Basic assumptions of deadlock theory:
If a process gets the resources it requests, it completes, exits, and releases resources.
There are no outside influences upon the processes.
Processes do not lie or collude with one another to deceive the operating system.

Deadlock occurs when:
There is a cycle in the resource allocation graph (for locks).
Incremental allocation (for a set of resources) leads to a situation in which N processes have managed to consume all resources, where each of the N processes needs more.

Deadlock cannot occur when:
Each process requests all resources that it needs in one atomic step. (Easy to understand)
All processes request locks in exactly the same order. (A bit more involved).
How do deadlocks arise?

- **Poor locking patterns:** certain patterns of locking create deadlocks inadvertently.
- **Resource contention:** a bunch of processes all need too much memory at the same time.
- **Poor understanding of I/O:** a set of processes end up waiting for one another due to poor planning or misunderstanding of I/O buffering.
### Three approaches to deadlock prevention

<table>
<thead>
<tr>
<th></th>
<th>Banker's algorithm</th>
<th>Schedule filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock priority</strong></td>
<td>Arrange for all locks to be done in order</td>
<td>Arrange for all locks to be done in order</td>
</tr>
<tr>
<td><strong>Banker's algorithm</strong></td>
<td>Reject unsafe states in which deadlock might occur</td>
<td>Reject unsafe states in which deadlock might occur</td>
</tr>
<tr>
<td><strong>Schedule filtering</strong></td>
<td>Reject schedules that lead to deadlock</td>
<td>Reject schedules that lead to deadlock</td>
</tr>
<tr>
<td><strong>Non-transparent:</strong></td>
<td>Non-transparent: application has to be aware of strategy.</td>
<td>Non-transparent: application does not need to know about strategy.</td>
</tr>
<tr>
<td></td>
<td>Transparent: application does not need to know about strategy.</td>
<td>Transparent: application does not need to be aware of strategy.</td>
</tr>
<tr>
<td><strong>Works best for:</strong></td>
<td>Works best for resources with multiple units (memory, disk, kernel tables, etc)</td>
<td>Resource-intensive: must maintain a tree of schedules that led to deadlock.</td>
</tr>
<tr>
<td></td>
<td>Works best for locks and binary resources.</td>
<td>Resource-intensive: must maintain a tree of schedules that led to deadlock.</td>
</tr>
</tbody>
</table>
Banker's algorithm (Dijkstra)

- Define a **safe state** as one in which deadlock *cannot* occur.
- A safe state is associated with a **completion schedule**.
- Deny resource allocations when they will result in an "unsafe" state in which a deadlock is *possible* (but not guaranteed).
- **Safety**: there is a path of execution that does not deadlock.
- Key to algorithm: **resource matrix**
Attributes of algorithm:

- **Paranoid and pessimistic**: assumes the worst about processes
- **Incremental**: resource needs need not be known in advance.
- **Transparent**: processes do not need to know about it.
- Can lead to livelock: it leaves processes that were denied resources in a runnable state.
Basic idea of the banker's algorithm

The operating system is a **banker**.
The banker **loans resources** to processes.
The processes **pay back the loan** by returning the resources.
Interest-free loans! :)
Banker's goal is to **assure that loans are paid back**.

(this is not realistic for housing loans! :)

---

The banker
Thursday, October 22, 2009    11:39 AM
The banker is paranoid:

The banker only gets payback on a loan if the client is able to complete the project, e.g., a housing development.

But sometimes a client comes back for more money. The banker has to decide whether to grant it. The banker only grants the request if she'll still have enough money for all current loans to complete on some schedule. Otherwise, the banker says no!
Central concern for the banker is **safety**

Safety is defined as a situation in which from the banker's point of view, **some schedule** allows all loans to be paid back.

A situation is unsafe if there is no such schedule.

Safety is **paranoid**

An unsafe situation does not necessarily deadlock (not all potential deadlocks occur, because processes can end at any time and release resources)

A safe situation can deadlock if a process doesn't complete (e.g., due to an infinite loop).
The bankers' algorithm is based upon the following (perhaps unfounded) assumptions:

All processes are at the same priority for receiving resources.
Processes that receive what they requested will complete and release resources.
Processes that do not receive what they requested will not complete and will continue to hold resources.
Processes will not release resources until they receive all resources that they need.
Processes are independent and do not collude.

