Deadlock
Resources & Deadlocks

Processes need access to resources in order to make progress

Examples of computer resources
- printers
- disk drives
- kernel data structures (scheduling queues ...)
- locks/semaphores to protect critical sections

Suppose a process holds resource A and requests resource B at the same time another process holds B and requests A both are blocked and remain so ... this is deadlock
Resource Usage Model

Sequence of events required to use a resource
- request the resource (eg. acquire mutex)
- use the resource
- release the resource (eg. release mutex)

Must wait if request is denied
- block
- busy wait
- fail with error code
Preemptable Resources

Preemptable resources
- Can be taken away with no ill effects

Nonpreemptable resources
- Will cause the holding process to fail if taken away
- May corrupt the resource itself

Deadlocks occur when processes are granted exclusive access to non-preemptable resources and wait when the resource is not available
Definition of Deadlock

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 the release of a currently held resource.

None of the processes can ...
- Be awakened
- Run
- Release its resources
Deadlock Conditions

A deadlock situation can occur if and only if the following conditions hold simultaneously:

- **Mutual exclusion** condition – resource assigned to one process only
- **Hold and wait** condition – processes can get more than one resource
- **No preemption** condition
- **Circular wait** condition – chain of two or more processes (must be waiting for resource from next one in chain)
Examples of Deadlock
Resource Acquisition Scenarios

Thread A:

acquire (resource_1)
use resource_1
release (resource_1)

Example:

```go
var r1_mutex: Mutex
...
rl_mutex.Lock()
Use resource_1
rl_mutex.Unlock()
```
Resource Acquisition Scenarios

Thread A:

acquire (resource_1)
use resource_1
release (resource_1)

Another Example:

var r1_sem: Semaphore
r1_sem.Up()
...
r1_sem.Down()
Use resource_1
r1_sem.Up()
Resource Acquisition Scenarios

**Thread A:**
- acquire (resource_1)
- use resource_1
- release (resource_1)

**Thread B:**
- acquire (resource_2)
- use resource_2
- release (resource_2)
Resource Acquisition Scenarios

**Thread A:**
- acquire (resource_1)
- use resource_1
- release (resource_1)

**Thread B:**
- acquire (resource_2)
- use resource_2
- release (resource_2)

No deadlock can occur here!
Resource Acquisition Scenarios

\textit{Thread A:}

- acquire (resource\_1)
- acquire (resource\_2)
- use resources 1 & 2
- release (resource\_2)
- release (resource\_1)

\textit{Thread B:}

- acquire (resource\_1)
- acquire (resource\_2)
- use resources 1 & 2
- release (resource\_2)
- release (resource\_1)
Resource Acquisition Scenarios

Thread A:
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

Thread B:
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

No deadlock can occur here!
Resource Acquisition Scenarios

**Thread A:**
- `acquire (resource_1)`
- `use resources 1`
- `release (resource_1)`
- `acquire (resource_2)`
- `use resource 2`
- `release (resource_2)`

**Thread B:**
- `acquire (resource_2)`
- `use resources 2`
- `release (resource_2)`
- `acquire (resource_1)`
- `use resource 1`
- `release (resource_1)`
### Resource Acquisition Scenarios

**Thread A:**
- `acquire (resource_1)`
- `use resources 1`
- `release (resource_1)`
- `acquire (resource_2)`
- `use resource 2`
- `release (resource_2)`

**Thread B:**
- `acquire (resource_2)`
- `use resources 2`
- `release (resource_2)`
- `acquire (resource_1)`
- `use resource 1`
- `release (resource_1)`

*No deadlock can occur here!*
Resource Acquisition Scenarios

**Thread A:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

**Thread B:**
- acquire (resource_2)
- acquire (resource_1)
- use resources 1 & 2
- release (resource_1)
- release (resource_2)
Resource Acquisition Scenarios

**Thread A:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

**Thread B:**
- acquire (resource_2)
- acquire (resource_1)
- use resources 1 & 2
- release (resource_1)
- release (resource_2)

*Deadlock is possible!*
Dealing With Deadlock

1. Ignore the problem
2. Detect it and recover from it
3. Dynamically avoid is via careful resource allocation
4. Prevent it by attacking one of the four necessary conditions
Deadlock Detection

Let the problem happen, then recover
- How do you know it happened?

