

Concurrency – Locking

Jinyang Li

based on Tiger Wang's slides

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++



```
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
```

Thread 1 

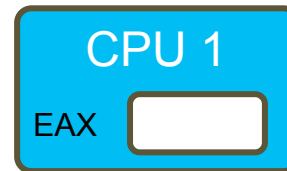
global: 0

Thread 2 

global++



global++



Time




mov 0x20072d(%rip), %eax

Example 1

global++

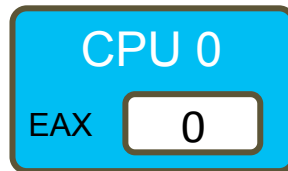
```
mov 0x20072d(%rip),%eax // load global into %eax
add $0x1,%eax           // update %eax by 1
mov %eax,0x200724(%rip) // restore global with %eax
```

Thread 1 

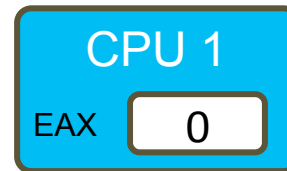
global: 0

Thread 2 

global++



global++



Time



mov 0x20072d(%rip), %eax

mov 0x20072d(%rip), %eax

Example 1

global++

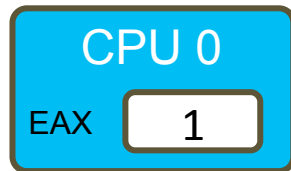
```
mov 0x20072d(%rip),%eax // load global into %eax
add $0x1,%eax           // update %eax by 1
mov %eax,0x200724(%rip) // restore global with %eax
```

Thread 1 

global: 0

Thread 2 

global++



global++



Time



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

```
add $0x1,%eax
```

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

Example 1

global++

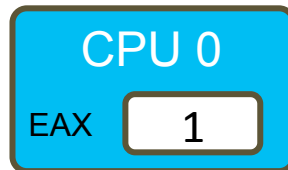
```
mov 0x20072d(%rip),%eax // load global into %eax
add $0x1,%eax           // update %eax by 1
mov %eax,0x200724(%rip) // restore global with %eax
```

