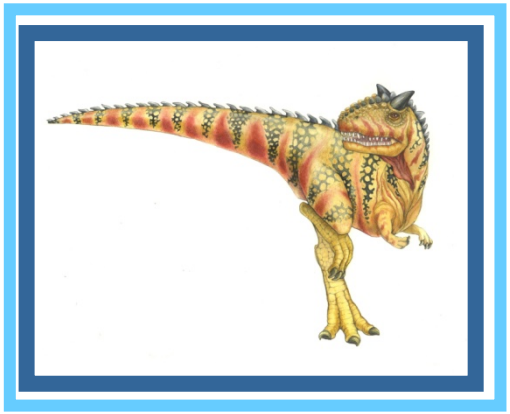
# Chapter 5: Process Synchronization



# Copyright

These slides have been modified by Dr. Marek A. Suchenek © in February 2012.

He reserves all rights for the said modifications.

Any copying, printing, downloading, sharing, or distributing without the permission of the copyright holder or holders is prohibited.

Permission for classroom use by the students currently enrolled in CSC 341 and CSC 541 course is granted for the duration of this semester.

# Module 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic 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 both software- and hardware-based 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
- 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

- The problem here is that both the Reader and the Writer can modify the variable count.

# Race Condition

- The problem here is that both the Reader and the Writer can modify the variable count.

If this happens at about the same time then the result may be unpredictable and incorrect.

# Race Condition

- count++ in the Producer could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- in the Consumer could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer executes register1 = count {register1 = 5}
S1: producer executes register1 = register1 + 1 {register1 = 6}
S2: consumer executes register2 = count {register2 = 5}
S3: consumer executes register2 = register2 - 1 {register2 = 4}
S4: producer executes count = register1 {count = 6 }
S5: consumer executes count = register2 {count = 4}

# Critical Section

- General structure of process $P_i$

```
do {
        entry section

        critical section

        exit section
        reminder section
} while (1);
```

# Critical Section

- *n* processes all competing to use (access and modify) some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Goal – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

# Critical-Section Problem

- **Problem statement**:

How to synchronize a set of concurrent processes so that the following criteria are met:

1. Mutual Exclusion - If a process is executing in its critical section then no other processes can be executing in their *respective* critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their *respective* critical sections, 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 *respective* critical sections after a process has made a request to enter its critical section and before that request is granted

# Critical-Section Problem

- **Problem statement**:

How to synchronize a set of concurrent processes so that the following criteria are met:

1. Mutual Exclusion - If a process is executing in its critical section then no other processes can be executing in their *respective* critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their *respective* critical sections, 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 *respective* critical sections after a process has made a request to enter its critical section and before that request is granted

Example: Bank teller

# 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 (for their *respective* critical sections):
  - 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 $P_i$

Here, i = 0, 1, is this process' ID

```
do {
    flag[i] = TRUE;
    turn = 1 - i; //the other process' ID
    while (flag[1 - i] && turn == 1 - i);
        critical section
    flag[i] = FALSE;
        remainder section
} until (FALSE);
```

# 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 is not broadly scalable
- Some modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

# Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
            critical section
    release lock
            remainder section
} until (FALSE);
```

# TestAndSet Instruction

- Definition (uses C pointers):

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

# Solution using TestAndSet

- Shared boolean variable lock, initialized to false.
- Solution:

```
do {
        while (TestAndSet (&lock))
                ;   // do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

} until (FALSE);
```

# Solution using TestAndSet

- This solution satisfies the Mutual Exclusion criterion and the Progress criterion, but ...

# Solution using TestAndSet

- This solution satisfies the Mutual Exclusion criterion and the Progress criterion, but ...
- It does not satisfy the Bounded Waiting criterion.

# 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:

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

                //    critical section

         lock = FALSE;

                //      remainder section

 } until (FALSE);
```

# Solution using Swap

This solution satisfies the Mutual Exclusion criterion and the Progress criterion, but ...

# Solution using Swap

This solution satisfies the Mutual Exclusion criterion and the Progress criterion, but ...

It does not satisfy the Bounded Waiting criterion.

