Agenda
- Threads Interleaving
- locks

```c
#include <stdio.h>
#include <pthread.h>

int global_x = 0;

void* increment() {
    global_x += 1;
    return NULL;
}

int main(int argc, char** argv) {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, increment, NULL);
    pthread_create(&tid1, NULL, increment, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("%d\n", global_x); // minimum? maximum?
    return 0;
}
```
```c
int global_x = 0;
pthread_t tids[2];

void* increment100() {
    for (int i=0; i<100; i++) {
        global_x = 1;
    }
    return NULL;
}

int main(int argc, char** argv) {
    for (int i=0; i<2; i++) {
        pthread_create(&tids[i], NULL, increment100, NULL);
    }
    for (int i=0; i<2; i++) {
        pthread_join(tids[i], NULL);
    }
    printf("%d\n", global_x); // minimum? maximum?
    return 0;
}
```

Nondeterministic (based on the scheduling order)

Race conditions → changing behaviors based on timing or orderings

Race conditions → changing behaviors based on timing or orderings

How to get 2?

1. Read x = 0
2. Execute 100 iterations
3. x = 100
4. Write x = 1
5. Read x = 1
6. Execute to completion
7. Write x = 100
8. Add 1 to reg
9. Write x = 99
10. Write x = 1
11. Write x = 2
12. How to get 100?
Problem: read & update variable might be interrupted.

Common pattern: if (flag) & <do something> modify flag

Too Much Milk

Goals:
1. If there’s no milk, someone gets milk
2. No more than 1 milk in the fridge.

**Roommate A**

if (no milk) {
    buy milk;
}

**Roommate B**

if (no milk) {
    buy milk;
}

goal 1: ✓
goal 2: ✗
Attempt 2

Roommate A

if (no milk) {
  if (no note) {
    buy milk;
  }
  leave note;
  buy milk;
}

Roommate B

if (no milk) {
  if (no note) {
    buy milk;
  }
}
goal 1 ✓
goal 2 ✗
Roommate A

if (no milk)
    lock the fridge
    go buy milk

Roommate B

check fridge if not locked
if (no milk)
    buy milk

☆ We need to have exclusive access to the fridge when performing operations related to the fridge (see next page)
Too Much Milk w/ Locks

**Roommate A**
- lock the fridge
  - if (no milk)
  - buy milk;
- unlock the fridge

**Roommate B**
- lock the fridge
  - if (no milk)
  - buy milk;
- unlock the fridge

- only one person has access to the fridge at any time

Don't put everything in the critical section!

Accessing shared variables need protection! (all shared var ?)
Locks

→ API: acquire(), release()

→ Mechanism to enable critical section

→ Lock should provide:

  1) Mutual Exclusion: only one thread can access critical section at a time

  2) Progress: if no one is in the critical section, someone can get in

  3) Bounded Waiting: there’s an upperbound to your waiting

→ often not guaranteed by most locks, cause it’s hard to provide.
2 Types of locks

① Spinlocks

- Spin in a loop trying to grab the lock → busy waits until you can grab the lock
- Relies on an atomic read modify write instr (test & set)

(consumes CPU)

[ test & set: a single instr that takes a memory address, checks if the value at addr is 0, if so, sets it to 1 and returns the value read ]

⇒ in this case the value indicates whether the lock is free or not! if multiple threads call test & set only one of them will be able to set the value to 1, the rest will fail.

⇒ release sets the value to 0.
② SLEEP LOCK / Mutex

- sleeps / blocks until you can grab the lock
- needs to keep track of threads waiting for the lock
- wake up a waiter on release

☆ won\'t be scheduled while waiting for the lock!

---

Trade-offs btwn spinlock vs. sleep lock

Spinlock
- wastes CPU while waiting
- what if there are lots of threads waiting?
- what if lock is released very quickly?

Sleeplock
- blocks while waiting (context switch involved)

☆ Places where you can't sleep: scheduler, interrupt handler.