Do a depth-first-search on a resource allocation graph
Resource Allocation Graphs

Process/Thread

Resource

A

R
Resource Allocation Graphs

Process/Thread

Resource

A

“is held by”
Resource Allocation Graphs

- **Process/Thread**: A
- **Resource**: S
- **Resource**: R
- **“is requesting”**: "is requesting"
Resource Allocation Graphs

A → S → B

R ← S ← B
Resource Allocation Graphs

Deadlock
Resource Allocation Graphs

Deadlock = a cycle in the graph
Deadlock Detection

Do a depth-first-search on the resource allocation graph

(a)  

(b)
Deadlock Detection

Do a depth-first-search on the resource allocation graph
Deadlock Detection

Do a depth-first-search on the resource allocation graph

(a)

(b)
Deadlock Detection

Do a depth-first-search on the resource allocation graph

(a) (b)
Deadlock Detection

Do a depth-first-search on the resource allocation graph

(a) Deadlock!
Multiple Instances of a Resource

Some resources have only one instance
- i.e. a lock or a printer
- Only one thread at a time may hold the resource

Some resources have several instances
- i.e. Page frames in memory
- All units are considered equal; any one will do
Multiple Instances of a Resource

**Theorem:** *If a graph does not contain a cycle then no processes are deadlocked*

- A cycle in a RAG is a **necessary** condition for deadlock
- Is it a **sufficient** condition?
Multiple Instances of a Resource
Deadlock Detection Issues

How often should the algorithm run?
- On every resource request?
- Periodically?
- When CPU utilization is low?
- When we suspect deadlock because some thread has been asleep for a long period of time?
Recovery From Deadlock

If we detect deadlock, what should be done to recover?
- Abort deadlocked processes and reclaim resources
- Abort one process at a time until deadlock cycle is eliminated

Where to start?
- Lowest priority process?
- Shortest running process?
- Process with fewest resources held?
- Batch processes before interactive processes?
- Minimize number of processes to be terminated?
Deadlock Recovery

How do we prevent resource corruption
- For example, shared variables protected by a lock?

Recovery through preemption and rollback
- Save state periodically (ie. at start of critical section)
- Take a checkpoint of memory
- Start computation again from checkpoint
- Can also make long-lived computation systems resilient
Deadlock Avoidance

Detection – *optimistic* approach
- Allocate resources
- Break system to fix the problem if necessary

Avoidance – *pessimistic* approach
- Don’t allocate resource if it may lead to deadlock
- If a process requests a resource make it wait until you are sure it’s OK

Which one to use depends upon the application and how easy is it to recover from deadlock!
Deadlock Avoidance
Deadlock Avoidance

Process A

Requests Printer
Requests CD-RW
Releases Printer
Releases CD-RW
Deadlock Avoidance
Deadlock Avoidance

Process B

- $t_w$
- $t_x$
- $t_y$
- $t_z$

Requests CD-RW

Requests Printer

Releases CD-RW

Releases Printer
Deadlock Avoidance

Process B

Process A

tz
ty
tx
tw
t1
t2
t3
t4

time
Deadlock Avoidance

Both processes hold CD-RW
Deadlock Avoidance

Both processes hold Printer
Deadlock Avoidance

Process A

Process B

t_w, t_x, t_y, t_z

Forbidden Zone

Forbidden Zone

Deadlock Avoidance
Deadlock Avoidance

Trajectory showing system progress
Deadlock Avoidance

B makes progress, A is not running

B makes progress, A is not running
Deadlock Avoidance

Process B

Process A

\( t_1 \) \( t_2 \) \( t_3 \) \( t_4 \)

B requests the CD-RW
Deadlock Avoidance

Request is granted
Deadlock Avoidance

A runs & makes a request for printer
Deadlock Avoidance

Request is granted; A proceeds
Deadlock Avoidance

B runs & requests the printer... MUST WAIT!
Deadlock Avoidance

Process A

Process B

A runs & requests the CD-RW
Deadlock Avoidance

Process A

Process B

A...
holds printer
requests CD-RW

B...
holds CD-RW
requests printer
Deadlock Avoidance

Process A

A...
holds printer
requests CD-RW

B...
holds CD-RW
requests printer

DEADLOCK!
Deadlock Avoidance

A danger occurred here.

Should the OS give A the printer, or make it wait???
Deadlock Avoidance

This area is “unsafe.”
Deadlock Avoidance

Within the “unsafe” area, deadlock is inevitable. We don’t want to enter this area. The OS should make A wait at this point!
Deadlock Avoidance

Process B

- $t_w$
- $t_x$
- $t_y$
- $t_z$

Process A

- $t_1$
- $t_2$
- $t_3$
- $t_4$

**Time**

- B requests the printer,
- B releases CD-RW,
- B releases printer,
- then A runs to completion!
Safe States

The current state: which processes hold which resources

A “safe” state:
- No deadlock, and
- There is some scheduling order in which every process can run to completion even if all of them request their maximum number of units immediately

The Banker’s Algorithm:

Goal: Avoid unsafe states!!!

*When a process requests more units, should the system grant the request or make it wait?*
Avoidance - Multiple Resources

Total resource vector

Resources in existence
\((E_1, E_2, E_3, \ldots, E_m)\)

Current allocation matrix

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

Row \(n\) is current allocation to process \(n\)

Available resource vector

Resources available
\((A_1, A_2, A_3, \ldots, A_m)\)

Maximum Request Vector

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

Row 2 is what process 2 might need

Note: These are the max. possible requests, which we assume are known ahead of time!
Banker’s Algorithm

Look for a row, \( R \), whose unmet resource needs are all smaller than or equal to \( A \). If no such row exists, the system will eventually deadlock since no process can run to completion.