# Bounded-waiting Mutual Exclusion with TestAndSet()

```
// Process Pi with private variable key
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

**NOTE**

- The “definitions” of wait and signal in the section 5.6 Semaphores in the textbook are **not** definitions.

- All the students are required to follow the definitions presented in these slides.

# 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 system calls (indivisible operations from the point of view of user programs)
  - wait (S) {
    S--;
    if S < 0 then move the process
    to the waiting (for S) queue
    }
    signal (S) {
    S++;
    if S <= 0 then move the first process waiting for S
    to the ready queue
    }

# 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
- **Incorrect code in your textbook:**
  - wait (S) {

```
        while S <= 0
          ; // no-op
        S--;
    } "Design a house around this."
```

  - signal (S) {

```
      S++;
// code is missing here
    }
```

) was originally called V (from *verhogen,* "to inc
it() is as follows:

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

of signal() is as follows:

```
signal(S) {
    S++;
}
```

ications to the integer value of the semaphore in the

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

phore has an integer value and a list of processes list.
nust wait on a semaphore, it is added to the list of proce
operation removes one process from the list of waiting pr
ns that process.
ne wait() semaphore operation can be defined as

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

# Semaphore

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of S ($S \leq 0$), as well as its possible modification (`S--`), must be executed without interruption. We shall see how these operations can be implemented in Section 5.6.2. First, let's see how semaphores can be used.

- **Incorrect code in your textbook:**
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    } "Design a house around this."
  - signal (S) {
    S++;
    // code is missing here
    }

# Semaphore as General Synchronization Tool

- Execute B in $P_0$ only after A in $P_1$.
- Let flag be a semaphore initialized to 0

| $P_0$ | $P_1$ |
|---|---|
| . . . | . . . |
| wait (flag); | |
| B | |
| . | . |
| . | . |
| . | A |
| | signal (flag); |

# Semaphore as General Synchronization Tool

- Counting semaphore – integer value (usually non-negative) 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
- One can implement a counting semaphore S using a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
    // remainder section
} until (FALSE
```

# Semaphore Implementation

- wait () and signal () are implemented as **system calls**
- This guarantees that no two user processes can execute wait () and signal () on the same semaphore at the same time
- **It makes busy waiting superfluous.**

# Semaphore Implementation - corrections

- With each semaphore there is an associated waiting queue. Each semaphore is composed of two data items:
  - value (of type integer)
  - list (a pair of pointers: to PCB of the first process and the last process in the queue)

- Two operations:
  - block() – place the process invoking the operation on the appropriate waiting queue remove the invoking process from the ready queue.
  - wakeup(P) – remove one of processes in the waiting queue and place it process P in the ready queue.

# Semaphore Implementation 3

- Copyright Dr. Marek A. Suchenek 2006 - 2025

- typedef struct {
- int value; // non-negative
- struct process *list; // a FIFO queue
- } semaphore;

# Semaphore Implementation 4

- Copyright Dr. Marek A. Suchenek 2006 - 2025

- wait(semaphore *S) /* a system call */ {
- if (S->value /* same as (*S).value */ == 0) }
- remove this process from the ready queue; // block()
- add this process to the end of S->list // same as (*S).list }
- else
- S->value--}

# Semaphore Implementation 5

- Copyright Dr. Marek A. Suchenek 2006 - 2025

- signal(semaphore *S) /* a system call */ {
- if (S->list != null) {
- remove P (the first process) from S->list;
- add P to the ready queue; // wakeup(P)}
- else
- S->value++}

# Semaphore Implementation - corrections(2)

- The following material is **optional**.
- Implementation of wait on a semaphore that admits negative values (this is the code from your textbook):

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

- Implementation of signal on a semaphore that admits negative values:

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

# Deadlocks and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes
- **Formal definition**:
- A deadlock set S is a finite set of processes
- Each element of S is waiting for an event that can only be caused by an element of S.
- A deadlock occurs iff there is a non-empty deadlock set S.
- Each element of S is called a deadlocked process.

# Deadlocks and Starvation 2

- 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); |

- S = {$P_0$, $P_1$} is a non-empty deadlock set.

# Deadlocks and Starvation 3

- Starvation – a process will never become running.
- Indefinite blocking - a process will never be removed from a waiting queue in which it is suspended (synchronization starvation), or it will never get selected for execution by the scheduler when it gets to the ready queue (scheduling starvation).
- Starvation may be circumstantial, while deadlock is a provable condition.
- One can recover from starvation, but not from a deadlock.

