Deadlocks
(Chapter 3, Tanenbaum)
What is a **deadlock**?

**Deadlock** is defined as the *permanent* blocking of a set of processes that compete for system resources, including database records or communication lines.

Unlike other problems in multiprogramming systems, there is **no** efficient solution to the deadlock problem in the general case.

**Deadlock prevention, by design**, is the “best” solution.

Deadlock occurs when a set of processes are in a wait state, because each process is waiting for a resource that is held by some other waiting process. **Therefore**, all deadlocks involve conflicting resource needs by two or more processes.
Classification of resources—I

Two general categories of resources can be distinguished:

• **Reusable**: something that can be safely used by one process at a time and is not depleted by that use. Processes obtain resources that they later release for reuse by others.
  
  E.g., CPU, memory, specific I/O devices, or files.

• **Consumable**: these can be created and destroyed. When a resource is acquired by a process, the resource ceases to exist.
  
  E.g., interrupts, signals, or messages.
Classification of resources—II

One other taxonomy again identifies two types of resources:

- **Preemptable**: these can be taken away from the process owning it with no ill effects (needs save/restore).
  
  E.g., memory or CPU.

- **Non-preemptable**: cannot be taken away from its current owner without causing the computation to fail.
  
  E.g., printer or floppy disk.

Deadlocks occur when sharing *reusable* and *non-preemptable* resources.
Four conditions that must hold for a deadlock to be possible:

- **Mutual exclusion**: processes require exclusive control of its resources (not sharing).
- **Hold and wait**: process may wait for a resource while holding others.
- **No preemption**: process will not give up a resource until it is finished with it. Also, **processes are irreversible**: unable to reset to an earlier state where resources not held.
- **Circular wait**: each process in the chain holds a resource requested by another
An example

C1
refuses to share intersection
mutual exclusion

C2
holds the intersection
hold and wait

C3
back-up not possible
no rollback

C4
Circular wait

C3
will not give-up intersection
no preemption
Discussion

If any one of the necessary conditions is prevented a deadlock need not occur. For example:

- Systems with only simultaneously shared resources cannot deadlock.
  - Negates *mutual exclusion*.
- Systems that abort processes which request a resource that is in use.
  - Negates *hold and wait*.
- Preemptions may be possible if a process does not use its resources until it has acquired all it needs.
  - Negates *no preemption*.
- Transaction processing systems provide checkpoints so that processes may back out of a transaction.
  - Negates *irreversible process*.
- Systems that prevent, detect, or avoid cycles.
  - Negates *circular wait*. Often, the preferred solution.
Resource allocation graphs

Set of Processes \( P = \{P_1, P_2, \ldots, P_n\} \)

Set of Resources \( R = \{R_1, R_2, \ldots, R_m\} \)  \( R_j \) has 2 units

Some resources come in multiple units.

Process \( P_i \) waits for (has requested) \( R_j \)

Resource \( R_j \) has been allocated to \( P_i \)

\[ P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_4 \rightarrow P_3 \rightarrow R_3 \]
Multiple resource unit case:

*No Deadlock—yet!*

Because, either $P_2$ or $P_4$ could relinquish a resource allowing $P_1$ or $P_3$ (which are currently blocked) to continue. $P_2$ is still executing, even if $P_4$ requests $R_1$. 
... a knot is required

Cycle is a *necessary condition* for a deadlock. But when dealing with multiple unit resources—*not sufficient*.

A *knot* must exist—a cycle with no non-cycle outgoing path from any involved node.

At the moment assume that:

- a process *halts* as soon as it waits for one resource, and
- processes can wait for only *one* resource at a time.
Further requests

(a) If $P_4$ requests $R_1$, no deadlock since $P_2$ may release $R_1$, allowing $P_4$ to complete.

(b) If $P_2$ requests $R_2$: Deadlock—Cycle—Knot.

