Chapter 6

Process Synchronization

Contents

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

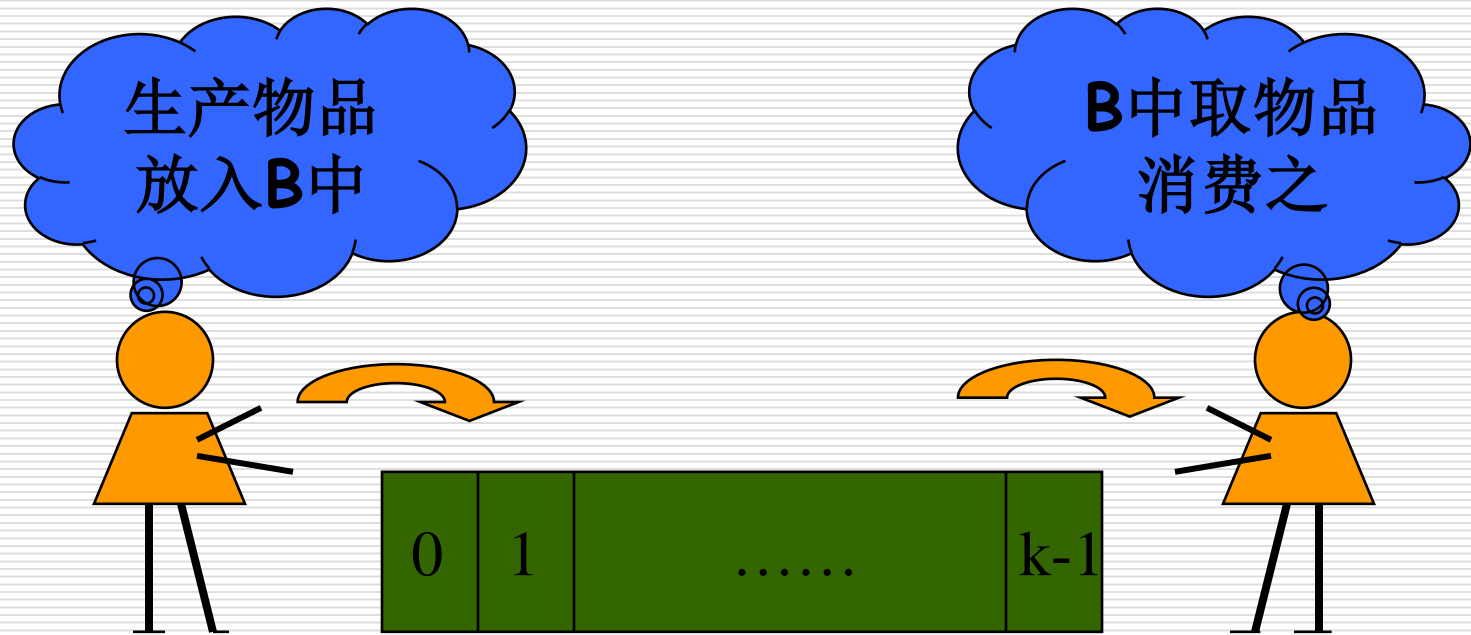
Objectives

- To introduce the critical section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of atomic transaction and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Producer and Consumer Problem



B:Array[0..k-1]Of item

Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
Typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

缓冲区下一个空位

缓冲区第一个非空位

Producer Process

```
item nextProduced;

    while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
    }
```

Consumer Process

```
item nextConsumed;

    while (1) {
        while (in == out)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    }
```

- Problem: Solution is correct, but can only use BUFFER_SIZE-1 elements

One solution to this problem

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
        while (counter == BUFFER_SIZE)
              ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            counter++;
}
```

Consumer

```
while (true)  {
        while (counter == 0)
            ; // do nothing
            nextConsumed =  buffer[out];
             out = (out + 1) % BUFFER_SIZE;
                counter--;

             /*  consume the item in nextConsumed
  }
```

Problem

- counter++ could be implemented as
  register1 = counter
  register1 = register1 + 1
  counter = register1
- counter-- could be implemented as
  register2 = counter
  register2 = register2 - 1
  counter = register2
- Consider this execution interleaving with "counter = 5" initially:
  S0: producer execute register1 = counter {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = counter {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute counter = register1 {counter = 6 }
  S5: consumer execute counter = register2 {counter = 4}

Race Condition

- A situation that when several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition
- So we must guarantee that only one process can operate the variable.

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Critical-Section

- The segment of code in which the process may be changing common variables, updating a table, writing a file, and so on, is called critical section

```
Do{
    entry section
    critical section
    exit section
    remainder section
}while(1)
```

Critical-Section Problem

- The Critical-Section Problem is to design a protocol that the processes can use to cooperate.

Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Principles to use critical section

- Only one process in critical section
- If there are many processes that want to enter the critical section, one process should be allowed within bounded time
- Process can stay in CS within bounded time

Algorithm 1

One shared integer variable: turn (0 or 1)

```
do {
    while (turn != i)  ;  //entry section
        Critical section
    turn = j; //exit section
    Remainder section
} while (1);
```

Is it correct?

Algorithm 2

