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

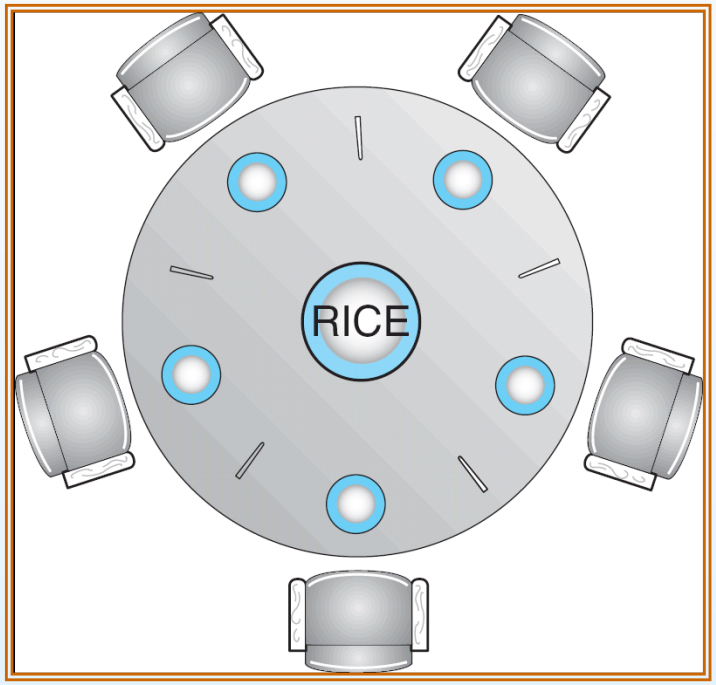
```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)

            // 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

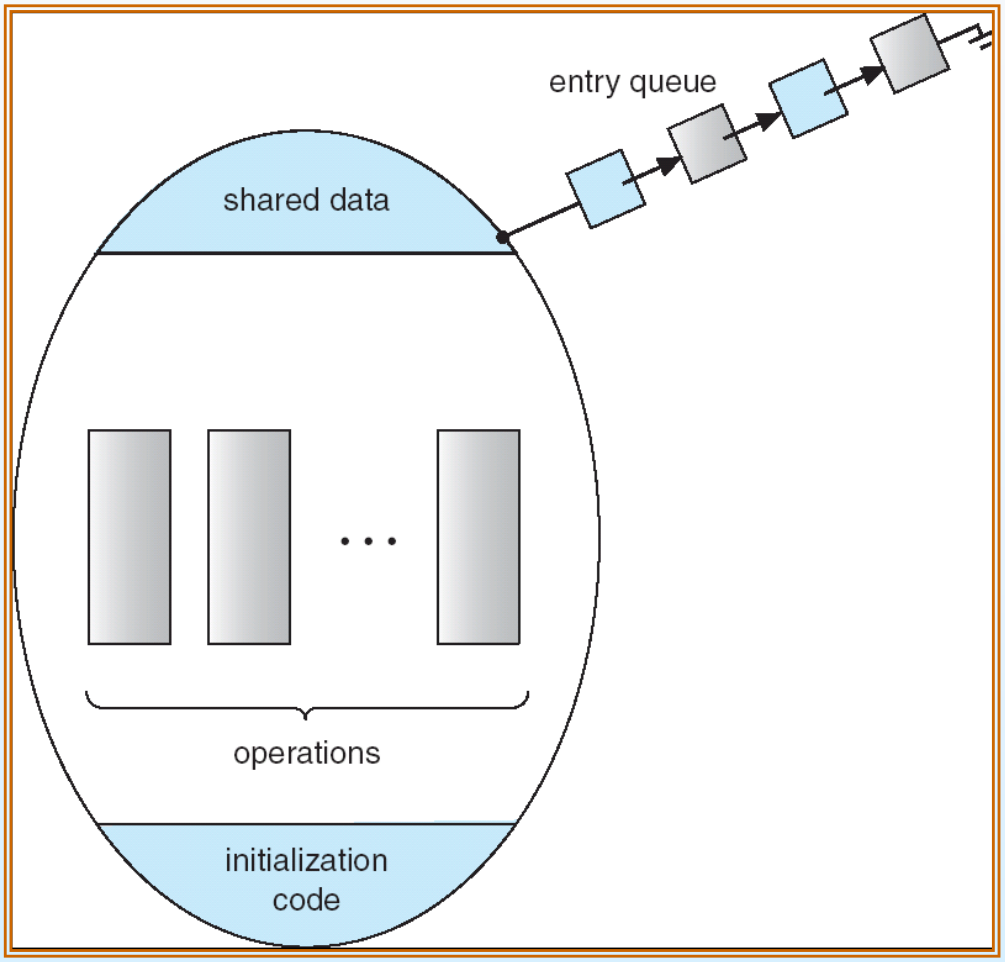
- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

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 ( ….) { … }
                …
    }
}
```