# Deadlocks and Starvation 3

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

- Remedy: temporarily grant the priority of Q to P

# Classical Problems of Synchronization

- Producer-Consumer Bounded-Buffer Problem
- Readers and Writers Problem (2)
- Dining-Philosophers Problem

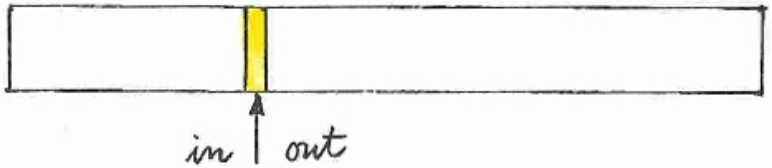
# Consumer-Producer Problem

- *N* buffers, each can hold one item
- SOLUTION with semaphores:
- **One Consumer and one Producer**
- ~~Semaphore mutex initialized to the value 1~~
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

- INVARIANT: empty + full = N

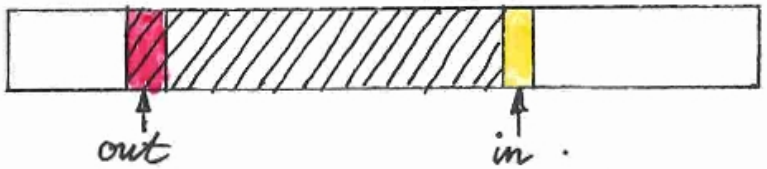
Buffer empty

(1)



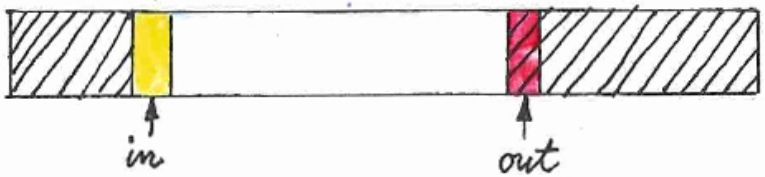

Buffer parially filled

(2)



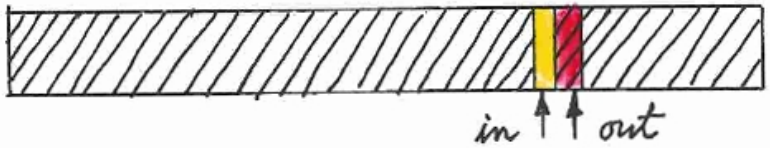

Buffer partially filled, wrapped around

(3)

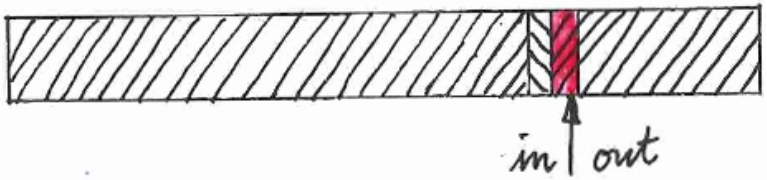



Buffer full

(4)




(5)




The above shows what would happen if the last ellement were allowed to be filled

# Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do  {//INVARIANT is true here (#p1) if consumer is at #c1 or #c2
        //   produce an item in nextp
      wait (empty);
      buffer[in] = nextp; // (#p0)
        //  add the item to the  buffer
       in = (in+1)%N;
       signal (full);
//INVARIANT is true here (#p2) if consumer is at #c1 or #c2
    } until (FALSE);
```

# Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do { //INVARIANT is true here (#c1) if producer is at #p1 or #p2
        wait (full);
        nextc = buffer[out]; // (#c0)
                //  remove an item from  buffer to nextc
        out = (out+1)%N;
        signal (empty);
//INVARIANT is true here (#c2) if producer is at #p1 or #p2
                //  consume the item in nextc
} until (FALSE);
```

# Bounded Buffer Problem (Cont.)

- Semaphore mutex in your textbook's code was used to assure that the producer and the consumer do not access the same element of the buffer at the same time.
- Such a situation can only happen when
  - in == out and
  - the producer is at #p0 and
  - the consumer is at #c0.
- This, however, cannot be the case.
  - If the producer is at #p0 then the buffer is not full.
  - If the consumer is at #c0 then the buffer is not empty.
  - Therefore, under the above assumptions, in != out (see the picture 3 slides age to find our why).
- Hence, the producer and the consumer are not accessing the same element of the buffer at the same time.
- Therefore, mutex in your textbook's code was superfluous.

