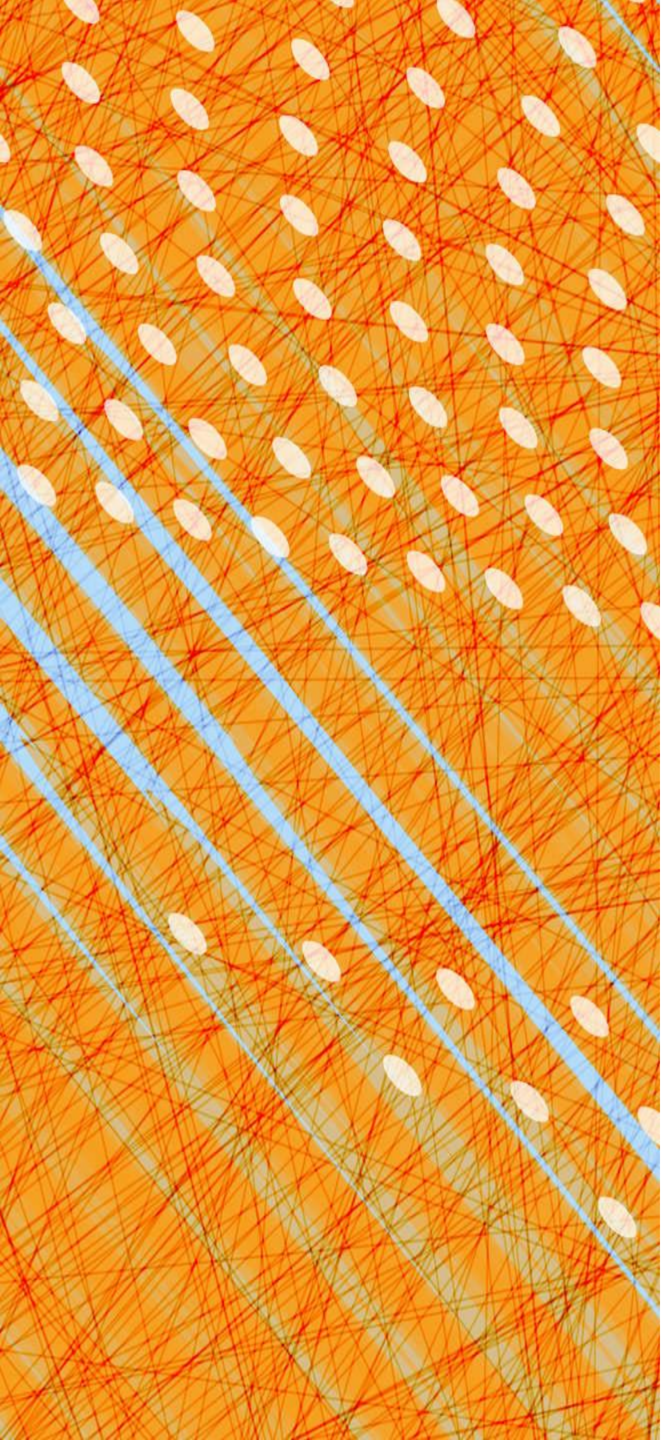
Synchronization

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:

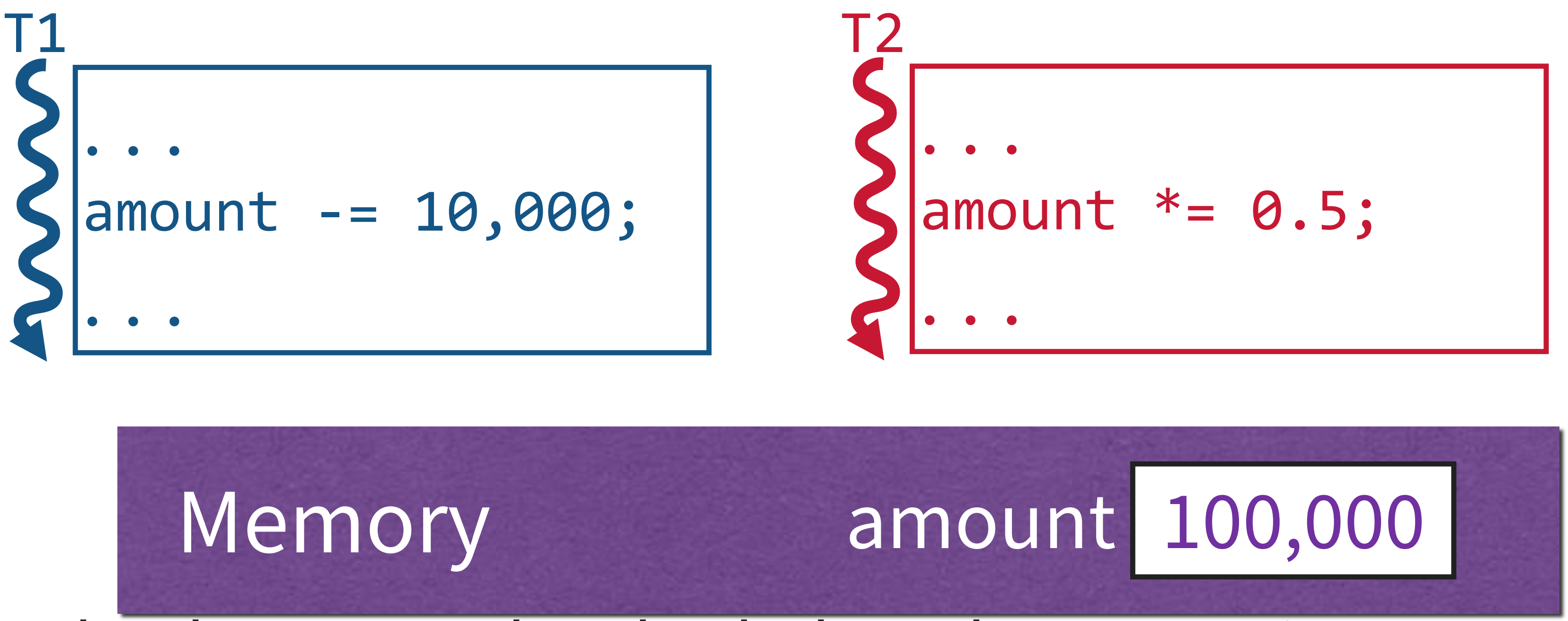
- Privilege Level
- Address Space
- Code, Data, Heap
- Shared I/O resources
- One or more Threads:
  - Stack
  - Registers
  - PC, SP

