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

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

```java
while (count >= BUFFER_SIZE)
    // do nothing
    // add an item to the buffer
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
```

Consumer

```java
while (count == 0)
    // do nothing
    // remove an item from the buffer
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
```

Race Condition

- count++ could be implemented as
  ```java
  register1 = count
  register1 = register1 + 1
  count = register1
  ```
- count-- could be implemented as
  ```java
  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

Critical-Section Problem

1. Race Condition - When there is concurrent access to shared data and the final outcome depends upon order of execution.
2. Critical Section - Section of code where shared data is accessed.
3. Entry Section - Code that requests permission to enter its critical section.
4. Exit Section - Code that is run after exiting the critical section.

Proof of Correctness

1. Each process eventually enters its critical section.
2. No process is in its critical section after all processes have executed their entry sections.
3. No process is in its critical section after it has been granted permission to enter.
4. No process is in its critical section when another process is trying to enter.

Structure of a Typical Process

```c
while (true) {
    entry section
    critical section
    exit section
    remainder section
}
```

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 \( P_i \)

```c
while (true) {
    flag[i] = TRUE;
    turn = i;
    while (flag[j] && turn == j);
    critical section
    flag[i] = FALSE;
    remainder section
}
```

Critical Section Using Locks

```c
while (true) {
    acquire lock;
    critical section
    release lock;
    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-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Data Structure for Hardware Solutions

```java
public class Semaphore {
    private boolean value = false;
    public boolean acquire() { return value; }
    public void release() { value = false; }
    public boolean acquire() { return value; }
    public void release() { value = false; }
    public boolean acquire() { return value; }
    public void release() { value = false; }
}
```

Solution using GetAndSet

```java
// lock is shared by all threads
HardwareBit lock = new HardwareBit(false);
while (true) {
    while (lock.getAndSet(true))
        Thread.sleep(0);
    criticalSection();
    lock.set(false);
    remainedSection();
}
```

Solution using Swap

```java
// lock is shared by all threads
HardwareBit lock = new HardwareBit(false);
// each thread has a local copy of key
HardwareBit key = new HardwareBit(true);
while (true) {
    key.set(true);
    do {
        lock.swap(key);
    } while (key.get() == true);
    criticalSection();
    lock.set(false);
    remainedSection();
}
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S = integer variable
- Two standard operations modify S: acquire() and release()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```java
acquire() {
    while (value == 0) {
        // wait
        value=1;
    }
    release() {
    value=1;
}
```

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

```java
Semaphore S = new Semaphore();
S.acquire();
// critical section
S.release();
// remainder section
```
Semaphore Implementation

- Must guarantee that no two processes can execute acquire() and release() 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
  - 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 acquire():

```java
class SemaphoreFactory {
    private static void makeSuccessor[](int i) {
        SemaphoreSemaphore = new Semaphore Sơn();
        Thread[] threads = new Thread();
        for (int i = 0; i < 0; i++) {
            new() = new Semaphoremethod()
            new() = new Semaphoremethod()
            thread.run();
        }
    }
}
```

- Implementation of release():

```java
class SemaphoreFactory {
    private static void makePredecessor[](int i) {
        SemaphoreSemaphore = new Semaphore Sơn();
        Thread[] threads = new Thread();
        for (int i = 0; i < 0; i++) {
            new() = new Semaphoremethod()
            new() = new Semaphoremethod()
            thread.run();
        }
    }
}
```

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
  - P1
  - S.acquire();
  - Q.acquire();
  - ...
  - ...;
  - Q.release();
  - S.release();

- Starvation – infinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Classical Problems of Synchronization

- Bounded-Buffer 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

```java
public class boundedbuffer implements Buffer {
    private static final int BUFFER_SIZE = 8;
    private Object[] buffer;
    private int in = 0;
    private Semaphore empty;
    private Semaphore full;

    public boundedbuffer() {
        // buffer is initially empty
        in = 0;
        buffer = new Object[BUFFER_SIZE];
        empty = new Semaphore(0);
        full = new Semaphore(BUFFER_SIZE);
    }

    public void insert(Object item) {
        // Figure 6.9
        public Object remove() {
            // Figure 6.10
    }
}
```

