Synchronization

CS 4410
Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, E. Sirer, R. Van Renesse]
• Foundations
• Semaphores
• Monitors & Condition Variables
Synchronization Foundations

- Race Conditions
- Critical Sections
- Example: Too Much Milk
- Basic Hardware Primitives
- Building a SpinLock
Recall: Process vs. Thread

Process:
- Privilege Level
- Address Space
- Code, Data, Heap
- Shared I/O resources

One or more Threads:
- Stack
- Registers
- PC, SP

Shared amongst threads
Two Theads, One Variable

2 threads updating a shared variable **amount**
- One thread wants to decrement amount by $10K
- Other thread wants to decrement amount by 50%

What happens when both threads are running?

Memory: `amount = 100,000`
Two Threads, One Variable

Might execute like this:

T1

. . .

r1 = load from amount
r1 = r1 - 10,000
store r1 to amount
. . .

T2

. . .

r2 = load from amount
r2 = 0.5 * r2
store r2 to amount
. . .

Memory

amount 40,000

Or vice versa (T1 then T2 → 45,000)…

either way is fine…
Two Threads, One Variable

Or it might execute like this:

T1

. . .
r1 = load from amount
r1 = r1 - 10,000
store r1 to amount
. . .

T2

. . .
r2 = load from amount
. . .
r2 = 0.5 * r2
store r2 to amount
. . .

Memory

amount 50,000

Lost Update!

Wrong ..and very difficult to debug
Race Conditions

= timing dependent error involving shared state

• Once thread A starts, it needs to “race” to finish
• Whether race condition happens depends on thread schedule
  • Different “schedules” or “interleavings” exist (total order on machine instructions)

All possible interleavings should be safe!
Problems with Sequential Reasoning

1. Program execution depends on the possible interleavings of threads’ access to shared state.

2. Program execution can be nondeterministic.

3. Compilers and processor hardware can reorder instructions.
Race Conditions are Hard to Debug

- Number of possible interleavings is huge
- Some interleavings are good
- Some interleavings are bad:
  - But bad interleavings may rarely happen!
  - Works 100x ≠ no race condition
- Timing dependent: small changes hide bugs
Example: Races with Queues

- 2 concurrent enqueue() operations?
- 2 concurrent dequeue() operations?

What could possibly go wrong?
Critical Section

Must be atomic due to shared memory access

\begin{align*}
\text{T1} & : \ldots \quad \text{CSEnter()}; \\
& \quad \text{Critical section} \\
& \quad \text{CSExit()}; \\
& \quad \ldots \\
\text{T2} & : \ldots \quad \text{CSEnter()}; \\
& \quad \text{Critical section} \\
& \quad \text{CSExit()}; \\
& \quad \ldots 
\end{align*}

Goals

**Safety:** 1 thread in a critical section at time

**Liveness:** all threads make it into the CS if desired

**Fairness:** equal chances of getting into CS

... in practice, fairness rarely guaranteed
Too Much Milk: Safety, Liveness, and Fairness with no hardware support
Too Much Milk Problem

2 roommates, fridge always stocked with milk
• fridge is empty → need to restock it
• don’t want to buy too much milk

Caveats
• Only communicate by a notepad on the fridge
• Notepad has cells with names, like variables:

    out_to_buy_milk

    0

**TASK:** Write the pseudo-code to ensure that at most one roommate goes to buy milk
Solution #1: No Protection

Safety: Only one person (at most) buys milk
Liveness: If milk is needed, someone eventually buys it.
Fairness: Roommates equally likely to go to buy milk.

