CSCI-UA.0201

Computer Systems Organization

Concurrency – Synchronization and Locking

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

Example - bigloop

Parallelize bigloop into two threads

```
#define LEN 1000000000

long bigloop(int *arr) {
  long r = 0;
  for(int i = 0; i < LEN; i++)
    r += arr[i];
  return r;
}

int main()
{
  int *arr = malloc(LEN * sizeof(int));
  ...
  long r = bigloop(arr);
  ...
}
```

Example - bigloop

```
void* loop_thr1(void *arg) {
  long *r = malloc(sizeof(long));
  int *arr = (int *) arg;
  for (int i = 0; i < LEN/2; i++)
    (*r) += arr[i];
  return (void *) r;
}
```

```
void* loop_thr2(void *arg) {
  long *r = malloc(sizeof(long));
  int *arr = (int *) arg;
  for (int i = LEN/2; I < LEN; i++)
    (*r) += arr[i];
  return (void *) r;
}
```

```
int main() {
  int *arr = malloc(LEN * sizeof(int));
  ...
  pthread_t tid1, tid2;
  pthread_create(&tid, NULL, &loop_thr1, (void *)arr);
  pthread_create(&tid, NULL, &loop_thr2, (void *)arr);
  long *res1, *res2;
  pthread_join(tid, &res1);
  pthread_join(tid, &res2);
  printf("result is %ld\n", (*res1) + (*res2));
  free(res1); free(res2);
}
```

Example - bigloop

```
void* loop_thr1(void *arg) {
  long *r = malloc(sizeof(long));
  int *arr = (int *) arg;
  for (int i = 0; i < LEN/2; i++)
    (*r) += arr[i];
  return (void *) r;
}
```

```
void* loop_thr2(void *arg) {
  long *r = malloc(sizeof(long));
  int *arr = (int *) arg;
  for (int i = LEN/2; I < LEN; i++)
    (*r) += arr[i];
  return (void *) r;
}
```

```
int main() {
  int *arr = malloc(LEN * sizeof(int));
  ...
  pthread_t tid1, tid2;
  pthread_create(&tid, NULL, &loop_thr1, (void *)arr);
  pthread_create(&tid, NULL, &loop_thr2, (void *)arr);
  long *res1, *res2;
  pthread_join(tid, &res1);
  pthread_join(tid, &res2);
  printf(“result is %ld\n”, (*res1) + (*res2));
  free(res1); free(res2);
}
```

Can we merge loop_thr1 with loop_thr2?

Example - bigloop

```c
typedef struct {
  int *arr;
  int len;
} loop_info;

int main() {
  int *arr = malloc(LEN * sizeof(int));
  ...
  pthread_t tids[2];
  for (int i = 0; i < 2; i++) {
    loop_info *info = (loop_info *)malloc(sizeof(loop_info));
    info->arr = arr + i * LEN/2;
    info->len = LEN/2;
    pthread_create(&tids[i], NULL, &loop, (void *)info);
  }
  for (int i = 0; i < 2; i++) {
    long *res;
    pthread_join(tids[i], &res);
    result += (*res); free(res);
  }
  free arr;
}
```

```c
void* loop(void *arg) {
  loop_info *info = (loop_info *)arg;
  long *r = malloc(sizeof(long));
  for (int i = 0; i < info->len; i++)
    (*r) += info->arr[i];
  free(arg);
  return (void *)r;
}
```

Synchronization and Locking

Example 1 – Mutual Exclusion

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 – Mutual Exclusion

global++

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

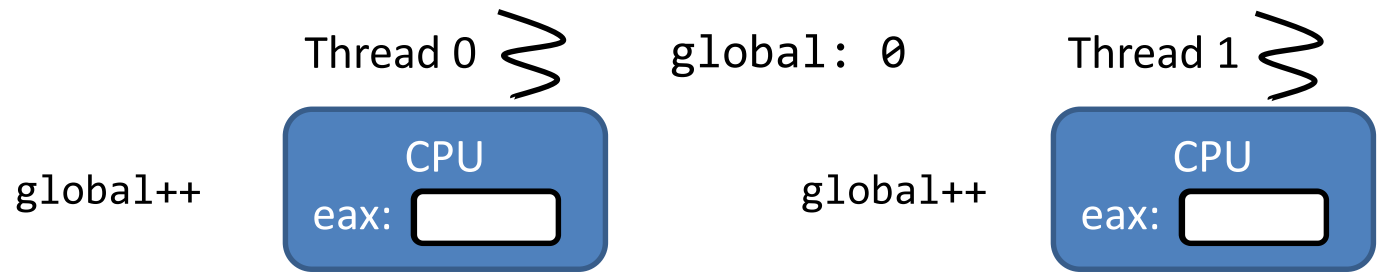
Thread 0   `global: 0`   Thread 1

`global++`   `global++`

Example 1 – Mutual Exclusion

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

Example 1 – Mutual Exclusion

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
```

Example 1 – Mutual Exclusion

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 0

global: 0

Thread 1

global++

CPU

eax: 0

global++

CPU

eax: 0

Time

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

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

Example 1 – Mutual Exclusion

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 0 | global: 0 | Thread 1

global++ — CPU eax: 1

global++ — CPU eax: 0

Time ↓

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

Example 1 – Mutual Exclusion

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 0 | global: 0 | Thread 1

global++ — CPU eax: 1

global++ — CPU eax: 1

Time ↓

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

Example 1 – Mutual Exclusion

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 0 | global: 1 | Thread 1

global++ CPU eax: 1

global++ CPU eax: 1

Time

```
mov 0x20072d(%rip),%eax
                                    mov 0x20072d(%rip),%eax
add $0x1,%eax
                                    add $0x1,%eax
mov %eax,0x200724(%rip)
```

Example 1 – Mutual Exclusion

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 0 | global: 1 | Thread 1

global++ — CPU eax: 1

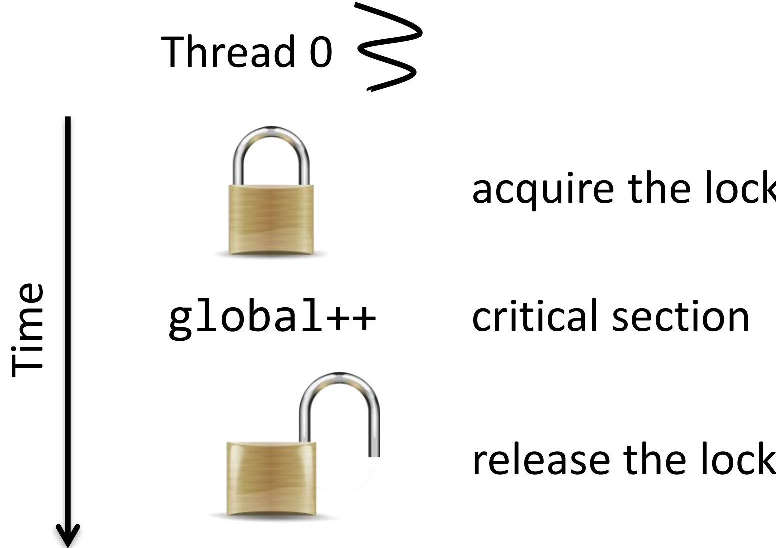
global++ — CPU eax: 1

Time

Thread 0	Thread 1
`mov 0x20072d(%rip),%eax`	
	`mov 0x20072d(%rip),%eax`
`add $0x1,%eax`	
	`add $0x1,%eax`
`mov %eax,0x200724(%rip)`	
	`mov %eax,0x200724(%rip)`

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);
}
```

