Too Much Milk

You

- Arrive home
- Look in the fridge; out of milk
- Leave for store
- Buy milk
- Arrive home; put milk away

Your roommate

- Arrive home
- Look in the fridge; out of milk
- Leave for store
- Buy milk
- Arrive home; put milk away
- Oh, no!

Too Much Milk

- What is the intention
  - Only one person buys milk at a time
  - Someone buys milk if you need it

- Correctness properties

- What went wrong?
  - Lack of communication/coordination

Formalizing Too Much Milk

- Shared variable
  - “Look in fridge for milk”: read the value of a variable
  - “Add milk”: increment a variable

- Safety
  - “nothing bad happens”
  - Holds for every finite execution prefix

- Liveness
  - “something good will eventually happen”
  - No partial execution is irremediable

- Bounded wait time
Too Much Milk in C

```c
static const char* P[2] = {"you", "your roommate");
int milk = 0;

void* person(void* arg)
{
    uintptr_t i = (uintptr_t) arg;
    printf("%s looks into the fridge\n", P[i]);
    if (milk == 0)
        printf("%s: no milk, I go and buy some\n", P[i]);
    milk++;
} else {
    printf("%s: there is milk, no need to buy more milk\n", P[i]);
}
return 0;
}
```

```c
int main()
{
    // start two threads
    // wait for them to join
    printf("we ended up with %d bottles of milk\n", milk);
    return 0;
}
```

Race Condition

- Outcome depends on the sequence or timing of the individual instructions

Race Condition

- Observer effect: “The act of observing a system alters its state”

Examples:
- Console output
- Running with debugging enabled
- Changing optimization level

Concurrency bugs are (unfortunately) often Heisenbugs

Heisenbugs

- Does it work?
- Safe?
- Live?

Too Much Milk – Solution 1

```c
if (milk == 0 && !note)
{
    note = 1; // leave note
    milk++;// buy milk
    note = 0; // remove note
}
```
Too Much Milk – Solution 2

You

\[\text{noteA} = 1; \text{ // leave note}\]
\[\text{if (!noteB)}\]
\[\{\text{if (milk==0)}\]
\[\text{milk++; // buy milk}\]
\[\}\text{noteA} = 0;\]

Your roommate

\[\text{noteB} = 1; \text{ // leave note}\]
\[\text{if (!noteA)}\]
\[\{\text{if (milk==0)}\]
\[\text{milk++; // buy milk}\]
\[\}\text{noteB} = 0;\]

Does it work?

Safe?

Live?

Solution 3 in Detail

Your roommate

\[\text{noteB} = 1; \text{ // leave note}\]
\[\text{if (!noteA)}\]
\[\{\text{if (milk==0)}\]
\[\text{milk++; // buy milk}\]
\[\}\text{noteB} = 0;\]

Two cases:

- No noteA
  - safe to check and buy milk since other thread has not yet started buying milk
  - noteA is present
  - Other thread is checking for milk and buying if needed or waiting for this thread to quit so this thread quits by removing its note

Solution 3 in Detail

You

\[\text{noteA} = 1; \text{ // leave note}\]
\[\text{while (noteB)}\]
\[\text{if (milk==0)}\]
\[\{\text{milk++; // buy milk}\]
\[\}\text{noteA} = 0;\]

Two cases:

- No noteB
  - It is safe to buy milk since other thread has not yet started
  - noteB is present
  - This thread waits until the other thread has left its note and then decided whether to buy milk or not
Solution – Yes, but is it Good?

- Safe?
  - Yes, but it is too complicated – it’s hard to convince ourselves it really works.
- Live?
  - Yes.
- Bounded?
  - Yes for B, No for A.
- So it is asymmetric
- Can’t be generalized to >2 threads.
- Involves busy waiting

Fundamental Issue: Atomicity

- What we would like to have is an atomic operation
  - “All or nothing”
  - Executes to completion without any interruption or (partial) failure
- Checking for milk and then incrementing (buying) was not atomic
  - Multiple instructions
- What we have is a critical section
  - Sequence of instructions
  - All code within the critical section needs to execute atomically

Mutual Exclusion

- What we have achieved with solution 3 is mutual exclusion
  - At most one thread can be in the critical section at the same time
- In order to be correct critical sections need our previous properties and one more:
  - Safety
  - Liveness
  - Bounded waiting
  - Failure atomicity
  - Thread can fail in the critical section without affecting other threads

Too Much Milk – Solution 3

```c
You

```

```c
Your roommate

```

Summary

- Threads provide us with concurrent execution but also require synchronizing access to shared data structure
- We can achieve mutual exclusion
- Scoreboard for our solutions:

<table>
<thead>
<tr>
<th></th>
<th>Solution 1</th>
<th>Solution 2</th>
<th>Solution 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Live</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Bounded Wait Time</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Petersons Algorithm

- Solution 3 was a simplified version of this:

```c
// Flag: in boolean array; and turn is an integer
flag[0] = false;
flag[1] = false;
turn:

P0: flag[0] = true;
turn = 1;
while (flag[1] == true && turn == 1) {
    // busy wait
    // critical section
    // end of critical section
    flag[0] = false;
}

P1: flag[1] = true;
turn = 0;

while (flag[0] == true && turn == 0) {
    // busy wait
    // critical section
    // end of critical section
    flag[1] = false;
}
```

- It satisfies all 3 conditions!
- But implementing it, as is, in C doesn't work!

Taming Complexity

- Concurrency and Mutual Exclusion are quite common
- Fortunately, there are established solutions
  - Locks
  - Semaphores
  - Condition Variables
  - Monitors
  - …