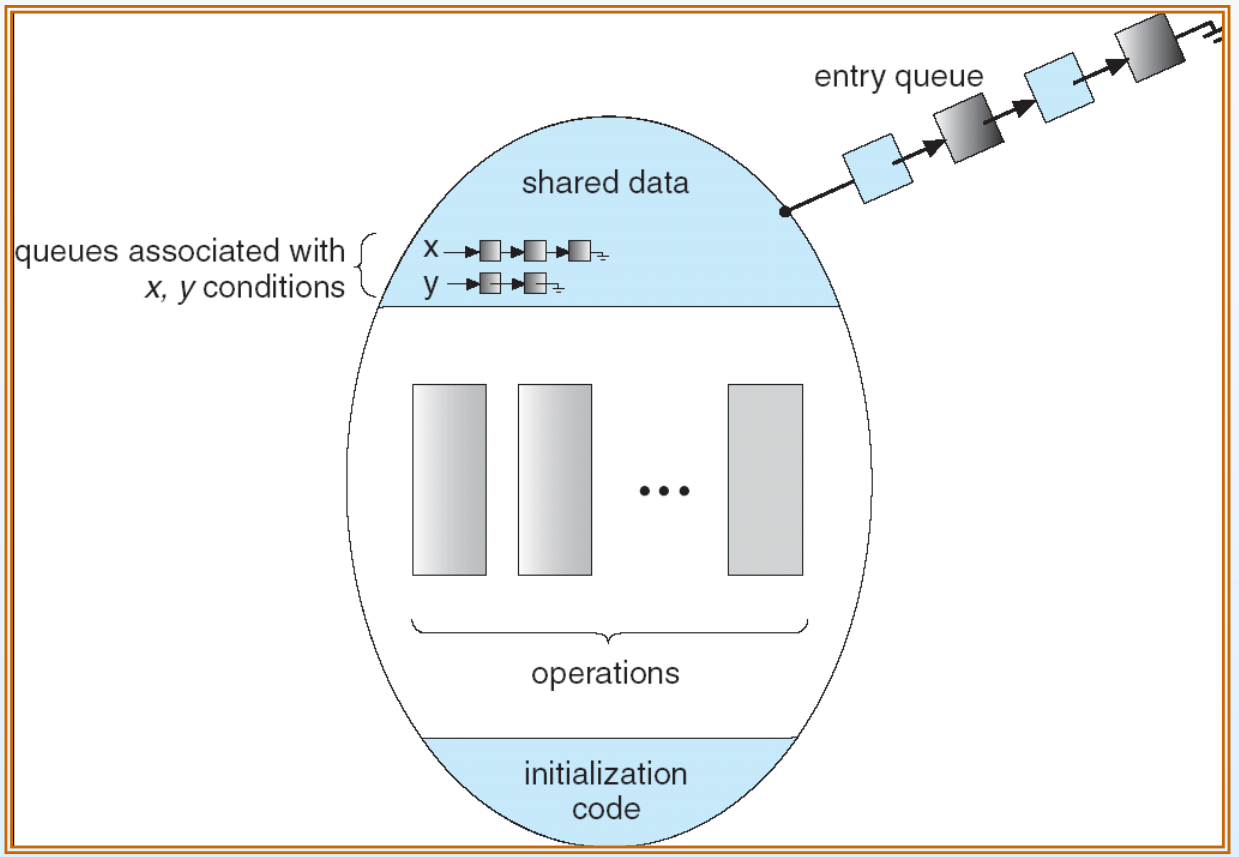
Schematic view of a Monitor



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， ISI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。

