Recap about Threads

- **Benefits:**
  - Simplify concurrent programming
  - Useful when there is a slow resource

- **Challenge:**
  - Share parts of address space
  - How to prevent undesired outcomes?

### Example

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i=0$</td>
<td>$i=0$</td>
</tr>
<tr>
<td>while ($i &lt; 10$) {</td>
<td>while ($i &gt; -10$) {</td>
</tr>
<tr>
<td>$i++$</td>
<td>$i--$</td>
</tr>
<tr>
<td>print “A finished”</td>
<td>print “B finished”</td>
</tr>
</tbody>
</table>

- Which thread will exit its while loop first?
- Is the winner guaranteed to print first?
- Is it guaranteed that someone will win?

### Debugging Multi-Threaded Programs

- Challenging due to non-deterministic interleaving
  - **Heisenbug**: a bug that occurs non-deterministically

- Something for you to worry about? **YES!!!**
  - Think Murphy’s Law
  - All possible interleavings must be correct

- Famous errors:
  - Over-radiation in Therac-25
  - Northeast blackout of 2003

### Synchronization

- Constrain interleavings between threads such that all possible interleavings produce a correct result
- Trivial solution?
- **Challenge:**
  - Constrain thread executions as little as possible
- **Insight:**
  - Some events are independent $\rightarrow$ order is irrelevant
  - Other events are dependent $\rightarrow$ order matters

### Too much milk

- **Problem definition**
  - Janet and Peter want to keep their refrigerator stocked with at most one milk jug
  - If either sees fridge empty, she/he goes to buy milk

- **Solution #0 (no synchronization)**
  - Peter
    ```
    if (!noMilk) {
      buy milk
    } else {
      Problems?
    }
    ```
  - Janet
    ```
    if (!noMilk) {
      buy milk
    } else {
      Problems?
    }
    ```

- **Race condition!**

### First type of synchronization: Mutual exclusion

- Ensure that only 1 thread is doing a certain thing at any moment in time
  - “Only 1 person goes shopping at a time”
  - Constrains interleavings of threads

- Does this remind you of any other concept we’ve talked about?
Critical section

- Section of code that needs to be run atomically with respect to selected other pieces of code
- Critical sections must be atomic w.r.t each other because they access a shared resource
- In our example, critical section is:
  - "if (no milk) buy milk"
  - How do we make this critical section atomic?

Too much milk (solution #1)

- Leave note that you're going to check on the milk, so other person doesn't also buy
  - Assume only atomic operations are load and store
  - Peter
    ```
    if (noNote) {
      leave note
      if (noMilk) {
        buy milk
      }
      remove note
    }
    ```
  - Janet
    ```
    if (noNote) {
      leave note
      if (noMilk) {
        buy milk
      }
      remove note
    }
    ```
  - Does this work?
  - Better solution than #0?

Too much milk (solution #2)

- Change the order of "leave note" and "check note"
- Notes need to be labelled (otherwise you'll see your note and think the other person left it)
  - Peter
    ```
    leave note
    if (no note) {
      if (noMilk) {
        buy milk
      }
      remove note
    }
    ```
  - Janet
    ```
    leave note
    if (no note) {
      if (noMilk) {
        buy milk
      }
      remove note
    }
    ```

Too much milk (solution #3)

- Decide who will buy milk when both leave notes at the same time. Peter hangs around to make sure job is done.
  - Peter
    ```
    leave note
    if (no note) {
      if (noMilk) {
        buy milk
      }
      remove note
    }
    ```
  - Janet
    ```
    leave note
    if (no note) {
      if (noMilk) {
        buy milk
      }
      remove note
    }
    ```
  - Peter's "while (noteJanet)" prevents him from entering the critical section at the same time as Janet

Proof of correctness

- Janet
  - if no note, then Peter hasn't started yet, so safe to buy
    - Peter will wait for Janet to be done before checking
  - if note, then Peter will eventually buy milk if needed
    - Note that Peter may be waiting for Janet to exit
- Peter
  - if no note, safe to buy
    - Already left note, which Janet will check
  - if note, Peter waits to see what Janet does and accordingly decides whether to buy

Analysis of solution #3

- Good
  - It works!
  - Relies on simple atomic operations
- Bad
  - Complicated; not obviously correct
  - Asymmetric
  - Not obvious how to scale to three people
  - Peter consumes CPU time while waiting
    - Called busy-waiting
Higher-level synchronization

- Raise the level of abstraction to make life easier for programmers

Concurrent programs

Applications

Higher-level synchronization primitives
(lock, monitor, semaphore)

Operating System

Atomic operations
(load/store, interrupt enable/disable, test&set)

Atomic operations

Hardware

Locks (mutexes)

- A lock prevents another thread from entering a

Why was the note in Too much milk (solutions #1 and #2) not a good lock?

- Two operations
  - lock(): wait until lock is free, then acquire it
    ```
    do {
      if (lock is free) {
        acquire lock
        break
      }
    } while (1)
    ```
  - unlock(): release lock

Locks (mutexes)

- Lock usage
  - Initialized to free
  - Thread acquires lock before entering critical section (waiting if needed)
  - Thread that has acquired lock should release when done with critical section

- All synchronization involves waiting
- Thread can be running or blocked

Efficiency

- But this prevents Janet from doing things while Peter is buying milk
- How to minimize the time the lock is held?

Peter

```java
milk.lock();
if (noMilk) {
  buy milk
} milk.unlock()
```

Janet

```java
milk.lock();
if (noMilk) {
  buy milk
} milk.unlock()
```