Concurrency – Locking

Zhaoguo Wang

Example 1

```
global++  →
```

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

Example 1

```
global++
```

$\rightarrow$

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

Thread 1 ≀

```
global++
```

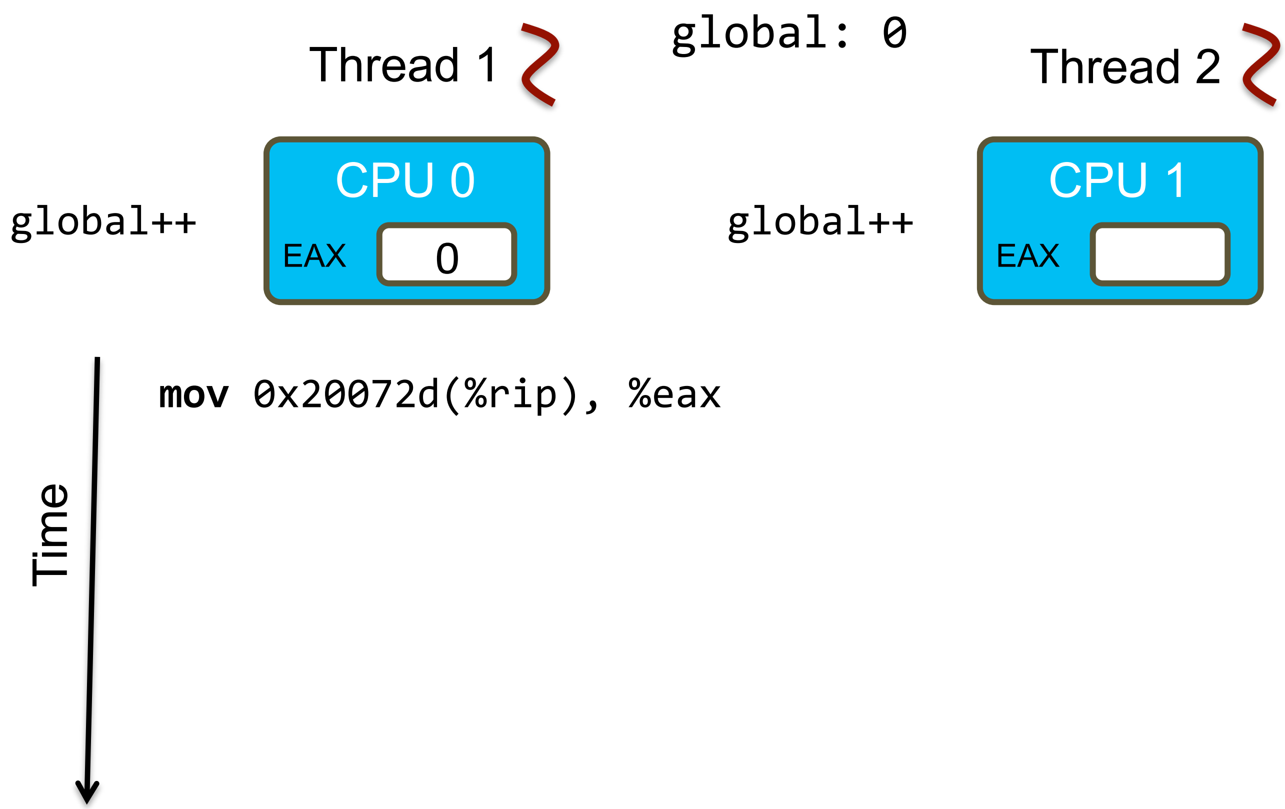
Thread 2 ≀

```
global++
```

Example 1

```
global++  →
```

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

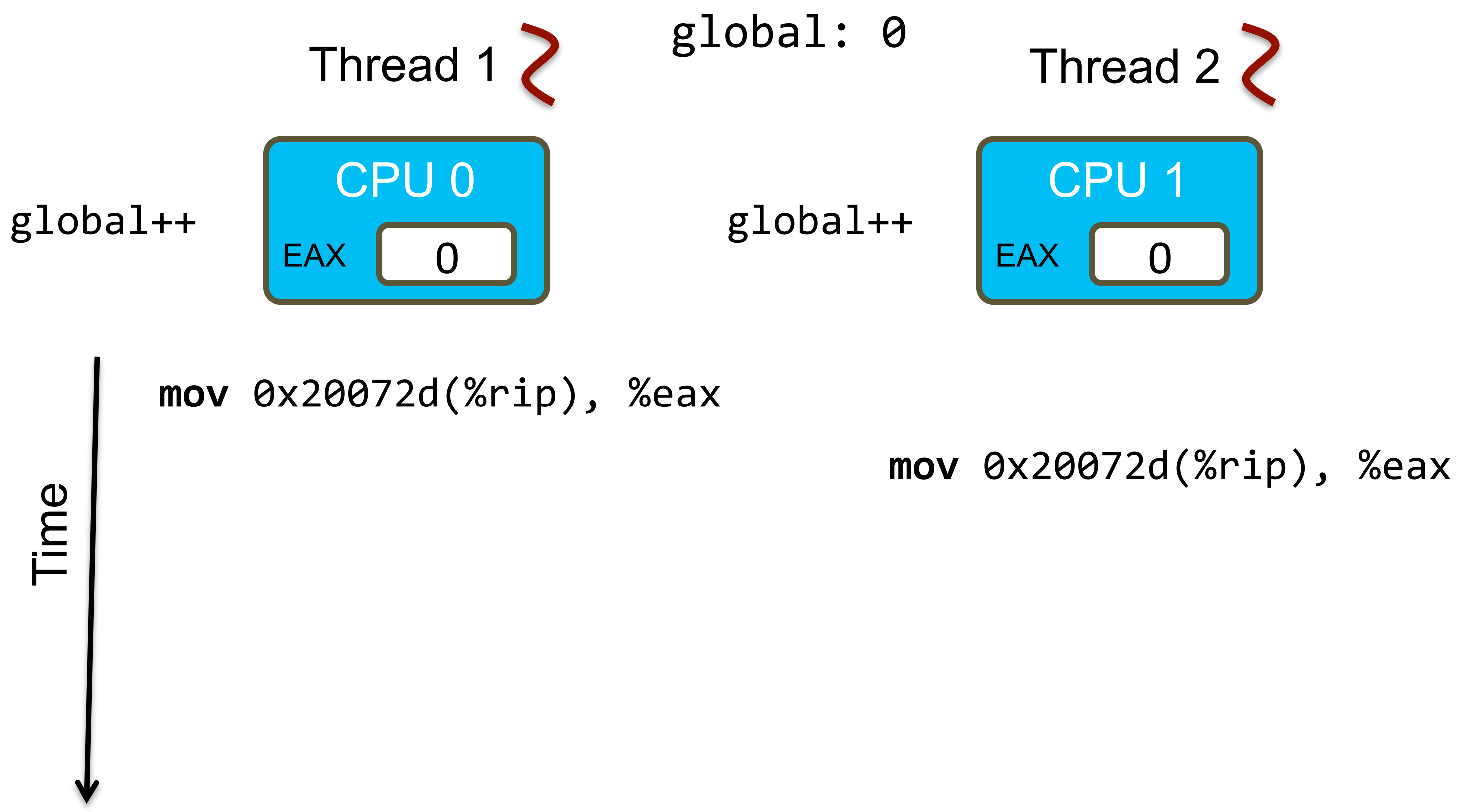


```
mov 0x20072d(%rip), %eax
```

Example 1

```
global++  →
```

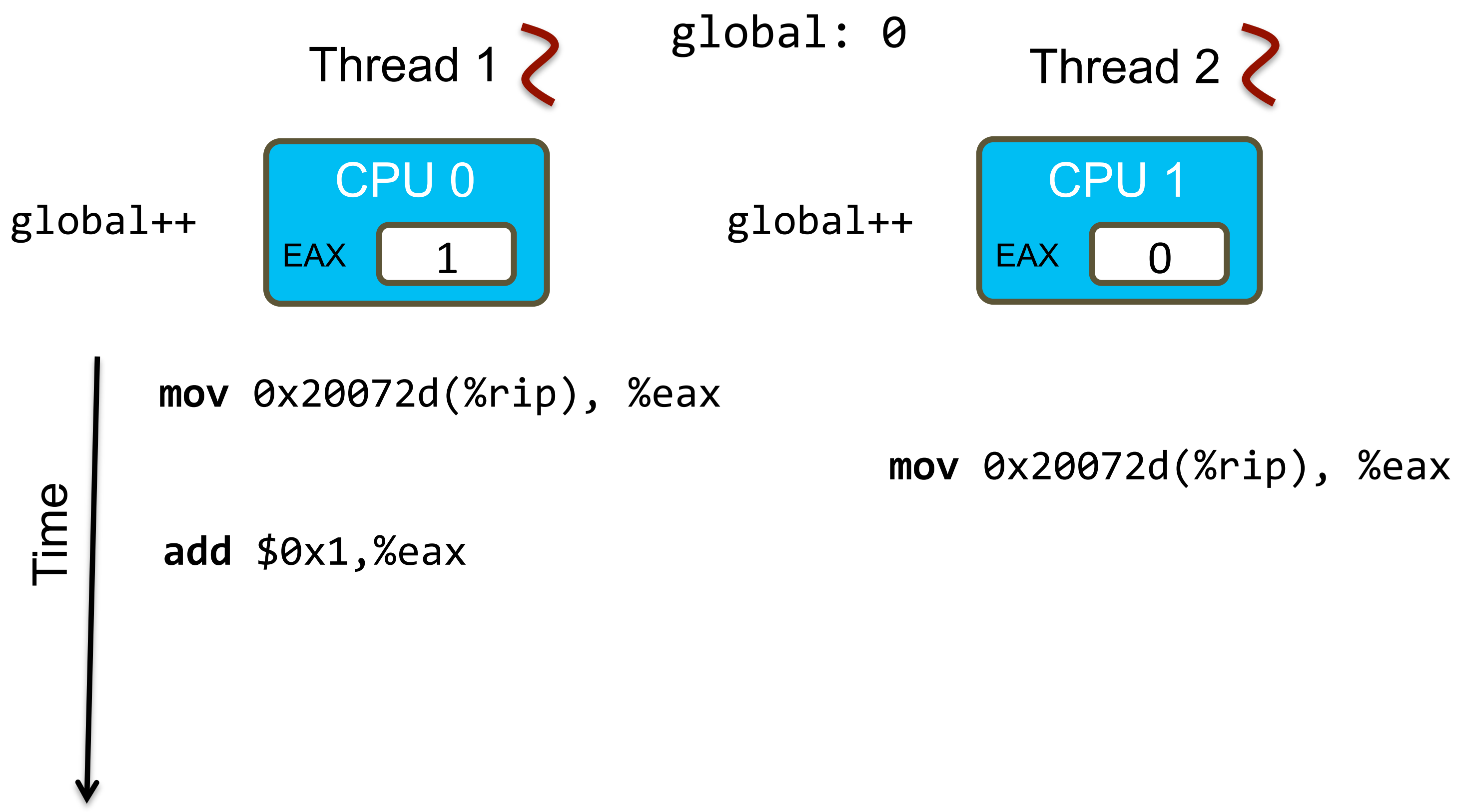
```
mov  0x20072d(%rip),%eax  // load global into %eax
add  $0x1,%eax            // update %eax by 1
mov  %eax,0x200724(%rip)  // restore global with %eax
```



```
mov 0x20072d(%rip), %eax
                                        mov 0x20072d(%rip), %eax
```

