10/17

Agenda

- Threads Interleaving
- Locks!

min:

1

t_1	t_2
read x (0)	
	read x (0)
	add 1 (1)
	write 1 to x
add 1 (1)	
write 1 to x.	

```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;
}
```

`global_x += 1;` → 3 instr.:
- read x to a reg
- add 1 to reg
- write reg to x

```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;
}
```

{3 instr.

100

200

Nondeterministic
(based on the scheduling order)

Race conditions
-> changing behaviors based on timing or orderings

How to get 100?

t1	t2
read x = 0 (reg)
	execute 100 iterations
	write x = 100
add 1 to reg.
writes x = 1
execute 100 iterations
x = 100.

How to get 2?

t1	t2
read x (0)
	execute 99 iter.
	write x = 99
add 1 to reg
write 1 to x
	read x = 1
execute to completion
write x = 100
	add 1 to reg
	write x = 2

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	Roommate B
if (no milk) {	
	if (no milk) {
	buy milk;
	}
buy milk;	
}	

goal 1 ✓

goal 2 ✗

Attempt 2

Roommate A

```
if (no milk) {



    leave note;
    buy milk;
}
```

Roommate B

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

goal 1 ✓

goal 2 ✗

Attempt 3

Roommate A	Roommate B
if (no milk) {	check fridge if not locked
lock the fridge.	if (no milk) {
go buy milk	buy milk;
}	}

Does this actually work?

☆ 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

(lock is a synchronization primitive)

(if (no milk) / buy milk; } critical section)

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. (consumes CPU)

→ busy waits until you can grab the lock

→ relies on an atomic read modify write instr (test&set)

[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]

Special instr. that guarantees all these steps are done in 1 instruction

☆ 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 Sleeplock / 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. sleeplock

spinlock	sleeplock
→ wastes CPU while waiting	→ blocks while waiting (context switch involved)

→ what if there are lots of threads waiting?

→ what if lock is released very quickly?

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