Thread 1 

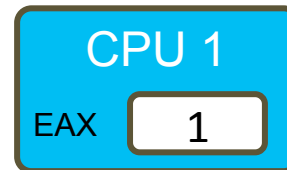
global: 0

Thread 2 

global++



global++



Time



mov 0x20072d(%rip), %eax

add \$0x1,%eax

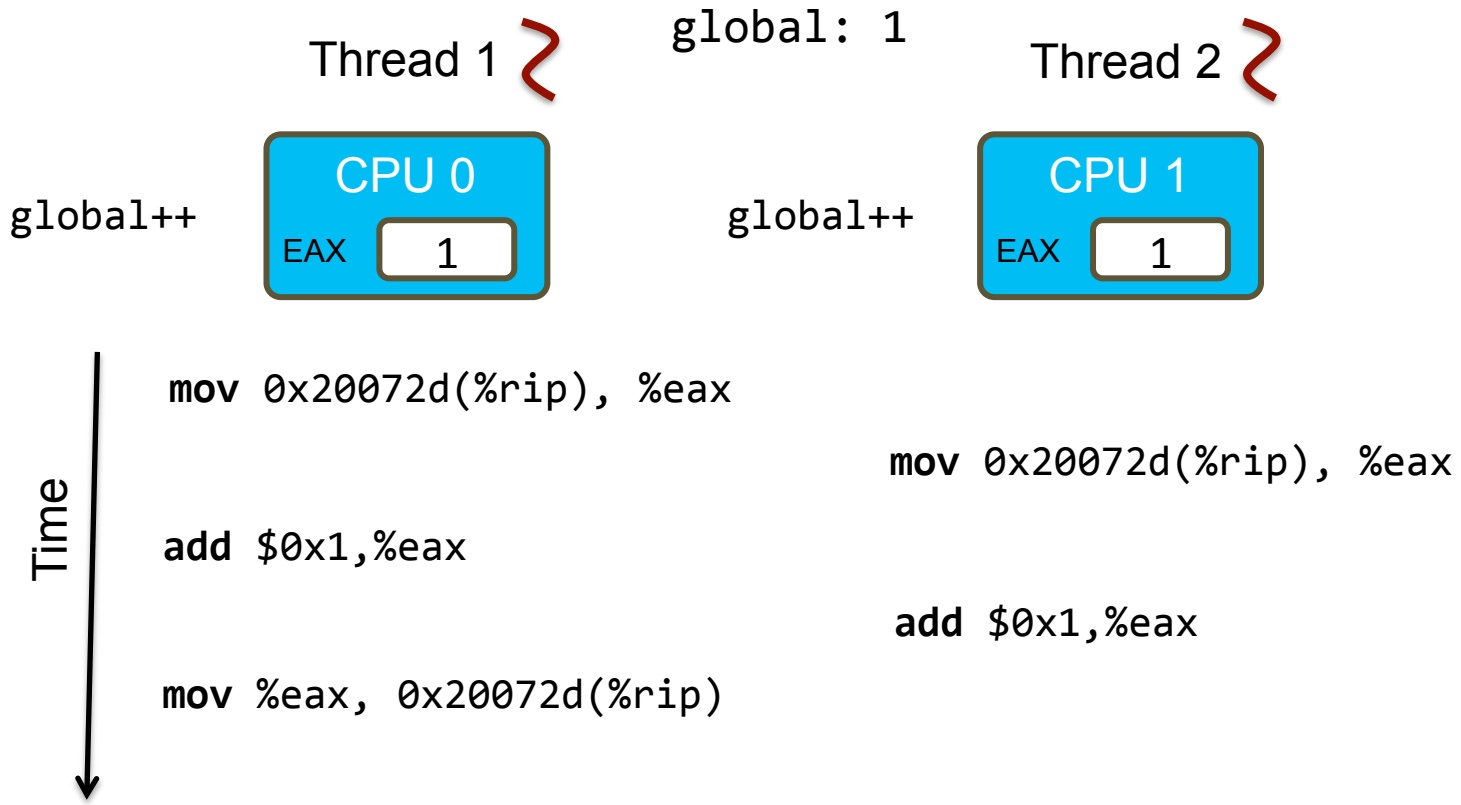
mov 0x20072d(%rip), %eax

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
```



Example 1

global++

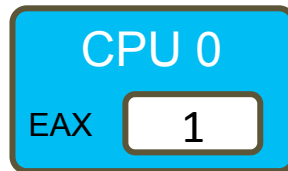
```
mov 0x20072d(%rip),%eax // load global into %eax
add $0x1,%eax           // update %eax by 1
mov %eax,0x200724(%rip) // restore global with %eax
```

Thread 1 

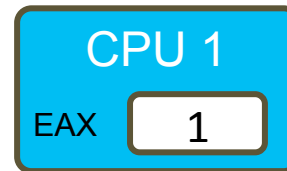
global: 1

Thread 2 

global++



global++



Time



mov 0x20072d(%rip), %eax

add \$0x1,%eax

mov %eax, 0x20072d(%rip)

mov 0x20072d(%rip), %eax

add \$0x1,%eax

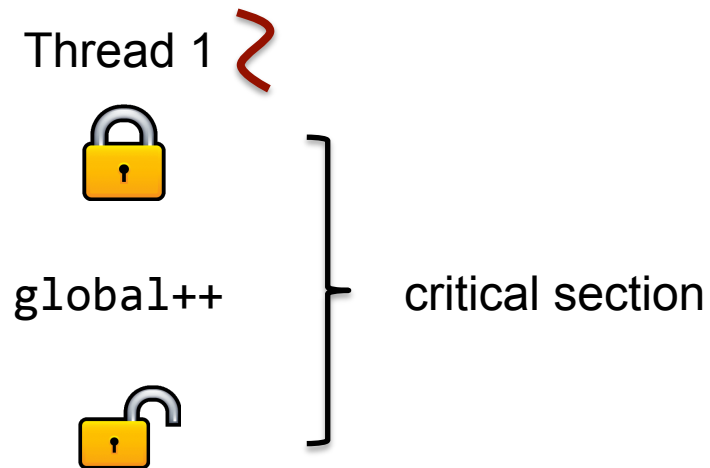
mov %eax, 0x20072d(%rip)

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



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;
...
int main() {
    ...
    pthread_mutex_init(&mu, NULL);
}
```

Lock/Mutex API in pthread lib

```
int pthread_mutex_lock(pthread_mutex_t *m)
```

- lock mutex m. If m is locked, caller blocks until m is unlocked
- return 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 mutex m
- return 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

Thread 1 

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

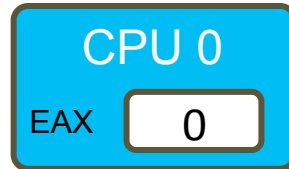
```
int global = 0;  
pthread_mutex_t mu;
```

Thread 2 

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

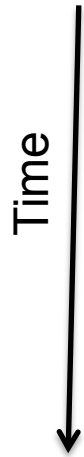
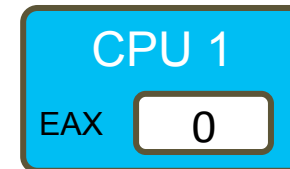
Example 1 with Lock