Example 1

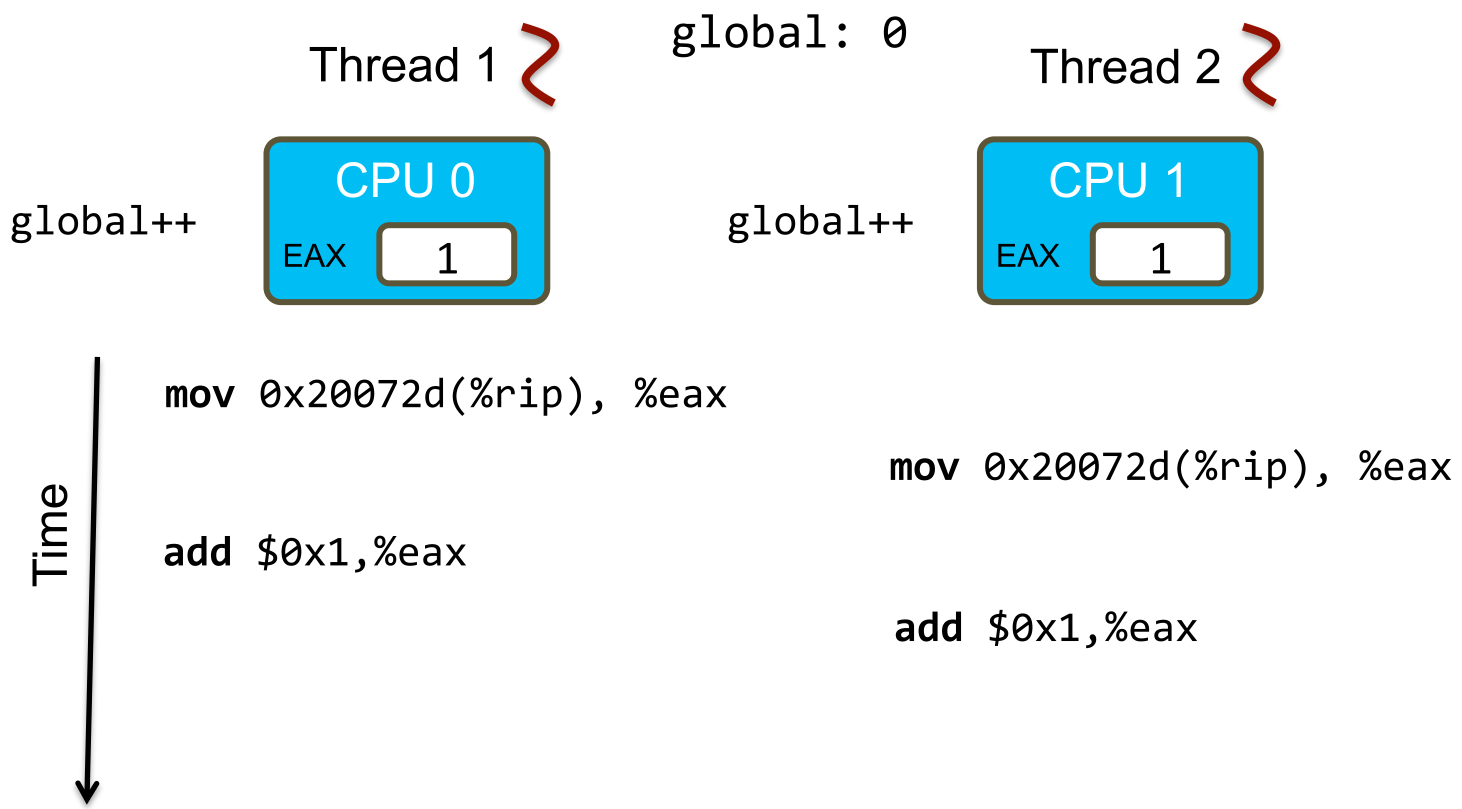
```
global++  →  mov  0x20072d(%rip),%eax  // load global into %eax
             add  $0x1,%eax            // update %eax by 1
             mov  %eax,0x200724(%rip)  // restore global with %eax
```



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

Example 1

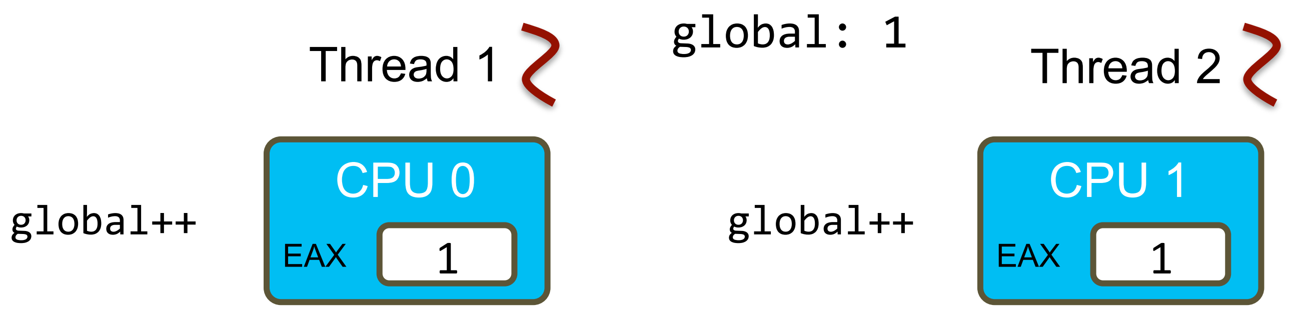
```
global++  →  mov  0x20072d(%rip),%eax  // load global into %eax
             add  $0x1,%eax             // update %eax by 1
             mov  %eax,0x200724(%rip)  // restore global with %eax
```



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

Example 1

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

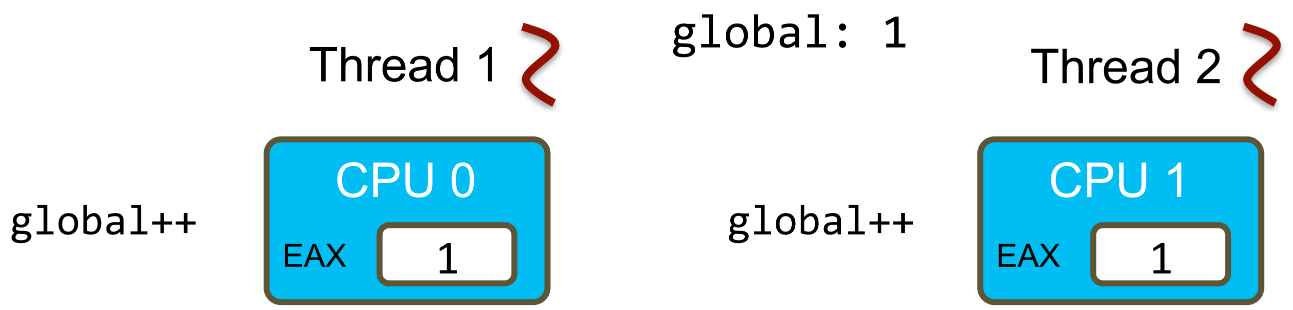


Time

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

Example 1

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



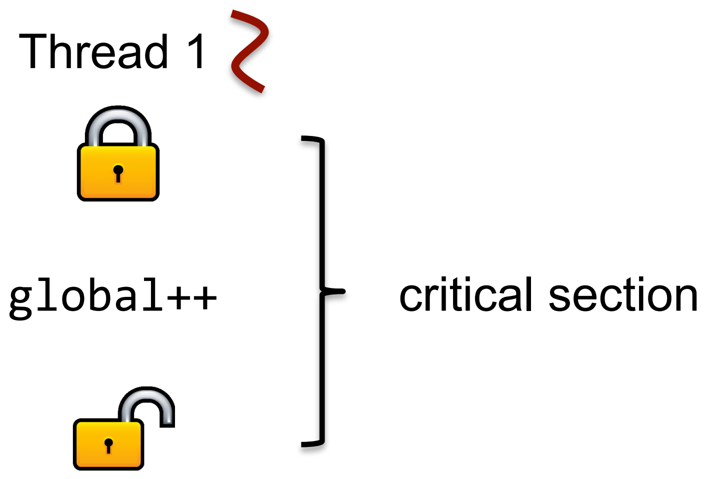
```
Time
 |   mov 0x20072d(%rip), %eax
 |                                      mov 0x20072d(%rip), %eax
 |   add $0x1,%eax
 |                                      add $0x1,%eax
 |   mov %eax, 0x20072d(%rip)
 v                                      mov %eax, 0x20072d(%rip)
```

Mutual exclusion

Prevent concurrent threads from accessing the shared resource at the same time.

Mutual exclusion

Prevent concurrent threads from accessing the shared resource at the same time. → Lock/Mutex



```
global++
```

Lock/Mutex API in pthread lib

`pthread_mutex_t`

- The type of mutex in pthread library
- Each mutex has two states: lock and unlock

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

Lock/Mutex API in pthread lib

`int pthread_mutex_lock(pthread_mutex_t *m)`

- lock the mutex `m`, if `m` is already locked, the calling threads blocks until the mutex is unlocked
- return value: 0 on success

```
int global = 0;
pthread_mutex_t mu;

void *add(void *) {
  pthread_mutex_lock(&mu);
  global++;

}
```

Lock/Mutex API in pthread lib

`int pthread_mutex_unlock(pthread_mutex_t *m)`

- unlock the mutex `m`
- return value: 0 on success

```
int global = 0;
pthread_mutex_t mu;

void *add(void *) {
  pthread_mutex_lock(&mu);
  global++;
  pthread_mutex_unlock(&mu);
}
```

