Conditional Variable

Jinyang Li

based on slides by Tiger Wang

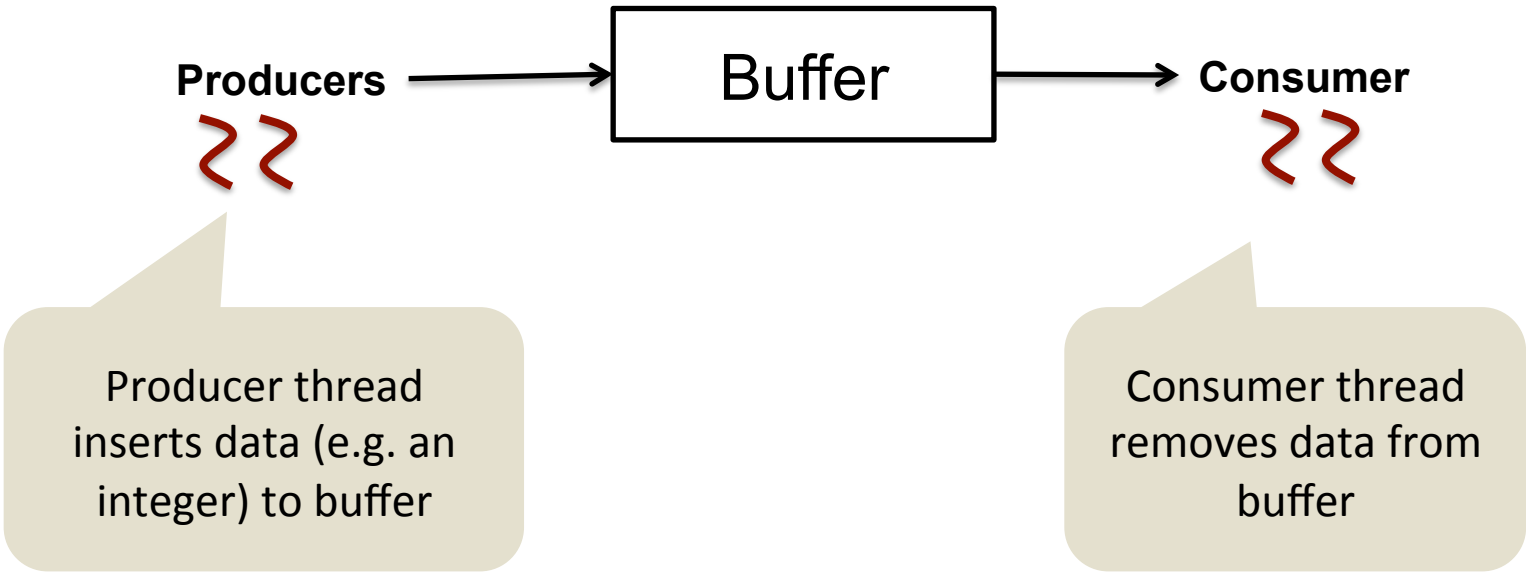
What we've learnt before

- Races
- One form of synchronization: mutual exclusion
  - use locks
- When to lock?
  - Whenever some state is accessed by >=2 threads and one thread writes the variable.
  - Mentally associate locks with state.

Today

- Another form of synchronization: condition
  - One thread waits for (another thread to make) some condition to be true

Producer-consumer example



- Producer must wait for consumer
  - if buffer is about to overflow
- Consumer must wait for producer
  - if buffer is empty

Producer-consumer example

```
typedef struct {
  int data[MAX];
  int size; // # of data in buffer, initialized to be 0
} buffer_t;

buffer_t buf;
```

```
void* producer(void *arg){
  int r = random(); //produce data
  while (1) {
    if (buf.size < MAX) {
      buf.data[buf.size] = r;
      buf.size++;
      break;
    }
  }
  printf("produced %d\n", r);
  return NULL;
}
```

```
void* consumer(void *arg){
  int r;
  while(1) {
    if (buf.size > 0) {
      r = buf.data[buf.size - 1];
      buf.size--;
      break;
    }
  }
  printf("consumed %d\n", r);
  return NULL;
}
```

correct??

