Synchronization Examples

Operating Systems
Wenbo Shen

Acknowledgement: some pages are based on the slides from Yajin Zhou

Review

- Why we need synchronization?
- Race condition, critical section
- Requirements: ME, Progress, Bounded waiting, Performance
- Locks: acquire, release
    - implementation: test-and-set, compare-and-swap
- Semaphores: wait and signal, implementation
- Condition variables: wait, signal, broadcast

Classical Synchronization Problems

- Bounded-buffer problem
- Readers-writers problem
- Dining-philosophers problem

Bounded-Buffer Problem

- Two processes, the producer and the consumer share **n** buffers
  - the producer generates data, puts it into the buffer
  - the consumer consumes data by removing it from the buffer
- The problem is to make sure:
  - **the producer won't try to add data into the buffer if it is full**
  - **the consumer won't try to remove data from an empty buffer**
  - also call producer-consumer problem
- Solution:
  - n buffers, each can hold one item
  - semaphore **mutex** initialized to the value **1**
  - semaphore **full-slots** initialized to the value **0**
  - semaphore **empty-slots** initialized to the value **N**

Bounded-Buffer Problem

- The producer process:

```
  do {
 //produce an item
     ...
     wait(empty-slots);
     wait(mutex);
     //add the item to the  buffer
     ...
     signal(mutex);
     signal(full-slots);
  } while (TRUE)
```

Bounded Buffer Problem

- The consumer process:

```
do {
   wait(full-slots);
   wait(mutex);
   //remove an item from  buffer
   …
   signal(mutex);
   signal(empty-slots);
   //consume the item
   …
} while (TRUE);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
    - readers: only read the data set; they do not perform any updates
    - writers: can both read and write
- The readers-writers problem:
    - allow multiple readers to read at the same time (**shared access**)
    - only one single writer can access the shared data (**exclusive access**)
- Solution:
    - semaphore **mutex** initialized to 1
    - semaphore **write** initialized to 1
    - integer **read_count** initialized to 0

Readers-Writers Problem

- The writer process

```
do {
   wait(write);
   //write the shared data
   ...
   signal(write);
} while (TRUE);
```

Readers-Writers Problem

- The structure of a reader process

```
do {
        wait(mutex);
        readcount++ ;
        if (readcount == 1)
            wait(write) ;
        signal(mutex)

        //reading data
        ...
        wait(mutex) ;
        readcount--;
        if (readcount == 0)
            signal(write) ;
        signal(mutex) ;
} while(TRUE);
```

Readers-Writers Problem

- The structure of a reader process

```
do {
        wait(mutex);
        readcount++ ;
        if (readcount == 1)        //first reader
             wait(write) ;         //block write
        signal(mutex)

