Deadlock

OS Lecture 10

UdS/TUKL WS 2015
Deadlock

When is a system *deadlocked*?

» If there exists a set of processes such that every process in the set is *waiting* for a resource held by another process in the set.

» Deadlock is *stable*: since all processes are waiting, the situation will persist.

» Examples: badly ordered `p()` operations on binary semaphores, two processes both waiting for messages from each other
Necessary Preconditions for Deadlock

What is required for deadlock to occur?

1. **Mutually exclusive access**: resources cannot be shared and processes must wait.

2. **No resource preemption**: once granted, access to a resource cannot be revoked.

3. **Hold and wait**: processes can hold resources while they are waiting.

4. **Cycle in the wait-for graph**
What to do about deadlock?

Deadlock is a fundamental problem that cannot be ignored in real-world systems. How to handle it?

1. **Detection**: at runtime, detect when a deadlock has occurred and start some recovery routine.
   - For example, restart all (a subset of the) deadlocked processes.

2. **Prevention**: organize the system such that deadlock is impossible.
   - Both at design time and at runtime (e.g., by following certain locking rules or protocols)
Preventing Deadlocks by Design

How can we reliably avoid deadlocks?

1. Prevent or prohibit one of the necessary (pre-)conditions of deadlock.

2. Predict future resource needs and delay “potentially problematic” requests.

   » Difficult in general-purpose systems...
Possible Avoidance Strategies

Which of these approaches are practical?

1. Don’t allow exclusive access.

2. Always have enough resources available:
   → *(over-)provision for the worst case*

3. Don’t allow processes to wait for resources.

4. Take away already granted resources (resource preemption).

5. Force all-or-nothing allocation semantics
   → *processes must state all needed resources up front*
Ordered Requests

Observation: *edges in the wait-for graph are determined by the order in which resources are requested...*

» So structure code such that cycles are impossible!

» Impose a *strict (i.e., irreflexive) partial order* “<” on the set of all resources

» **Rule**: a resource $R_2$ may be requested while already holding $R_1$ *if and only if* $R_1 < R_2$.

» Finding such a strict partial order requires design-time knowledge.
Banker’s algorithm (by Dijkstra)

1. Each process declares \textit{maximum number of needed instances} of each resource type (e.g., tapes, semaphores, pages, etc.).

2. Track for each process the number of \textit{currently loaned instances}.

3. Define (remaining) \#\textit{needed} = \#\textit{max} - \#\textit{loaned}.

4. When a request for more resources is made, check that granting it results in a \textit{safe state}:
   
   \begin{itemize}
   \item \textit{assume that each process will request \#\textit{needed} resources of each type}
   \item \textit{assume that resources are released only on termination}
   \item \textit{there must exist a feasible sequence of process terminations}
   \end{itemize}
Finding a feasible termination sequence

1. While there are “running” processes:
2. Does there exist a process $P$ such that, for each resource type, $\#\text{needed} \leq \#\text{available}$? If not, return **unsafe**.
3. Otherwise, assume $P$ terminates and releases all resources ($\rightarrow$ update $\#\text{available}$); go to 1.
4. When no more “running” processes remain, return **safe** ($\rightarrow$ a feasible termination sequence has been found).
Priority Ceiling Protocol

For priority-scheduled uniprocessor systems.

» Before runtime, for each resource $R$, define the priority ceiling as the priority of the highest-priority process that will ever request $R$.

» At runtime, define the system ceiling as the maximum of the priority ceilings of all resources currently allocated.

» When a process $P$ requests a resource $R$, the request is granted only if either (i) $P$'s priority is higher than the current system ceiling, or (ii) $P$ was the last process to raise the system ceiling.