Producer-consumer example

```
typedef struct {
  int data[MAX];
  int size; // # of data in buffer, initialized to be 0
  pthread_mutex_t mu; //protects data and size
} buffer_t;

buffer_t buf;
```

```
void* producer(void *arg){
  int r = random(); //produce data
  while (1) {
    pthread_mutex_lock(&buf.mu);
    if (buf.size < MAX) {
      buf.data[buf.size] = r;
      buf.size++;
      break;
    }
    pthread_mutex_unlock(&buf.mu);
  }
  printf("produced %d\n", r);
  return NULL;
}
```

```
void* consumer(void *arg){
  int r;
  while(1) {
    pthread_mutex_lock(&buf.mu);
    if (buf.size > 0) {
      r = buf.data[buf.size - 1];
      buf.size--;
      break;
    }
    pthread_mutex_unlock(&buf.mu);
  }
  printf("consumed %d\n", r);
  return NULL;
}
```

Problem with previous naive solution

- Naive solution: busy checking whether condition is true or false

  X wastes CPU

✓ Solution: a notification mechanism

Conditional variables

- A mechanism to block a thread until some condition becomes true
- Programmers mentally associate a conditional variable with some condition
  - A thread can wait on the condition (to become true):
    - `pthread_cond_wait`
  - A thread can wake up some waiting thread (after it has made the condition true):
    - `pthread_cond_signal`
  - A thread can wake up every waiting thread (after it has to made the condition true:
    - `pthread_cond_broadcast`

You must initialize a conditional variable before using with pthread_cond_init(...)

pthread_cond_wait

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

- Atomically releases `mutex` and puts the calling thread to sleep in an internal waiting queue for `cond`.

No other thread can grab the released mutex before calling thread is put to sleep in the waiting queue

- Condition involves some shared state.
  - e.g. the condition "buffer is not full" involves shared state buffer.
- Mutex is the lock protecting access to the condition's shared state.

- On successful return, `mutex` is locked (which the calling thread should unlock later)

pthread_cond_signal

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

- Wake up at least one of the threads blocked on `cond`

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

- Wake up all threads blocked on `cond`.

