, what next?
• Terminate one of the tasks to stop circular wait?
  » Task will likely have to start over from scratch
  » Which task should you choose?
• Take a resource away from a task?
  » Again, which task should you choose?
  » How can you roll back the task to the state before it had the coveted resource?
  » Make sure you don’t keep on preempting from the same task: avoid starvation
Ignoring Deadlock

• Not a bad policy for operating systems
• The mechanisms outlined previously for handling deadlock may be expensive
  » if the alternative is to have a forced reboot once a year, that might be acceptable
• However, for thread deadlocks, your users may not be quite so tolerant
  » “the program only locks up once in a while”