Example 1 with Lock

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

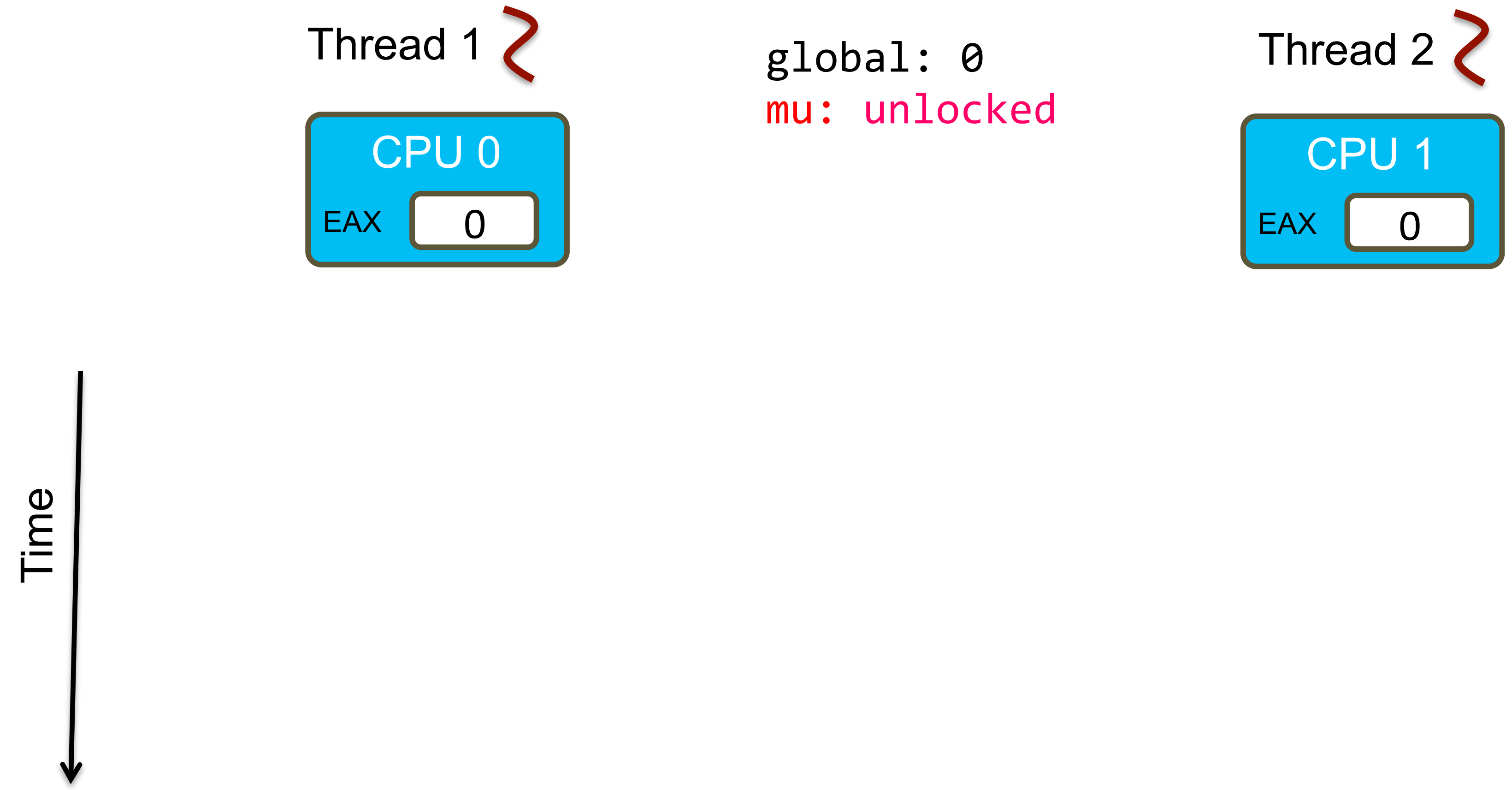
Thread 1

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

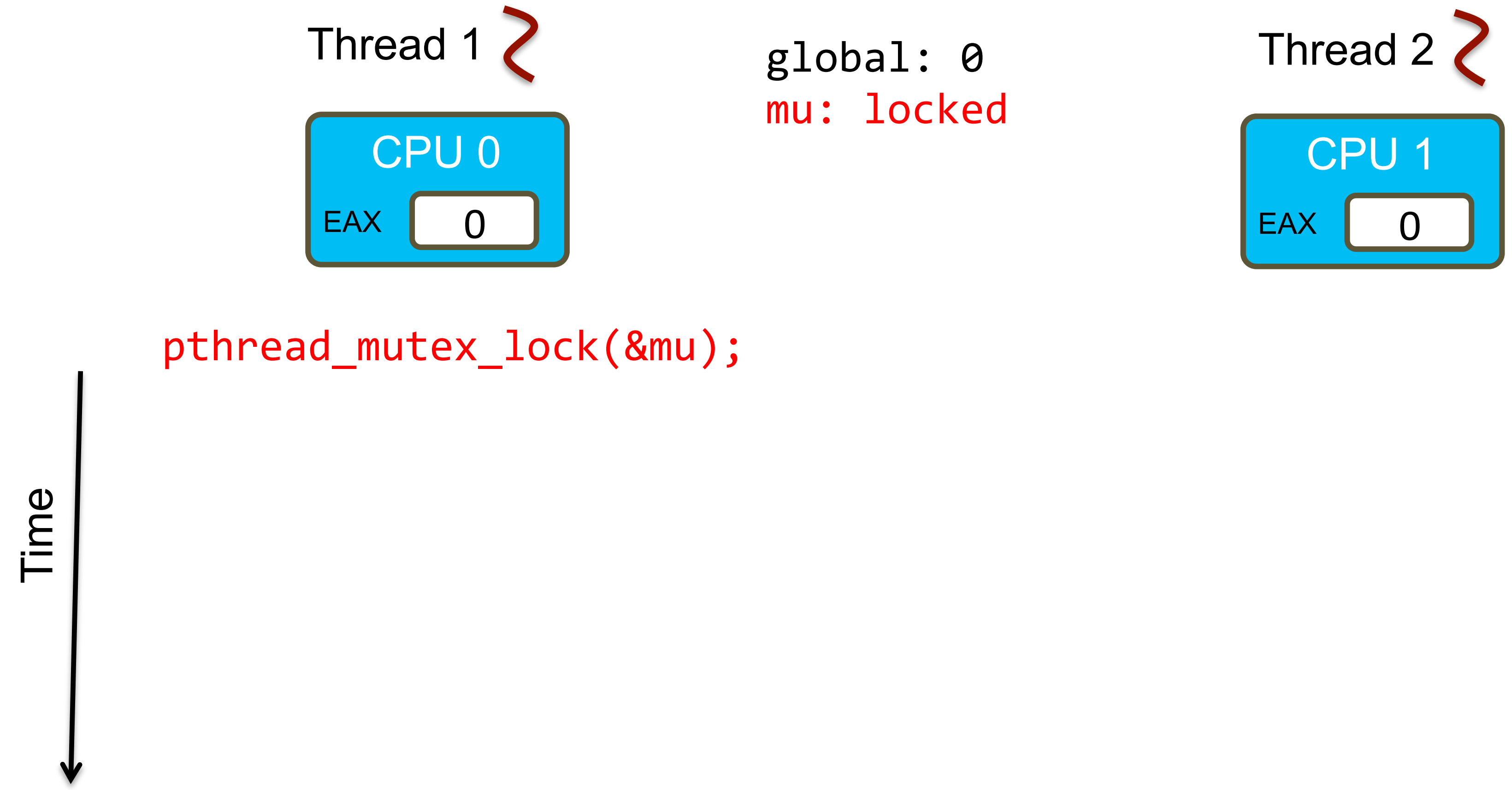
Thread 2

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

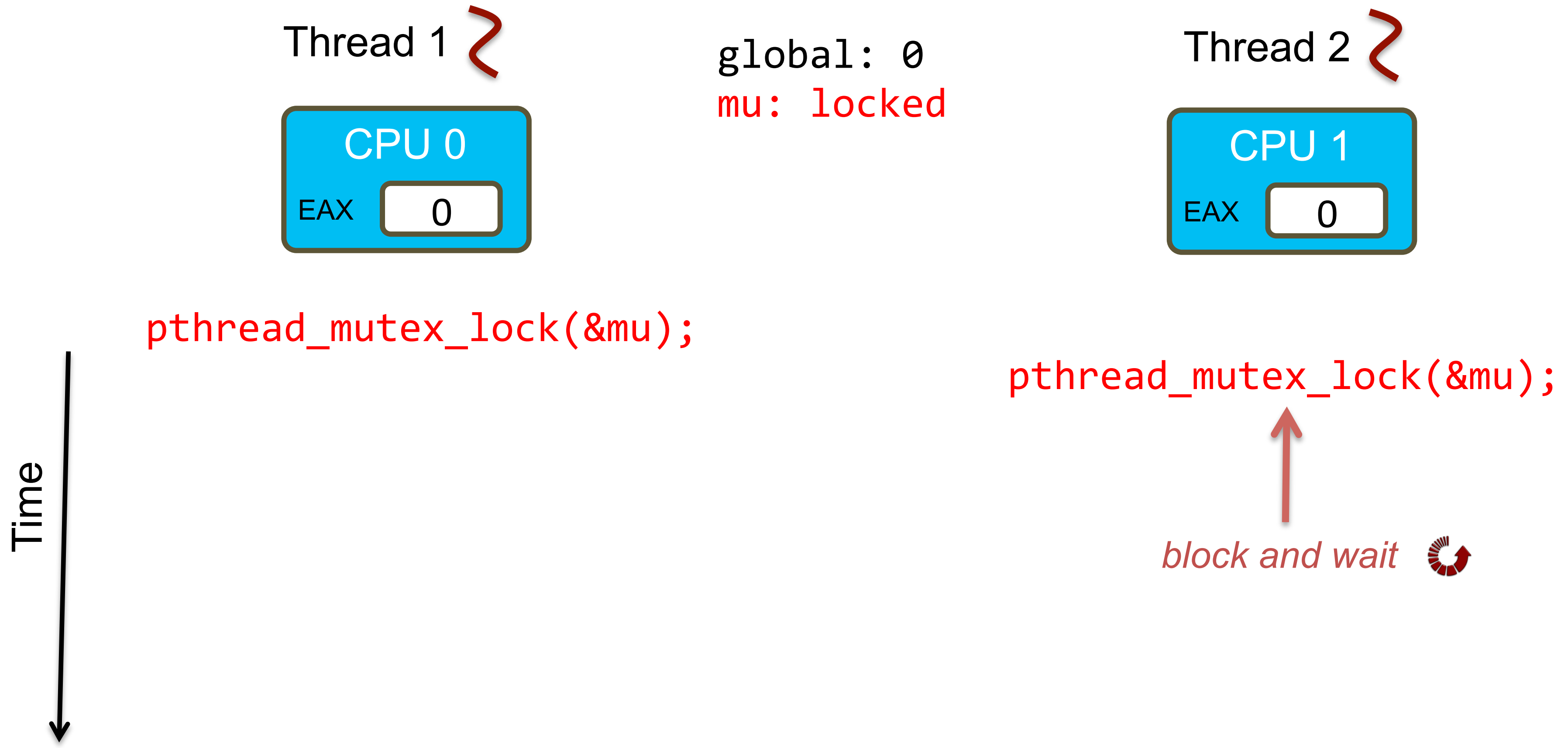
Example 1 with Lock



Example 1 with Lock



Example 1 with Lock

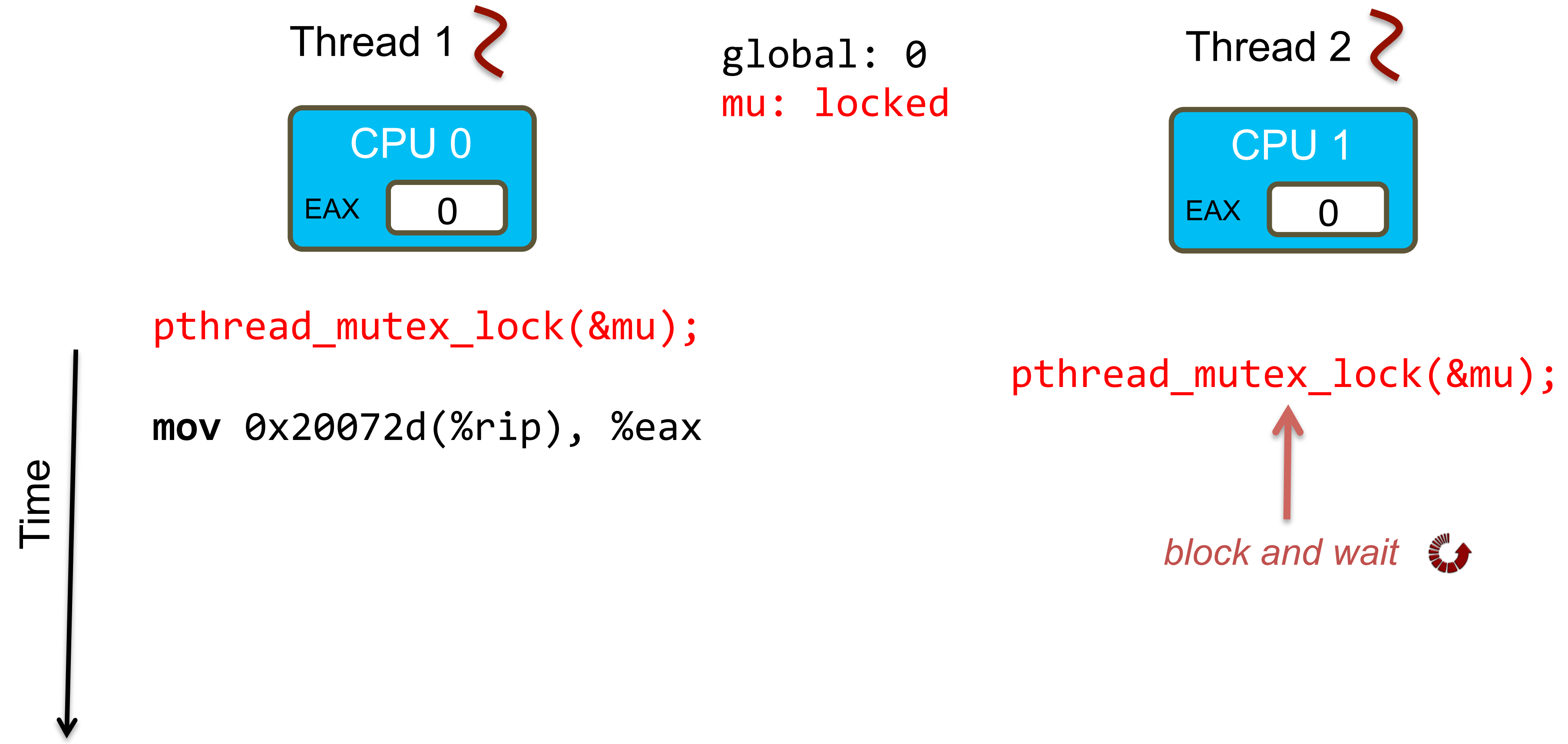


```
global: 0
mu: locked
```

```
pthread_mutex_lock(&mu);
```

```
pthread_mutex_lock(&mu);
```

Example 1 with Lock



Example 1 with Lock

Thread 1

CPU 0

EAX 1

global: 0

mu: locked

Thread 2

CPU 1

EAX 0

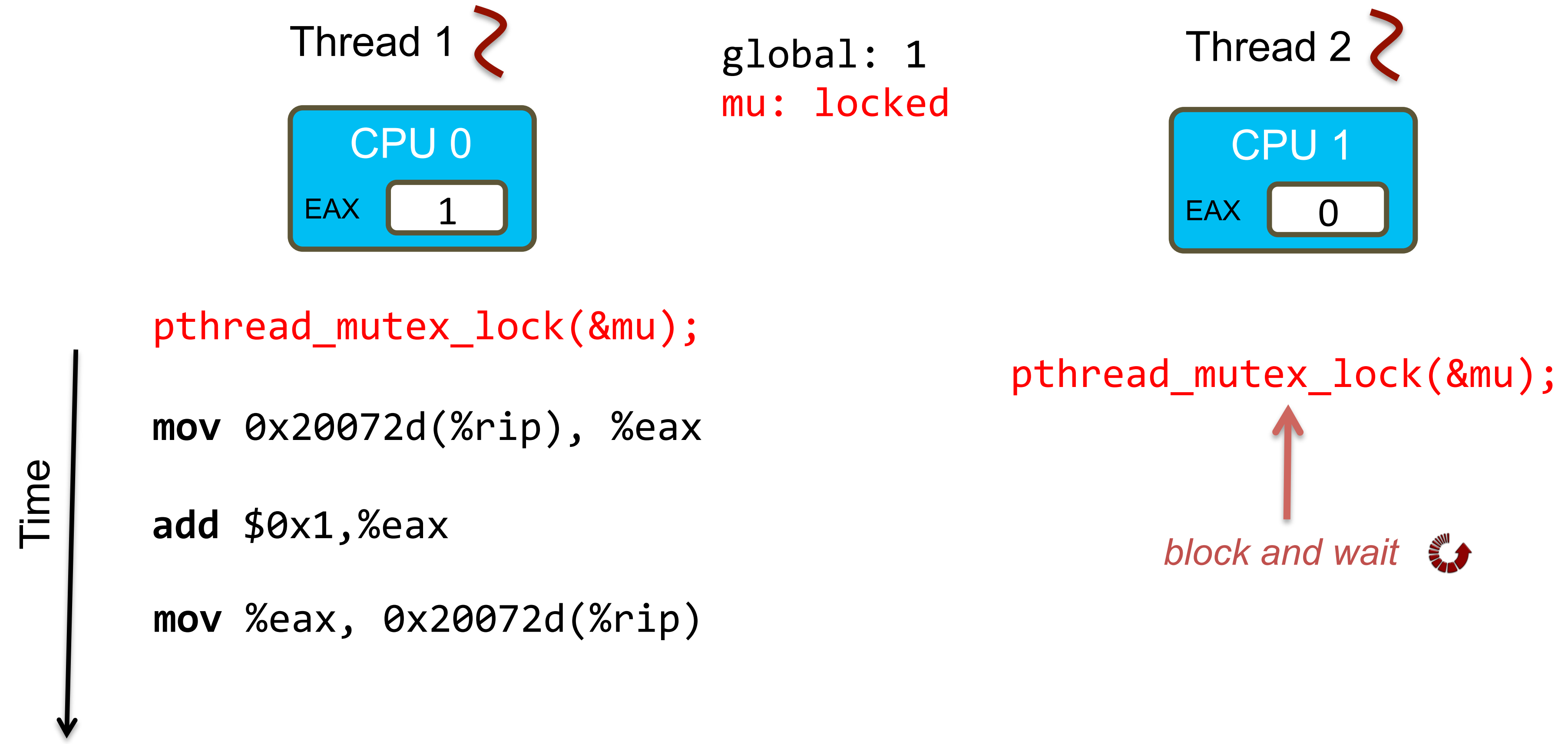
Time

```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax

add $0x1,%eax
```

```
pthread_mutex_lock(&mu);
```