Pseudo-code

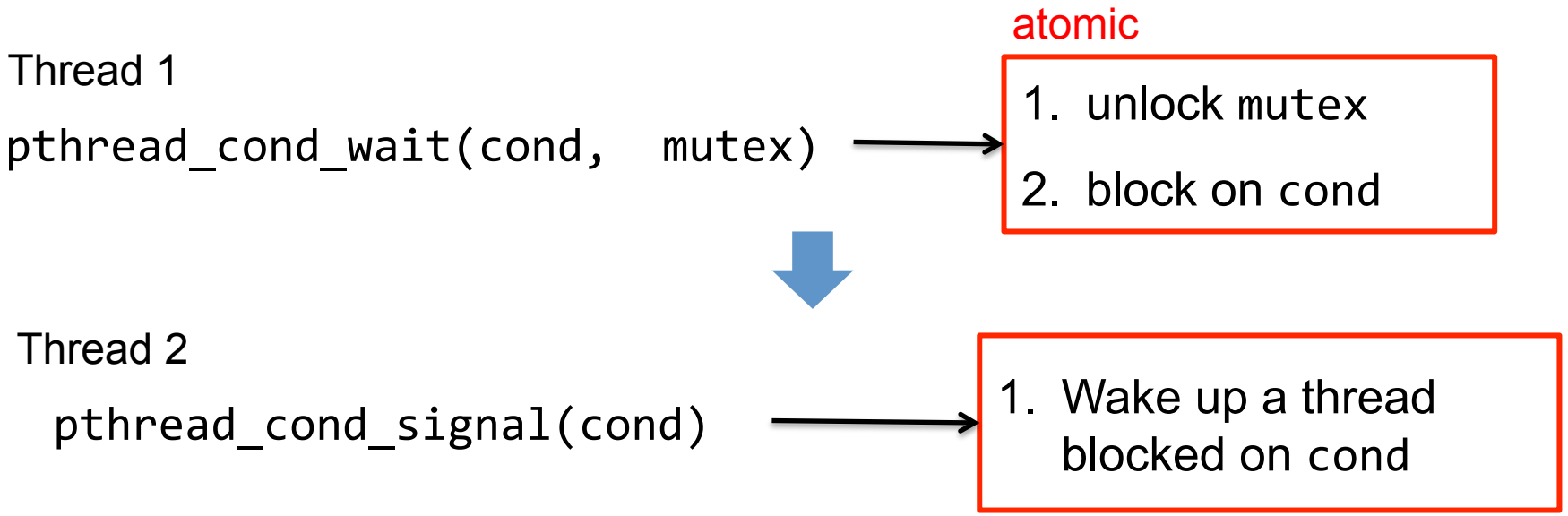
Thread 1

`pthread_cond_wait(cond, mutex)` →

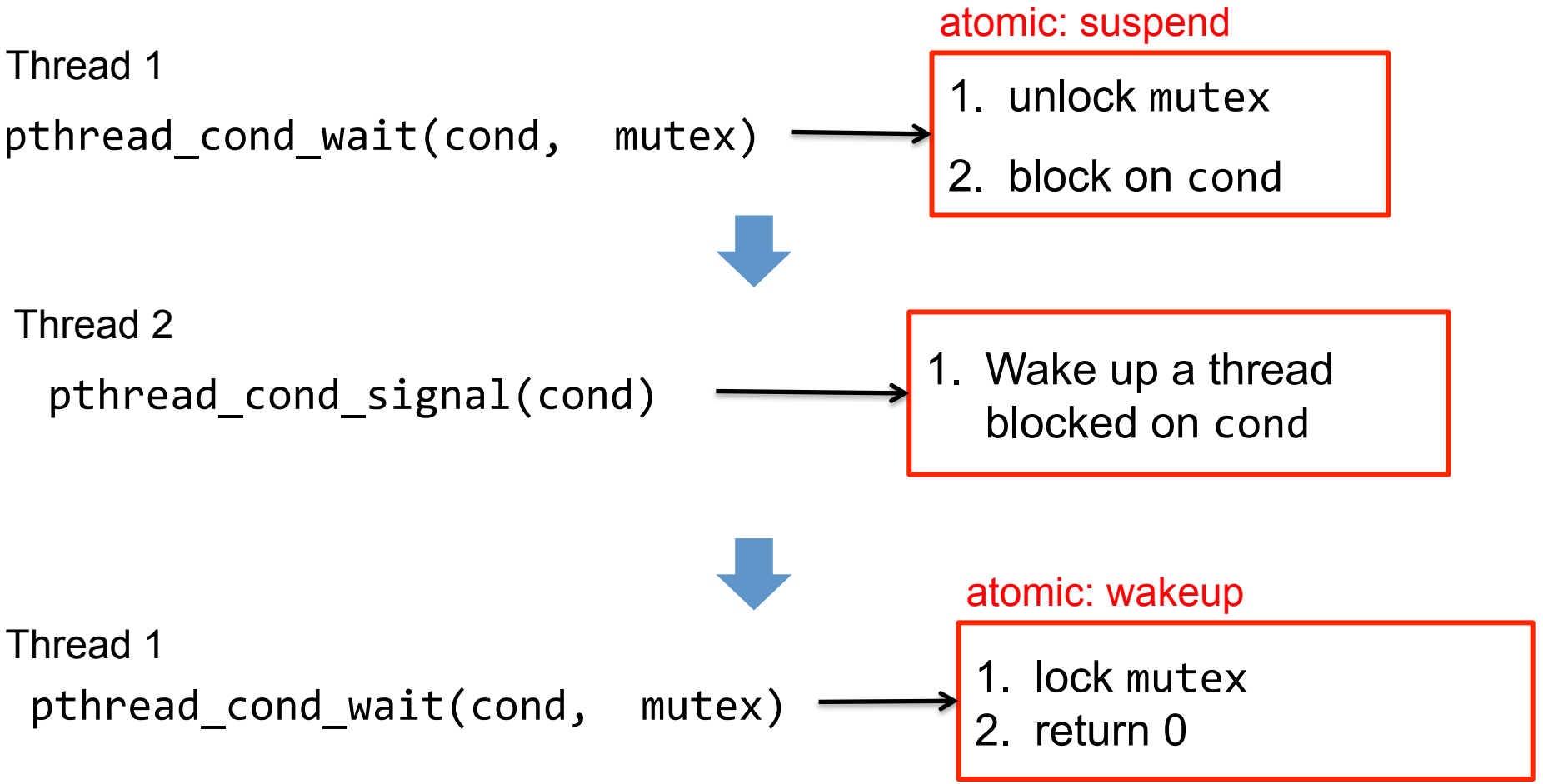
atomic

1. unlock `mutex`
2. block on `cond`

Pseudo-code


atomic
Thread 1
pthread_cond_wait(cond, mutex)
1. unlock mutex
2. block on cond
Thread 2
pthread_cond_signal(cond)
1. Wake up a thread blocked on cond

