CS323 Operating Systems
Process Synchronization

Yuanyuan Zhou
Lecture 10
2/12/2003

Content of this lecture

- Administrative announcements
- Form groups
  - this lecture: many group discussions
- Summary

Administrative

- None

Review

- Semaphores
  - The Down (P) operation is used to acquire a resource and decrements count.
  - The Up (V) operation is used to release a resource and increments count
  - Atomic (Indivisible)
  - Counter-based or binary
  - Implementations:
    - What is Spinlock? What is blocked-lock?
    - Tradeoff?
- Monitor
  - Only one process can enter it

Classic Problems

- Producer-Consumer problem ✓
- Bounded buffer problem
- First Reader-writer problem
- Dining philosophers problem
- Sleeping Barber Problem

Bounded Buffer Problem

- Group discussion(2 minute)
  - A producer: in an infinite loop and produce one item each iteration into the buffer
  - A consumer: in an infinite look and consumes one item each iteration from the buffer
  - Buffer size: can only hold at most N items
- Show it on white-board

First Reader-Writer Problem

- A reader: read data
- A writer: write data
- Rule:
  - Multiple readers can read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot in critical section together.
- Locking table: whether any two can be in the critical section simultaneously

	Reader	Writer
Reader	OK	No
Writer	NO	No

8 2/8/2003 CS 323 - Operating Systems, Yuanyuan Zhou

First Reader-Writer Solution

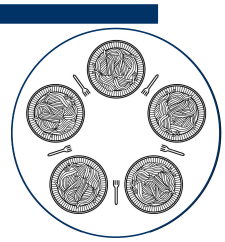
- Group discussion (2 minutes)
- Does it work? Why?
- Problem with this solution

```
Semaphore mutex, wrt; // shared and initialized to 1;
int readcount;        // shared and initialized to 0

// Writer                          // Reader

                                   wait(mutex);
                                   readcount:=readcount+1;
  wait(wrt);                         if readcount == 1 then wait(wrt);
  ......                             signal(mutex);
  writing performed                  ....
  .....                              reading performed
                                     wait(mutex);
  signal(wrt);                       readcount:=readcount-1;
                                     if readcount == 0 then signal(wrt);
                                     signal(mutex);
```

9 2/8/2003 CS 323 - Operating Systems, Yuanyuan Zhou

Dining Philosophers: an intellectual game

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- Possible deadlock?
- How to prevent deadlock?



10 2/8/2003 CS 323 - Operating Systems, Yuanyuan Zhou

Does it solve the Dining Philosophers Problem?

```
#define N 5                         /* number of philosophers */

void philosopher(int i)             /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                   /* philosopher is thinking */
        take_fork(i);               /* take left fork */
        take_fork((i+1) % N);       /* take right fork; % is modulo operator */
        eat( );                     /* yum-yum, spaghetti */
        put_fork(i);                /* put left fork back on the table */
        put_fork((i+1) % N);        /* put right fork back on the table */
    }
}
```

A <u>non</u>solution to the dining philosophers problem

11 2/8/2003 CS 323 - Operating Systems, Yuanyuan Zhou

Dining Philosophers Solution

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N−1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}
```

12 2/8/2003 CS 323 - Operating Systems, Yuanyuan Zhou

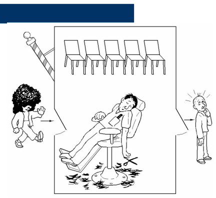
Dining Philosophers Solution

```
void take_forks(int i)          /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}
void put_forks(i)               /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void test(i)                    /* i: philosopher number, from 0 to N−1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

13 2/8/2003 CS 323 - Operating Systems, Yuanyuan Zhou

The Sleeping Barber Problem

- N customer Chair
- One barber can cut one customer's hair at any time
- No customer, goes to sleep



- Group discussion (2 minutes)
  - Solution in the textbook
  - Explain it

The Sleeping Barber Solution (1)

```
#define CHAIRS 5                /* # chairs for waiting customers */

typedef int semaphore;          /* use your imagination */

semaphore customers = 0;        /* # of customers waiting for service */
semaphore barbers = 0;          /* # of barbers waiting for customers */
semaphore mutex = 1;            /* for mutual exclusion */
int waiting = 0;                /* customers are waiting (not being cut) */
```


The Sleeping Barber Solution (2)

```
void barber(void)
{
    while (TRUE) {
        down(&customers);       /* go to sleep if # of customers is 0 */
        down(&mutex);           /* acquire access to 'waiting' */
        waiting = waiting – 1;  /* decrement count of waiting customers */
        up(&barbers);           /* one barber is now ready to cut hair */
        up(&mutex);             /* release 'waiting' */
        cut_hair( );            /* cut hair (outside critical region) */
    }
}
```


The Sleeping Barber Solution (3)

```
void customer(void)
{
    down(&mutex);               /* enter critical region */
    if (waiting < CHAIRS) {     /* if there are no free chairs, leave */
        waiting = waiting + 1;  /* increment count of waiting customers */
        up(&customers);         /* wake up barber if necessary */
        up(&mutex);             /* release access to 'waiting' */
        down(&barbers);         /* go to sleep if # of free barbers is 0 */
        get_haircut( );         /* be seated and be serviced */
    } else {
        up(&mutex);             /* shop is full; do not wait */
    }
}
```


Message Passing

- Send (destination, &message)
- Receive (source, &message)
- Message size: *Fixed* or *Variable* size.
- Real life analogy: conversation

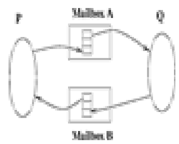
Message Passing

```
#define N 100                           /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                          /* message buffer */

    while (TRUE) {
        item = produce_item( );         /* generate something to put in buffer */
        receive(consumer, &m);          /* wait for an empty to arrive */
        build_message(&m, item);        /* construct a message to send */
        send(consumer, &m);             /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);          /* get message containing item */
        item = extract_item(&m);        /* extract item from message */
        send(producer, &m);             /* send back empty reply */
        consume_item(item);             /* do something with the item */
    }
}
```

Indirect Communication

send(A,message) /* send a message to mailbox A */

receive(A,message) /* receive a message from mailbox A */



- Mailbox is an abstract object into which a message can be placed to or removed from.

Advantage with Indirect Communication

- Allows greater variety of schemes:
  - two processes per link
  - 1 link per pair processes
  - uni or bidirectional
  - allow 1 process to receive a message from a link
  - allow 1 process to all receive a message from a link

Reminder

- Next lecture: Synchronization (chapter 2.3 & 2.4)
- Quiz1 due today at 5pm(only 1 try)
- MP1