These are closed-world assumptions.
If processes lie, or are intelligent, or collude, the banker's algorithm doesn't work!
Resource matrices:

- **Rows**: processes
- **Columns**: resources
- **At row i, column j**: demand or supply of resource j by process i.
Simple model: one resource = one column

\[
\begin{align*}
\text{requested} & & \text{granted} & & \text{deficit} \\
\text{p1} & & (4) & - & (3) \\
\text{p2} & & (3) & - & (1) \\
\text{p3} & & (1) & - & (1) \\
\end{align*}
\]

\[
\begin{align*}
\text{R} & - \text{G} = \text{D} \\
\text{Available} & \uparrow \text{how many total units}
\end{align*}
\]

SAFETY:
Start with this configuration
Complete the unblocked processes, release resources, grant to other processes.
If you can complete everything, you're safe.
Simple model: one resource = one column

\[
\begin{array}{ccc}
\text{requested} & \text{granted} & \text{deficit} \\
\hline
P_1 & (4) & (3) & 1 \\
P_2 & (3) & (1) & 2 \\
\text{P} & (1) & (1) & 0 \\
\end{array}
\]

\[R - G = D\]

Available (5)
Simple model: one resource = one column

\[
\begin{align*}
\text{requested} & \quad \text{granted} & \quad \text{deficit} \\
\hline
p_1 & (4) & (4) & (0) \\
p_2 & (3) & (3) & (0) \\
p_3 & (1) & (1) & (0) \\
\hline
\end{align*}
\]

\[
R - G = D
\]

Available (5)
Simple model: one resource = one column

\[
\begin{array}{c|cc|c}
\text{Client} & \text{Requested} & \text{Granted} & \text{Deficit} \\
\hline
\#2 p_1 & (4) & (4) & 0 \\
\#3 p_2 & (3) & (3) & 0 \\
\#1 p_3 & (1) & (1) & 0 \\
\end{array}
\]

A request may not be done unless all complete!

\[
R - G = D
\]

Available (5)
Simple model: one resource = one column

\[
\begin{pmatrix}
\text{requested} & \text{granted} & \text{deficit} \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
6 \\
4 \\
1 \\
\end{pmatrix} - 
\begin{pmatrix}
4 \\
2 \\
1 \\
\end{pmatrix} = 
\begin{pmatrix}
2 \\
2 \\
0 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
R - G = P
\end{pmatrix}
\]

Available (7)

The banker's algorithm will never allow this state! It will reject the request.
Simple model: one resource = one column

<table>
<thead>
<tr>
<th></th>
<th>requested</th>
<th>granted</th>
<th>deficit</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ R - G = D \]

Available \((7)\)

I'll have to press control-C on P1 or P2. So, I tell whoever asked for resources "no".
The nature of safety

Safety is a matter of the ability to complete tasks via resource granting
A situation is safe if there is a completion schedule where every process eventually ends.
An unsafe situation is one in which "too much money has been lent out for any completion schedule to exist".
Employing the algorithm

Start with some allocation, some processes running, and some processes potentially blocked waiting for resources.

Get a new request, and compute what would happen if we grant it.

If granting the request puts the system into a safe state, **record the request as granted, queue the process, and block the process** until it is fulfilled. If granting the request puts the system into an unsafe state, **reject the request** and **do not block the process!**
Two resources

\[
\begin{align*}
\text{P1:} & \quad \begin{pmatrix} 1 & 2 \\ 2 & 4 \\ 1 & 0 \end{pmatrix} & \text{P2:} & \quad \begin{pmatrix} 1 & 2 \\ 1 & 0 \\ 1 & 0 \end{pmatrix} & \text{P3:} & \quad \begin{pmatrix} 0 & 1 \\ 2 & 4 \\ 0 & 0 \end{pmatrix}
\end{align*}
\]

Requested - Available
\[
\begin{align*}
\text{R1:} & \quad G = (7, 5) & \text{R2:} & \quad \text{sum(granted)} \leq \text{avail}
\end{align*}
\]
Two resources

Schedule = 3, 1, 2.