Pseudo-code


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

Simple Example: hello bye

```
pthread_mutex_t mutex;
bool saidHello = false;
```

mutex protects saidHello

```
void* sayHello(void *arg){

  pthread_mutex_lock(&mutex);
  printf("hello\n");
  saidHello = true;
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

```
void* sayBye(void *arg){
  while (1) {
    pthread_mutex_lock(&mutex);
    if (saidHello) {
     printf("bye\n");
     break;
    }
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
```

Simple example using cond variables

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

mutex protects saidHello

associated with the condition “saidHello is true”

```
void* sayHello(void *arg){

   pthread_mutex_lock(&mutex);
   printf(“hello\n”);
   saidHello = true;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&mutex);
   return NULL;
}
```

```
void* sayBye(void *arg){

   pthread_mutex_lock(&mutex);
   while(!saidHello) {
      pthread_cond_wait(&mutex, &cond);
   }
   printf(“bye\n”);
   pthread_mutex_unlock(&mutex);
   return NULL;
}
```

Common pattern for using cond variables: use while not if

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

void* sayHello(void *arg){

   pthread_mutex_lock(&mutex);
   printf("hello\n");
   saidHello = true;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&mutex);
   return NULL;
}
```

```
void* sayBye(void *arg){

   pthread_mutex_lock(&mutex);
   while(!saidHello) {
      pthread_cond_wait(&mutex, &cond);
   }
   printf("bye\n");
   pthread_mutex_unlock(&mutex);
   return NULL;
}
```

Use "while" instead of "if",
because spurious wakeups from the
`pthread_cond_wait()` may occur.

Common pattern for using cond variables: hold lock while signaling

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

void* sayHello(void *arg){
    pthread_mutex_lock(&mutex);
    printf("hello\n");
    saidHello = true;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

```
void* sayBye(void *arg){
    pthread_mutex_lock(&mutex);
    while(!saidHello) {
        pthread_cond_wait(&mutex, &cond);
    }
    printf("bye\n");
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Why is this incorrect?

Answer: result in "Lost Signal"

while (!saidHello) //saidHello is false

saidHello = true

pthread_cond_signal

pthread_cond_wait

Why must pthread_cond_wait atomically release mutex?

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

void* sayHello(void *arg){

   pthread_mutex_lock(&mutex);
   printf("hello\n");
   saidHello = true;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&mutex);
   return NULL;
}
```

```
void* sayBye(void *arg){

   pthread_mutex_lock(&mutex);
   while(!saidHello) {
      pthread_cond_wait(&mutex, &cond);
      pthread_mutex_unlock(&mutex);
      pthread_cond_sleep(&cond);
   }
   printf("bye\n");
   pthread_mutex_unlock(&mutex);
   return NULL;
}
```

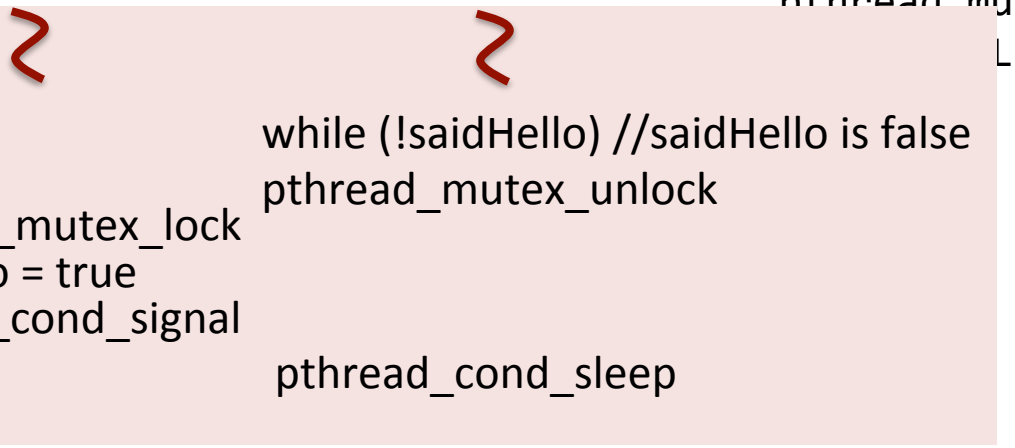
Why atomically release the lock and block calling thread?

