Deadlock

COMP3231 Operating Systems
2005 S2

What is a deadlock?
- Permanent blocking of a set of processes that either
  - compete for system resources or
  - communicate with each other (message as resource)
- Resources:
  - preemptable
  - nonpreemptable resources
- Deadlocks involve conflicting needs for nonpreemptable resources by two or more processes
- Deadlocks can occur on many levels in the system

Unfortunately, there is no efficient method to prevent a deadlock in the general case

Let's look at some examples and at the conditions for deadlock

Danger of deadlock in continental driving rules:

Reusable versus consumable resources
- Reusable resource: used by one process at a time and not depleted by that use
- Consumable resource: created (produced) and destroyed (consumed) by a process

Reusable Resources:
- Processes obtain resources that they later release for reuse by other processes
- Examples are processors, I/O channels, main and secondary memory, files, databases, and semaphores
- In case of two processes and two resources, deadlock occurs if each process holds one resource and requests the other
Typical deadlock with reusable resources:

<table>
<thead>
<tr>
<th>Process P</th>
<th>Process Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>Action</td>
</tr>
<tr>
<td>p₀</td>
<td>Request (D)</td>
</tr>
<tr>
<td>p₁</td>
<td>Lock (D)</td>
</tr>
<tr>
<td>p₂</td>
<td>Request (T)</td>
</tr>
<tr>
<td>p₃</td>
<td>Lock (T)</td>
</tr>
<tr>
<td>p₄</td>
<td>Perform function</td>
</tr>
<tr>
<td>p₅</td>
<td>Unlock (D)</td>
</tr>
<tr>
<td>p₆</td>
<td>Unlock (T)</td>
</tr>
</tbody>
</table>

The following sequence leads to a deadlock:

\[ p₀, p₁, q₀, q₁, p₂, q₂ \]

Should this really be the problem of the OS designer?

Another example of deadlock with reusable resources:

- Space is available for allocation of 200K bytes and the following sequence of events occur

\[
\begin{align*}
  & P_1 & P_2 \\
  & \cdots & \cdots \\
  & \text{Request 80kB;} & \text{Request 70kB;} \\
  & \cdots & \cdots \\
  & \text{Request 60kB;} & \text{Request 80kB;} \\
\end{align*}
\]

- Deadlock occurs if both processes progress to their second request
- In this case, the problem can be solved by using virtual memory (this is an example of resource preemption)

Consumable Resources:
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Example of deadlock:
- Deadlock occurs if receive is blocking

\[
\begin{align*}
  & P_1 & P_2 \\
  & \cdots & \cdots \\
  & \text{Receive}(P_1); & \text{Receive}(P_2); \\
  & \cdots & \cdots \\
  & \text{Send}(P_1, M_1); & \text{Send}(P_2, M_2); \\
\end{align*}
\]

**Conditions for Deadlock**

How can we accurately characterise the conditions that lead to a deadlock?

**Necessary conditions for deadlock:**
1. **Mutual exclusion**: only one process may use a resource at a time
2. **Hold-and-wait**: a process holds a resource while awaiting assignment of others
3. **No preemption of resources**: a process that is denied a request must not release the resources it already has
   - When one process requests a resource held by another, the second one is not preempted by the OS
4. **Circular wait**: we have a closed chain of processes, such that each process holds at least one resource needed by the next in the chain, e.g.
STRATEGIES TO DEAL WITH DEADLOCKS

1. The Ostrich Algorithm
2. Prevention
3. Avoidance by careful resource allocation
4. Detection and Recovery: let them occur, detect them and take action

The Ostrich Algorithm:
Stick your head in the sand and pretend there is no problem at all!

- Unix & Windows
- Avoid deadlock in the kernel

DEADLOCK PREVENTION

What is deadlock prevention?
Make it impossible that one of the four conditions for deadlock arise
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Mutual exclusion:
- we can’t generally exclude it
- we can avoid assigning resources when not absolutely necessary
- as few processes as possible should claim the resource

Hold-and-wait:
- Can we require processes to request all resources at once?
- Most processes do not statically know about the resources they need
- Used in some mainframe batch systems
- Wasteful, but works
- Variation: before requesting new resource, temporarily release other resources
No preemption:
Preemption is feasible for some resources (e.g., processor and memory), but not for others (state must be saved and restored)

