The picture can't be displayed.

Chapter 6: Process Synchronization

Chapter 6: Process Synchronization

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

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- 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 count 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 (count == BUFFER_SIZE)
                        ; // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          count++;
}
```

Consumer

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

        /*  consume the item in nextConsumed
 }
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

Critical-section problem

- To design a protocol that the processes can use to cooperate

```
Do {
    Entry section
        Critical section
    Exit section
        Remainder section
}while(TRUE);
```

General structure of a typical process Pj

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 the selection of the processes that will enter the critical section next 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

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

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

            CRITICAL SECTION

        flag[i] = FALSE;

            REMAINDER SECTION

}
```

If two processes are running the statement simultaneously, only one will last.

Algorithm for Process P_i

- Pi

```
flag[i] = TRUE;
turn = i;

while ( flag[j] && turn
== j);
```

- Pj

```
flag[j] = TRUE;

turn = j;
while ( flag[i] && turn
== i);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – 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
  - Either test memory word and set value
  - Or 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

}
```

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

 }
```

Semaphore

- Synchronization tool that is less complicated
- Semaphore *S* – integer variable
- Two atomic standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }
- Can be implemented without busy waiting

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

    Critical Section

    signal (S);

Usage as General Synchronization Tool(2)

- P1 has a statement S1, P2 has S2
- Statement S1 to be executed before S2

P1

```
S1;
Signal(S);
```

Question: What's the initial value of S?

P2

```
Wait(S);
S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each semaphore has two data items:
  - value (of type integer)
  - pointer to a linked-list of PCBs.
  - Typedef struct{
    - Int value;
    - Struct process *list;

    } semaphone;

- Two operations (provided as basic system calls):
  - 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){
        value--;
        if (value < 0) {
                add this process to waiting queue
                block(); }
}
```

- Implementation of signal:

```
Signal (S){
        value++;
        if (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.

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- *How many Semaphores do we need?*
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0, counting full items
- Semaphore empty initialized to the value N, counting empty items.

Bounded Buffer Problem (Cont.)

- 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);
}
```

Bounded Buffer Problem (Cont.)

- 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

}
```

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.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1, to ensure mutual exclusion when readcount is updated.
  - 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

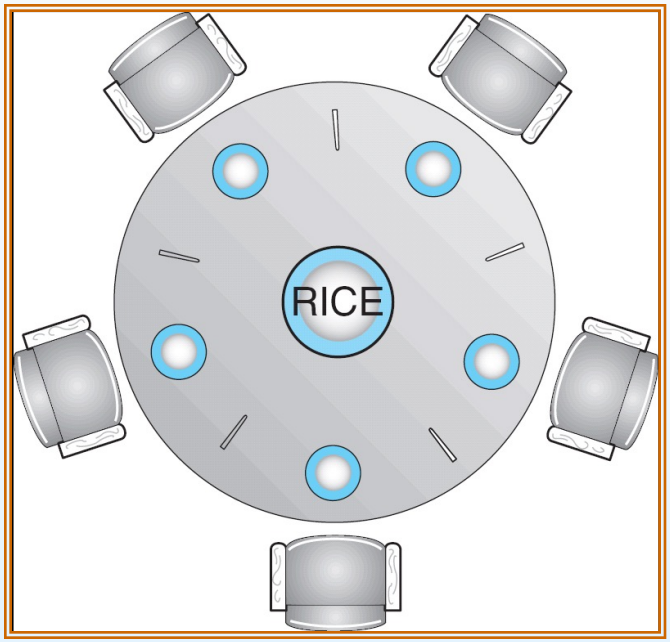
```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)  signal (wrt) ;
        signal (mutex) ;
}
```

"Locking" the wrt semarphore, rather than "waiting"

Reason is that wrt is initialized to "1"

"Unlocking" the wrt semarphore, rather than "signaling"

Dining-Philosophers Problem



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

Dining-Philosophers Problem (Cont.)

