# Operating System Labs

Yuanbin Wu
CS@ECNU

# Operating System Labs

- Project 3
  - Due: 8 Dec.
- Timeline updated

| | 9.2 | 9 | 16 | 23 | 30 | 10.7 | 14 | 21 | 28 | 11.4 | 11 | 18 | 25 | 12.2 | 9 | 16 | 23 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | w1 | w2 | w3 | w4 | w5 | w6 | w7 | w8 | w9 | w10 | w11 | w12 | w13 | w14 | w15 | w16 | w17 | w18 |
| P0 | L | L | | | | | | | | | | | | | | | | |
| P1 | | | L | L | | | | | | | | | | | | | | |
| P2 | | | | | | L | L | | | | | | | | | | | |
| P3 | | | | | | | | | | | L | L | oral | | | | | |
| P4 | | | | | | | | | | | | | | | L | oral | | oral |

# Operating System Labs

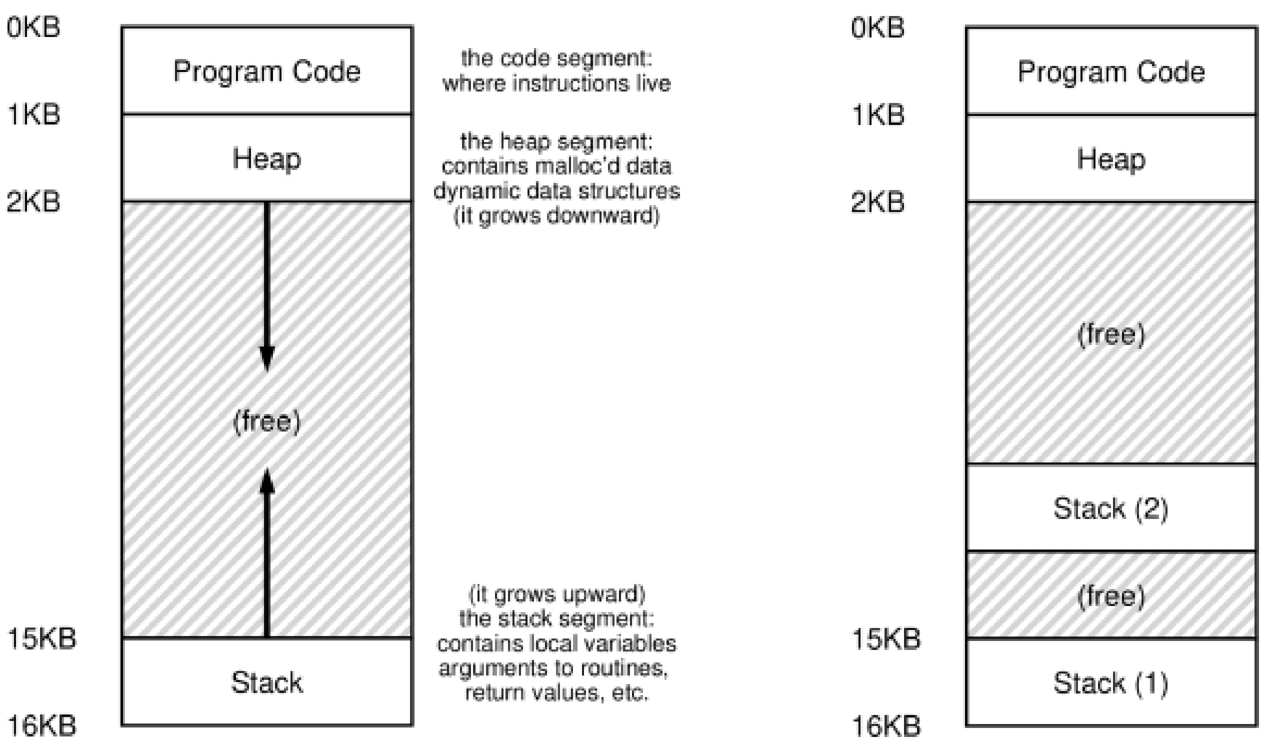
- Overview of concurrency
  - Thread
  - Two scenarios of concurrency control
    - Locks
    - Condition Variables
- Project 3 parta

# Process and Thread

- In Linux, threads are processes with shared address space
  - clone() system call with CLONE_THREAD
- Program Counter (PC)
  - Process: one PC
  - Threads: multiple PCs
- Context switch
  - Process: PCB
  - Thread: TCBs in PCB

# Process and Thread

- Stack
  - Process: one stack
  - Thread: multiple stacks (thread local storage)



