Condition Variable

Zhaoguo Wang

Example I

```
typedef struct {
  int data[MAX];
  int size;
} buffer_t;

buffer_t buf;
int result;
```


```
void* sender(void *arg){

  srandom(time());

  while(1) {
    while(buf.size < Max) {
      buf.size = buf.size + 1;
      buf.data[buf.size - 1] = random();
    }
  }
    return NULL;
}
```

```
void* receiver(void *arg){

  srandom(time());

  while(1) {
    while(buf.size > 0) {
      total += buf.data[buf.size - 1];
      buf.size = buf.size - 1;
    }
  }
    return NULL;
}
```

Example I


```
typedef struct {
  int data[MAX];
  int size;
  pthread_mutex_t mutex;
} buffer_t;

buffer_t buf;
int result;

void* sender(void *arg){

  srandom(time());

  while(1) {
    pthread_mutex_lock(&buf.mutex);
    while(buf.size < Max) {
      buf.size = buf.size + 1;
      buf.data[buf.size - 1] = random();
    }
    pthread_mutex_unlock(&buf.mutex);
  }
    return NULL;
}
```

```
void* receiver(void *arg){

  srandom(time());

  while(1) {
    pthread_mutex_lock(&buf.mutex);
    while(buf.size > 0) {
      total += buf.data[buf.size - 1];
      buf.size = buf.size - 1;
    }
    pthread_mutex_unlock(&buf.mutex);
  }
    return NULL;
}
```

Example I


```
typedef struct {
  int data[MAX];
  int size;
  pthread_mutex_t mutex;
} buffer_t;

buffer_t buf;
int result;

void* sender(void *arg){

  srandom(time());

  while(1) {
    pthread_mutex_lock(&buf.mutex);
    while(buf.size < Max) {
      buf.size = buf.size + 1;
      buf.data[buf.size - 1] = random();
    }
    pthread_mutex_unlock(&buf.mutex);
  }
    return NULL;
}
```

```
void* receiver(void *arg){

  srandom(time());

  while(1) {
    pthread_mutex_lock(&buf.mutex);
    while(buf.size > 0) {
      total += buf.data[buf.size - 1];
      buf.size = buf.size - 1;
    }
    pthread_mutex_unlock(&buf.mutex);
  }
    return NULL;
}
```

Problem?

Problem

Sender and Receiver need to explicitly check the buffer is empty or full

- Waste CPU cycles on falsely check

Solution: a notification mechanism

Condition variables

A mechanism to block a thread until some condition is true

- Usually being used to implement notification mechanisms

Conditional variable API in pthread

- `pthread_cond_t`
- `pthread_cond_wait / pthread_cond_timedwait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`

pthread_cond_wait

```
int pthread_cond_wait(pthread_cond_t * cond,
                      pthread_mutex_t * mutex);
```

This function atomically releases `mutex` and cause the calling thread to block on the condition variable `cond`.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

pthread_cond_wait

```
int pthread_cond_wait(pthread_cond_t * cond,
                      pthread_mutex_t * mutex);
```

This function atomically releases `mutex` and cause the calling thread to block on the condition variable `cond`.

all occur, or nothing occurs

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

pthread_cond_signal

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Unblock at least one of the threads blocked on a condition variable.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Unblock all threads blocked on a condition variable.

Pseudo-code

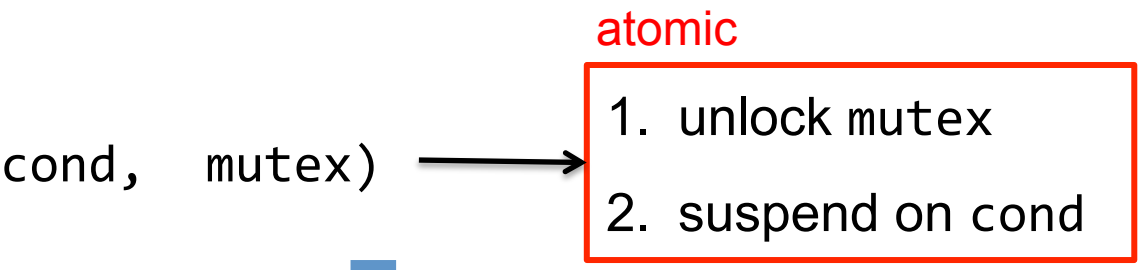
Thread 1

```
pthread_cond_wait(cond,  mutex)
```

