**Advanced Synchronization and Deadlock**

- 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

---

**Contra Threads: Events**

*John Ousterhout: “Why Threads Are a Bad Idea (for most purposes)”*

- Casual
- All programmers
- Wizard
- Visual basic programmers
- C programmers
- C++ programmers
- Thread programmers

---

**Contra Threads: Events**

*John Ousterhout: “Why Threads Are a Bad Idea (for most purposes)”*

- Casual
- All programmers
- Wizard
- Visual basic programmers
- C programmers
- C++ programmers
- Thread programmers

---

**Event-driven Programming**

- No concurrency: one execution stream
- Register interest in events (callbacks)
- Wait for events; invoke (short-lived) handlers
- Complicated only for unusual cases
- Easier to debug
Multi-object synchronization

- Transfer $100 from account A to account B
  - A->subtract(100);
  - B->add(100);

- Fine-grain locking
  - Hash table:
    - put(key, value)  value = get(key)  value = remove(key)
    - one lock for whole table? one lock per bucket?

- Complexity vs Performance
  - Beware of premature optimizations!

Solutions: Careful class design

- You design the API!
  - Too Much Milk with 2 objects
    - Fridge  Fridge::checkForMilk(); Fridge::addMilk()
    - Note   Note::readNote(); noteWriteNote()
  - back to square one...
  - Instead
    - Fridge::checkForMilkAndSetNoteIfNeeded()
    - Fridge::addMilk()

- No panacea
  - still need to think carefully how objects interact

Solutions: Serialization

- Divide work into logically separate tasks

- Ensure serializable execution of tasks
  - tasks may execute concurrently...
  - ...but result of each task equivalent to what would be obtained if tasks executed one at a time in some serial order

- A few ways to get there
  - one big lock
  - lock-all/release-all
  - two phase locking

- Equivalent sequential execution

- Divide work into logically separate tasks

- Ensure serializable execution of tasks
  - tasks may execute concurrently...
  - ...but result of each task equivalent to what would be obtained if tasks executed one at a time in some serial order

- A few ways to get there
  - one big lock
  - lock-all/release-all
  - two phase locking
Solutions:
Serialization

- Divide work into logically separate tasks
- Ensure **serializable execution** of tasks
  - tasks may execute concurrently...
  - ...but result of each task equivalent to what would be obtained if tasks executed one at a time in some serial order
- A few ways to get there
  - one big lock
  - lock-all/release-all
  - **two phase locking**
    - Phase 1:
      - acquire locks
      - upgrade reader to writer
      - lock if necessary
    - Phase 2:
      - release locks
      - downgrade writer to reader
      - lock if necessary

Solutions:
ownership pattern

- **Shared container**
  - put things in; take them out; access them without a lock (own them)

Solution:
staged architecture

- Each stage has local state and some thread that operate on it
- No state shared across stages

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
- Nested waiting
Dining Philosophers

- N philosophers; N plates; N chopsticks
- If all philosophers grab right chopstick → deadlock!

Necessary conditions for 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

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”
    - Nested waiting?

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

E.W. Dijkstra & N. Habermann
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.

The Banker's books

- \( \text{Max}_i = \max \) amount of units of resource \( R_j \) needed by \( P_i \)
- \( \text{MaxClaim}_i = \sum_{j=1}^{m} \text{Max}_{ij} \)
- \( \text{Alloc}_{ij} = \) current allocation of \( R_j \) held by \( P_i \)
- \( \text{HasNow}_i = \sum_{j=1}^{m} \text{Alloc}_{ij} \)
- \( \text{Avail}_j = \) 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,

\[
\text{MaxClaim}_i - \text{HasNow}_i \leq \text{Avail} + \sum_{j=1}^{i-1} \text{HasNow}_j
\]

An Example

5 processes, 4 resources

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

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

<table>
<thead>
<tr>
<th>MaxRequest</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P5</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Is this a safe state?

- While safe sequence does not include all processes:
  - Is there a \( P_i \) such that MaxRequest \( \leq \) Avail?
    - if no, exit with unsafe
    - if yes, add \( P_i \) to the sequence and set Avail = Avail + HasNow;

- Exit with safe
An Example

- 5 processes, 4 resources
- P2 wants to change its allocation to \(0 4 2 0\)
- Safe?

<table>
<thead>
<tr>
<th>Max</th>
<th>Alloc</th>
<th>Avail</th>
<th>MaxRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R_1)</td>
<td>(R_2)</td>
<td>(R_3)</td>
<td>(R_4)</td>
</tr>
<tr>
<td>(P_1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(P_2)</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>(P_3)</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>(P_4)</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>(P_5)</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Detecting Deadlock

- 5 processes, 3 resources
- Given the set of pending requests, is there a safe sequence?
  - If no, deadlock

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

Given the set of pending requests, is there a safe sequence?
- If no, deadlock
- Deadlock triggered when request is formulated, not granted