```
do {
   flag [i] = true;
   while (flag [j])  ;  //entry section
       critical section
   flag [i] = false;  //exit section
   remainder section
} while (1);
```

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

- Process P_i

```
do {
    flag [i] = true;
    turn = j;
    while (flag [j] && turn = = j) ; //entry
    critical section
    flag [i] = false;   //exit
    remainder section
} while (1);
```

- Process P_j

```
do {
    flag [j] = true;
    turn = i;
    while (flag [i] && turn = = i) ; //entry
    critical section
    flag [j] = false;   //exit section
    remainder section
} while (1);
```

Solution to multiprocess problem

- Idea
  - Every client will receive a number(1, 2, …, n), the client with minimal number will be served first
- Problem
  - Can not guarantee that different clients get different numbers
- Solution
  - Except the number, process's name is used

Solution to multiprocess problem

- implementation
  - boolean choosing[n]; //false
  - int number[n]; //0
  - definition:
    - $\max(a_0, \ldots, a_{n-1})$is the number k, and $k \geq a_i$, for all i = 0, …, n-1
    - (a,b)<(c,d) iff (a<c)or(a=c and b<d)

Solution to multiprocess problem

Program for p_i

```
do {
    choosing [i] = true;
    number [i] = max(number[0], number[1], …, number[n-1]) + 1;
    choosing [i] = false;
    for (j = 0; j < n; j ++) {
        while (choosing [j]) ;
        while ((number [j] != 0) && (number [j], j) < (number [i], i)) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
```

Eisenberg/Mcguire

- Data structure
  - enum *flag*[*n*] (idle, want_in, in_cs);
  - int turn; //in the range of (0,*n*-1)
  - Initialize all elements of *flag to be i*dle, *turn* takes the value between 0 and *n*-1. In addition, every process has a local variable, as follows:
  - int *j*; //in the range of (0,*n*-1)

Eisenberg/Mcguire

```
do{
    while(true){
        flag[i] = want_in;
        j = turn;
        while (j != i){
            if (flag[j] != idle)  j = turn;
            else                  j = (j+1)% n;
        }
        flag[i] = in_cs;     j = 0;
        while ((j<n) && (j == i || flag[j] != in_cs))      j ++;  //the # of processes not in_cs+myself
        if((j>=n) && (turn ==i || flag[turn]==idle))   //I am the only one whose state is in_cs
            break;
    }
    turn =i;
    Critical section
    j = (turn+1)% n;
    while (flag[j] == idle)
        j = (j+1)% n;
    turn = j; flag[i] = idle;
    //Remainder section
}while(true);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessor environment – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - TestAndSet: test memory word and set value
  - Swap: swap contents of two memory words

TestAndSet Instruction

- Definition:

```
boolean TestAndSet (boolean *target)
 {
     boolean rv = *target;
     *target = TRUE;
     return rv;
 }
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
while (true) {
        while ( TestAndSet (&lock ))
          ;   /* do nothing
        critical section
        lock = FALSE;
        //remainder section
}
```

Solution using TestAndSet

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestAndSet(lock);
    waiting[i] = false;
    critical section
    j = (i + 1) %n;
    while ((j != i) && !waiting[j])
        j = (j + 1) %n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    remainder section
} while (1);
```

```
do {
    waiting[j] = true;
    key = true;
    while (waiting[j] && key)
        key = TestAndSet(lock);
    waiting[j] = false;
    critical section
    i = (j + 1) %n;
    while ((i != j) && !waiting[k])
        i = (i + 1) %n;
    if (i== j)
        lock = false;
    else
        waiting[i] = false;
    remainder section
} while (1);
```

It must be atomic!

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
        critical section
        lock = FALSE;
        // remainder section
}
```

Give a correct solution using swap()!

Solution using Swap

```
do {
	waiting[i] = true;
	key = true;
	while (waiting[i] && key)
		swap(lock, key)
	waiting[i] = false;
	critical section
	j = (i + 1) %n;
	while ((j != i) && !waiting[j])
		j = (j + 1) %n;
	if (j == i)
		lock = false;
	else
		waiting[j] = false;
	remainder section
} while (1);
```

Semaphore

- The hardware-based solutions to the critical-section problem are complicated for application programmers to use.
- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
  - Passeren, Vrijgeven
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    ```
    wait (S) {
        while S <= 0
            ; // no-op
          S--;
    }
    ```
  - signal (S) {
    ```
    signal (S) {
      S++;
    }
    ```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore

How to provide mutual exclusion and synchronization

- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

        Critical Section

    signal (S);
- Provides synchronization
  - P_1:

    S_1;

    signal(synch);
  - P_2:

    wait(synch);

    S_2;

Semaphore Implementation

- The main disadvantage of the semaphore definition given above is that it requires busy waiting.
- This type of semaphore is also called a spinlock because the process “spins” while waiting for the lock.
- This is a waste of CPU time.