Pseudo-code

Thread 1

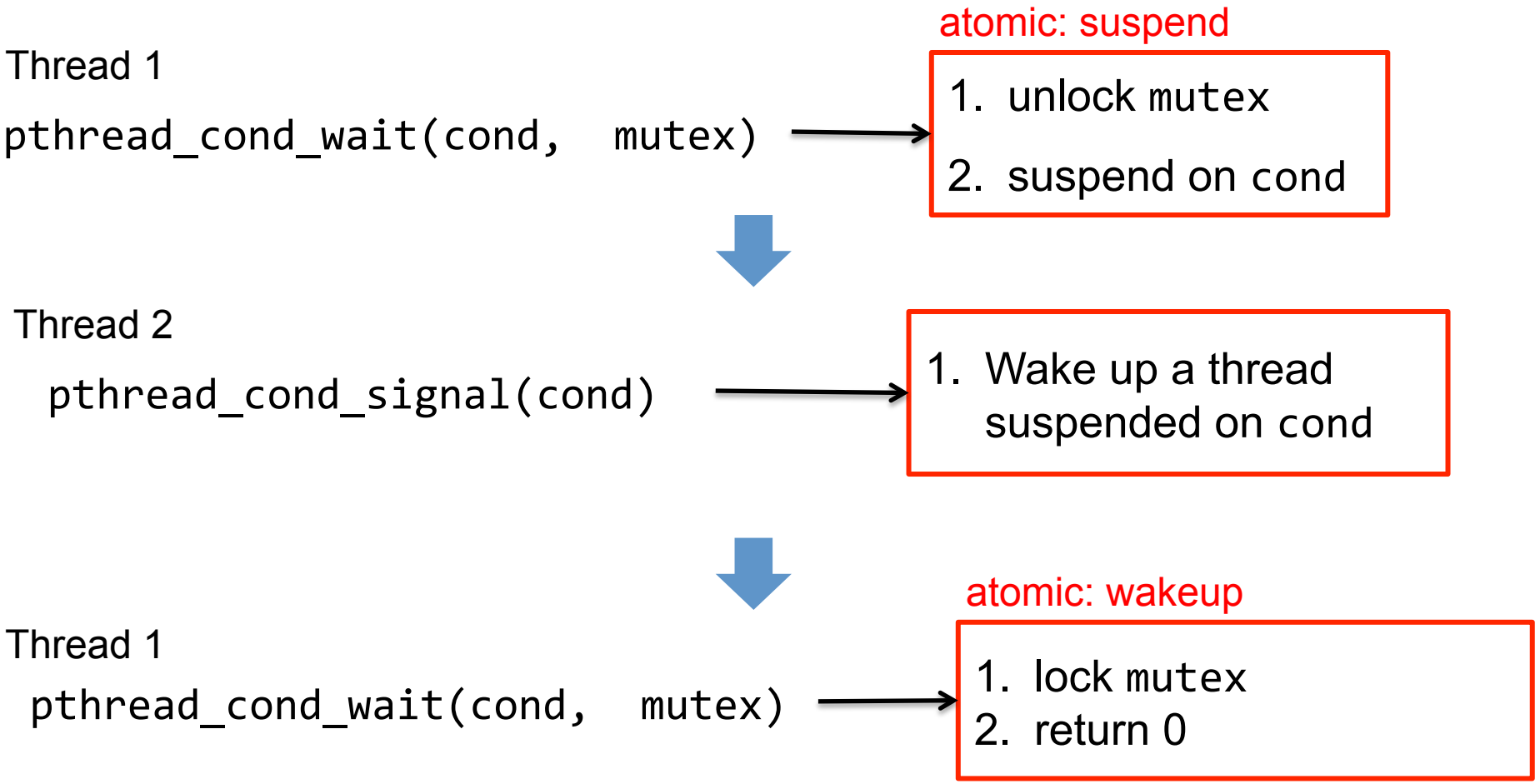
```
pthread_cond_wait(cond,  mutex)
```



Thread 2

```
pthread_cond_signal(cond)
```

Pseudo-code


atomic: suspend
Thread 1
pthread_cond_wait(cond, mutex)
1. unlock mutex
2. suspend on cond
Thread 2
pthread_cond_signal(cond)
1. Wake up a thread suspended on cond
atomic: wakeup
Thread 1
pthread_cond_wait(cond, mutex)
1. lock mutex
2. return 0

Example II – hellotiger

```
pthread_mutex_t mutex;
bool print = false;


void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    if(print) {
      printf("tiger [number %d]\n", id);
      print = false;
    }
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

Example II – hellotiger

```
pthread_mutex_t mutex;
pthread_cond_t cond;
bool print = false;


void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    while(!print) {
      pthread_cond_wait(&mutex, &cond);
    }
    print =false;
    printf("tiger [number %d]\n", id);
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

Example II – hellotiger

```
pthread_mutex_t mutex;
pthread_cond_t cond;
bool print = false;

void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    while(!print) {
      pthread_cond_wait(&mutex, &cond);
    }
    print =false;
    printf("tiger [number %d]\n", id);
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

Spurious wakeups from the pthread_cond_timedwait() or pthread_cond_wait() functions may occur.

Example II – hellotiger

```
pthread_mutex_t mutex;
pthread_cond_t cond;
bool print = false;

void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

Why need lock to protect signal?
Is this correct?

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    while(!print) {
      pthread_cond_wait(&mutex, &cond);
    }
    print =false;
    printf("tiger [number %d]\n", id);
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

Example II – hellotiger

```
pthread_mutex_t mutex;
pthread_cond_t cond;
bool print = false;

void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    while(!print) {
      pthread_cond_wait(&mutex, &cond);
    }
    print =false;
    printf("tiger [number %d]\n", id);
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

Why need lock to protect signal?
Is this correct?
No! Lose signal!

Example II – hellotiger

```
pthread_mutex_t mutex;
pthread_cond_t cond;
bool print = false;

void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

Why need lock to protect signal?
Is this correct?
No! Lose signal!

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    while(!print) {
      pthread_cond_wait(&mutex, &cond);
    }
    print =false;
    printf("tiger [number %d]\n", id);
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

Example II – hellotiger

```
pthread_mutex_t mutex;
pthread_cond_t cond;
bool print = false;

void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    while(!print) {
      pthread_cond_wait(&mutex, &cond);
    }
    print =false;
    printf("tiger [number %d]\n", id);
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

Why it needs to atomically release the lock and suspend threads?

Example II – hellotiger

```
pthread_mutex_t mutex;
pthread_cond_t cond;
bool print = false;

void* notify(void *arg){

  while(1) {
    pthread_mutex_lock(&mutex);
    print = true;
    printf("hello ");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
  }
   return NULL;
}
```

```
void* tiger(void *arg){

  int id = *(int *)arg;

  while(1) {
    pthread_mutex_lock(&mutex);
    while(!print) {
      pthread_cond_wait(&mutex, &cond);
      pthread_mutex_unlock(&mutex);
      pthread_cond_block(&cond);
    }
    print =false;
    printf("tiger [number %d]\n", id);
    pthread_mutex_unlock(&mutex);
  }
    return NULL;
}
```

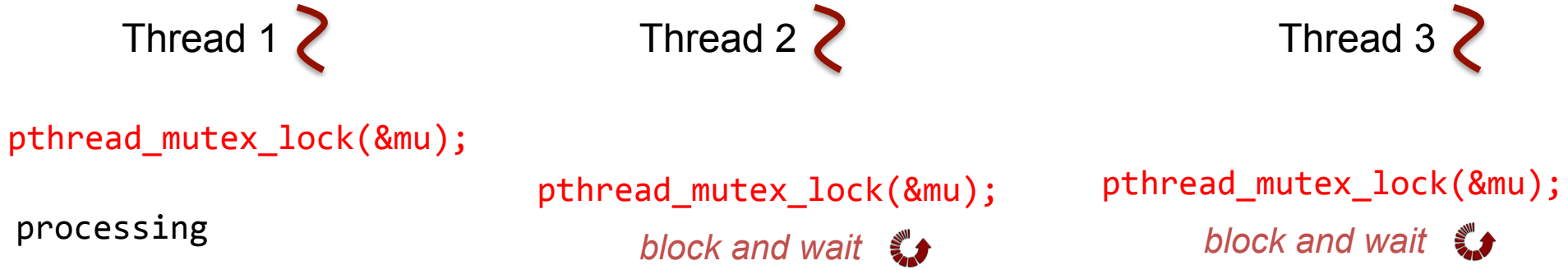
Why it needs to atomically release the lock and suspend threads?
avoid losing signal!

Example I

```
typedef struct {
  int data[MAX];
  int size;
  pthread_mutex_t mutex;
  pthread_cond_t empty;
  pthread_cond_t full;
} buffer_t;

buffer_t buf;
int result;

void* sender(void *arg){

  srandom(time());

  while(1) {
    pthread_mutex_lock(&buf.mutex);
    while(buffer_is_full()) {
      pthread_cond_wait(&buf.empty,
                        &buf.mutex);
    }
    fill_buffer();
    pthread_cond_signal(&buf.full);
    pthread_mutex_unlock(&buf.mutex);
  }
    return NULL;
}

void* receiver(void *arg){

  srandom(time());

  while(1) {
    pthread_mutex_lock(&buf.mutex);
    while(buffer_is_empty()) {
      pthread_cond_wait(&buf.full,
                        &buf.mutex);

    }
    read_buffer();
    pthread_cond_signal(&buf.empty);
    pthread_mutex_unlock(&buf.mutex);
  }
    return NULL;
}
```

Example III

The unfairness of pthread_mutex_lock



Example III

The unfairness of pthread_mutex_lock

Thread 1