        //reading data
        ...
        wait(mutex) ;
        readcount--;
        if (readcount == 0)
             signal(write) ;
        signal(mutex) ;
} while(TRUE);
```

Readers-Writers Problem Variations

- Two variations of readers-writers problem (different **priority** policy)
  - Reader first
    - no reader kept waiting unless writer is updating data
    - If reader holds data, new reader just moves on and reads
    - writer may starve
  - Writer first
    - once writer is ready, it performs write ASAP
    - If reader holds data, new reader will wait for suspended writer
- Which variation is implemented by the previous code example???

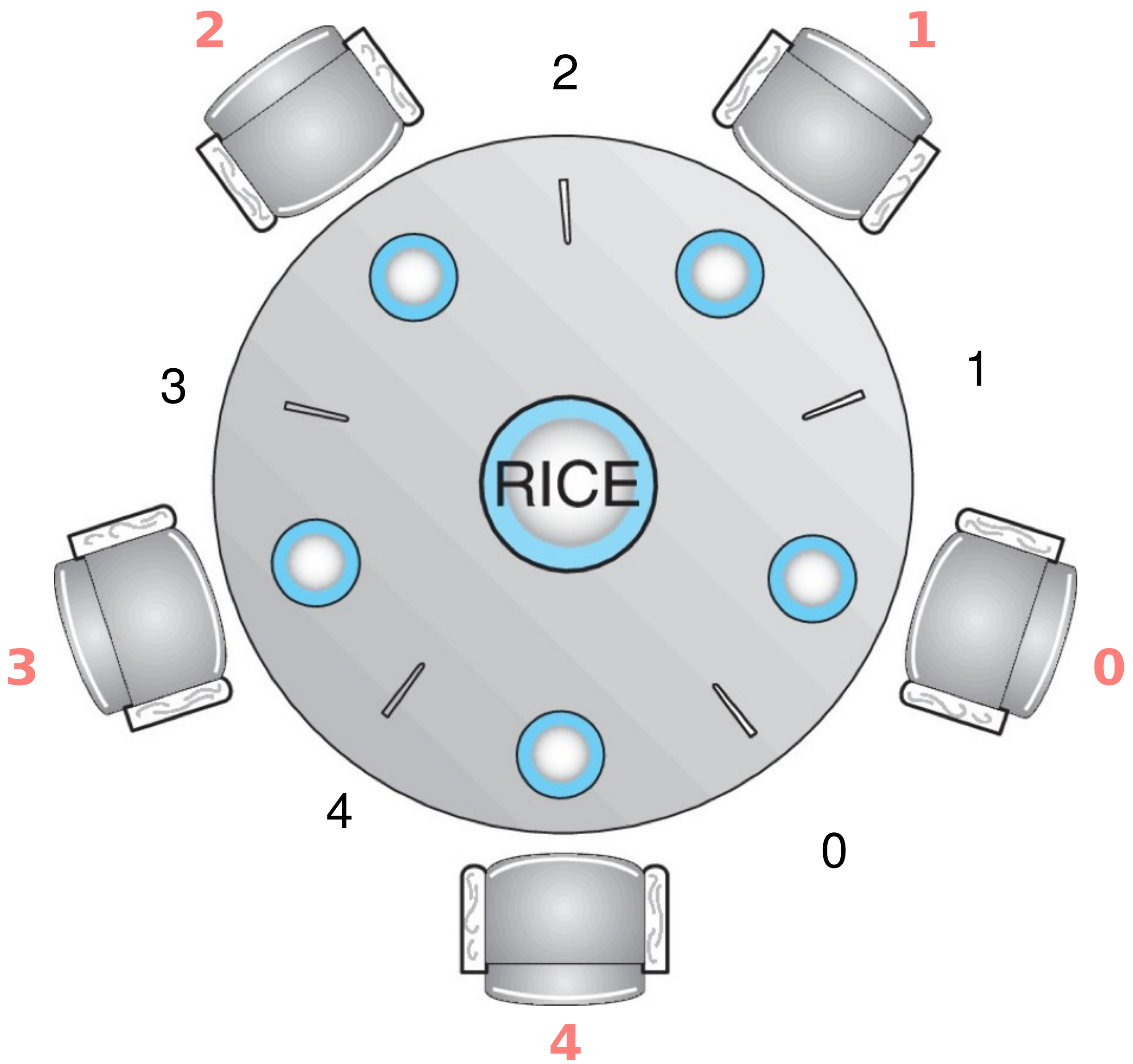
Readers-Writers Problem Variations

- Which variation is implemented by the previous code example???
  - Reader first
- Both variation may have starvation leading to even more variations
  - If writer is in CS and n readers are waiting, one is on **write**, and n-1 are on **mutex**

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
  - they sit in a round table, but don't interact with each other
- They occasionally try to pick up 2 chopsticks (one at a time) to eat
  - one chopstick between each adjacent two philosophers
  - need both chopsticks to eat, then release both when done
  - Dining-philosopher problem represents **multi-resource synchronization**
- Solution (assuming **5 philosophers**):
  - semaphore **chopstick[5]** initialized to 1

Dining-Philosophers Problem



http://web.eecs.utk.edu/~mbeck/classes/cs560/560/notes/Dphil/lecture.html

Dining-Philosophers Problem

- Philosopher i (out of 5):

```
do  {
   wait(chopstick[i]);
   wait(chopStick[(i+1)%5]);
   eat
   signal(chopstick[i]);
   signal(chopstick[(i+1)%5]);
   think
} while (TRUE);
```

- What is the problem with this algorithm?
  - **deadlock**

Dining-Philosophers Problem in Practice

```
void *philosopher(void *v)
{
  Phil_struct *ps;
  int st;
  int t;

  ps = (Phil_struct *) v;

  while(1) {

    /* First the philosopher thinks for a random number of seconds */
    ...