block and wait

Example 1 with Lock



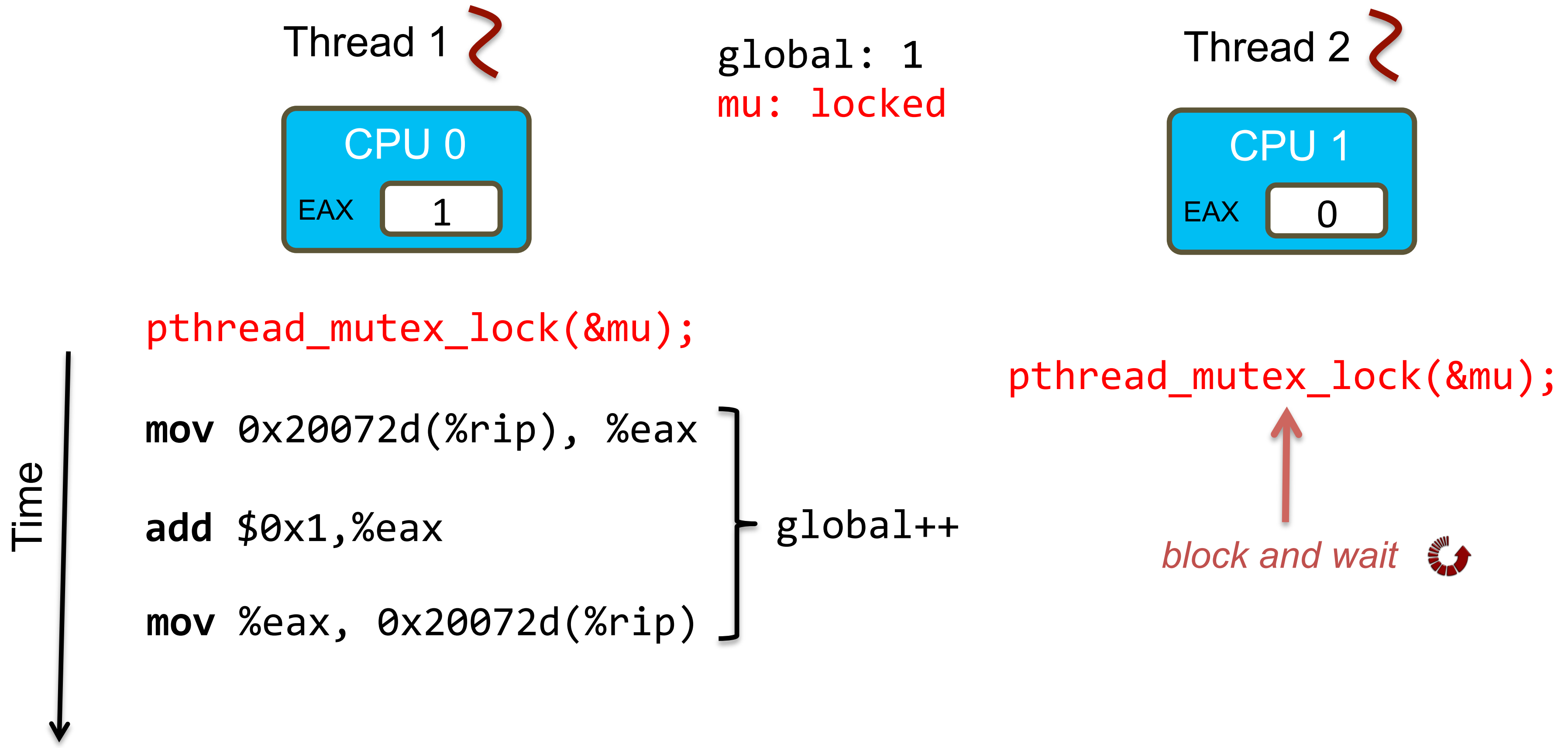
Thread 1:

```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax

add $0x1,%eax

mov %eax, 0x20072d(%rip)
```

Thread 2:

```
pthread_mutex_lock(&mu);
```

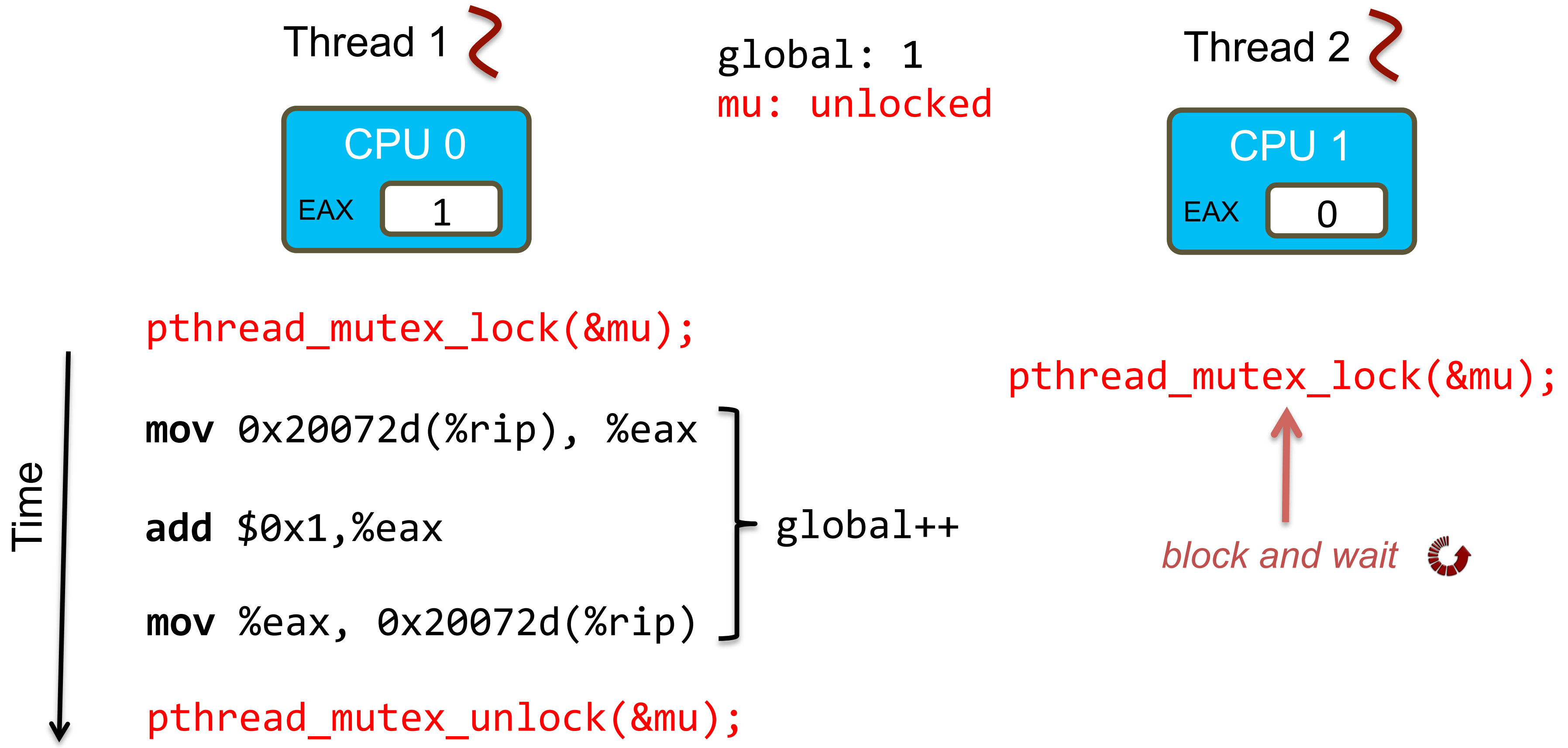
Example 1 with Lock



```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax

add $0x1,%eax

mov %eax, 0x20072d(%rip)
```

```
pthread_mutex_lock(&mu);
```

Example 1 with Lock



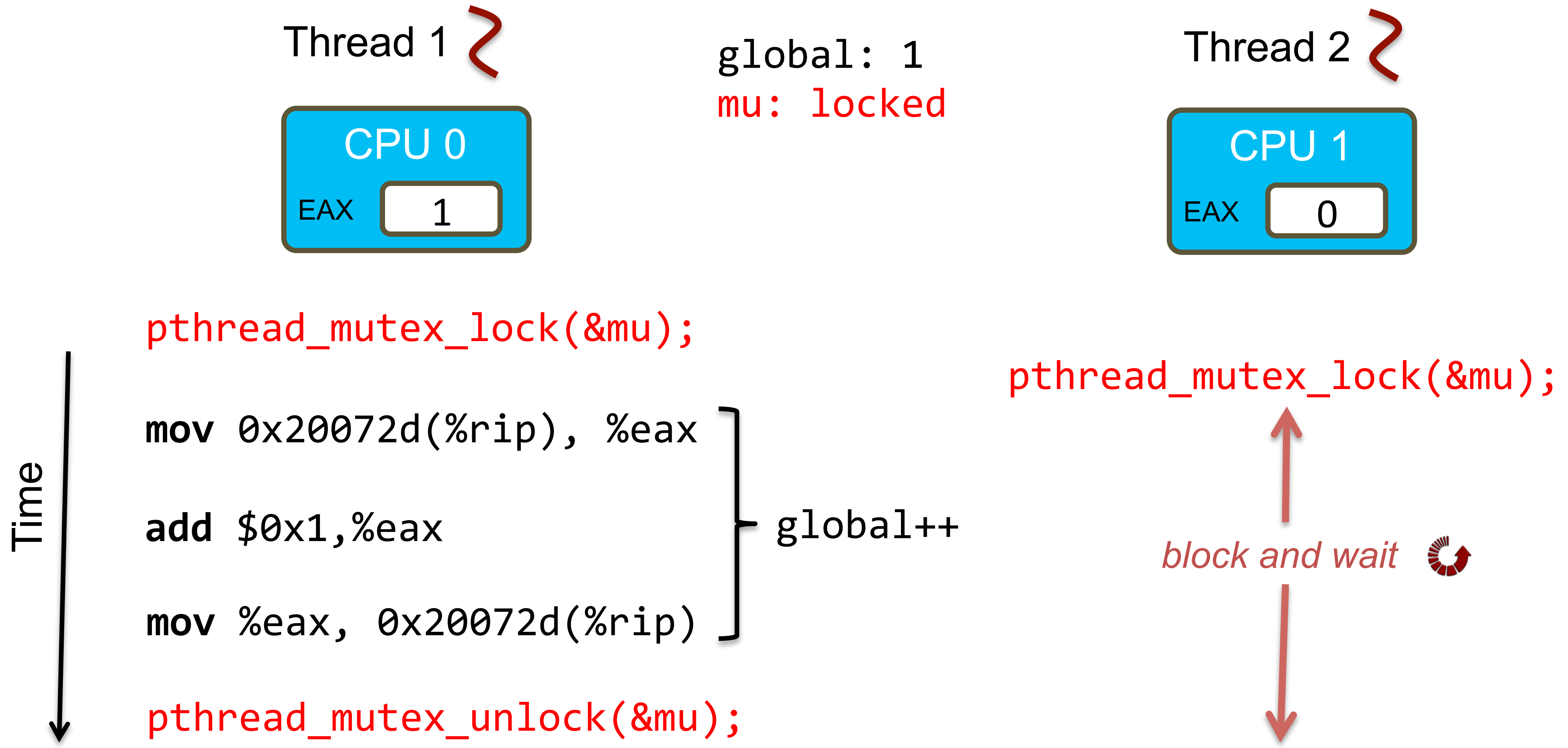
Thread 1:

```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax

add $0x1,%eax

mov %eax, 0x20072d(%rip)

pthread_mutex_unlock(&mu);
```