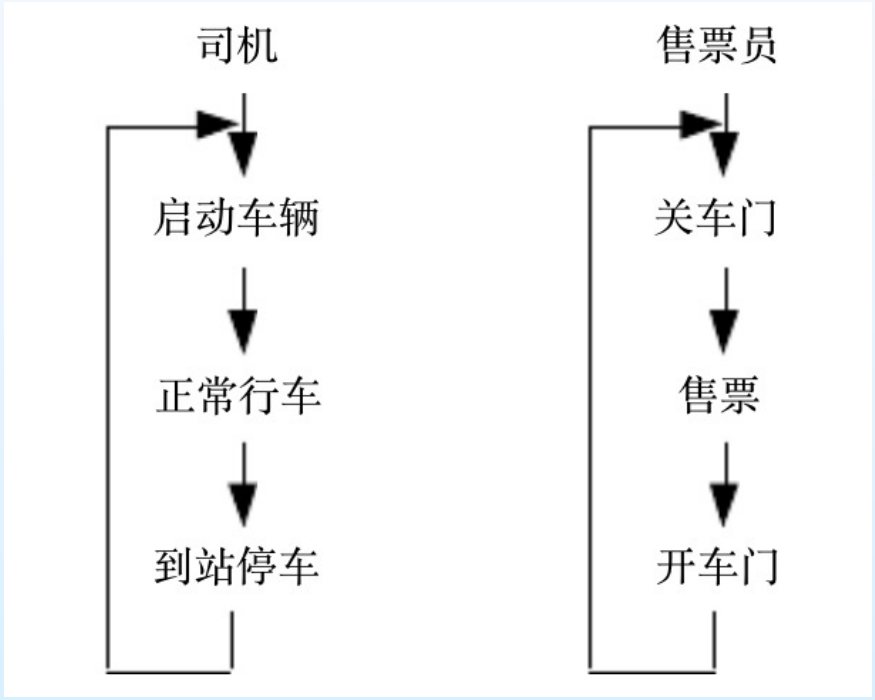
```
Sem S=1;   //盘子是否为空
Sem Sa=0;  //是否有苹果
Sem So=0;  //是否有橘子
main( )
{
    cobegin
    father();
    son();
    daughter();
    coend
}
```

```
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表示正在等待理发的顾客数量（不包括正在理发的顾客）
  - 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。

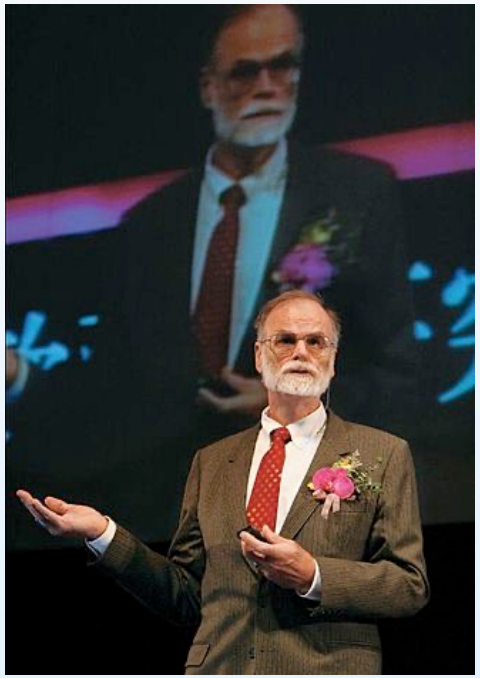
Some Exercises

```
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 customer( )//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

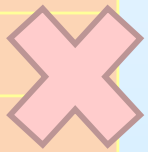
- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - Transaction name
    - Data item name
    - Old value
    - New value
  - $<T_i$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