- 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

}
```

Problems with Semaphores

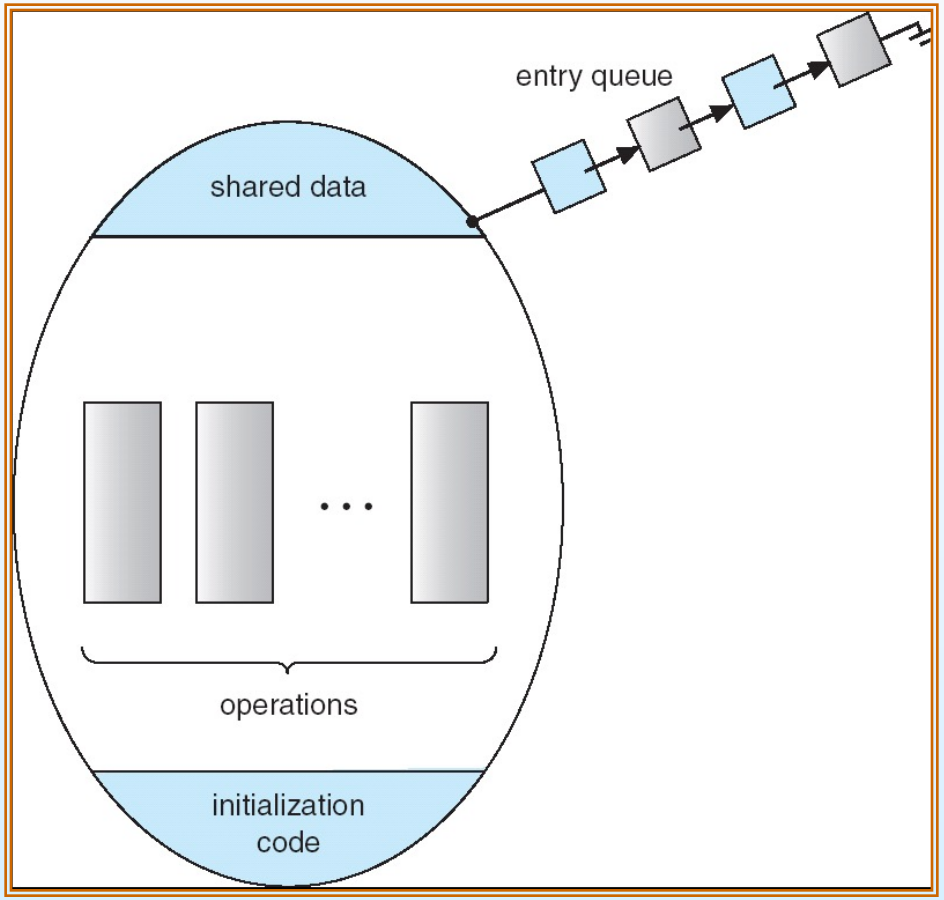
- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- 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

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

    procedure Pn (…) {……}

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

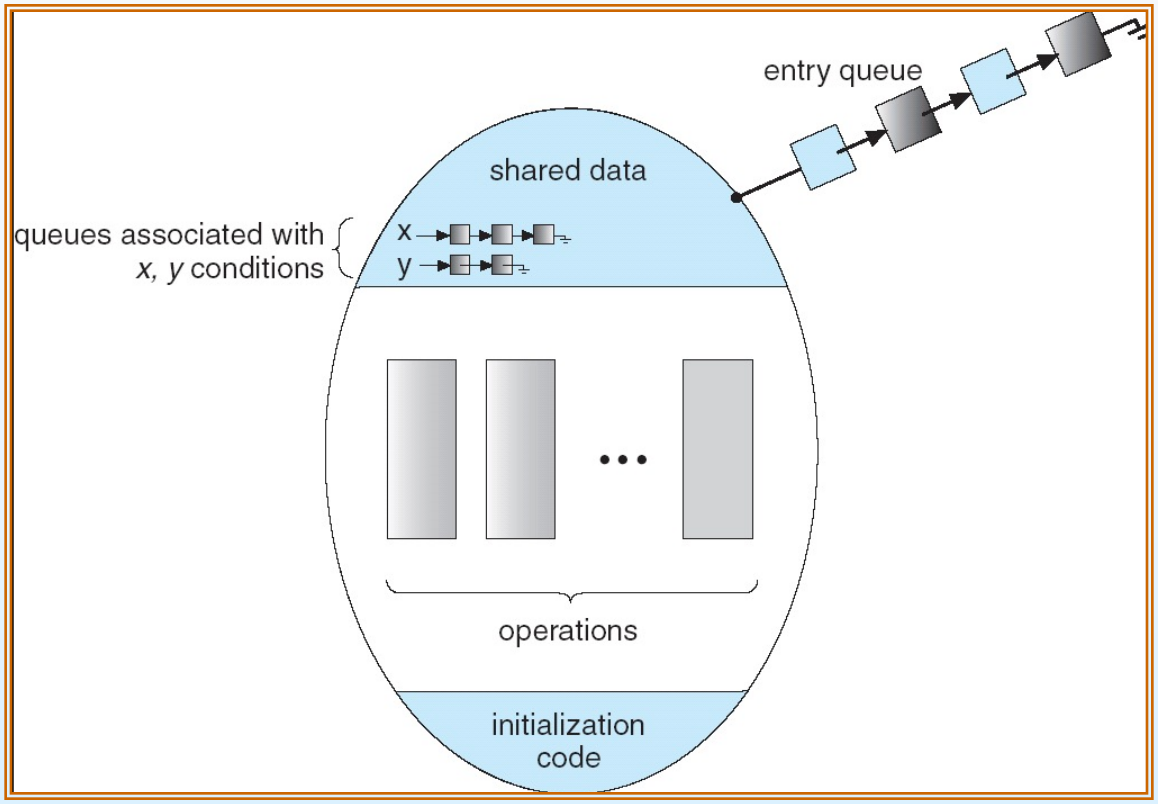
Schematic view of a Monitor



Condition Variables

- 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];  //philosopher i can delay herself when unable to get
     chopsticks


     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)

- When the left and right philosophers, **self[(i+4)%5]** and **self[(i+1)%5]** continue to eat, **self[i]** may starve.

Monitor Implementation Using Semaphores

- Variables

  semaphore mutex; // (initially = 1), entry protection

  semaphore next; // (initially = 0), signaling process may suspend themselves.

  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);
```