Semaphore Implementation with no Busy waiting

- With each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list.
  - value (of type integer)
  - pointer to next record in the list

```
Typedef struct{
    int value;
    struct process *list;
}semphore
```

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
        s->value--;
        if (s->value < 0) {
                add this process to waiting queue
                block();
        }
}
```

- Implementation of signal:

```
Signal (S){
        s->value++;
        if (s->value <= 0) {
                remove a process P from the waiting queue
                wakeup(P);  }
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Binary Semaphore

- the semaphore value is restricted to 0 and 1.
- Wait() succeeds only when the semaphore value is 1
- Signal() does not change the semaphore value when it is 1
- How to implement a counting semaphore S as a binary semaphore
- binary-semaphore S1, S2;
  - int C;
  - Initialize S1 = 1, S2 = 0, C is equal to S

Binary Semaphore

- Wait
  - wait(S1);
  - C--;
  - if (C < 0) {
    - signal(S1);
    - wait(S2);
  - }
  - signal(S1);
- Signal
  - wait(S1);
  - C++;
  - if (C <= 0)
    - signal(S2);
  - else
    - signal(S1);

Wait() and signal() operation

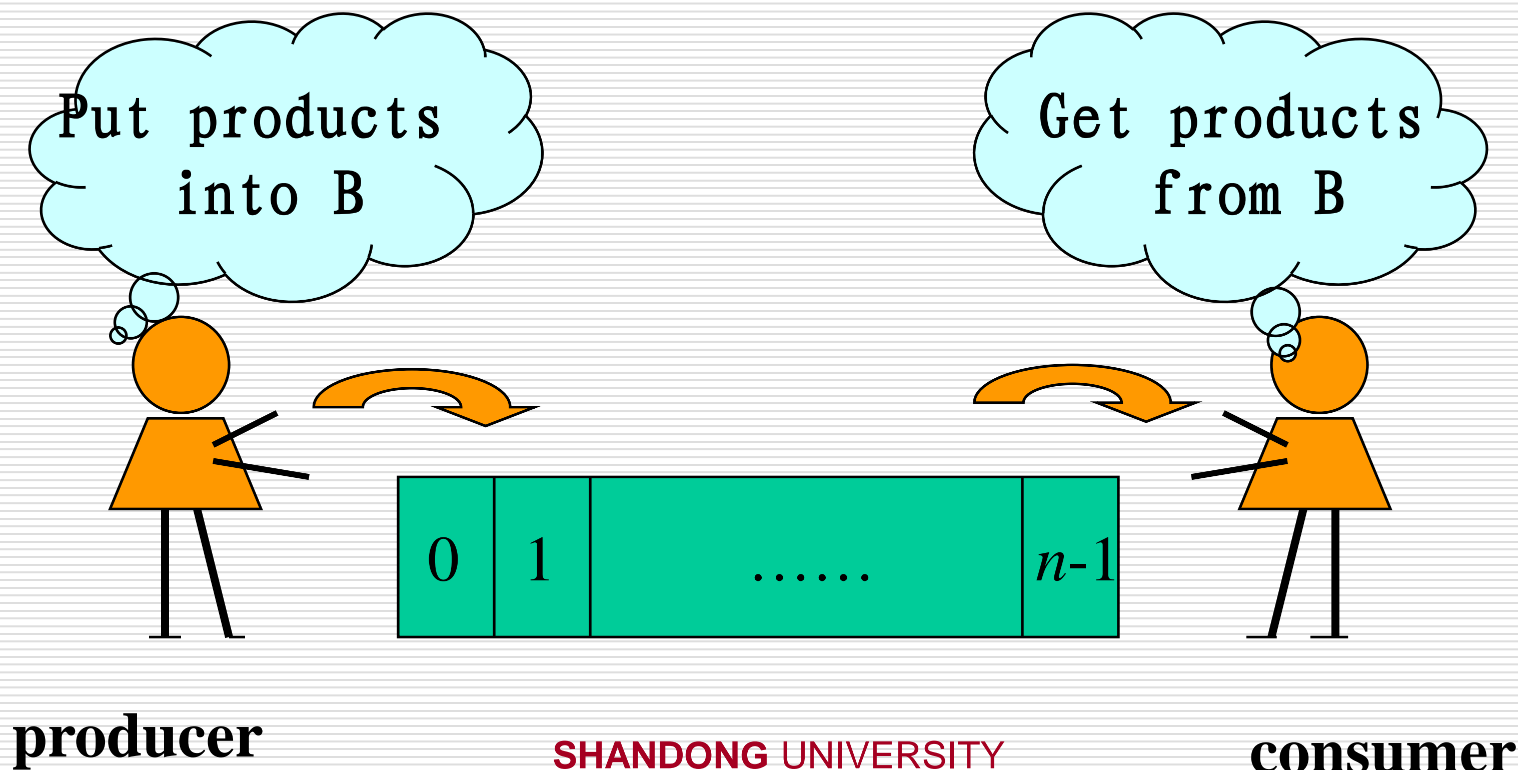
- Usually, they are used with pair
- For mutual exclusion, they are in the same process
- For synchronization, they are in different processes
- If there are wait() operations, the sequence is very important. However, it is not important for two signal() operations.
- Wait() for synchronization is before that for mutual exclusion

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *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.



producer

consumer

Bounded Buffer Problem (Cont.)

