• **What is the meaning of life?** [One answer](#)

• **CPU utilization:** fraction of the time CPU is actually used. CPU may remain unused if there is nothing to run.

• **Round robin with different arrival times:** next process is picked from the head of ready queue, processes coming from outside/switched out from CPU enter at the tail. Rules for breaking ties.

• **What are resources?** Drives, memory blocks, locks for critical sections, etc.

• **Why modern OSs do not actively prevent deadlocks?**
  – May be used by embedded/rear-time OSs,
  – If a process does not progress for a few seconds, the system may generate a message. “.. not responding”
  – data-bases, locked files may checked for deadlocks
  – Some version of [Windows](https://www.microsoft.com) and [Linux](https://www.linux.org) may have capability to check for deadlocks involving locks for critical sections.
  – Use of mechanism by which locks are always acquired in a defined order
• **How does resource ordering help?**

<table>
<thead>
<tr>
<th>thread one function</th>
<th>thread two function</th>
</tr>
</thead>
</table>
| **void *do_work_one(void *param)** {  
  pthread_mutex_lock(&first_mutex);  
  pthread_mutex_lock(&second_mutex);  
  /** * Do some work */  
  pthread_mutex_unlock(&second_mutex);  
  pthread_mutex_unlock(&first_mutex);  
  pthread_exit(0);  
} | **void *do_work_two(void *param)** {  
  pthread_mutex_lock(&second_mutex);  
  pthread_mutex_lock(&first_mutex);  
  /** * Do some work */  
  pthread_mutex_unlock(&first_mutex);  
  pthread_mutex_unlock(&second_mutex);  
  pthread_exit(0);  
} |

```c
void *do_work_one(void *param)  
{
  pthread_mutex_lock(&first_mutex);
  pthread_mutex_lock(&second_mutex);
  /** * Do some work */
  pthread_mutex_unlock(&second_mutex);
  pthread_mutex_unlock(&first_mutex);
  pthread_exit(0);
}

void *do_work_two(void *param)  
{
  pthread_mutex_lock(&second_mutex);
  pthread_mutex_lock(&first_mutex);
  /** * Do some work */
  pthread_mutex_unlock(&first_mutex);
  pthread_mutex_unlock(&second_mutex);
  pthread_exit(0);
}
```
Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state:
  - **Deadlock prevention**
    - ensuring that at least one of the 4 conditions cannot hold
  - **Deadlock avoidance**
    - Dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Allow the system to enter a deadlock state
  - **Detect and then recover.** Hope is that it happens rarely.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can **prevent** the occurrence of a deadlock.

**Mutual exclusion:** only one process at a time can use a resource

**Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes

**No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

**Circular wait:** there exists a set \( \{P_0, P_1, ..., P_n\} \) of waiting processes that are circularly waiting.
Deadlock Avoidance

Manage resource allocation to ensure the system never enters an unsafe state.
Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- *Resource-allocation state* is defined by the number of available and allocated resources, and the maximum demands of the processes
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - Ensures all allocations result in a safe state
Deadlock Avoidance: Handling resource requests

• For each resource request:
  – Decide whether or not process should wait
    • To avoid possible future deadlock

• Predicated on:
  1. Currently available resources
  2. Currently allocated resources
  3. Future requests and releases of each process
     • Finding a Safe sequence
Avoidance: amount and type of information needed

- **Resource allocation state**
  - Number of available and allocated resources
  - Maximum demands of processes

- **Dynamically** examine resource allocation state
  - Ensure circular-wait cannot exist

- Simplest model:
  - Declare maximum number of resources for each type
  - Use information to avoid deadlock
Safe Sequence

System must decide if immediate allocation leaves the system in a safe state

System is in **safe state** if there exists a sequence \( <P_1, P_2, ..., P_n> \) of ALL the processes such that

- for each \( P_i \), the resources that \( P_i \) can still request can be satisfied by
  - currently available resources +
  - resources held by all the \( P_j \), with \( j < i \)
  - That is
    - If \( P_i \) resource needs are not immediately available, then \( P_i \) can wait until all \( P_j \) have finished and released resources
    - When \( P_i \) terminates, \( P_{i+1} \) can obtain its needed resources, and so on

- If no such sequence exists: system state is **unsafe**
Deadlock avoidance: Safe states

- If the system can:
  - Allocate resources to each process in some order
    - Up to the maximum for the process
    - Still avoid deadlock
    - Then it is in a **safe state**
- A system is safe ONLY IF there is a safe sequence
- A safe state is not a deadlocked state
  - Deadlocked state is an unsafe state
  - Not all unsafe states are deadlock
Safe, Unsafe, Deadlock State

Examples of safe and unsafe states in next 3 slides
Example A: Assume 12 Units in the system

<table>
<thead>
<tr>
<th></th>
<th>Max need (initially declared)</th>
<th>Current holding at time T0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

**At T0:**
9 units allocated
12-9 = 3 units available

A unit could be a drive, a block of memory etc.

