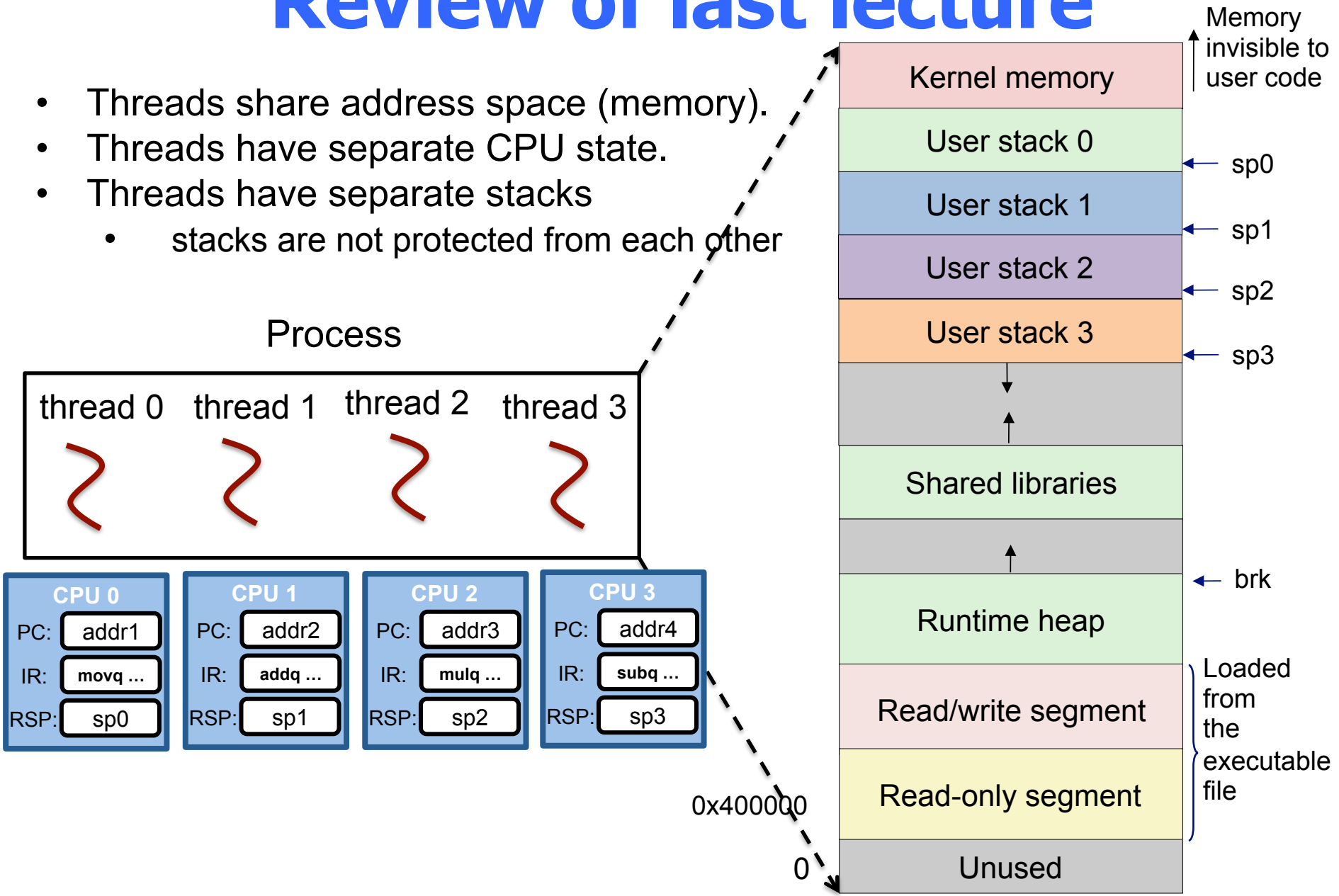
Concurrency – Locking

Jinyang Li

based on Tiger Wang's slides

Review of last lecture

- Threads share address space (memory).
- Threads have separate CPU state.
- Threads have separate stacks
  - stacks are not protected from each other



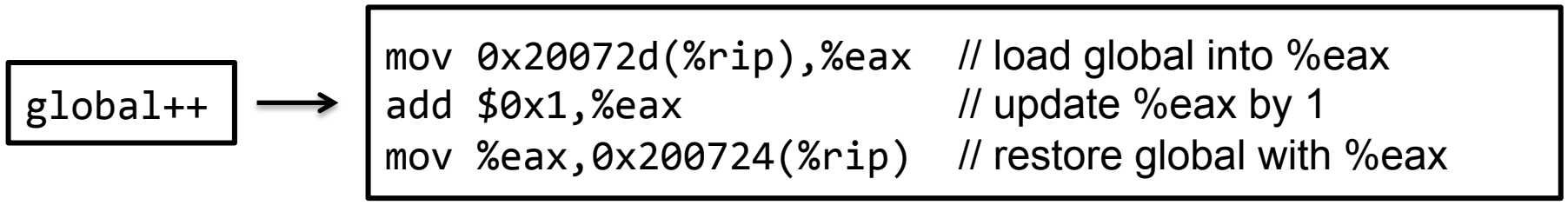
Review of last lecture

- Create threads.
  - `pthread_create(&tid, NULL, &start, &args);`
- Wait for threads to finish.
  - `pthread_join(tid, &res);`
- The interleaving of threads are non-deterministics