Acquiring a Lock

- `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++;

}
```

Releasing a Lock

- `int pthread_mutex_unlock(pthread_mutex_t *m)`
  - unlocks 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 0 ≷   Thread 1

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

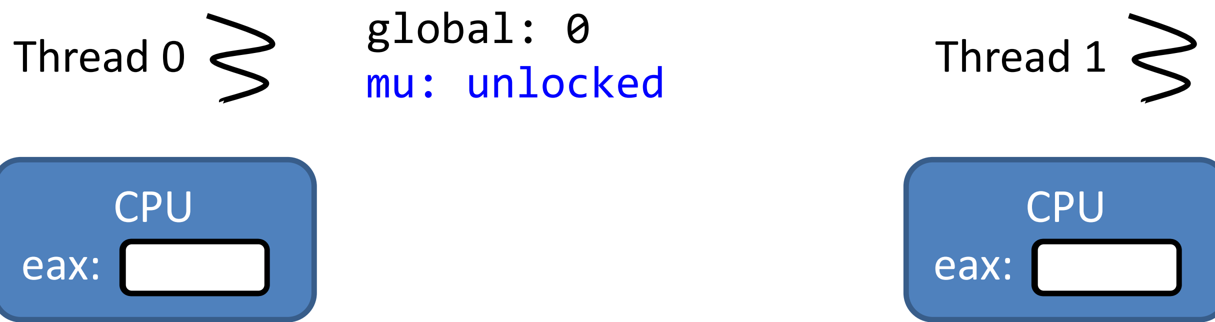
Thread 0:

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

Thread 1:

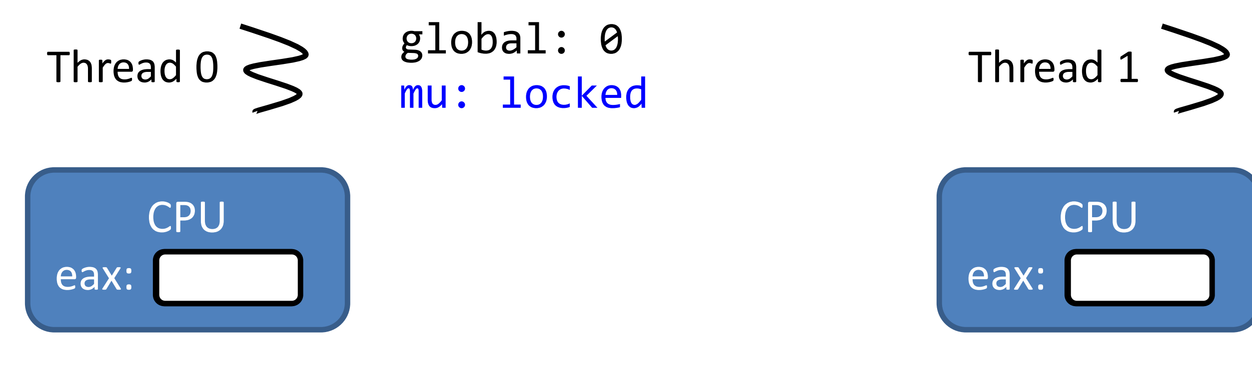
```
pthread_mutex_lock(&mu);
global++;
pthread_mutex_unlock(&mu);
```

Example 1 with Lock



Time

Example 1 with Lock

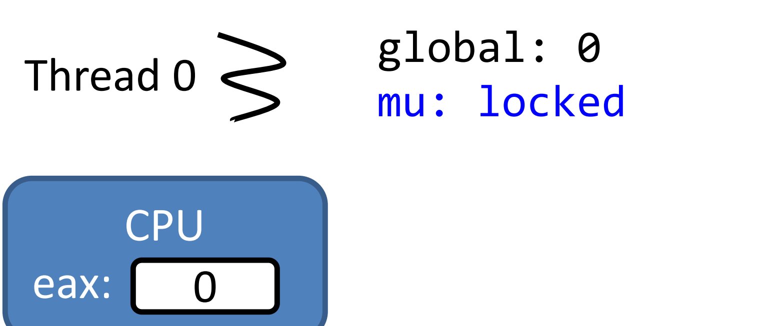


Time

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

Example 1 with Lock

Example 1 with Lock



Time

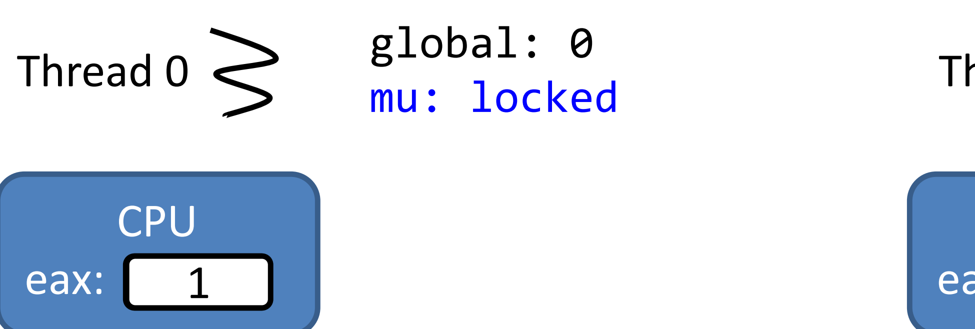
```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
```

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

block and wait

Z Z Z

Example 1 with Lock



Time

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

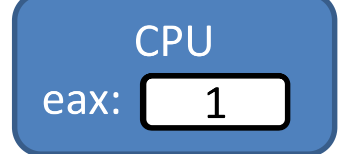
```
pthread_mutex_lock(&mu);
```

block and wait

Z Z Z

Example 1 with Lock

Thread 1



CPU
eax:

Time

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
```

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

block and wait

z z z

Example 1 with Lock

Thread 0

global: 1
mu: locked

Thread 1

CPU
eax: 1

CPU
eax:

