Advanced Synchronization and Deadlock
A house of cards?

- Locks + CV/signal a great way to regulate access to a single shared object...
- ...but general multi-threaded programs touch multiple shared objects
- How can we atomically modify multiple objects to maintain
  - Safety: prevent applications from seeing inconsistent states
  - Liveness: avoid deadlock
    - a cycle of threads forever stuck waiting for one another
Deadlock

A cycle of waiting among a set of threads, where each thread is waiting for some other thread in the cycle to take some action

```java
Producer1() {
    emptyBuffer.acquire()
    producerMutexLock.acquire()
    : 
}

Producer2() {
    producerMutexLock.acquire()
    emptyBuffer.acquire()
    : 
}
```
Deadlock

A cycle of waiting among a set of threads, where each thread is waiting for some other thread in the cycle to take some action.

Mutually recursive locking

$S_1$ waiting for unlock

lock.Acquire()

while (must wait) {
    cv.wait(&lock2)
}

lock2.release()

lock1.release()

$S_2$ waiting for unlock

lock.Acquire()

lock1.acquire()

lock2.acquire()

lock1.release()
Deadlock

A cycle of waiting among a set of threads, where each thread is waiting for some other thread in the cycle to take some action.

Mutually recursive locking

lock.Acquire()

$S_1$ waiting for unlock

$S_2$ waiting for unlock

Nested waiting

lock.Acquire()

$S_1$ waiting for unlock

lock.Acquire()

$S_2$ waiting for signal
Deadlock

A cycle of waiting among a set of threads, where each thread is waiting for some other thread in the cycle to take some action.

Mutually recursive locking

\[
\begin{align*}
S_1 & \quad \text{waiting for unlock} \\
lock.\text{Acquire}() & \\
S_2 & \quad \text{waiting for unlock} \\
lock.\text{Acquire}() & \\
\end{align*}
\]

Nested waiting

\[
\begin{align*}
S_1 & \quad \text{waiting for unlock} \\
lock.\text{Acquire}() & \\
S_2 & \quad \text{waiting for signal} \\
lock.\text{Acquire}() & \\
\end{align*}
\]
Dining Philosophers

- N philosophers; N plates; N chopsticks
- If all philosophers grab right chopstick
  - deadlock!
Deadlock only if the all hold

- Bounded resources
  - A finite number of threads can use a resource; resources are finite
- No preemption
  - the resource is mine, MINE! (until I release it)
- Wait while holding
  - holds one resource while waiting for another
- Circular waiting
  - \( T_i \) waits for \( T_{i+1} \) and holds a resource requested by \( T_{i-1} \)
  - sufficient if one instance of each resource

Not sufficient in general
Preventing deadlock

- Remove one of the necessary conditions
  - Provide sufficient resources
    - Removes “Bounded resources”
  - Preempt resources
    - Removes “No preemption”
  - Abort requests
    - Removes “Wait while holding”
  - Atomically acquire all resources
    - Removes “Wait while holding”
  - Lock ordering
    - Removes “Circular waiting”
Lock ordering

- A program code convention
  - Developers get together, have lunch, plan lock order
  - Usually reflects static assumptions about the structure of data
    - lock items in a list in order — what if order changes?
  - Nothing at compile time or run time prevents violating this convention!
  - Active research on making it better
    - Finding locking bugs
    - Automatically locking things properly
    - Transactional memory
Avoiding Deadlock: The Banker’s Algorithm

Sum of maximum resources needs can exceed the total available resources if there exists a schedule of loan fulfillments such that all clients receive their maximal loan build their house pay back all the loan

More efficient than acquiring atomically all resources
Living dangerously: Safe, Unsafe, Deadlocked

- **Safe**: For any possible set of resource requests, there exists one safe schedule of processing requests that succeeds in granting all pending and future requests, no deadlock as long as system can enforce safe schedule.

- **Unsafe**: There exists a set of (pending and future) resource requests that leads to a deadlock, for any schedule in which requests are processed, unlucky set of requests can force deadlock.

- **Deadlocked**: The system has at least one deadlock.