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore “next” is introduced, on which the signaling process may suspend themselves.

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

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

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

- The operation x.wait can be implemented as:

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

If someone has been waiting, wake her up because I'll be entering the waiting state.

No one else waiting in the monitor. I'm going to block. Allow someone else to enter the monitor now.

Monitor Implementation

- The operation x.signal can be implemented as:

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

This is the signaling process. It will wait on the “next” semaphore.

Semaphore vs. Monitor

Semaphores	*Condition Variables*
Can be used anywhere in a program, but should not be used in a monitor	Can only be used in monitors
Wait() does not always block the caller (*i.e.*, when the semaphore counter is greater than zero).	Wait() always blocks the caller.
Signal() either releases a blocked thread, if there is one, or increases the semaphore counter.	Signal() either releases a blocked thread, if there is one, or the signal is lost as if it never happens.
If Signal() releases a blocked thread, the caller and the released thread ***both*** continue.	If Signal() releases a blocked thread, the caller yields the monitor blocks(Hoare type) or continues (Mesa Type). Only one of the caller or the released thread can continue, but not both.

Semaphore Implementation using Monitor (Java)

```
public class Semaphore {
 private int value;
 public Semaphore (int initial)
    {value = initial;}
 synchronized public void up() {
    ++value;
    notify();
 }
 synchronized public void down()
    throws InterruptedException {
    while (value == 0) wait();
    --value;
 }
 } // Semaphore
```

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 (page 218)
  - The idea is to use a spinlock when trying to access a resource locked by a currently-running thread, but to sleep if the thread is not currently running.
- 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 (busy-waiting semaphore) on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events and timer
  - An event acts much like a condition variable
  - A timer is used to notify one thread if a specified amount of time has expired
- Dispatcher object from signaled state to nonsignaled state

Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

Single Processor	Multiple Processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

Pthreads Synchronization

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

Pthread mutex example

- void reader_function (void);
  void writer_function (void);

```
char buffer;
int buffer_has_item=0;
pthread_mutex_t mutex;
struct timespec delay;
void main ( void ){
   pthread_t reader;

   delay.tv_sec = 2;
   delay.tv_nec = 0;

   pthread_mutex_init (&mutex,NULL);
   pthread_create(&reader, pthread_attr_default, (void *)&reader_function),
NULL);
   writer_function( );
}
```

The writer thread

- void writer_function (void){
  while(1){

  pthread_mutex_lock (&mutex);
  if (buffer_has_item==0){
  buffer=make_new_item( );
  buffer_has_item=1;
  }

  pthread_mutex_unlock(&mutex);
  pthread_delay_np(&delay);
  }
  }

The reader thread

```
void reader_function(void){
   while(1){
      pthread_mutex_lock(&mutex);
      if(buffer_has_item==1){
         consume_item(buffer);
         buffer_has_item=0;
      }
      pthread_mutex_unlock(&mutex);
      pthread_delay_np(&delay);
   }
}
```