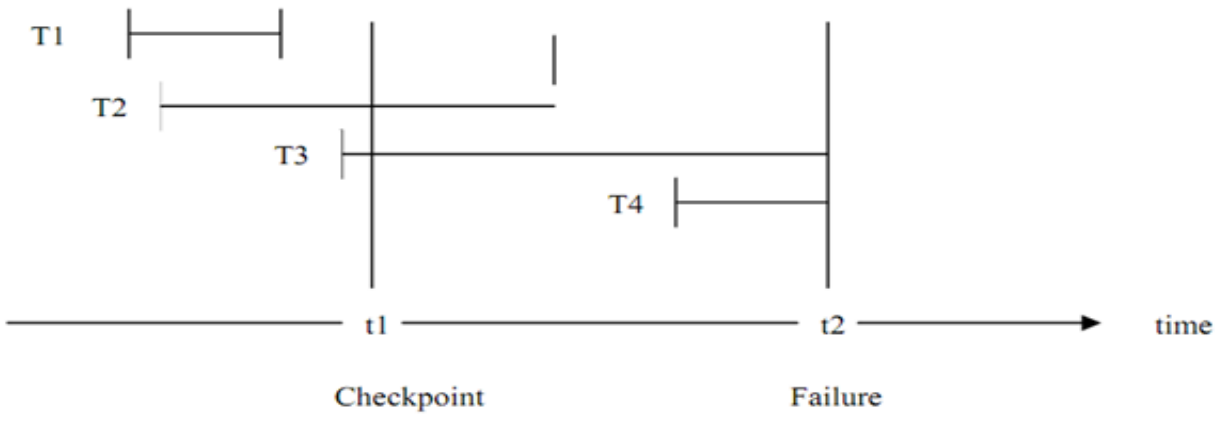


Log-Based Recovery Algorithm

- 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

查尔斯·巴赫曼

- 20世纪60年代中期以来，数据库技术的形成、发展和日趋成熟，使计算机数据处理技术跃上了一个新台阶，并且极大的推动了计算机的普及与应用。因此，1973年的图灵奖首次授予在这方面作出杰出贡献的数据库先驱查尔斯·巴赫曼（Charles W. Bachman）。

查尔斯·巴赫曼主要经历

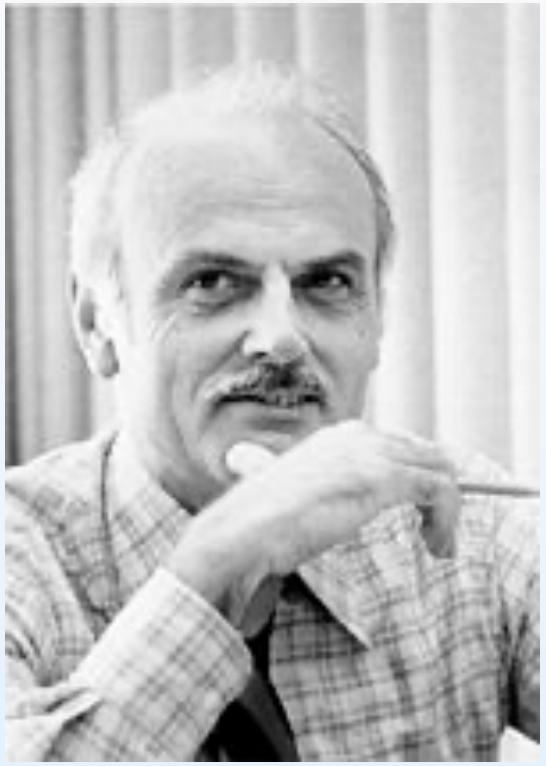
- 1924年12月11日生于堪萨斯州的曼哈顿；
- 1948年在密歇根州立大学取得工学学士学位；
- 1950年在宾夕法尼亚大学取得硕士学位；
- 20世纪50年代在Dow化工公司工作；
- 1961－1970年在通用电气公司任程序设计部门经理；
- 1970－1981年在Honeywell公司任总工程师，同时兼任Culllinet软件公司的副总裁和产品经理。
- 1983年巴赫曼创办了自己的公司Bachman Information System，Inc.。

查尔斯·巴赫曼主要成就

- 第一就是在通用电气公司任程序设计部门经理期间，主持设计与开发了最早的网状数据库管理系统IDS。IDS于1964年推出以后，成为最受欢迎的数据库产品之一，而且它的设计思想和实现技术被后来的许多数据库产品所仿效。
- 其二就是巴赫曼积极推动与促进了数据库标准的制定，那就是美国数据系统语言委员会CODASYL下属的数据库任务组DBTG提出的网状数据库模型及数据定义和数据操纵语言，即DDL和DML的规范说明，于1971年推出了第一个正式报告—DBTG报告，成为数据库历史上具有里程碑意义的文献。

埃德加·弗兰克·科德

- 埃德加·弗兰克·科德（Edgar Frank Codd）（1923年8月23日－2003年4月18日）是一位英国计算机科学家。他为关系型数据库理论做出了奠基性的贡献。他在IBM工作期间，首创了关系模型理论。他一生中为计算机科学做出了很多有价值的贡献，而关系模型，作为一个在数据库管理方面非常具有影响力的基础理论，仍然被认为是他最引人瞩目的成就。