A system’s trajectory through its state space.
The Banker's books

- $\text{Max}_{ij} =$ max amount of units of resource $R_j$ needed by $P_i$
  - $\text{MaxClaim}_i =$ $\text{Max}_{ij}$

- $\text{Alloc}_{ij} =$ current allocation of $R_j$ held by $P_i$
  - $\text{HasNow}_i =$ $\text{Alloc}_{ij}$

- $\text{Avail}_j =$ number of units of $R_j$ available

- A request by $P_k$ is safe if there is schedule $P_1, P_2, \ldots P_n$ such that, for all $P_i$, assuming the request is granted,

$$\text{MaxClaim}_i - \text{HasNow}_i \leq \text{Avail} + \sum_{j=1}^{i-1} \text{HasNow}_i$$
### An Example

- **5 processes, 4 resources**

<table>
<thead>
<tr>
<th>Max</th>
<th>Alloc</th>
<th>Avail</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 2</td>
<td>0 0 1 2</td>
<td></td>
</tr>
<tr>
<td>1 7 5 0</td>
<td>1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>2 3 5 6</td>
<td>1 3 5 3</td>
<td></td>
</tr>
<tr>
<td>0 6 5 2</td>
<td>0 6 3 2</td>
<td></td>
</tr>
<tr>
<td>0 6 5 6</td>
<td>0 0 1 4</td>
<td></td>
</tr>
</tbody>
</table>

- **Is this a safe state?**
An Example

5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Alloc</th>
<th>Avail</th>
<th>MaxRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 2</td>
<td>0 0 1 2</td>
<td>0 0 1 2</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 7 5 0</td>
<td>1 0 0 0</td>
<td>1 0 0 0</td>
<td>0 7 5 0</td>
</tr>
<tr>
<td>2 3 5 6</td>
<td>1 3 5 3</td>
<td>1 3 5 3</td>
<td>1 0 0 3</td>
</tr>
<tr>
<td>0 6 5 2</td>
<td>0 6 3 2</td>
<td>0 6 3 2</td>
<td>0 0 2 0</td>
</tr>
<tr>
<td>0 6 5 6</td>
<td>0 0 1 4</td>
<td>0 0 1 4</td>
<td>0 6 4 2</td>
</tr>
</tbody>
</table>

Is this a safe state? $P_1, P_4, P_2, P_3, P_5$

While safe sequence does not include all processes:

Is there a $P_i$ such that $\text{MaxRequest}_i \leq \text{Avail}$?

if no, exit with unsafe

if yes, add $P_i$ to the sequence and set $\text{Avail} = \text{Avail} + \text{HasNow}_i$

Exit with safe
An Example

<table>
<thead>
<tr>
<th>Max</th>
<th>Alloc</th>
<th>Avail</th>
<th>MaxRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0 0 1 2</td>
<td>P₁ 0 0 1 2</td>
<td>1 5 2 0</td>
</tr>
<tr>
<td>P₂</td>
<td>1 7 5 0</td>
<td>P₂ 1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td>2 3 5 6</td>
<td>P₃ 1 3 5 3</td>
<td></td>
</tr>
<tr>
<td>P₄</td>
<td>0 6 5 2</td>
<td>P₄ 0 6 3 2</td>
<td></td>
</tr>
<tr>
<td>P₅</td>
<td>0 6 5 6</td>
<td>P₅ 0 0 1 4</td>
<td></td>
</tr>
</tbody>
</table>

P₂ want to change its allocation to Safe?
An Example

5 processes, 4 resources

<table>
<thead>
<tr>
<th>Max</th>
<th>Alloc</th>
<th>Avail</th>
<th>MaxRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

P2 want to change its allocation to

Safe?
Detecting Deadlock

5 processes, 3 resources. We no longer know Max.

<table>
<thead>
<tr>
<th>Alloc</th>
<th>Avail</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁  R₂  R₃</td>
<td>R₁  R₂  R₃</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P₁  0  1  0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P₂  2  0  0</td>
<td>2 0 2</td>
<td>2 0 2</td>
</tr>
<tr>
<td>P₃  3  0  3</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P₄  2  1  1</td>
<td>1 0 2</td>
<td>1 0 2</td>
</tr>
<tr>
<td>P₅  0  0  2</td>
<td>0 0 2</td>
<td>0 0 2</td>
</tr>
</tbody>
</table>

Given the set of pending requests, is there a safe sequence?

If no, deadlock
Detecting Deadlock

5 processes, 3 resources. We no longer know Max.

<table>
<thead>
<tr>
<th></th>
<th>Alloc</th>
<th>Avail</th>
<th>Pending</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$R_1$</td>
<td>$R_2$</td>
<td>$R_3$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$P_4$</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_5$</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Given the set of pending requests, is there a safe sequence?

- If no, deadlock

Can we avoid deadlock by delaying granting requests?

- Deadlock triggered when request formulated, not granted