Some Exercises

- 某车站售票厅，任何时刻最多可容纳20名购票者进入，当售票厅中少于20购票者时，则厅外的购票者可立即进入，否则需在外面等待。若把一个购票者看作一个进程，请回答问题： 用P、V操作管理这些并发进程时，应怎样定义信号量？写出信号量的初值以及信号量各种取值的含义，以及相关的伪代码

定义一信号量S，初始值为20。 S>0， S的值表示可继续进入售票厅的人数；
S=0， 表示售票厅中已有20名顾客； S<0， |S|的值为等待进入售票厅中的人数

```
Process Pi(i=1,2,…,)
{
   P(S)
     进入售票厅；
     购票；
   V(S)
}
```

Some Exercises

- 桌上有一空盘，只允许存放一个水果。爸爸可向盘中放苹果，也可向盘中放桔子。儿子专等吃盘中的桔子，女儿专等吃盘中的苹果。规定当盘中空时一次只能放一只水果供吃者取用，请用P、V原语实现爸爸、儿子、女儿三个并发进程的同步。

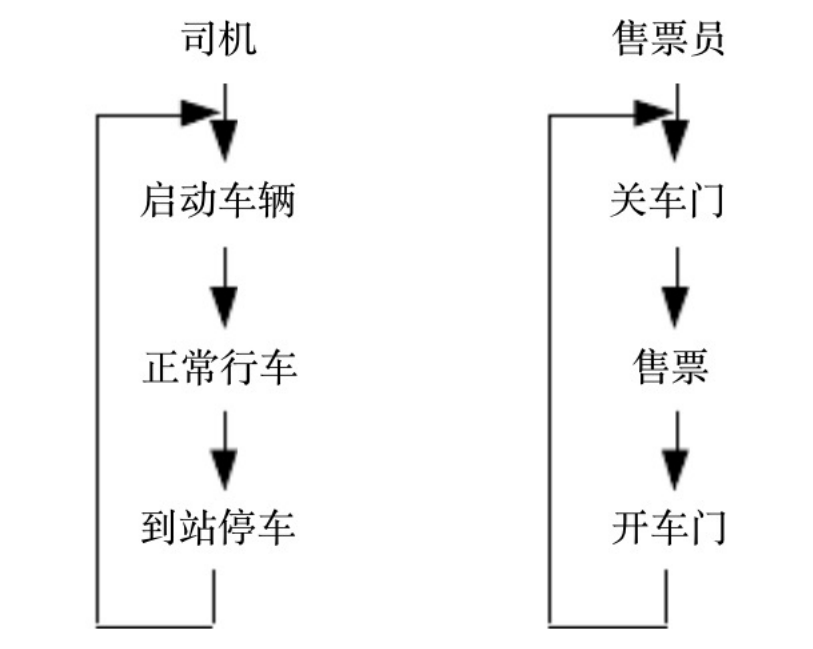
```
int S=1;   //盘子是否为空
int Sa=0;  //是否有苹果
int So=0;  //是否有橘子
main( )
{
    cobegin
    father();
    son();
    daughter();
    coend
}
```

```
father()
{
    while(1)
    {
        P(S );
        将水果放入盘中；
        if （放入的是桔子）
            V(So);
        else V(Sa);
    }
}
```

```
son( )
{
    while(1)
    {
        P(So);
        从盘中取出桔子；
        V(S);
        吃桔子；
    }
}
```

Some Exercises

- 在公共汽车上，司机和售票员的工作流程如图所示。为保证乘客的安全，司机和售票员应密切配合协调工作。请用信号量来实现司机与售票员之间的同步。



Some Exercises

```
Driver：
  while(1){
      Wait (s1);
      启动车辆；
      正常行驶；
      到站停车；
      Signal(s2)
  }
```

```
Conductor：
  while(1){
      关车门；
      Signal(s1);
      售票；
      Wait(s2)
      开车门；
  }
```