# 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 as many multiple readers as possible to read at the same time, but if a writer performs an operation (read or write) on the data set, no other process has an access to the data set.**
- 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) ;

            //    writer is accessing the data set

        signal (wrt) ;
} until (FALSE);
```

# Readers-Writers Problem (Cont.)

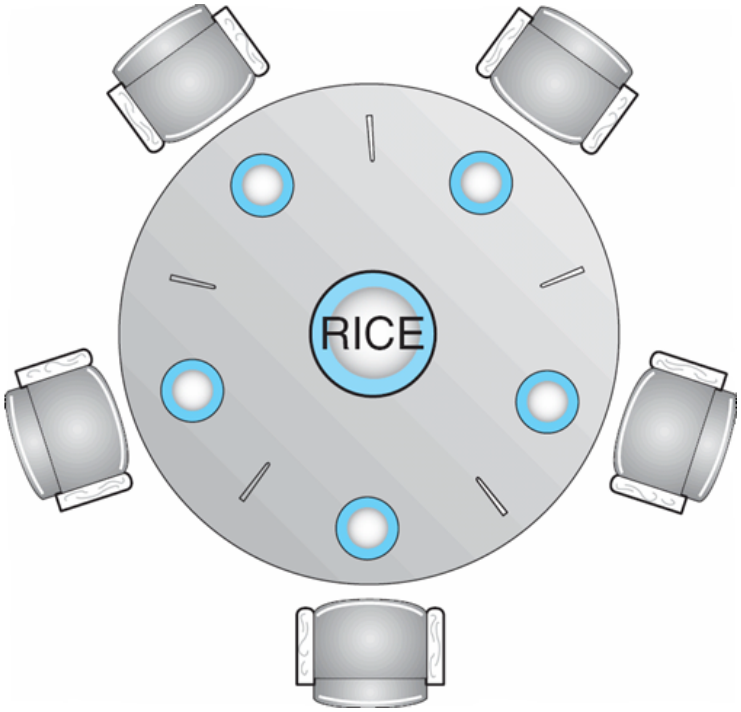
- The structure of a reader process

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

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (wrt) ;
        signal (mutex) ;
    } until (FALSE);
```

# Readers-Writers Problem (Cont.)

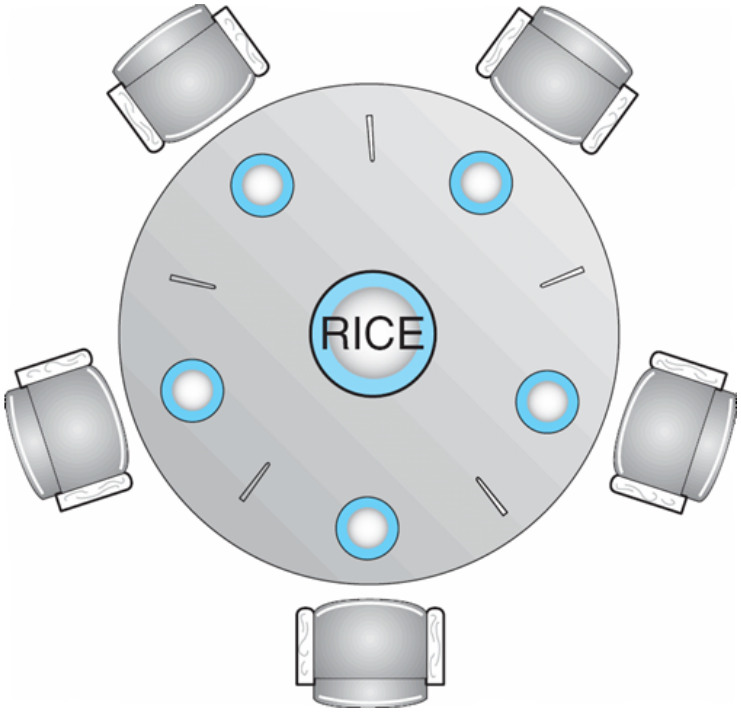
- The above solution (as well as any other correct solution) of the readers-writes problem stated 3 slides ago may cause starvation of writers.
- In order to prevent that possibility form happening, the problem may be modified by adding and extra requirement:
  - **If a writer is waiting to access the data set, no new readers can be allowed to begin reading it.**
- The above is called the **second readers-writers problem**.
- Any correct solution of the second readers-writers problem may cause starvation or readers as long as there is more than one writer.