Producer process

```
while (true)  {

    //   produce an item

    wait (empty);
    wait (mutex);

    //  add the item to the  buffer

     signal (mutex);
     signal (full);

}
```

Consumer process

```
while (true) {
    wait (full);
    wait (mutex);

    //  remove an item from  buffer

    signal (mutex);
    signal (empty);

    //  consume the removed item

}
```

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.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

Two classes

- The first class
  - Writer can access only when this is no reader waiting for access
- The second class
  - The writer has a higher priority

- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
}
```

Readers-Writers Problem (Cont.)

- 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) ;
}
```

Problems

- If there are many readers, the writer has no chance to access the data.
- Solution
  - Once there is a writer to wait for access, the following readers will have to wait. When all readers finish their operation, then the writer will begin to access the data.

Readers-Writers Problem (Cont.)

- int readcount, writecount; (initial value = 0)
- semaphore mutex 1, mutex 2, mutex 3, w, r ; (initial value = 1)

```
READER
While(true){
        wait(mutex 3); //ensure at most one reader will go
                       //before a pending write
            wait(r);
              wait(mutex 1);
                readcount := readcount + 1;
                if readcount = 1 then wait(w);
              signal(mutex 1);
            signal(r);
        signal(mutex 3);
        reading is done
        wait(mutex 1);
            readcount := readcount - 1;
            if readcount = 0 then signal(w);
        signal(mutex 1);
}
```

```
WRITER
While(ture){
        wait(mutex 2);
            writecount := writecount + 1;
            if writecount = 1 then wait(r);
        signal(mutex 2);
        wait(w);
            writing is performed
        signal(w);
        wait(mutex 2);
        writecount := writecount - 1;
        if writecount = 0 then signal(r);
        signal(mutex 2);
}
```

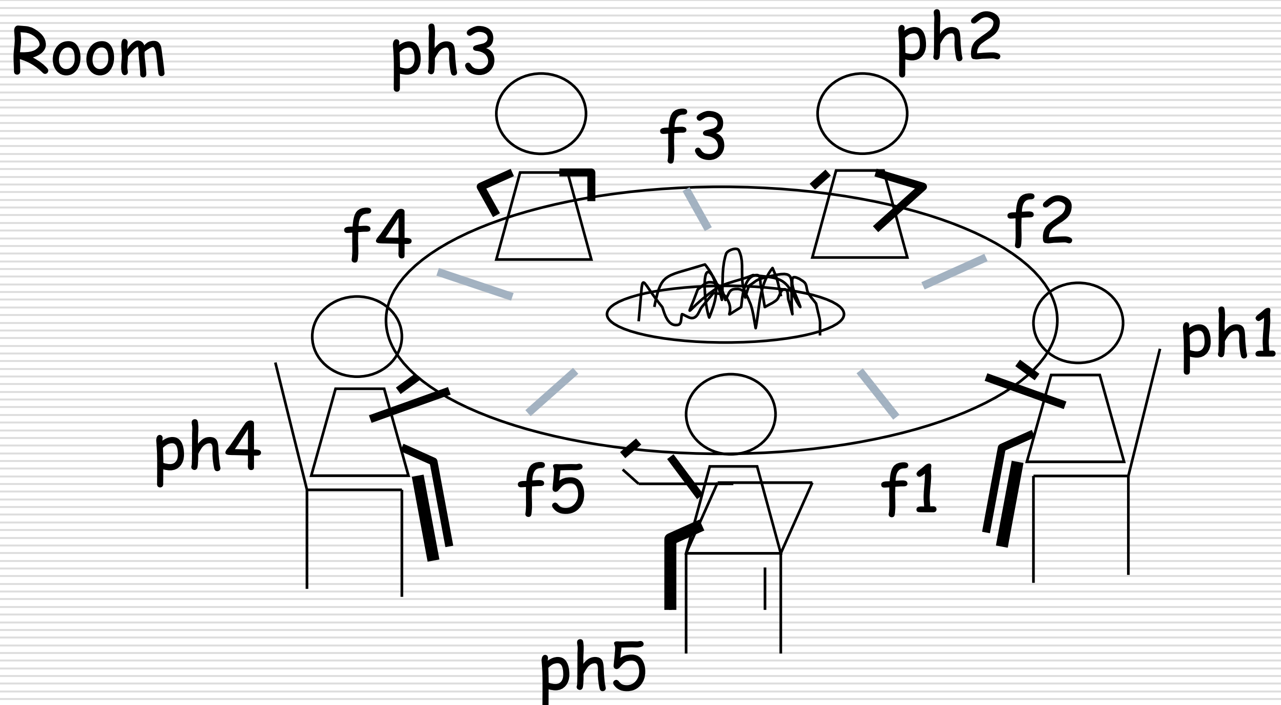
Readers-Writers Problem (Cont.)

