Deadlock Avoidance

Introduction
- Resources are generally requested one at a time
- System must decide whether a particular allocation is safe or not
- Is there an algorithm that can always avoid deadlock by making the right choice all the time?

Resource Trajectories
- Most algorithms for deadlock avoidance use the concept of “safe states”
  - Safe state – OS can guarantee that all processes will finish
  - Unsafe state – deadlock is possible, but still might not happen

Graphical example
- Two resources (printer and plotter) and two processes

Graph explanation
- Horizontal axis represents instructions executed by A
- Vertical axis represents instructions executed by B
- The $I_i$ points represent spots where resources are requested or released
- Every point in the graph represents a joint state of the two processes
  - At $p$ neither has executed
  - At $q$ A ceases to execute and $B$ is given the CPU
- All paths must be horizontal or vertical (in a single CPU system)
  - Must always run north or east (time must advance)
o Shaded boxes represent areas that would violate mutual exclusion, so they can never be entered due to policy.

o At \( t \) a decision must be made to allocate the plotter to \( B \) or run \( A \).
  - If \( B \) is allowed to acquire the plotter, the system will eventually deadlock
  - The unsafe state is bounded by \( I_1, I_2, I_5, \) and \( I_6 \).

• Implication – to avoid deadlock, \( B \) should be suspended until \( A \) has requested and released the plotter.
  o How can we do this in practice?

o Safe and unsafe states
  • Assumptions
    • At any instant in time, we have state that consists of the following
      o number of existing resources
      o number of available (unused) resources
      o number of resources currently allocated to each process
      o maximum number each process will eventually need

    • A state is safe if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately

    • Example – since resource type with multiple instances of that resource
      • 10 instances of the resource, with the following states:

        | Has | Max |
        |-----|-----|
        | A   | 3   | 9   |
        | B   | 2   | 4   |
        | C   | 2   | 7   |

        Free: 3

        • This is safe
          o Run \( B \)
Then run C

Unsafe progression
  o Instead of running B, let A request a single additional resource

  B can still run to completion

However, now there is a deadlock
In retrospect, the allocation to A that seems harmless shouldn’t have been made
- Even looking one step into the future wasn’t enough to prevent the problem
- Unsafe doesn’t guarantee deadlock
  - A could release one of the resources before deadlocking.

Banker’s Algorithm – famous algorithm by Dijkstra to solve this problem
- Consider each request as it occurs, and see if granting it leads to a safe state
  - If not, postpone the request until it would be safe
- To see if a state is safe:
  - See if there are enough resources to satisfy one process
  - If so, that process runs to completion and those resources are considered reclaimed and available for others
  - Continue this process until all processes are terminated
    - If this happens, the state is safe and the resource is allocated
- If several processes are eligible for completion, it does not matter which one is chosen
  - After completion, the pool of available resources with either be larger or the same size

Discussion
- In theory this works well. In practice, there are major issues
- Processes rarely know in advance what their maximum resource needs will be
- Number of processes isn’t fixed, and dynamically changes over time
- Resources that were available might go offline
- End result – deadlock avoidance is not particularly practical either

Deadlock Prevention
- Introduction
- Besides ignoring the problem, deadlock detection/recovery and deadlock avoidance have been shown to be impractical
- If we can eliminate one of the necessary conditions for deadlock, we can render them impossible
  - Attacking mutual exclusion
    - We can’t eliminate mutual exclusion entirely without introducing race conditions
    - Consider a printer
      - By spooling printer output and allocating the actual printer to only the printer daemon, we can improve the situation
        - Printer daemon doesn’t request other resources, so no deadlock for the printer possible
      - Question – Should the daemon start printing before all the output is spooled?
        - If so, the output process can cause the printer to idle by not completing its output for several hours
        - As a consequence, most daemons wait for the entire output to print
      - Spooling space is limited, so deadlock is still possible with this decision
        - Two processes each fill up half the available space and can’t continue, so there is deadlock on the disk.
  - Good ideas
    - Avoid assigning resources that aren’t absolutely necessary to assign
    - Minimize the number of processes that can actually claim the resource
  - Attacking Hold and Wait
    - Goal – prevent processes that currently hold resources from waiting for more resources
    - Idea – require all processes to request their resources before starting execution
      - If everything is available, the process runs to completion. If not, nothing is allocated and the process just waits
    - Problems
      - Most processes don’t know their resource allocations in advance
        - If they did, we could use the Banker’s algorithm
      - Resources will not be used efficiently
        - Read data from tape, then analyze if for a week, then write back to the tape
        - The tape would be unusable by other processes that entire time
    - Idea – require processes requesting a resource to first temporarily release all resources it holds, then tries to get everything at once.
Might work for some programs, but many could not possibly (or efficiently) do this.

- Idea – only allow a single resource to be held at a time
- Not practical in many programs

  - Attacking no preemption
    - Some resources simply can’t be preempted easily
    - Virtualizing the resource and using a daemon to control actual access might help
      - Printer spooling (though another deadlock on the disk is possible)
    - Many resources can’t be virtualized – OS tables, etc.

  - Attacking circular wait
    - Idea – Provide a global numbering of all resources

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD-ROM drive

- Rule – processes can request resources whenever they want, but requests must be made in *numerical* order
  - Process with a plotter couldn’t request a scanner

- Logic for multiple processes and resources
  - At every instant, one of the assigned resources will be the highest.
  - The process holding that resource will *never* ask for a resource already assigned
  - It will either finish or request higher numbered resources, which are necessarily available
  - No cycle can be created

- Minor variation – no process can request a resource lower than what it is already holding
  - If process initially requests 9 and 10 then releases them, it can then request resource 1.
    - If effectively started over with relation to deadlock, so this isn’t dangerous

- Problems
  - Finding a suitable numbering to satisfy everybody could be difficult/impossible
  - Increases burden on programmers to know the numbering