No active processes to release resources.
Strategies for deadlocks

In general, four strategies are used for dealing with deadlocks:

- **Ignore**: stick your head in the sand and pretend there is no problem at all.

- **Prevent**: design a system in such a way that the possibility of deadlock is excluded *a priori* (e.g., compile-time-statically, by design)

- **Avoid**: make a decision dynamically checking whether the request will, if granted, potentially lead to a deadlock or not (e.g., run-time-dynamically, before it happens)

- **Detect**: let the deadlock occur and detect when it happens, and take some action to recover after the fact (e.g., run-time-dynamically, after it happens)
Different people react to this strategy in different ways:

• **Mathematicians**: find deadlock totally unacceptable, and say that it must be prevented at all costs.

• **Engineers**: ask how serious it is, and do not want to pay a penalty in performance and convenience.

The UNIX approach is just to ignore the problem on the assumption that most users would prefer an occasional deadlock, to a rule restricting user access to only one resource at a time.

The problem is that the prevention price is high, mostly in terms of putting inconvenient restrictions on processes.
Deadlock prevention

The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is excluded *a priori*. Methods for preventing deadlock are of two classes:

- *indirect methods* prevent the occurrence of one of the necessary conditions listed earlier.

- *direct methods* prevent the occurrence of a circular wait condition.

Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes.
More on deadlock prevention

- **Mutual exclusion**
  - In general, this condition cannot be disallowed.

- **Hold-and-wait**
  - The hold and-wait condition can be prevented by requiring that a process request all its required resources at one time, and blocking the process until all requests can be granted simultaneously.

- **No preemption**
  - One solution is that if a process holding certain resources is denied a further request, that process must release its unused resources and request them again, together with the additional resource.

- **Circular Wait**
  - The circular wait condition can be prevented by defining a linear ordering of resource types (e.g. Directed Acyclic Graph). If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.
Deadlock avoidance

Deadlock avoidance, allows the necessary conditions but makes judicious choices to ensure that a deadlock-free system remains free from deadlock. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future requests for process resources.

Ways to avoid deadlock by careful resource allocation:

• Resource trajectories.
• Safe/unsafe states.
• Dijkstra’s Banker's algorithm.
Banker’s algorithm—definitions

Assume $N$ Processes $\{P_i\}$
M Resources $\{R_j\}$

Availability vector $\text{Avail}_j$, units of each resource (initialized to maximum, changes dynamically).

Let $[\text{Max}_{ij}]$ be an $N \times M$ matrix.

$\text{Max}_{ij} = L$ means Process $P_i$ will request at most $L$ units of $R_j$.

$[\text{Hold}_{ij}]$ Units of $R_j$ currently held by $P_i$

$[\text{Need}_{ij}]$ Remaining need by $P_i$ for units of $R_j$

$\text{Need}_{ij} = \text{Max}_{ij} - \text{Hold}_{ij}$, for all $i$ & $j$
Banker’s Algorithm—resource request

At any instance, $P_i$ posts its request for resources in vector $\text{REQ}_j$ (i.e., no hold-and-wait)

**Step 1:** verify that a process matches its needs.

$\text{if } \text{REQ}_j > \text{Need}_i \text{ abort—error, impossible}$

**Step 2:** check if the requested amount is available.

$\text{if } \text{REQ}_j > \text{Avail}_j \text{ goto Step 1—}P_i \text{ must wait}$

**Step 3:** provisional allocation (i.e., guess and check)

$\text{Avail}_j = \text{Avail}_j - \text{REQ}_j$

$\text{Hold}_{ij} = \text{Hold}_{ij} + \text{REQ}_j$

$\text{Need}_{ij} = \text{Need}_{ij} - \text{REQ}_j$

$\text{if isSafe() then grant resources—system is safe}$

$\text{else cancel allocation; goto Step 1—}P_i \text{ must wait}$
Banker’s Algorithm—\textit{isSafe}

Find out whether the system is in a safe state. Work and Finish are two temporary vectors.