This lecture

- Races lead to buggy execution
- Preventing races using locks

What are races? An example



`global: 0`

Thread 1

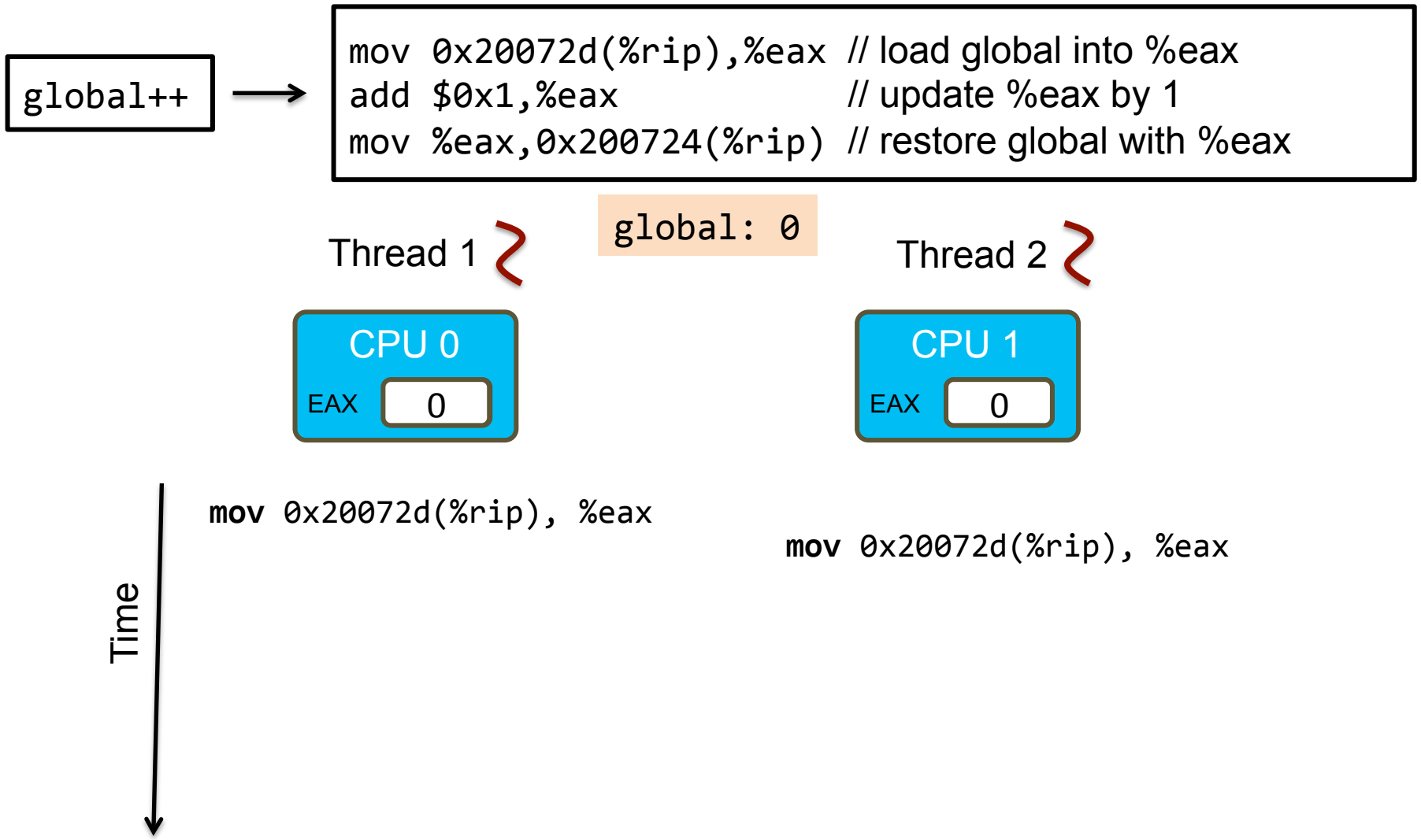
```
global++
```

Thread 2

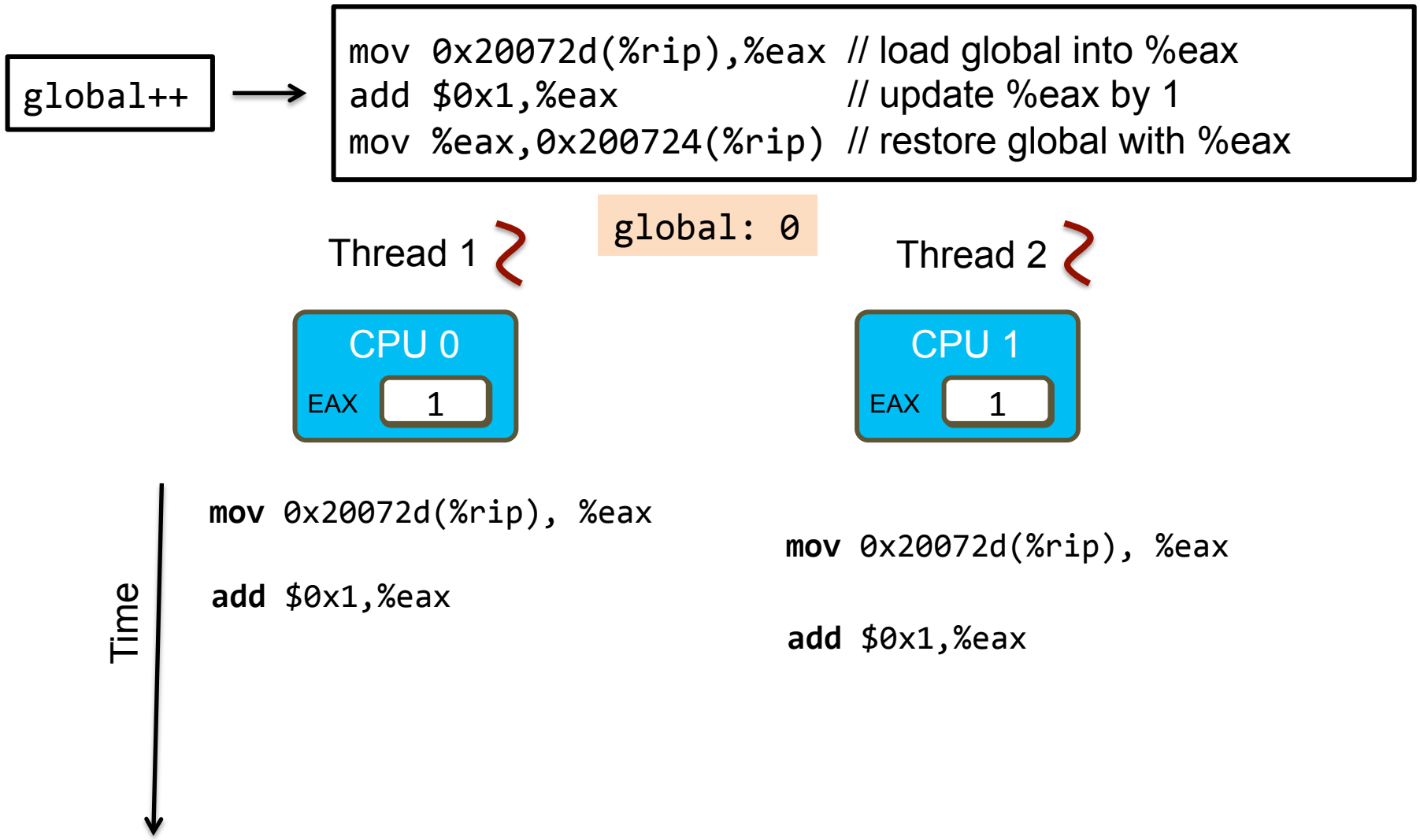
```
global++
```

What should be the value of `global` after both threads finish?

What are races? An example



What are races? An example



What are races? An example

`global++` →

```
mov 0x20072d(%rip),%eax  // load global into %eax
add $0x1,%eax            // update %eax by 1
mov %eax,0x200724(%rip)  // restore global with %eax
```

`global: 1`

Thread 1

CPU 0

EAX 1

Thread 2

CPU 1

EAX 1

Time

```
mov 0x20072d(%rip), %eax
add $0x1,%eax
mov %eax, 0x20072d(%rip)
```

```
mov 0x20072d(%rip), %eax
add $0x1,%eax
mov %eax, 0x20072d(%rip)
```

global = 1 ?!!

Worse. Sometimes global=2, sometimes global=1

How to prevent race conditions?

- Mutual exclusion: only one thread enters critical section at any time.

```
void *thread_start(void *args) {
   global++;
}

void main() {
   for (int i = 0; i < 10; i++) {
      pthread_create(.., thread_start, NULL);
   }
   pthread_join(...)
}
```

critical section

How to prevent race conditions?

- Use locks/mutexes to enforce mutual exclusion
- In pthread library, use pthread_mutex_lock/unlock

```
pthread_mutex_t mu;
void *thread_start(void *args) {
    pthread_mutex_lock(&mu);
    global++;
    pthread_mutex_unlock(&mu);

}
void main() {
    pthread_mutex_init(&mu, NULL);
    for (int i = 0; i < 10; i++) {
        pthread_create(.., thread_start, NULL);
    }
    pthread_join(...)
}
```

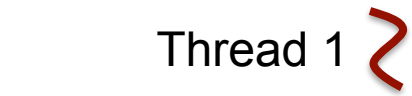
You can also malloc a mutex

Blocks caller until mutx becomes unlocked (returns 0 on success)

Unlocks mutex (returns 0 on success)

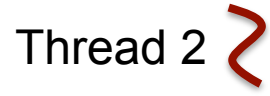
You must initialize a mutex before using it

Lock enforces mutual exclusion



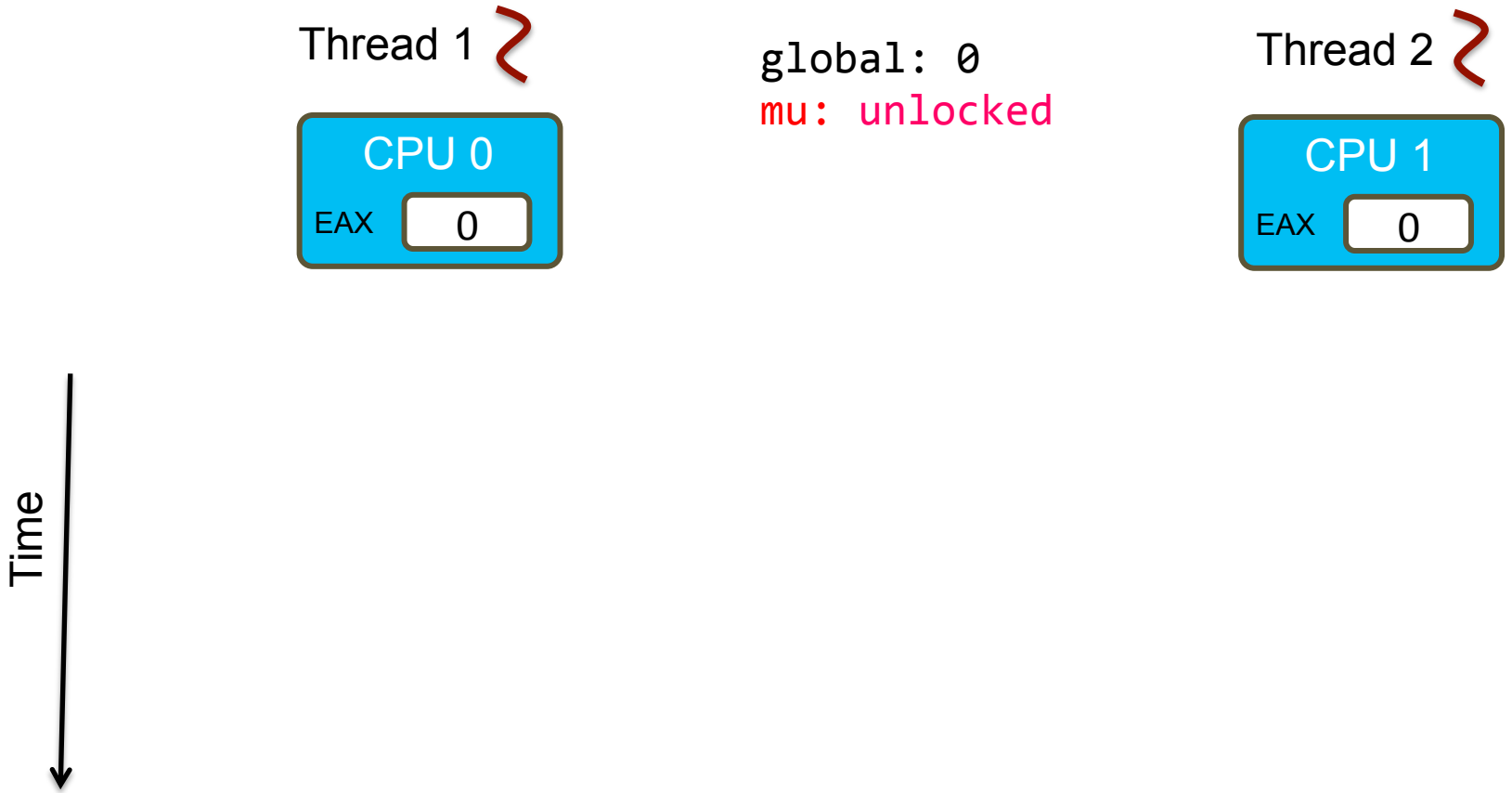

```
pthread_mutex_lock(&mu);
global++;
pthread_mutex_unlock(&mu);
```

```
int global = 0;
pthread_mutex_t mu;
```

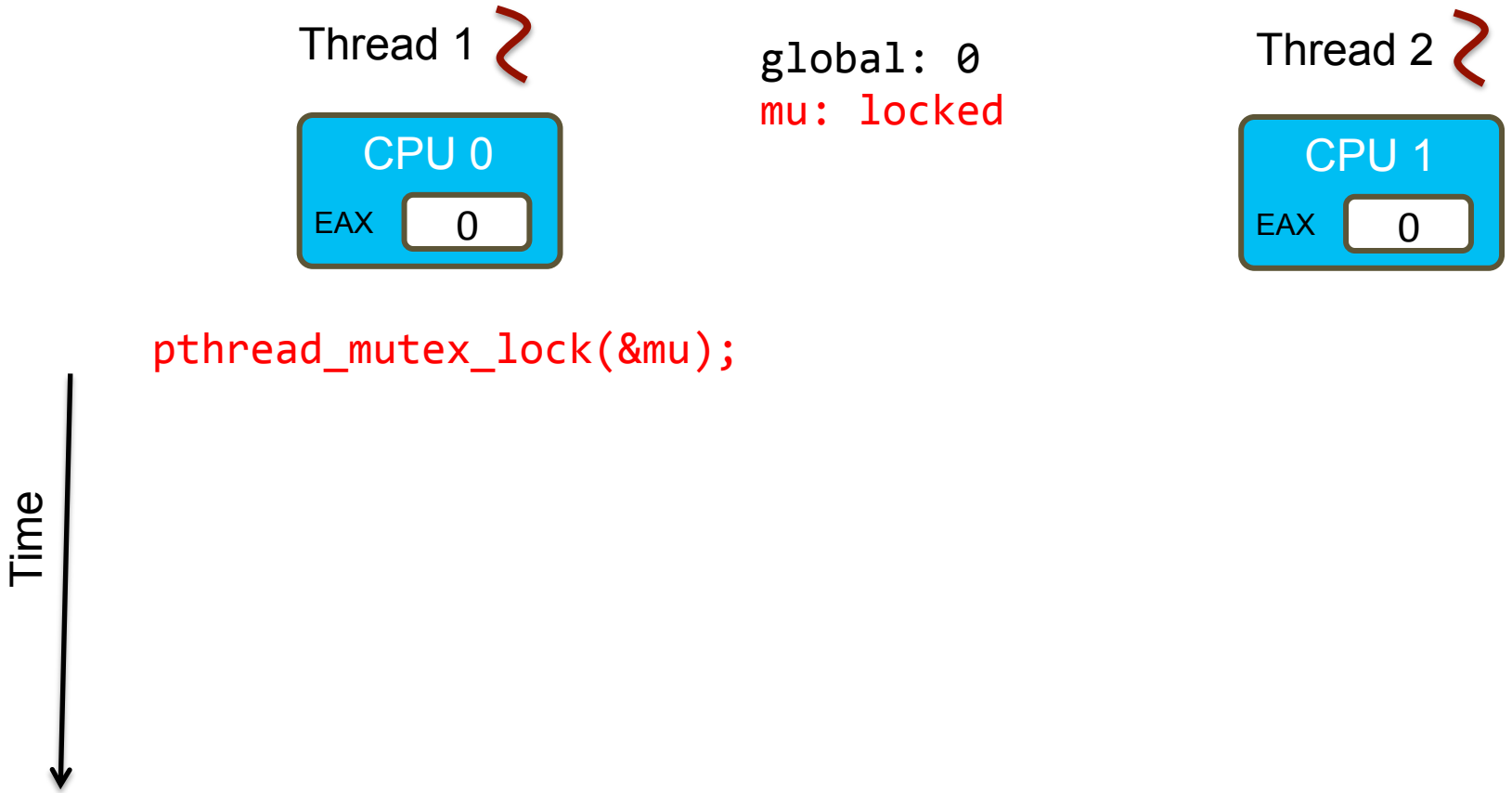



```
pthread_mutex_lock(&mu);
global++;
pthread_mutex_unlock(&mu);
```

