Recap: Processes

● Hardware interface:

\[
\text{app1} + \text{app2} + \text{app3} \\
\text{CPU + memory}
\]

● OS interface:

\[
\text{app1} \\
\text{CPU + memory}
\]

\[
\text{app2} \\
\text{CPU + memory}
\]

\[
\text{app3} \\
\text{CPU + memory}
\]
Recap: Threads

● Benefits:
  ◆ Simplify concurrent programming
  ◆ Useful when there is a slow resource

● Challenge:
  ◆ Share parts of address space
  ◆ How to prevent undesired outcomes?
Non-deterministic ordering \( \rightarrow \) Non-deterministic results

- **Arithmetic example (y is initially 10)**
  - Possible results?
    - If A runs first: \( x = 11 \) and \( y = 20 \)
    - If B runs first: \( x = 21 \) and \( y = 20 \)

- **Another example (x is initially 0)**
  - Possible results?
    - \( x = 1 \) or \(-1\)
  - Impossible results?
    - \( x = 0 \)
Atomic operations

- To reason about cooperating threads, we must know which operations are **atomic**
  - Effects of operation are seen in entirety, or not at all

- Most computers:
  - Memory load and store are atomic
  - Many other instructions are not atomic
    - Example: double-precision floating point
  - Need an atomic op to build a bigger atomic op
**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>}</td>
<td>}</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 thread that exits loop first guaranteed to print first?
- Is it guaranteed that someone will print?
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

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

- All possible interleavings must be correct
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
  - Alice and Bob want to always have one can of milk
  - No room for two cans of milk
  - Whoever sees fridge empty goes to buy milk

- Solution #0 (no synchronization)

  ```
  Bob
  if (noMilk) {
    buy milk
  }

  Alice
  if (noMilk) {
    buy milk
  }

  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 must 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

```plaintext
Bob
if (noNote) {
    leave note
    if (noMilk) {
        buy milk
    }
    remove note
}

Alice
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

```
Bob
  leave noteBob
  if (no noteAlice) {
    if (noMilk) {
      buy milk
    }
  }
  remove noteBob

Alice
  leave noteAlice
  if (no noteBob) {
    if (noMilk) {
      buy milk
    }
  }
  remove noteAlice
```

Problems?
Announcements

- First project is out
  - Due on Jan. 29th
  - Office hour schedule on Google calendar on web page
  - Get familiar with git, gdb, valgrind, etc.

- Mid-term: 6:30-8:30pm on February 20th
Too much milk (solution #3)

- Decide who will buy milk when both leave notes at the same time. Bob hangs around to make sure job is done.

**Bob**

leave noteBob

while (noteAlice) {
    do nothing
}

if (noMilk) {
    buy milk
}

remove noteBob

**Alice**

leave noteAlice

if (no noteBob) {
    if (noMilk) {
        buy milk
    }
}

remove noteAlice

- Bob’s “while (noteAlice)” prevents him from entering the critical section at the same time as Alice
Proof of correctness

- Alice
  - if no noteBob, then Bob hasn’t started yet, so safe to buy
    - Bob will wait for Alice to be done before checking
  - if noteBob, then Bob will eventually buy milk if needed
    - Note that Bob may be waiting for Alice to exit

- Bob
  - if no noteAlice, safe to buy
    - Already left noteBob, which Alice will check
  - if noteAlice, Bob waits to see what Alice 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
  - Bob consumes CPU time while waiting
    » Called **busy-waiting**
Higher-level synchronization

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

![Diagram showing the relationship between Higher-level synchronization primitives and Atomic operations within the context of Concurrent programs.]

- Concurrent programs
- Higher-level synchronization primitives (lock, monitor, semaphore)
- Atomic operations (load/store, interrupt enable/disable, test&set)
Locks (mutexes)

- A lock prevents another thread from entering a critical section
  - “Lock fridge while checking milk status and shopping”

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

A  lock  prevents  another  thread  from  entering  a  critical  section
Locks (mutexes)

- A lock prevents another thread from entering a critical section

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**
  - Initialize to free
  - Acquire lock before entering critical section
  - Release lock when done with critical section

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

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

Alice
milk.lock()
if (noMilk) {
    buy milk
}
milk.unlock()
```
Efficiency

- But this prevents Alice from doing things while Bob is buying milk

- How to minimize the time the lock is held?

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

Alice
milk.lock()
if (noMilk) {
    buy milk
}
milk.unlock()
```
Efficiency

- Use lock to protect posting/looking up of note

```java
note.lock()
if (noNote) {
    leave note
    note.unlock()
    if (noMilk) {
        buy milk
    }
    note.lock()
    remove note
    note.unlock()
}
else {
    note.unlock()
}
```