```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;
}
```

t1

loads 0

adds 1 to 0

writes x = 1

t2 ~ t100 complete

0 100

1 100

```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;
}
```

to get 100:

t1 loads 0.

t2 runs to completion
x = 100

runs 100 iter.
x = 100

to get 2:

t1

loads 0

adds 1 to 0
writes x = 1

runs to completion
x = 100

t2

runs 99 iter.
x = 99

runs the 100th iteration
loads 1

adds 1 to 1
writes x = 2

2 200.

10/13/23

data race, race condition

→ 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; }
```

```
9       while (__sync_bool_compare_and_swap(&global_x, x, x+1) == false) {
   0x00000000000011ba <+17>:    jmp    0x11c5 <thread_func+28>
   0x00000000000011c5 <+28>:    mov    -0x4(%rbp),%eax
   0x00000000000011c8 <+31>:    add    $0x1,%eax
   0x00000000000011cb <+34>:    mov    %eax,%edx
   0x00000000000011cd <+36>:    mov    -0x4(%rbp),%eax
   0x00000000000011d0 <+39>:    lock cmpxchg %edx,0x2e3c(%rip)        # 0x4014 <global_x>
   0x00000000000011d8 <+47>:    sete   %al
   0x00000000000011db <+50>:    xor    $0x1,%eax
   0x00000000000011de <+53>:    test   %al,%al
   0x00000000000011e0 <+55>:    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 (mutual exclusion)

- 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

★ only works if threads calls lock_acquire before accessing shared data

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

it's a design pattern!

→ use lock to protect access around shared data

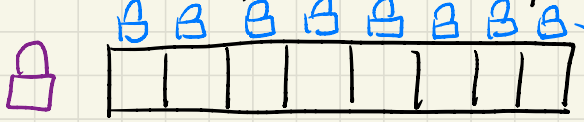
→ How much data should the lock protect? (lock granularity)

→ a single lock for all system call data?

(getpid & read)

- can only process 1 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

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)

fine-grained locking

Locks Impl. / Types of Locks

lock = 0 if free
1 if busy

→ Spinlock

while(!testandset(&lock)){;}

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

→ sleeplock

while(lock != free){ block(); }

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