Time

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
```

global++

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

block and wait

Z Z z

Example 1 with Lock

Thread 0

global: 1
mu: unlocked

Thread 1

CPU
eax: 1

CPU
eax:

Time

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
pthread_mutex_unlock(&mu);
```

global++

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

block and wait

z z z

Example 1 with Lock

Thread 0

global: 1
mu: locked

Thread 1

CPU
eax: 1

CPU
eax:

Time

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
pthread_mutex_unlock(&mu);
```

global++

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

Example 1 with Lock

Thread 0

global: 1
mu: locked

Thread 1

CPU
eax: 1

CPU
eax: 1

Time

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
pthread_mutex_unlock(&mu);
```

global++

```
pthread_mutex_lock(&mu);

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

Example 1 with Lock

Thread 0

global: 1
mu: locked

Thread 1

CPU
eax: 1

CPU
eax: 2

Time

Thread 0:

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
pthread_mutex_unlock(&mu);
```

global++

Thread 1:

```
pthread_mutex_lock(&mu);

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

Example 1 with Lock

Thread 0

global: 2
mu: locked

Thread 1

CPU
eax: 1

CPU
eax: 2

Time

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
pthread_mutex_unlock(&mu);
```

global++

```
pthread_mutex_lock(&mu);

mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
```

Example 1 with Lock

Thread 0

global: 2
mu: unlocked

Thread 1

CPU
eax: 1

CPU
eax: 2

Time

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
pthread_mutex_unlock(&mu);
```

global++

```
pthread_mutex_lock(&mu);
mov 0x20072d(%rip),%eax
add $0x1,%eax
mov %eax,0x200724(%rip)
pthread_mutex_unlock(&mu);
```

Example 2

- Each thread increments two randomly chosen 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 increments two randomly chosen 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);
}
```

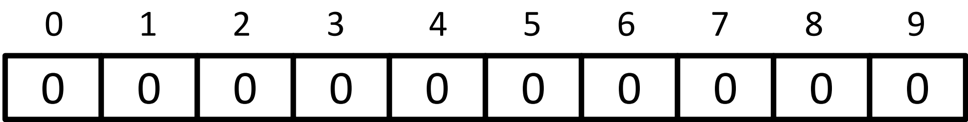
```
int array[10];
pthread_mutex_t mu;

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

```
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);
}
```

Both threads update elements 3 and 4



Example 2.1

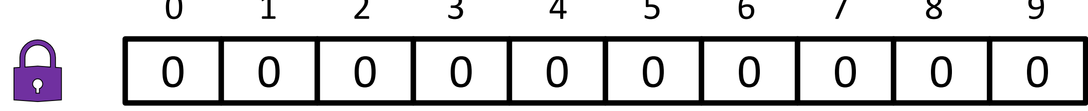
```
int array[10];
pthread_mutex_t mu;

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

Both threads update elements 3 and 4

Thread 0

Thread 1

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



Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

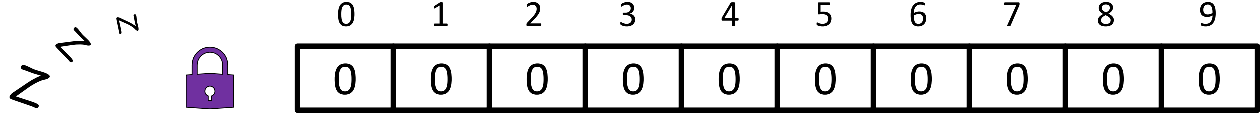
Thread 0

Thread 1

`pthread_mutex_lock(&mu);`

`pthread_mutex_lock(&mu);`

(block and wait)



Thread 1 Thread 0

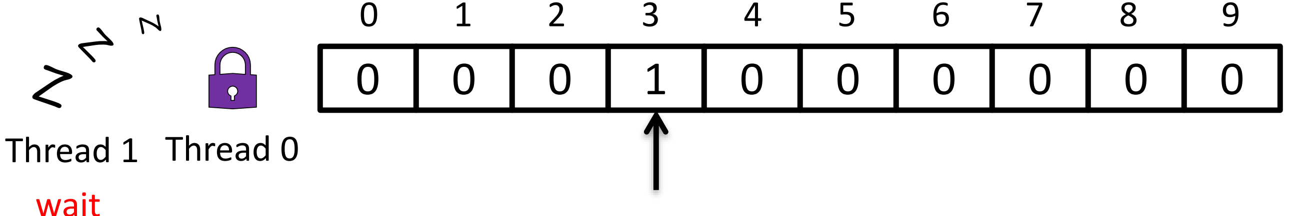
wait

Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	(block and wait)

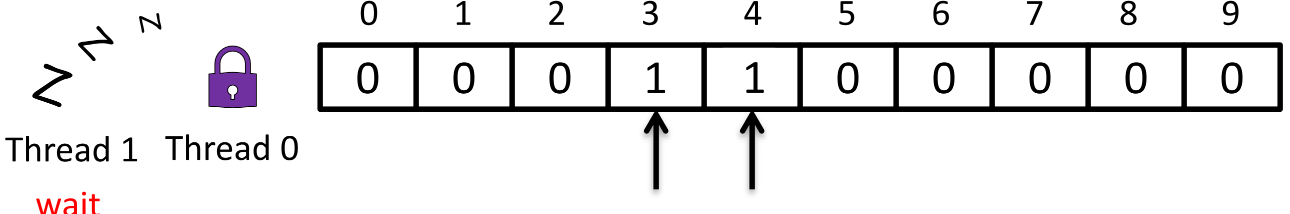


Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);` (block and wait)
`array[3]++;`	
`array[4]++;`	



Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

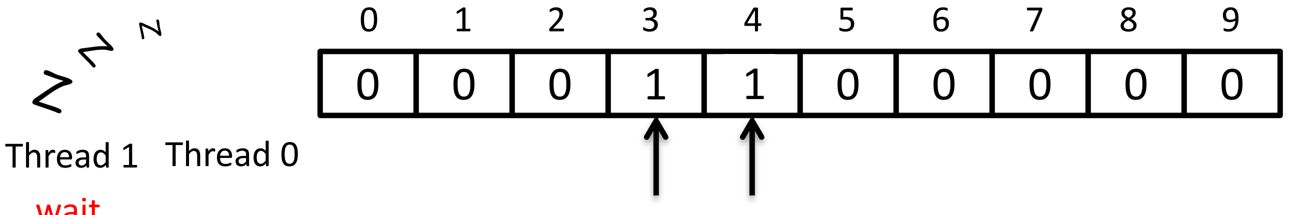
Thread 0

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

Thread 1

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

(block and wait)



