### Safe to Unsafe Transition

Current safe state doesn’t imply future states are safe:

<table>
<thead>
<tr>
<th></th>
<th>Current Loan</th>
<th>Max Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>User1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>User2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>User3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Available: 2

Suppose User3 requests and gets one more resource:

<table>
<thead>
<tr>
<th></th>
<th>Current Loan</th>
<th>Max Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>User1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>User2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>User3</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Available: 1
Essence of Banker's Algorithm

Find schedule to complete jobs:
\[ \text{schedule exists iff safe} \]

Method:
"Pretend" you are the CPU.

1. Scan table row by row and find a job that can finish.
2. Add finished job's resources to number available.

Repeat 1 and 2 until:
- no more jobs can finish (unsafe), or
- all jobs finish (safe).
Banker's Algorithm

Constants:  
N (number of processes)
Total_Units
Max_Demand [i]

Variables:  
i (denotes a process)
Unused_units
Allocated [i]
Claim [i]
Cannot_Finish [i]

Function:  
Claim [i] = Max_Demand[i] - Allocated[i]
Banker's Algorithm

Begin

Unused_units = Total_Units;

For i = 1 to N Do
    Begin
        Unused_Units = Unused_Units - Allocated[i];
        Cannot_Finish[i] = TRUE;
    End;

i = 1;

While (i <= N) Do
    Begin
        If (Cannot_Finish[i] AND Claim[i] <= Unused_Units)
            Then Begin
                Cannot_Finish[i] = FALSE;
                Unused_units = Unused_Units - Allocated[i];
                i = i;
            End;
        Else i = i + 1;
    End;

If (Unused_Units <= Total_Units)
    Then Return (SAFE)
Else Return (UNSAFE);

End;
Banker's Example #1

Total Units = 10 units
N = 3 processes

Process: 1  2  3  1
Request: 2  3  4  1

Can the fourth request be satisfied?

<table>
<thead>
<tr>
<th>Process</th>
<th>MaxDemand</th>
<th>Allocated</th>
<th>Claim</th>
<th>MNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unused Units =
1 =
Banker's Example #2

Total Units = 10 units
N = 3 processes

Process: 1 2 3 1
Request: 4 1 1 2

Can the fourth request be satisfied?

<table>
<thead>
<tr>
<th>Process</th>
<th>MaxDemand</th>
<th>Allocated</th>
<th>Claim</th>
<th>MNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unused Units =
1 =
Banker's Algorithm: Summary

(+) PRO's:

+ Deadlock never occurs.
+ More flexible, efficient than deadlock prevention. (Why?)

(-) CON's:

- Must know max use of each resource when job starts.
  => No truly dynamic allocation
- Process could block even though deadlock would never occur.