Thread 2:

```
pthread_mutex_lock(&mu);
```

Example 1 with Lock



```
global: 1
mu: locked
```

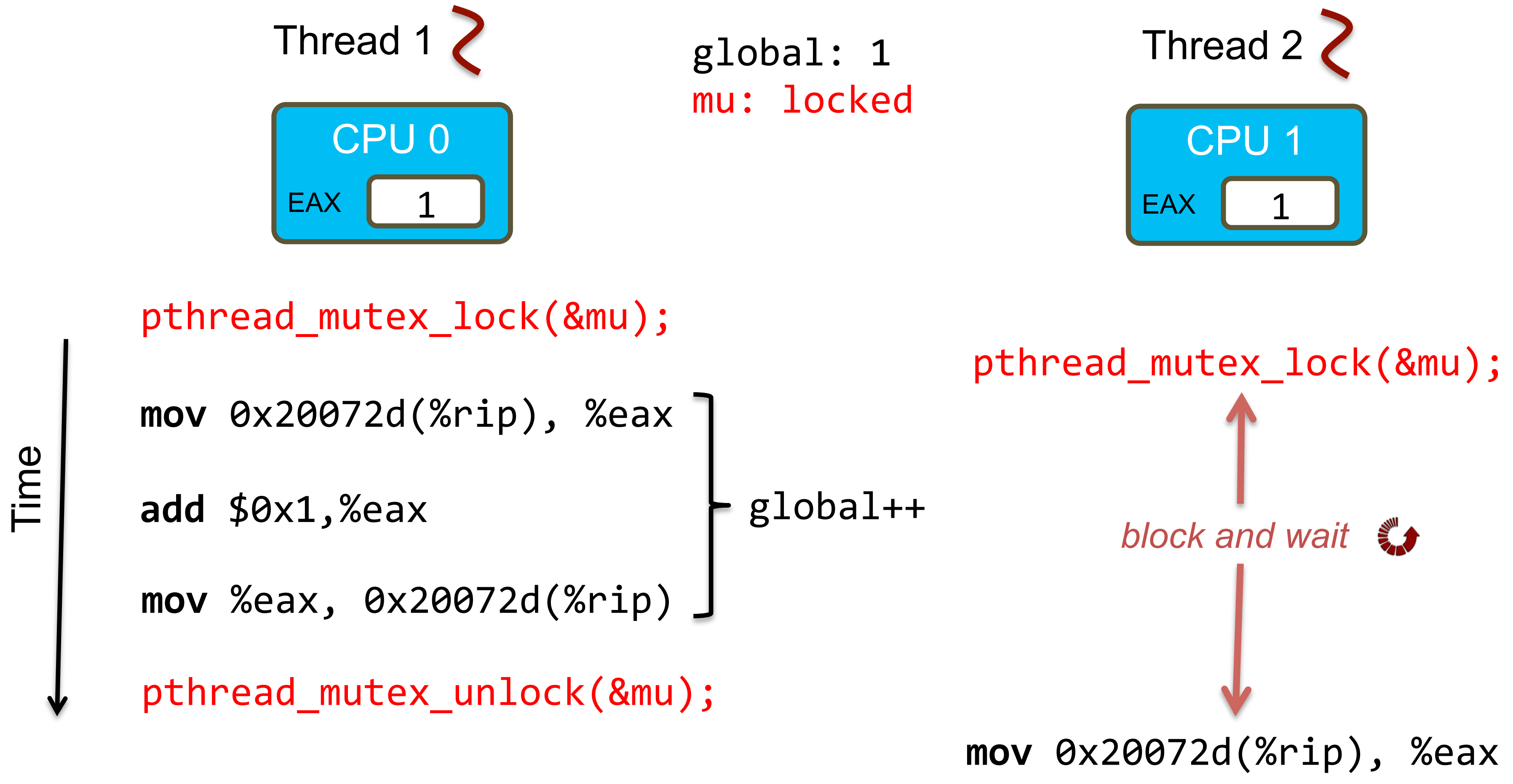
Thread 1:

```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax

add $0x1,%eax

mov %eax, 0x20072d(%rip)

pthread_mutex_unlock(&mu);
```

Thread 2:

```
pthread_mutex_lock(&mu);
```

Example 1 with Lock



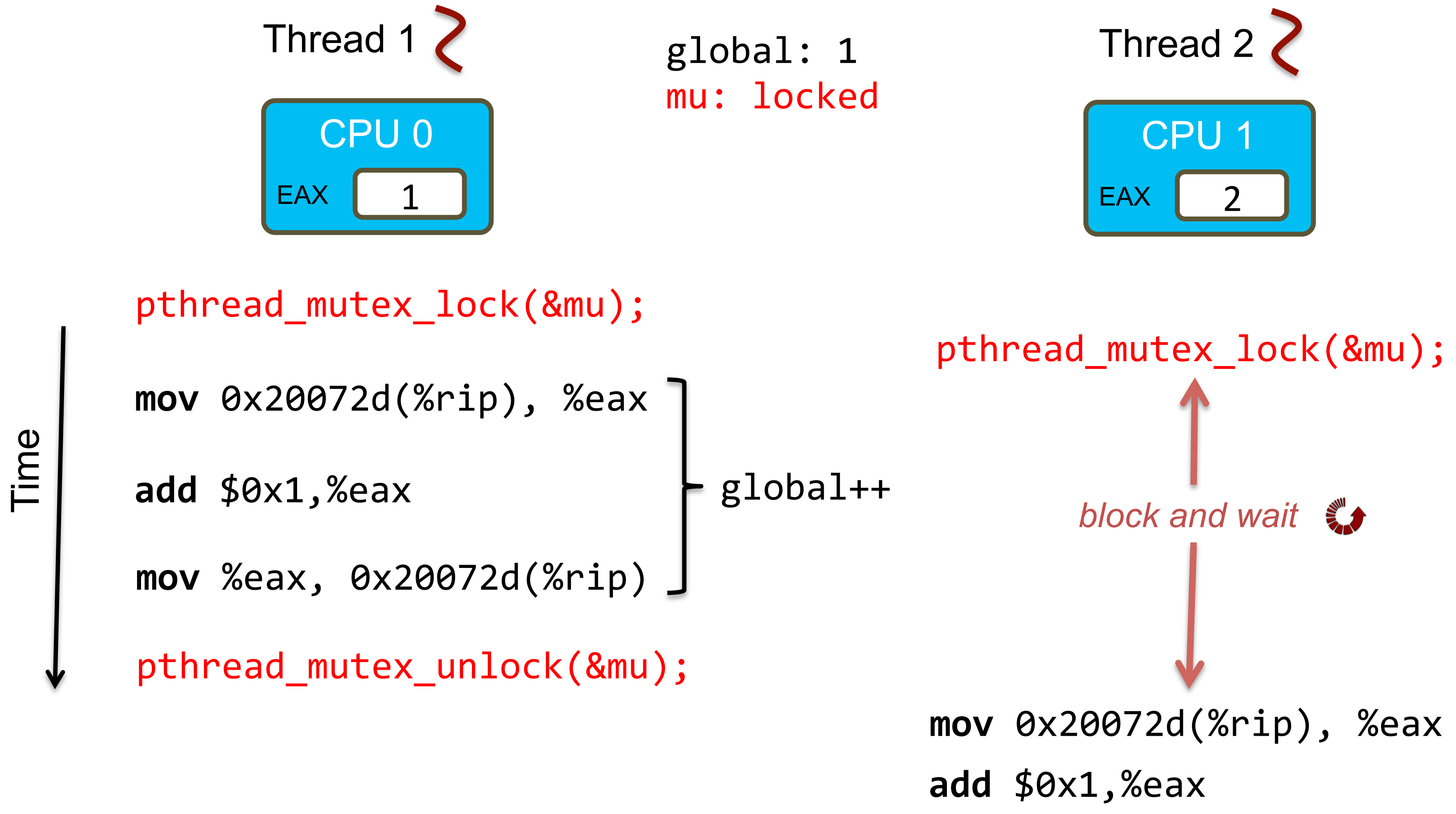
Thread 1:

```
pthread_mutex_lock(&mu);

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

pthread_mutex_unlock(&mu);
```

Thread 2:

```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax
```

Example 1 with Lock



```
global: 1
mu: locked
```

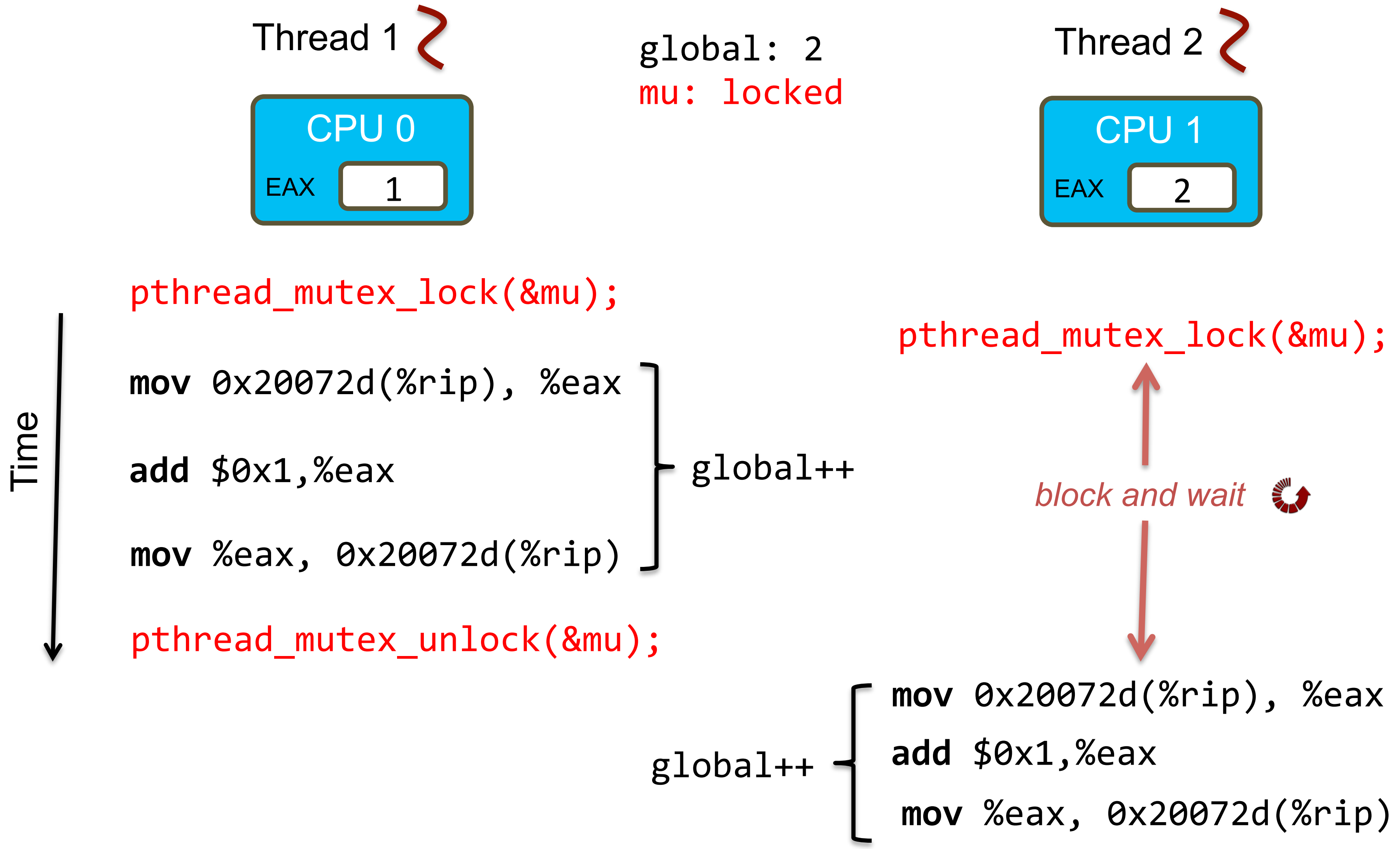
Thread 1:

```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax

add $0x1,%eax

mov %eax, 0x20072d(%rip)

pthread_mutex_unlock(&mu);
```