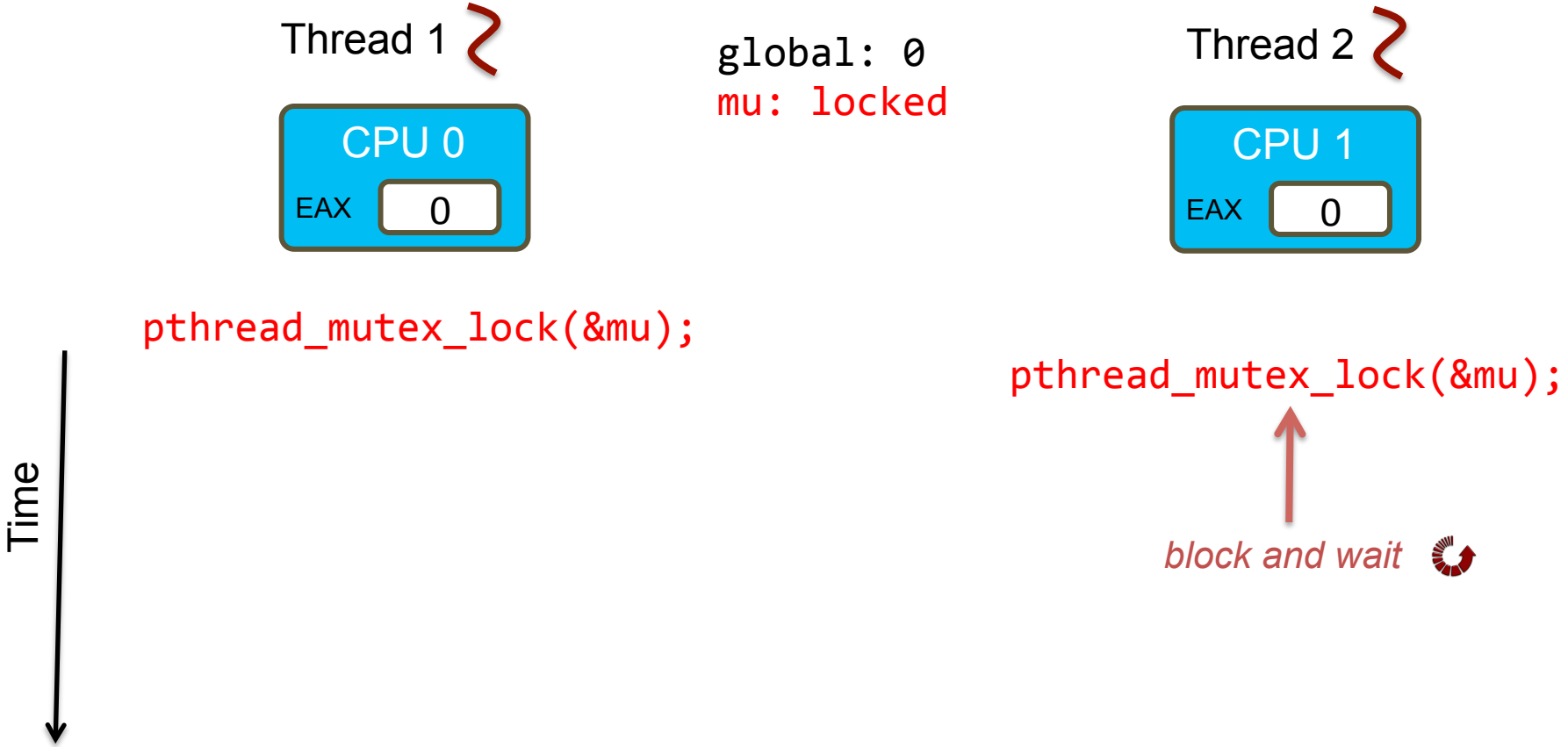
Lock enforces mutual exclusion



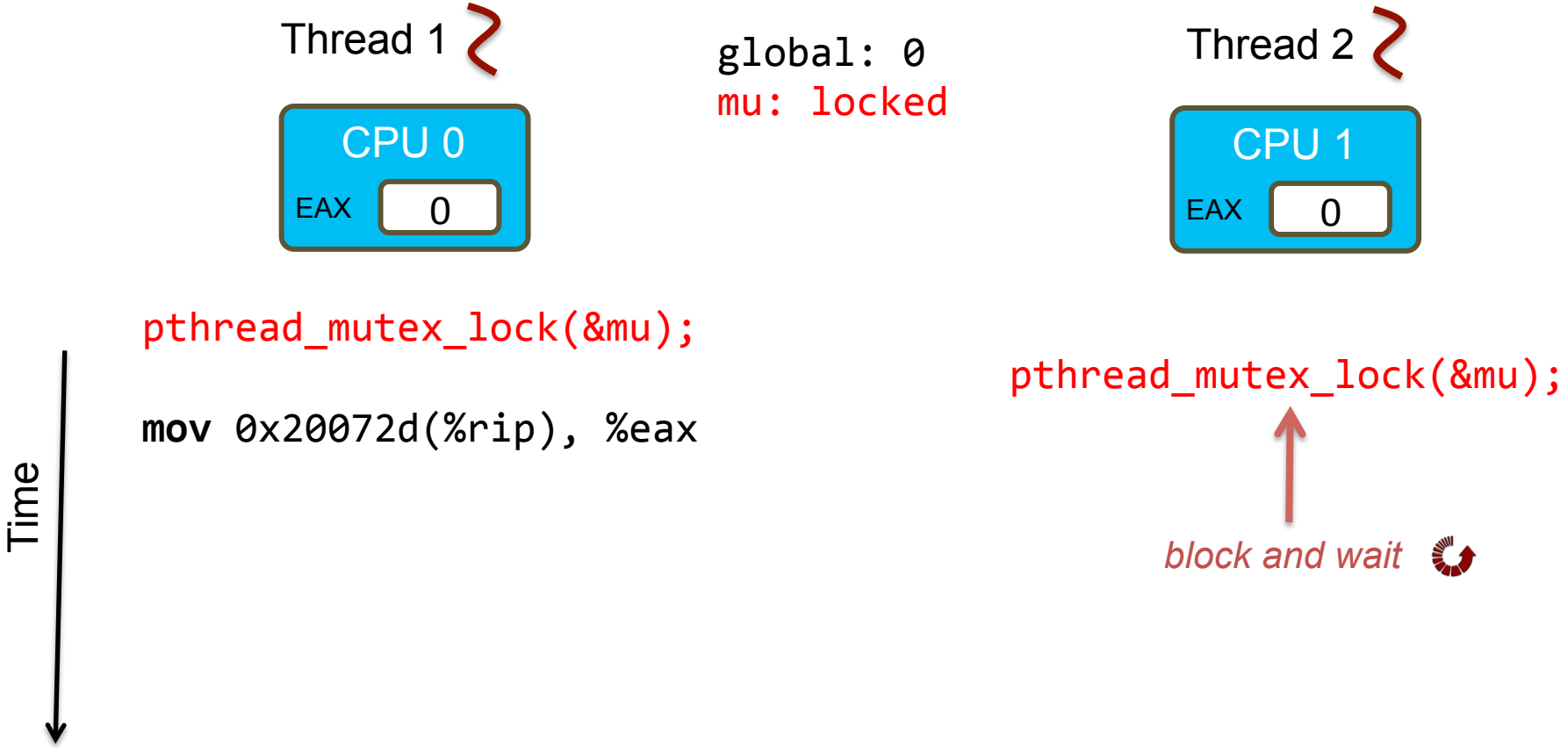
Lock enforces mutual exclusion



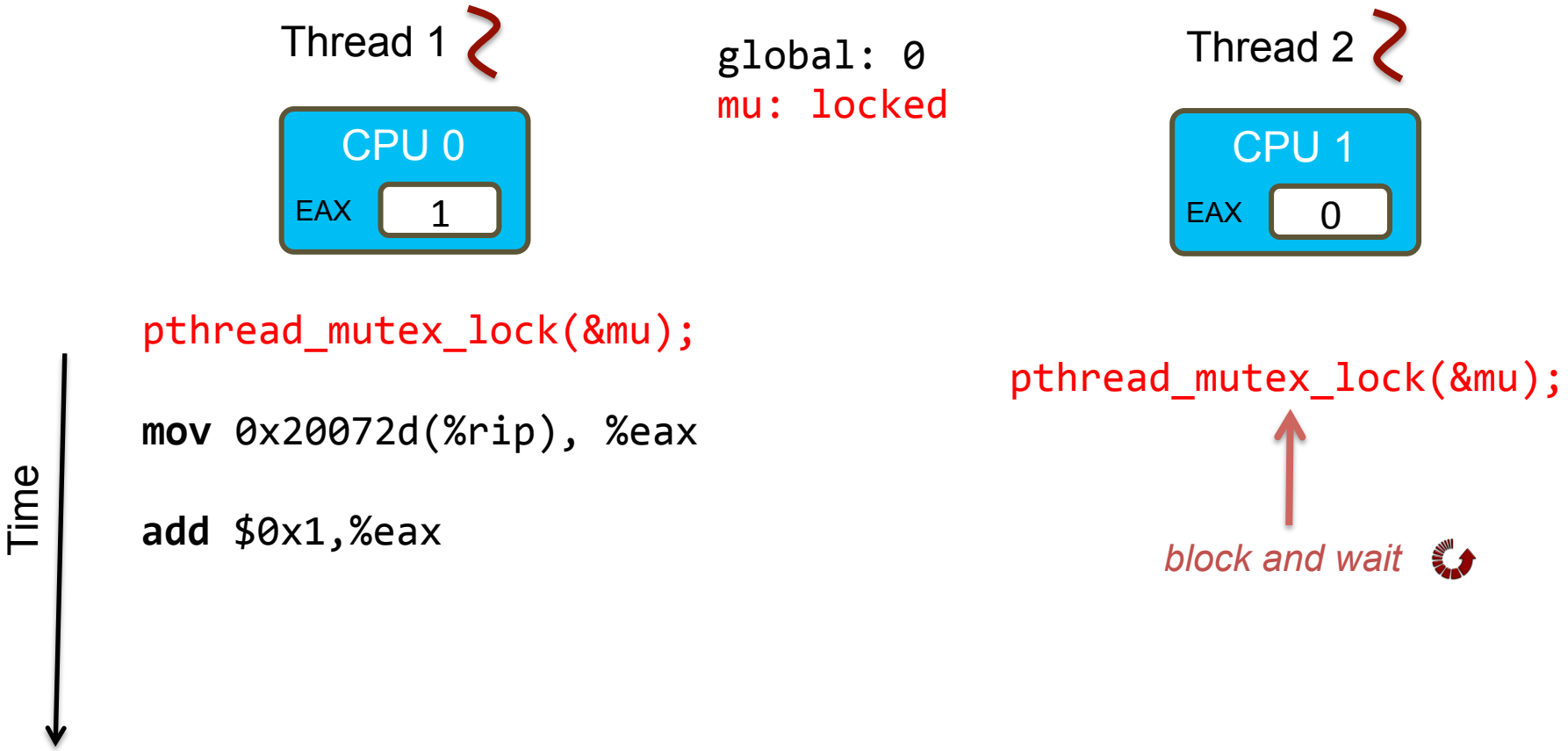
Lock enforces mutual exclusion