# Process and Thread

- Why threads
  - Accelerate performance
    - Multiple processors, multi-core
  - Light Weight
    - Faster creating and managing than processes
  - Efficient communication
    - Shared address space

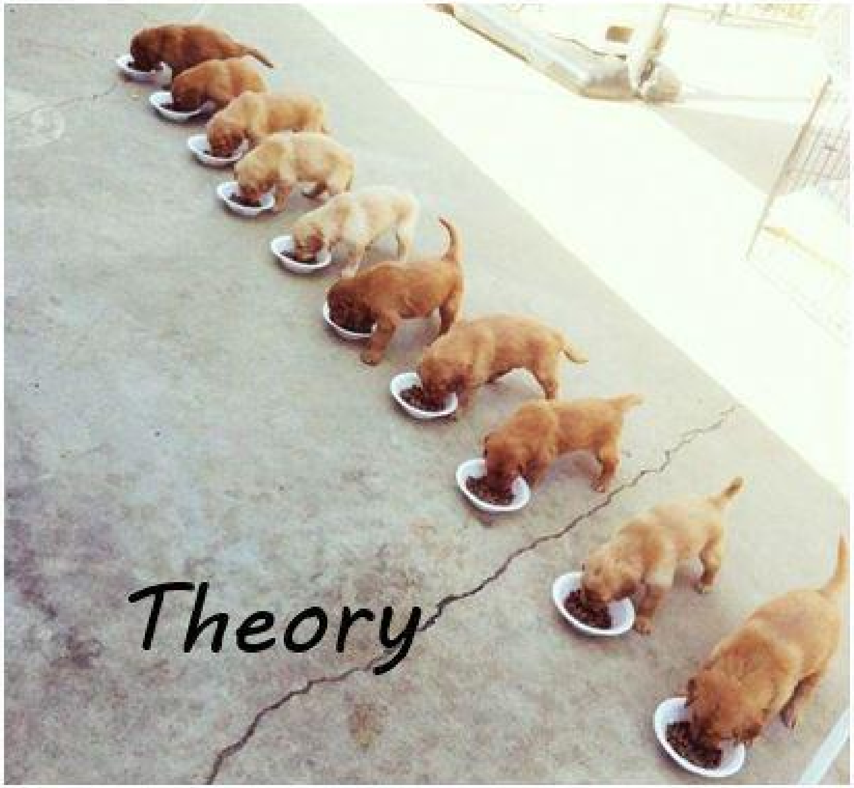
- Example 1: multithread

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

Int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A");
    assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B");
    assert(rc == 0);

    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);

    printf("main: end\n");
    return 0;
}
```

- Example 2: multithread with shared objects

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

static volatile int counter = 0;

void * mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++)
        counter = counter + 1;
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;
    int rc;
    printf("main: begin (counter = %d)\n", counter);
    rc = pthread_create(&p1, NULL, mythread, "A");  assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B");  assert(rc == 0);

    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);

    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

# Multithreaded programming




# Thread

- Terms
  - Race condition
  - Critical section
  - Mutual exclusion

# Thread

- Concurrency Control
  - Where
    - Processes with shared objects
    - Threads
  - How
    - Atomic operations
    - Helps from hardwares: Synchronizing primitives

# Thread

- POSIX Thread API
  - pthread
  - Thread
    - create, control, destroy...
  - Synchronizing primitives
    - Locks
    - Condition variables
    - Semaphore
    - ...

# Thread

- pthread_create()
  - Create a thread

```
#include <pthread.h>

Int pthread_create(
    pthread_t *  thread,                // structure of thread
    const pthread_attr_t * attr,        // thread attributes
    void* (*start_routine)(void*),      // the job
    void *arg                           // arguments of the job
);
```

# Thread

- pthread_join()
  - Wait thread complete

```
#include <pthread.h>

int pthread_join(
    pthread_t thread,        // structure of thread
    void **value_ptr         // return value
);
```

# Thread

- Locks (mutex)

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int rc;

rc = pthread_mutex_lock(&lock);
assert(rc == 0);
x = x + 1;   // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

# Thread

- Conditional variables