```
pthread_mutex_lock(&mu);

processing

pthread_mutex_unlock(&mu);
```

Thread 2

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

block and wait

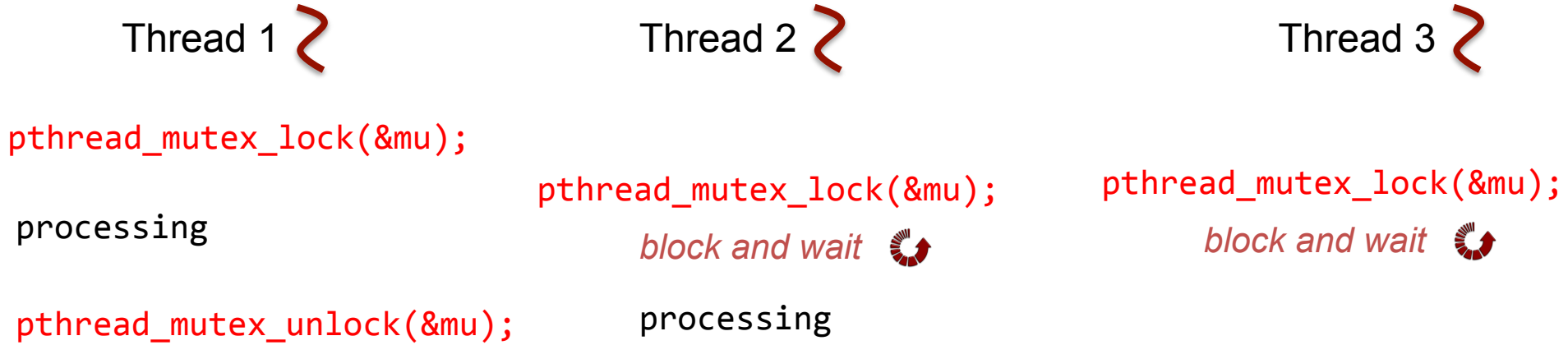
Thread 3

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

block and wait

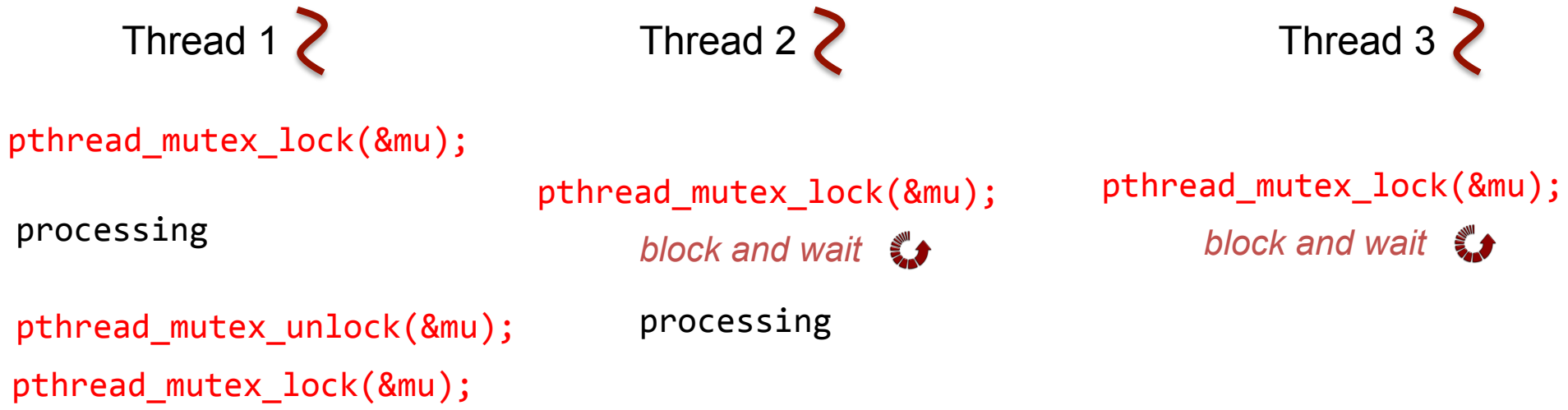
Example III

The unfairness of pthread_mutex_lock



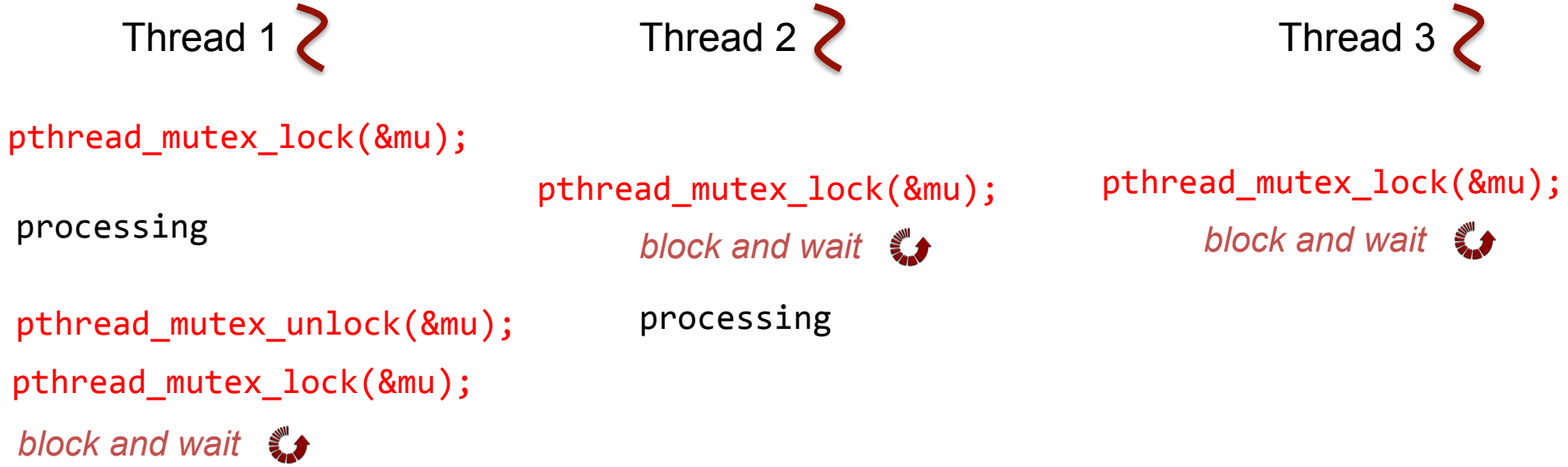
Example III

The unfairness of pthread_mutex_lock



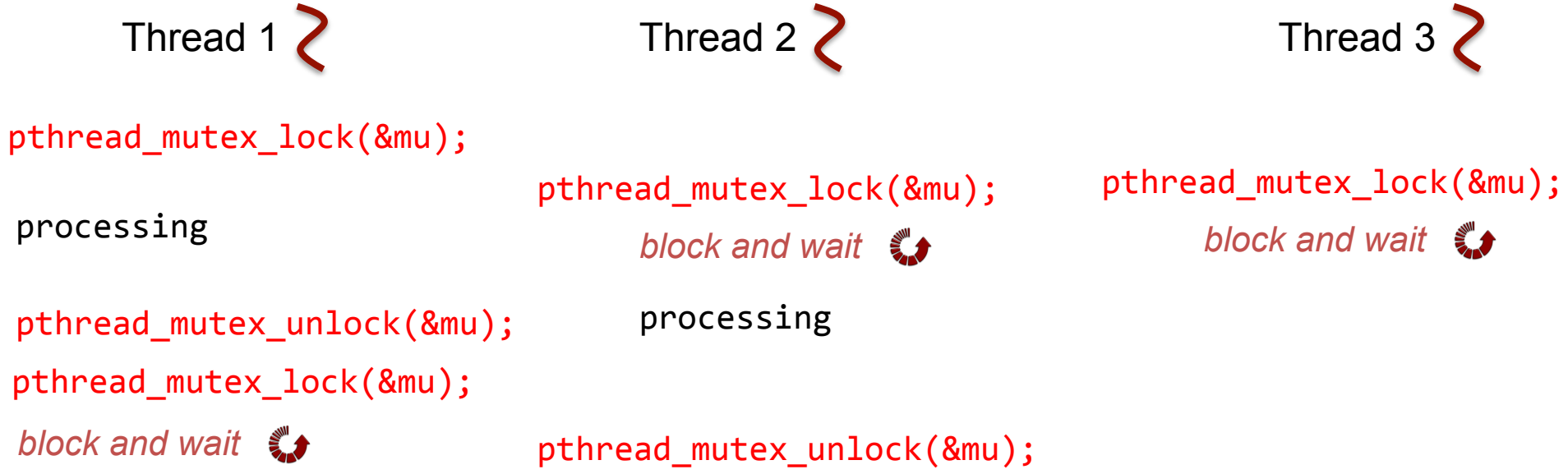
Example III

The unfairness of pthread_mutex_lock



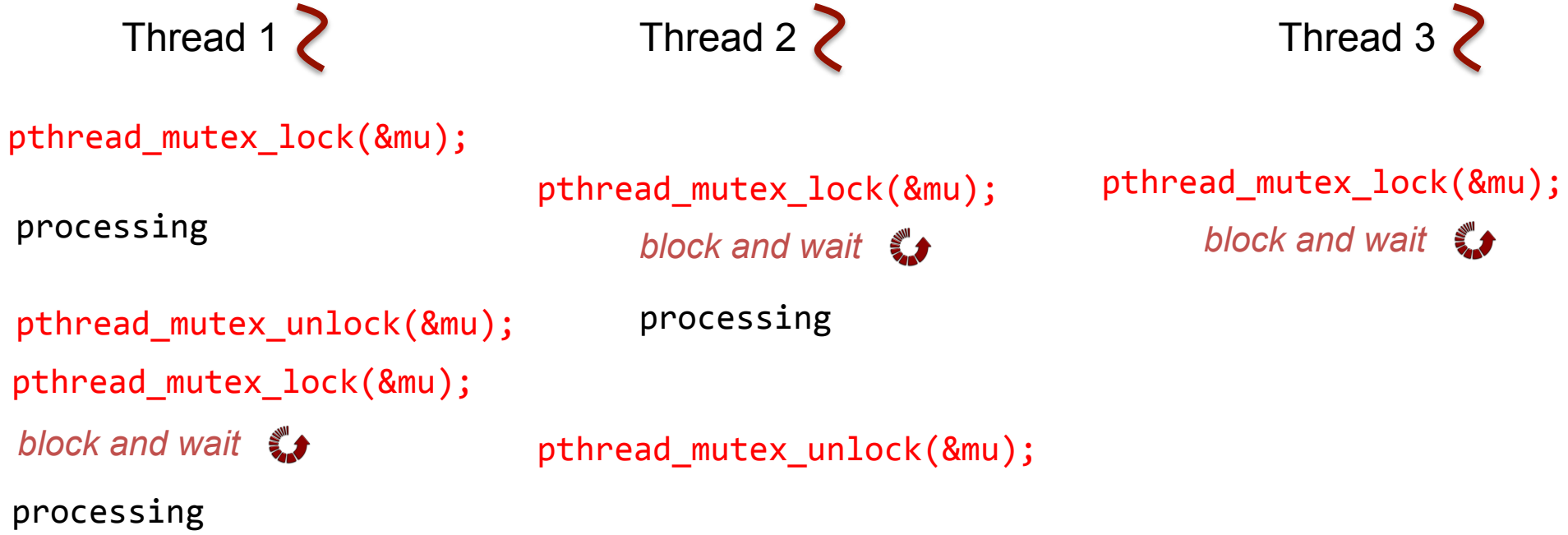
Example III

The unfairness of pthread_mutex_lock



Example III

The unfairness of pthread_mutex_lock



Starving of Thread 3!

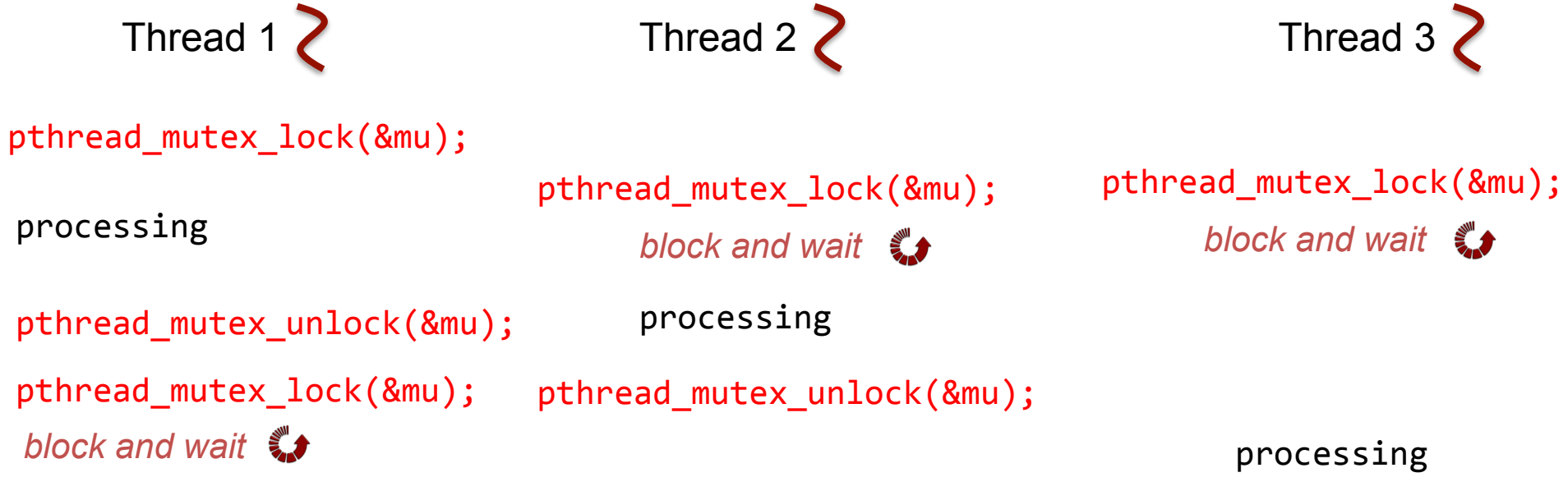
Example III

Add fairness to the mutex → FIFO Lock

- A first in first out queue-based locking mechanism
- Locks are dealt out in the order they are requested
  - If t_1 tries to acquire lock before t_2, then t_1 always gets the lock first

Example III

The unfairness of pthread_mutex_lock

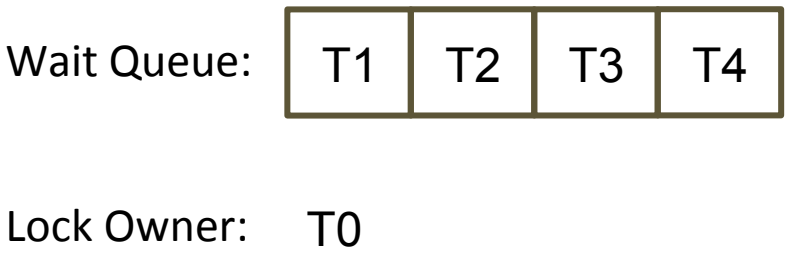