• **Is the system is in a safe state at time T0?**
  – Try sequence <P1, P0, P2>
  – P1 can be given 2 units
  – When P1 releases its resources; there are 5 units
  – P0 uses 5 and subsequently releases them (# 10 now)
  – P2 can then proceed.

• **Thus <P1, P0, P2> is a safe sequence, and at T0 system was in a safe state**
Example A: Assume 12 Units in the system (timing)

Is the state at T0 safe? Detailed look for instants T0, T1, T2, etc..

<table>
<thead>
<tr>
<th>Time</th>
<th>Max need</th>
<th>Current holding</th>
<th>+2 allo to P1</th>
<th>P1 releases all</th>
<th>...</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>av</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10 done</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>4 done</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2 done</td>
</tr>
</tbody>
</table>

Thus the state at T0 is safe.
Example B: 12 Units initially available in the system

<table>
<thead>
<tr>
<th>Max need</th>
<th>T0</th>
<th>T1 safe?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

Before T1:
- 3 units available

At T1:
- 2 units available

- At time $T_1$, P2 is allocated 1 more units. Is that a good decision?
  - Now only P1 can proceed (already has 2, and given be given 2 more)
  - When P1 releases its resources; there are 4 units
  - P0 needs 5 more, P2 needs 6 more. Deadlock.
    - **Mistake** in granting P2 the additional unit.
- The state at $T_1$ is not a safe state. Wasn’t a good decision.
Avoidance Algorithms

- **Single instance** of a resource type
  - Use a resource-allocation graph scheme

- **Multiple instances** of a resource type
  - Use the banker’s algorithm (Dijkstra)
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process $P_i$ may request resource $R_j$; represented by a dashed line. This is new.
- Claim edge converts to **request edge** when a process requests a resource
- Request edge converted to an **assignment edge** when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Requirement: Resources must be claimed *a priori* in the system
Suppose $P_2$ requests $R_2$. Can $R_2$ be allocated to $P_2$?

Although $R_2$ is currently free, we cannot allocate it to $P_2$, since this action will create a cycle getting system in an unsafe state. If $P_1$ requests $R_2$, and $P_2$ requests $R_1$, then a deadlock will occur.
• Suppose that process $P_i$ requests a resource $R_j$

• The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
Banker’s Algorithm: examining a request

- Multiple instances of resources.
- Each process must a priori claim maximum use.
- When a process requests a resource,
  - it may have to wait until the resource becomes available (resource request algorithm)
  - Request should not be granted if the resulting system state is unsafe (safety algorithm)
- When a process gets all its resources it must return them in a finite amount of time.
- Modeled after a banker in a small town making loans.
Let $n =$ number of processes, and $m =$ number of resources types.

- **Available:** Vector of length $m$. If available $[j] = k$, there are $k$ instances of resource type $R_j$ available

**Processes vs resources:**

- **Max:** $n \times m$ matrix. If $Max [i,j] = k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$

- **Allocation:** $n \times m$ matrix. If $Allocation[i,j] = k$ then $P_i$ is currently allocated $k$ instances of $R_j$

- **Need:** $n \times m$ matrix. If $Need[i,j] = k$, then $P_i$ may need $k$ more instances of $R_j$ to complete its task

$$Need [i,j] = Max[i,j] – Allocation [i,j]$$
Safety Algorithm: Is System in safe state?

1. Let \textbf{Work} and \textbf{Finish} be vectors of length \( m \) and \( n \), respectively. Initialize:
   \begin{align*}
   \text{Work} & = \text{Initially Available resources} \\
   \text{Finish} [i] & = \text{false for } i = 0, 1, \ldots, n-1
   \end{align*}

2. Find a process \( i \) such that both:
   \begin{enumerate}
   \item \( \text{Finish} [i] = \text{false} \)
   \item \( \text{Need}_i \leq \text{Work} \)
   \end{enumerate}
   If no such \( i \) exists, go to step 4

3. \( \text{Work} = \text{Work} + \text{Allocation}_i; \)
   \( \text{Finish}[i] = \text{true} \)
   go to step 2