```
#include <pthread.h>
int pthread_cond_wait(
    pthread_cond_t      *cond,
    pthread_mutex_t     *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
```

# Thread

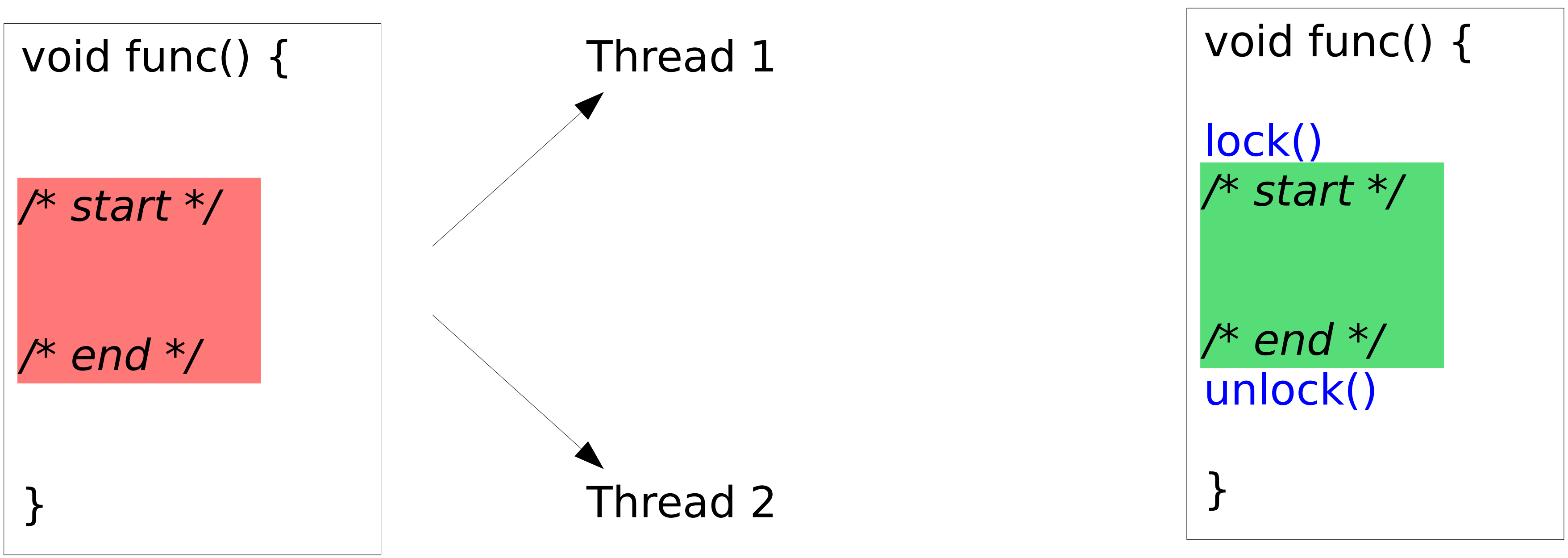
- Summary
  - Processes with shared address space
  - Concurrency control
    - Critical section, race condition
  - POSIX thread library
    - pthread.h

# Two Scenarios in Concurrency Control

- Protect critical sections
  - Lock
  - lock(), unlock()
- Synchronize different threads
  - Conditional variable
  - signal(), wait()

# Two Scenarios in Concurrency Control

- Protect critical sections



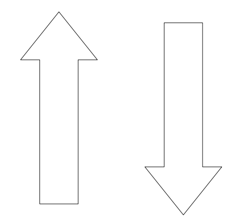
# Two Scenarios in Concurrency Control

- Synchronize threads

```
void func1() {

...
/* wait some condition becomes true*/
...

}
```

```
void func2() {

...
/* set the condition */
...

}
```

Thread 1



Thread 2

Also called
1. Message passing
2. Communication
3. Syncronization

# Two Scenarios in Concurrency Control

- Synchronize threads

```
void func1() {

...
/* wait some condition becomes true*/
wait(condition_variable)
...

}
```

```
void func2() {

...
/* set the condition */
signal(condition_variable)
...

}
```

# Lock

- Lock: a variable

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;  // critical section
unlock(&mutex);
```

  - Around critical sections
  - Two states: free(unlocked) or held(locked)

# Lock

- Two APIs
  - lock():
    - try to acquire the lock
    - If the lock is free (no other thread hold the lock)
      - Get the lock and enter the critical section
    - Won't return while the lock is not free
  - unlock():
    - The state of the lock is changed to free
    - One waiting thread (stuck by lock()) gets the lock

# Lock

- Revoke some control from OS
  - Threads are scheduled by OS
  - Lock provide a way for programmers to break the scheduling

# Lock

- How to implement a lock?
  - Criteria
    - Correctness
    - Fairness
    - Performance

# Lock

- Lock implementation: disable interrupts

```
void lock(){
   disable_interrupts();
}

void unlock(){
  enable_interrupts();
}
```

- Problems
  - Require privileged operations
  - Only works for the single processor case

# Lock

- Lock implementation: using flag

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;            // 0 -> lock is free, 1 -> held
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ;                     // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