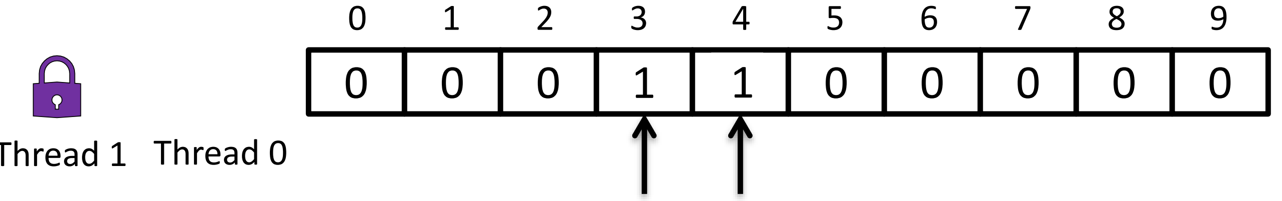
Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

Thread 0 | Thread 1

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

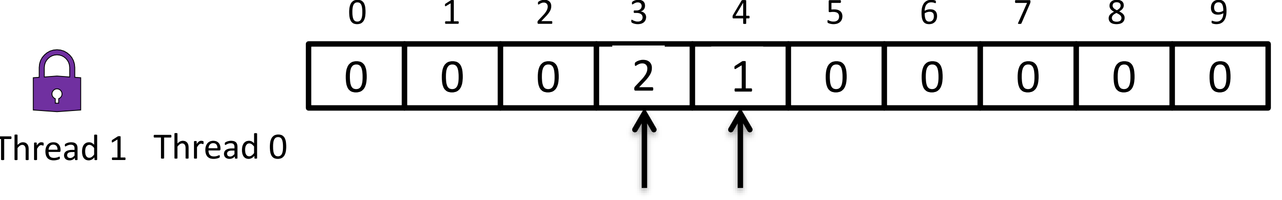


Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`array[4]++;`	
`pthread_mutex_unlock(&mu);`	
	`array[3]++;`

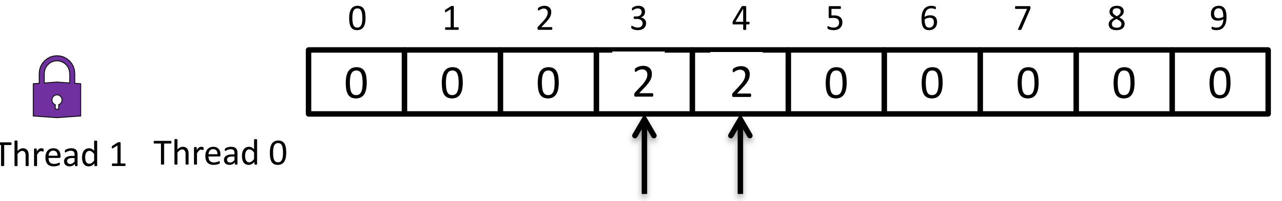


Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`array[4]++;`	
`pthread_mutex_unlock(&mu);`	
	`array[3]++;`
	`array[4]++;`

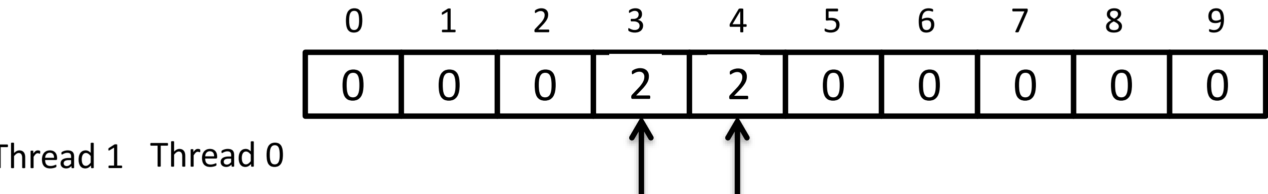


Example 2.1

```
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);
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`array[4]++;`	
`pthread_mutex_unlock(&mu);`	
	`array[3]++;`
	`array[4]++;`
	`pthread_mutex_unlock(&mu);`

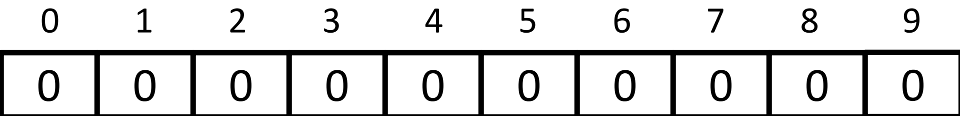


Example 2.2

```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);
  }
}
```

Both threads update elements 3 and 4



Thread 1 Thread 0

Example 2.2

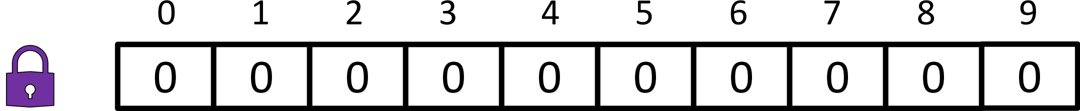
```
int array[10];
pthread_mutex_t mu;

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

Both threads update elements 3 and 4

Thread 0 Thread 1

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



Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

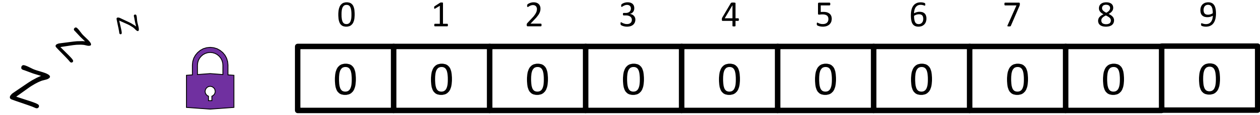
Thread 0

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

Thread 1

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

(block and wait)



Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0

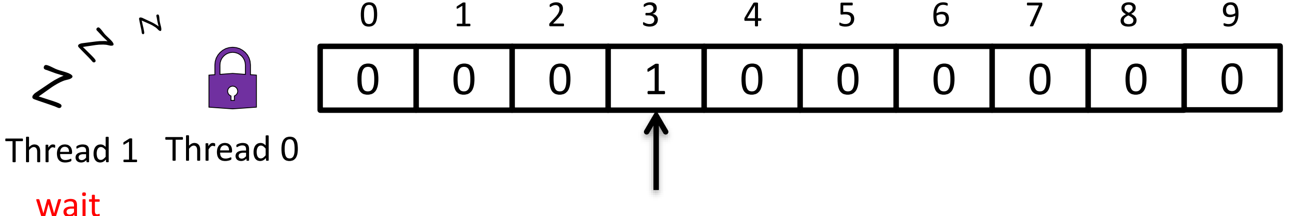
```
pthread_mutex_lock(&mu);

array[3]++;
```

Thread 1

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

(block and wait)



Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0

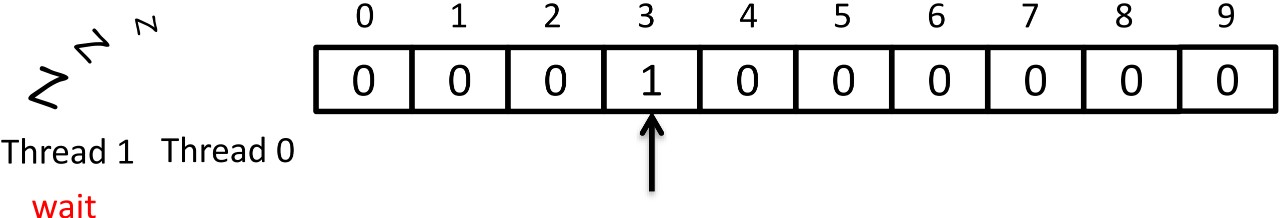
```
pthread_mutex_lock(&mu);

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

Thread 1

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

(block and wait)



Example 2.2

```
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);
  }
}
```

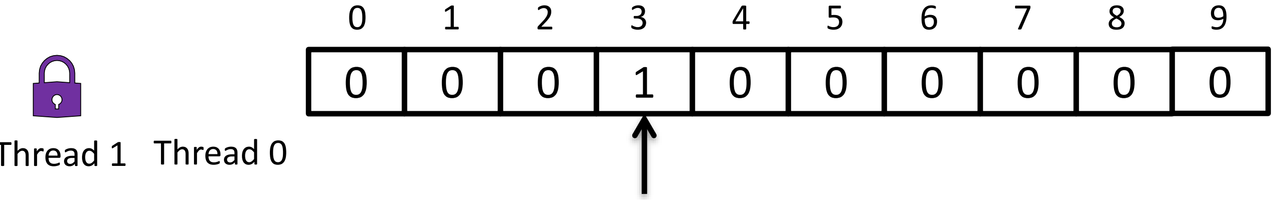
Both threads update elements 3 and 4

Thread 0

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

Thread 1

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



Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

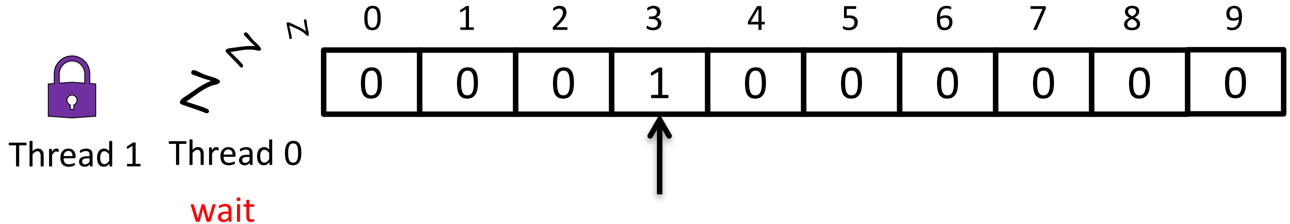
Thread 0

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

(block and wait)

Thread 1

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



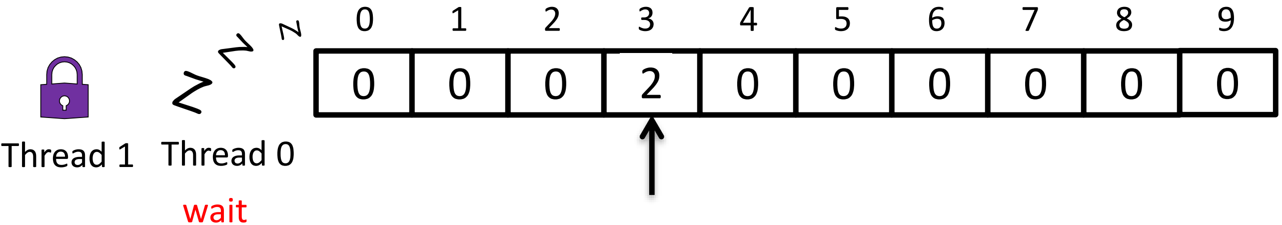
Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`pthread_mutex_unlock(&mu);`	
`pthread_mutex_lock(&mu);`	
	`array[3]++;`
(block and wait)	



Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0

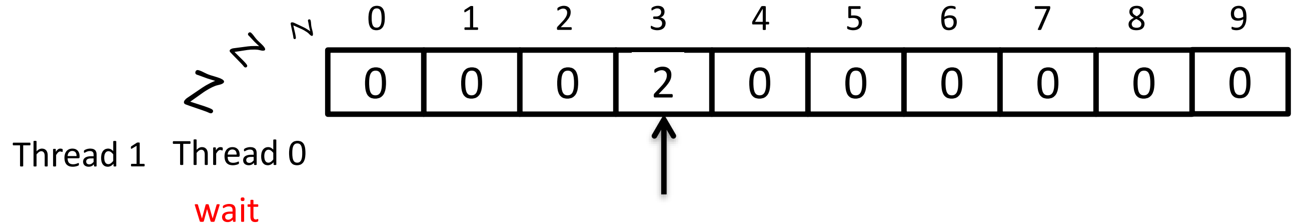
```
pthread_mutex_lock(&mu);
array[3]++;
pthread_mutex_unlock(&mu);
pthread_mutex_lock(&mu);
```

(block and wait)

Thread 1

```
pthread_mutex_lock(&mu);

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



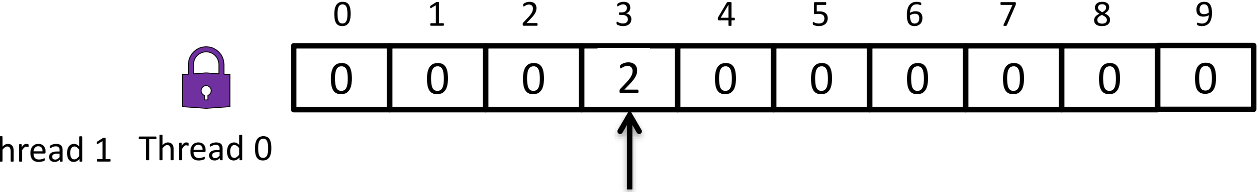
Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`pthread_mutex_unlock(&mu);`	
`pthread_mutex_lock(&mu);`	
	`array[3]++;`
	`pthread_mutex_unlock(&mu);`



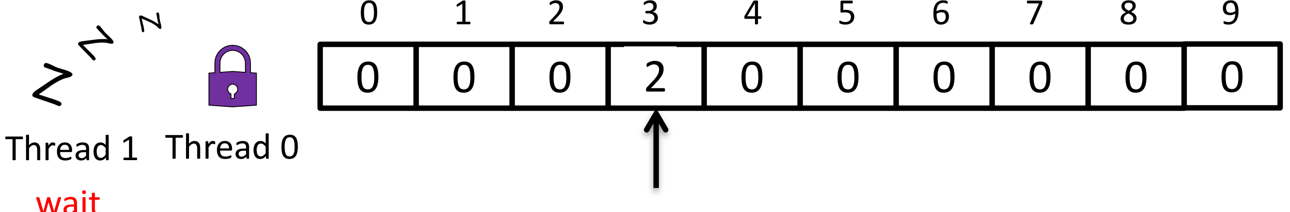
Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`pthread_mutex_unlock(&mu);`	
`pthread_mutex_lock(&mu);`	
	`array[3]++;`
	`pthread_mutex_unlock(&mu);`
	`pthread_mutex_lock(&mu);`
	(block and wait)



Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

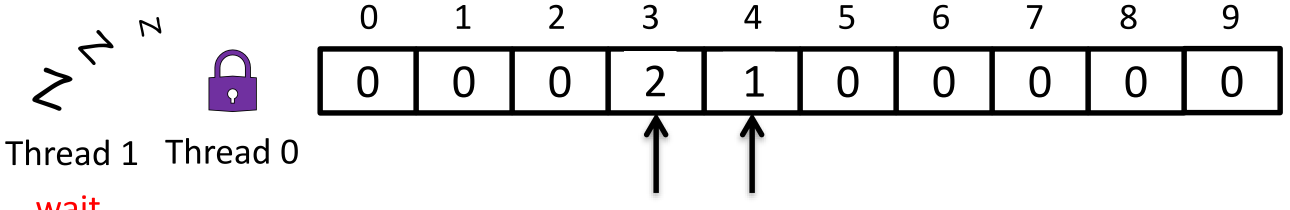
Thread 0

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

Thread 1

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

(block and wait)



Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0

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

Thread 1

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

(block and wait)

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

Z Z Z

Thread 1 Thread 0

wait

Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`pthread_mutex_unlock(&mu);`	
`pthread_mutex_lock(&mu);`	
	`array[3]++;`
	`pthread_mutex_unlock(&mu);`
	`pthread_mutex_lock(&mu);`
`array[4]++;`	
`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 Thread 0

Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`pthread_mutex_unlock(&mu);`	
`pthread_mutex_lock(&mu);`	
	`array[3]++;`
	`pthread_mutex_unlock(&mu);`
	`pthread_mutex_lock(&mu);`
`array[4]++;`	
`pthread_mutex_unlock(&mu);`	
	`array[4]++;`

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

Thread 1 Thread 0

Example 2.2

```
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);
  }
}
```

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`pthread_mutex_unlock(&mu);`	
`pthread_mutex_lock(&mu);`	
	`array[3]++;`
	`pthread_mutex_unlock(&mu);`
	`pthread_mutex_lock(&mu);`
`array[4]++;`	
`pthread_mutex_unlock(&mu);`	
	`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 1 Thread 0

Example 2.2

```
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);
  }
}
```

Do you see any problem?

Both threads update elements 3 and 4

Thread 0	Thread 1
`pthread_mutex_lock(&mu);`	
	`pthread_mutex_lock(&mu);`
`array[3]++;`	
`pthread_mutex_unlock(&mu);`	
`pthread_mutex_lock(&mu);`	
	`array[3]++;`
	`pthread_mutex_unlock(&mu);`
	`pthread_mutex_lock(&mu);`
`array[4]++;`	
`pthread_mutex_unlock(&mu);`	
	`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 1 Thread 0

Example 2.3

Thread 0

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

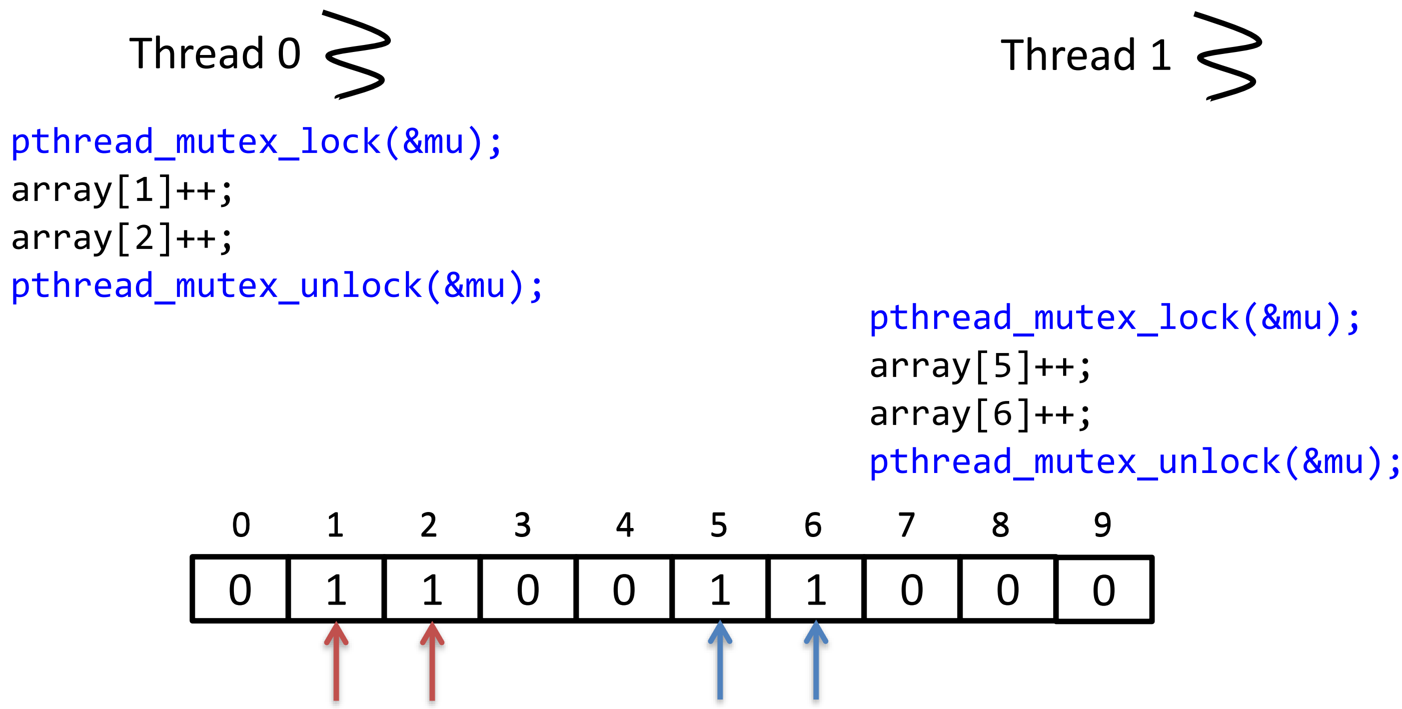
Thread 1

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

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

The executions of these two threads will always be serialized, even though they access different elements.

Problem: over-synchronization



The executions of these two threads will always be serialized, even though they access different elements.