4. If \( \text{Finish} [i] = \text{true} \) for all \( i \), then the system is in a safe state

\[ n = \text{number of processes}, \]
\[ m = \text{number of resources types} \]
\[ \text{Need}_i: \text{additional res needed} \]
\[ \text{Work}: \text{res currently free} \]
\[ \text{Finish}_i: \text{processes finished} \]
\[ \text{Allocation}_i: \text{allocated to } i \]
Resource-Request Algorithm for Process $P_i$

**Notation:** $\text{Request}_i = \text{request vector for process } P_i$. If $\text{Request}_i[j] = k$ then process $P_i$ wants $k$ instances of resource type $R_j$

**Algorithm: Should the allocation request be granted?**

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise $P_i$ must wait, since resources are not available
3. **Is allocation safe?:** Pretend to allocate requested resources to $P_i$ by modifying the state as follows:
   - $\text{Available} = \text{Available} - \text{Request}_i$
   - $\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i$
   - $\text{Need}_i = \text{Need}_i - \text{Request}_i$
   - **If safe** ⇒ the resources are allocated to $P_i$
   - **If unsafe** ⇒ $P_i$ must wait, and the old resource-allocation state is preserved.
Example 1A: Banker’s Algorithm

- 5 processes $P_0$ through $P_4$;
- 3 resource types: $A$ (10 instances), $B$ (5 instances), and $C$ (7 instances);
- **Is it a safe state?**

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Currently available</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The Need matrix is redundant.
Example 1A: Banker’s Algorithm

- Is it a safe state?
- Yes, since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria

<table>
<thead>
<tr>
<th>Process</th>
<th>Max A</th>
<th>Max B</th>
<th>Max C</th>
<th>Allocation A</th>
<th>Allocation B</th>
<th>Allocation C</th>
<th>Need A</th>
<th>Need B</th>
<th>Need C</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Available</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

P1 run to completion. Available becomes $[3 \ 3 \ 2] + [2 \ 0 \ 0] = [5 \ 3 \ 2]$

P3 run to completion. Available becomes $[5 \ 3 \ 2] + [2 \ 1 \ 1] = [7 \ 4 \ 3]$

P4 run to completion. Available becomes $[7 \ 4 \ 3] + [0 \ 0 \ 2] = [7 \ 4 \ 5]$

P2 run to completion. Available becomes $[7 \ 4 \ 5] + [3 \ 0 \ 2] = [10 \ 4 \ 7]$

P0 run to completion. Available becomes $[10 \ 4 \ 7] + [0 \ 1 \ 0] = [10 \ 5 \ 7]$ all done

**Hence state above is safe.**
Ex 1B: Assume now $P_1$ Requests (1,0,2)

- Check that $\text{Request}_i \leq \text{Need}_i$ and $\text{Request}_i \leq \text{Available}$. $(1,0,2) \leq (3,3,2) \rightarrow \text{true}.$
- Check for safety after pretend allocation. $P_1$ allocation would be $(2\ 0\ 0) + (1\ 0\ 2) = 302$

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Pretend Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Available</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement.
Hence state above is safe, thus the allocation would be safe.
Ex 1C,1D: Additional Requests ..

- Given State is (same as previous slide)

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>available</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P0</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

P4 request for (3,3,0): cannot be granted - resources are not available.
P0 request for (0,2,0): cannot be granted since the resulting state is unsafe.
Bankers Algorithm: Practical Issues

- Processes may not know in advance about their maximum resource needs
- Number of processes is not fixed
  - Varies dynamically
- Resources thought to be available can disappear
- Few systems use this algorithm
Deadlock Detection

• Allow system to enter deadlock state. If that happens, detect the deadlock and do something about it.

• Detection algorithm
  – Single instance of each resource:
    • wait-for graph
  – Multiple instances:
    • detection algorithm (based on Banker’s algorithm)

• Recovery scheme
Single Instance of Each Resource Type

• Maintain **wait-for graph** (based on resource allocation graph)
  – Nodes are processes
  – $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$
  – **Deadlock if cycles**

• Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

• An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph
Resource-Allocation Graph and Wait-for Graph

Has cycles. Deadlock.
Banker’s algorithm: Can requests by all process be satisfied?

- **Available**: A vector of length $m$ indicates the number of available (currently free) resources of each type.
- **Allocation**: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An $n \times m$ matrix indicates the current request of each process. If $Request\ [i][j] = k$, then process $P_i$ is requesting $k$ more instances of resource type $R_j$. 
Let us take 5 minutes to discuss COVID-19. You are invited to share what you think or know.

- Measures that are not effective or are least effective.
- Measures that people don’t talk about, but could be worth trying.