    /* Now, the philosopher wakes up and wants to eat.  He calls pickup
       to pick up the chopsticks */
    ...
    pickup(ps);
    ...

    /* When pickup returns, the philosopher can eat for a random number of
       seconds */
    ...

    /* Finally, the philosopher is done eating, and calls putdown to
       put down the chopsticks */
    ...
    putdown(ps);
  }
}
```

Solution 1: do nothing

```
void pickup(Phil_struct *ps)
{
  return;
}

void putdown(Phil_struct *ps)
{
  return;
}
```

```
0 Philosopher 0 thinking for 2 seconds
0 Total blocktime:     0 :     0     0     0     0     0
0 Philosopher 4 thinking for 2 seconds
0 Philosopher 3 thinking for 1 second
0 Philosopher 1 thinking for 2 seconds
0 Philosopher 2 thinking for 1 second
1 Philosopher 3 no longer thinking -- calling pickup()
1 Philosopher 3 eating for 2 seconds
1 Philosopher 2 no longer thinking -- calling pickup()
1 Philosopher 2 eating for 1 second
2 Philosopher 4 no longer thinking -- calling pickup()
2 Philosopher 4 eating for 1 second
2 Philosopher 1 no longer thinking -- calling pickup()
2 Philosopher 1 eating for 2 seconds
2 Philosopher 0 no longer thinking -- calling pickup()
2 Philosopher 0 eating for 1 second
2 Philosopher 2 no longer eating -- calling putdown()
2 Philosopher 2 thinking for 1 second
3 Philosopher 3 no longer eating -- calling putdown()
3 Philosopher 3 thinking for 1 second
3 Philosopher 2 no longer thinking -- calling pickup()
3 Philosopher 2 eating for 2 seconds
3 Philosopher 0 no longer eating -- calling putdown()
3 Philosopher 0 thinking for 2 seconds
3 Philosopher 4 no longer eating -- calling putdown()
3 Philosopher 4 thinking for 2 seconds
```

P2 and p3 cannot
eat at the same time!

Solution 2: A mutex for each chopstick

```
void pickup(Phil_struct *ps)
{
  Sticks *pp;
  int i;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  pthread_mutex_lock(pp->lock[ps->id]);       /* lock up left stick */
  pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]); /* lock up right stick */
}
```

```
void putdown(Phil_struct *ps)
{
  Sticks *pp;
  int i;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]); /* unlock right stick */
  pthread_mutex_unlock(pp->lock[ps->id]);  /* unlock left stick */
}
```

Solution 2: A mutex for each chopstick

```
0 Total blocktime:     0 :     0     0     0     0     0
0 Philosopher 0 thinking for 2 seconds
0 Philosopher 1 thinking for 2 seconds
0 Philosopher 2 thinking for 1 second
0 Philosopher 3 thinking for 2 seconds
0 Philosopher 4 thinking for 1 second
1 Philosopher 2 no longer thinking -- calling pickup()
1 Philosopher 2 eating for 2 seconds
1 Philosopher 4 no longer thinking -- calling pickup()
1 Philosopher 4 eating for 1 second
2 Philosopher 0 no longer thinking -- calling pickup()
2 Philosopher 1 no longer thinking -- calling pickup()
2 Philosopher 3 no longer thinking -- calling pickup()
2 Philosopher 4 no longer eating -- calling putdown()
2 Philosopher 4 thinking for 1 second
3 Philosopher 2 no longer eating -- calling putdown()
3 Philosopher 2 thinking for 2 seconds
3 Philosopher 1 eating for 2 seconds
3 Philosopher 3 eating for 2 seconds
3 Philosopher 4 no longer thinking -- calling pickup()
5 Philosopher 3 no longer eating -- calling putdown()
5 Philosopher 3 thinking for 1 second
5 Philosopher 1 no longer eating -- calling putdown()
5 Philosopher 1 thinking for 1 second
```

Could be deadlock, but …

Solution 3: Show how deadlock occurs

```
void pickup(Phil_struct *ps)
{
  Sticks *pp;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  pthread_mutex_lock(pp->lock[ps->id]);       /* lock up left stick */
  sleep(3);
  pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]); /* lock up right stick */
}
```

