Process Synchronization

Gaurav Somani

gaurav@lnmiit.ac.in

Reference : [Galvin] Operating System Principals, Seventh Edition, Silberschatz, Galvin and Gagne. http://www.os-book.com/

Contents

- Background
- The Critical-Section Problem
- Peterson's Solution
- Semaphores
- Classical 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 solutions of the critical-section problem
- To introduce the concept of an 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

Background (Cont.)

Shared Memory Implementation (POSIX)

1. **Create - segment_id =** shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR)
2. **Attach to process memory -** shared_memory = (char *) shmat (id, NULL, 0) ;
3. **Edit -** sprintf(shared_memory , "Writing to shared memory") ;
4. **Read -** printf("%s\n", shared_memory) ;
5. **detach -** shmdt (shared_memory) ;
6. **Remove –** shmctl (segment_id, IPC_RMID, NULL) ;

Producer- Consumer Problem

- Producer process produces information that is consumed by a consumer process.
- Producer fills all the buffers. Consumer use these buffers.
- Aim –
  - Producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
  - Producer does not try to produce an item if buffer is already full. (only in the case of *bounded buffer*)

- 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++;
}
```

in = next free position
out = first full position

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

1. Consider a system of n processes $\{P_0, P_1, \ldots, P_{n-1}\}$. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.
2. The Important feature of this is only one process is allowed to enter in critical section at a time.
3. The critical section problem is to design a protocol that the processes can use to cooperate/synchronize.

Solution to Critical-Section Problem

Criteria for an algorithm to become a solution for 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
    1. Assume that each process executes at a nonzero speed
    2. No assumption concerning relative speed of the N processes

Peterson's Solution

- A solution for two processes.
- *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

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

            critical section

    flag[i] = FALSE;

            remainder section

} while (TRUE);
```

Explanation : Peterson Solution

```
boolean flag[0..1] := false;
integer turn := 0 or 1;

entry(i) {
    flag[i] := true;
    turn := j;
    while flag[j] and turn=j
     do skip;
       }

  exit(i) {
    flag[i] := false;
       }
```

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
            critical section
    release lock
            remainder section
} while (TRUE);
```

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
    - Message Passing delays
    - System clock
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable

Semaphore

- 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 mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
        // remainder section
} while (TRUE);
```

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.
- Applications may spend lots of time in critical sections which results into *busy waiting* and therefore this is not a good solution. (Also known as *spinlock*)

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 as a structure

```
typedef struct {
        int value;
        struct process *list;
                }semaphore;
```

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

*Negative values of semaphore - magnitude represent no. of processes waiting

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
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Three semaphores mutex for whole buffer, full (No. of filled buffers), empty (No. of empty buffers)
- 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

```
do {
        // produce an item in nextp

    wait (empty);
    wait (mutex);

        // add the item to the buffer

     signal (mutex);
     signal (full);
} while (TRUE);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
        wait (full);
        wait (mutex);

                //  remove an item from  buffer to nextc

        signal (mutex);
        signal (empty);

                //  consume the item in nextc

} while (TRUE);
```

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

```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

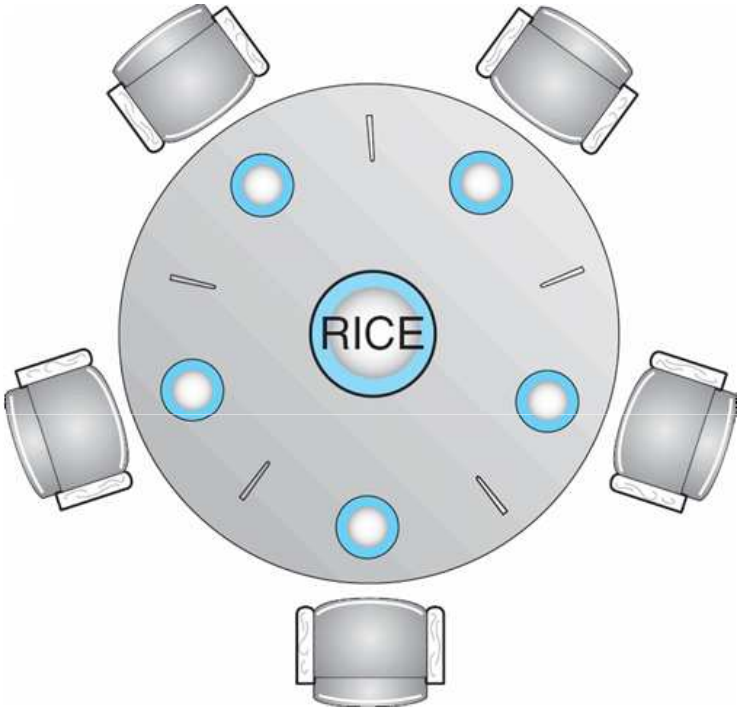
```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)