Thread 2 passes the lock to Thread 3.

Example III

Add fairness to the mutex → FIFO Lock

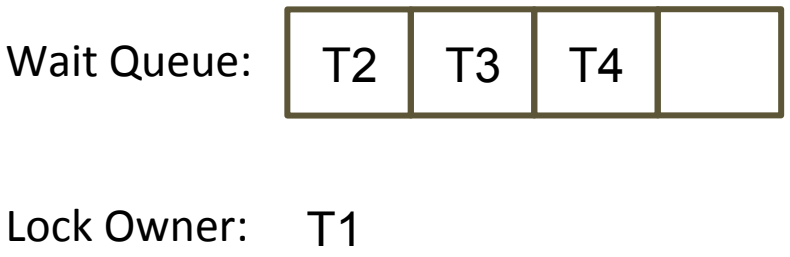
- A first in first out queue-based locking mechanism
- Locks are dealt out in the order they are requested
  - If t_1 tries to acquire lock before t_2, then t_1 always gets the lock first



Example III

Add fairness to the mutex → FIFO Lock

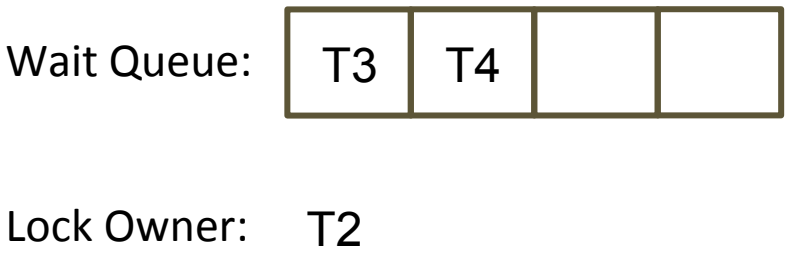
- A first in first out queue-based locking mechanism
- Locks are dealt out in the order they are requested
  - If t_1 tries to acquire lock before t_2, then t_1 always gets the lock first



Example III

Add fairness to the mutex → FIFO Lock

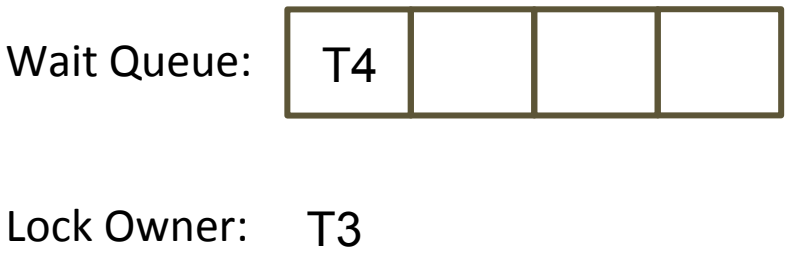
- A first in first out queue-based locking mechanism
- Locks are dealt out in the order they are requested
  - If t_1 tries to acquire lock before t_2, then t_1 always gets the lock first



Example III

Add fairness to the mutex → FIFO Lock

- A first in first out queue-based locking mechanism
- Locks are dealt out in the order they are requested
  - If t_1 tries to acquire lock before t_2, then t_1 always gets the lock first



Example III

Add fairness to the mutex → FIFO Lock

- A first in first out queue-based locking mechanism
- Locks are dealt out in the order they are requested
  - If t_1 tries to acquire lock before t_2, then t_1 always gets the lock first

Lock Owner: T4

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

Lock type: queue is implemented as a linked list

1. Keep the head and tail of the linked list
2. Use a bit (busy) to indicate the lock is locked or unlocked
3. A mutex is needed to protect the accesses to these shared fields

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

Lock type: queue is implemented as a linked list

1. Keep the head and tail of the linked list
2. Use a bit (busy) to indicate the lock is locked or unlocked
3. A mutex is needed to protect the accesses to these shared fields

Each node in the linked list: represent a waiting thread

1. The waiting thread is blocking on the cond field
2. Blocked indicates if the thread should be blocked or not

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
```