```
 0 Philosopher 0 thinking for 1 second
 0 Philosopher 2 thinking for 3 seconds
 0 Philosopher 3 thinking for 1 second
 0 Philosopher 4 thinking for 2 seconds
 0 Philosopher 1 thinking for 1 second
 0 Total blocktime:     0 :     0     0     0     0     0
 1 Philosopher 3 no longer thinking -- calling pickup()
 1 Philosopher 1 no longer thinking -- calling pickup()
 1 Philosopher 0 no longer thinking -- calling pickup()
 2 Philosopher 4 no longer thinking -- calling pickup()
 3 Philosopher 2 no longer thinking -- calling pickup()
10 Total blocktime:    42 :     9     9     7     9     8
```

Solution 4: An asymmetrical solution

- only odd philosophers start left-hand first, and even philosophers start right-hand first. This does not deadlock.

```
void pickup(Phil_struct *ps)
{
  Sticks *pp;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  if (ps->id % 2 == 1) {
    pthread_mutex_lock(pp->lock[ps->id]);        /* lock up left stick */
    pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]); /* lock right stick */
  } else {
    pthread_mutex_lock(pp->lock[(ps->id+1)%phil_count]); /* lock right stick */
    pthread_mutex_lock(pp->lock[ps->id]);        /* lock up left stick */
  }
}

void putdown(Phil_struct *ps)
{
  Sticks *pp;
  int i;
  int phil_count;

  pp = (Sticks *) ps->v;
  phil_count = pp->phil_count;

  if (ps->id % 2 == 1) {
    pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]); /* unlock right stick */
    pthread_mutex_unlock(pp->lock[ps->id]);  /* unlock left stick */
  } else {
    pthread_mutex_unlock(pp->lock[ps->id]);  /* unlock left stick */
    pthread_mutex_unlock(pp->lock[(ps->id+1)%phil_count]); /* unlock right stick */
  }
}
```

Linux Synchronization

- Linux:
  - prior to version 2.6, disables interrupts to implement short critical sections
  - version 2.6 and later, fully preemptive
- Linux provides:
  - **atomic integers**
  - **spinlocks**
  - **semaphores**
    - on single-cpu system, spinlocks replaced by enabling/disabling kernel preemption
  - **reader-writer locks**

Linux Synchronization

- Atomic variables
  - linux/include/linux/atomic.h
  - linux/include/asm-generic/atomic.h
  - atomic_t is the type for atomic integer
- Simple operations
  - atomic_read(), atomic_set(), …
- Arithmetic
  - atomic_{add,sub,inc,dec}()
  - atomic_{add,sub,inc,dec}_return{,_relaxed,_acquire,_release}()
  - atomic_fetch_{add,sub,inc,dec}{,_relaxed,_acquire,_release}()

https://github.com/torvalds/linux/blob/master/Documentation/atomic_t.txt

Linux Synchronization

- Bitwise
  - atomic_{and,or,xor,andnot}()
  - atomic_fetch_{and,or,xor,andnot}{,_relaxed,_acquire,_release}()
- Swap
  - atomic_xchg{,_relaxed,_acquire,_release}()
  - atomic_cmpxchg{,_relaxed,_acquire,_release}()
  - atomic_try_cmpxchg{,_relaxed,_acquire,_release}()
- Others
  - Reference count, Misc

https://github.com/torvalds/linux/blob/master/Documentation/atomic_t.txt

Linux Synchronization

- Spinlock
  - linux/include/linux/spinlock.h
  - Multiple lock/unlock operations supported

- Linux spinlock – x86 implementation

```
ffffffff81a35c00 <_raw_spin_lock>:
ffffffff81a35c00:       31 c0                   xor    %eax,%eax
ffffffff81a35c02:       ba 01 00 00 00          mov    $0x1,%edx
ffffffff81a35c07:       f0 0f b1 17             lock cmpxchg %edx,(%rdi)
ffffffff81a35c0b:       75 02                   jne    ffffffff81a35c0f <_raw_spin_lock+0xf>
ffffffff81a35c0d:       f3 c3                   repz retq
ffffffff81a35c0f:       89 c6                   mov    %eax,%esi
ffffffff81a35c11:       e9 5a f8 66 ff          jmpq   ffffffff810a5470 <queued_spin_lock_slowpath>
ffffffff81a35c16:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
ffffffff81a35c1d:       00 00 00
```

Linux Synchronization

- Linux spinlock – ARM64 implementation