Thread 1 



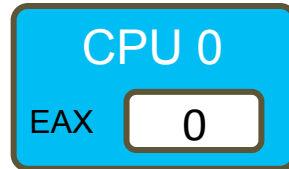
global: 0
mu: unlocked

Thread 2 



Example 1 with Lock

Thread 1 

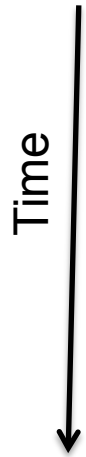


global: 0
mu: **locked**

Thread 2 

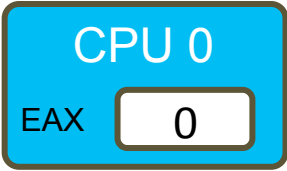


`pthread_mutex_lock(&mu);`



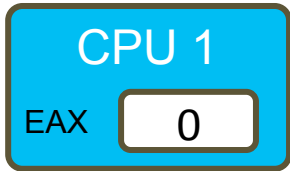
Example 1 with Lock

Thread 1 



global: 0
mu: **locked**

Thread 2 



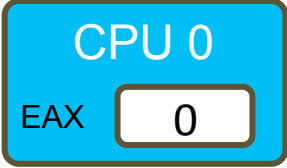
```
pthread_mutex_lock(&mu);
```

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



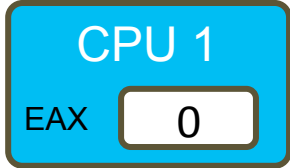
Example 1 with Lock

Thread 1 



global: 0
mu: **locked**

Thread 2 



```
pthread_mutex_lock(&mu);  
mov 0x20072d(%rip), %eax
```

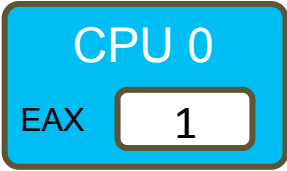
```
pthread_mutex_lock(&mu);
```

block and wait 



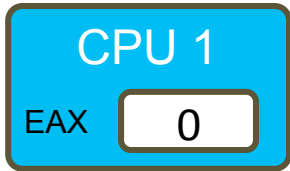
Example 1 with Lock


Thread 1 




global: 0
mu: **locked**

Thread 2 



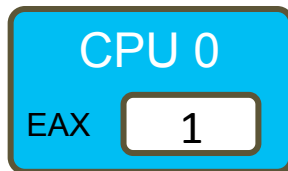
Time 

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

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

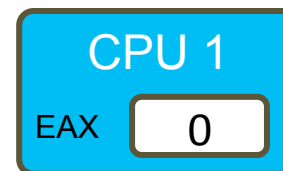
Example 1 with Lock

Thread 1 



global: 1
mu: **locked**

Thread 2 



`pthread_mutex_lock(&mu);`

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

`add $0x1,%eax`

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

`pthread_mutex_lock(&mu);`

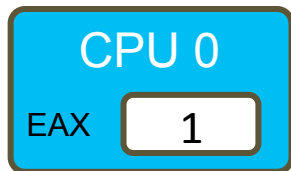
block and wait 

Time



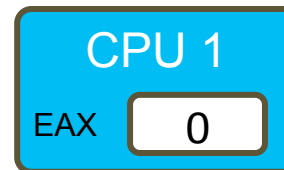
Example 1 with Lock

Thread 1 



global: 1
mu: **locked**

Thread 2 



`pthread_mutex_lock(&mu);`

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

`add $0x1,%eax`

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

} global++

`pthread_mutex_lock(&mu);`

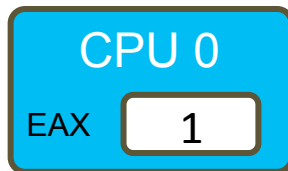
block and wait 

Time



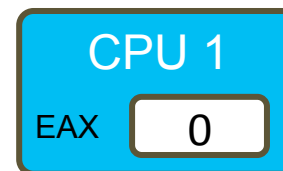
Example 1 with Lock

Thread 1 



global: 1
mu: unlocked

Thread 2 



`pthread_mutex_lock(&mu);`

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

`add $0x1,%eax`

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

} global++

`pthread_mutex_unlock(&mu);`

`pthread_mutex_lock(&mu);`

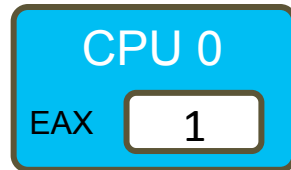
block and wait 

Time



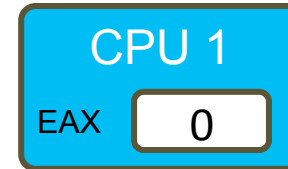
Example 1 with Lock

Thread 1 



global: 1
mu: **locked**

Thread 2 



`pthread_mutex_lock(&mu);`

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

`add $0x1,%eax`

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

`pthread_mutex_unlock(&mu);`

} global++

`pthread_mutex_lock(&mu);`

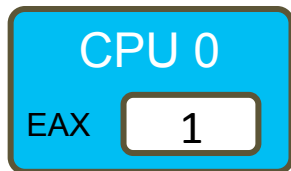
block and wait 

Time



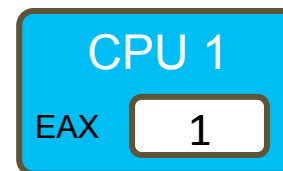
Example 1 with Lock

Thread 1 



global: 1
mu: **locked**

Thread 2 



`pthread_mutex_lock(&mu);`

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

`add $0x1,%eax`

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

`pthread_mutex_unlock(&mu);`

} global++

`pthread_mutex_lock(&mu);`

block and wait 

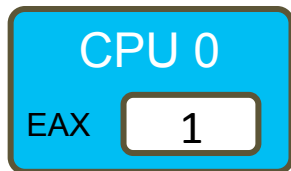
`mov 0x20072d(%rip), %eax`

Time



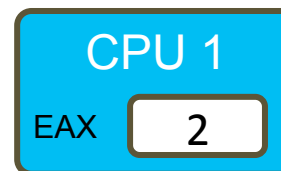
Example 1 with Lock

Thread 1 



global: 1
mu: **locked**

Thread 2 



Time
↓

`pthread_mutex_lock(&mu);`

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

`add $0x1,%eax`

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

`pthread_mutex_unlock(&mu);`

} global++

`pthread_mutex_lock(&mu);`

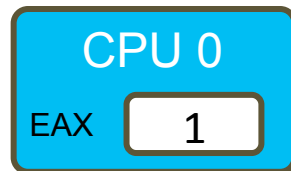
block and wait 

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

`add $0x1,%eax`

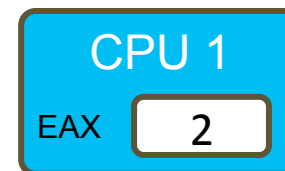
Example 1 with Lock

Thread 1 



global: 2
mu: **locked**

Thread 2 



Time
↓

pthread_mutex_lock(&mu);

mov 0x20072d(%rip), %eax

add \$0x1,%eax

mov %eax, 0x20072d(%rip)

} global++

pthread_mutex_unlock(&mu);

pthread_mutex_lock(&mu);

block and wait 


global++

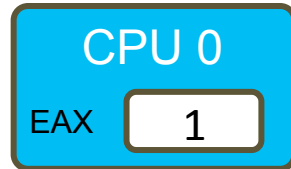
mov 0x20072d(%rip), %eax

add \$0x1,%eax

mov %eax, 0x20072d(%rip)

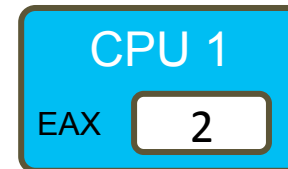
Example 1 with Lock

Thread 1 



global: 2
mu: **locked**

Thread 2 



Time
↓

`pthread_mutex_lock(&mu);`

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

`add $0x1,%eax`

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

`pthread_mutex_unlock(&mu);`

} global++

} global++

`pthread_mutex_lock(&mu);`

block and wait 

`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

```
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);
}
```

```
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);
    }
}
```

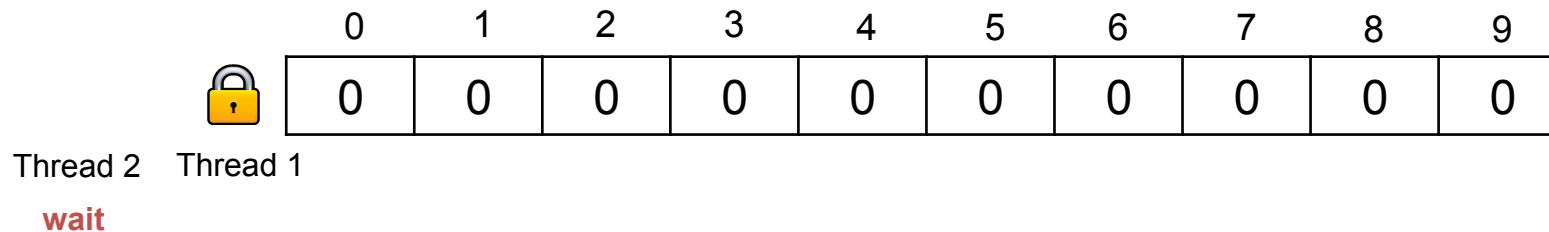
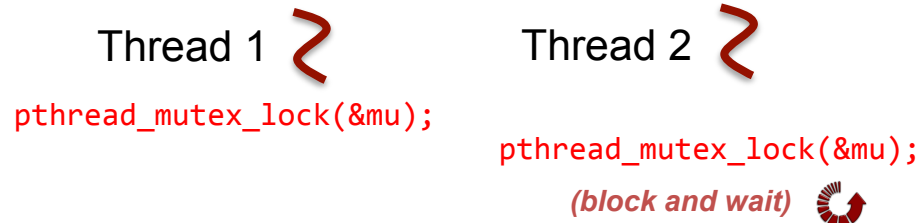
Which one is correct?

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




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);  
}
```

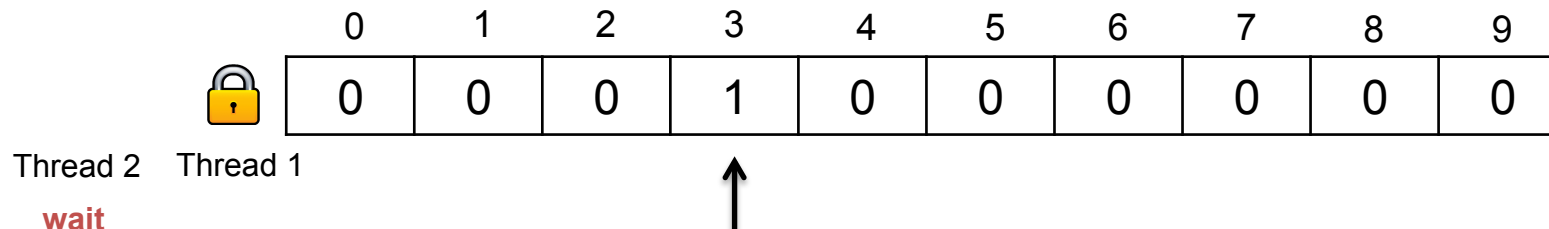
Both of them update elements 3 and 4

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);  
}
```