# Dining-Philosophers Problem




- Shared data
  - Bowl of rice (data set)
  - Array of semaphores chopstick [5] ,all initialized to 1

# Dining-Philosophers Problem (2)




- **Problem: How to synchronize them that no chopstick is used by more than one philosophers at a time?**

# Solution to Dining-Philosophers Problem

- The structure of Philosopher *i*:

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

            //  eat

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

            //  think

} until (FALSE);
```

# Troubles with Semaphores

- Examples of incorrect 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
   function P1 (…) { …. }
     …

   function Pn (…) {……}

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

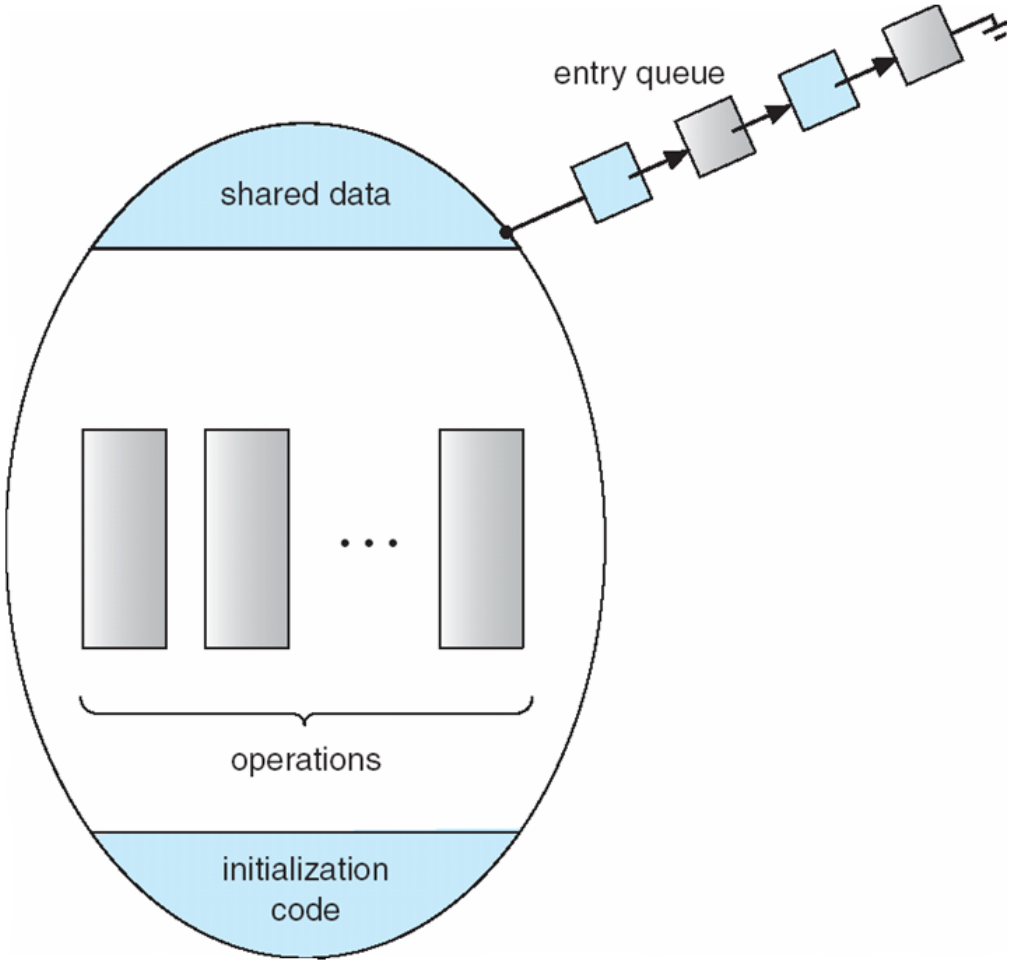
# Monitors

- **Theorem**. **Every synchronization scheme that is implementable with semaphores is also implementable with monitors (no conditions needed).**

**Proof**. Since semaphores are implementable with TestAndSet (first, one can implement *critical section* with TestAndSet, and then one can implement wait and signal with *critical section*), if suffices to prove that TestAndSet is implementable with monitor. The following code demonstrates that.