- semaphores: no_writers, no_readers, counter_mutex (initial value is 1)
- shared variables: nreaders (initial value is 0)
- local variables: prev, current

```
READER
While(ture){
    P( no_writers );
        P( counter_mutex );
            prev := nreaders;
            nreaders := nreaders + 1;
      V( counter_mutex );
      if prev = 0 then P( no_readers );
   V( no_writers );
   ... read ...
   P( counter_mutex );
       nreaders := nreaders - 1;
       current := nreaders;
   V( counter_mutex );
   if current = 0 then V( no_readers );
}
```

```
WRITER
While(ture){
        P( no_writers );
        P( no_readers );
        V( no_writers );
        ... write ...
        V( no_readers );}
```

Dining-Philosophers Problem



Dining-Philosophers Problem (Cont.)

- Shared data
  - Bowl (data set)
  - Semaphore chopstick [5] initialized to 1

- The structure of Philosopher *i*:

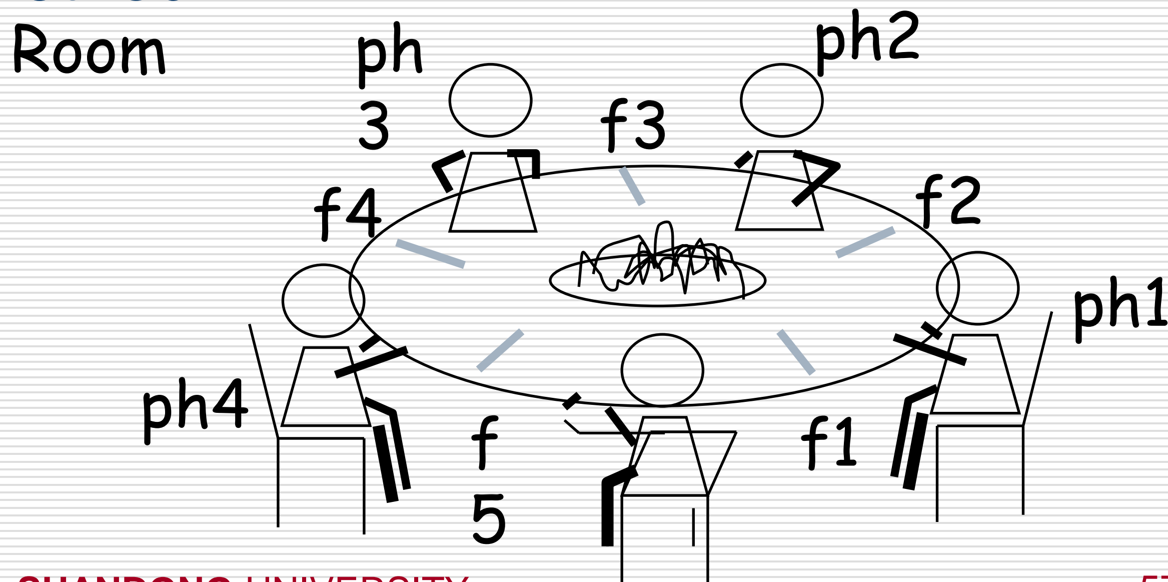
```
While (true)  {
          wait ( chopstick[i] );
           wait ( chopstick[ (i + 1) % 5] );

                   //  eat

           signal ( chopstick[i] );
           signal (chopstick[ (i + 1) % 5] );
                 //  think
}
```

Problem

- When everyone got one chopstick, what will happen.
- Solution
  - 4 persons, at most, are permitted sitting in chair.
  - Two chopsticks are available, pick them up
  - Asymmetric method



Sleeping-Barber problem

- A barbershop consists of a waiting room with n chairs and a barber room with one barber chair. If there is no customer to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the chairs. If the barber is asleep, the customer wakes up the barber.

Sleeping-Barber problem

```
int waiting = 0;            // 等候理发的顾客数
int CHAIRS = n;             // 为顾客准备的椅子数
semaphore customers, barber, mutex;
customers = 0; barber = 0; mutex = 1;
```

```
procedure barber
{
  while (true)
  {
      wait(customers); //等待顾客, 无则睡眠
      wait(mutex);     // 进程互斥
      waiting--;       //等候数减一
      signal(barber);  //理发师叫一个顾客
      signal(mutex);   // 开放临界区
      cut_hair();      // 正在理发
  }
}
```

```
procedure customer
{
    wait(mutex);                 // 进程互斥
    if (waiting < CHAIRS)        //有没有空椅子
    {
        waiting++;               // 等候顾客数加1
        signal(customers);       // 有可能唤醒理发师
        signal(mutex);           // 开放临界区
        wait(barber);            //理发师忙, 顾客等待
        get_haircut();           // 一个顾客坐下理发
    }
    else
        signal(mutex);           // 人满了，走吧！
}
```

Apple and orange problem

- Father, mother, son, and daughter
- One plate on a table
- Father puts one apple on the plate every time
- Mother puts one orange every time
- Son eats apples
- Daughter eats oranges

- Use semaphores to implement this problem

Problems with Semaphores

- Correct use of semaphore operations？
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Problems to use semaphore

- It is very difficult to guarantee that there is no error.
- It is not very easy to read the code.
- It is difficult to modify or maintain the semaphores

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- The idea is first introduced by Dijkstra in 1971. He suggested to combine all operations on critical resources into a single program module– secretary process.
- In 1973, the monitor concept was proposed by Brinch Hansen.
- In 1974, Hoare also described the monitor concept.
- The monitor concept is a fusion of ideas from abstract data types and critical regions. A monitor provides the rest of the world with a limited set of mechanisms for accessing some protected data. The visible part of a monitor consists of the headers of various procedures, each of which can be called by any other process.
- The monitor is implemented in such a way as to allow only one process to be executing any of its procedures at any time.