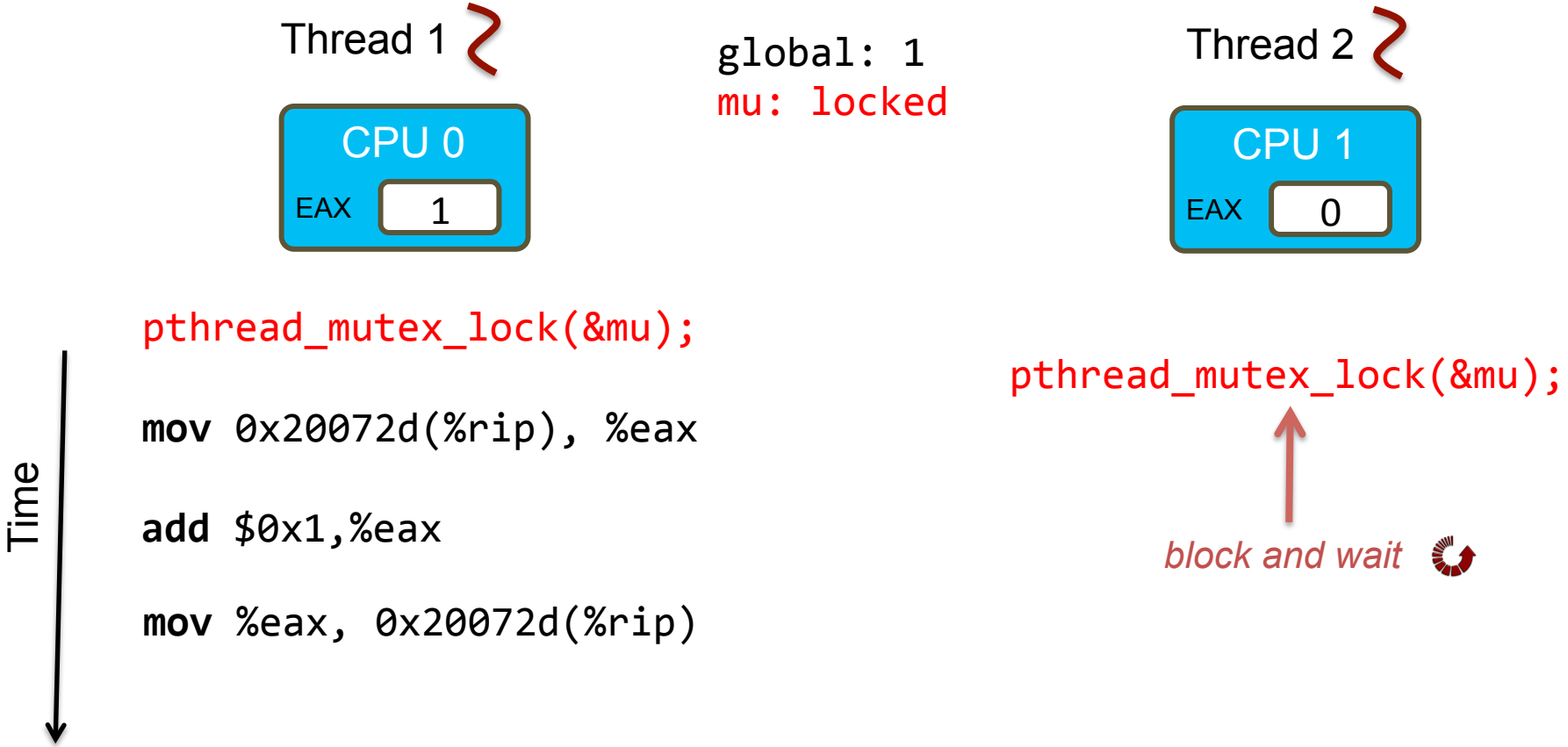
Lock enforces mutual exclusion



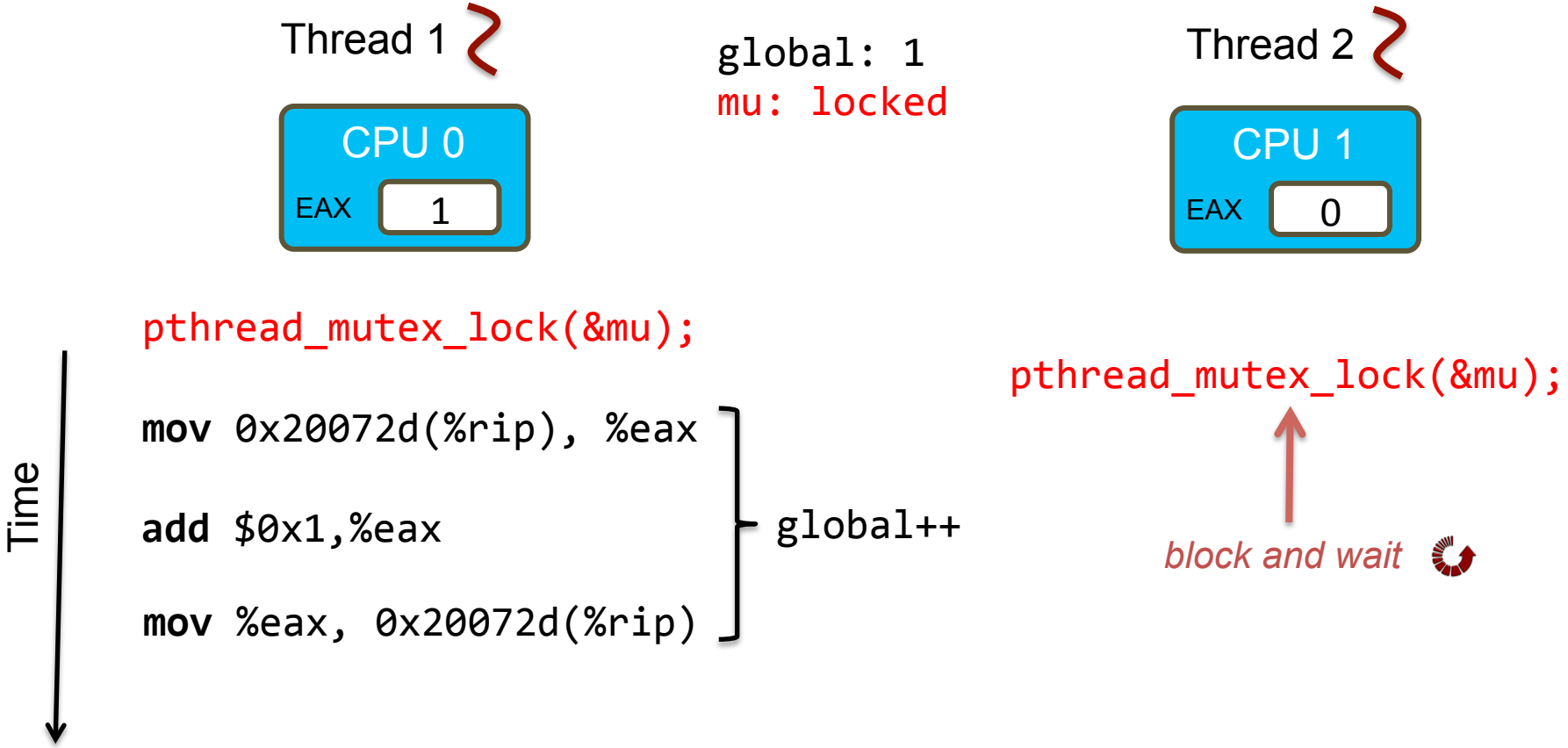
Lock enforces mutual exclusion



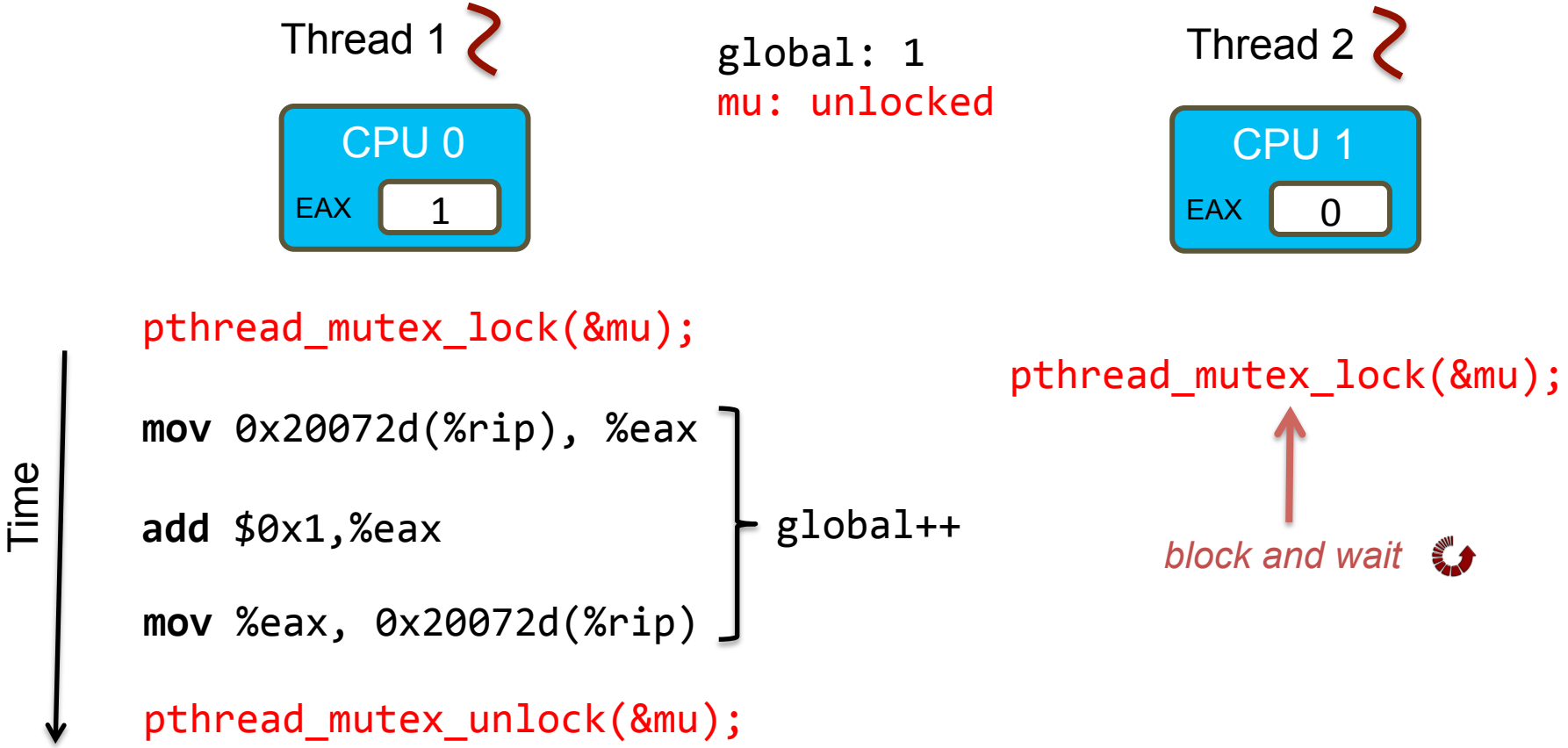
Lock enforces mutual exclusion