Thread 2:

```
pthread_mutex_lock(&mu);

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

Example 1 with Lock



global: 2
mu: locked

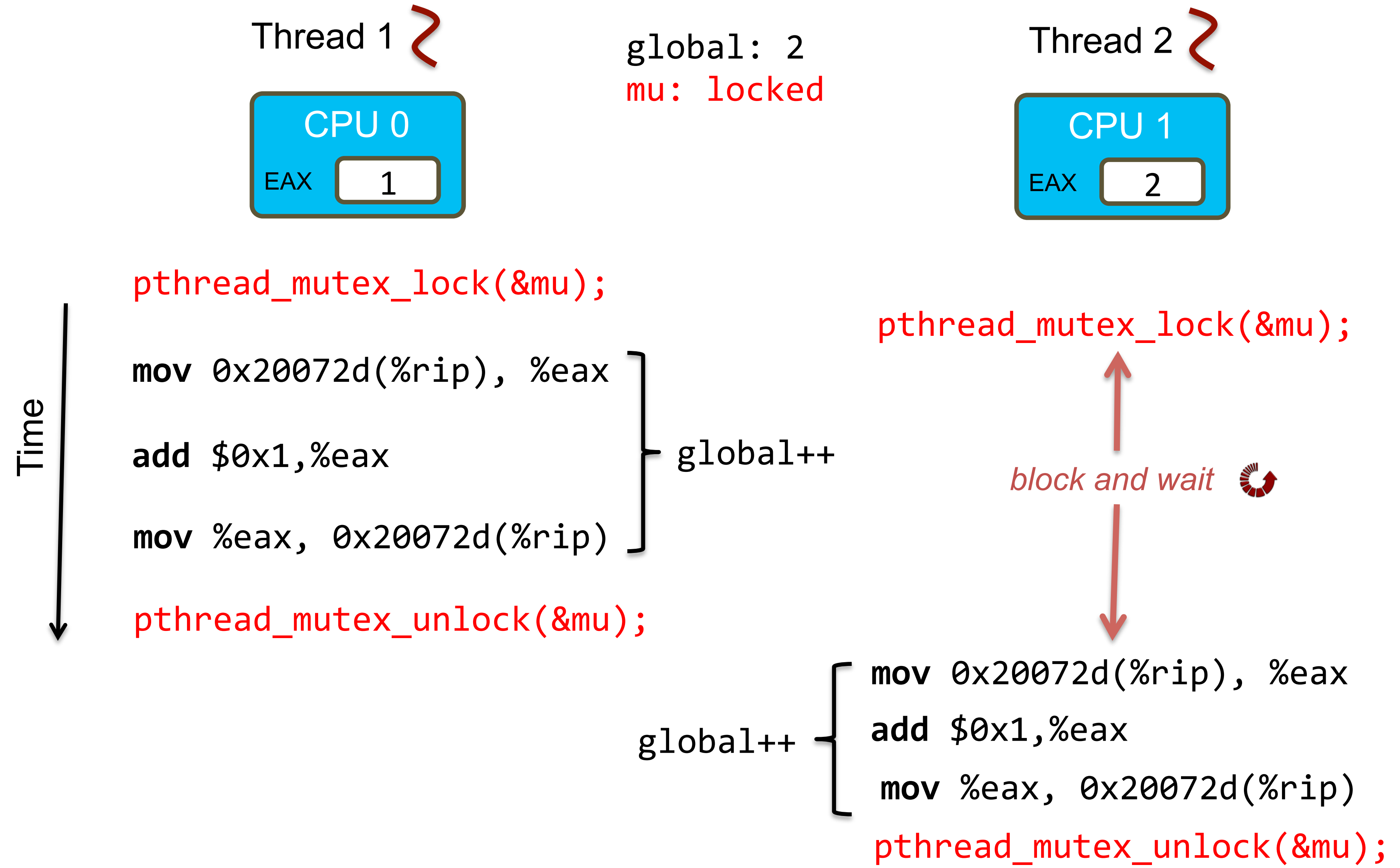
Thread 1:

```
pthread_mutex_lock(&mu);

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

pthread_mutex_unlock(&mu);
```

Thread 2:

```
pthread_mutex_lock(&mu);

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

Example 1 with Lock



Thread 1:

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip), %eax
add $0x1,%eax
mov %eax, 0x20072d(%rip)
pthread_mutex_unlock(&mu);
```

Thread 2:

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip), %eax
add $0x1,%eax
mov %eax, 0x20072d(%rip)
pthread_mutex_unlock(&mu);
```

Example 2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
}
```

Example 2

Each thread updates 2 random elements from a shared array

```c
int array[10];
pthread_mutex_t mu;

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Which one is correct?

```c
int array[10];
pthread_mutex_t mu;

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    pthread_mutex_lock(&mu);
    array[idx]++;
    pthread_mutex_unlock(&mu);
  }
}
```

Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

Thread 1 Thread 2

```
pthread_mutex_lock(&mu);
```

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

🔒 Thread 1

Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

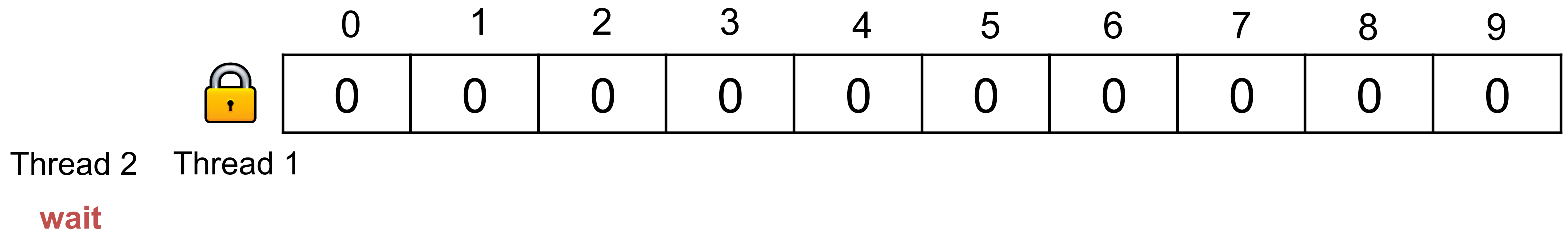
Thread 1

`pthread_mutex_lock(&mu);`

Thread 2

`pthread_mutex_lock(&mu);`

(block and wait)



Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

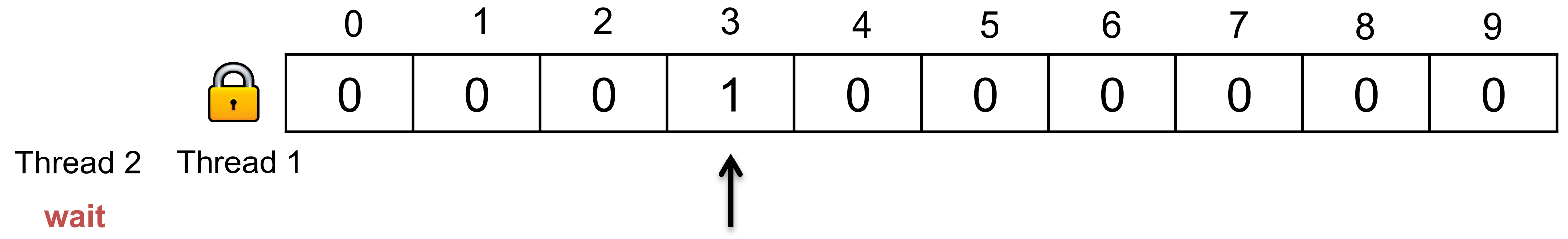
Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)



Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

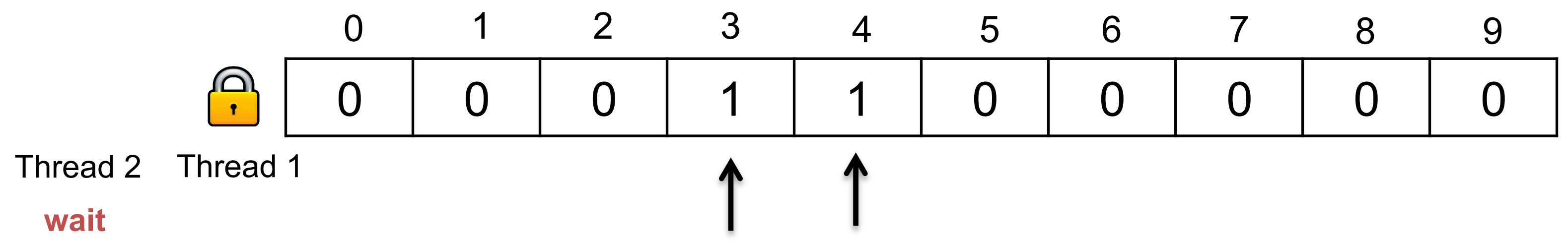
Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
array[4]++;
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)



Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

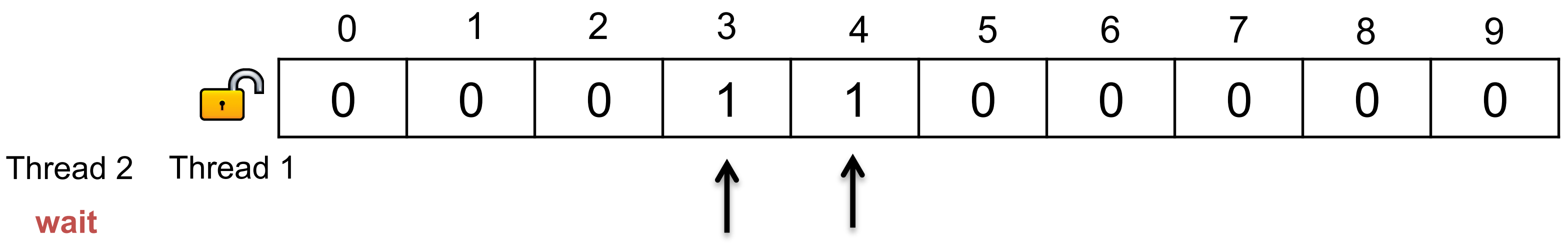
Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
array[4]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)



Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
array[4]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)

0	1	2	3	4	5	6	7	8	9
0	0	0	1	1	0	0	0	0	0

Thread 2

Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

Thread 1

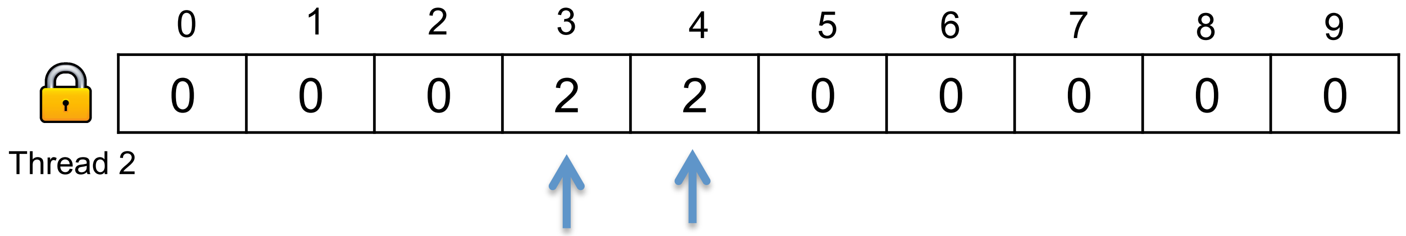
```
pthread_mutex_lock(&mu);
array[3]++;
array[4]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)

```
array[3]++;
array[4]++;
```



0	1	2	3	4	5	6	7	8	9
0	0	0	2	2	0	0	0	0	0

Example 2.1

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  pthread_mutex_lock(&mu);
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    array[idx]++;
  }
  pthread_mutex_unlock(&mu);
}
```

Both of them update elements 3 and 4

Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
array[4]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)

```
array[3]++;
array[4]++;
pthread_mutex_unlock(&mu);
```

0	1	2	3	4	5	6	7	8	9
0	0	0	2	2	0	0	0	0	0

Thread 2

Example 2.2

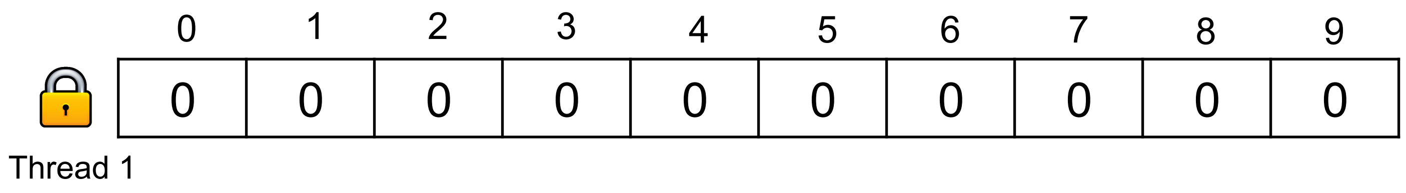
Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

Thread 1 Thread 2

```
pthread_mutex_lock(&mu);
```



Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

Thread 1

`pthread_mutex_lock(&mu);`

Thread 2

`pthread_mutex_lock(&mu);`

(block and wait)

