EECS 482
Introduction to Operating Systems

Winter 2019

Manos Kapritsos

Thanks to Harsha Madhyastha and Peter Chen for the slides and notes
Recap: Processes

- **Hardware interface:**

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

- **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?

![Diagram showing stack for different threads and a data segment with code]
Example

Thread A

\[ i=0 \]

While (\( i < 10 \)) {
    \[ i++ \]
}\n
Print “A finished”

Thread B

\[ i=0 \]

While (\( i > -10 \)) {
    \[ i-- \]
}\n
Print “B finished”

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

- If both threads run at the same speed and start within a few instructions of each other, are they guaranteed to loop forever?

<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>
Atomic operations

Before we can reason at all about cooperating threads, we must know that some operation is **atomic**
- Indivisible, i.e., happens in its entirety or not at all
- No events from other threads can occur in between

Most computers:
- Memory load and store are atomic
- Many other instructions are not atomic
  - Example: double-precision floating point
- Need an atomic operation to build a bigger atomic operation
Debugging Multi-Threaded Programs

- Challenging due to non-deterministic interleaving
  - **Heisenbug**: a bug that occurs non-deterministically
    » (and your program will be breaking badly soon enough).
- 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
Announcements

- First project is out
  - Due in 2.5 weeks (Feb. 4th)
  - Read the spec carefully!
  - Office hour schedule on Google calendar on web page
  - Get familiar with git, gdb, valgrind, etc.

- Check out Piazza if looking for project group

- Discussion section questions for this Friday posted

- Send me your picture if you haven’t already
Too much milk

- Problem definition
  - Kostas and Manos want to keep their refrigerator stocked with at most one milk jug
  - If either sees fridge empty, he goes to buy milk
- Solution #0 (no synchronization)

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

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

**Manos**

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

**Kostas**

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

```java
Manos
leave noteManos
if (no noteKostas) {
  if (noMilk) {
    buy milk
  }
}
remove noteManos

Kostas
leave noteKostas
if (no noteManos) {
  if (noMilk) {
    buy milk
  }
}
remove noteKostas
```
Too much milk (solution #3)

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

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

- **Manos**
  - if no noteKostas, safe to buy
    - Already left noteManos, which Kostas will check
  - if noteKostas, Manos waits to see what Kostas does and accordingly decides whether to buy
Analysis of solution #3

- Good
- Bad
Higher-level synchronization

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

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
    ```
    do {
      if (lock is free) {
        acquire lock
        break
      }
    } while (1)
    ```
  - unlock(): release lock
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)

- How to use a lock
  - 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

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

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

- But this prevents Kostas from doing things while Manos is buying milk

- How to minimize the time the lock is held?
Efficiency

- Use lock to protect posting/looking up of note