Why must pthread_cond_wait atomically release mutex?

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

void* sayHello(void *arg){
   pthread_mutex_lock(&mutex);
   printf("hello\n");
   saidHello = true;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&mutex);
   return NULL;
}
```

```
void* sayBye(void *arg){
   pthread_mutex_lock(&mutex);
   while(!saidHello) {
      pthread_cond_wait(&mutex, &cond);
      pthread_mutex_unlock(&mutex);
      pthread_cond_sleep(&cond);
   }
   printf("bye\n");
   pthread_mutex_unlock(&mutex);
   LL;
```



Lost signal problem!

Back to producer-consumer example

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

buffer_t buf;

void* producer(void *arg){
  int r = random(); //produce data
  while (1) {
    pthread_mutex_lock(&buf.mu);
    if (buf.size < MAX) {
      buf.data[buf.size] = r;
      buf.size++;
      break;
    }
    pthread_mutex_unlock(&buf.mu);
  }
  printf("produced %d\n", r);
  return NULL;
}
```

How to get rid of busy loop in producer? What is the condition that producer must wait for?

```
void* consumer(void *arg){
  int r;
  while(1) {
    pthread_mutex_lock(&buf.mu);
    if (buf.size > 0) {
      r = buf.data[buf.size - 1];
      buf.size--;
      break;
    }
    pthread_mutex_unlock(&buf.mu);
  }
  printf("consumed %d\n", r);
  return NULL;
}
```

Back to producer-consumer example

```
typedef struct {
  int data[MAX];
  int size;
  pthread_mutex_t mu;
  pthread_cond_t notfull;
} buffer_t;

buffer_t buf;

void* producer(void *arg){
  int r = random(); //produce data
  pthread_mutex_lock(&buf.mu);
  while(buf.size == MAX) {
    pthread_cond_wait(&buf.notfull,
                      &buf.mutex);
  }
  buf.data[buf.size] = r;
  buf.size++;
  pthread_mutex_unlock(&buf.mu);
  printf("produced %d\n", r);
  return NULL;
}
```

How to get rid of busy loop in consumer? What is the condition that consumer must wait for?

```
void* consumer(void *arg){
  int r;
  while(1) {
    pthread_mutex_lock(&buf.mu);
    if (buf.size > 0) {
      r = buf.data[buf.size - 1];
      buf.size--;
      pthread_cond_signal(&buf.notfull);
      break;
    }
    pthread_mutex_unlock(&buf.mu);
  }
  printf("consumed %d\n", r);
  return NULL;
}
```

Back to producer-consumer example

```
typedef struct {
  int data[MAX];
  int size;
  pthread_mutex_t mu;
  pthread_cond_t notfull;
  pthread_cond_t notempty;
} buffer_t;
buffer_t buf;

void* producer(void *arg){
  int r = random(); //produce data
  pthread_mutex_lock(&buf.mu);
  while(buf.size == MAX) {
    pthread_cond_wait(&buf.notfull,
                      &buf.mu);
  }
  buf.data[buf.size] = r;
  buf.size++;
  pthread_cond_signal(&buf.notempty);
  pthread_mutex_unlock(&buf.mu);
  printf("produced %d\n", r);
  return NULL;
}
```

```
void* consumer(void *arg){
  int r;
  pthread_mutex_lock(&buf.mu);
  while (buf.size == 0) {
     pthread_cond_wait(&buf.notempty,
                       &buf.mu);
  }
  r = buf.data[buf.size - 1];
  buf.size--;
  pthread_cond_signal(&buf.notfull);
  pthread_mutex_unlock(&buf.mu);
  printf("consumed %d\n", r);
  return NULL;
}
```

Another example: FIFO lock

- pthread_mutex does not provide fairness
  - a latecomer might get lock before an earlier waiter
- Add fairness → FIFO Lock
  - Locks are granted in the order they are requested

How to implement a FIFO lock?