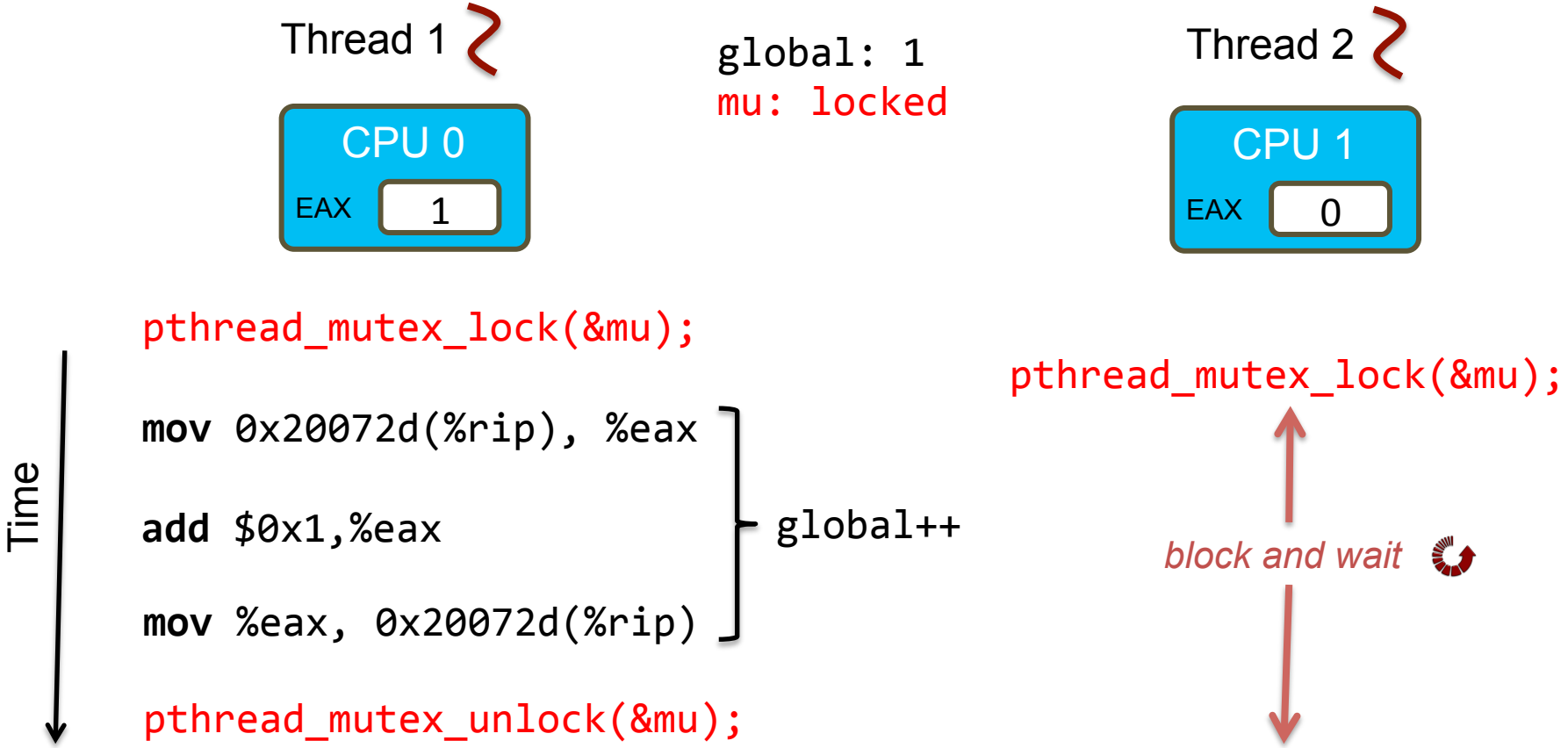
Lock enforces mutual exclusion



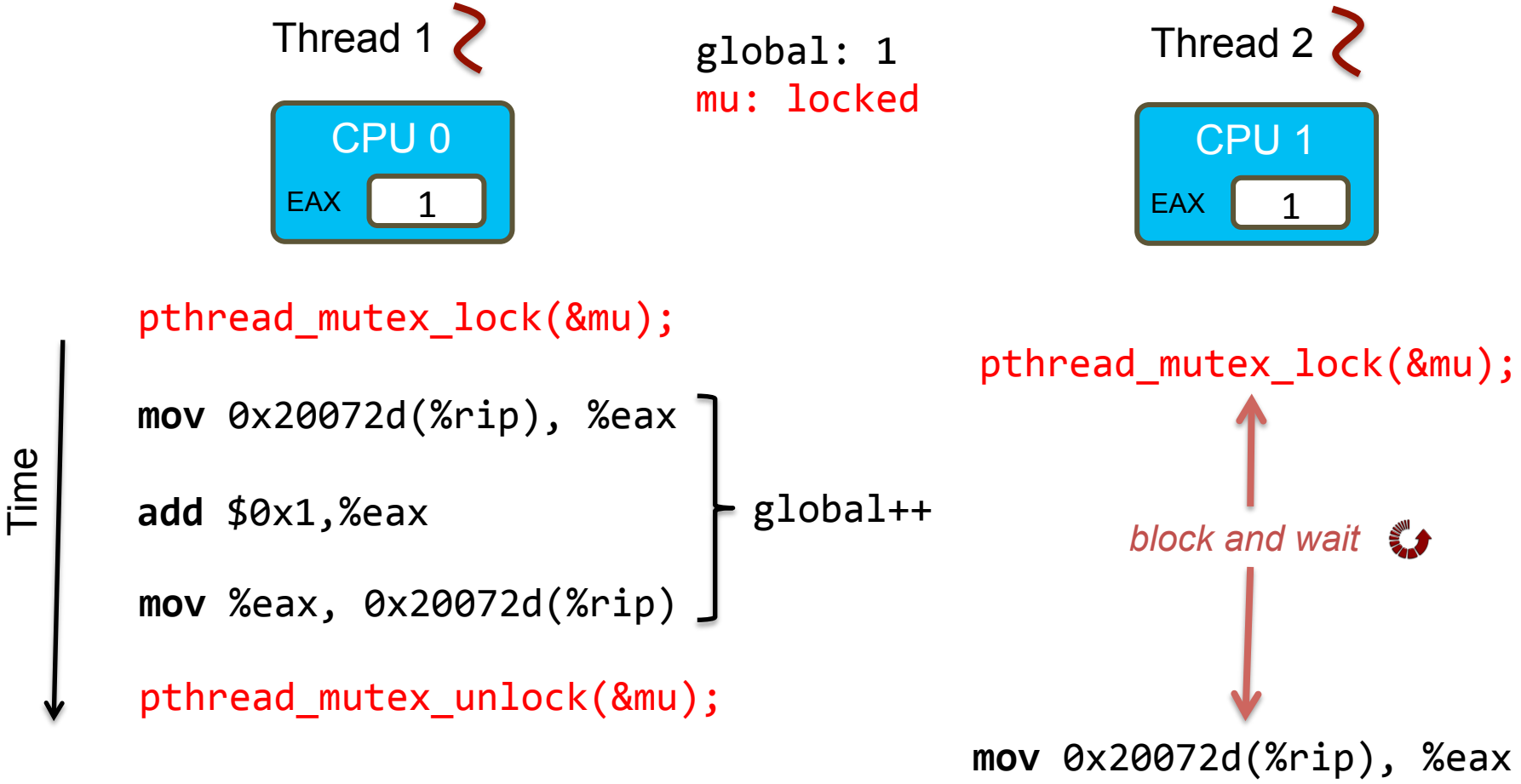
Lock enforces mutual exclusion



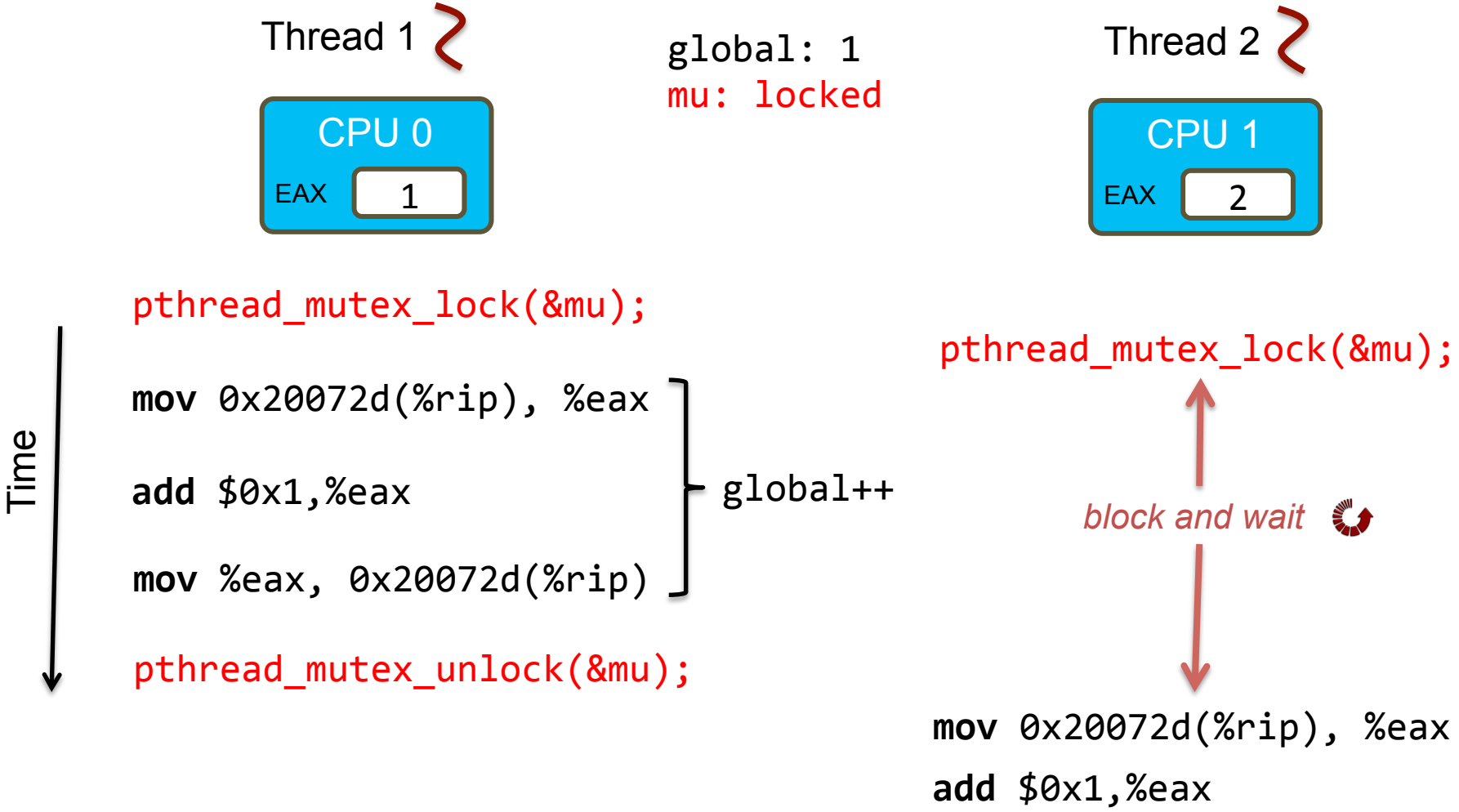
Lock enforces mutual exclusion