            // reading is performed

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

If writer is in CS and n readers are waiting than one reader is queued to wrt and n-1 readers are queued on 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*:

```
do  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

Some solutions

1. Allow a philosopher to pick up her chopsticks only if both chopsticks are available(Critical Section).
2. Asymmetric solution : odd philosopher picks up her left chopstick first and then right one. whereas an even philosopher will pick her right chopstick first and then left.
3. Allow at most four philosophers simultaneously at the table.

Problems with Semaphores

- Incorrect use of semaphore operations: can result into deadlocks and violation of mutual exclusion

  - signal (mutex)
    ….CS…….
    wait (mutex)

  - wait (mutex)
    ….CS…….
    wait (mutex)

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

Monitors

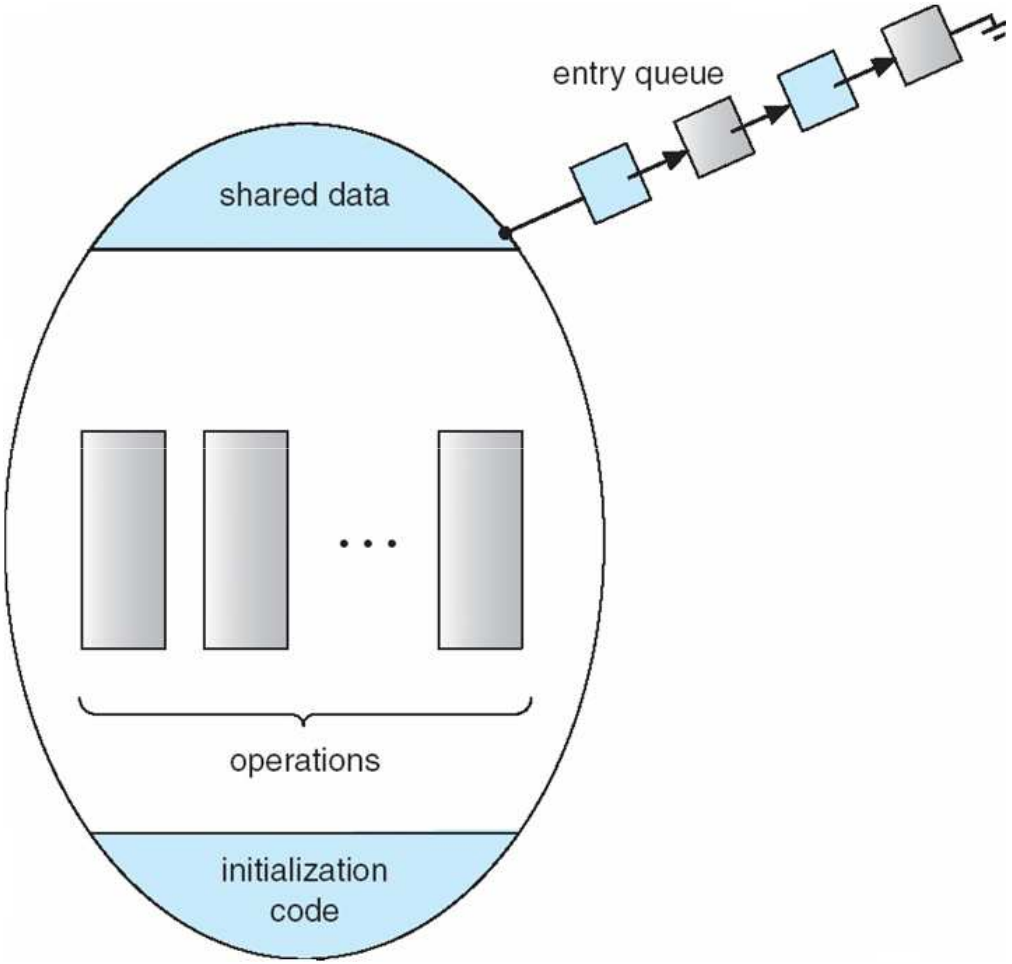
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

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

Schematic view of a Monitor

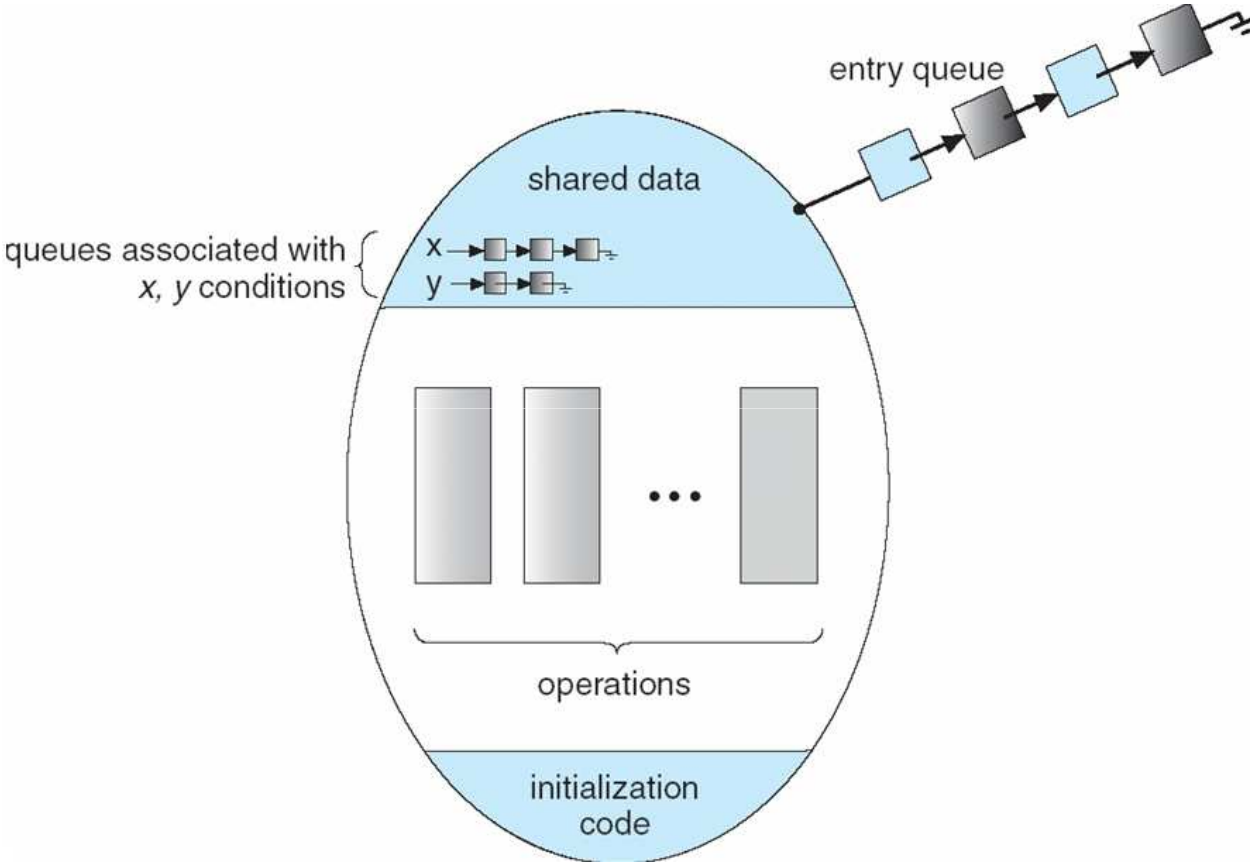


Condition Variables

- 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 (No. 1)

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

```
dp.pickup (i);

    EAT

dp.putdown (i);
```

Solution to Dining Philosophers

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

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

      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); // Give a chance to neighbors to eat
             test((i + 1) % 5);
      }
     initialization_code() {
            for (int i = 0; i < 5; i++)
            state[i] = THINKING;
      }}
```

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = TRUE;
    }
    void release() {
            busy = FALSE;
            x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive

- Single Processor – Disable/Enable kernel preemption.
- Multiprocessor – Acquire/Release spin lock.

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides: (refer thread tutorial)
  - 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
- 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-timestamp(Q)
    - Write operation rejected, T_i rolled back
  - If TS(T_i) < W-tiimestamp(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