Lock Granularity

- Coarse-grained locking
  – One big lock, associated with the entire array

- Fine-grained locking
  – Multiple locks, each associated with a single element

Example 2.4

- Each thread increments two randomly chosen 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 2.4

Thread 0

Thread 1

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

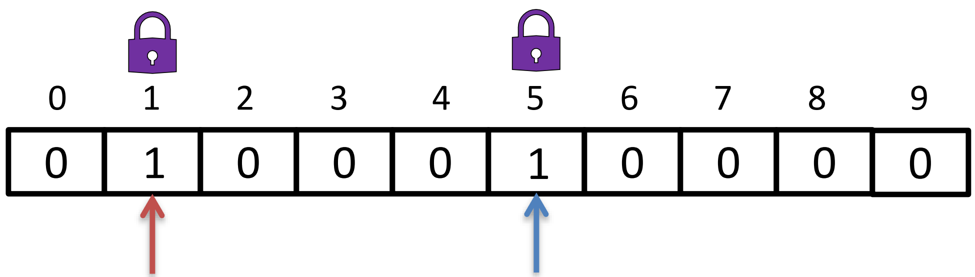
Example 2.4

Thread 0

```
pthread_mutex_lock(&locks[1]);
array[1]++;
```

Thread 1

```
pthread_mutex_lock(&locks[5]);
array[5]++;
```



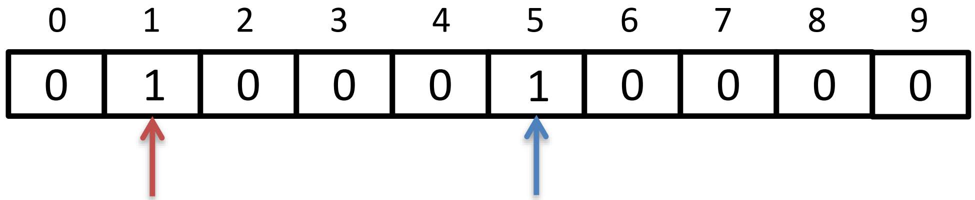
Example 2.4

Thread 0

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

Thread 1

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



Example 2.4

Thread 0

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

Thread 1

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

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

Example 2.4

Thread 0

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

Thread 1

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

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

Compare and Swap (CAS)

- ```
  bool __sync_bool_compare_and_swap (type *ptr,
                                     type oldval,
                                     type newval)
  ```
  - If the current value of `*ptr` is `oldval`, then write `newval` into `*ptr`
  - Return value:
    - Zero: if value of `*ptr` was different from `oldval`
    - Non-Zero: if comparison was successful and `newval` was written

- We'll write just `CAS(ptr, oldval, newval)` in the following

- CAS is implemented directly by hardware instructions
  ```

Example 2.5

```
int array[10];

void* thr(void*) {
  for (int i = 0; i < 2; i++) {
    int idx = random() % 10;
    int oldval;
    do {
      oldval = array[idx];
    } while (!CAS(&array[idx], oldval, oldval+1))
  }
}
```

Implementing a Spinlock with CAS

```
struct {
  int is_locked;
  pid_t thread_id;
} mutex_t;

int mutex_lock(mutex_t *mu) {
  int l;
  do {
    l = mu->is_locked;
  } while (l || !CAS(&mu->is_locked, 0, 1))
  mu->thread_id = gettid();   <- returns unique ID of current thread
  return 0;
}

int mutex_unlock(mutex_t *mu) {
  if (!mu->is_locked || mu->thread_id != gettid())
    return 1;
  mu->is_locked = 0;
  return 0;
}
```

Example 3

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

account *accounts[10];

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

// return the total of accounts x and y
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 money from account x to account y
void transfer(int x, int y, int amount) {
  accounts[x]->val -= amount;
  accounts[y]->val += amount;
}

// return the total of accounts 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.

Thread 1: transfer(1, 2, 10);

Thread 2: sum(1, 2)

Example 3

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

account *accounts[10];
pthread_mutex_t mu;

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

// return the total of accounts x and y
int sum(int x, int y) {
  pthread_mutex_lock(&mu);
  int res = accounts[x]->val + accounts[y]->val;
  pthread_mutex_unlock(&mu);
  returns res;
}
```

Each thread may invoke transfer or sum

No thread should observe the intermediate state of a transfer.

Thread 1: transfer(1, 2, 10);

Thread 2: sum(1, 2)

Example 3

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

account *accounts[10];
pthread_mutex_t mu;

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

// return the total of accounts x and y
int sum(int x, int y) {
  pthread_mutex_lock(&mu);
  int res = accounts[x]->val + accounts[y]->val;
  pthread_mutex_unlock(&mu);
  returns res;
}
```

Each thread may invoke transfer or sum

No thread should observe the intermediate state of a transfer.

Can we improve this implementation with fine-grained locking?

Thread 1: transfer(1, 2, 10);

Thread 2: sum(1, 2)

Example 3

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

// transfer money from account x to account y
void transfer(int x, int y, int amount) {
  pthread_mutex_lock(&locks[x]);
  accounts[x]->val -= amount;
  pthread_mutex_unlock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  accounts[y]->val += amount;
  pthread_mutex_unlock(&locks[y]);
}

// return the total of accounts x and y
int sum(int x, int y) {
  pthread_mutex_lock(&locks[x]);
  int vx = accounts[x]->val;
  pthread_mutex_unlock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  int vy = accounts[y]->val;
  pthread_mutex_unlock(&locks[y]);
  return vx + vy;
}
```

Each thread may invoke transfer or sum

No thread should observe the intermediate state of a transfer.

Thread 1: transfer(1, 2, 10);

Thread 2: sum(1, 2)

Example 3

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

// transfer money from account x to account y
void transfer(int x, int y, int amount) {
  pthread_mutex_lock(&locks[x]);
  accounts[x]->val -= amount;
  pthread_mutex_unlock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  accounts[y]->val += amount;
  pthread_mutex_unlock(&locks[y]);
}

// return the total of accounts x and y
int sum(int x, int y) {
  pthread_mutex_lock(&locks[x]);
  int vx = accounts[x]->val;
  pthread_mutex_unlock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  int vy = accounts[y]->val;
  pthread_mutex_unlock(&locks[y]);
  return vx + vy;
}
```

Each thread may invoke transfer or sum

No thread should observe the intermediate state of a transfer.

Any problems?

Thread 1: transfer(1, 2, 10);

Thread 2: sum(1, 2)

Example 3

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

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

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

Thread 0

`transfer(1, 2, 10)`

Thread 1

`sum(1, 2)`

0	1	2	3	4	5	6	7	8	9
50	50	50	50	50	50	50	50	50	50

Example 3

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

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

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

Thread 0

```
transfer(1, 2, 10)
pthread_mutex_lock(&locks[1]);
accounts[1]->val -= 10;
pthread_mutex_unlock(&locks[1]);
```

Thread 1

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

0	1	2	3	4	5	6	7	8	9
50	40	50	50	50	50	50	50	50	50

Example 3

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

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

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

Thread 0