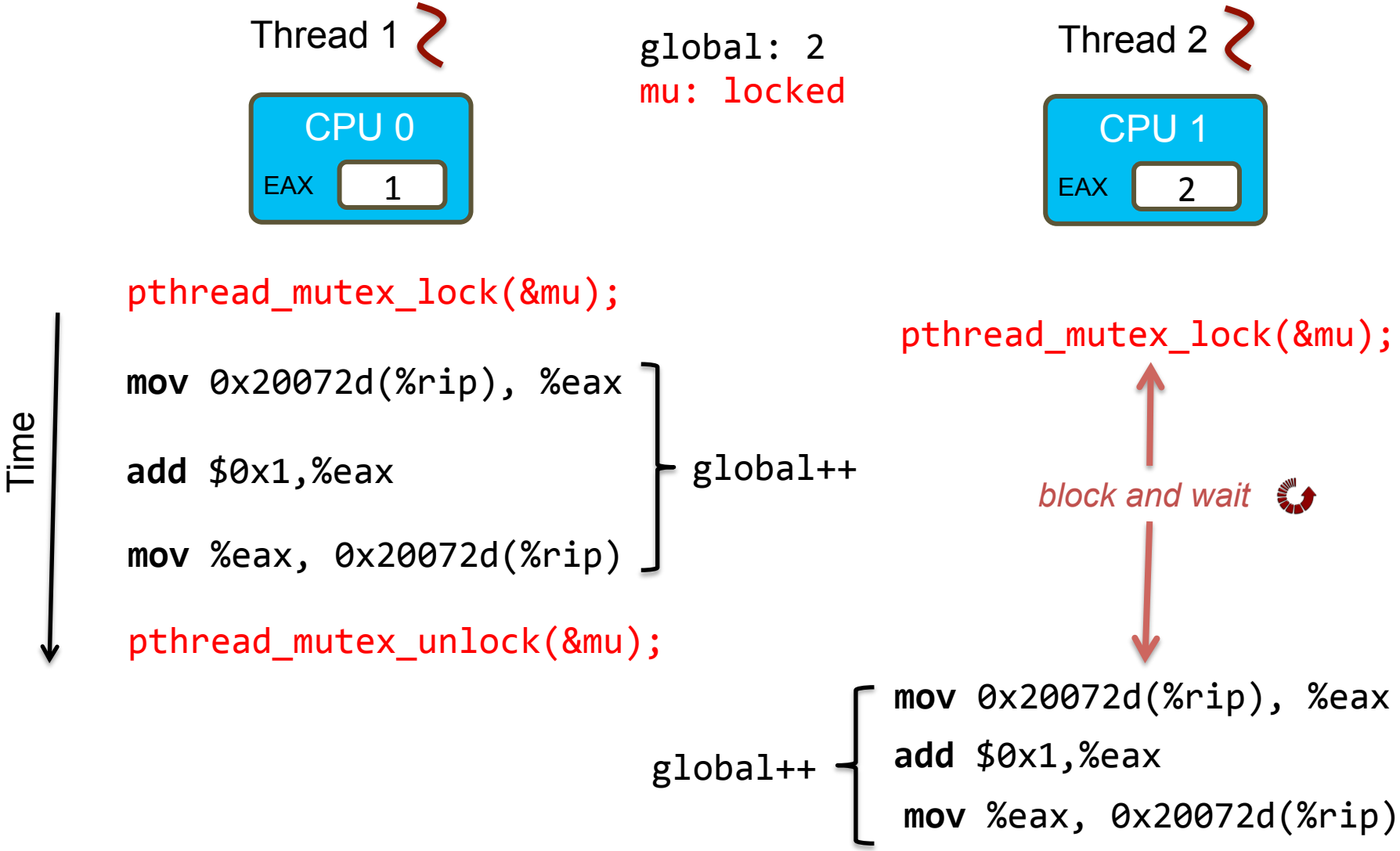
Lock enforces mutual exclusion



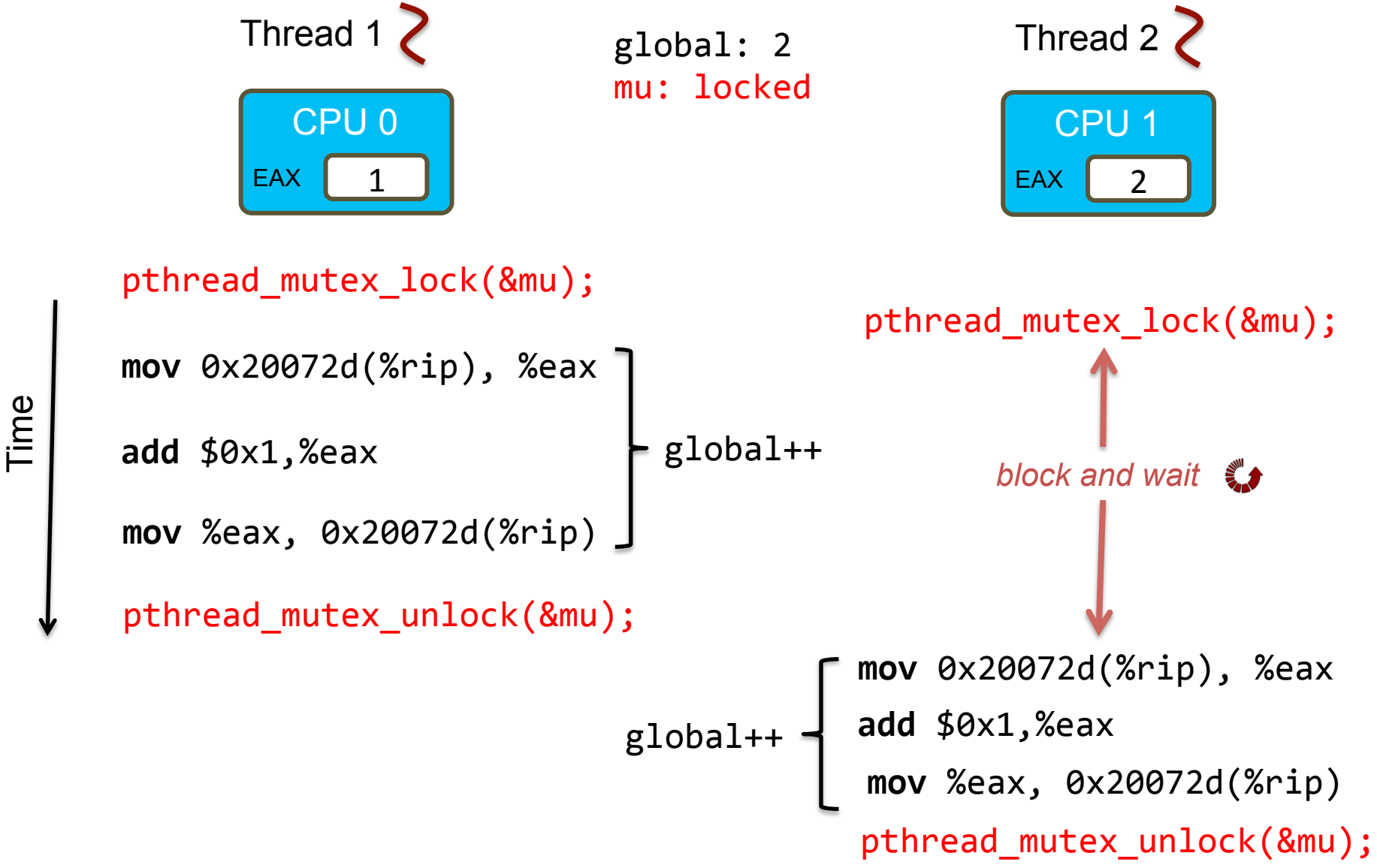
Lock enforces mutual exclusion



Lock enforces mutual exclusion



Lock enforces mutual exclusion



How to lock?