Monitors

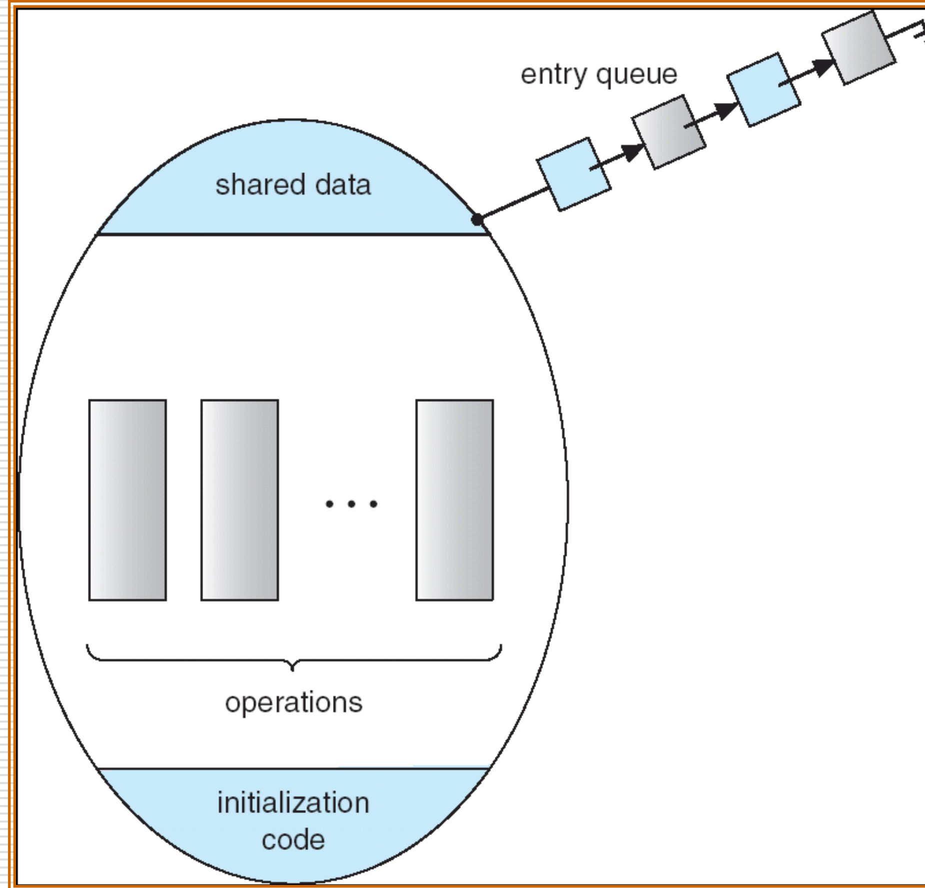
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
          …

    procedure Pn (…) {……}

    Initialization code ( ….) { … }
          …
    }
}
```

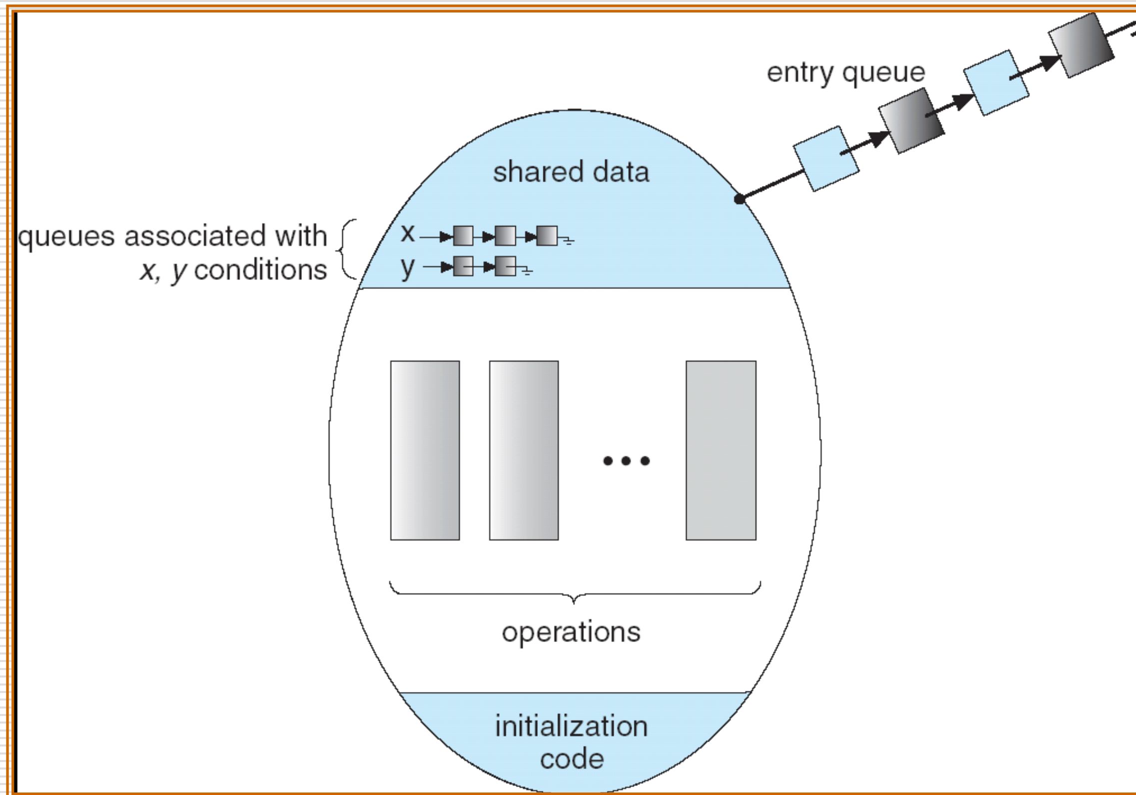
Schematic view of a Monitor



Condition Variables

- To allow a process to wait within the monitor, a condition variable must be declared, as
- 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 ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
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 left and right neighbors
             test((i + 4) % 5);
             test((i + 1) % 5);
      }
```

