Synchronization Examples

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: some pages are based on the slides from Zhi Wang(fsu).

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 its 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** initialized to the value **0**
  - semaphore **empty** initialized to the value **N**

Bounded-Buffer Problem

- The producer process:

```
do {
   //produce an item

   …

   wait(empty);

   wait(mutex);

   //add the item to the  buffer

   …

   signal(mutex);

   signal(full);

} while (TRUE)
```

Bounded Buffer Problem

- The consumer process:

```
do {
   wait(full);
   wait(mutex);
   //remove an item from  buffer
   …
   signal(mutex);
   signal(empty);
   //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 **wrt** initialized to 1
  - integer **read_count** initialized to 0

Readers-Writers Problem

- The writer process

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

Readers-Writers Problem

- The structure of a reader process

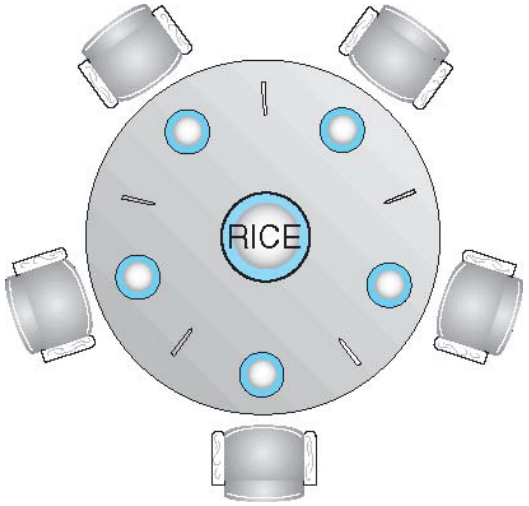
```
do {
   wait(mutex);
   readcount++ ;
   if (readcount == 1)
       wait(wrt) ;
   signal(mutex)

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

Readers-Writers Problem Variations

- Two variations of readers-writers problem (different **priority** policy)
  - no reader kept waiting unless writer is updating data
  - once writer is ready, it performs write ASAP
- Which variation is implemented by the previous code example???
- Both variation may have starvation leading to even more variations
- If writer is in CS and n readers are waiting, one is on wrt, 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

- 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**

Linux Synchronization

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

Linux Synchronization

- Atomic variables
  - atomic_t is the type for atomic integer
- Consider the variables
  - atomic_t counter;
  - int value;

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

- How?

```
static inline int fetch_and_add(int* variable, int value)
{
    __asm__ volatile("lock; xaddl %0, %1"
      : "+r" (value), "+m" (*variable) // input+output
      : // No input-only
      : "memory"
    );
    return value;
}
```

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);
```

POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, 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:

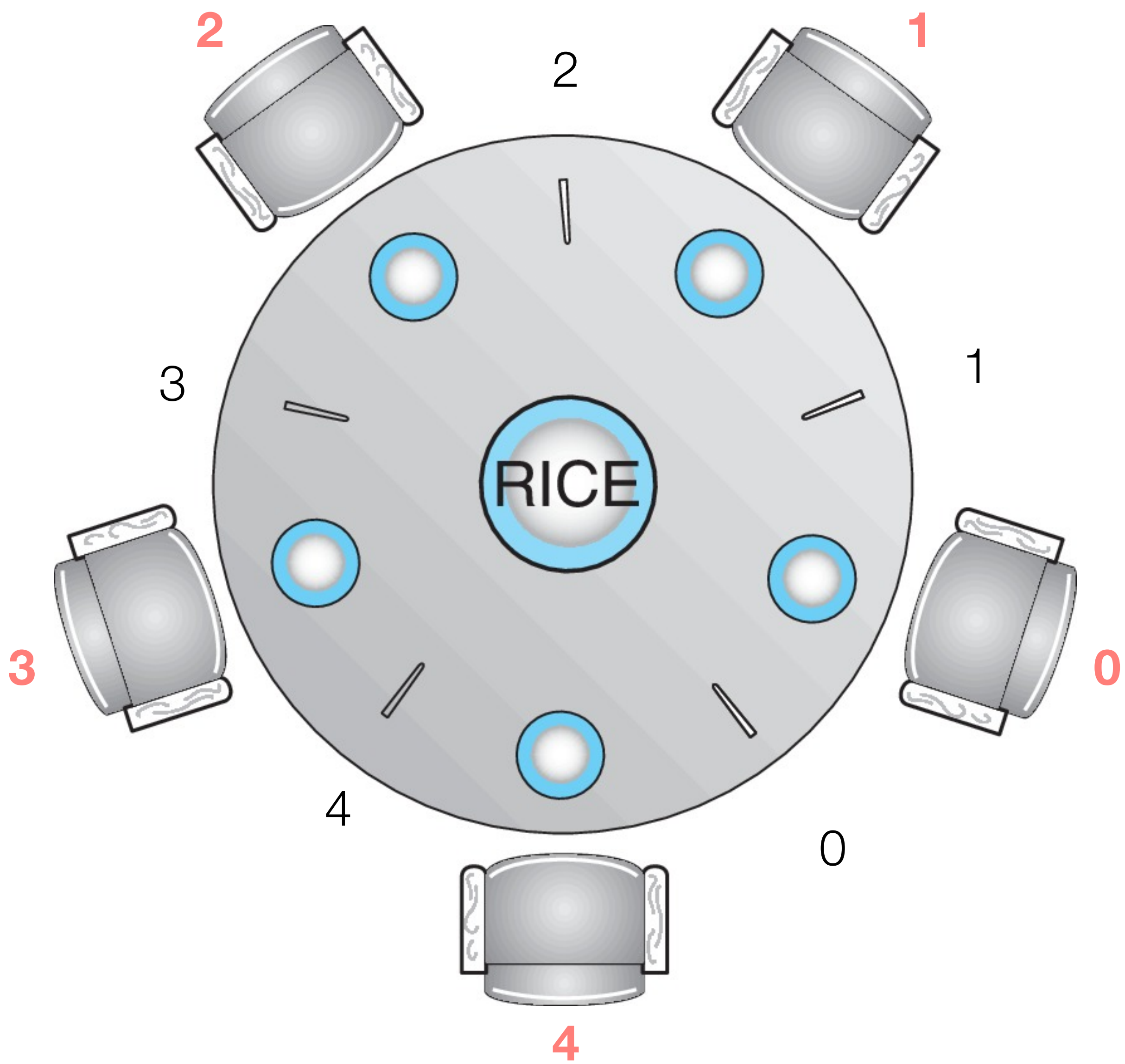
```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

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);
```

Dining-Philosophers Problem in Practice



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

```
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 */
  }
}
```