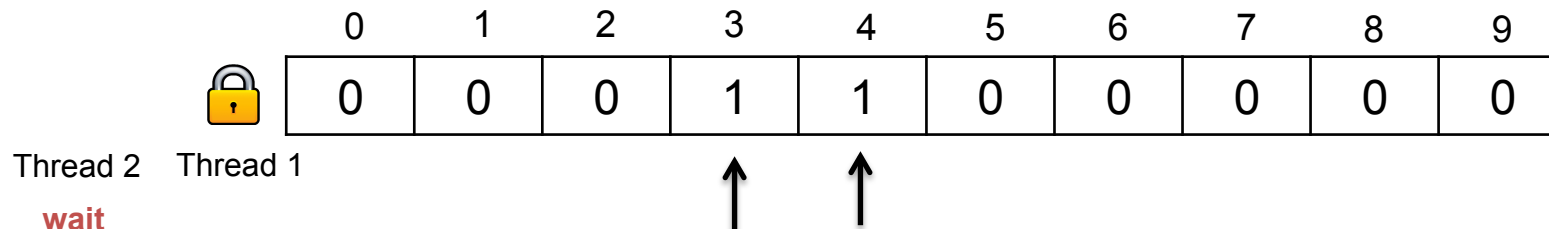
Both of them update elements 3 and 4

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);  
}
```

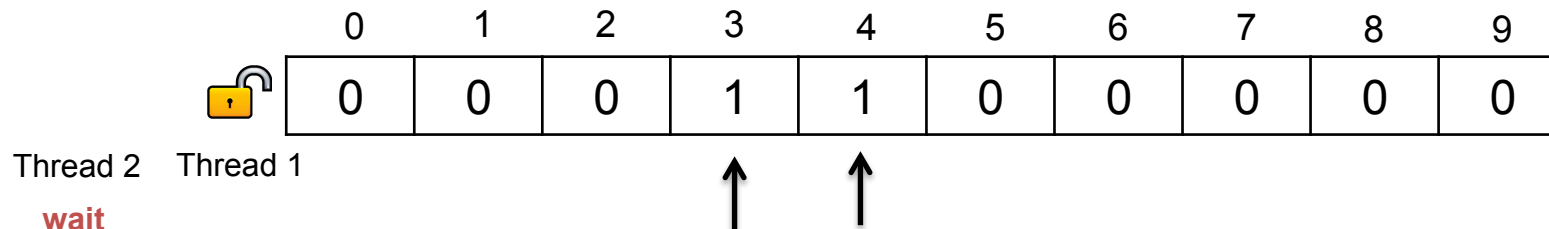
Both of them update elements 3 and 4

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);  
}
```

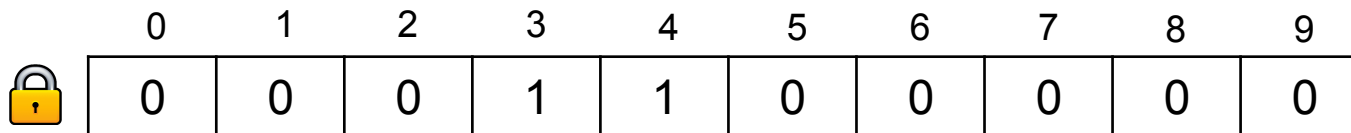
Both of them update elements 3 and 4

Thread 1 

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

Thread 2 

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




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);  
}
```

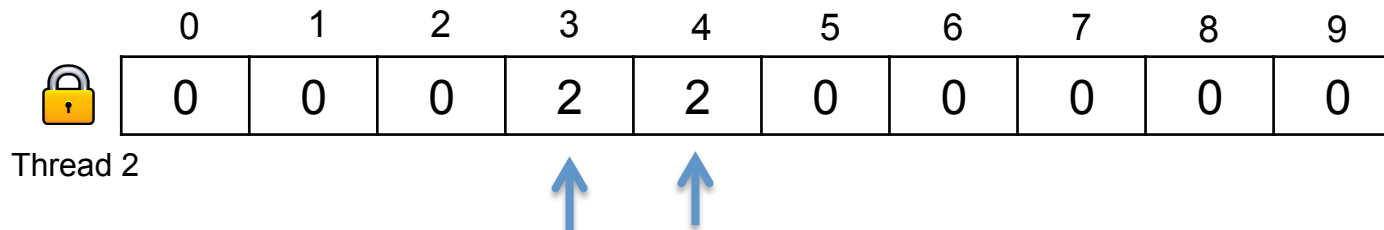
Both of them update elements 3 and 4

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]++;
```




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);  
}
```

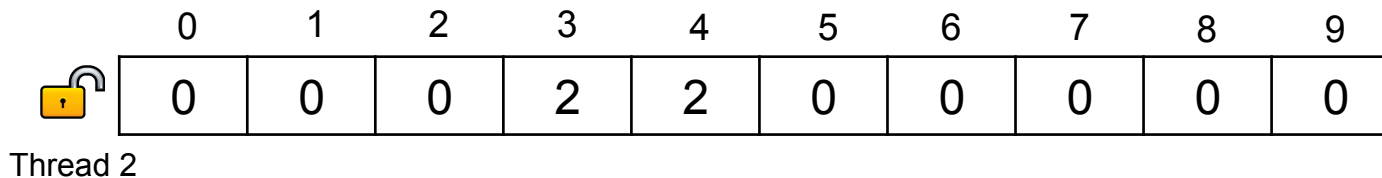
Both of them update elements 3 and 4

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);
```

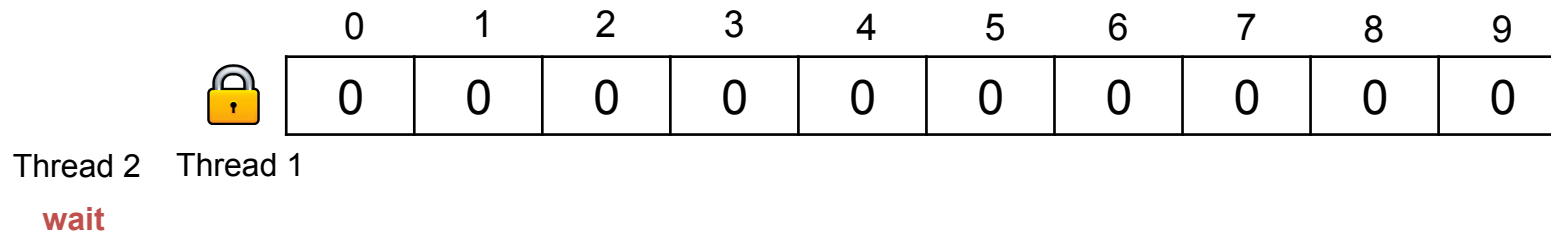
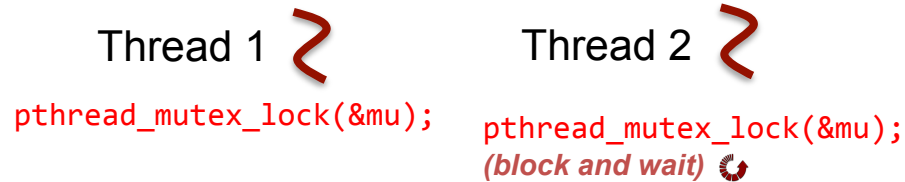