Some Exercises

- 如果有三个进程R、W1、W2共享一个缓冲器B，而B中每次只能存放一个数。当缓冲器中无数时，进程R可以将从输入设备上读入的数存放到缓冲器中。若存放到缓冲器中的是奇数，则允许进程W1将其取出打印；若存放到缓冲器中的是偶数，则允许进程W2将其取出打印。同时规定：进程R必须等缓冲区中的数被取出打印后才能再存放一个数；进程W1或W2对每次存入缓冲器的数只能打印一次；W1和W2都不能从空缓冲中取数。写出这三个并发进程能正确工作的程序

Some Exercises

```
semaphore S=1,SO=SE=0;
buffer B;
process R（）
{
  int x;
 while(1)
 {
  从输入设备上读一个数;
  x=接收的数;
  wait(S);
  B=x;
  if B=奇数
  then signal(SO);
  else signal(SE);
 }
}
```

```
process W1()
{
   int y;
  while(1)
  {
     wait(SO);
     y=B;
     signal(S);
    打印y;
  }
}
```

```
process W2()
{
   int z;
   while(1)
   {
     wait(SE);
     z=B;
     signal(S);
    打印z;
   }
}
```

Some Exercises

- a，b两点之间是一段东西向的单行车道，现要设计一个自动管理系统，管理规则如下：
  1. 当ab之间有车辆在行驶时同方向的车可以同时驶入ab段，但另一方向的车必须在ab段外等待；
  2. 当ab之间无车辆在行驶时，到达a点（或b点）的车辆可以进入ab段，但不能从a点和b点同时驶入；
  3. 当某方向在ab段行驶的车辆驶出了ab段且暂无车辆进入ab段时，应让另一方向等待的车辆进入ab段行驶。 请用信号量为工具，对ab段实现正确管理以保证行驶安全。

Some Exercises

```
semaphore S1=1,S2=1,Sab=1;
int ab=ba=0;
void Pab()
{
  while(1)
  {
    wait(S1);
    if(ab==0)
      wait(Sab);
    ab=ab+1;
    signal(S1);
    车辆从a点驶向b点;
    wait(S1);
    ab=ab-1;
    if(ab==0)
      signal(Sab);
    signal(S1);
  }
}
```

```
void Pba()
{
   while(1)
   {
      wait(S2);
      if(ba==0)
         wait(Sab);
      ba=ba+1;
      signal(S2);
      车辆从b点驶向a点;
      wait(S2);
      ba=ba-1;
      if(ba==0)
         signal(Sab)
      signal(S2)
   }
}
```

Some Exercises

- 理发师里有一个理发员为顾客理发，有5把椅子供顾客休息等待理发。如果没有顾客，则理发师休息。当顾客来到理发室时，如果有空椅子则坐下来，并唤醒理发师；如果没有空椅子则必须离开理发室。

  - customers表示正在等待复印的顾客数量（不包括正在复印的顾客）
  - operator记录正在等候顾客的操作员数，只有1和0
  - mutex用于对waiting的访问; waiting表示等待的顾客数量。之所以使用waiting是因为无法读取信号量的当前值。

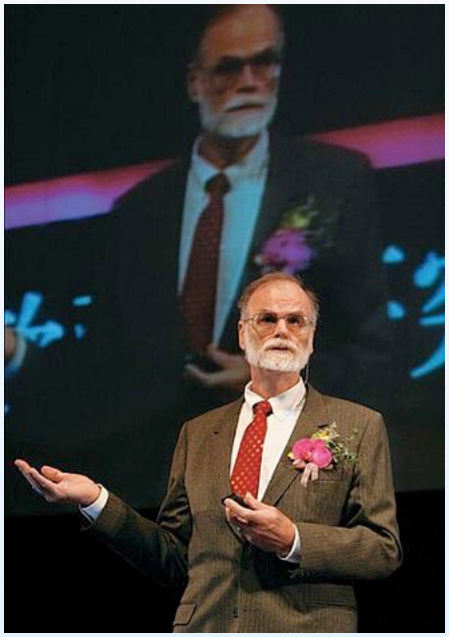
Some Exercises

```
semaphore customers=0,operator=0,mutex=1;
waiting=0;

process operator( )//理发师进程
{
   while(1)
   {
      wait(customers); //等待顾客到来
      理发;
      signal(operator); //通知顾客已经完成理发
   }
}
```

```
process cusotmer( )//顾客进程
{
    wait(mutex);
    if(waiting<5)
   {
      waiting++;
      signal(customers);
      signal(mutex);
      wait(operator);
      wait(mutex);
      waiting--;
      signal(mutex);
   }
   else
   {
      signal(mutex);
      离开理发室;
   }
}
```

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions
- ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability



James Nicholas "Jim" Gray (born January 12, 1944; lost at sea January 28, 2007; declared deceased May 16, 2012) was an American computer scientist who received the Turing Award in 1998 "for seminal contributions to database and transaction processing research and technical leadership in system implementation."

