# Chapter 6:  Process Synchronization

# Module 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}
```

# 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

 }
```

# 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

# TestAndndSet 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 does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
        while S <= 0
            ; // no-op
          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
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

```
        Critical Section
    signal (S);
```

# 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 crtical 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 entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

- 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){
        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
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

# 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.
  - 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.)

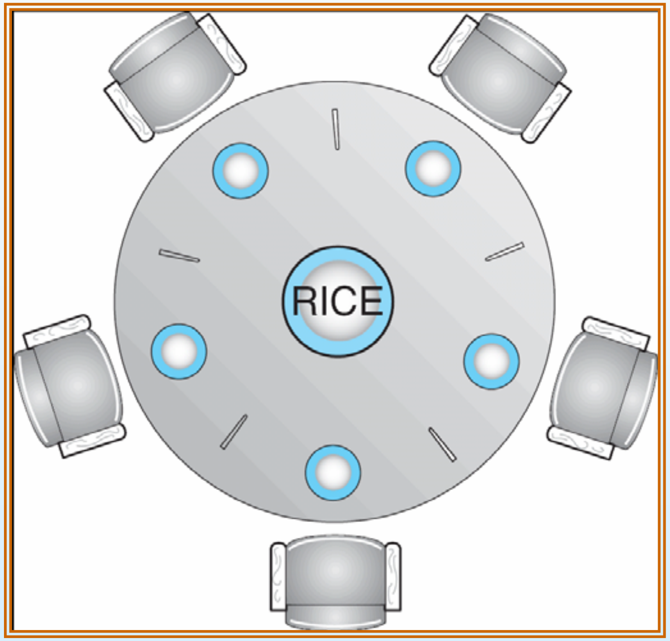
- The structure of a reader process

```
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) ;
}
```

# 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

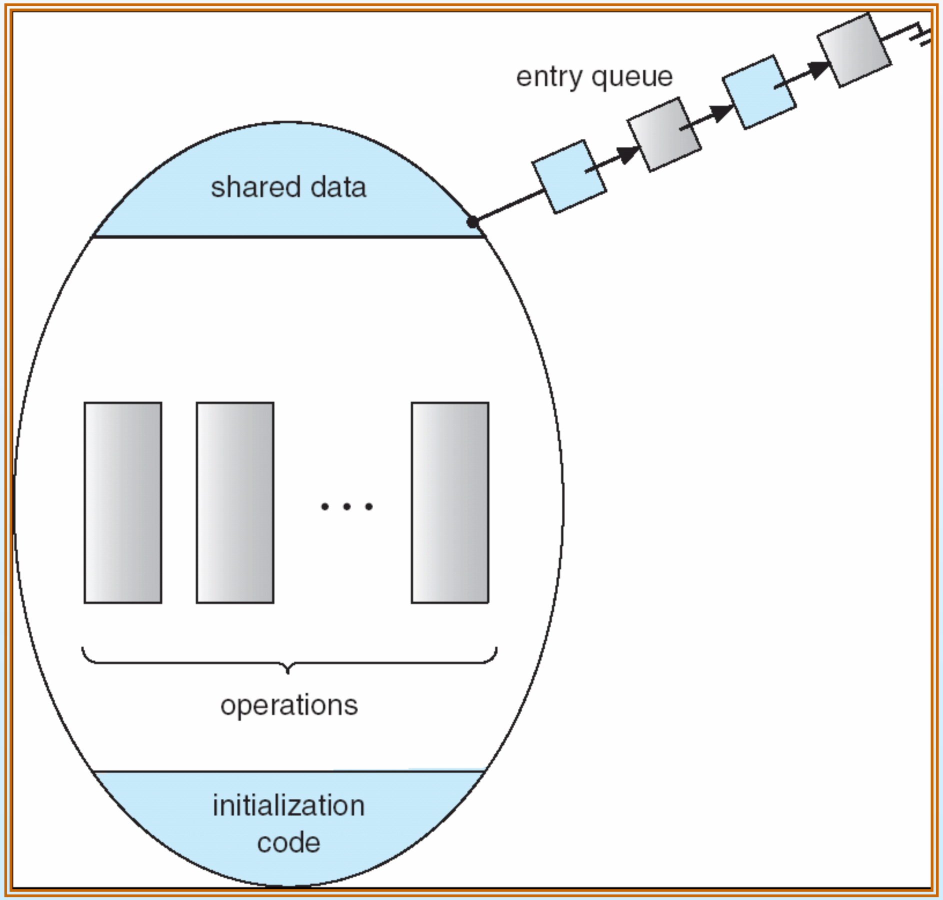
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

