4/10/24 Threads Wrap-up & Locks

→ Thread Execution

```
int x = 0 ; // global var

thread_func() {
    if (x < 1) {
        x++;
    }
}
```

SS

value of x at the end

→ 1 : t1 finishes then t2 runs

→ 2 :

```
t1                    t2
if(x<1){
                      if(x<1){
                        x++; }
x++; }
```

☆ time of check to time of use!

→ race condition = scheduling orders cause semantically different results

```c
int global_x = 0;

void* thread_func() {
  global_x++;
  return NULL;
}

int main() {
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL, thread_func, NULL);
  pthread_create(&tid2, NULL, thread_func, NULL);

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

  printf("global_x: %d\n", global_x);

  return 0;
}
```

t1 t2

load x into reg
add 1 to reg
write reg back to memory of x

Value of global_x ?

1 or 2.

t1	t2
load x=0	
	load x=0
add 1	
	add 1.
write 1 back to x	
	write 1 back to x

```c
int global_x = 0;
pthread_t tids[100];

void* thread_func() {
  global_x++;
  return NULL;
}

int main() {
  for (int i=0; i<100; i++) {
    pthread_create(&tids[i], NULL, thread_func, NULL);
  }
  printf("global_x: %d\n", global_x); // minimum? maximum?

  for (int i=0; i<100; i++) {
    pthread_join(tids[i], NULL);
  }
  printf("global_x: %d\n", global_x); // minimum? maximum?

  return 0;
}
```

1 100

How to get 1?

t1 t2-t100

load x=0

any interleaving

add 1

write 1 to x

```c
int global_x = 0;

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

int main() {
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL, thread_func, NULL);
  pthread_create(&tid2, NULL, thread_func, NULL);

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

  printf("global_x: %d\n", global_x); // minimum? maximum?

  return 0;
}
```

minimum? 2 — maximum? 200

to get 100:

t1 loads 0.

t2 runs to completion
x = 100

runs 100 iter.
x = 100

to get 2:

t1	t2
loads 0.	
	loads 0 runs 99 iter x = 99
adds 1 to 0 writes x = 1	
	runs the 100th iteration loads 1
runs to completion x = 100	
	adds 1 to 1 writes x = 2

Thread Execution

→ reasoning about shared state is difficult w/out thread coordination/synchronization

→ synchronization primitive: tools that help us synchronize threads

Locks

→ a synchronization primitive that guarantees exclusive access to a designated section of code (critical section) (mutual exclusion)

→ APIs:

```
lock_acquire();  // acquires the lock, doesn't return until the caller becomes the lock holder

lock_release();  // release the lock
```

Locks Properties

- → Safety : only one thread in the critical section at a time
- → Progress : a thread can enter the critical section if no one else is in it
  a liveness property
- → Bounded Wait : there's an upperbound to how long a thread waits before entering the critical section
  a fairness property

☆ lock is just a tool, programmers need to use it properly for it to be effective :

```
lock_acquire();
//access shared state
lock_release();
```

Lock Implementation (First Attempt)

```
struct lk { bool locked; }
```

```
lock_acquire(struct lk * lk) {
    while (lk->locked) { ; }
    lk->locked = true;
}
```

called by multiple threads

```
lock_release(struct lk * lk) {
    lk->locked = false;
}
```

* violates safety!

Lock Implementation (Another Attempt)

→ requires hw support for atomic read & modify

→ test & set instr.

atomic! [→ sets val to 1 if the current val is 0, returns old val (0)
→ otherwise, does nothing, returns value read (1)

→ lock_acquire(struct lk* lk) {
```
    while (test&set(&lk->locked)) { }
        // locked is already set to true by this thread
}
```
→ expensive instr

Compare & swap (another atomic instr.)

```
while(1){
    while (lk->locked) { }
    if (!test&set(&lk->locked)){
        return;
    }
}
```

Types of Locks

→ Spinlock (spins / busy waits on CPU)

→ when the lock is not free, keep checking until it acquires the lock

→ sleeplock / mutex

→ when the lock is not free, blocks/sleeps until it's free (gives up the CPU)

When to use spinlock?

→ short critical section

→ few waiters

When to use sleeplock?

→ long critical section (I/O access)

→ long wait time (many waiters)