Shared amongst threads (applies to: Privilege Level; Address Space; Code, Data, Heap; Shared I/O resources)

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?

Two Theads, One Variable

Might execute like this:

T2

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

T1

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

Memory amount 40,000

Or vice versa (T1 then T2 → 45,000)…
either way is fine…

Two Theads, 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.

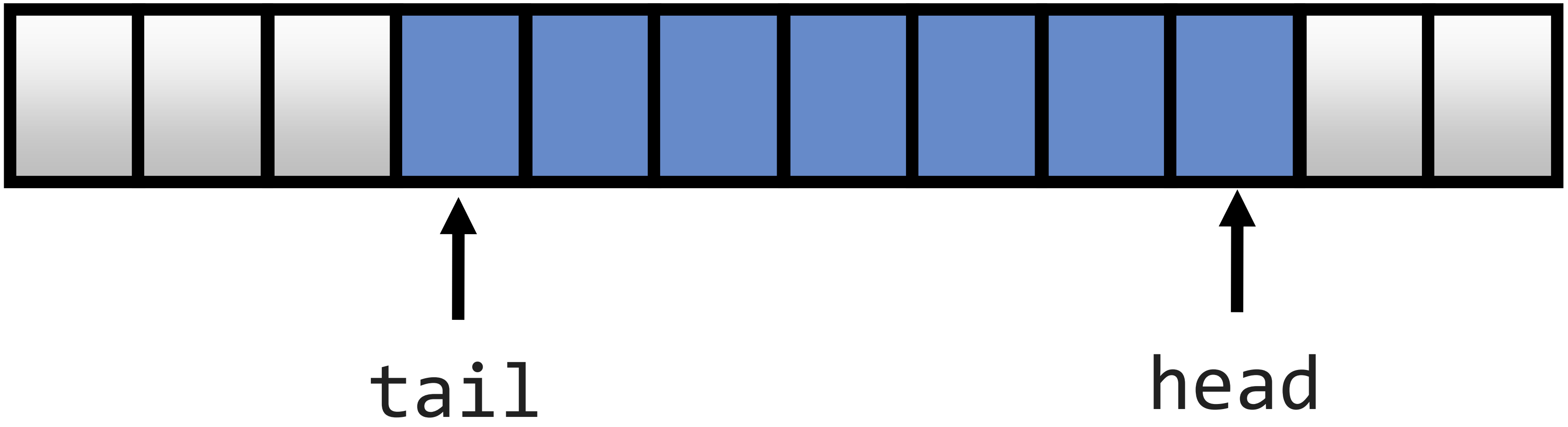
3. Compilers and processor hardware can reorder instructions.

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

T1

```
. . .
CSEnter();
  Critical section
CSExit();
. . .
```

T2

```
. . .
CSEnter();
  Critical section
CSExit();
. . .
```

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

T1

```
if fridge_empty():
    buy_milk()
```

T2

```
if fridge_empty():
    buy_milk()
```

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

```
blues_got_this = 1
if !reds_got_this and
      fridge_empty():
   buy_milk()
blues_got_this = 0
```

T2

```
reds_got_this = 1
if !blues_got_this and
      fridge_empty():
   buy_milk()
reds_got_this = 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 #4: asymmetric flags!

one for each roommate (initially false):

blues_got_this, ***reds_got_this***

T1

```
blues_got_this = 1
while reds_got_this:
    do_nothing()
if fridge_empty():
   buy_milk()
blues_got_this = 0
```

T2

```
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}`

T1

```
blues_got_this = 1
turn = red
while (reds_got_this
   and turn==red):
     do_nothing()
if fridge_empty():
   buy_milk()
blues_got_this = 0
```

T2

```
reds_got_this = 1
turn = blue
while (blues_got_this
   and turn==blue):
     do_nothing()
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

– 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:

```
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!

```
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: **Spin**Locks

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