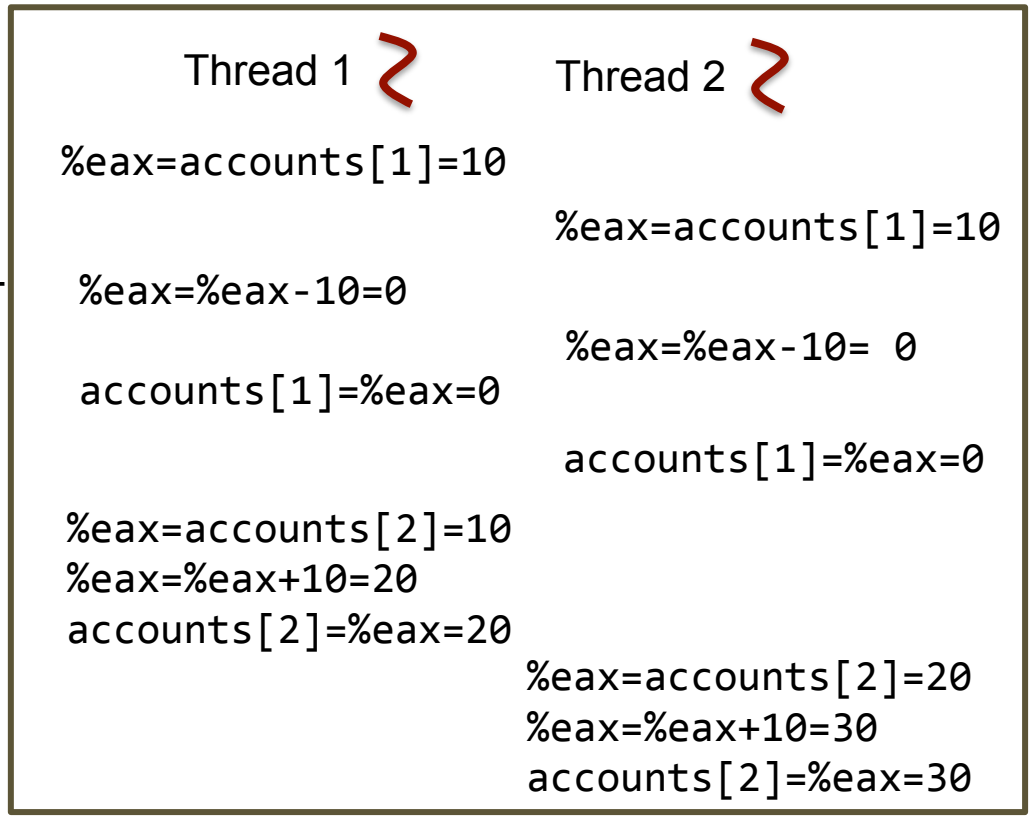
- Any vulnerable state must be locked before access
  - Vulnerable: state can be accessed by >= 2 threads, one of them is a writer.
- We mentally associate a separate lock to protect a different piece of vulnerable state

Old example: What does mu protect?

```
pthread_mutex_t mu;
int global = 0;
void *thread_start(void *args) {
    pthread_mutex_lock(&mu);
    global++;
    pthread_mutex_unlock(&mu);

}

void main() {
    for (int i = 0; i < 10; i++) {
        pthread_create(.., thread_start, NULL);
    }
    pthread_join(...)
}
```

Lock granularity: an example

```
const int n=10;
int accounts[total];

void transfer(int x, int y, int amount) {
    accounts[x] -= amount;
    accounts[y] += amount;
}

int balance(User u) {
    return accounts[u.checking]+
           accounts[u.saving];
}
```



A real hack

"The hacker discovered that if you place several withdrawals all in practically the same instant, they will get processed at more or less the same time," a user named busoni, who identified himself as the owner of the Poloniex exchange, said on the BitcoinTalk forum.

Lock granularity: an example

```
const int n=10;
int accounts[total];

void transfer(int x, int y, int amount) {
    accounts[x] -= amount;
    accounts[y] += amount;
}

int balance(User u) {
    return accounts[u.checking]+
           accounts[u.saving];
}
```

How to lock?

One big lock is simple

```
const int n=10;
int accounts[total];
pthread_mutex_t mu;

void transfer(int x, int y, int amount) {
    pthread_mutex_lock(&mu);
    accounts[x] -= amount;
    accounts[y] += amount;
    pthread_mutex_unlock(&mu);
}

int balance(User u) {
    pthread_mutex_lock(&mu);
    int bal = accounts[u.checking]+
              accounts[u.saving];
    pthread_mutex_unlock(&mu);
    return bal
}
```

What does mu protect?

What's the downside?

- Serializes execution.
- Only one thread can execute transfer() or balance() at a time

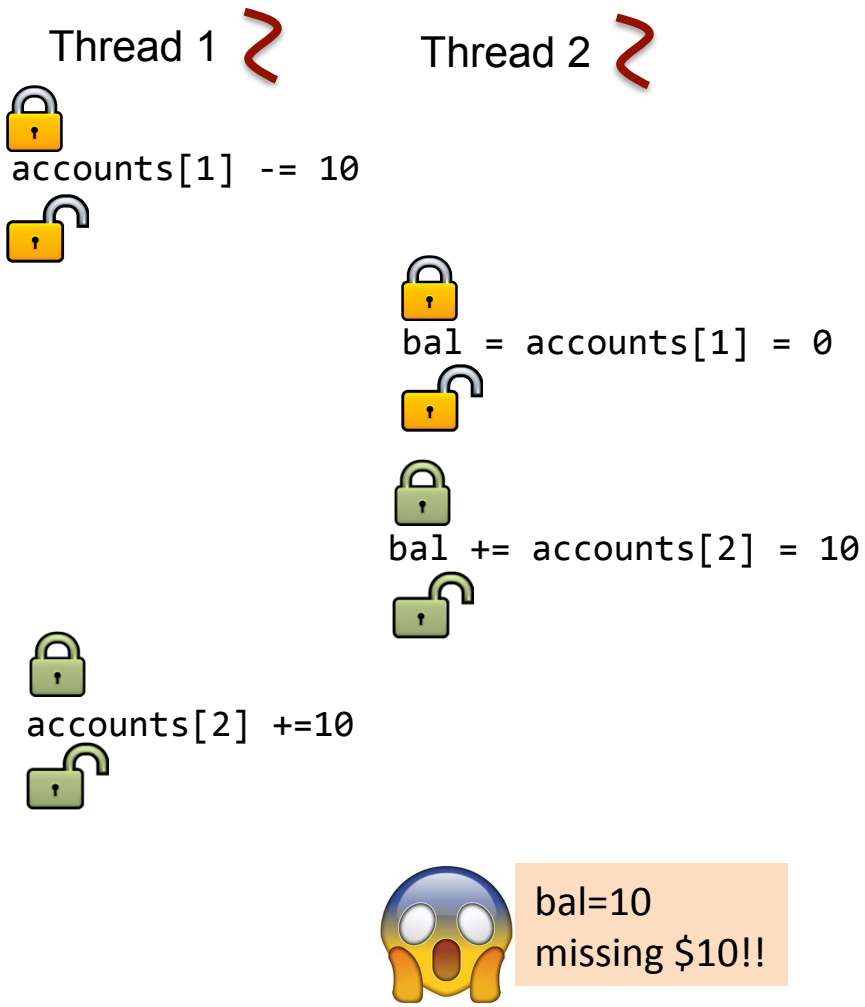
Fine-grained locks have better performance, but…

```
const int n=10;
int accounts[total];
pthread_mutex_t mu[total];

void transfer(int x, int y, int amount) {
    pthread_mutex_lock(&mu[x]);
    accounts[x] -= amount;
    pthread_mutex_unlock(&mu[x]);
    pthread_mutex_lock(&mu[y]);
    accounts[y] += amount;
    pthread_mutex_unlock(&mu[y]);
}

int balance(User u) {
    pthread_mutex_lock(&mu[u.checking]);
    int bal = accounts[u.checking];
    pthread_mutex_unlock(&mu[u.checking]);
    pthread_mutex_lock(&mu[u.saving]);
    bal += accounts[u.saving];
    pthread_mutex_unlock(&mu[u.saving]);
    return bal;
}
```

Easy to lock incorrectly

Fine-grained locks have better performance, but…

```
const int n=10;
int accounts[total];
pthread_mutex_t mu[total];

void transfer(int x, int y, int amount) {
    pthread_mutex_lock(&mu[x]);
     accounts[x] -= amount;
    pthread_mutex_unlock(&mu[x]);
    pthread_mutex_lock(&mu[y]);
     accounts[y] += amount;
    pthread_mutex_unlock(&mu[y]);
}

int balance(User u) {
    pthread_mutex_lock(&mu[u.checking]);
     int bal = accounts[u.checking];
    pthread_mutex_unlock(&mu[u.checking]);
    pthread_mutex_lock(&mu[u.saving]);
     bal += accounts[u.saving];
    pthread_mutex_unlock(&mu[u.saving]);
    return bal;
}
```