- Design #1: Use one cond per thread. Each thread sleeps on its own cond. Wake up only the thread whose turn it is to grab the lock.
- Design #2: Use one cond for all threads.

Wake up all threads to check their turn. All but one grabs the lock.

Design #1: one cond per waiting thread

```
typedef struct {
  pthread_mutex_t mu;
  bool busy;
  node_t *head;
  node_t *tail;
} lock_t;
```

protect access to struct fields

Status of the lock. True if granted. False if free

A linked list corresponding to the waiting threads

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

each thread to block on one linked list node

indicates whether thread should be blocked or not

```
void fifo_lock_init(lock_t *l) {
   pthread_mutex_init(&l->mu);
   pthread_cond_init(&l->cond);
   l->busy = false;
   l->head = l->tail = NULL;
}
```

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

```
typedef struct {
  pthread_mutex_t mu;
  node_t *head;
  bool busy;
} lock_t;
```

```
int fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mu);
  if(!l->busy) {
    l->busy = true;
    pthread_mutex_unlock(&l->mu);
    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;
  bool blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mu;
  node_t *head, *tail;
  bool busy;
} lock_t;
```

```
int fifo_lock(lock_t *l)
{
  pthread_mutex_lock(&l->mu);
  if(!l->busy) {
    l->busy = true;
    pthread_mutex_unlock(&l->mu);
    return 0;
  }
  // lock is busy, block on a new cond
  node_t *n = malloc(sizeof(node_t));
  pthread_cond_init(&n->cond, NULL);
  n->blocked = true;
  n->next = NULL;
  if(l->head == NULL) {
    l->head = n;
    l->tail = n;
  } else {
    l->tail->next = n;
    l->tail = n;
  }
```

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;
  bool blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mu;
  node_t *head, *tail;
  bool busy;
} lock_t;
```

```
int fifo_lock(lock_t *l) {

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

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;
  bool blocked;
} node_t;
```

```
typedef struct {
  pthread_mutex_t mu;
  node_t *head, *tail;
  bool busy;
} lock_t;
```

```
int fifo_lock(lock_t *l) {

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

```
int fifo_unlock(lock_t *l) {

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

Release Lock

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

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

```
typedef struct {
  pthread_mutex_t mu;
  node_t *head, *tail;
  bool busy;
} lock_t;
```

```
int fifo_lock(lock_t *l) {

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

```
int fifo_unlock(lock_t *l) {

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

```
typedef struct {
  pthread_mutex_t mu;
  node_t *head, *tail;
  bool busy;
} lock_t;
```

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

```
int fifo_unlock(lock_t *l) {

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

Acquire Lock

4. Remove and free the node from the waiting list

```
lock_t l < busy: false, head: null, tail: null >
```

Thread 1

```
int fifo_lock(lock_t *l) {

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

lock_t l < busy: true, head: null, tail: null >

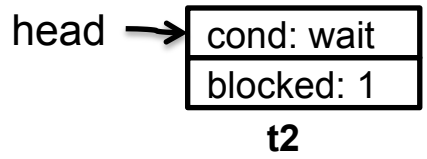
Thread 1

fifo_lock(&l)

```
int fifo_lock(lock_t *l) {

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

```
lock_t l < busy: true, head: t2, tail: t2>
```



Thread 1

```
fifo_lock(&l)
```

Thread 2

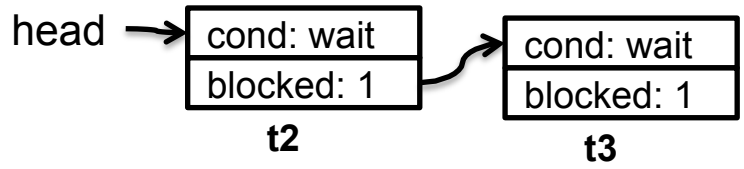
```
fifo_lock(&l)
```

wait and block

```
int fifo_lock(lock_t *l) {

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

lock_t l < busy: true, head: t2, tail: t3>



Thread 1

fifo_lock(&l)

Thread 2

fifo_lock(&l)

wait and block

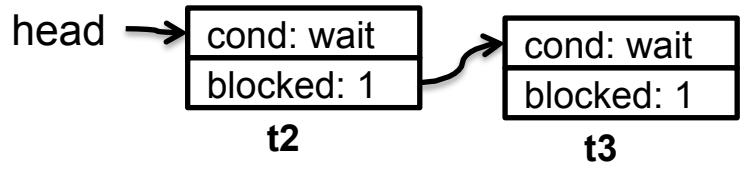
Thread 3

fifo_lock(&l)