Bounded-Buffer Problem

insert() Method

```java
public void insert(Object item) {
    mutex.acquire();
    mutex.acquire();
    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    mutex.release();
    mutex.release();
    full.release();
}
```

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```java
public class boundedbuffer implements Buffer {
    public Object remove() {
        mutex.acquire();
        Object item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        mutex.release();
        empty.release();
        return item;
    }
}
```
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```java
public class Consumer implements Runnable {
    private Buffer buffer;
    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }
    public void run() {
        while (true) {
            if (buffer.isFull()) {
                System.out.println("Buffer is full.");
            } else {
                System.out.println("Consuming item from the buffer");
                buffer.remove();
            }
        }
    }
}
```

Bounded Buffer Problem (Cont.)

- The Factory

```java
public class Factory {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        // Create producer and consumer threads
        Thread producer = new Producer(buffer);
        producer.start();
        Thread consumer = new Consumer(buffer);
        consumer.start();
    }
}
```

Dining-Philosophers Problem

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

```java
public class Producer implements Runnable {
    private Buffer buffer;
    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }
    public void run() {
        while (true) {
            // Get left chopstick
            chopStick[i].acquire();
            // Get right chopstick
            chopStick[i].acquire();
            System.out.println("Philosopher i is eating.");
            // Release chopstick
            chopStick[i].release();
            chopStick[i].release();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // ignored
            }
        }
    }
}
```

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher i

```java
while (true) {
    // Get left chopstick
    chopStick[i].acquire();
    // Get right chopstick
    chopStick[i].acquire();
    System.out.println("Philosopher i is eating.");
    // Release chopstick
    chopStick[i].release();
    chopStick[i].release();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // ignored
    }
}
```

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

```java
public class Monitor {
    public void acquire() {
        // acquire
    }
    public void release() {
        // release
    }
}
```

Problems with Semaphores

- Correct use of semaphore operations critical:
  - Misorder of:
    - `mutex.acquire()` ..... `mutex.release()`
  - Omission of:
    - `mutex.acquire()` and/or `mutex.release()`

```
while (true) {
    // Get left chopstick
    chopStick[i].acquire();
    // Get right chopstick
    chopStick[i].acquire();
    System.out.println("Philosopher i is eating.");
    // Release chopstick
    chopStick[i].release();
    chopStick[i].release();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // ignored
    }
}
```
Syntax of a Monitor

```java
monitor monitorName
{
    // shared variable declarations
    initialization code {...}

    public PS {...}
    ...
    public PS {...}
    ...
    public PS {...}
    ...
}
```

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

```java
public DiningPhilosophers
{
    new Proc(TAKEFORKS, MOUTH, RIGHT, LEFT, i); // each philosopher
    while (true)
    {
        checkVoids();
        takeForks(i);
        EAT;
        returnForks(i);
    }
}
```

Solution to Dining Philosophers (cont)

- Each philosopher `i` invokes the operations `takeForks(i)` and `returnForks(i)` in the following sequence:
  - `dp.takeForks(i)`
  - `EAT`
  - `dp.returnForks(i)`
Java Synchronization

- Java provides synchronization at the language-level.
- Each Java object has an associated lock.
- This lock is acquired by invoking a **synchronized** method.
- This lock is released when exiting the synchronized method.
- Threads waiting to acquire the object lock are placed in the **entry set** for the object lock.

Java Synchronization

- Each object has an associated **entry set**.

Java Synchronization

Synchronized insert() and remove() methods

```
public synchronized void insert(ThreadItem item) {  
    while (count == BUFFER_SIZE)  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            return;  
        }  
    item = buffer[count];  
    count = (count + 1) % BUFFER_SIZE;  
    notify();  
}

public synchronized void remove() {  
    while (count == 0)  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            return;  
        }  
    item = buffer[count];  
    count = (count + 1) % BUFFER_SIZE;  
    return item;  
}
```

Java Synchronization

Synchronized insert() and remove() methods

- When a thread invokes **wait()**:
  1. The thread releases the object lock;
  2. The state of the thread is set to Blocked;
  3. The thread is placed in the **wait set** for the object.
- When a thread invokes **notify()**:
  1. An arbitrary thread T from the wait set is selected;
  2. T is moved from the wait to the entry set;
  3. The state of T is set to Runnable.