Solution to Dining Philosophers (cont)

```
    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;
    }
}
```

Solution to Dining Philosophers (cont)

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

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

Options after x.signal( )

- Suppose P invokes x.signal(), and Q is a suspended process with condition x
  - Signal and wait -- P either waits until Q leaves the monitor or waits for another condition
  - Signal and continue – Q either waits until P leaves the monitor or waits for another condition
  - The first one is supported by Hoare.

Monitor Implementation Using Semaphores

- Semaphore mutex (initialized to 1)
  - use it to call the procedure exclusively
  - So, before a process calls a procedure in a monitor, it should first execute wait(mutex).
  - After it exits from the monitor, it should execute signal(mutex).

Monitor Implementation Using Semaphores

- Semaphore next (initialized to 0)
  - Once a process calls x.signal(), it uses wait(next) to suspend itself.
- Int next-count (initialized to 0)
  - Use it to record how many processes are waiting on next.

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each 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 Using Semaphores

- For each condition variable ***x***, we have:

  ```
  semaphore x-sem; // (initially  = 0)
  int x-count = 0;
  ```

- Semaphore x-sem (initialized to 0)
  - Use it to suspend a process, when the resources this process applies are not enough.
- When a type of resource is released, we should know whether there are other processes waiting on this resource. So a counter should be used to record the number of processes waiting on this resource.
- Int x-count (initialized to 0)

Monitor Implementation

- The operation x.wait can be implemented as:

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

Monitor Implementation

- The operation x.signal can be implemented as:

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

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

- 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

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - <T_i starts> written to log when transaction T_i starts
  - <T_i commits> written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo(T_i) restores value of all data updated by T_i
  - Redo(T_i) sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains <T_i starts> without <T_i commits>, undo(T_i)
  - If log contains <T_i starts> and <T_i commits>, redo(T_i)

Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record <checkpoint> to the log on stable storage

Checkpoints

- Now recovery only includes T_i, such that T_i started executing before the most recent checkpoint, and all transactions after T_i. All other transactions already on stable storage
- The recovery operations are as follows:
  - For all transactions Tk that the record <Tk, commits> appears in the log, execute redo(Tk)
  - For all transactions Tk that have no <Tk commits> record in the log, execute undo(Tk)

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are N! valid serial schedules

Schedule 1: T_0 then T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Nonserial Schedule

- Nonserial schedule allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i, O_j
  - Conflict if access same data item, with at least one write
- If O_i, O_j consecutive operations of different transactions O_i and O_j don't conflict
  - Then S' with swapped order O_j O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
  - S is conflict serializable

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Serializable Schedule

T1: 读B; A=B+1; 写回A;　T2: 读A; B=A+1; 写回B;　A,B初值均为2

T1	T2
Slock B	
Y = B(=2)	
Unlock B	
Xlock A	
A = Y + 1	
写回A(**=3**)	
Unlock A	
	Slock A
	X = A(= 3)
	Unlock A
	Xlock B
	B = X + 1
	写回B(**=4**)
	Unlock B

a　串行调度

T1	T2
	Slock A
	X = A(=2)
	Unlock A
	Xlock B
	B = X + 1
	写回B(**=3**)
	Unlock B
Slock B	
Y = B(=3)	
Unlock B	
Xlock A	
A = Y + 1	
写回A(**=4**)	
Unlock A	

b　串行调度

T1	T2
Slock B	
Y = B(=2)	
	Slock A
	X = A(=2)
Unlock B	
	Unlock A
Xlock A	
A = Y + 1	
写回A(**=3**)	
	Xlock B
	B = X + 1
	写回B(**=3**)
Unlock A	
	Unlock B

c　不可串行化的调度
(结果与a,b不同，错误调度)