Acquire Lock

1. If the lock is unlocked, set the busy bit and return

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
```

Acquire Lock

1. If the lock is unlocked, set the busy bit and return
2. Otherwise create a node and append it to the linked list. (Blocked is initialized to be 1)

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
```

Acquire Lock

1. If the lock is unlocked, set the busy bit and return
2. Otherwise create a node and append it to the linked list. (Blocked is initialized to be 1)
3. Suspend itself on the cond variable of the created node.

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
```

Release Lock

1. If there is no waiters, clear the busy field.

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

Release Lock

1. If there is no waiters, clear the busy field.
2. Otherwise, clear the blocked field of the first node in the waiting list and wakeup the suspended thread.

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

Acquire Lock

4. Remove and free the node from the waiting list

```
lock_t l < busy: 0, head: null, tail: null >,  int global: 0
```

Thread 1

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

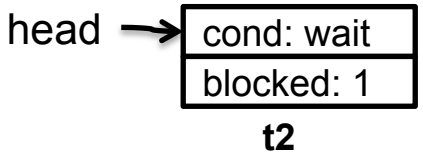
lock_t l < busy: 1, head: null, tail: null >, int global: 0

Thread 1

tthread_fifo_lock(&l)

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t2, tail: t2>,  int global: 0`



Thread 1

Thread 2

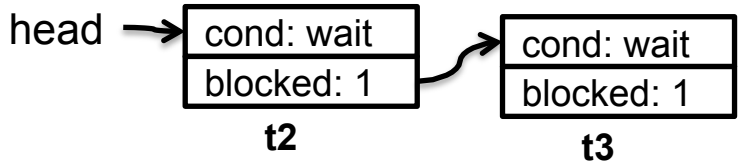
`tthread_fifo_lock(&l)`

`tthread_fifo_lock(&l)`

wait and block

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t2, tail: t3>,  int global: 0`



Thread 1

`tthread_fifo_lock(&l)`

Thread 2

`tthread_fifo_lock(&l)`

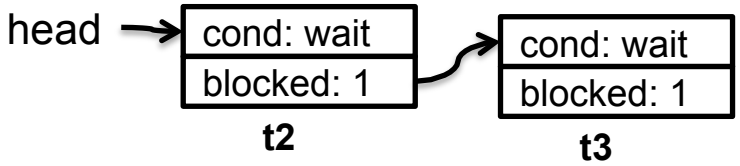
wait and block

Thread 3

`tthread_fifo_lock(&l)`

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t2, tail: t3>,  int global: 1`



Thread 1 | Thread 2 | Thread 3

`tthread_fifo_lock(&l)`

`tthread_fifo_lock(&l)`

`global++`

wait and block

`tthread_fifo_lock(&l)`

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t2, tail: t3>,  int global: 1`

Thread 1 | Thread 2 | Thread 3

`tthread_fifo_lock(&l)`

`tthread_fifo_lock(&l)`

`global++`

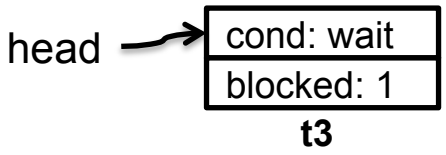
wait and block

`tthread_fifo_lock(&l)`

`tthread_fifo_unlock(&l)`

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
   // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t3, tail: t3>,  int global: 2`



Thread 1

Thread 2

Thread 3

`tthread_fifo_lock(&l)`

`tthread_fifo_lock(&l)`

`global++`

wait and block

`tthread_fifo_lock(&l)`

`tthread_fifo_unlock(&l)`

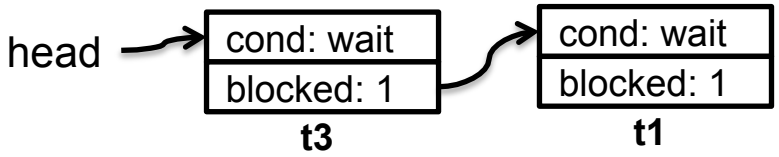
wakeup

`global++`

wait and block

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

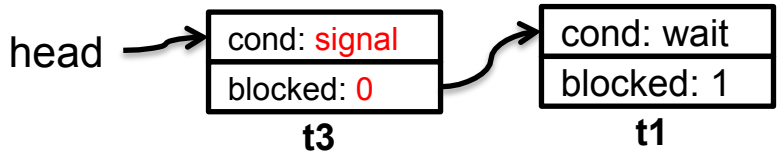
`lock_t l < busy: 1, head: t3, tail: t1>,  int global: 2`



Thread 1	Thread 2	Thread 3
`tthread_fifo_lock(&l)`		
	`tthread_fifo_lock(&l)`	
`global++`	**wait and block**	`tthread_fifo_lock(&l)`
`tthread_fifo_unlock(&l)`	**wakeup**	**wait and block**
`tthread_fifo_lock(&l)`	`global++`	

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t3, tail: t1>,  int global: 2`



Thread 1 ≀ Thread 2 ≀ Thread 3 ≀

```
tthread_fifo_lock(&l)
                          tthread_fifo_lock(&l)
  global++                wait and block          tthread_fifo_lock(&l)

tthread_fifo_unlock(&l)   wakeup                  wait and block
tthread_fifo_lock(&l)     global++
                          tthread_fifo_unlock(&l)
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
   // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t1, tail: t1>,  int global: 3`

Thread 1

```
tthread_fifo_lock(&l)

  global++

tthread_fifo_unlock(&l)
tthread_fifo_lock(&l)
```

wait and block

Thread 2

```
tthread_fifo_lock(&l)
```

wait and block

wakeup

```
global++
tthread_fifo_unlock(&l)
```

Thread 3

```
tthread_fifo_lock(&l)
```

wait and block

wakeup

```
global++
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
   // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 1, head: t1, tail: t1>,  int global: 3`

Thread 1

```
tthread_fifo_lock(&l)

  global++

tthread_fifo_unlock(&l)
tthread_fifo_lock(&l)
```

wait and block

Thread 2

```
tthread_fifo_lock(&l)
```

wait and block

wakeup

```
global++

tthread_fifo_unlock(&l)
```

Thread 3

```
tthread_fifo_lock(&l)
```

wait and block

wakeup

```
global++

tthread_fifo_unlock(&l)
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
   // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

```
lock_t l < busy: 1, head: null, tail: null>,  int global: 3
```

t1

Thread 1

```
tthread_fifo_lock(&l)

  global++

tthread_fifo_unlock(&l)
tthread_fifo_lock(&l)
```

wait and block

wakeup

Thread 2

```
tthread_fifo_lock(&l)
```

wait and block

wakeup

```
global++
tthread_fifo_unlock(&l)
```

Thread 3

```
tthread_fifo_lock(&l)
```

wait and block

wakeup

```
global++
tthread_fifo_unlock(&l)
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
   // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

`lock_t l < busy: 0, head: null, tail: null>,  int global: 4`

t1

Thread 1

`tthread_fifo_lock(&l)`

`global++`

`tthread_fifo_unlock(&l)`

`tthread_fifo_lock(&l)`

wait and block

wakeup

`global++`

`tthread_fifo_unlock(&l)`

Thread 2

`tthread_fifo_lock(&l)`

wait and block

wakeup

`global++`

`tthread_fifo_unlock(&l)`

Thread 3

`tthread_fifo_lock(&l)`

wait and block

wakeup

`global++`

`tthread_fifo_unlock(&l)`

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
   // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

```
typedef struct node_t {
  pthread_cond_t cond;
  struct node_t* next;
  int blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mutex;
  node_t *head, *tail;
  int busy; // 0: free, 1: busy
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // lock is free, hold the lock
  if(l->busy == 0) {
    l->busy = 1;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  // lock is busy, suspend on a new cond
  node_t *n = malloc(sizeof(node_t));
  n->blocked = 1;
  if(l->head == NULL) {
    l->head = n;
    l->tail = l->head;
  } else {
    l->tail->next = n;
    l->tail = l->tail->next;
  }
  while(l->head->blocked) {
    pthread_cond_wait(&l->tail->cond, &l->mutex);
  }
  l->head = l->head->next;
  if(l->head == NULL) l->tail = NULL;
  free(n);
  pthread_mutex_unlock(&l->mutex);
  return 0;
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  // no waiters
  if(l->head == NULL) {
    l->busy = 0;
    pthread_mutex_unlock(&l->mutex);
    return 0;
  }
  l->head->blocked = 0;
  pthread_cond_signal(&l->head->cond);
  pthread_mutex_unlock(&l->mutex);
  return 0;
}
```

Can you get rid of the list?