Example 2.2

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




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);  
    }  
}
```

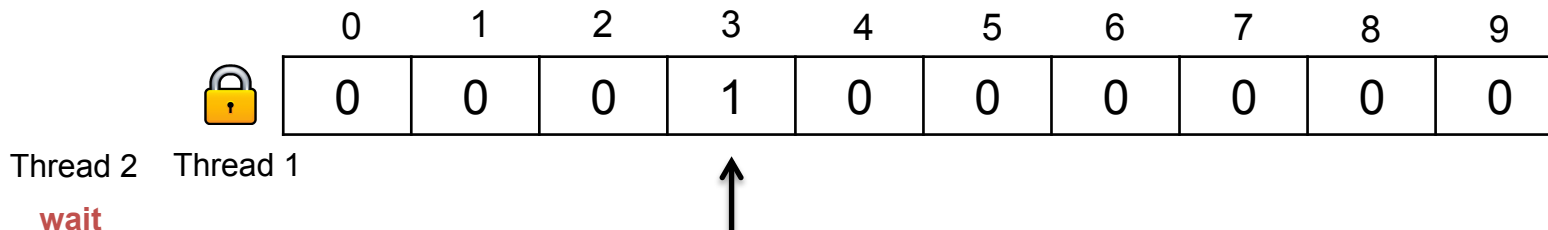
Both of them update elements 3 and 4

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);  
    }  
}
```

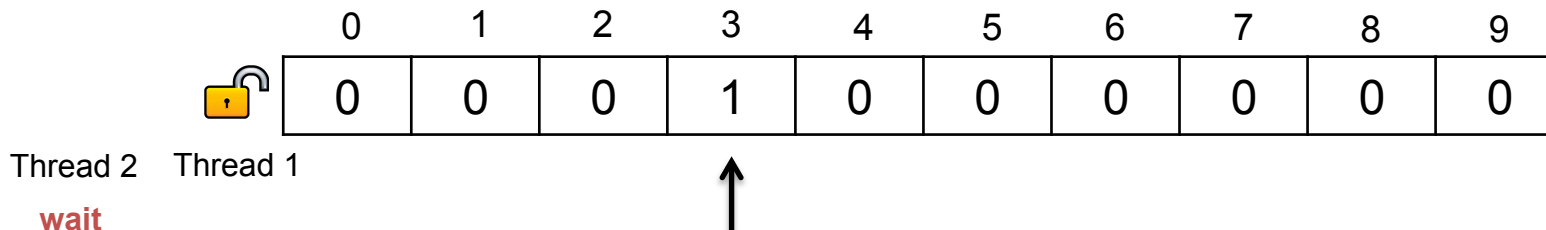
Both of them update elements 3 and 4

Thread 1 

Thread 2 

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

```
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);  
    }  
}
```

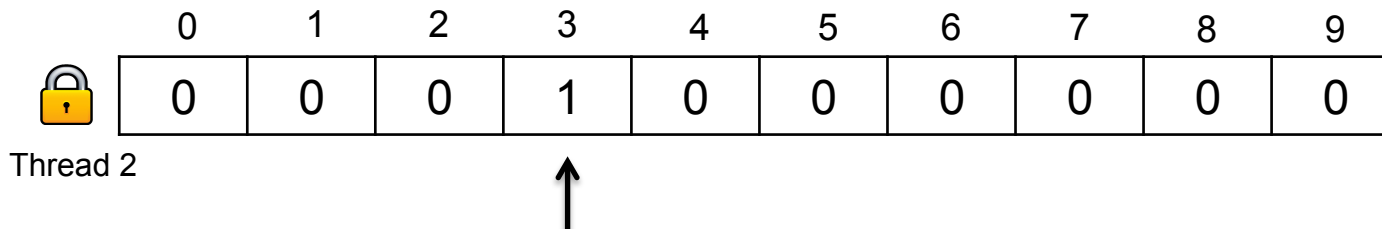
Both of them update elements 3 and 4

Thread 1 

Thread 2 

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

```
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);  
    }  
}
```

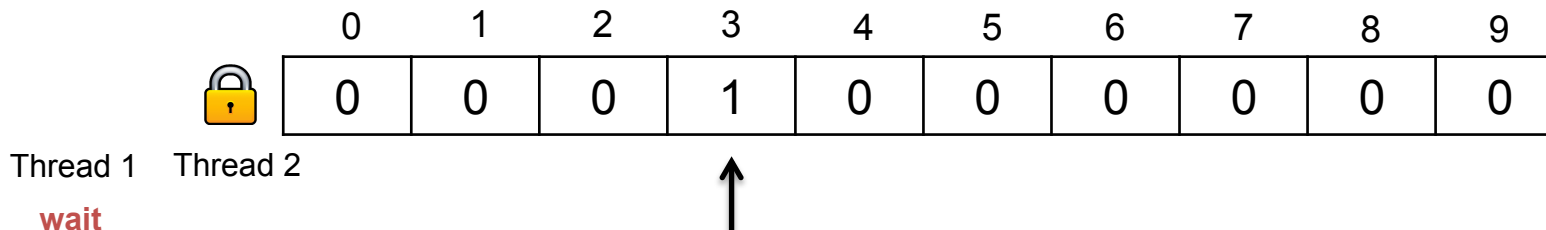
Both of them update elements 3 and 4

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) 
```





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);  
    }  
}
```