```
ffffff8008a98e78 <_raw_spin_lock>:
ffffff8008a98e78:       a9bf7bfd        stp     x29, x30, [sp, #-16]!
ffffff8008a98e7c:       aa0003e3        mov     x3, x0
ffffff8008a98e80:       d5384102        mrs     x2, sp_el0
ffffff8008a98e84:       910003fd        mov     x29, sp
ffffff8008a98e88:       b9401041        ldr     w1, [x2, #16]
ffffff8008a98e8c:       11000421        add     w1, w1, #0x1
ffffff8008a98e90:       b9001041        str     w1, [x2, #16]
ffffff8008a98e94:       d2800001        mov     x1, #0x0                        // #0
ffffff8008a98e98:       d2800022        mov     x2, #0x1                        // #1
ffffff8008a98e9c:       97ff8f53        bl      ffffff8008a7cbe8 <__ll_sc___cmpxchg_case_acq_4>
ffffff8008a98ea0:       d503201f        nop
ffffff8008a98ea4:       d503201f        nop
ffffff8008a98ea8:       35000060        cbnz    w0, ffffff8008a98eb4 <_raw_spin_lock+0x3c>
ffffff8008a98eac:       a8c17bfd        ldp     x29, x30, [sp], #16
ffffff8008a98eb0:       d65f03c0        ret
ffffff8008a98eb4:       aa0003e1        mov     x1, x0
ffffff8008a98eb8:       aa0303e0        mov     x0, x3
ffffff8008a98ebc:       97d98c51        bl      ffffff80080fc000 <queued_spin_lock_slowpath>
ffffff8008a98ec0:       a8c17bfd        ldp     x29, x30, [sp], #16
ffffff8008a98ec4:       d65f03c0        ret
```

```
ffffff8008a7cbe8 <__ll_sc___cmpxchg_case_acq_4>:
ffffff8008a7cbe8:       f9800011        prfm    pstl1strm, [x0]
ffffff8008a7cbec:       885ffc10        ldaxr   w16, [x0]
ffffff8008a7cbf0:       4a010211        eor     w17, w16, w1
ffffff8008a7cbf4:       35000071        cbnz    w17, ffffff8008a7cc00 <__ll_sc___cmpxchg_case_acq_4+0x18>
ffffff8008a7cbf8:       88117c02        stxr    w17, w2, [x0]
ffffff8008a7cbfc:       35ffff91        cbnz    w17, ffffff8008a7cbec <__ll_sc___cmpxchg_case_acq_4+0x4>
ffffff8008a7cc00:       aa1003e0        mov     x0, x16
ffffff8008a7cc04:       d65f03c0        ret
```

Linux Synchronization

- Semaphore
  - linux/include/linux/semaphore.h

```
struct semaphore {
        raw_spinlock_t          lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

  - down(), up()

Linux Synchronization

- Semaphore
  - down()

```
void down(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

- Sleep when holding spinlock?
  - Check __down_common()
  - No, spinlock is released before sleeping

Linux Synchronization

- Semaphore
  - up()

```
void up(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);
```

Linux Synchronization

- reader-writer locks
  - rw_semaphore
  - RCU (from wiki)
    - **read-copy-update** (**RCU**) is a synchronization mechanism based on mutual exclusion. It is used when performance of reads is crucial and is an example of space–time tradeoff, enabling fast operations at the cost of more space.
    - Read-copy-update allows multiple threads to efficiently read from shared memory by deferring updates after pre-existing reads to a later time while simultaneously marking the data, ensuring new readers will read the updated data.
    - This makes all readers proceed as if there were no synchronization involved, hence they will be fast, but also making updates more difficult.

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.

POSIX Named Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

Condition Variables

- **`condition x;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Condition Variables

- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition);
- some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition).
- The idea goes back to Dijkstra's use of "private semaphores";
- a similar idea was later named a "condition variable" by Hoare.

POSIX Condition Variables

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX Condition Variables

- Thread waiting for the condition a == b to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

release lock when wait
acquire lock when being signaled

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Takeaway

- Bounded-buffer problem
- Readers-writers problem
- Dining-philosophers problem
- Linux provides:
    - atomic integers
    - spinlocks
    - semaphores
    - reader-writer locks