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

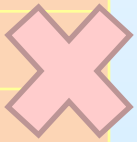
- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - $<T_i$ 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

Write-ahead Log

- Logging should write as the operating order
- It need to write log first and then update database

Logging	Database
$<T_0$ start>	
$<T_0$, A, 90, 100>	
$<T_0$, B, 150, 200>	
	A=100
	B=200
$<T_0$ commit>	

Logging	Database
$<T_0$ start>	
	A=100
	B=200
$<T_0$, A, 90, 100>	
$<T_0$, B, 150, 200>	
$<T_0$ commit>	

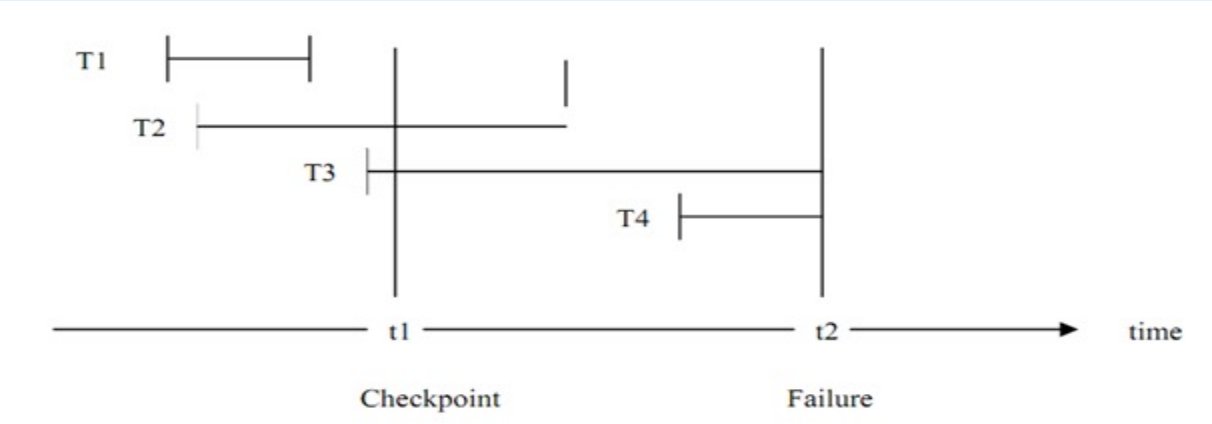


Log-Based Recovery Algorithm

- 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
- Now recovery only includes Ti, such that Ti started executing before the most recent checkpoint, and all transactions after Ti. All other transactions already on stable storage



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 and 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' (a serial schedule) via swapping nonconflicting operations
  - S is conflict serializable

Schedule 2: Concurrent Serializable Schedule

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

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
- Does not prevent deadlock

T1 get lock of A	T2 get lock of B
T1 get lock of B	T2 get lock A
Do something	Do something
T1 release lock of A	T2 release lock of B
T1 release lock of B	T2 release lock of A

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 $TS(T_i) < W\text{-timestamp}(Q)$, Ti needs to read value of Q that was already overwritten
    - read operation rejected and T_i rolled back
  - If $TS(T_i) \geq W\text{-timestamp}(Q)$
    - read executed, R-timestamp(Q) set to max(R-timestamp(Q), $TS(T_i)$)

Timestamp-ordering Protocol

- Suppose Ti executes write(Q)
  - If $TS(T_i) < R\text{-timestamp}(Q)$, 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 $TS(T_i) < W\text{-timestamp}(Q)$, 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)

Consistent Model

- Sequential consistency
  - The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program
- Causal consistency
  - A system provides causal consistency if memory operations that potentially are causally related are seen by every node of the system in the same order. Concurrent writes (i.e. ones that are not causally related) may be seen in different order by different nodes.
- Eventual consistency
  - if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value

The picture can't be displayed.

End of Chapter 6