Chapter 3

Deadlocks

3.1. Resource
3.2. Introduction to deadlocks
3.3. The ostrich algorithm
3.4. Deadlock detection and recovery
3.5. Deadlock avoidance
3.6. Deadlock prevention
3.7. Other issues

Learning Outcomes

- Understand what deadlock is and how it can occur when giving mutually exclusive access to multiple resources.
- Understand several approaches to mitigating the issue of deadlock in operating systems.
  - Including deadlock detection and recovery, deadlock avoidance, and deadlock prevention.

Resources

- Examples of computer resources
  - printers
  - tape drives
  - Tables in a database
- Processes need access to resources in reasonable order
- Suppose a process holds resource A and requests resource B
  - at same time another process holds B and requests A
  - both are blocked and remain so

Example Resource usage

```c
semaphore res_1, res_2;
void proc_A() {
    down(&res_1);
    down(&res_2);
    use_both_res();
    up(&res_2);
    up(&res_1);
}
void proc_B() {
    down(&res_2);
    down(&res_1);
    use_both_res();
    up(&res_1);
    up(&res_2);
}
```

Resources

- Deadlocks occur when ...
  - processes are granted exclusive access to devices
  - we refer to these devices generally as resources
- Preemptable resources
  - can be taken away from a process with no ill effects
- Nonpreemptable resources
  - will cause the process to fail if taken away

Resources

- Sequence of events required to use a resource
  1. request the resource
  2. use the resource
  3. release the resource
- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code
Introduction to Deadlocks

- Formal definition: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- Usually the event is release of a currently held resource.
- None of the processes can …
  - run
  - release resources
  - be awakened

Four Conditions for Deadlock

1. Mutual exclusion condition
   - each resource assigned to 1 process or is available
2. Hold and wait condition
   - process holding resources can request additional
3. No preemption condition
   - previously granted resources cannot forcibly taken away
4. Circular wait condition
   - must be a circular chain of 2 or more processes
   - each is waiting for resource held by next member of the chain

Deadlock Modeling

- Modeled with directed graphs

Deadlock

Strategies for dealing with Deadlocks

1. just ignore the problem altogether
2. detection and recovery
3. dynamic avoidance
   - careful resource allocation
4. prevention
   - negating one of the four necessary conditions

Deadlock Modeling

How deadlock occurs

How deadlock can be avoided
Approach 1: The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
    - Example of “cost”, only one process runs at a time
- UNIX and Windows takes this approach for some of the more complex resources to manage
- It’s a trade off between
  - Convenience (engineering approach)
  - Correctness (mathematical approach)

Approach 2: Detection and Recovery

- Need a method to determine if a system is deadlocked.
- Assuming deadlocked is detected, we need a method of recovery to restore progress to the system.

Detection with One Resource of Each Type

- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

What about resources with multiple units?

- We need an approach for dealing with resources that consist of more than a single unit.

Detection with Multiple Resources of Each Type

Note the following invariant

Sum of current resource allocation + resources available = resources that exist

\[ \sum_{i=1}^{n} C_{ij} + A_j = E_j \]
Detection with Multiple Resources of Each Type

\[ E = \begin{bmatrix} 4 & 2 & 3 \end{bmatrix}, \quad A = \begin{bmatrix} 2 & 1 & 0 \end{bmatrix} \]

Current allocation matrix:

\[ C = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \]

Request matrix:

\[ R = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \]

An example for the deadlock detection algorithm

Detection Algorithm

1. Look for an unmarked process \( P_i \), for which the \( i \)-th row of \( R \) is less than or equal to \( A \).
2. If found, add the \( i \)-th row of \( C \) to \( A \), and mark \( P_i \). Go to step 1.
3. If no such process exists, terminate. Remaining processes are deadlocked.

Example Deadlock Detection

\[ E = \begin{bmatrix} 4 & 2 & 3 & 1 \end{bmatrix}, \quad A = \begin{bmatrix} 2 & 1 & 0 & 0 \end{bmatrix} \]

\[ C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}, \quad R = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \]

Example Deadlock Detection

\[ E = \begin{bmatrix} 4 & 2 & 3 & 1 \end{bmatrix}, \quad A = \begin{bmatrix} 2 & 1 & 0 & 0 \end{bmatrix} \]

\[ C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}, \quad R = \begin{bmatrix} 2 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \]
**Example Deadlock Detection**

\[ E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}, \quad A = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix} \]

\[ C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}, \quad \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \]

\[ R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \]

**Example Deadlock Detection**

\[ E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}, \quad A = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix} \]

\[ C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}, \quad \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \]

\[ R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \]

**Example Deadlock Detection**

- Algorithm terminates with no unmarked processes
  - We have no dead lock

**Example 2: Deadlock Detection**

- Suppose, P3 needs a CD-ROM as well as 2 Tapes and a Plotter

\[ E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}, \quad A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix} \]

\[ C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}, \quad \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix} \]
Recovery from Deadlock

- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart the process if it is found deadlocked

Recovery from Deadlock

- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning

Approach 3
Deadlock Avoidance

- Instead of detecting deadlock, can we simply avoid it?
  - YES, but only if enough information is available in advance.
  - Maximum number of each resource required