埃德加·弗兰克·科德主要经历

- 1923年8月19日生于英格兰中部濒临大西洋的港口城市波特兰(Portland)；
- 参加第二次世界大战，在皇家空军服役，参与了许多惊心动魄的空战；
- 二战结束以后，Codd进入牛津大学学习数学，于1948年取得学士和硕士学位
- 毕业后在IBM公司取得职位，为IBM初期的计算机之一SSEC(Selective Sequence Electronic Calculator)编制程序，为他的计算机生涯奠定了基础。
- 1953年，他应聘到加拿大渥太华的Computing Device公司工作，出任加拿大开发导弹项目的经理。

埃德加·弗兰克·科德主要经历（续）

- 1957年Codd重返美国IBM，任“多道程序设计系统”的部门主任，其间参加了IBM第一台科学计算机701和第一台大型晶体管计算机STRETCH的逻辑设计。
- 1959年11月，他在《ACM通讯》上发表的介绍STRETCH的多道程序操作系统的文章，是这方面最早的学术论文之一。
- 在20世纪60年代初毅然决定重返大学校园(当时他已年近40)，到密歇根大学进修计算机与通信专业，并于1963年获得硕士学位，1965年又获得博士学位。
- 两年后，科德去往IBM公司位于圣何塞的阿尔马登研究中心工作。

埃德加·弗兰克·科德主要经历（续）

- 1970年6月，Codd在Communications of ACM上发表了题为“用于大型共享数据库的关系数据模型”一文。
- 1970年以后，Codd继续致力于完善和发展关系理论。1972年，他提出了关系代数和关系演算，为日后成为标准的结构化查询语言SQL 奠定了基础。
- Codd还创办了一个研究所：关系研究所(The Relational Institute)和一个公司：Codd&Associations，进行关系数据库产品的研发与销售。Codd本人则是美国国内和国外许多企业的数据库技术顾问。
- 1990年，编写出版了专著《数据库管理的关系模型》，全面总结了他几十年的理论探索和实践经验。

埃德加·弗兰克·科德主要成就

- 在数据库技术发展的历史上，1970年是发生伟大转折的一年。这一年的6月，Codd在Communications of ACM上发表了题为“用于大型共享数据库的关系数据模型”一文
- ACM在1983年把这篇论文列为从1958年以来的四分之一个世纪中具有里程碑式意义的最重要的25篇研究论文之一，因为它首次明确而清晰地为数据库系统提出了一种崭新的模型即关系模型。
- 用关系的概念来建立数据模型，用以描述、设计与操纵数据库，是Codd在1970年这篇论文中的创举。

A relational model of data for large shared data banks

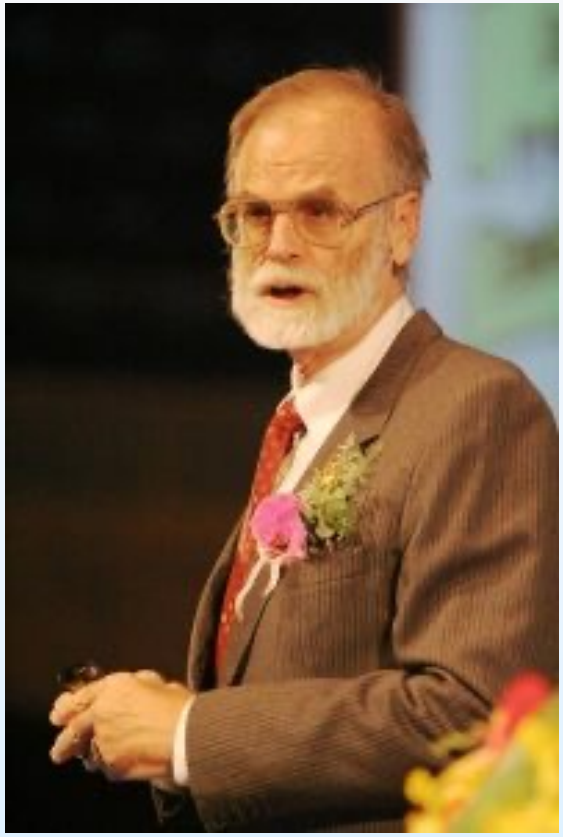
EF Codd - Software pioneers, 2002 - Springer