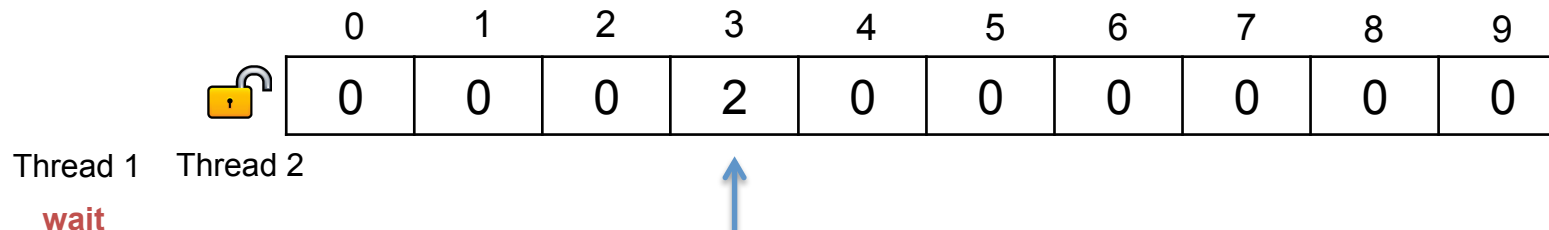
Both of them update elements 3 and 4

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);  
    }  
}
```

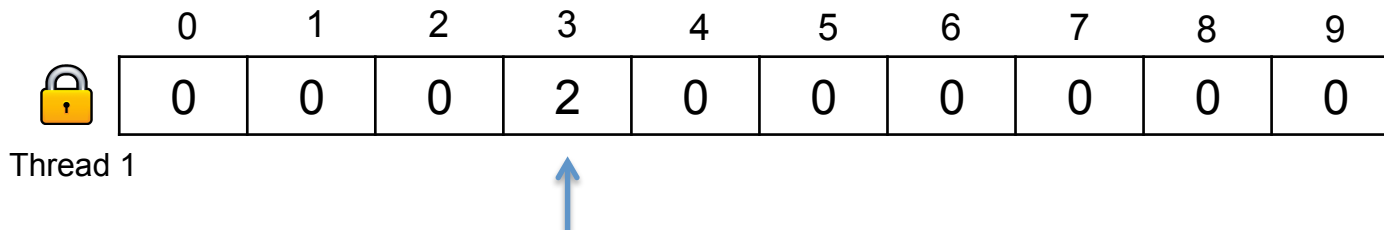
Both of them update elements 3 and 4

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);  
    }  
}
```

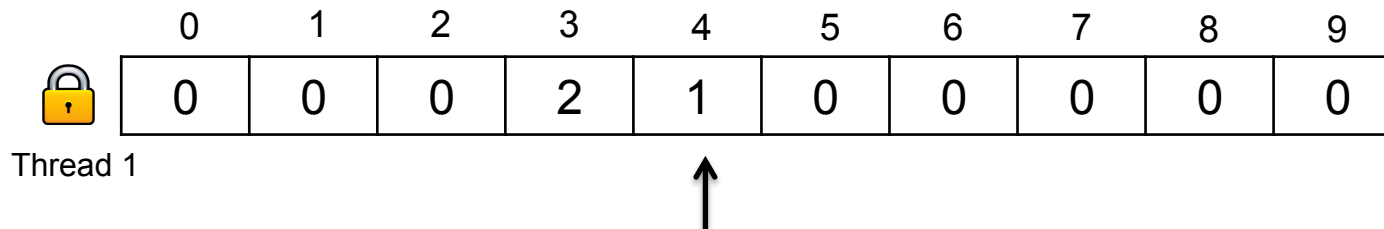
Both of them update elements 3 and 4

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);
```





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);  
    }  
}
```

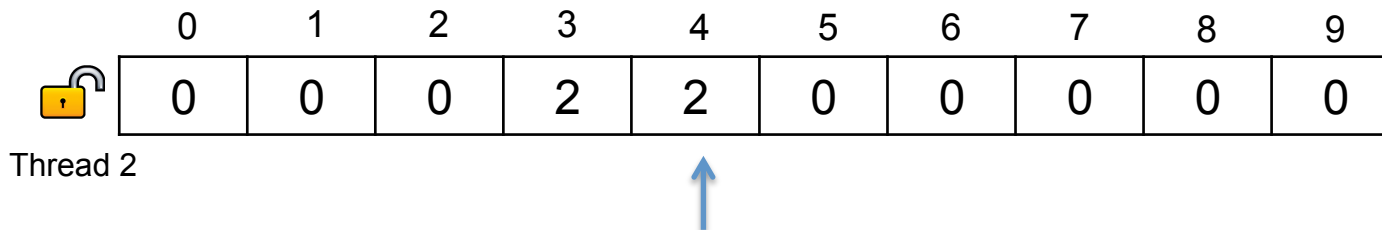
Both of them update elements 3 and 4

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);  
    }  
}
```

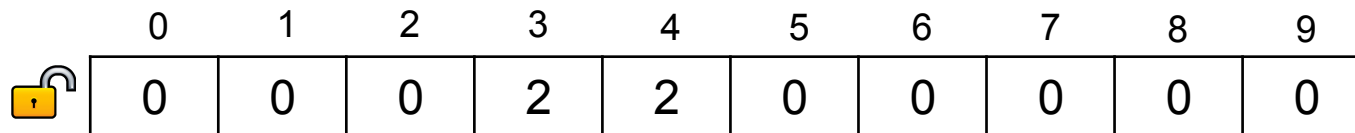
Both of them update elements 3 and 4

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);
```

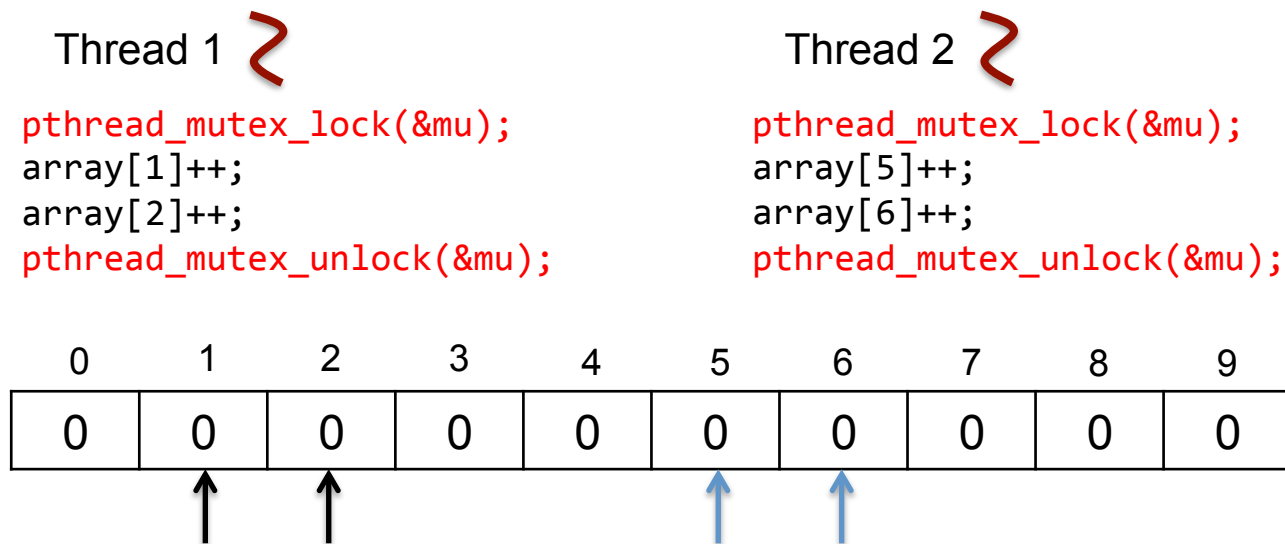


Thread 2

What is the problem?