Deadlock Avoidance

Resource Trajectories

Two process resource trajectories

Safe and Unsafe States

- A state is safe if
  - The system is not deadlocked
  - There exists a scheduling order that results in every process running to completion, even if they all request their maximum resources immediately

Safe and Unsafe States

Note: We have 10 units of the resource

<table>
<thead>
<tr>
<th>Process</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>3</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Free 3</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Free 1</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>Free 5</td>
<td>C</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Free 6</td>
<td>A</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>Free 7</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

Demonstration that the state in (a) is safe
Safe and Unsafe States

A requests one extra unit resulting in (b)

<table>
<thead>
<tr>
<th>Has Max</th>
<th>Has Max</th>
<th>Has Max</th>
<th>Has Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 3 2</td>
<td>A 4 2</td>
<td>A 4 5</td>
<td>A 4 9</td>
</tr>
<tr>
<td>B 2 4</td>
<td>B 2 4</td>
<td>B 4 4</td>
<td>B 4 4</td>
</tr>
<tr>
<td>C 2 7</td>
<td>C 2 7</td>
<td>C 2 7</td>
<td>C 2 7</td>
</tr>
</tbody>
</table>

Demonstration that the state in b is not safe

Safe and Unsafe State

- Unsafe states are not necessarily deadlocked
  - With a lucky sequence, all process may complete
  - However, we cannot guarantee that they will complete (not deadlock)
- Safe states guarantee we will eventually complete all processes
- Deadlock avoidance algorithm
  - Only grant requests that result in safe states

Bankers Algorithm

- Modelled on a Banker with Customers
  - The banker has a limited amount of money to loan customers
    - Limited number of resources
  - Each customer can borrow money up to the customer's credit limit
    - Maximum number of resources required
- Basic Idea
  - Keep the bank in a safe state
  - So all customers are happy even if they all request to borrow up to their credit limit at the same time.
  - Customers wishing to borrow such that the bank would enter an unsafe state must wait until somebody else repays their loan such that the transaction becomes safe.

Banker's Algorithm for a Single Resource

<table>
<thead>
<tr>
<th>Has Max</th>
<th>Has Max</th>
<th>Has Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 0 6</td>
<td>A 1 6</td>
<td>A 1 6</td>
</tr>
<tr>
<td>B 0 5</td>
<td>B 1 5</td>
<td>B 2 5</td>
</tr>
<tr>
<td>C 0 4</td>
<td>C 2 4</td>
<td>C 2 4</td>
</tr>
<tr>
<td>D 0 7</td>
<td>D 4 7</td>
<td>D 4 7</td>
</tr>
</tbody>
</table>

- Three resource allocation states
  - safe
  - safe
  - unsafe

Banker's Algorithm for Multiple Resources

Example of banker's algorithm with multiple resources

Should we allow a request by B & E for 1 scanner to succeed??

Bankers Algorithm is used rarely in practice

- It is difficult (sometimes impossible) to know in advance
  - the resources a process will require
  - the number of processes in a dynamic system
Approach 4: Deadlock Prevention

- Resource allocation rules prevent deadlock by prevent one of the four conditions required for deadlock from occurring
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular Wait

Approach 4
Deadlock Prevention

Attacking the Mutual Exclusion Condition

- Not feasible in general
  - Some devices/resource are intrinsically not shareable.

Attacking the Hold and Wait Condition

- Require processes to request resources before starting
  - a process never has to wait for what it needs
- Issues
  - may not know required resources at start of run
  - not always possible
  - also ties up resources other processes could be using
- Variations:
  - process must give up all resources if it would block hold a resource
  - then request all immediately needed
  - prone to starvation

Attacking the No Preemption Condition

- This is not a viable option
  - Consider a process given the printer
    - halfway through its job
    - now forcibly take away printer
    - !?!

Attacking the Circular Wait Condition

- Numerically ordered resources
  - The displayed deadlock cannot happen
    - If A requires 1, it must acquire it before acquiring 2
    - Note: If B has 1, all higher numbered resources must be free or held by processes who doesn’t need 1
  - Resources ordering is a common technique in practice!!!!!
Summary of approaches to deadlock prevention

<table>
<thead>
<tr>
<th>Condition</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual Exclusion</td>
<td>Not feasible</td>
</tr>
<tr>
<td>Hold and Wait</td>
<td>Request resources initially</td>
</tr>
<tr>
<td>No Preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular Wait</td>
<td>Order resources</td>
</tr>
</tbody>
</table>

Nonresource Deadlocks

- Possible for two processes to deadlock
  - each is waiting for the other to do some task
- Can happen with semaphores
  - each process required to do a `down()` on two semaphores (`mutex` and another)
  - if done in wrong order, deadlock results

Starvation

- Starvation is where the overall system makes progress, but one or more processes never make progress.
  - Example: An algorithm to allocate a resource may be to give to shortest job first
  - Works great for multiple short jobs in a system
  - May cause long job to be postponed indefinitely, even though not blocked
- Solution:
  - First-come, first-serve policy