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Thread 2 Thread 1

wait

Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

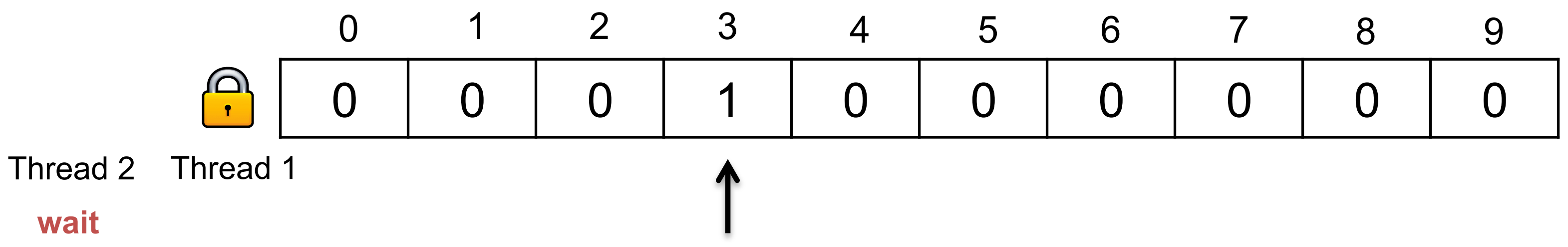
Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)



Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

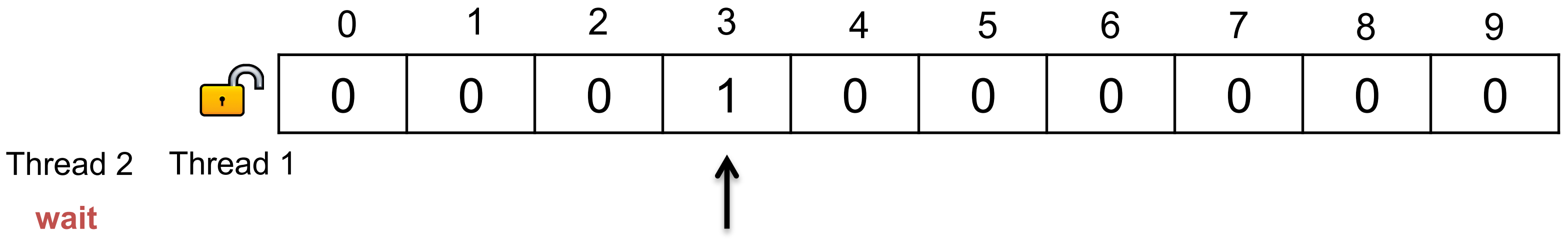
Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)



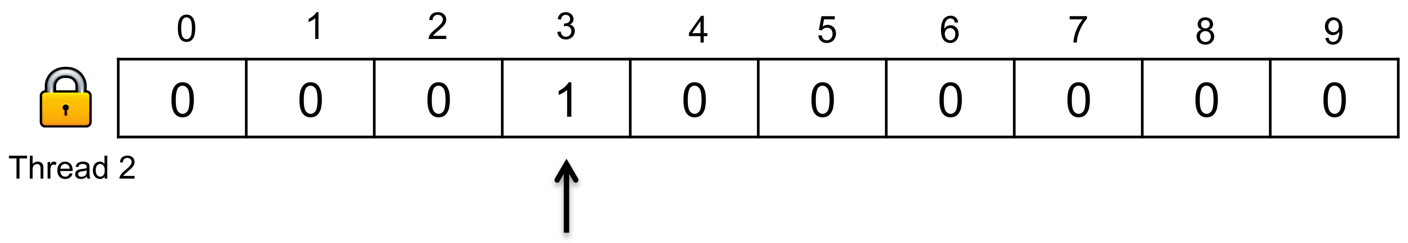
Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

Thread 1	Thread 2
`pthread_mutex_lock(&mu);`	`pthread_mutex_lock(&mu);`
`array[3]++;`	***(block and wait)***
`pthread_mutex_unlock(&mu);`	



Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

Thread 1

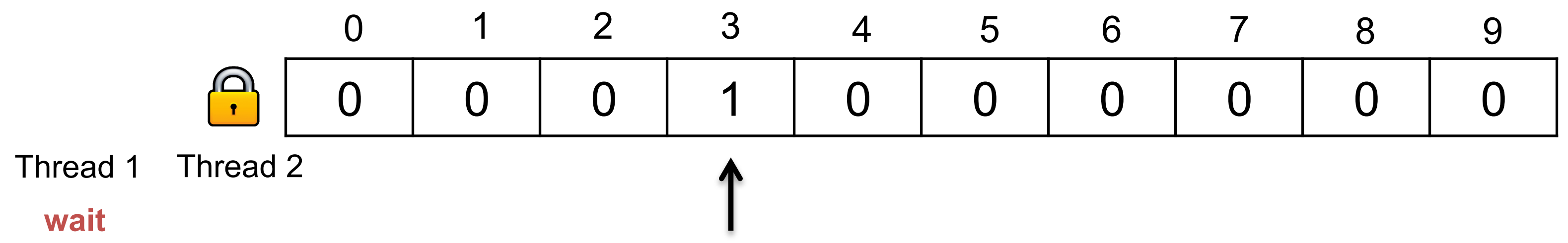
```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
```

(block and wait)

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)



Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
```

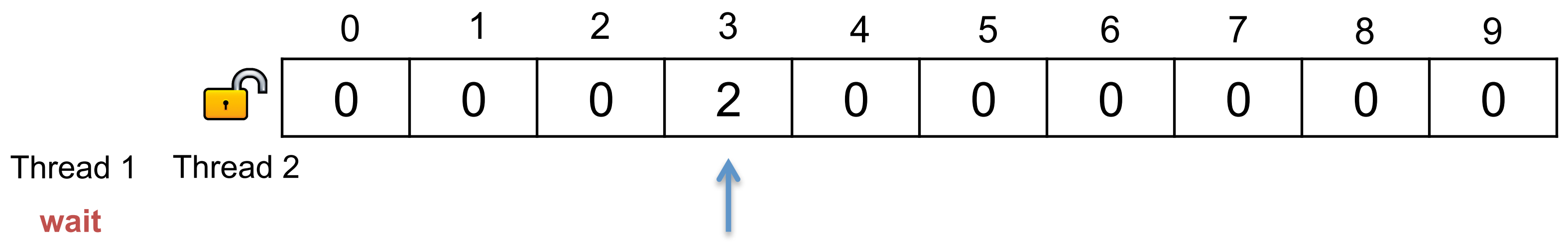
(block and wait)

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)

```
array[3]++;
pthread_mutex_unlock(&mu);
```



Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
```

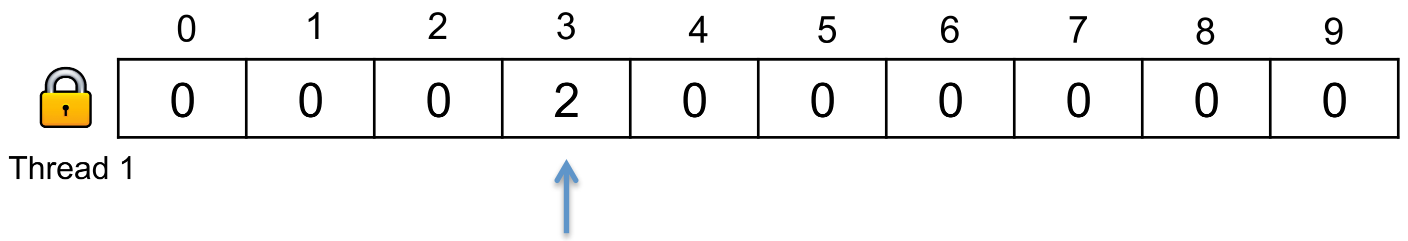
(block and wait)

Thread 2

```
pthread_mutex_lock(&mu);
```

(block and wait)

```
array[3]++;
pthread_mutex_unlock(&mu);
```



Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

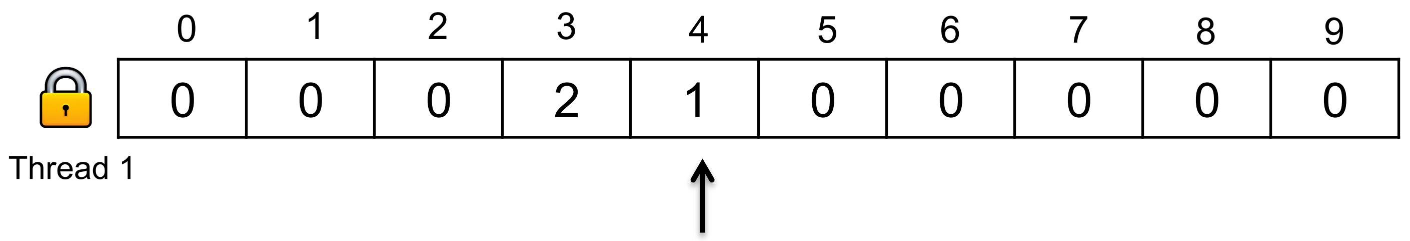
Both of them update elements 3 and 4

Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
(block and wait)
array[4]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
(block and wait)

array[3]++;
pthread_mutex_unlock(&mu);
```



0	1	2	3	4	5	6	7	8	9
0	0	0	2	1	0	0	0	0	0

Thread 1

Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

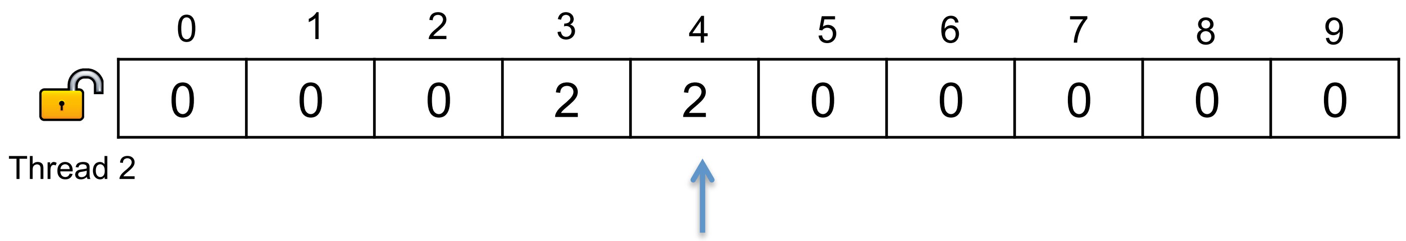
Both of them update elements 3 and 4

Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
(block and wait)
array[4]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
(block and wait)

array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
(block and wait)
 array[4]++;
pthread_mutex_unlock(&mu);
```



Example 2.2

Each thread updates 2 random elements from a shared array

```
int array[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
   pthread_mutex_lock(&mu);
    array[idx]++;
   pthread_mutex_unlock(&mu);
  }
}
```

Both of them update elements 3 and 4

Thread 1

```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
(block and wait)
array[4]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
(block and wait)

array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
(block and wait)
 array[4]++;
pthread_mutex_unlock(&mu);
```

0	1	2	3	4	5	6	7	8	9
0	0	0	2	2	0	0	0	0	0

Thread 2

What is the problem?

Example 2.3

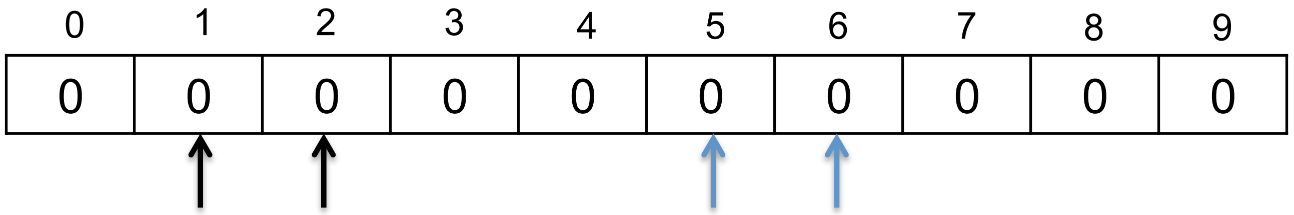
Each thread updates 2 random elements from a shared array

Thread 1

```
pthread_mutex_lock(&mu);
array[1]++;
array[2]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
array[5]++;
array[6]++;
pthread_mutex_unlock(&mu);
```



These two threads' execution always be serialized, even they access different elements.

False contention

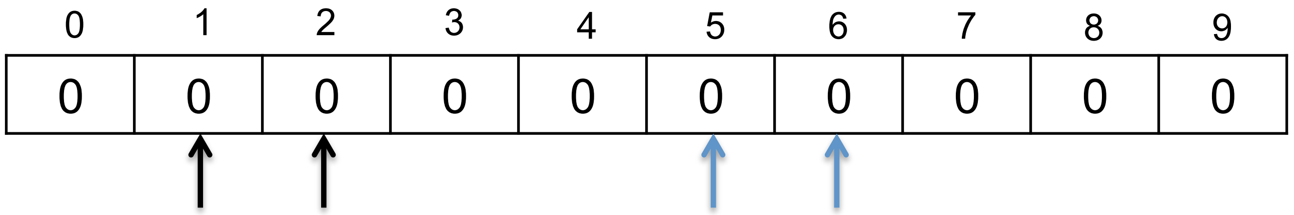
Each thread updates 2 random elements from a shared array

Thread 1

```
pthread_mutex_lock(&mu);
array[1]++;
array[2]++;
pthread_mutex_unlock(&mu);
```

Thread 2

```
pthread_mutex_lock(&mu);
array[5]++;
array[6]++;
pthread_mutex_unlock(&mu);
```



These two threads' execution always be serialized, even they access different elements.