T1	T2
Slock B	
Y = B(=2)	
Unlock B	
Xlock A	
	Slock A
A = Y + 1	等待
写回A(**=3**)	等待
Unlock A	等待
	X = A(=3)
	Unlock A
	Xlock B
	B = X + 1
	写回B(**=4**)
	Unlock B

d　可串行化的调度
(结果与a相同，正确调度)

Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
  - Exclusive – Ti has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Problem
  - Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
  - $TS(T_i) < TS(T_j)$ if Ti entered system before T_j
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose Ti executes read(Q)
  - If TS(T_i) < W-timestamp(Q), Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If TS(T_i) ≥ W-timestamp(Q)
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), TS(T_i))

Timestamp-ordering Protocol

- Suppose Ti executes write(Q)
  - If $TS(T_i)$ < R-timestamp(Q), value Q produced by T_i was needed previously and T_i assumed it would never be produced
    - Write operation rejected, T_i rolled back
  - If $TS(T_i)$ < W-timestamp(Q), T_i attempting to write obsolete value of Q
    - Write operation rejected and T_i rolled back
  - Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Example(1)

关于临界区问题（critical section problem）的一个算法（假设只有进程P0和P1可能会进入该临界区）如下（i为0或1），该算法__________。

A、不能保证进程互斥进入临界区，且会出现“饥饿”（Starvation）

B、不能保证进程互斥进入临界区，但不会出现“饥饿”

C、保证进程互斥进入临界区，但会出现“饥饿”

D、保证进程互斥进入临界区，不会出现“饥饿”

```
repeat
 retry: if (turn!= -1 ) turn := i;
        if (turn!=i ) go to retry;
        turn := -1;
        Critical Section(临界区)
        turn := 0;
        remainder Section(其它区域)
until false;
```

Example(2)

下述关于双进程临界区问题的算法（对编号为id的进程）是否正确：

```
do{
      blocked[id]=true;
      while(turn !=id)
     {
       while(blocked[1-id]);
        turn=id;
     }
      <编号为id的进程的临界区 CS>

       blocked[id]=false;

       编号为id的进程的非临界区
   } while (true):
```

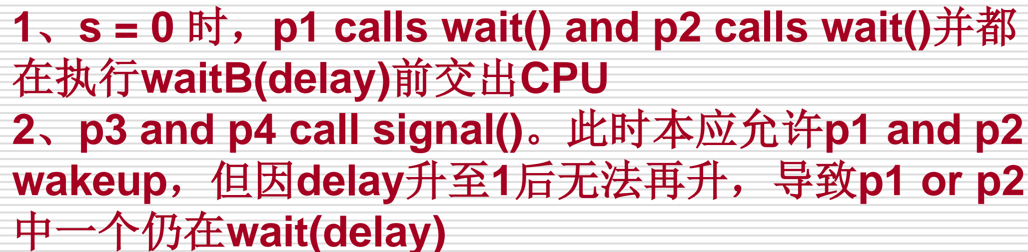
若此时进程切换，且让对方再次进入临界区，互斥条件无法满足

其中，布尔型数组blocked[2]初始值为为{false,false}，整型turn初始值为0，id代表进程编号（0或1）。请说明它的正确性，或指出错误所在。

Example(3)

```
wait(semaphore  s)
{
        waitB(mutex);
        s = s-1;
        if (s<0)
        {
          signalB(mutex);
          waitB(delay);
        }
        else
          signalB(mutex);
}
```

```
signal(semaphore  s)
{
       waitB(mutex);
       s= s+1;
       if(s<=0)
          signalB(delay);
       else
       signalB(mutex);
}
```



Example(4)

- 某银行提供1个服务窗口和10个供顾客等待的座位。顾客到达银行时，若有空座位，则到取号机上领取一个号，等待叫号。取号机每次仅允许一位顾客使用。当营业员空闲时，通过叫号选取一位顾客，并为其服务。顾客和营业员的活动过程描述如下：

```
process 顾客i
{
    从取号机获得一个号码;
    等待叫号;
    获得服务;
}
```

```
process 营业员
{
    while (TRUE)
    {
        叫号;
        为顾客服务;
    }
}
```

- 请添加必要的信号量和P、V(或wait()、signal())操作，实现上述过程中的互斥与同步。要求写出完整的过程，说明信号量的含义并赋初值。

- Semaphore seats =10;//表示空余座位数量的资源信号量，初值为10
- Semaphore mutex = 1; //管理取号机的互斥信号量，初值为1，表示取号机空闲
- Semaphore custom = 0; //表示顾客数量的资源信号量，初值为0

```
Process 顾客
{
    P(seats); //找个空座位
    P(mutex); //在看看取号机是否空闲
    从取号机取号;
    V(mutex) //放开那个取号机
    V(custom); //取到号，告诉营业员有顾客
    等待叫号;
    V(seats) //被叫号，离开座位
    接受服务;
}
```

```
Process 营业员
{
    While(true)
    {
        P(custom); //看看有没有等待的顾客
        叫号;
        为顾客服务;
    }
}
```

Assignments

- 6.3, 6.7, 6.9, 6.11, 6.16
- Apple, orange problem

End of Chapter 6

Any Question?