**Step 1:** initialize.
\[
\text{Work}_j = \text{Avail}_j \text{ for all } j; \text{ Finish}_i = \text{false} \text{ for all } i.
\]

**Step 2:** find a process $P_i$ such that
\[
\text{Finish}_i = \text{false} \text{ and } \text{Need}_{ij} \leq \text{Work}_j \text{, for all } j
\]
if no such process, \textit{goto Step 4}.

**Step 3:**
\[
\text{Work}_j = \text{Work}_j + \text{Hold}_{ij} \text{ (i.e., pretend it finishes and Finish}_i = \text{true) frees up the resources)
\]
\textit{goto Step 2}.

**Step 4:**
\[
\text{if Finish}_i = \text{true for all } i
\]
\[
\text{then return true—yes, the system is safe}
\]
\[
\text{else return false—no, the system is NOT safe}
\]
Banker’s algorithm—what is safe?

Safe with respect to some resource allocation.

• very safe
  \[ \text{NEED}_i \leq \text{AVAIL} \text{ for all Processes } P_i \]
  Processes can run to completion in any order.

• safe (but take care)
  \[ \text{NEED}_i > \text{AVAIL} \text{ for some } P_i \]
  \[ \text{NEED}_i \leq \text{AVAIL} \text{ for at least one } P_i \text{ such that} \]
  There is at least one correct order in which the processes may complete their use of resources.

• unsafe (deadlock inevitable)
  \[ \text{NEED}_i > \text{AVAIL} \text{ for some } P_i \]
  \[ \text{NEED}_i \leq \text{AVAIL} \text{ for at least one } P_i \]
  But some processes cannot complete successfully.

• deadlock
  \[ \text{NEED}_i > \text{AVAIL} \text{ for all } P_i \]
  Processes are already blocked or will become so as they request a resource.
### Example—safe allocation

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Hold</th>
<th>Need</th>
<th>Finish</th>
<th>Avail</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>(P_2)</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_3)</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For simplicity, assume that all the resources are identical.

Assume \(P_1\) acquires one unit. **Very safe? No!** \(\text{Need}_2 > 2\)

**Safe?** Let us see with the safe/unsafe algorithm...

\[\begin{align*}
i &= 1; \text{does } P_1 \text{ agree with Step 2? No.} \\
i &= 2; \text{does } P_2 \text{ agree with Step 2? No.} \\
i &= 3; \text{does } P_3 \text{ agree with Step 2? Yes. Work = Work+Hold}_3; \text{Finish}_3 = T \\
i &= 1; \text{does } P_1 \text{ agree with Step 2? Yes. Work = Work+Hold}_1; \text{Finish}_1 \\
\end{align*}\]

No more (unfinished) \(P_i\), therefore **safe.**

\[\begin{align*}
i &= 2; \text{does } P_2 \text{ agree with Step 2? Yes. Work = Work+Hold}_2; \text{Finish}_2 \\
\end{align*}\]
### Example—safe allocation

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Hold</th>
<th>Need</th>
<th>Finish</th>
<th>Avail</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>P_2</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P_3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume P_1 acquires one unit.
Example—safe allocation

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Hold</th>
<th>Need</th>
<th>Finish</th>
<th>Avail</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>F</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>P₂</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P₃ can acquire the last unit and finish. Then, P₃ frees up resources.
Example—safe allocation

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Hold</th>
<th>Need</th>
<th>Finish</th>
<th>Avail</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>F</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>$P_2$</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>F</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Then, $P_1$ can acquire two more units and finish.
Example—safe allocation

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Hold</th>
<th>Need</th>
<th>Finish</th>
<th>Avail</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_1)</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>5</td>
<td>5_2</td>
</tr>
<tr>
<td>(P_2)</td>
<td>4</td>
<td>1_4</td>
<td>3_0</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(P_3)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finally, \(P_2\) can acquire three more units and finish.
Example—unsafe allocation

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Hold</th>
<th>Need</th>
<th>Finish</th>
<th>Avail</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>F</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P₂</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: New numbers here!