How to improve it?

Lock granularity

Coarse granularity

- A global lock, the lock is associated with the entire array

Fine granularity

- Multiple locks, each lock is associated with a single element

Example 2.3

Each thread updates 2 random elements from a shared array

```
int array[10];
pthread_mutex_t locks[10];

void *thr(void *) {
  for(int i = 0; i < 2; i++) {
    int idx = random() % 10;
    pthread_mutex_lock(&locks[idx]);
    array[idx]++;
    pthread_mutex_unlock(&locks[idx]);
  }
}
```

Example 2.3

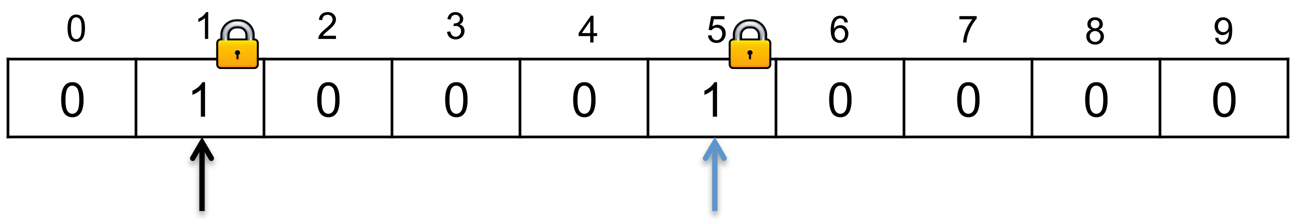
Each thread updates 2 random elements from a shared array

Thread 1

```
pthread_mutex_lock(&mu[1]);
array[1]++;
pthread_mutex_unlock(&mu[1]);
```

Thread 2

```
pthread_mutex_lock(&mu[5]);
array[5]++;
pthread_mutex_unlock(&mu[5]);
```



Example 2.3

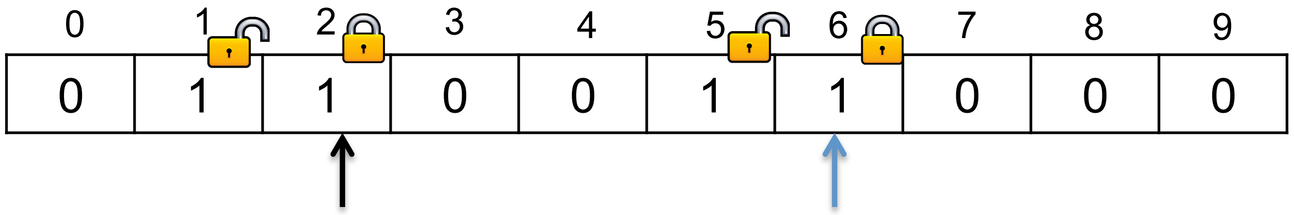
Each thread updates 2 random elements from a shared array

Thread 1

```
pthread_mutex_lock(&mu[1]);
array[1]++;
pthread_mutex_unlock(&mu[1]);
pthread_mutex_lock(&mu[2]);
array[2]++;
pthread_mutex_unlock(&mu[2]);
```

Thread 2

```
pthread_mutex_lock(&mu[5]);
array[5]++;
pthread_mutex_unlock(&mu[5]);
pthread_mutex_lock(&mu[6]);
array[6]++;
pthread_mutex_unlock(&mu[6]);
```



Example 3

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];

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

int sum(int x, int y)
{
    return accounts[x]->val + accounts[y]->val;
}
```

Example 3

Each thread may invoke `transfer` to transfer money from x to y, or invoke `sum` to read the account information.

No thread is able to observe the middle state of the transfer.

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];

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

int sum(int x, int y)
{
    return accounts[x]->val + accounts[y]->val;
}
```

Thread 1	Thread 2
`transfer(1, 2, 10)`	`sum(1, 2)`

Example 3

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mu;

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mu);
    int a = accounts[x]->val + accounts[y]->val;
    pthread_mutex_unlock(&mu);
    return a;
}
```

Each thread may invoke `transfer` to transfer money from x to y, or invoke `sum` to read the account information.

No thread is able to observe the middle state of the transfer.

Thread 1	Thread 2
`transfer(1, 2, 10)`	`sum(1, 2)`

Example 3

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mu;

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mu);
    int a = accounts[x]->val + accounts[y]->val;
    pthread_mutex_unlock(&mu);
    return a;
}
```

Each thread may invoke `transfer` to transfer money from x to y, or invoke sum to read the account information

Thread 1	Thread 2
`transfer(1, 2, 10)`	`sum(1, 2)`

Can you improve this impl. with fine-grained lock?

Example 3

```c
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    int xv = accounts[x]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Each thread may invoke `transfer` to transfer money from x to y, or invoke `sum` to read the account information.

No thread is able to observe the middle state of the transfer.

Thread 1	Thread 2
`transfer(1, 2, 10)`	`sum(1, 2)`

```
typedef struct {
   char *name;
   int val;
} account;
```

Example 3

```
account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    int xv = accounts[x]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Each thread may invoke `transfer` to transfer money from x to y, or invoke sum to read the account information.

No thread is able to observe the middle state of the transfer.

Thread 1	Thread 2
`transfer(1, 2, 10)`	`sum(1, 2)`

Any problem?

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    int xv = accounts[x]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Example 3

Thread 1

```
transfer(1, 2, 10)
```

Thread 2

```
sum(1, 2)
```

0	1	2	3	4	5	6	7	8	9
100	100	100	100	100	100	100	100	100	100

Example 3

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    int xv = accounts[x]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Thread 1

```
transfer(1, 2, 10)

pthread_mutex_lock(&mus[1]);
accounts[1]->val -= 10;
pthread_mutex_unlock(&mus[1]);
```

Thread 2

```
sum(1, 2)
```

0	1	2	3	4	5	6	7	8	9
100	90	100	100	100	100	100	100	100	100

Example 3

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    int xv = accounts[x]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Thread 1

```
transfer(1, 2, 10)

pthread_mutex_lock(&mus[1]);
accounts[1]->val -= 10;
pthread_mutex_unlock(&mus[1]);
```

Thread 2

```
sum(1, 2)  (190)

pthread_mutex_lock(&mus[1]);
int xv = accounts[1]->val;
pthread_mutex_unlock(&mus[1]);
pthread_mutex_lock(&mus[2]);
int yv = accounts[2]->val;
pthread_mutex_unlock(&mus[2]);
return xv + yv;
```

0	1	2	3	4	5	6	7	8	9
100	90	100	100	100	100	100	100	100	100

Example 3

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    int xv = accounts[x]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Thread 1

```
transfer(1, 2, 10)

pthread_mutex_lock(&mus[1]);
accounts[1]->val -= 10;
pthread_mutex_unlock(&mus[1]);
```

Thread 2

```
sum(1, 2)  (190)

pthread_mutex_lock(&mus[1]);
int xv = accounts[1]->val;
pthread_mutex_unlock(&mus[1]);
pthread_mutex_lock(&mus[2]);
int yv = accounts[2]->val;
pthread_mutex_unlock(&mus[2]);
return xv + yv;
```

```
pthread_mutex_lock(&mus[2]);
accounts[2]->val += 10;
pthread_mutex_unlock(&mus[2]);
```

0	1	2	3	4	5	6	7	8	9
100	90	100	100	100	100	100	100	100	100

```c
typedef struct {
   char *name;
   int val;
} account;
```

Example 3

```c
account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int xv = accounts[x]->val;
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

No thread is able to observe the middle state of the transfer.

➔ Still hold x's lock when access y.

Thread 1	Thread 2
`transfer(1, 2, 10)`	`sum(1, 2)`

Example 3

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int xv = accounts[x]->val;
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

No thread is able to observe the middle state of the transfer.

➔ Still hold x's lock when access y.

Thread 1	Thread 2
`transfer(1, 2, 10)`	`sum(1, 2)`

Any problem?

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int xv = accounts[x]->val;
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Deadlock

Thread 1

```
transfer(1, 2, 10)
```

Thread 2

```
sum(2, 1)
```

Deadlock

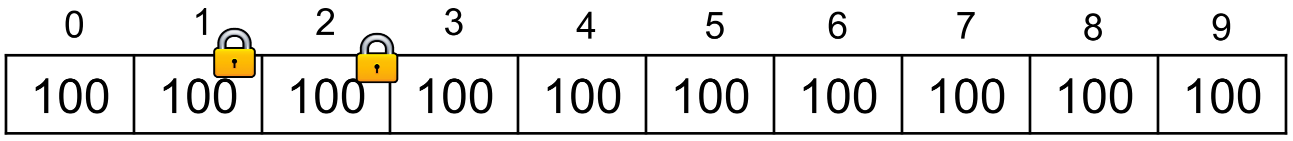
```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int xv = accounts[x]->val;
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Thread 1

```
transfer(1, 2, 10)
pthread_mutex_lock(&mus[1]);
```

Thread 2

```
sum(2, 1)
pthread_mutex_lock(&mus[2]);
```



Deadlock

```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int xv = accounts[x]->val;
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Thread 1

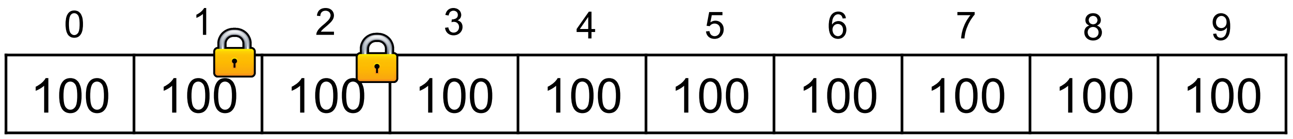
```
transfer(1, 2, 10)
pthread_mutex_lock(&mus[1]);
pthread_mutex_lock(&mus[2]);
```

wait for thread 2 to release mus[2]

Thread 2

```
sum(2, 1)
pthread_mutex_lock(&mus[2]);
pthread_mutex_lock(&mus[1]);
```

wait for thread 1 to release mus[1



```
typedef struct {
   char *name;
   int val;
} account;

account *accounts[10];
pthread_mutex_t mus[10];

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

int sum(int x, int y)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    int xv = accounts[x]->val;
    int yv = accounts[y]->val;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
    return xv + yv;
}
```

Deadlock

Thread 1

```
transfer(1, 2, 10)
pthread_mutex_lock(&mus[1]);
pthread_mutex_lock(&mus[2]);
```

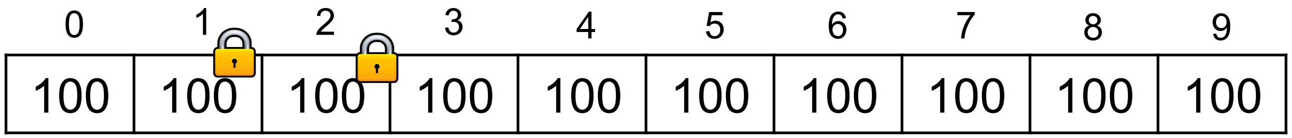
wait for thread 2 to release mus[2]

Thread 2

```
sum(2, 1)
pthread_mutex_lock(&mus[2]);
pthread_mutex_lock(&mus[1]);
```

wait for thread 1 to release mus[1

Program can not make progress!



Trick I to prevent deadlock

Observation

- A deadlock occurs only if each thread is holding at least one lock and being blocked by another lock.

Trick

- Use "`trylock`" to avoid thread being blocked.

Trick I to prevent deadlock

- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - Be equivalent to `pthread_mutex_lock`(), except that if the mutex is currently locked, the call shall return immediately.
  - Return value:

    Zero: acquire the lock successfully;

    Non-Zero: lock is held by others

Trick I to prevent deadlock

- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - Be equivalent to `pthread_mutex_lock()`, except that if the mutex is currently locked, the call shall return immediately.
  - Return value:

    Zero: acquire the lock successfully;

    Non-Zero: lock is held by others

```
void transfer(int x, int y, int amount)
{
retry:
    pthread_mutex_lock(&mus[x]);
    int succ = pthread_mutex_trylock(&mus[y]);
    if (succ != 0) {
        pthread_mutex_unlock(&mus[x]);
           goto retry;
    }
     accounts[x]->val -= amount;
     accounts[y]->val += amount;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
}
```

Trick II to prevent deadlock

Observation

- A deadlock occurs only if concurrent threads try to acquire locks in different order

Trick

- Each thread acquires lock in the same order

Trick II to prevent deadlock

Each thread acquires lock in the same order

```
void transfer(int x, int y, int amount)
{
    if(x < y) {
        pthread_mutex_lock(&mus[x]);
        pthread_mutex_lock(&mus[y]);
    } else {
        pthread_mutex_lock(&mus[y]);
        pthread_mutex_lock(&mus[x]);
    }
    accounts[x]->val -= amount;
     accounts[y]->val += amount;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
}
```