Circular wait:
- order resources by an index: $R_1, R_2, \ldots$
- requires that resources are always requested in order
- $P_1$ holds $R_i$ and requests $R_j$, and $P_2$ holds $R_j$ and requests $R_i$ is impossible, as it implies $i < j$ and $i > j$
- is sometimes a feasible strategy, but not generally efficient

Deadlock Avoidance

What is deadlock avoidance?:
- We don’t exclude any of the four conditions for deadlock per se
- Instead we decide on a per case basis whether a process is deemed likely to deadlock
- Thus, we have to possess some knowledge about future allocation requests of processes

Generally, we can distinguish two approaches to deadlock avoidance:
- Process initiation denial: we just don’t start a process if it might deadlock
- Resource allocation denial: we deny allocation requests, which are likely to lead to deadlock in the future

Process Initiation Denial

Consider a system of $n$ processes and $m$ types of resources:
- Resource vector: $(R_1, R_2, \ldots, R_n)$
- Available vector: $(V_1, V_2, \ldots, V_n)$
- Matrices:
  - Claim matrix: $C_{ij}$ requirement for process $i$ for resource $j$
  - Allocation matrix: $A_{ij}$ allocation of resource $j$ to process $i$

Example:
We have two processes $P_1$ and $P_2$ and three resources $R_1$, $R_2$ and $R_3$. Each of the three resources can be allocated to only a single process at each point in time
- $P_1$
  - holds $R_1$
  - requires $R_1, R_2$
- $P_2$
  - holds no resource
  - requires $R_2, R_3$
- Resource vector: $(1, 1, 1)$
- Available vector: $(0, 1, 1)$

Claim matrix: $\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$
Allocation matrix: $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$
The following relationships hold:

1. \( R_i = V_i + \sum_{k=1}^{n} A_{ki} \) : all resources are either available or allocated
2. \( C_{kj} \leq R_i \) : no process can hold more than the total amount of resources in the system
3. \( A_{ki} \leq C_{kj} \) : no process is allocated more than it originally claimed to need

Slide 17

Deadlock avoidance policy:

- Start a new process \( P_{n+1} \) only if, for all \( i \),

\[
V_i \geq C_{i,n+1} + \sum_{k=1}^{n} C_{ki}
\]

- Unfortunately, this strategy is very wasteful!
- Assumes all processes make their claims together

Slide 18

Resource Allocation Denial

- At any request of a resource, it is tested whether granting this request bears the potential of deadlock
- The standard algorithm to execute this test is due to Dijkstra and known as the banker’s algorithm

Banker’s algorithm:

- Resource and available vector & claim and allocation matrix as before
- The algorithm passes out resources to processes if it has enough on hand to meet potential future demand
- Whenever we can guarantee that future demand can be met, we are in a safe state
- A request for resources is granted only if the state after the resource is granted is safe

Slide 19

How do we know whether a state is safe?

- A state is safe if there is at least one sequence of resource allocations that does not result in deadlock
- Pick a process whose outstanding resource claim can be met and run it to completion
- Repeat until either all process have completed, or the system locks up

Slide 20

Check that this state is safe:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>R3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

P2 runs to completion:

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Resource Allocation Denial

9

Resource Allocation Denial

10
Disadvantages of the Banker’s algorithm:

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

**Deadlock Detection**

- An alternative to deadlock avoidance is **deadlock detection**
- An alternative to deadlock detection is to **check for cycles**
- We need a request matrix \( Q \) (outstanding requests) instead of the claim matrix
- Consider process completed if outstanding requests are satisfied
- Checks can be made each time a resource is allocated
  - Early deadlock detection
  - Expensive
Algorithm:
Initially, all processes are unmarked
① mark each process with zero-row in Request matrix
② set temporary vector \( W \) to Available vector
③ find \( i \) such that process \( i \) is unmarked, \( Q_{ik} \leq W_k \) for \( 1 \leq k \leq n \)
   - no such process \( \Rightarrow \) terminate
④ mark process \( i \), add row of allocation matrix to \( W \), go to step 3

Slide 25

Recovery:
① Abort all deadlocked processes (most common solution)
② Rollback each deadlocked process to some previously defined checkpoint and restart them (original deadlock may reoccur)
③ Successively abort deadlocked processes until deadlock no longer exists (invoke deadlock detection algorithm each time)
④ Successively preempt some resources from process until deadlock no longer exists
   - a process that has a resource preempted must be rolled back prior to its acquisition

Slide 27

Available Vector

Resource Vector