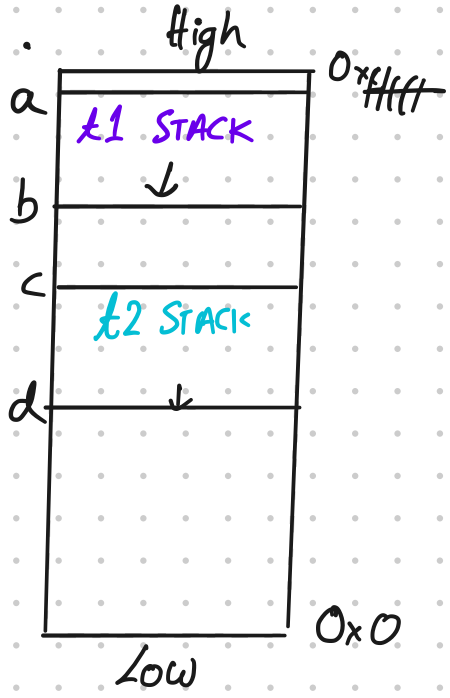
CS202 (-002) Lecture 5

- Lab2 Is Out
  - ↳ Review Session Friday
- Quiz Thursday

- Last Class
  - Shell
  - File Descriptors, Files As An Abstraction
  - Threads
    - ↳ In response to a question after class

t1	t2
RBP	RBP
RSP	RSP



(Figure 26.1 in last week's reading)

Today: Concurrency.

How do threads affect program correctness.

```
uint64_t x = 3   <-- GLOBAL
int main (...) {
    ...
    tid1 = thread_create (t1, NULL);
    tid2 = thread_create (t2, NULL);
    thread_join (tid1); thread_join (tid2);
    <-- What is the value of x?
}
```

t1 t2

```
void t1(void*){
  @ x = x * 2;
  b x = x + 7
    thread_exit();
}
```

```
void t2(void *){
  (i)  x = x + 6;
  (ii) x = x*3;
       thread_exit();
}
```

a	i	b	ii
6	12	19	57
9	18	54	61

i	a	b	ii
9	18	25	75

. . .

Observe: @ x can take many different final values

→ Contrast with

```
int main(...) {
    t1();
    t2();
}
```

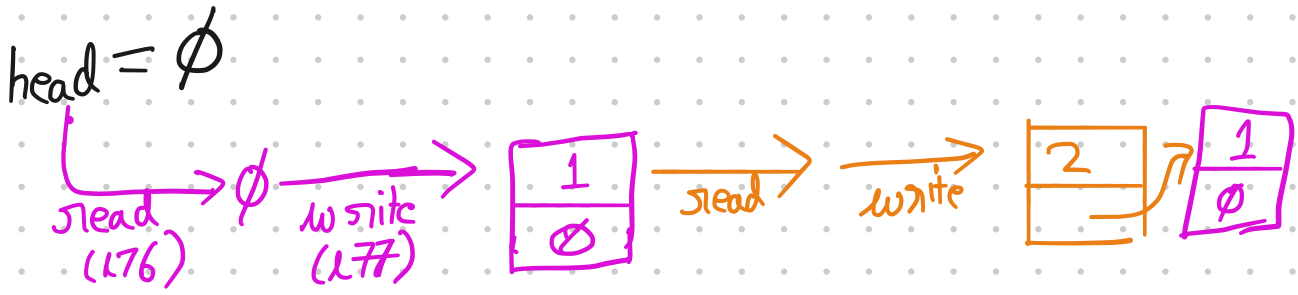
(b) Accesses from different threads can interleave.

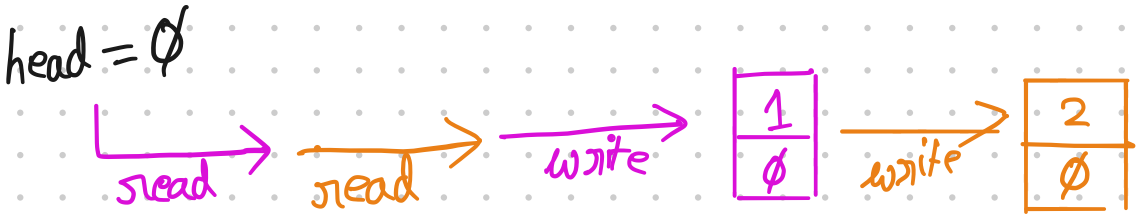
```
66  struct List_elem {
67      int data;
68      struct List_elem* next;
69  };
70
71  List_elem* head = 0;
72
73  insert(int data) {
74      List_elem* l = new List_elem;
```

head = NULL

t1	t2
insert(1);	insert(2);

```
75          l->data = data;
76          l->next = head;
77          head = l;
78      }
```





...

© Not all interleavings are safe/correct.

```
66      struct List_elem {
67          int data;
68          struct List_elem* next;
69      };
70
71      List_elem* head = 0;
72
73      insert(int data) {
74          List_elem* l = new List_elem;
75          l->data = data;
76          l->next = head;
77          head = l;
78      }
```

Don't want another thread to read or write to head b/w line 76 & 77

Should be ATOMIC

- Synchronization primitives
  - ↳ Tools to control interleaving

↳ Limit concurrent actions.