Safe? Live? Fair?
Solution #2: add a boolean flag

outtobuymilk initially false

T1

while(outtobuymilk):
    do_nothing();
if fridge_empty():
    outtobuymilk = 1
buy_milk()
outtobuymilk = 0

T2

while(outtobuymilk):
    do_nothing();
if fridge_empty():
    outtobuymilk = 1
buy_milk()
outtobuymilk = 0

Safety: Only one person (at most) buys milk
Liveness: If milk is needed, someone eventually buys it.
Fairness: Roommates equally likely to go to buy milk.
Safe? Live? Fair?
Solution #3: add two boolean flags!

one for each roommate (initially false):

blues\_got\_this, reds\_got\_this

**T1**

\[
\begin{align*}
\text{blues\_got\_this} &= 1 \\
\text{if} \not\exists \text{reds\_got\_this} \text{ and fridge\_empty}(): \\
&\quad \text{buy\_milk()} \\
\text{blues\_got\_this} &= 0
\end{align*}
\]

**T2**

\[
\begin{align*}
\text{reds\_got\_this} &= 1 \\
\text{if} \not\exists \text{blues\_got\_this} \text{ and fridge\_empty}(): \\
&\quad \text{buy\_milk()} \\
\text{reds\_got\_this} &= 0
\end{align*}
\]

**Safety:** Only one person (at most) buys milk

**Liveness:** If milk is needed, someone eventually buys it.

**Fairness:** Roommates equally likely to go to buy milk.

Safe? Live? Fair?
Solution #4: asymmetric flags!

one for each roommate (initially false):

\[ \textit{blues\_got\_this}, \textit{reds\_got\_this} \]

```
blues\_got\_this = 1
while reds\_got\_this:
    do_nothing()
if fridge\_empty():
    buy\_milk()
blues\_got\_this = 0
```

```
reds\_got\_this = 1
if not blues\_got\_this:
    if fridge\_empty():
        buy\_milk()
    reds\_got\_this = 0
```

Safe? Live? Fair?

- complicated (and this is a simple example!)
- hard to ascertain that it is correct
- asymmetric code is hard to generalize & unfair
Last Solution: Peterson’s Solution

another flag turn \{blue, red\}

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>blues_got_this = 1</td>
<td>reds_got_this = 1</td>
</tr>
<tr>
<td>turn = red</td>
<td>turn = blue</td>
</tr>
<tr>
<td>while (reds_got_this and turn==red):</td>
<td>while (blues_got_this and turn==blue):</td>
</tr>
<tr>
<td>do_nothing()</td>
<td>do_nothing()</td>
</tr>
<tr>
<td>if fridge_empty():</td>
<td>if fridge_empty():</td>
</tr>
<tr>
<td>buy_milk()</td>
<td>buy_milk()</td>
</tr>
<tr>
<td>blues_got_this = 0</td>
<td>reds_got_this = 0</td>
</tr>
</tbody>
</table>

Safe? Live? Fair?
– complicated (and this is a simple example!)
– hard to ascertain that it is correct
– hard to generalize
Hardware Solution

- HW primitives to provide mutual exclusion
- A **machine instruction** (part of the ISA!) that:
  - Reads & updates a memory location
  - Is atomic (other cores can’t see intermediate state)
- Example: Test-And-Set
  1 instruction with the following semantics:

```c
ATOMIC int TestAndSet(int *var) {
    int oldVal = *var;
    *var = 1;
    return oldVal;
}
```

sets the value to 1, returns former value
Buying Milk with TAS

Shared variable: int buyingmilk, initially 0

T1
while(TAS(&buyingmilk))
   do_nothing();
   if fridge_empty():
      buy_milk()
buyingmilk := 0

T2
while(TAS(&buyingmilk))
   do_nothing();
   if fridge_empty():
      buy_milk()
buyingmilk := 0

A little hard on the eyes. Can we do better?
Enter: Locks!

```c
acquire(int *lock) {
    while(test_and_set(lock))
        /* do nothing */;
}

release(int *lock) {
    *lock = 0;
}
```
Buying Milk with Locks

Shared lock: int buyingmilk, initially 0

T1

acquire(&buyingmilk);
if fridge_empty():
    buy_milk()
release(&buyingmilk);

T2

acquire(&buyingmilk);
if fridge_empty():
    buy_milk()
release(&buyingmilk);

Now we’re getting somewhere!
Is anyone not happy with this?
Thou shalt not busy-wait!
Not just any locks: SpinLocks

Participants not in critical section must spin
→ wasting CPU cycles
• Replace the “do nothing” loop with a “yield()”?
• Threads would still be scheduled and descheduled
  (context switches are expensive)

Need a better primitive:
• allows one thread to pass through
• all others sleep until they can execute again