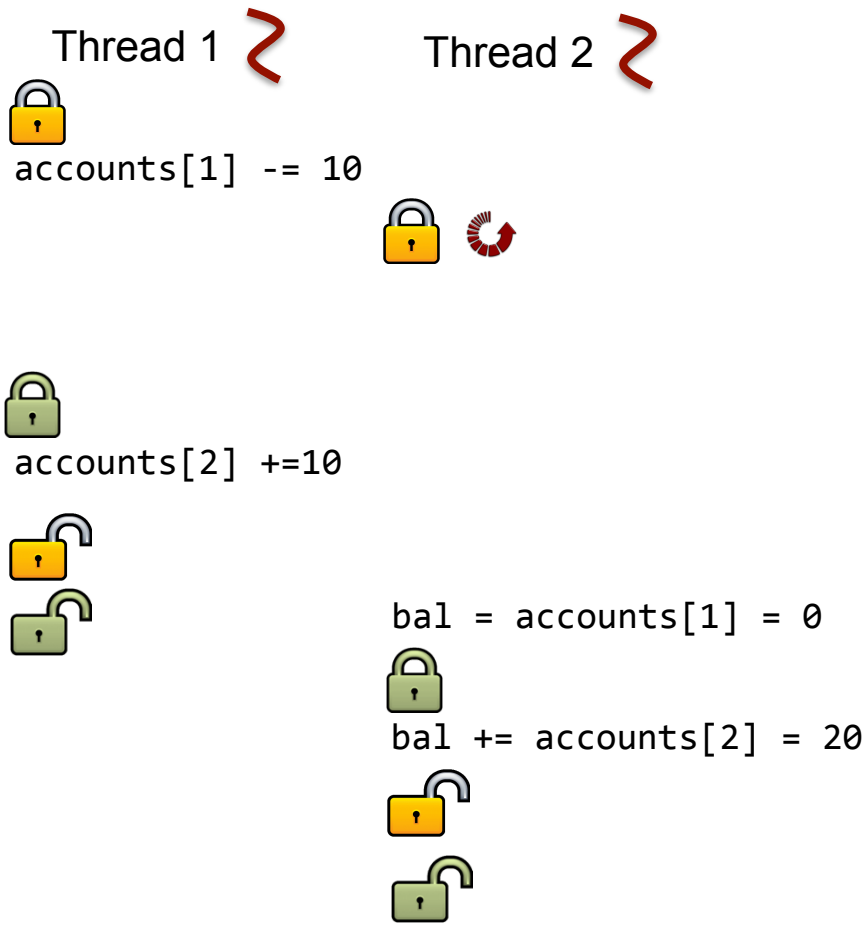


Fixing Fine-grained locks

```
int n=10;
int accounts[total];
pthread_mutex_t mu[total];

void transfer(int x, int y, int amount) {
    pthread_mutex_lock(&mu[x]);
    accounts[x] -= amount;
    pthread_mutex_lock(&mu[y]);
    accounts[y] += amount;
    pthread_mutex_unlock(&mu[x]);
    pthread_mutex_unlock(&mu[y]);
}

int balance(User u) {
    pthread_mutex_lock(&mu[u.checking]);
    int bal = accounts[u.checking];
    pthread_mutex_lock(&mu[u.saving]);
    bal += accounts[u.saving];
    pthread_mutex_unlock(&mu[u.checking]);
    pthread_mutex_unlock(&mu[u.saving]);
    return bal;
}
```

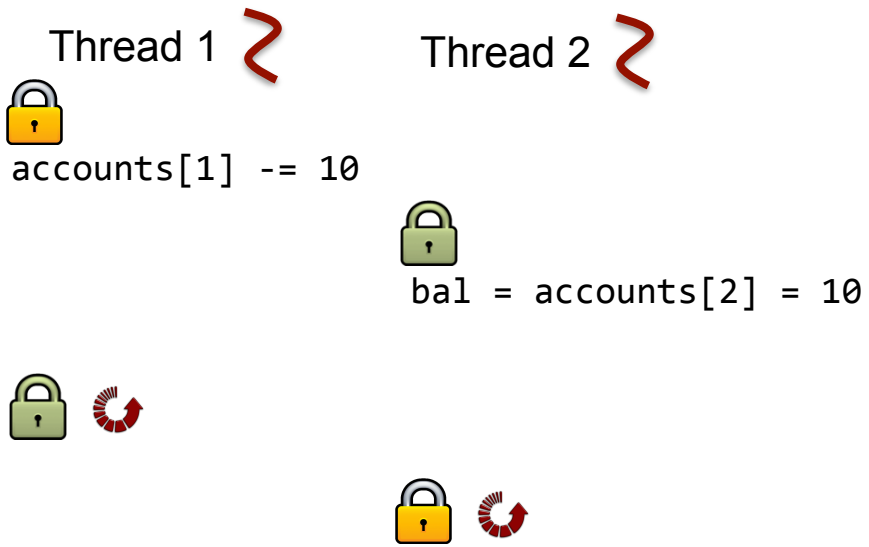
Fixing Fine-grained locks

```
const int n=10;
int accounts[total];
pthread_mutex_t mu[total];

void transfer(int x, int y, int amount) {
    pthread_mutex_lock(&mu[x]);
    accounts[x] -= amount;
    pthread_mutex_lock(&mu[y]);
    accounts[y] += amount;
    pthread_mutex_unlock(&mu[x]);
    pthread_mutex_unlock(&mu[y]);
}

int balance(User u) {
    pthread_mutex_lock(&mu[u.checking]);
    int bal = accounts[u.checking];
    pthread_mutex_lock(&mu[u.saving]);
    bal += accounts[u.saving];
    pthread_mutex_unlock(&mu[u.checking]);
    pthread_mutex_unlock(&mu[u.saving]);
    return bal;
}
```



Fine-grained locks may have deadlocks

```
const int n=10;
int accounts[total];
pthread_mutex_t mu[total];

void transfer(int x, int y, int amount) {
    pthread_mutex_lock(&mu[x]);
    accounts[x] -= amount;
    pthread_mutex_lock(&mu[y]);
    accounts[y] += amount;
    pthread_mutex_unlock(&mu[x]);
    pthread_mutex_unlock(&mu[y]);
}

int balance(User u) {
    pthread_mutex_lock(&mu[u.checking]);
    int bal = accounts[u.checking];
    pthread_mutex_lock(&mu[u.saving]);
    bal += accounts[u.saving];
    pthread_mutex_unlock(&mu[u.checking]);
    pthread_mutex_unlock(&mu[u.saving]);
    return bal;
}
```



Deadlocks sound scary, but they are not

- Stack traces indicate why deadlocks happen
  - no data corruption

```
(gdb) thread apply all bt

Thread 3 (Thread 0x7fd57746e700 (LWP 9450)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007fd57806b023 in __GI___pthread_mutex_lock (mutex=0x5627aad9f0a8 <mu+40>) at ../nptl/pthread_mutex_lock.c:78
#2  0x00005627aab9dace in balance (u=...) at thread.c:38
#3  0x00005627aab9db5f in th_balance (args=0x0) at thread.c:53
#4  0x00007fd5780686db in start_thread (arg=0x7fd57746e700) at pthread_create.c:463
#5  0x00007fd577d9188f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95

Thread 2 (Thread 0x7fd577c6f700 (LWP 9449)):
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
#1  0x00007fd57806b023 in __GI___pthread_mutex_lock (mutex=0x5627aad9f0d0 <mu+80>) at ../nptl/pthread_mutex_lock.c:78
#2  0x00005627aab9d9aa in transfer (x=1, y=2, amt=10) at thread.c:20
#3  0x00005627aab9da4d in th_transfer (args=0x0) at thread.c:29
#4  0x00007fd5780686db in start_thread (arg=0x7fd577c6f700) at pthread_create.c:463
#5  0x00007fd577d9188f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95

Thread 1 (Thread 0x7fd57847e740 (LWP 9448)):
#0  0x00007fd578069d2d in __GI___pthread_timedjoin_ex (threadid=140554814289664, thread_return=0x0, abstime=0x0, block=<optimized ou
#1  0x00005627aab9dc7b in main (argc=1, argv=0x7ffdda8e2f38) at thread.c:67
(gdb)
```

Deadlocks sound scary, but they are not

- Stack traces indicate why deadlocks happen
  - no data corruption

```
(gdb) thread 3
[Switching to thread 3 (Thread 0x7fd57746e700 (LWP 9450))]
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
135     in ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S
(gdb) up
#1  0x00007fd57806b023 in __GI___pthread_mutex_lock (mutex=0x5627aad9f0a8 <mu+40>) at ../nptl/pthread_mutex_lock.c:78
78      ../nptl/pthread_mutex_lock.c: No such file or directory.
(gdb)
#2  0x00005627aab9dace in balance (u=...) at thread.c:38
38              pthread_mutex_lock(&mu[u.saving]);
(gdb) p u
$4 = {checking = 2, saving = 1}
(gdb) l
33      int
34      balance(User u)
35      {
36              pthread_mutex_lock(&mu[u.checking]);
37              int bal = accounts[u.checking];
38              pthread_mutex_lock(&mu[u.saving]);
39              bal += accounts[u.saving];
40              pthread_mutex_unlock(&mu[u.checking]);
41              pthread_mutex_unlock(&mu[u.saving]);
42              return bal;
(gdb)
```

Deadlocks sound scary, but they are not

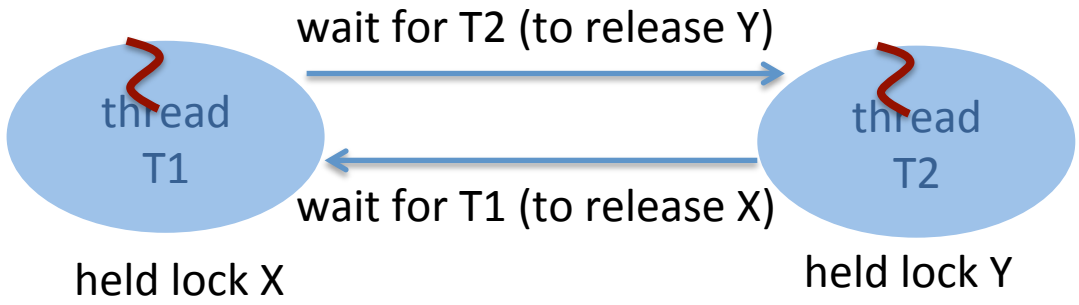
- Stack traces indicate why deadlocks happen
  - no data corruption

```
(gdb) thread 2
[Switching to thread 2 (Thread 0x7fd577c6f700 (LWP 9449))]
#0  __lll_lock_wait () at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:135
135     ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S: No such file or directory.
(gdb) up
#1  0x00007fd57806b023 in __GI___pthread_mutex_lock (mutex=0x5627aad9f0d0 <mu+80>) at ../nptl/pthread_mutex_lock.c:78
78      ../nptl/pthread_mutex_lock.c: No such file or directory.
(gdb)
#2  0x00005627aab9d9aa in transfer (x=1, y=2, amt=10) at thread.c:20
20              pthread_mutex_lock(&mu[y]);
(gdb) l
15      void
16      transfer(int x, int y, int amt)
17      {
18              pthread_mutex_lock(&mu[x]);
19              accounts[x] -= amt;
20              pthread_mutex_lock(&mu[y]);
21              accounts[y] += amt;
22              pthread_mutex_unlock(&mu[x]);
23              pthread_mutex_unlock(&mu[y]);
24      }
(gdb)
```

Use lock ordering to prevent deadlock

Observation:

– Deadlock occurs only if concurrent threads try to acquire locks in different order



Technique:

– Each thread acquires lock in the same order

Fine-grained locking with no deadlocks

```
const int n=10;
int accounts[total];
pthread_mutex_t mu[total];

void lock_in_order(int x, int y) {
    if (x > y)
       swap(&x, &y);
    pthread_mutex_lock(&mu[x]);
    pthread_mutex_lock(&mu[y]);
}

void transfer(int x, int y, int amount) {
    lock_in_order(x,y);
     accounts[x] -= amount;
     accounts[y] += amount;
    pthread_mutex_unlock(&mu[x]);
    pthread_mutex_unlock(&mu[y]);
}

int balance(User u) {
    lock_in_order(u.checking, u.saving);
     int bal = accounts[u.checking] += accounts[u.saving];
    pthread_mutex_unlock(&mu[u.checking]);
    pthread_mutex_unlock(&mu[u.saving]);
    return bal;
}
```