Two resources

\[
\begin{align*}
  &P_1 \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}, & &P_2 \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix}, & &P_3 \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \\
  &R1 & R2 & R1 & R2 & R1 & R2 \\
  &\text{Requested} & = & \text{ Granted} & - & \text{ Deficit} & \text{ Runnable} \\
  &\text{ Available} & \begin{pmatrix} 7 & 5 \end{pmatrix} & \begin{pmatrix} 0 & 0 \end{pmatrix} \end{align*}
\]
Safety algorithm:

○ We are in a **safe state** (of allocation) if there is a path to completion for all outstanding resource requests, where one request at a time is granted.

○ Algorithm:
  ▪ start with p processes,
  ▪ choose one that can complete now,
  ▪ put its resources back into the pool,
  ▪ continue until you have no processes.

○ If you can do this, you have a "safe" state.

But really important, we assume that:
no process is fillibustering.
all processes are independent.
each process, given that its requests are granted, will complete.
Practicality

On the surface, it appears that we have to search for the appropriate schedule every time.

We don't!

Dijkstra's observation: the position of a particular process in the completion sequence

○ only changes if its resource requirements change.
○ moves toward rear if requirements increase
○ moves toward front if requirements decrease

So we store the schedule and update it as resource requirements change (via bubblesort)!
Constraints and runtime

○ There must be a sequence of completions that results in completing all processes.
○ Requires computing the sequence.
○ Computing the sequence takes $O(n^2)$ where $n$ is the number of processes:
  ▪ for each process, if it can complete, complete it and continue.
  ▪ Each step requires $O(n)$ and there are $O(n)$ steps.
○ But maintaining the sequence takes only $O(n)$ per change, because processes only move back and forward when their requirements change.
○ Thus we need $O(n)$ steps per iteration after an initialization of $O(n^2)$ steps.

If requests are small, the time is on average one step.
Key to $O(n)$ update: bubblesort the order.
If one process $P_i$ asks for one more unit, then
Order for other processes is still the same.
Algorithm: swap $P_i$ with $P_{i+1}$, $P_{i+2}$, etc, until
you come to a process with more resource
needs.
On average, swap with at most one other
process!

Your requirement: keep one completion schedule.
Another way to avoid deadlock is that all processes lock resources in a predetermined order.

We cannot do that, but we can do something equivalent.
Why does locking in order prevent deadlock?

Number the locks in increasing order.
Lock in that order.
Lemma: if two processes start at the same lock and lock a contiguous block of locks in order, there is no deadlock.
Proof: one process wins the race, and thus acquires the first lock, blocking the second process. Thus the first process runs to completion. Thus there is no deadlock.

Lemma: If two processes start at different locks and each locks a contiguous block of locks, then there is no deadlock.
Proof: Let processes 1 and 2 start at locks k and m. Without loss of generality assume that k<m.
• If blocks do not overlap, there is no deadlock.
• Else one of the two processes locks m first.
  ○ If it is process 1, that process completes.
  ○ Else if it is process 2, that process completes.
• Thus there is no deadlock in either case.

Lemma: If n processes lock contiguous blocks of locks, there is no deadlock.
Proof: by induction:
Basis step: n=2, no deadlock by above lemma.
Inductive step: Assume true for n. Then if processes 1-n start by locking locks numbered \( k_1-k_n \), then there is no deadlock.
Consider n+1 processes locking locks \( k_1-k_{n+1} \). Without loss of generality, number these in order: \( k_1\leq k_2\leq...\leq k_{n+1} \). Thus \( k_{n+1} \) is the highest numbered lock.
Claim: the process that locks \( k_{n+1} \) first completes.
Proof: Since it is the highest numbered lock, every other
process must wait for the process that locks it first. Then there are \( n \) processes left, which complete by the inductive hypothesis.
But...

Existing code doesn't have to lock resources in order. There is often no logic that gives an appropriate lock order.

And...

Locking non-contiguous sets of locks can lead to deadlock.
Lemma: If two processes request all resources at once, then there is no deadlock.
Proof: If either process requests more resources than are available, the request is denied and the process exits. Else one wins the race and completes. Then the other completes, so there is no deadlock.

Lemma: If n processes request all resources at once, then there is no deadlock.
Proof is trivial by induction on the number of processes n.
But...

Several common computational patterns require incremental allocation
  Reading unbounded streams of input
  Dynamically allocating 2+-dimensional arrays.
Two algorithms for deadlock prevention

Lock prioritization prevents lock sequence deadlocks.
- works on individual locks
- concentrates on when locks are made
- not transparent: application must be aware of it.