```
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  volatile unsigned long owner, ticket;
} lock_t;
```

Basic Idea

When a thread requests the lock, it will be assigned with a ticket number. The thread needs to wait until its turn is up.

Lock

1. owner: the holder's ticket number
2. ticket: the ticket number waits to be assigned
3. cond: all waiting threads are blocked on cond

```
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  volatile unsigned long owner, ticket;
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

Acquire a lock

1. Get a ticket number from ticket and update it

```
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  volatile unsigned long owner, ticket;
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

Acquire a lock

1. Get a ticket number from ticket and update it
2. Check if its turn is up by comparing holder's ticket number with its local ticket number

```
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  volatile unsigned long owner, ticket;
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  l->owner++;
  pthread_cond_broadcast(&l->cond);
  pthread_mutex_unlock(&l->mutex);
}
```

Release the lock

1. Increate the holder's ticket number to pass the lock the next waiter
2. Wakeup all the waiters

```
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  volatile unsigned long owner, ticket;
} lock_t;
```

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  l->owner++;
  pthread_cond_broadcast(&l->cond);
  pthread_mutex_unlock(&l->mutex);
}
```

Acquire the lock

1. After all waiters wakeup, each one will its local ticket number with the holder's ticket number.
2. Only the thread which has the same ticket number with the owner will hold the lock, all the others will suspend them self again.

```
lock_t l < owner: 0, ticket: 0 >,  int global: 0
```

Thread 1

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

```
lock_t l < owner: 0, ticket: 1 >,  int global: 0
```

Thread 1

```
tthread_fifo_lock(&l)
```

Get ticket 0

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

```
lock_t l < owner: 0, ticket: 2 >,  int global: 0
```

Thread 1

```
tthread_fifo_lock(&l)
```

Get ticket 0

Thread 2

```
tthread_fifo_lock(&l)
```

Get ticket 1, wait for owner to be 1

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

```
lock_t l < owner: 0, ticket: 3 >,  int global: 0
```

Thread 1

```
tthread_fifo_lock(&l)
```

Get ticket 0

Thread 2

```
tthread_fifo_lock(&l)
```

Get ticket 1, wait for owner to be 1

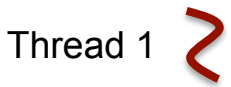
Thread 3

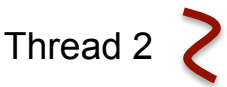
```
tthread_fifo_lock(&l)
```

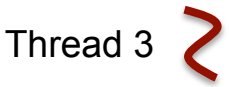
Get ticket 2, wait for owner to be 2

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

```
lock_t l < owner: 1, ticket: 3 >,  int global: 1
```








```
tthread_fifo_lock(&l)
```

Get ticket 0

```
global++

tthread_fifo_unlock(&l)
```

Pass the lock to next, by updating owner

```
tthread_fifo_lock(&l)
```

Get ticket 1, wait for owner to be 1

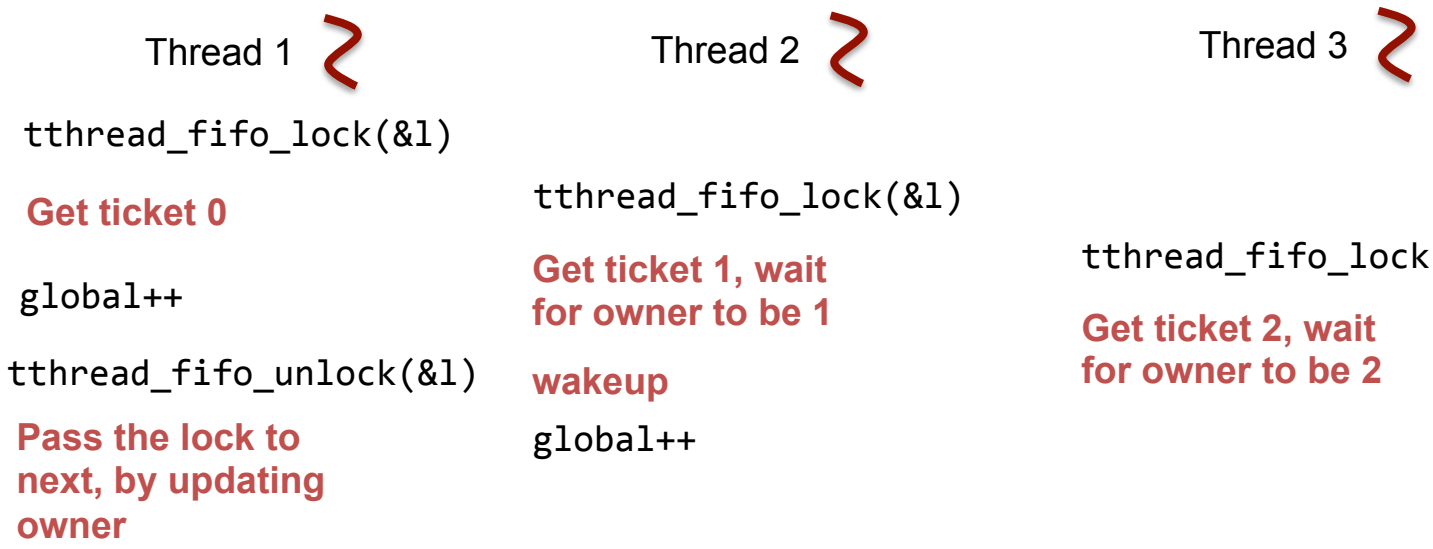
```
tthread_fifo_lock(&l)
```

Get ticket 2, wait for owner to be 2

```
int tthread_fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mutex);
  l->owner++;
  pthread_cond_broadcast(&l->cond);
  pthread_mutex_unlock(&l->mutex);
}
```