# Lock

- Lock implementation: using flag
  - Correctness

| Thread 1 | Thread 2 |
|---|---|
| call `lock()` | |
| while (flag == 1) | |
| **interrupt: switch to Thread 2** | |
| | call `lock()` |
| | while (flag == 1) |
| | flag = 1; |
| | **interrupt: switch to Thread 1** |
| flag = 1; // set flag to 1 (too!) | |

  - Efficiency
    - Waste CPU time

# Lock

- Lock implementation: atomic test-and-set

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr;    // fetch old value at old_ptr
    *old_ptr = new;        // store 'new' into old_ptr
    return old;            // return the old value
}
```

**Atomically!**
**Hardware Instructions**

- In x86
  - xchg

# Lock

- Lock implementation: atomic test-and-set
  - Spin lock
    - Requires a preemptive scheduler

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;          // 0 -> lock is free, 1 -> held
}

void lock(lock_t *mutex) {
    while (TestAndSet(&lock->flag, 1)== 1)
        ;                     // spin-wait (do nothing)
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

# Lock

- Lock implementation: atomic test-and-set
  - **Spin lock**
  - Correctness
    - Yes
  - Fairness
    - No guarantees
  - Performance
    - Spin using CPU cycles
    - Single processor: painful
    - Multiple processors: reasonable

# Lock

- Other hardware primitives
  - test-and-set
  - Compare-and-swap
  - Load-linked and store-conditional
  - Fetch-and-add
    - Fairness: assign a ticket for each waiting thread

# Lock

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)
        ;
    mutex->flag = 1;
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Buggy flag

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (TestAndSet(&lock->flag, 1)== 1)
        ;
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Spin lock

# Lock

- Problems of spin locks
  - Waste cpu time
    - N threads, only 1 hold the lock, other thread will spin
  - No guarantee on fairness (in general)
    - starvation
- How to improve?

# Lock

- Sleep instead of spin
  - Assume an OS primitive yield()
    - Move the caller from running to ready

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (TestAndSet(&lock->flag, 1)== 1)
        yield();
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Problem:
1. cost of context switch is substantia
2. fairness: still not handled

# Lock

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (TestAndSet(&lock->flag, 1)== 1)
        ;
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Spin lock

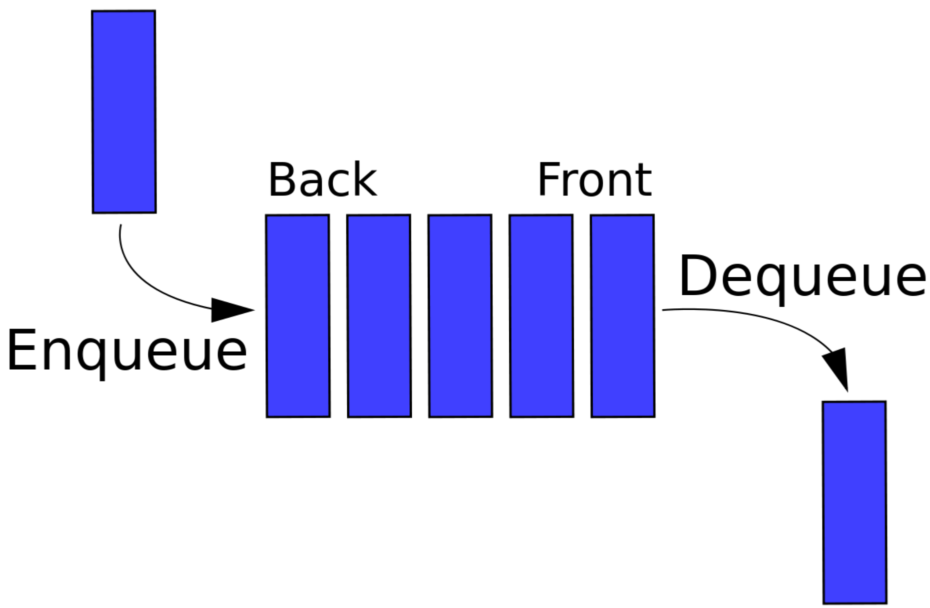
```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (TestAndSet(&lock->flag, 1)== 1)
        yield();
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Sleep instead of spin

Question:
How to improve fairness?

# Lock

- Improve fairness
  - Queue



# Lock

- Improve fairness

```
typedef struct __lock_t { int flag; } lock_t;
queue_t q;

void init(lock_t *mutex) {
    mutex->flag = 0;
    init_queue(q);
}

void lock(lock_t *mutex) {
    while (TestAndSet(&lock->flag, 1)== 1) {
        enqueue(q); // put the thread in q
        yield();
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
    dequeue(q); // wakeup a thread in q
}
```

enqueue/dequeue should be **thread-safe.**

How to?
1. play queue operations in kernel
2. spin lock for the queue operation.

