Operating Systems

Lecture 4: Deadlocks

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Deadlocks

Occur among processes who need to acquire resources in order to progress
Resources

• Anything that must be acquired, used, and released over the course of time.
• Hardware or software resources
• Preemptable and Nonpreemtatable resources:
  – Preemptable: can be taken away from the process with no ill-effect
  – Nonpreemptable: cannot be taken away from the process without causing the computation to fail
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources();
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources();
    up(&resource_2);
    up(&resource_1);
}
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources();
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources();
    up(&resource_2);
    up(&resource_1);
}

semaphe resource_1;
semaphe resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources();
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources();
    up(&resource_1);
    up(&resource_2);
}
So ...

- **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.

- **Assumptions**
  - If a process is denied a resource, it is put to sleep
  - Only single-thread processes
  - No interrupts possible to wake up a blocked process
Conditions for Resource Deadlocks

1. Each resource is either currently assigned to exactly one process or is available.
2. Processes currently holding resources that were granted earlier can request new resources.
3. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
Process A is holding resource R
Resource Allocation Graph

Process B is requesting resource S
Resource Allocation Graph

Deadlock!
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
deadlock

(a) Request R
Request S
Release R
Release S

(b) Request S
Request T
Release S
Release T

(c) Request T
Request R
Release T
Release R

(d) A
B
C
R
S
T

(e) A
B
C
R
S
T

(f) A
B
C
R
S
T

(g) A
B
C
R
S
T

(h) A
B
C
R
S
T

(i) A
B
C
R
S
T

(j) A
B
C
R
S
T

(k) A
B
C
R
S
T
A
Request R
Request S
Release R
Release S

B
Request S
Request T
Release S
Release T

C
Request T
Request R
Release T
Release R

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
no deadlock

(k)

(l)

(m)

(n)

(o)

(p)

(q)
How to Deal with Deadlocks

1. Just ignore the problem!
2. Let deadlocks occur, detect them, and take action
3. Dynamic avoidance by careful resource allocation
4. Prevention, by structurally negating one of the four required conditions.
The Ostrich Algorithm
Deadlock Detection and Recovery

• The system does not attempt to prevent deadlocks.
• It tries to detect it when it happens.
• Then it takes some actions to recover
• Several issues here:
  – Deadlock detection with one resource of each type
  – Deadlock detection with multiple resources of each type
  – Recovery from deadlock
Deadlock Detection:
One Resource of Each Type

• Construct a resource graph
• If it contains one or more cycles, a deadlock exists
Formal Algorithm to Detect Cycles in the Allocation Graph

For Each node \( N \) in the graph do:

1. Initialize \( L \) to empty list and designate all arcs as unmarked
2. Add the current node to end of \( L \). If the node appears in \( L \) twice then we have a cycle and the algorithm terminates
3. From the given node pick any unmarked outgoing arc. If none is available go to 5.
4. Pick an outgoing arc at random and mark it. Then follow it to the new current node and go to 2.
5. If the node is the initial node then no cycles and the algorithm terminates. Otherwise, we are in dead end. Remove that node and go back to the previous one. Go to 2.
Deadlock Detection:
Multiple Resources of Each Type

n processes and m resource types

Resources in existence
(E₁, E₂, E₃, É, Eₘ)

Current allocation matrix

\[
\begin{bmatrix}
C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\
C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm}
\end{bmatrix}
\]

Row n is current allocation to process n

Resources available
(A₁, A₂, A₃, É, Aₘ)

Request matrix

\[
\begin{bmatrix}
R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\
R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm}
\end{bmatrix}
\]

Row 2 is what process 2 needs
When to Check for Deadlocks?

- Check every time a resource request is made
- Check every k minutes
- When CPU utilization has dropped below a threshold
Recovery from Deadlock

• We have detected a deadlock ... What next?