… support the **relational model** are not discussed. … **data** language of the **relational model** into corresponding-and efficient--actions on the current stored representation. For a high level **data** …

☆ Save Cite Cited by 13334 Related articles All 164 versions

詹姆斯·尼古拉·格雷

- 詹姆斯·尼古拉·格雷（James Nicholas Gray，1944年－），美国资讯工程学家。他提出了资料方块、锁定颗粒等概念。他也参与开发Windows Live Local。1998年度的图灵奖授予了声誉卓著的数据库专家詹姆斯·格雷。这是图灵奖诞生32年的历史上，继数据库技术的先驱者查尔斯.巴赫曼和关系数据库之父埃德加·科德之后，第三位因在推动数据库技术发展中作出重大贡献而获此殊荣的学者。



詹姆斯·尼古拉·格雷主要经历

- 他就读于柏克莱加州大学，1966年取得工程数学学士，1969年取得计算机科学博士。
- 学成以后在贝尔实验室、IBM、Tandem、DEC等公司工作，研究方向转向数据库领域。
- 在IBM期间，他参与和主持过IMS、System R、SQL/DS、DB2等项目的开发，其中除System R仅作为研究原型，没有成为产品外，其他几个都成为IBM在数据库市场上有影响力的产品。
- 在Tandem期间，格雷对该公司的主要数据库产品ENCOMPASS进行了改进与扩充，并参与了系统字典、并行排序、分布式SQL、Nonstop SQL 等项目的研制工作。

詹姆斯·尼古拉·格雷主要经历（续）

- 在DEC，他仍然主要负责数据库产品的技术。
- 1995年成为微软研究员，领导一个研制小组开发出了MS SQL Server 7.0，成为微软历史上一个里程碑式的版本，而且也成为当今关系数据库市场上的佼佼者。
- 1999年5月4日在亚特兰大举行的ACM全国会议上接受图灵奖，并发表了“信息技术今后的目标”的演说，纵论了信息技术发展中有关的几个方向性问题。

埃德加·弗兰克·科德主要成就

- 格雷进入数据库领域时，关系数据库的基本理论已经成熟，但各大公司在关系数据库管理系统（RDBMS）的实现和产品开发中，都遇到了一系列技术问题，即如何保障数据的完整性（Integrity）、安全性（Security）、并发性（Concurrency），以及一旦出现故障后，数据库如何实现从故障中恢复（Recovery）。
- 格雷在事务处理技术上的创造性思维和开拓性工作，使他成为该技术领域公认的权威。他的研究成果反映在他发表的一系列论文和研究报告之中，最后结晶为一部厚厚的专著 Transaction Processing: Concepts and Techniques。
- 格雷的另一部著作是The Benchmark Handbook: for Database and Transaction Processing Systems。

迈克尔·斯通布雷克

- 迈克尔·斯通布雷克 （Michael Stonebraker ） 被称为数据库领域的布道者，美国工程院院士，还曾获“冯诺依曼奖” 。他在1992年提出对象关系数据库模型。Stonebraker **发明了许多几乎应用在所有现代数据库系统中的概念，并且创立多家公司，成功地商业化了他关于数据库技术的开创性工作**。因“对现代数据库系统底层的概念与实践所做出的基础性贡献”，Michael Stonebraker获得2015年图灵奖。

迈克尔·斯通布雷克主要经历

- 生于1943年10月11 日，1966年在普林斯顿大学获得学士学位，分别于1967年和1971年在密歇根大学获得硕士和博士学位。

- 此后，他在加州大学伯克利分校计算机科学系任教达29年，在此期间领导开发了关系数据库系统Ingres、对象–关系数据库系统Postgres、联邦数据库系统Mariposa，同时也是以下数据库公司的创始人：Ingres、Illustra、Cohera、StreamBase Systems和Vertica等

- 迈克尔·斯通布雷克教授1992年凭借Ingres系统获得著名的ACM系统软件奖，

迈克尔·斯通布雷克主要经历（续）

- 1994年被ACM SIGMOD授予最佳年度创新奖，此外他还获得了IEEE颁发的冯·诺依曼奖章以及第一届SIGMOD Edgar F. Codd创新奖。
- 他于1994年成为ACM会士，1997年当选为美国国家工程院院士。
- 2015年因其在数据库系统大量的创新成就成为了数据库领域至今第4位图灵奖得主，