# Lock

- Improve fairness
  - Assume two OS primitives (from Solaris)
    - park(): put the calling thread to sleep
    - unpark(tid): wake a particular thread
    - A parked thread wakes up when an unpark is called.
  - A queue: prevent starvation

```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m)
{
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

Questions:
1. does it avoid spin?
   does guard necessary?
2. can we change order?
3. why no flag=0 in unlock?
4. wakeup/waiting race

```c
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

```c
setpark();
m->guard = 0;
park();
```

```c
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // no one wants it
    else
        m->flag = 0;
        unpark(queue_remove(m->q));
    m->guard = 0;
}
```

# Lock

- sleeping instead of spin + spin on queue
  - Two OS(Soloari) primitives
    - park(), setpark()
    - unpark()
  - Avoid spinning by guard
  - Avoid starvation by a queue

# Lock

- Queue in kernel (the linux way)
  - OS primitive: futex
    - A memory location
    - A in-kernel queue (for every futex)

# Lock

- Futex: wait and wake
  - futex_wait(address, expected)
    - If (*address == expected)
      - put caller in the queue, let it sleep
    - else
      - Return immediately
  - futex_wake(address)
    - Wake one thread waiting for the futex
- Let's see some real code (lowlevellock.h of glibc)

```
void mutex_lock (int *mutex) {
    int v;
    // Bit 31 was clear, we got the mutex (this is the fastpath)
    if (atomic_bit_test_set (mutex, 31) == 0)
        return;
    atomic_increment (mutex);
    while (1) {
        if (atomic_bit_test_set (mutex, 31) == 0) {
            atomic_decrement (mutex);
            return;
        }

        /* We have to wait now. First make sure the futex value
            we are monitoring is truly negative (i.e. locked). */
        v = *mutex;
        if (v >= 0)
            continue;
        futex_wait (mutex, v);
    }
}

void mutex_unlock (int *mutex) {









}
```

# Lock

- Lock implementation: two-phase locks
  - park/futex are system call!
  - Spinning could be useful when the lock is about to release
- Two-phase locks
  - In the first phase: spins for a while
  - In the second phase: sleep and wait
  - Hybrid approach
- Above Linux lock is a two-phase lock
  - Spin only once

Can you implement a two-phase lock in your project?

# Lock

- Lock implementation: summary
  - Disable interrupts
  - Flag
  - Spin locks
    - Test-and-set
    - Compare-and-swap
    - Fetch-and-add
  - Sleep instead of spinning
    - park(), setpark(), unpark()
    - Futex
  - Two-phase locks

# Condition Variables

- Lock
  - Protect critical sections
- Condition Variables
  - A thread wait some condition becomes true
  - Another thread signal changes of the condition
  - wait(), signal()

# Condition Variables

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

# Condition Variables

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    while (done == 0)
        ;
    printf("parent: end\n");
    return 0;
}
```

1. Waste CPU cycles
2. May be incorrect

# Condition Variables

- Definition
  - A CV is an explicit queue
  - Threads put themselves on the queue when some condition is not satisfied, and sleep
  - Some other threads change the condition, and wake one/more threads in the queue
- Two API associated with a CV
  - wait()
  - signal()

# Condition Variables

- POSIX calls (pthread.h)

```
#include <pthread.h>

// declear
pthread_cond_t c;

// wait()
int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);

// signal()
int pthread_cond_signal(pthread_cond_t *c);
```

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;









void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}










int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

pthread_cond_signal(&c)
Wakeup a waiting thread on c

pthread_cond_wait(&c, &m)
1. assume the lock is held
2. put caller at the queue
   release the lock
   let caller sleep
3. when wakeup:
   - re-acquire the lock,
   - pop from the queue

Consider two possibilities:
1. parent first
2. child first

Alternative 1: No “done”

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
```

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
```

```
void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}
```

```
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

```
void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

```
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Alternative 1: No “done”

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

```c
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

Alternative 2: No lock

```c
void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
```

```c
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}
```

```c
void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}
```

```c
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

```c
void thr_join() {
    if (done == 0)
    Pthread_cond_wait(&c);
}
```

```c
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Alternative 2: No lock

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    if (done == 0)
    Pthread_cond_wait(&c);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

# Condition Variables

- Summary
  - A queue
  - Wait() and signal()
  - Hold the lock!

# Project 4

**The Dark Forest of Threads**