• We have some options:
  – Recovery through preemption
  – Recovery through rollback
  – Recovery through killing processes
Recovery from Deadlock: Through Preemption

• Temporary take a resource away from its owner and give it to another process
• Manual intervention may be required (e.g. in case of printer)
• Highly dependent on the nature of the resource.
• Recovering this way is frequently impossible.
Recovery from Deadlock: Through Rollback

• Have processes **checkpointed** periodically

• Checkpoint of a process: its **state** is written to a file so that it can be restarted later

• In case of deadlock, a process that owns a needed resource is rolled back to the point before it acquired that resource
Recovery from Deadlock: Through Killing Processes

• Kill a process in the cycle.
• Can be repeated (i.e. kill other processes) till deadlock is resolved
• The victim can also be a process NOT in the cycle
Deadlock Avoidance

• In most systems, resources are requested one at a time.
• Resource is granted only if it is **safe** to do so
Safe and Unsafe States

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

• An **unsafe** state is NOT a deadlock state.
### Safe and Unsafe States

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

Free: 3 (a)

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

Free: 1 (b)

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 5 (c)

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Free: 0 (d)

<table>
<thead>
<tr>
<th></th>
<th>Has</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Free: 7 (e)
# Safe and Unsafe States

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

Free: 3

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

Free: 2

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

Free: 0

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

Free: 4
The Banker's Algorithm

• Dijkstra 1965
• Checks if granting the request leads to an unsafe state
• If it does, the request is denied.
The Banker’s Algorithm: The Main Idea

- The algorithm checks to see if it has enough resources to satisfy some customers.
- If so, the process closest to the limit is assumed to be done and resources are back, and so on.
- If all loans (resources) can eventually be repaid, the state is safe.
The Banker’s Algorithm: Example (single resource type)

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

Free: 10

(a)

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

Free: 2

(b)

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

Free: 1

(c)
The Banker’s Algorithm: Example (multiple resources)

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Printers</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Resources assigned

<table>
<thead>
<tr>
<th>Process</th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Printers</th>
<th>CD ROMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Resources still needed

E = (6342)  
P = (5322)  
A = (1020)
The Banker’s Algorithm

• Very nice theoretically
• Practically useless!
  – Processes rarely know in advance what their maximum resource needs will be.
  – The number of processes is not fixed.
  – Resources can suddenly vanish.
Deadlock Prevention

• Deadlock avoidance is essentially impossible.

• If we can ensure that at least one of the four conditions of the deadlock is never satisfied, then deadlocks will be structurally impossible.
Deadlock Prevention: Attacking the Mutual Exclusion

• Can be done for some resources (e.g. the printer) but not all.
• Spooling
• Words of wisdom:
  – Avoid assigning a resource when that is not absolutely necessary.
  – Try to make sure that as few processes as possible may actually claim the resource
Deadlock Prevention: Attacking the Hold and Wait Condition

• Prevent processes holding resources from waiting for more resources
• This requires all processes to request all their resources before starting execution
• A different strategy: require a process requesting a resource to first temporarily release all the resources it currently holds. Then tries to get everything it needs all at once
Deadlock Prevention: Attacking No Preemption Condition

• Virtualizing some resources can be a good strategy (e.g. virtualizing a printer)
• Not all resources can be virtualized (e.g. records in a database)
Deadlock Prevention: The circular Wait Condition

• Method 1: Have a rule saying that a process is entitled only to a single resource at a moment.

• Method 2:
  – Provide a global numbering of all resources.
  – A process can request resources whenever they want to, but all requests must be done in numerical order
  – With this rule, resource allocation graph can never have cycles.
# Deadlock Prevention: Summary

<table>
<thead>
<tr>
<th>Condition</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual exclusion</td>
<td>Spool everything</td>
</tr>
<tr>
<td>Hold and wait</td>
<td>Request all resources initially</td>
</tr>
<tr>
<td>No preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular wait</td>
<td>Order resources numerically</td>
</tr>
</tbody>
</table>
Conclusions

• Deadlocks can occur on hardware/software resources

• OS need to be able to:
  – Detect deadlocks
  – Deal with them when detected
  – Try to avoid them if possible

• We are done with Chapter 6

• Skim 6.7 and enjoy the rest!