操作系统原理与设计

第6章 Processe Synchronization2（进程同步2）

陈香兰

中国科学技术大学计算机学院

2009年10月28日

提纲

Outline

Classical Problems of Synchronization

- Use semaphores to solve
  - Bounded-Buffer Problem，生产者-消费者问题（PC Problem）
  - Readers and Writers Problem，读者-写者问题
  - Dining-Philosophers Problem ，哲学家就餐问题

Solution to Bounded-Buffer Problem I

- 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.

- The structure of the producer process

```
while (true) {
        // produce an item
    wait (empty);
    wait (mutex);
        // add the item to the buffer
    signal (mutex);
    signal (full);
}
```

- The structure of the consumer process

```
while (true) {
    wait (full);
    wait (mutex);
        // remove an item from buffer
    signal (mutex);
    signal (empty);
        // consume the removed item
}
```

Sulotion to Readers-Writers Problem I

- 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.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore **mutex** initialized to 1.
  - Semaphore **wrt** initialized to 1.
  - Integer **readcount** initialized to 0.

Sulotion to Readers-Writers Problem II

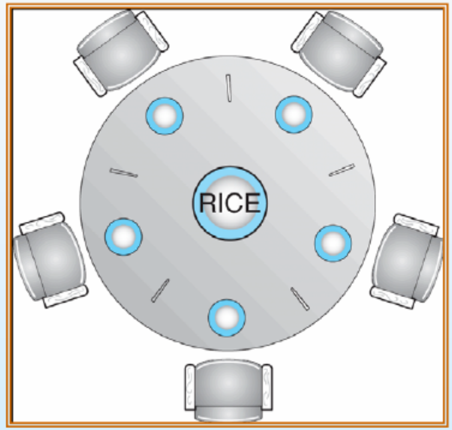
- The structure of a writer process

```
while (true) {
    wait(wrt);
        // writing is performed
    signal(wrt);
}
```

- The structure of a reader process

```
while (true) {
    wait(mutex);
    readcount ++;
    if (readcount == 1) wait(wrt);
    signal(mutex)
        // reading is performed
    wait(mutex);
    readcount - -;
    if (readcount == 0) signal(wrt);
    signal (mutex);
}
```

Dining-Philosophers Problem I



Dining-Philosophers Problem II

- Shared data
  - Bowl of rice (data set)
  - Semaphore **chopstick [5]** initialized to 1
- The structure of Philosopher i:

```
While (true) {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
                // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
                // think
}
```

- This solution may cause a deadlock.

Dining-Philosophers Problem III

- Several possible remedies
  - Allow **at most 4 philosophers** to be sitting simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - Odd philosophers pick up first her left chopstick and then her right chopstick, while even philosophers pick up first her right chopstick and then her left chopstick.
- 注： deadlock-free & starvation-free

Problems with Semaphores

- Incorrect use of semaphore operations:

  signal (mutex) ⋯. wait (mutex)

  - the mutual-exclusion requirement is violated, processes may in their CS simultaneously

  wait (mutex) ⋯ wait (mutex)

  - a deadlock will occur.

  Omitting of wait (mutex) or signal (mutex) (or both)

  - either mutual-exclusion requirement is violated, or a deadlock will occur

Outline

1. Classical Problems of Synchronization
2. Monitors
3. Synchronization Examples
4. 小结和作业

Monitors I

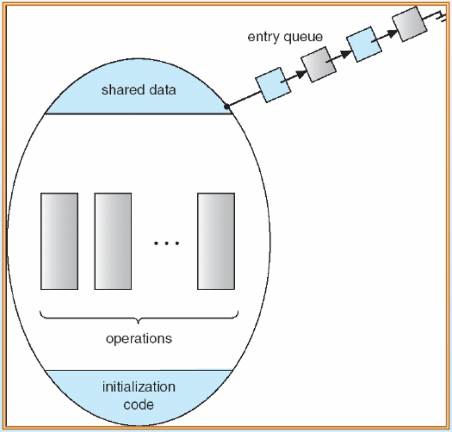
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

Syntax of a monitor

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (···) {···}
        ···
    procedure Pn (···) {···}
    Initialization code (···.) {···}
}
```

Monitors II

- Schematic view of a Monitor



Condition Variables I

- the monitor construct is not sufficiently powerful for modeling some synchronization scheme.

  condition x, y;

- Two operations on a condition variable:

  x.wait()

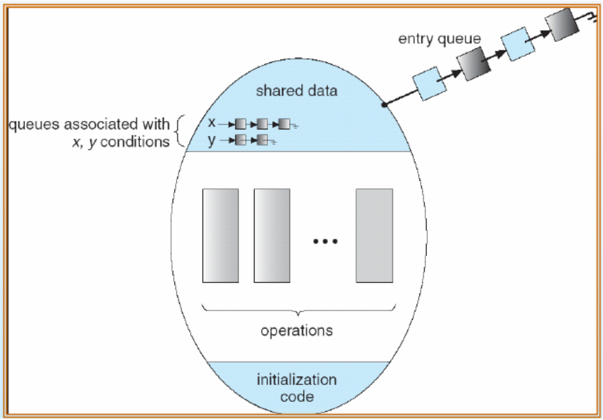
  - a process that invokes the operation is suspended.

  x.signal()

  - resumes one of processes (if any) that invoked x.wait ()

Condition Variables II

- Monitor with Condition Variables



Condition Variables III

- Problem with x.signal()
  - process P invokes x.signal, and a suspended process Q is allowed to resume its execution, then ?
    - **signal and wait**
    - **signal and continue**
  - in the language Concurrent Pascal, a compromise was adopted
    - when P executes the signal operation, it immediately leaves the monitor, hence, Q is immediately resumed.

A deadlock-free solution to Dining Philosophers I

- the monitor

```
monitor DP
{
    enum { THINKING; HUNGRY, EATING} state[5] ;
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
```

A deadlock-free solution to Dining Philosophers II

```
    }

    void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ;
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

A deadlock-free solution to Dining Philosophers III

- Each philosopher i invokes the operations **pickup()** and **putdown()** in the following sequence:

  dp.pickup (i)

  EAT

  dp.putdown (i)

- not **starvation-free**

Monitor Implementation Using Semaphores I

- Variables

  semaphore mutex; // (initially = 1) , for enter and exit monitor

  semaphore next; // (initially = *0*)

  int next-count = 0;

Monitor Implementation Using Semaphores II

- Each **external** procedure F will be replaced by

```
wait(mutex);
  ...
    body of F;
  ...
if (next-count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation I

- For each condition variable x, we have:

  semaphore x-sem; // (initially = *0*)

  int x-count = 0;

- The operation x.wait can be implemented as:

Monitor Implementation II

```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count–;
```

Monitor Implementation III

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count–;
}
```

Outline

1. Classical Problems of Synchronization
2. Monitors
3. Synchronization Examples
4. 小结和作业

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
- Uses **condition variables** and **readers-writers locks** when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses **interrupt masks** to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
- Also provides **dispatcher objects** which may act as either mutexes and semaphores
- Dispatcher objects may also provide **events**
    - An event acts much like a condition variable

Linux Synchronization

- before 2.6, nonpreemptive kernel
- now, fully preemptive
- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Outline

1. Classical Problems of Synchronization
2. Monitors
3. Synchronization Examples
4. 小结和作业

小结

1. Classical Problems of Synchronization
2. Monitors
3. Synchronization Examples
4. 小结和作业

作业

- 华夏班：6.3，6.11，6.13
- 非华夏班：
  - 临界区问题的解答必须满足的三个要求是什么？
  - 7.1，7.8

谢谢！