http://knowyourmeme.com/memes/mind-blown
Recap: Processes

- Hardware interface:

  \[
  \begin{array}{c}
  \text{app1+app2+app3} \\
  \text{CPU + memory}
  \end{array}
  \]

- OS interface:

  \[
  \begin{array}{ccc}
  \text{app1} & \text{app2} & \text{app3} \\
  \text{CPU + memory} & \text{CPU + memory} & \text{CPU + memory}
  \end{array}
  \]
Recap: Threads

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

● Challenge:
  • Share parts of address space
  • How to prevent undesired outcomes?
Review of Stack Frames

A(int tmp) {
    B(tmp); 
}

B(int val) {
    C(val, val + 2);
    A(val - 1);
}

C(int foo, int bar) {
    int v = bar - foo;
}

A(tmp = 1)
   B(val = 1)
       C(foo, bar = 3)
Example

 Thread A
 i=0
 while (i < 10) {
   i++
 }
 print "A finished"

 Thread B
 i=0
 while (i > -10) {
   i--
 }
 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

Thread A
i=0
while (i < 10) {
    i++
}
print “A finished”

Thread B
i=0
while (i > -10) {
    i--
}
print “B finished”

- If both threads run at the same speed and start within a few instructions, are they guaranteed to loop forever?
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
Therac 25
Northeastern Blackout
Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel.
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 → order is irrelevant
  - Other events are dependent → order matters
Announcements

● First project is out
  • Due on Feb. 4th
  • Office hour schedule on Google calendar on web page
  • Get familiar with git, gdb, valgrind, etc.

● Midterm will be on March 1st
  • 10am-12pm

● 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

- Tia and Baris want to keep their refrigerator stocked with at most one milk jug
- If either sees fridge empty, she/he goes to buy milk
Too much milk

● Problem definition
  • Tia and Baris 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)

Baris
if (noMilk) {
  buy milk
}

Tia
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 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

\[
\begin{align*}
\text{Baris} & \quad \text{if (noNote) } \{ \\
& \quad \quad \text{leave note} \\
& \quad \quad \text{if (noMilk) } \{ \\
& \quad \quad \quad \text{buy milk} \\
& \quad \quad \} \\
& \quad \quad \text{remove note} \\
& \quad \} \\
\text{Tia} & \quad \text{if (noNote) } \{ \\
& \quad \quad \text{leave note} \\
& \quad \quad \text{if (noMilk) } \{ \\
& \quad \quad \quad \text{buy milk} \\
& \quad \quad \} \\
& \quad \quad \text{remove note} \\
& \} \\
\end{align*}
\]

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)

Baris

---
leave noteBaris

if (no noteTia) {
  if (noMilk) {
    buy milk
  }
}
remove noteBaris

Tia

---
leave noteTia

if (no noteBaris) {
  if (noMilk) {
    buy milk
  }
}
remove noteTia

Problems?
Too much milk (solution #3)

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

Baris

leave noteBaris
while (noteTia) {
  do nothing
}
if (noMilk) {
  buy milk
}
remove noteBaris

Tia

leave noteTia

if (no noteBaris) {
  if (noMilk) {
    buy milk
  }
} remove noteTia

- Baris’s “while (noteTia)” prevents him from entering the critical section at the same time as Tia
Proof of correctness

● Tia
  • if no noteBaris, then Baris hasn’t started yet, so safe to buy
    » Baris will wait for Tia to be done before checking
  • if noteBaris, then Baris will eventually buy milk if needed
    » Note that Baris may be waiting for Tia to exit

● Baris
  • if no noteTia, safe to buy
    » Already left noteBaris, which Tia will check
  • if noteTia, Baris waits to see what Tia 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 code
  - Not obvious how to scale to three people
  - Baris consumes CPU time while waiting
    - Called *busy-waiting*
Higher-level synchronization

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

Operating System

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

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

Concurrent programs

Applications
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
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
    
    ```c
    do {
        if (lock is free) {
            acquire lock
            break
        }
    } while (1)
    ```
    
    Atomic
  
  - `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

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

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

- But this prevents Tia from doing things while Baris is buying milk

- How to minimize the time the lock is held?
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()
}
```