Thread Execution

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

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

$\text{value of } x \text{ at the end}$

$\Rightarrow 1: t_1 \text{ finishes then } t_2 \text{ runs}$

$\Rightarrow 2: t_1 \quad t_2$

$\Rightarrow \text{ time of check to time of use!}$

$\Rightarrow \text{ race condition = scheduling orders cause semantically different results}$
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;
}
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;
}
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;
}
Thread Execution

→ reasoning about shared state is difficult without 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)

→ 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:*

```c
lock_acquire();
// access shared state
lock_release();
```
Lock Implementation (First Attempt)

```c
struct 1k { bool locked; }

lock_acquire(struct 1k *1k) {
    while (!1k->locked) {}
    1k->locked = true;
}

lock_release(struct 1k *1k) {
    1k->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)) {
    if locked is already set to true by this thread
      return;
  }
  compare & swap (another atomic instr)
}

while (!lk->locked) {
  if (!test & set (&lk->locked))
    return;
}
Types of Locks

→ Spinlock
  → when the lock is not free, keep checking until it acquires the lock (spins/busy waits on CPU)

→ 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)