Critical Section

- A way to implement/get atomicity

Three properties

- Mutual Exclusion: At most one thread is executing logic in crit. section at a time.
- Progress
- Bounded Waiting

} See notes.

```
f() {
  print(beep)
}
```

How to implement critical sections?

- Mutex
  - mutex_init
  - mutex_lock
  - mutex_unlock

Book

pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock

```
66  struct List_elem {
67      int data;
68      struct List_elem* next;
69  };
70
71  List_elem* head = 0;
72
73  insert(int data) {
74      List_elem* l = new List_elem;
75      l->data = data;
76      l->next = head;
77      head = l;
78  }
```

mutex_t m;

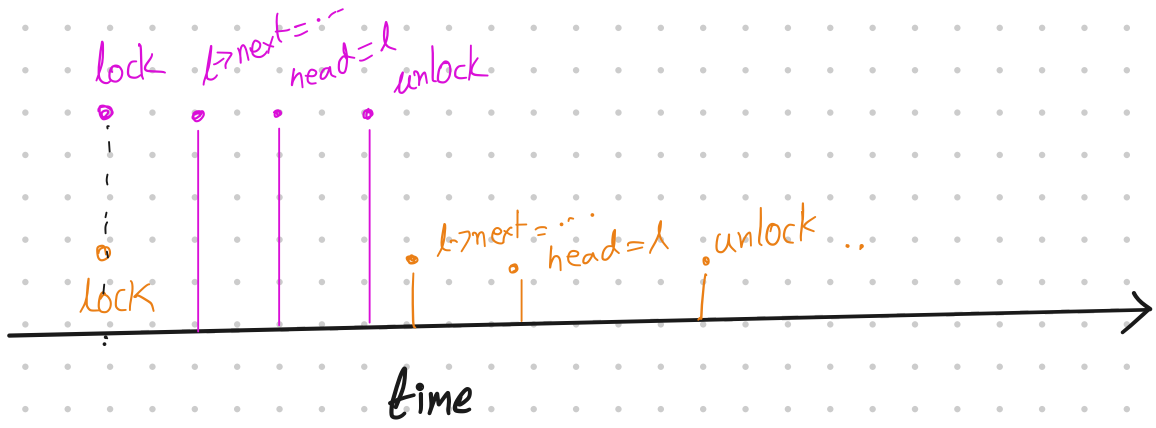
mutex_lock(&m);

mutex_unlock(&m);

Warning: Illustrative, later lecture will suggest a different style that you must follow!

t1 || t2

insert(1); || insert(2);



Semantics

- When mutex_lock returns caller holds the lock
- Locks can be granted to waiting threads in ANY order

TIME

t1: lock(l)
t1: CRIT SEC
t2: lock(l)
t3: lock(l)
t1: unlock(l)
t3: CRIT SEC

t1: lock(l)
t1: CRIT SEC
t2: lock(l)
t3: lock(l)
t1: unlock(l)
t2: CRIT SEC

NEED TO BE CAREFUL ABOUT WHAT LOCK YOU USE FOR A CRIT SECTION

```
66      struct List_elem {
```

```
68      struct List_elem* next;
69  };
70
71  List_elem* head = 0;
72
73  insert(int data) {
74      List_elem* l = new List_elem;
75      l->data = data;
76      l->next = head;
77      head = l;
78  }
```

mutex_t m;

mutex_lock(&m);

mutex_unlock(&m);

```
mutex_t m2;
int remove() {
    int ret = 0;
    mutex_lock(&m2);
    if (head != NULL) {
        ret = head->data;
        head = head->next;
    }
    mutex_unlock(&m2);
    return ret;
}
```

Is the list still correct?

insert(List_elem *hd, data)
remove(hd, data)

```
int main(...) {
    List_elem hd;
    thread_create(..&l, &hd)
```

Will reinforce this over the next several lectures, but

WHEN USING SYNCH. PRIMITIVES (locks, cond vars, ...) KEEP TRACK OF WHAT DATA THEY ARE ASSOCIATED WITH

→ variables / structs / ...

CONDITION VARIABLES

m,

x = ∅

t1

t2

- wait until
  $x \geq 15$
- PRINT PROGRESS
- $x = 0$

```
while (true) {
  Do WORK.
  x++;
}
```

ⓐ Need to protect x with a lock. Why?

ⓑ Ideally, want a way to notify t1 only when $x \geq 15$.

↳ CONDITION VARIABLES!

- cond_init(cond*, ...)
- cond_wait(mutex *m, cond* c)
- cond_signal(mutex *m, cond *c)
- cond_broadcast(mutex *m, cond *c)

t1

```
mutex_lock(&m)
- check x
relq→ cond_wait(&m, &c)
.
.
.
lock(m)
```

t2

```
:
mutex_lock(&m)
cond_bcast(&m, &c);
:
mutex_unlock(&m);
```