```
transfer(1, 2, 10)
pthread_mutex_lock(&locks[1]);
accounts[1]->val -= 10;
pthread_mutex_unlock(&locks[1]);
```

Thread 1

```
sum(1, 2)   -> 90
pthread_mutex_lock(&locks[1]);
int vx = accounts[1]->val;
pthread_mutex_unlock(&locks[1]);
pthread_mutex_lock(&locks[2]);
int vx = accounts[2]->val;
pthread_mutex_unlock(&locks[2]);
```

0	1	2	3	4	5	6	7	8	9
50	40	50	50	50	50	50	50	50	50

Example 3

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

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

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

Thread 0

```
transfer(1, 2, 10)
pthread_mutex_lock(&locks[1]);
accounts[1]->val -= 10;
pthread_mutex_unlock(&locks[1]);

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

Thread 1

```
sum(1, 2)   -> 90

pthread_mutex_lock(&locks[1]);
int vx = accounts[1]->val;
pthread_mutex_unlock(&locks[1]);
pthread_mutex_lock(&locks[2]);
int vx = accounts[2]->val;
pthread_mutex_unlock(&locks[2]);
```

0	1	2	3	4	5	6	7	8	9
50	40	60	50	50	50	50	50	50	50

Example 3

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

// transfer money from account x to account y
void transfer(int x, int y, int amount) {
  pthread_mutex_lock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  accounts[x]->val -= amount;
  accounts[y]->val += amount;
  pthread_mutex_unlock(&locks[y]);
  pthread_mutex_unlock(&locks[x]);
}

// return the total of accounts x and y
int sum(int x, int y) {
  pthread_mutex_lock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  int vx = accounts[x]->val;
  int vy = accounts[y]->val;
  pthread_mutex_unlock(&locks[y]);
  pthread_mutex_unlock(&locks[x]);
  return vx + vy;
}
```

No thread can observe the intermediate state of a transfer.

Each thread still holds x's lock when accessing y.

Thread 1: transfer(1, 2, 10);

Thread 2: sum(1, 2)

Example 3

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

// transfer money from account x to account y
void transfer(int x, int y, int amount) {
  pthread_mutex_lock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  accounts[x]->val -= amount;
  accounts[y]->val += amount;
  pthread_mutex_unlock(&locks[y]);
  pthread_mutex_unlock(&locks[x]);
}

// return the total of accounts x and y
int sum(int x, int y) {
  pthread_mutex_lock(&locks[x]);
  pthread_mutex_lock(&locks[y]);
  int vx = accounts[x]->val;
  int vy = accounts[y]->val;
  pthread_mutex_unlock(&locks[y]);
  pthread_mutex_unlock(&locks[x]);
  return vx + vy;
}
```

No thread can observe the intermediate state of a transfer.

Each thread still holds x's lock when accessing y.

Any problems?

Thread 1: transfer(1, 2, 10);

Thread 2: sum(1, 2)

Deadlock

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

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

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

Thread 0

`transfer(1, 2, 10)`

Thread 1

`sum(2, 1)`

0	1	2	3	4	5	6	7	8	9
50	50	50	50	50	50	50	50	50	50

Deadlock

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

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

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

Thread 0	Thread 1
`transfer(1, 2, 10)`	`sum(2, 1)`
`pthread_mutex_lock(&locks[1]);`	`pthread_mutex_lock(&locks[2]);`

0	1	2	3	4	5	6	7	8	9
50	50	50	50	50	50	50	50	50	50

Deadlock

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

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

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

Thread 0

```
transfer(1, 2, 10)

pthread_mutex_lock(&locks[1]);
pthread_mutex_lock(&locks[2]);
```

block and wait

Z Z Z

Thread 1

```
sum(2, 1)

pthread_mutex_lock(&locks[2]);
pthread_mutex_lock(&locks[1]);
```

block and wait

Z Z Z

0	1	2	3	4	5	6	7	8	9
50	50	50	50	50	50	50	50	50	50

Deadlock

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

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

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

Thread 0

```
transfer(1, 2, 10)

pthread_mutex_lock(&locks[1]);
pthread_mutex_lock(&locks[2]);
```

block and wait

Thread 1

```
sum(2, 1)

pthread_mutex_lock(&locks[2]);
pthread_mutex_lock(&locks[1]);
```

block and wait

Program cannot make any progress!

0	1	2	3	4	5	6	7	8	9
50	50	50	50	50	50	50	50	50	50

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.

Technique 1: `trylock`

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

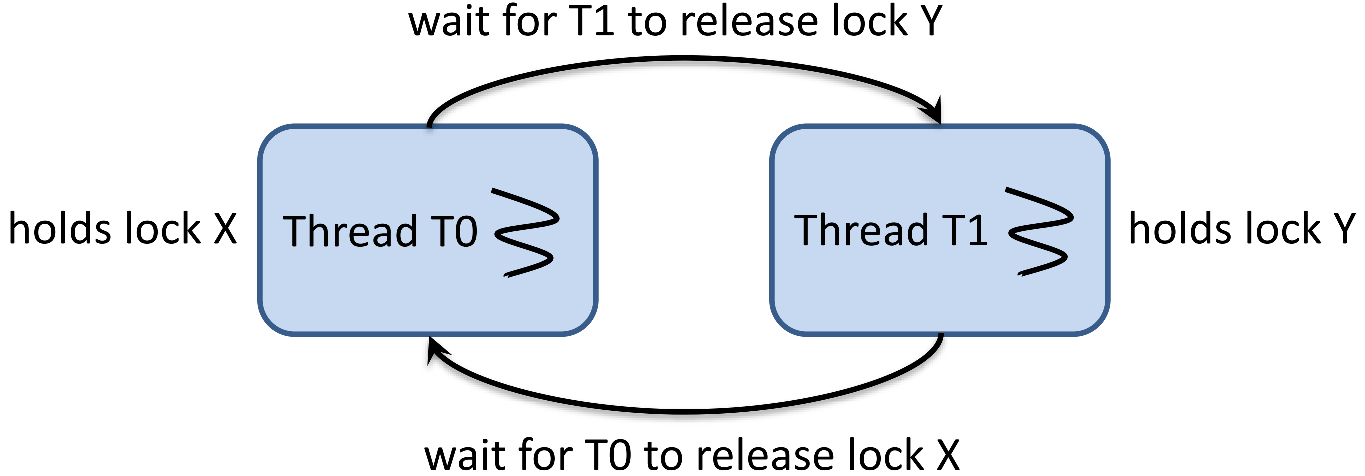
Technique 1: trylock

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

must release lock on x if trylock is unsuccessful

Technique 2: Lock Ordering

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



- Technique: all threads acquire locks in the same order