迈克尔·斯通布雷克主要成就

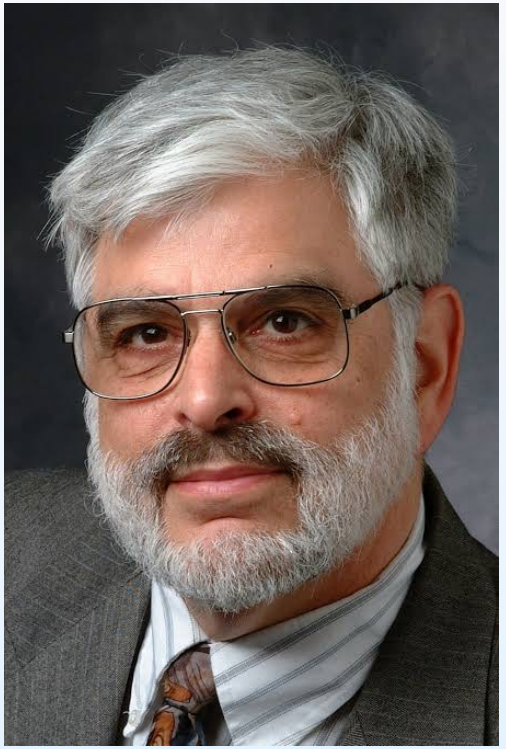
- 早在1970年代前期，Michael Stonebraker就在Edgar Codd的关系数据库论文启发下，组织伯克利的师生，开始开发最早的两个关系数据库之一Ingres。
- 1980年代他又开发了POSTGRES项目，目的是在关系数据库之上增加对更复杂的数据类型的支持，包括对象、地理数据、时间序列数据等。
- 1990年代，他启动了联邦数据库Mariposa，基于此创办了Cohera公司。Mariposa和稍早的XPRS和Distributed Ingres两个项目开了一代分布式数据库风起之先。
- 2002 年，开发流数据库 Aurora，以此创办 StreamBase 公司，产品用于许多金融机构的 CEP 系统。
- 2005 年，开发并行的列式数据仓库系统C-Store，创办 Vertica 公司。

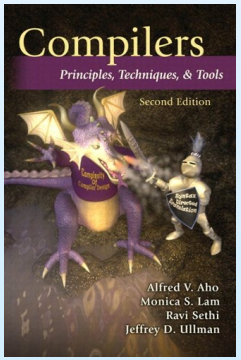
迈克尔·斯通布雷克主要成就(续)

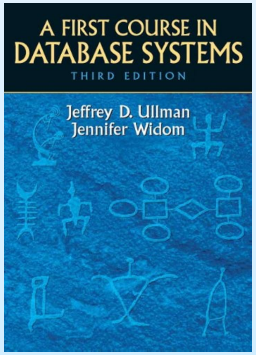
- 2006 年，开发数据集成项目 Morpheus，并据此创办本地搜索公司 Goby。
- 2007 年，开发分布式内存 OLTP 系统H-Store，创办 VoltDB 公司，已获得 1360 万美元投资。
- 2008 年，开发数组数据库 SciDB，创办 Paradigm4 公司。
- 2013 年，70 岁的他还与一个卡塔尔的年轻人共同创办企业数据集成公司 Tamr，次年获得 Google 等 1600 万美元投资。

Jeffrey Ullman (杰弗里·戴维·乌尔曼)

- Ullman 本科毕业于哥伦比亚大学，并普林斯顿大学获得计算机科学博士学位。
- Ullman 是斯坦福大学名誉教授，也是 Gradiance 公司的首席执行官；Gradiance 公司是一个针对各种计算机科学主题的在线学习平台。
- 他于 1979 年加入斯坦福大学，在斯坦福大学之前，他曾于 1969 年至 1979 年在普林斯顿大学任教，并于 1966 年至 1969 年在贝尔实验室担任技术人员。
- 合著有“龙书”《编译原理》、数据库名著《数据库系统实现》等多部经典著作；培养的多名学生已成为数据库领域的专家，其中包括谷歌联合创始人 Sergey Brin







Jeffrey Ullman

- Ullman在研究领域主要包括数据库理论、数据集成、数据挖掘，以及利用信息基础设施实现教育。他是数据库理论的奠基人之一，他是很多下一代数据库理论专家的博士生导师
- 撰写了10多本计算机教材