Process Synchronization

- 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

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

while (true) {
    while (counter == 0); // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}

Race Condition

- counter++ could be implemented as:
  register1 = counter
  register1 = register1 + 1
  counter = register1

- counter-- could be implemented as:
  register2 = counter
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with “count = 5” initially:
  S0: producer execute register1 = counter {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = counter {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute counter = register1 {count = 6}
  S5: consumer execute counter = register2 {count = 4}
Critical Section Problem

- Consider system of $n$ processes $\{p_0, p_1, \ldots, p_{n-1}\}$

- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section

- Critical section problem is to design protocol to solve this

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

- Especially challenging with preemptive kernels

---

Critical Section

- General structure of process $p_i$ is

```c
do {
    entry section
    critical section
    exit section
    remainder section
} while (TRUE);
```
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 $\text{turn}$;
  - Boolean $\text{flag}[2]$

- The variable $\text{turn}$ indicates whose turn it is to enter the critical section.

- The $\text{flag}$ array is used to indicate if a process is ready to enter the critical section.
  - $\text{flag}[i] = \text{true} \rightarrow P_i$ is ready!
Algorithm for Process $P_i$

do {
    flag[i] = TRUE;
    turn = i;
    while (flag[j] && turn == j);  
        critical section
    flag[i] = FALSE;  
        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
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
    - Either test memory word and set value (testandset)
    - Or swap contents of two memory words (swap)
**Solution to CSP using Locks**

```c
# Solution to CSP using Locks

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

**TestAndSet Instruction**

```c
# TestAndSet Instruction

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

- Shared boolean variable lock, initialized to FALSE

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

Swap Instruction

- Definition:

```c
void Swap (boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
**Solution using Swap**

- Shared Boolean lock initialized to FALSE
- Local Boolean key

```c
do {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key);

    // critical section

    lock = FALSE;

    // remainder section
} while (TRUE);
```

---

**Bounded-waiting Mutual Exclusion - TestandSet()**

```c
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // remainder section
} while (TRUE);
```
Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore \( S \) – integer variable
- \( S \): \texttt{wait()} and \texttt{signal()} (atomic operations)
  - Originally called \texttt{P()} and \texttt{V()}
  - \texttt{wait (S) \{ }
    - \texttt{while S <= 0; // no-op}
    - \texttt{S--;}
  \}
  - \texttt{signal (S) \{ }
    - \texttt{S++;}
  \}

Semaphores – Synchronization Tools

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and simpler to implement
  - Also known as **mutex locks** (**mutual exclusion**)
- Can implement a counting semaphore \( S \) as a binary semaphore

```c
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
  - 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 – 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 – no Busy waiting

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

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

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
- Let S and Q be two semaphores initialized to 1
  ```
  P_0
  wait (S); wait (Q);
  wait (Q); wait (S);
  .
  .
  .
  signal (S); signal (Q);
  signal (Q); signal (S);
  ```

- **Starvation** – indefinite blocking
  - Process never removed from the semaphore's queue
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- $N$ buffers, each can hold one item
- Semaphore $\text{mutex}$ initialized to the value 1
- Semaphore $\text{full}$ initialized to the value 0
- Semaphore $\text{empty}$ initialized to the value $N$
Bounded Buffer – Producer

```c
do {
   // produce an item in nextp
   wait (empty);
   wait (mutex);

   // add the item to the buffer
   signal (mutex);
   signal (full);
} while (TRUE);
```

Bounded Buffer – Consumer

```c
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
  - 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

- Several variations of how readers and writers are treated – all involve priorities

- 1st RW problem
  - No reader has to wait unless a writer is in CS
  - Readers pass waiting writers

- 2nd RW Problem
  - Writers access CS asap
  - Writers pass waiting readers

Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore `mutex` initialized to 1
  - Semaphore `wrt` initialized to 1
  - Integer `readcount` initialized to 0
**RW Problem – Writer**

```plaintext
do {   wait (wrt) ;
    //  writing is performed
    signal (wrt) ;
} while (TRUE);
```

**RW Problem – Reader**

```plaintext
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);
```
Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Algorithm

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

    // eat

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

    // think
}
```

What is the problem with this algorithm?
Problems with Semaphores

- Incorrect use of semaphore operations:
  
  ```
  signal (mutex)  wait (mutex)  
  ... 
  wait (mutex)  wait (mutex)  
  ```

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

Monitors

- Abstract data type – predecessor of classes
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (...) { .... }
  procedure Pn (...) {......}
  Initialization code (...) { ... }
}
```
Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until x.signal ()
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - If no x.wait () on the variable, then it has no effect on the variable
Condition Variables Choices

- If process P invokes `x.signal()`, with Q in `x.wait()` state, what should happen next?
  - If Q is resumed, then P must wait

- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition

- Both have pros and cons – language implementer can decide
- Monitors implemented in Concurrent Pascal compromise
  - P executing signal immediately leaves the monitor, Q is resumed
- Implemented in other languages including Mesa, C#, Java
Java Monitors

- Java associates a monitor with each object.
- The monitor enforces mutual exclusive access to synchronized methods
- When a thread calls a synchronized method
  - JVM checks if the monitor is *unowned*
  - ownership is assigned to the calling thread
  - thread allowed to proceed with the method call
  - if the monitor is *owned* by another thread, thread put on hold
- Upon exit, it releases the monitor, allowing a waiting thread (if any) to proceed with its synchronized method call