Example 2.3

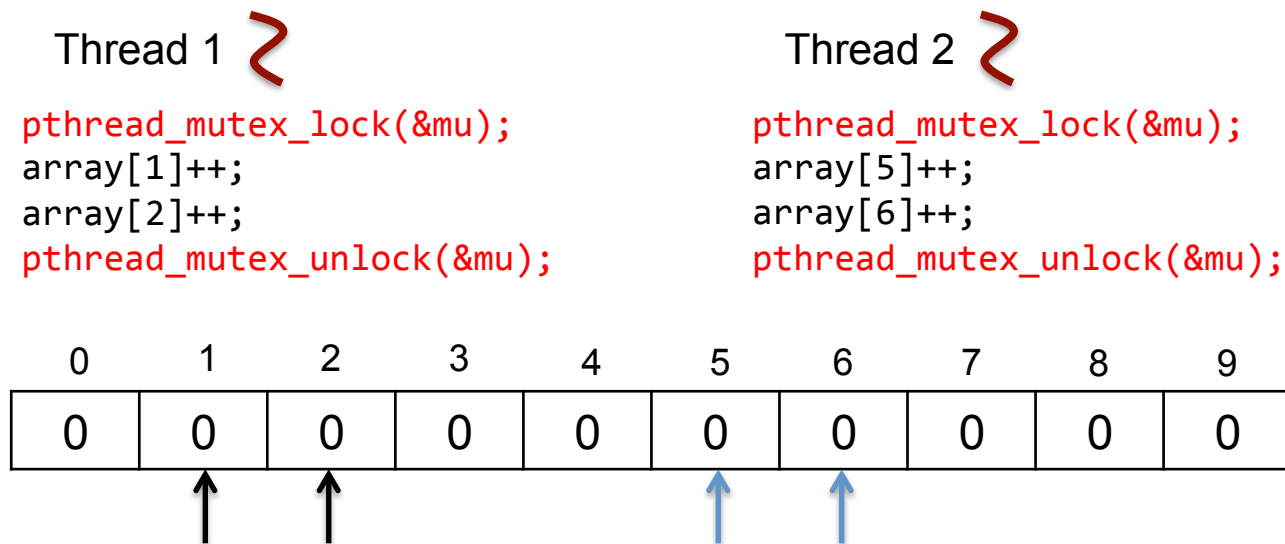
Each thread updates 2 random elements from a shared array



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

Problem: over-synchronization

Each thread updates 2 random elements from a shared array



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

How to improve it?

Lock granularity

Coarse granularity

- One big lock, associated with the entire array

Fine granularity

- Multiple locks, each 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

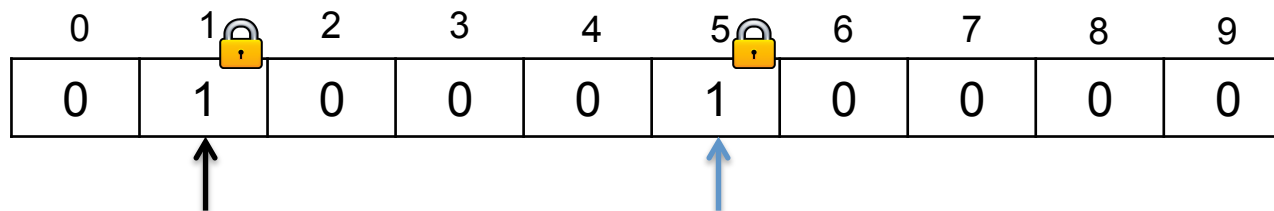
Each thread updates 2 random elements from a shared array

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

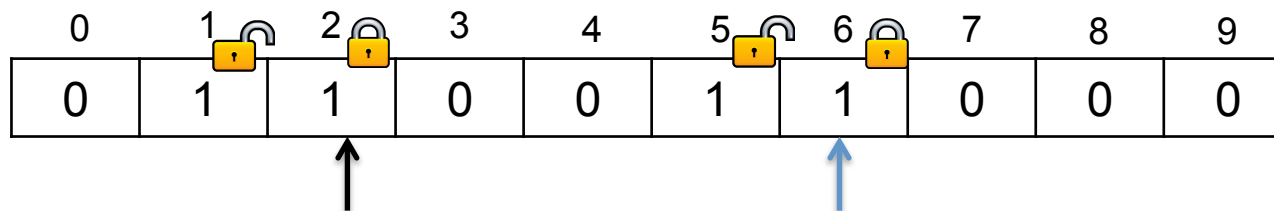
Each thread updates 2 random elements from a shared array

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

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

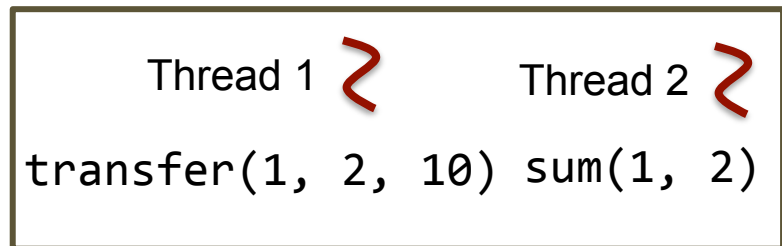
```
account *accounts[10];
```

```
//transfer monkey from account x to y  
void transfer(int x, int y, int amount)  
{  
    accounts[x]->val -= amount;  
    accounts[y]->val += amount;  
}
```

```
// read the total of account x and y  
int sum(int x, int y)  
{  
    return accounts[x]->val + accounts[y]->val;  
}
```

Each thread may invoke transfer or sum

No thread should observe the intermediate state of a transfer.