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

lock_t l < owner: 1, ticket: 3 >, int global: 2




```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

lock_t l < owner: 1, ticket: 4 >, int global: 2

tthread_fifo_lock(&l)

Get ticket 0

global++

tthread_fifo_unlock(&l)

Pass the lock to next, by updating owner

tthread_fifo_lock(&l)

Get ticket 3, wait for owner to be 3

tthread_fifo_lock(&l)

Get ticket 1, wait for owner to be 1

wakeup

global++

tthread_fifo_lock(&l)

Get ticket 2, wait for owner to be 2

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

lock_t l < owner: 2, ticket: 4 >, int global: 2

Thread 1

tthread_fifo_lock(&l)

Get ticket 0

global++

tthread_fifo_unlock(&l)

Pass the lock to next, by updating owner

tthread_fifo_lock(&l)

Get ticket 3, wait for owner to be 3

Thread 2

tthread_fifo_lock(&l)

Get ticket 1, wait for owner to be 1

wakeup

global++

tthread_fifo_unlock(&l)

Pass the lock to next, by updating owner

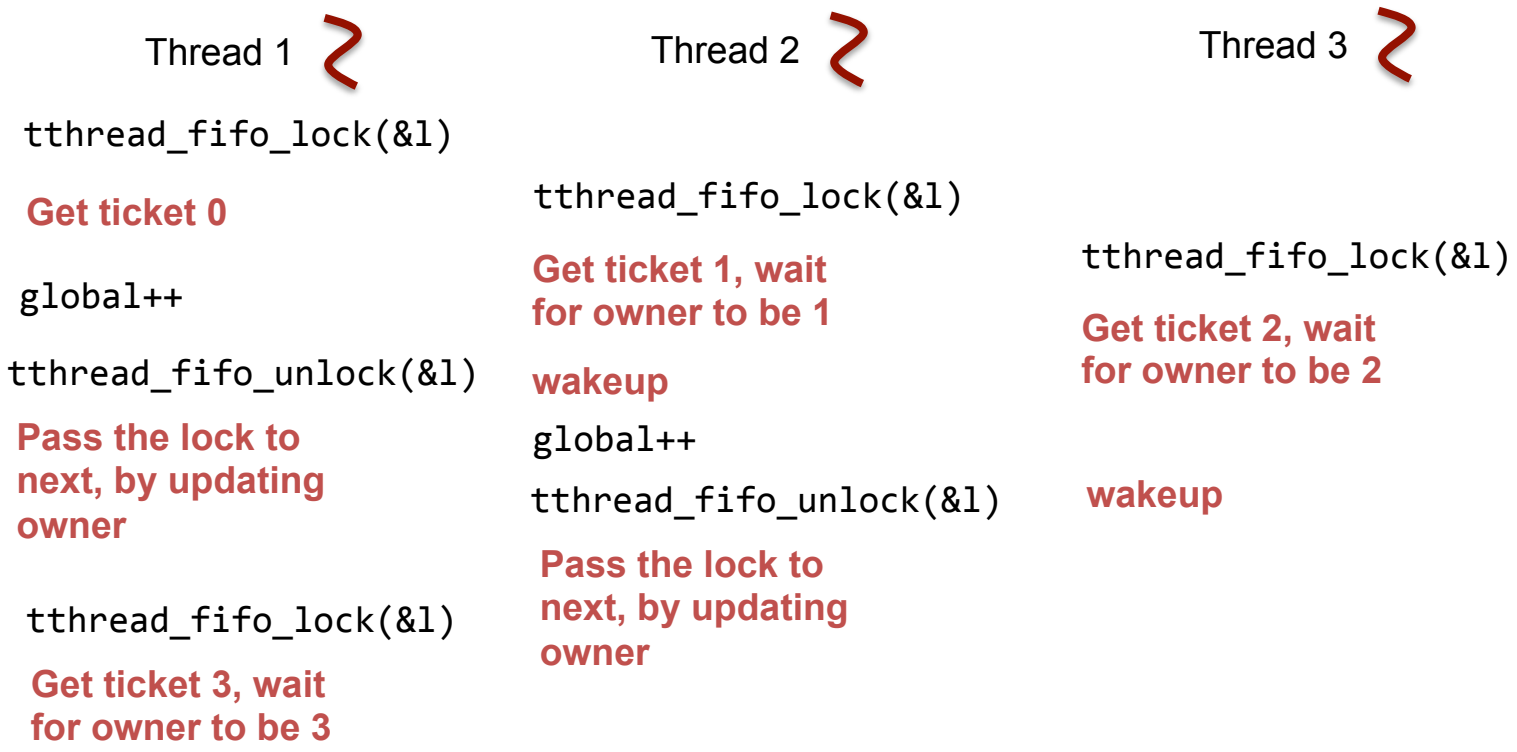
Thread 3

tthread_fifo_lock(&l)

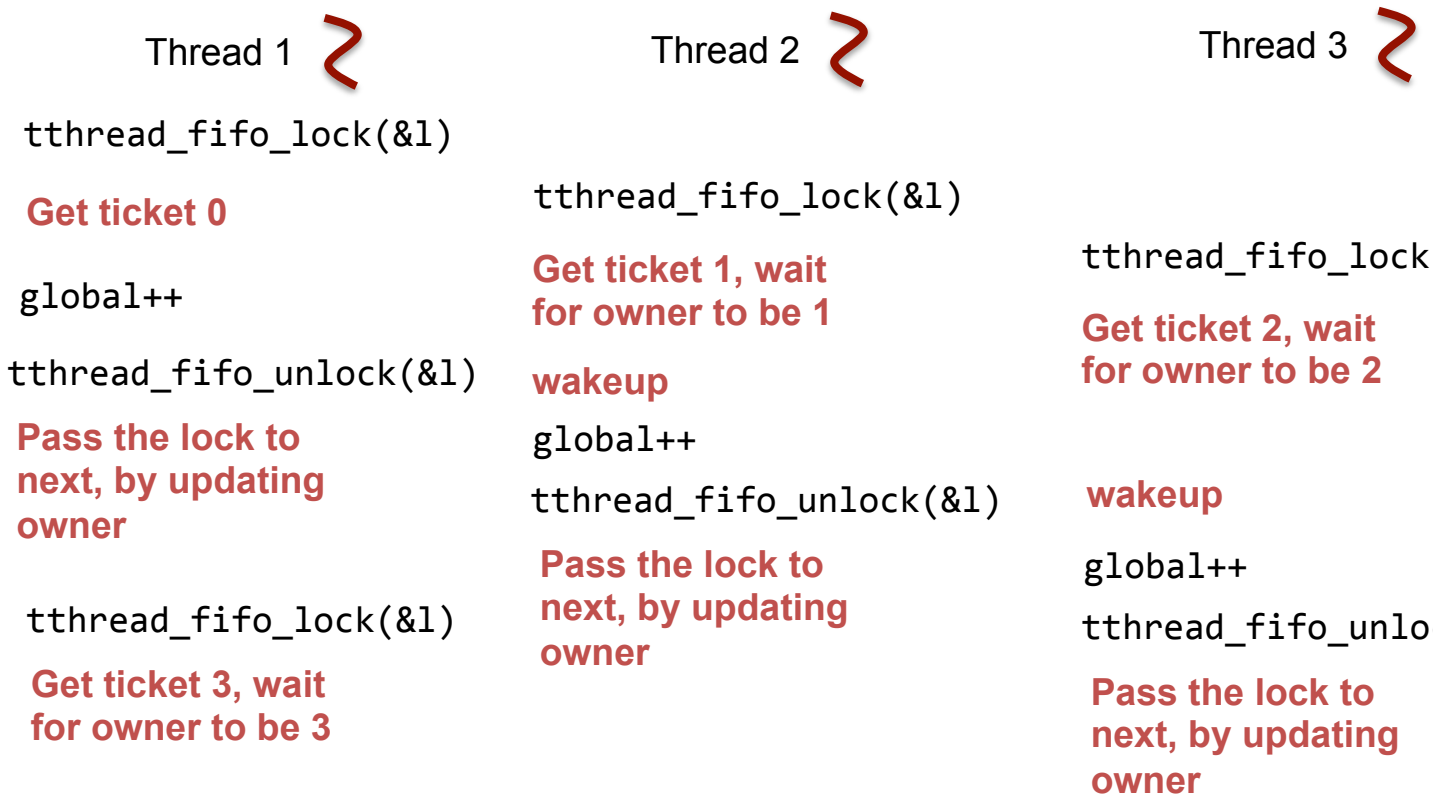
Get ticket 2, wait for owner to be 2

```
int tthread_fifo_lock(lock_t *l) {

  unsigned long me;

  pthread_mutex_lock(&l->mutex);
  me = l->ticket++;
  while(me != l->owner) {
    pthread_cond_wait(&l->cond, &l->mutex);
  }
  pthread_mutex_unlock(&l->mutex);

  return 0;
}
```

```
lock_t l < owner: 2, ticket: 4 >,  int global: 2
```



```
lock_t l < owner: 3, ticket: 4 >,  int global: 2
```



lock_t l < owner: 4, ticket: 4 >, int global: 2