```java
class Counter {
    private int count = 0;
    public void Increment() {
        int n = count;
        count = n + 1;
    }
}
```
Java Monitors

```
class Counter {
    private int count = 0;

    public void synchronized Increment() {
        int n = count;  // 0
        count = n + 1;  // 1
    }
}
```
Java Monitors

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter.increment();</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>(acquires the monitor)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>n = count;</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>count = n + 1;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(can't acquire monitor)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Java Monitors: Houston, we have a problem

class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: "+ n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: "+ n);    
    }
}
Java Monitors: Houston, we have a problem

class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}

Java Monitors: Houston, we have a problem

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}
Java Monitors: Houston, we have a problem

class PC {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}

Java Monitors: wait and notify

- Java includes an elegant interprocess communication mechanism
- `wait()`, `notify()`, and `notifyAll()` are implemented as `final` methods in `Object` – all classes have them
- Can be called only from within a `synchronized` method

- `wait()` tells the calling thread to give up the monitor and go to sleep until some other thread in the same monitor calls `notify()`
- `notify()` wakes up the first thread that called `wait()` on the same object
- `notifyAll()` wakes up all the threads that called `wait()` on the same object
Java Monitors: Fixed Solution

class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        if(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }
}

Java Monitors: Fixed Solution

synchronized void put(int n) {
    if(valueSet)
        try {
            wait();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
    this.n = n;
    valueSet = true;
    System.out.println("Put: " + n);
    notify();
}

Java Monitors: Fixed Solution

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}

Java Monitors: Fixed Solution

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}
Java Monitors: Fixed Solution

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
    }
}

Java Monitors: Houston, we have a problem

- // An incorrect implementation of a producer and consumer.
  class Q {
      int n;
      synchronized int get() {
          System.out.println("Got: " + n);
          return n;
      }
      synchronized void put(int n) {
          this.n = n;
          System.out.println("Put: " + n);
      }
  }

- class Producer implements Runnable {
      Q q;
      Producer(Q q) {
          this.q = q;
          new Thread(this, "Producer").start();
      }
      public void run() {
          int i = 0;
      }
  }
Solution to Dining Philosophers

monitor DiningPhilosophers
{
    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);
    }

    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 $\text{pickup()}$ and $\text{putdown()}$ in the following sequence:

  ```
  DiningPhilosophers.pickup (i);
  EAT
  DiningPhilosophers.putdown (i);
  ```

- No deadlock, but starvation is possible

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 – Condition Variables

- For each condition variable \( x \), we have:
  
  ```
  semaphore x_sem; // (initially = 0)
  int x_count = 0;
  ```

- The operation \( x\text{.wait} \) can be implemented as:
  
  ```
  x\text{-count}++; 
  if (next_count > 0)
  
  signal(next);
  else
  
  signal(mutex);
  wait(x_sem);
  x\text{-count}--;
  ```

Monitor Implementation (Cont.)

- The operation \( x\text{.signal} \) can be implemented as:
  
  ```
  if (x-count > 0) {
  
  next_count++;
  
  signal(x_sem);
  signal(x_sem);
  wait(next);
  next_count--;
  }
  ```
Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?

- FCFS frequently not adequate

- conditional-wait construct of the form x.wait(c)
  - Where c is priority number
  - Process with lowest number (highest priority) is scheduled next

A Monitor to Allocate Single Resource

```java
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;
    }
}
```
Pthreads Synchronization

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

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
  - \(\text{Undo}(T_i)\) restores value of all data updated by \(T_i\)
  - \(\text{Redo}(T_i)\) sets values of all data in transaction \(T_i\) to new values
- \(\text{Undo}(T_i)\) and \(\text{redo}(T_j)\) 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 \(<T_i\ commits>, \text{undo}(T_i)\)
  - If log contains \(<T_i\ starts>\) and \(<T_i\ commits>, \text{redo}(T_j)\)
Checkpoint

- 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 $T_1$

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>
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 & $O_i$ 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

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>
### Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - Shared – Ti 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 $TS(T_j)$ before $T_i$ starts
  - $TS(T_j) < TS(T_i)$ if $T_i$ 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 $T_i$ executes read($Q$)
  - If $TS(T_j) < W$-timestamp($Q$), $T_i$ needs to read value of $Q$ that was already overwritten
    - read operation rejected and $T_i$ rolled back
  - If $TS(T_j) ≥ W$-timestamp($Q$)
    - read executed, R-timestamp($Q$) set to max(R-timestamp($Q$), $TS(T_j)$)
### Timestamp-ordering Protocol

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

### Schedule Possible Under Timestamp Protocol

<table>
<thead>
<tr>
<th></th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td>write(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
</tr>
</tbody>
</table>