Example 3

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

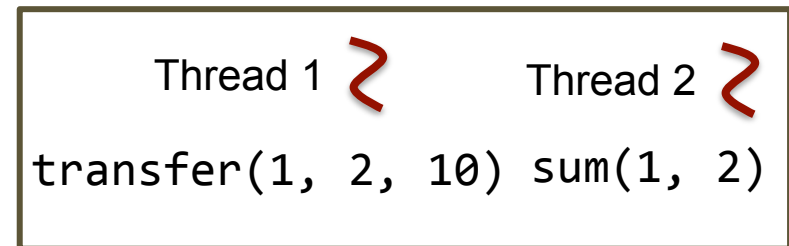
```
account *accounts[10];  
pthread_mutex_t mu;
```

```
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 or sum

No thread should observe the intermediate state of a transfer.



Example 3

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

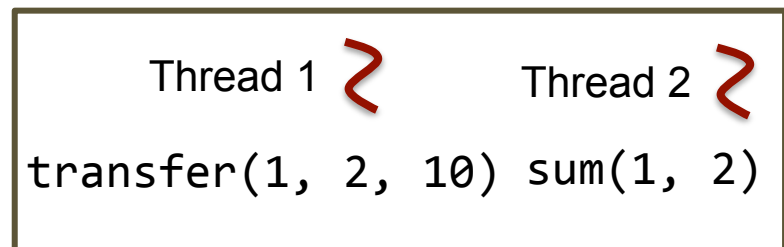
account *accounts[10];
pthread_mutex_t mu;

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 or sum

No thread should observe the intermediate state of a transfer.



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

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

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

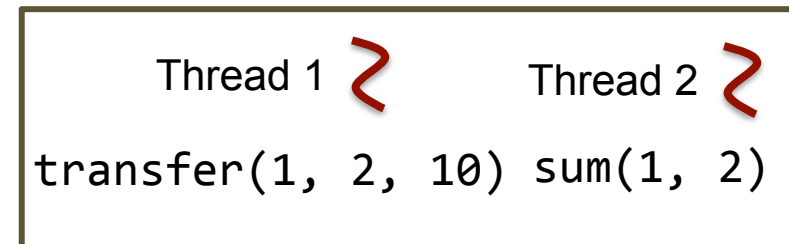
```
void transfer(int x, int y, int amount)
{
    pthread_mutex_lock(&mus[x]);
    accounts[x]->val -= amount;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    accounts[y]->val += amount;
    pthread_mutex_unlock(&mus[y]);
}
```

```
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 may invoke transfer or sum

No thread should observe intermediate state of a transfer.



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

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

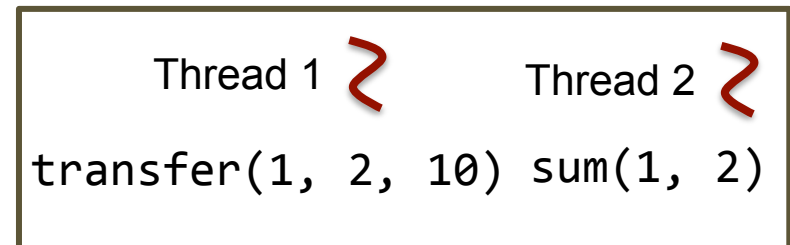
```
void transfer(int x, int y, int amount)
{
    pthread_mutex_lock(&mus[x]);
    accounts[x]->val -= amount;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    accounts[y]->val += amount;
    pthread_mutex_unlock(&mus[y]);
}
```

```
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 may invoke transfer or sum

No thread should observe intermediate state of a transfer.



Any problem?


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

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

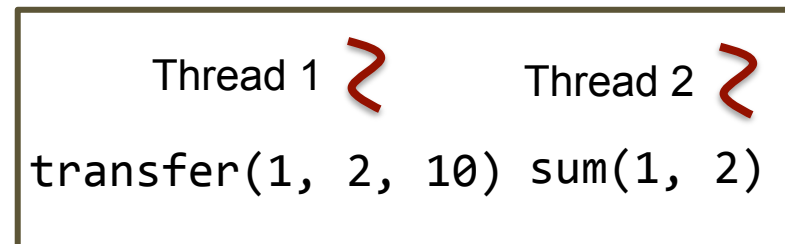
```
void transfer(int x, int y, int amount)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    accounts[x]->val -= amount;
    accounts[y]->val += amount;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
}
```

```
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;
}
```

Example 3

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

→ Still hold x's lock when access y.



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

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

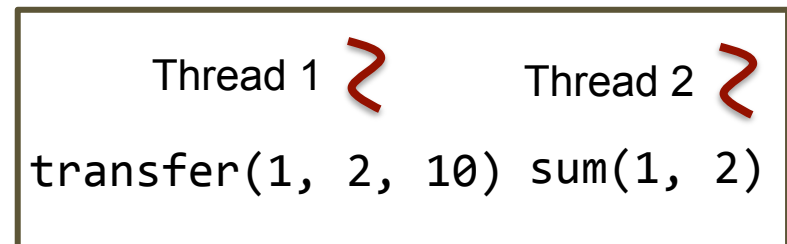
```
void transfer(int x, int y, int amount)
{
    pthread_mutex_lock(&mus[x]);
    pthread_mutex_lock(&mus[y]);
    accounts[x]->val -= amount;
    accounts[y]->val += amount;
    pthread_mutex_unlock(&mus[x]);
    pthread_mutex_unlock(&mus[y]);
}
```

```
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;
}
```

Example 3

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

→ Still hold x's lock when access y.



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)

Techniques to prevent deadlock

Observation

- A deadlock occurs if a thread who's holding one lock is blocked trying to grab another lock

Trick

- Use “trylock” to avoid thread being blocked.

Use trylock to avoid deadlock

- `int pthread_mutex_trylock(pthread_mutex_t *mu);`
 - If the mutex is locked, the call returns immediately.
 - Return value:
 - Zero: acquired the lock successfully;
 - Non-Zero: lock is held by others

Use trylock to avoid deadlock

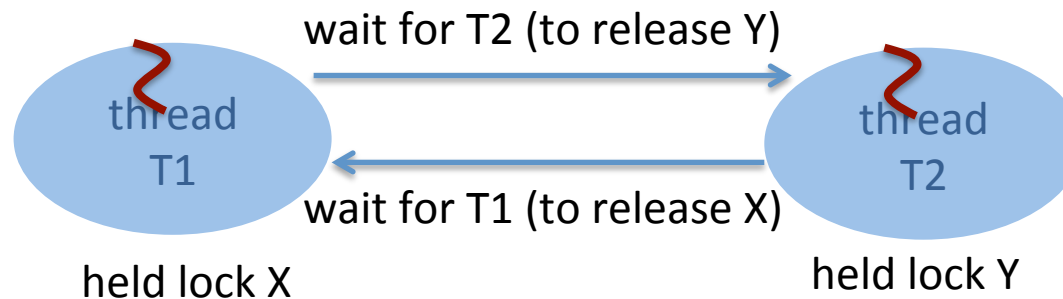
- `int pthread_mutex_trylock(pthread_mutex_t *mu);`
 - If the mutex is locked, the call returns immediately.
 - Return value:
 - Zero: acquired the lock successfully;
 - Non-Zero: lock is held by others

```
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]); ← must release held lock
            goto retry;                       if trylock is unsuccessful
        }
        accounts[x]->val -= amount;
        accounts[y]->val += amount;
        pthread_mutex_unlock(&mus[x]);
        pthread_mutex_unlock(&mus[y]);
}
```

Technique 2: Lock ordering

Observation

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



Technique:

- 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]);
}
```