```
monitor test-and-set{
    function boolean TestAndSet (boolean *target)
        {
            boolean rv = *target;
            *target = TRUE;
            return rv:
        }
    }
```

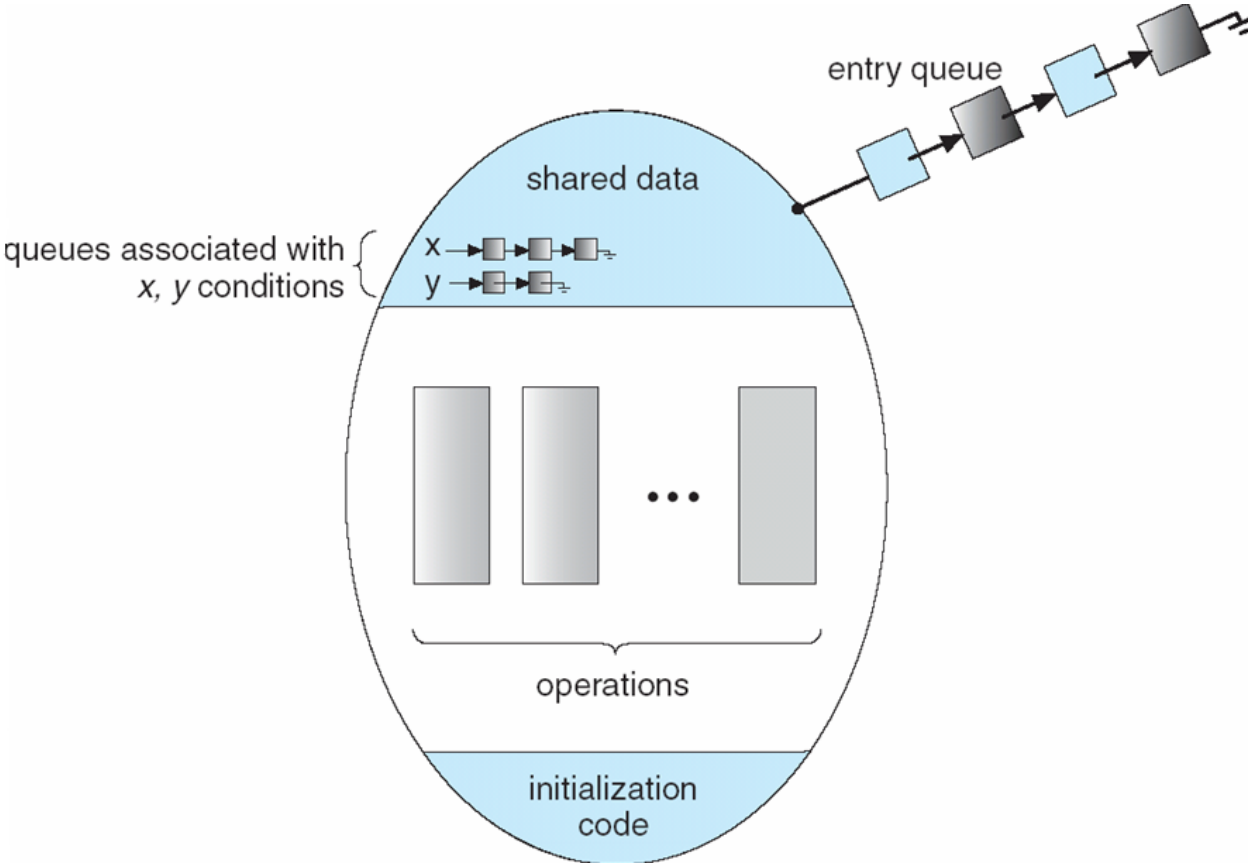
# Schematic view of a Monitor




# 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 ()
- Conditions are (degenerated) semaphores with constant int value = 0.

# 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)

- dp DiningPhilosophers = new dp();
- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophters.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

# Solution to Dining Philosophers (cont)

- **Exercise (hard)**: Modify the code of monitor dp so that it allows construction of individual philosophers.
- It requires execution of the following statement for each philosopher:
- dp DiningPhilosopherN = new dp();
- Then each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophterN.pickup ();

    EAT

DiningPhilosopherN.putdown ();
```

# 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--;
}
```

# 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;
    }
}
```

# Atomic Indivisible Transactions

## (Optional material)

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Indivisible 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 integrity  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 integrity 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 (that would be a must if the transactions were truly atomic)
  - 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 indivisibly
- Execution sequence called schedule
- Atomically Actually 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 $TS(T_i) < W\text{-timestamp}(Q)$, 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 $\max(R\text{-timestamp}(Q), TS(T_i))$

# 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($A$) |

# Chap 5 Sec 11 will follow