```
int fifo_lock(lock_t *l) {

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

lock_t l < busy: true, head: t2, tail: t3>,



Thread 1

Thread 2

Thread 3

fifo_lock(&l)

fifo_lock(&l)

critical section

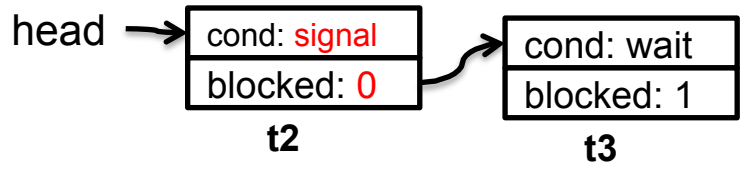
wait and block

fifo_lock(&l)

```
int fifo_lock(lock_t *l) {

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

lock_t l < busy: true, head: t2, tail: t3>



Thread 1 | Thread 2 | Thread 3

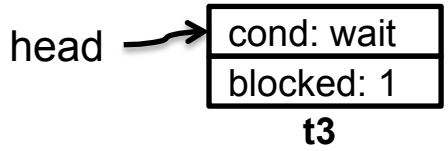
```
fifo_lock(&l)
                        fifo_lock(&l)
 critical section       wait and block        fifo_lock(&l)

fifo_unlock(&l)
```

```
int fifo_unlock(lock_t *l) {

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

```
lock_t l < busy: true, head: t3, tail: t3>
```



Thread 1

```
fifo_lock(&l)

 critical section

fifo_unlock(&l)
```

Thread 2

```
fifo_lock(&l)
```

wait and block

wakeup

```
critical section
```

Thread 3

```
fifo_lock(&l)
```

wait and block

```
int fifo_lock(lock_t *l) {

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

```
typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  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

Design #2: one cond for all waiting threads

- Basic Idea:
  - Assign thread a ticket number.
  - Each thread waits for its turn to get the lock according to its ticket number.

Design #2: one cond for all waiting threads

```
typedef struct {
  pthread_mutex_t mu;
  pthread_cond_t cond;
  unsigned long int turn;
  unsigned long ticker;
} lock_t;
```

whose turn it is to grab the lock now

an ever increasing ticket counter

```
void fifo_lock_init(lock_t *l) {
  pthread_mutex_init(&l->mu);
  pthread_cond_init(&l->cond);
  l->turn = 0;
  l->ticker= 0;
} lock_t;
```

```
typedef struct {
  pthread_mutex_t mu;
  pthread_cond_t cond;
  unsigned long turn, ticker;
} lock_t;
```

```
int fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mu);
  unsigned long ticket = l->ticker++;
  while(ticket != l->turn) {
    pthread_cond_wait(&l->cond, &l->mu);
  }
  pthread_mutex_unlock(&l->mu);

  return 0;
}
```

Acquire a lock

1. Get a ticket from ticker (update ticker)
2. Check if its turn is up by comparing its ticket with l->turn

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

```
int fifo_lock(lock_t *l) {

  pthread_mutex_lock(&l->mu);
  unsigned long ticket = l->ticker++;
  while(ticket != l->turn) {
    pthread_cond_wait(&l->cond, &l->mu);
  }
  pthread_mutex_unlock(&l->mu);

  return 0;
}
```

```
int fifo_unlock(lock_t *l) {

  pthread_mutex_lock(&l->mu);
  l->turn++;
  pthread_cond_broadcast(&l->cond);
  pthread_mutex_unlock(&l->mu);
}
```

Release the lock

1. Increase turn number
2. Wakeup all the waiters

Only one of the waiting threads will see that it's his turn