Banker's algorithm prevents resource starvation deadlocks.
- works on resources
- concentrates on total resource usage
- transparent
Lock prioritization
○ Give each lock a priority
○ Require locking in order of priority
○ If attempt is made to access out of order,
  ▪ Break lower priority locks
  ▪ Relock in priority order
Reason this works
If everyone locks in priority order, then deadlock is impossible.
If they don't, we undo their locks, and repeat them in priority order.
So, the OS sees locks only in order. The waiting process thus gets the released lock first. (strong semantics: FCFS).
Key idea in lock prioritization

Key idea:

Whenever a lock request is made,

- consider all of the locks we hold.
- at end of request, will hold all of them.
- allow (temporary) releases in the middle of the request.

Good news: no deadlock possible.
Bad news: not transparent.
This is difficult to understand:
Suppose \( A < B \); \( A \) must be locked first

\[ P: \text{Lock } A; \text{Lock } B; \text{do something}; \text{Unlock } A; \text{Unlock } B; \]
\[ Q: \text{Lock } B; \text{Lock } A; \text{do something}; \text{Unlock } B; \text{Unlock } A; \]

**strong semantics**: when I ask for a lock, it's granted in order of the request.
processes request locks all at once, and don't wait between requests.

In words:
When I have two schedules:
\[ \text{lock}(A); \text{lock}(B); \]
\[ \text{lock}(B); \text{lock}(A); \]
then the second is rewritten as
\[ \text{lock}(B); \text{unlock}(B); \text{lock}(A); \text{lock}(B); \]
Good news: if locks are acquired with no intervening work, then this is equivalent to atomic allocation (allocating all at once). Thus deadlocks are prevented.

Bad news: if locks are acquired with intervening work, then it is possible for an application not to honor a lock due to races.

Note: application programmer must be prepared for revocation of potentially all locks in acquisition of another.
Implementation

- semaphores are a kernel table
  - refer to semaphores by number.
  - number is offset into table.
- Low-numbered semaphore wins.
  - arbitrary.
  - unknown to process.

Linux does not do this.
Controversy: this takes (perhaps too much) time
Reason: **orthogonality**: it is possible for processes to utilize "orthogonal sets" of locks, i.e., locks that have nothing to do with one another.

if $R_1 < S_1 < R_2 < S_2$,
   
   lock($R_2$) followed by lock($R_1$)
in the application results in:
   
   lock($R_2$) unlock($R_2$) lock($R_1$) lock($R_2$)

in the OS, even if there is no other process using $R_1$, $R_2$!
Example: file locking

R1: file lock.
R2: some other thing, e.g., tape

Code

lock(R2);
FILE *F = fopen("foo","r");
lock(R1);
copy contents of F onto /dev/mt0
unlock(R2)
unlock(R1)

Code2:
lock(R2);
FILE *g = fopen("foo","r");
fgets(buf,1024,g);
unlock(R2);

OOPS! Code2 (unknown to code1) reads first line of file out from under Code1, so contents of tape are 2nd line onward!
Priority locking caveats

- Only works for single resources (not resources having multiple equivalent instances)
- All priorities must be different or ties ensue, deadlock possible.
- If two processes contend, and one locks in order while the other one does not, then the one that locks in order goes first.
- If two processes contend in order, then first one wins.
Two problems with lock priority:

➢ lack of **transparency**: application has to know that locking has these properties

➢ lack of handling of **multiplicity**: every resource must have one instance, to disambiguate priorities.
One issue in deadlock avoidance is application awareness; must the application be written differently in order to deal with what the OS does?

- **Transparent mechanisms**: application doesn't need to be rewritten.
- **Non-transparent mechanisms**: application needs to be aware of locking and unlocking behavior.

Problem: for lock priority mechanism to work, application must be aware that any lock can be broken during a wait for another lock, then relocked. Thus it is not transparent.

Thus -- more important -- in practice, the lock priority algorithm is equivalent to atomic locking of all required locks.
Some operating systems research has been done on "schedule filtering".

The guiding principle: deadlock is based upon the particular schedule of events. If that schedule is not repeated, no deadlock will occur.

So, we
detect deadlocks normally.
remember what schedules led to which deadlocks.
reject schedules in which deadlock will occur by delaying events so that the schedule does not occur.