    procedure Pn (…) {……}

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

# Schematic view of a Monitor

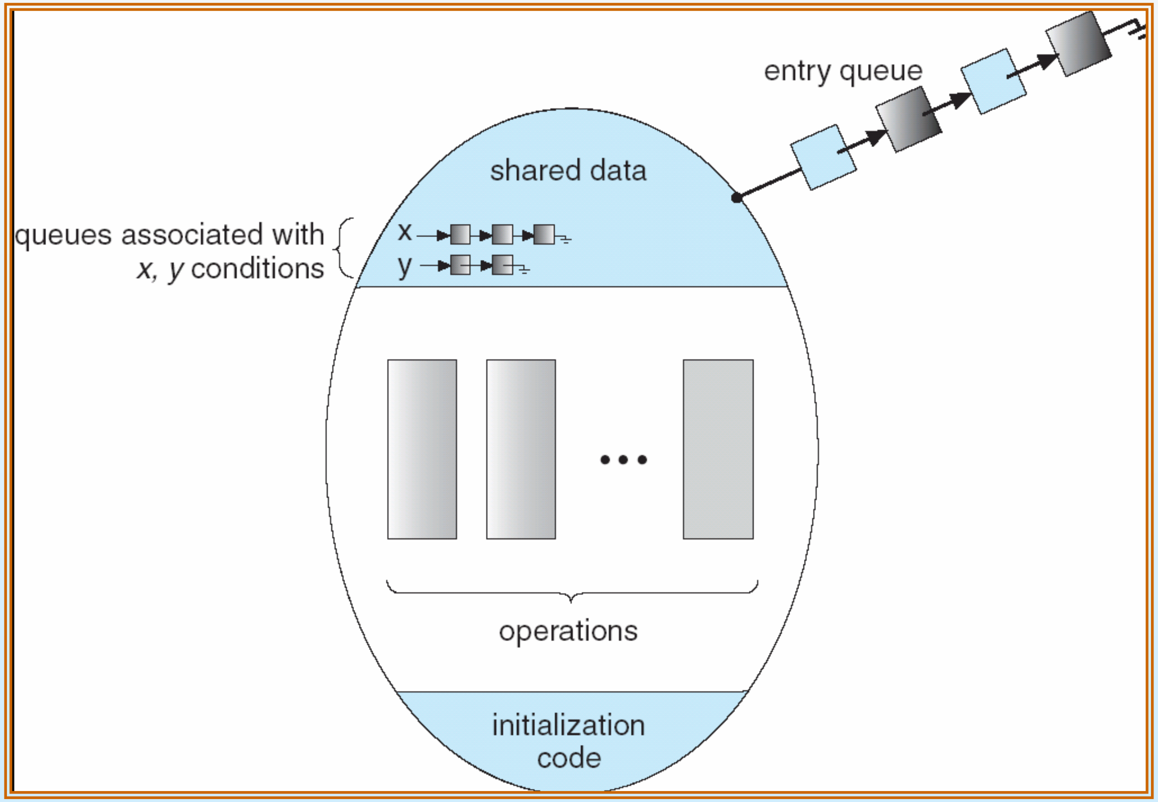


# Condition Variables

- 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];

     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)
```

# 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

- 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--;
```