Java Synchronization

- Entry and wait sets

Java Synchronization - Bounded Buffer

```
public class Buffer implements Buffer {  
    private static final int BUFFER_SIZE = 8;  
    private int count;  
    private ThreadItem[] buffer;  

    public Buffer() {  
        // In this method, we do nothing.  
        int i = 0;  
        while (i < BUFFER_SIZE)  
            buffer[i] = new ThreadItem();  
    
    public synchronized void insert(ThreadItem item) {  
        // Remove item from buffer.  
        buffer[count] = item;  
        count = (count + 1) % BUFFER_SIZE;  

    public synchronized void remove() {  
        // Remove item from buffer.  
        count = (count - 1) % BUFFER_SIZE;  
        buffer[count] = null;  
    }
```

Java Synchronization - Bounded Buffer

- Entry and wait sets
Java Synchronization - Bounded Buffer

```java
class BoundedBuffer implements Monitor {
    int count;
    int writePosition;
    int readPosition;
    final Object lock = new Object();
    synchronized void deposit(Object item) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
        }
        writePosition = (writePosition + 1) % BUFFER_SIZE;
        count = count + 1;
        notify();
    }
    synchronized void withdraw() {
        try {
            wait();
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
        }
        count = count - 1;
        notify();
    }
}
```

Java Synchronization

- The call to `notify()` selects an arbitrary thread from the wait set. It is possible the selected thread is in fact not waiting upon the condition for which it was notified.
- The call `notifyAll()` selects all threads in the wait set and moves them to the entry set.
- In general, `notifyAll()` is a more conservative strategy than `notify()`.

Java Synchronization - Readers-Writers

```java
public class ReentrantLock implements Lock {
    private final State state;
    private final Condition condition;
    // ... other methods...

    public synchronized void lock() {
        if (state != null) {
            throw new IllegalMonitorStateException();
        }
        state = new AcquireQueuedSema(1);
        condition = new ConditionObject();
        acquire(1);
    }
}
```

Java Synchronization - Readers-Writers

**Methods called by readers**

```java
public synchronized void tryLock(int forDuration) {
    for (int waitStatus = state.get(); waitStatus >= 0; waitStatus =
         state.get()) {
        if (isOnWaitQueueFor proposesAcquire(waitStatus)) {
            // ... acquire...
        } else {
            if (!compareAndSetState(waitStatus, waitStatus + 1)) {
                return;
            }
            // ... acquire...
        }
    }
}
```

Java Synchronization

Rather than synchronizing an entire method, blocks of code may be declared as synchronized

```java
synchronized (mutexLock) {
    criticalSection1();
}
```
Java Synchronization

Block synchronization using `wait()/notify()`

```java
Object mutexLock = new Object();
...
synchronized(mutexLock) {
    try {
        mutexLock.wait();
    } catch (InterruptedException ie) { }
}
```

Concurrency Features in Java 5

Semaphores

```java
Semaphore sem = new Semaphore(1);
try {
    sem.acquire();
    // critical section
} catch (InterruptedException ie) { } finally {
    sem.release();
}
```

Concurrency Features in Java 5

A condition variable is created by first creating a `ReentrantLock` and invoking its `newCondition()` method:

```java
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

Once this is done, it is possible to invoke the `await()` and `signal()` methods.

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

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, O_j don't conflict
    - Then S' with swapped order of O_i, O_j equivalent to S
  - If S can become S' via swapping nonconflicting operations
    - S is conflict serializable

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>
### 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 – $T_i$ has shared-mode lock (S) on item Q, $T_i$ can read Q but not write Q
  - Exclusive – $T_i$ 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
    - $TS(T_i) < TS(T_j)$ 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 $Ti$ 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 $Ti$ rolled back
  - If $TS(T_i) ≥ W$-timestamp(Q), $Ti$ 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 $TS(T_i) < W$-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 $Ti$ is assigned new timestamp and restarted
  - Algorithm ensures conflict serializability and freedom from deadlock
Schedule Possible Under Timestamp Protocol

<table>
<thead>
<tr>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
</tbody>
</table>

End of Chapter 6