Assume P₂ acquires one unit.
As before, P₃ can finish and release its resources.

*BUT*...

i = 1; does P₁ agree with Step 2? No.
i = 2; does P₃ agree with Step 2? No.
i = 3; does P₃ agree with Step 2? Yes. Work = Work + Hold₂;

Any more unfinished Pᵢ? Yes.

P₁ and P₂ cannot finish. Therefore *unsafe*.
Example—unsafe allocation cont.

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Hold</th>
<th>Need</th>
<th>Finish</th>
<th>Avail</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>F</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P₂</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOW...
Deadlock detection

This technique does not attempt to prevent deadlocks; instead, it lets them occur. The system detects when this happens, and then takes some action to recover after the fact (i.e., is reactive). With deadlock detection, requested resources are granted to processes whenever possible. Periodically, the operating system performs an algorithm that allows it to detect the circular wait wait condition.

A check for deadlock can be made as frequently as resource request, or less frequently, depending on how likely it is for a deadlock to occur. Checking at each resource request has two advantages: It leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system. On the other hand, such frequent checks consume considerable processor time.
Once the deadlock algorithm has successfully detected a deadlock, some strategy is needed for recovery. There are various ways:

- **Recovery through *Preemption***
  In some cases, it may be possible to temporarily take a resource away from its current owner and give it to another.

- **Recovery through *Rollback***
  If it is known that deadlocks are likely, one can arrange to have processes *checkpointed* periodically. For example, can undo transactions, thus free locks on database records. This often requires extra software functionality.

- **Recovery through *Termination***
  The most trivial way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. **Warning! Irrecoverable losses or erroneous results may occur, even if this is the least advanced process.**
# Summary of strategies

<table>
<thead>
<tr>
<th>Principle</th>
<th>Resource Allocation Strategy</th>
<th>Different Schemes</th>
<th>Major Advantages</th>
<th>Major Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETECTION</td>
<td>• ζερν λιβεραλ; γραν</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Ινποκε περιοδικαλ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Νεπερ δελαψσ προχσοσ ινιτιατιον</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Φαχιλιτατεσ ον−λινε ηανδλινγ.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Νεϖερ δελαψσ προχεσσ ινιτιατιον.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Νεκεντ πρεεµπτιον λοσσεσ.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Μανιπυλατε το φιν</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| PRESENTATION | • Χονσερϖατιϖε; υνδε ρεσουρχεσ. | | | |
|              | • Πρεεµπτιον ισ νεεδεδ. | | | |
|              | • Πρεεµπτιον ορδερινγ | | | |
|              | • Φεασιβλε το ενφορχε ρεσουρχεσ. | | | |
|              | • Νεεδσ νο ρυ−τιµε χοµπυτατιον. | | | |
|              | • Φυτυρε ρεσουρχε ρεθυεστσ μυστ βε κνοων. | | | |

| ASOLANXE | • Σελεχτσ μιδωασ βετεεν ηηατ οφ δετεχτιον ανδ πρεδεντιον. | | | |
|           | • Μανιπυλατε το φιν | | | |
|           | • Νο πρεεµπτιον νεχεσσαρψ. | | | |

---

*July 99*
Other issues

Two-phase Locking

Although both avoidance and prevention are not very promising in general, many excellent special-purpose algorithms are known. The best data base algorithm is known as two-phase locking (covered in detail in another course).

Non-resource Deadlocks

Deadlocks can also occur in other situations, where no single resource is involved. E.g., two processes exchanging messages, where both are listening and waiting for the other to send a message.

Starvation

A problem closely related to deadlock is starvation. In a dynamic system, requests for resources happen all the time. The key is to make a decision about who gets which resources when. This decision sometimes may lead to some processes never receiving service, though they are not deadlocked!