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, 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;
}
data race

reasoning about multithreaded code that access shared data is difficult!

time-of-check to time-of-use

might be preempted in between, data might be changed by another thread

reasoning about global-x++ is much easier if it's done atomically.

atomic instr.

test & set (loc)

if *loc == 0 & *loc == 1; return true;
else return false;

compare & swap (CAS)

args: loc, old-val, new-val

if (*loc == old-val) {
    *loc = new-val,
    return true;
} else return false;

while (sync_bool_compare_and_swap(&global_x, x, x+1) == false) {
    jmp 0x11c5 <thread_func+28>
    mov -0x4(%rbp),%eax
    add $0x1,%eax
    mov %eax,%edx
    mov -0x4(%rbp),%eax
    lock cmpxchg %edx,0x2e3c(%rip)
    # 0x4014 <global_x>
    sete %al
    xor $0x1,%eax
    test %al,%al
    jne 0x11bc <thread_func+19>
exclusive to shared data makes it much easier to reason
- atomic instr. provides this but only for 1 memory loc.
- how can we do this for arbitrary amount of data access?

what causes us to lose exclusive access? (single core)
- timer interrupt, preemption!
- disable interrupts would then provide us exclusive access

problems: user processes don't have the privilege for this,
- blocking interrupts ⇒ may lose hw events,
  is a per-core operation!
- doesn't guarantee exclusive access on multicore machines
What then? Build software abstraction for exclusive access!

**Locks**
- a synchronization primitive that provides exclusive access to a designated section of code (critical section)

**Locks API**
- `lock_acquire`: doesn’t return until it acquires the lock (grants exclusive access)
- `lock_release`: gives back exclusive access

It only works if threads call `lock_acquire` before accessing shared data

A thread holding the lock can still be preempted, but data in critical section can’t change since no other thread can acquire the lock.

(mutual exclusion)
- Use lock to protect access around shared data
  - How much data should the lock protect? (lock granularity)
  - How much data should the lock protect?
  - A single lock for all system call data?
    - Can only process one syscall at a time, even if they don't share data
  - A lock for the entire file info array or a lock for each entry?

- Easy to reason about, limits concurrent access to different entries
- Fine-grained locking

- Coarse-grained locking
- Allows for concurrent access to different entries
- Harder to reason (what if you need to access multiple entries together?)
- The ordering of acquiring locks can cause troubles
Locks Impl. / Types of locks

Spinlock
  - lock acquire: spins while the lock is busy
  - lock release: clears lock state to free

Sleeplock
  - lock acquire: sleeps while the lock is busy
  - lock release: clears lock state to free, wakes up a waiter.