Assume the process of the row chosen requests all the resources that it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to \( A \) vector.

Repeat steps 1 and 2, until either all process are marked terminated, in which case the initial state was safe, or until deadlock occurs, in which case it was not.
Avoidance - Multiple Resources

**Total resource vector**
Resources in existence
\( (E_1, E_2, E_3, \ldots, E_m) \)

**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 \( n \)

**Available resource vector**
Resources available
\( (A_1, A_2, A_3, \ldots, A_m) \)

**Maximum Request Vector**

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

Row 2 is what process 2 might need

*Run algorithm on every resource request!*
## Avoidance - Multiple Resources

### Current allocation matrix

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

### Max request matrix

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

### Request matrix

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

### Resource matrix \((E)\)

\[
E = \begin{bmatrix}
4 & 2 & 3 & 1
\end{bmatrix}
\]
Avoidance - Multiple Resources

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

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

Current allocation matrix

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

Max request matrix

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 \\
\end{bmatrix}
\]
# Avoidance - Multiple Resources

Matrix $E$ represents the current allocation:

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Matrix $A$ represents the maximum request:

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

The current allocation matrix $C$ and the max request matrix $R$ are:

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$
Avoidance - Multiple Resources

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

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

Max request matrix
\[ R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \]
## Avoidance - Multiple Resources

<table>
<thead>
<tr>
<th></th>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E</strong></td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Current allocation matrix**

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

**Max request matrix**

\[
R = \begin{bmatrix}
2 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0
\end{bmatrix}
\]
## Avoidance - Multiple Resources

### Max request matrix

<table>
<thead>
<tr>
<th>Tape drives</th>
<th>Plotters</th>
<th>Scanners</th>
<th>CD Roms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Current allocation matrix

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

### Max request matrix

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

Deadlock avoidance may be impossible because you don’t know in advance what resources a process will need!

Alternative approach *Deadlock Prevention*
- Make deadlock impossible
- Attack one of the four conditions that are necessary for deadlock to be possible
Deadlock Prevention

Conditions necessary for deadlock

- Mutual exclusion condition
- Hold and wait condition
- No preemption condition
- Circular wait condition
Deadlock Prevention

Attacking mutual exclusion?
- Some resource types resource could be corrupted
- May be OK if a resource can be partitioned

Attacking no preemption?
- Some resources could be left in an inconsistent state
- May work with support of checkpointing and rollback of idempotent operations
Deadlock Prevention

Attacking hold and wait?

- Require processes to request all resources before they start
- Processes may not know what they need ahead of time
- When problems occur a process must release all its resources and start again
Deadlock Prevention

Attacking circular waiting?

- Number each of the resources
- Require each process to acquire lower numbered resources before higher numbered resources
- More precisely: A process is not allowed to request a resource whose number is lower than the highest numbered resource it currently holds
Example

**Thread A:**
- acquire (resource_1)
- acquire (resource_2)
- use resources 1 & 2
- release (resource_2)
- release (resource_1)

**Thread B:**
- acquire (resource_2)
- acquire (resource_1)
- use resources 1 & 2
- release (resource_1)
- release (resource_2)

Assume that resources are ordered:
1. Resource_1
2. Resource_2
3. ...etc...
Example

Assume that resources are ordered:

1. Resource_1
2. Resource_2
3. ...etc...

Thread B violates the ordering!
Resource Ordering

Assume deadlock has occurred.
Process A
- holds X
- requests Y
Process B
- holds Y
- requests Z
Process C
- holds Z
- requests X
Resource Ordering

Assume deadlock has occurred.

Process A
- holds X
- requests Y

Process B
- holds Y
- requests Z

Process C
- holds Z
- requests X
Resource Ordering

Assume deadlock has occurred.

Process A
- holds X
- requests Y

Process B
- holds Y
- requests Z

Process C
- holds Z
- requests X

\( X < Y \)

\( Y < Z \)
Resource Ordering

Assume deadlock has occurred.

Process A
- holds X
- requests Y

Process B
- holds Y
- requests Z

Process C
- holds Z
- requests X

\[ X < Y \]
\[ Y < Z \]
\[ Z < X \]
Resource Ordering

Assume deadlock has occurred.

Process A
- holds X
- requests Y

Process B
- holds Y
- requests Z

Process C
- holds Z
- requests X

This is impossible!
Resource Ordering

The chief problem:

*It may be hard to come up with an acceptable ordering of resources!*
Starvation

Starvation and deadlock are different

- With deadlock – no work is being accomplished for the processes that are deadlocked, because processes are waiting for each other. Once present, it will not go away.

- With starvation – work (progress) is getting done, however, a particular set of processes may not be getting any work done because they cannot obtain the resource they need
Quiz

What is deadlock?
What conditions must hold for deadlock to be possible?
What are the main approaches for dealing with deadlock?
Why does resource ordering help?