# 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
- 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

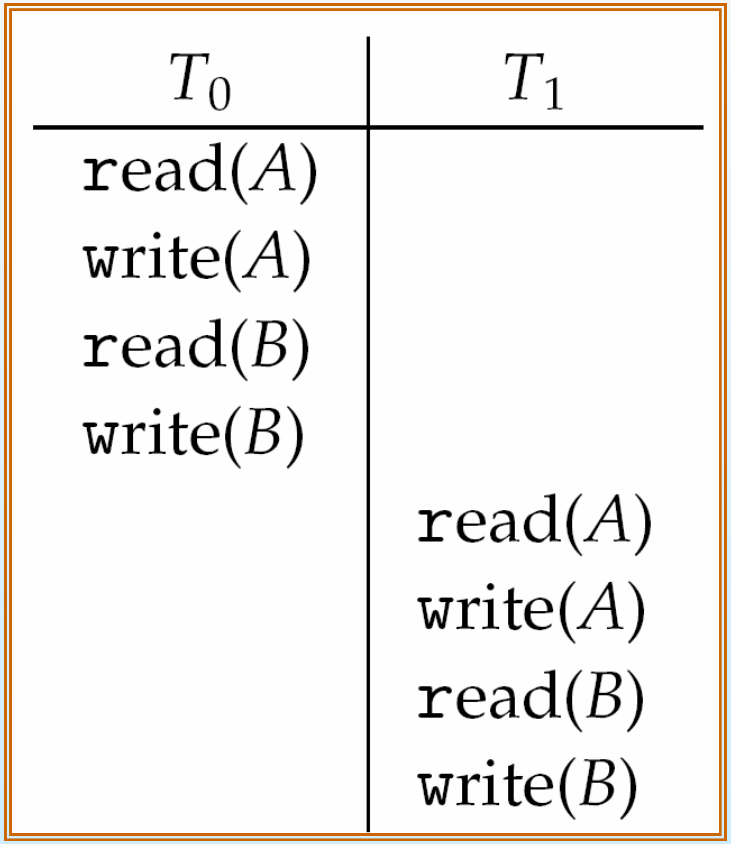
# Concurrent Transactions

- 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 $T_1$

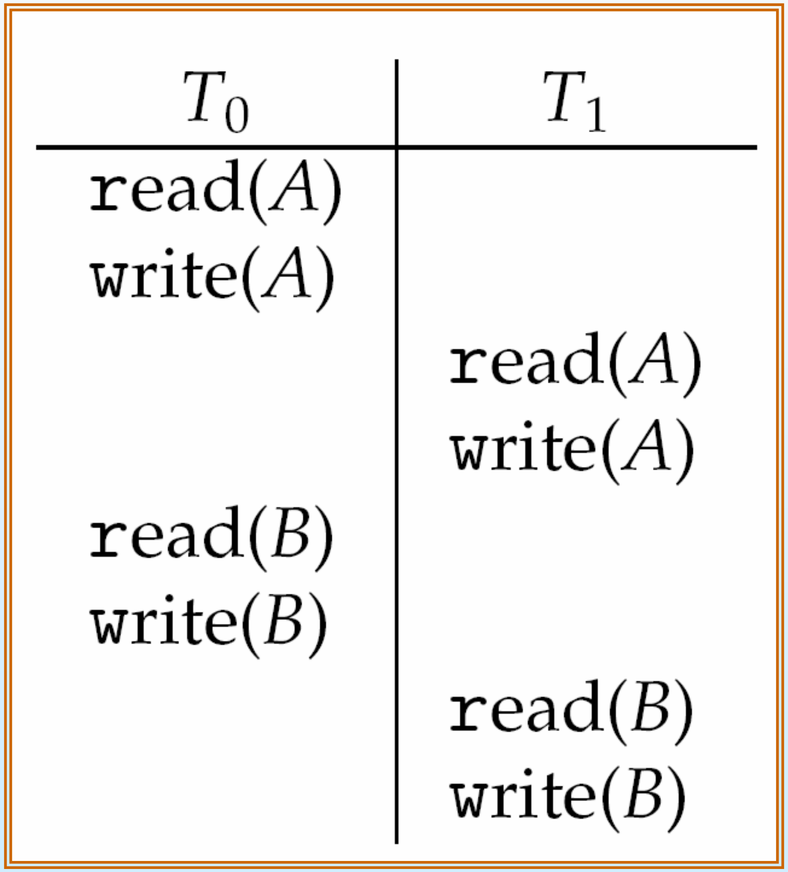


| $T_0$ | $T_1$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |

# 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' 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

# 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 $T_j$

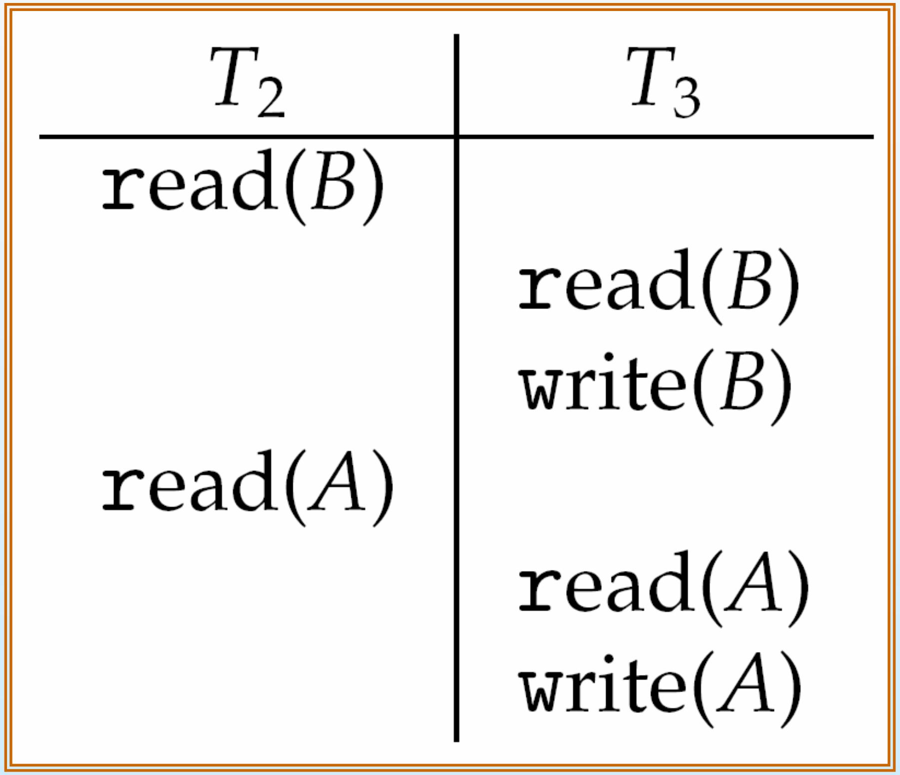
# Timestamp-based Protocol Implementation

- 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-tiimestamp(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($A$) |

# End of Chapter 6