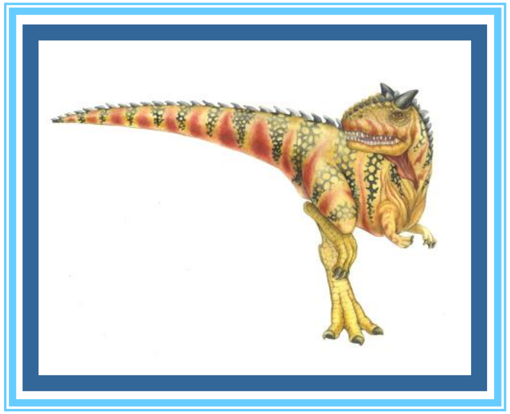
Process Coordination



Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space or be allowed to share data only through files or messages.

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 producer–consumer problem that fills **all** the buffers. We can do so by having an integer **count** that keeps track of the number of full buffers.
- Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

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

Consumer

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

Race Condition

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with “count = 5” initially:

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

Critical Section Problem

- Consider a system consisting of n processes { p_0, p_1,, ….p_{n-1}}.
- In the lifetime of a process, if a process want to access any kind of common sharable variables, common sharable file, updating a file etc, then we should say that process is entering into the critical section.
- The important feature of the system is that, when one process is executing in the CS, no other process is to be allowed to execute in its CS.
- Means that no two processes are executing in its CS at the same time.
- The CS problem is to design a protocol that the processes can use to cooperate.
- Each process must request permission to enter its CS.
- The section of code implementing this request is ***entry section***
- The critical section may be followed by and ***exit section.***
- The remaining code is ***remainder section.***

- The general structure of a typical process p_i is
- Do {

    entry section

    critical section

    exit section

    remainder section

  } while (TRUE);

- In the entry section, the process request the OS to enter into the CS.
- If the OS allows to enter into the CS, then it will come into the CS.
- It will work with the common sharable variables.
- Then before leaving, it will intimate the OS that it is leaving the CS. This is known as exit section.
- After entering into the CS, the process will not terminate.
- It may have to enter into the same or other CS in its lifetime and that is the remainder section.
- Thus in the remainder section, the proces will again try to enter into CS.

Solution to Critical-Section Problem

A solution to the CS problem must satisfy three requirements

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the **N** processes

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int **turn**;
  - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process P_i is ready!

Algorithm for Process P_i

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

J=1-i

P_0

```
flag[0]=true;
turn=1;
while(flag[1]==true && turn==1);
flag[0]=false;
```

P1

```
flag[1]=true;
turn=0;
while(flag[0]==true && turn==0);
flag[1]=false;
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    ```
    wait (S) {
            while S <= 0
                ; // no-op
              S--;
     }
    ```
  - signal (S) {
    ```
    signal (S) {
        S++;
    }
    ```

Semaphore as General Synchronization Tool

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

- P1{ statement1

  signal(mutex)

- P2{wait(mutex)

  statement2

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 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

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);
        }
}
```

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
- 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. RR is possible
- RW not possible
- Only one single writer can access the shared data at the same time.
- WR NOT POSSIBLE
- WW is not possible
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1 (controls access to readcount)
  - Semaphore wrt initialized to 1 (writer access)
  - Integer readcount initialized to 0 (how many processes are reading object)

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

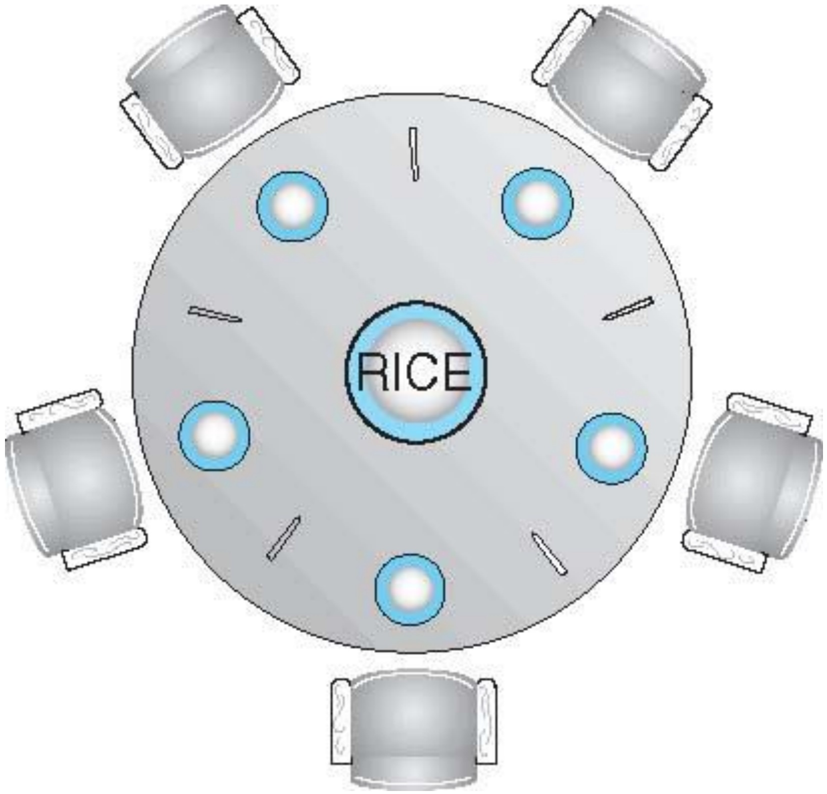
WW is not possible

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ; 1 0 1 0 1
        readcount ++ ; 0 1 2
        if (readcount == 1)
                wait (wrt) ;1 0
        signal (mutex)
            // reading is performed R1 R2
        wait (mutex) ; 0
        readcount  - - ; 1
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Dining-Philosophers Problem



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

- Problem definition
- There are N philosophers, sitting around a round dining table.
- There are N plates placed on the table such that each plate is infront of a philosopher.
- There are N forks placed between the plates.
- There is a bowl of Noodles placed at the centre of the table.
- Whenever the philosopher feels hungry, he tries to pick two forks/chopsticks which are shared with his nearest neighbors.
- If any of the neighbors happens to be eating at the time, the philosopher has to wait.
- Whenever a hungry philosopher gets two chopsticks, he pours Noodles in his plate and eats.
- After he finishes, he places the chopsticks back on to the table and starts thinking.
- Now these chopsticks are available for his neighbors.

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);
```

What is the problem with the above?

More Problems with Semaphores

- Relies too much on programmers not making mistakes (accidental or deliberate)
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

- Monitor is a programming language construct that supports controlled access to shared data.
- A monitor type presents a set of programmer defined operations that are provided mutual exclusion within the monitor.
- A monitor is a software module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data
  - Synchronization between concurrent processes that invoke those procedures

  Monitor protects the data from unstructured access.

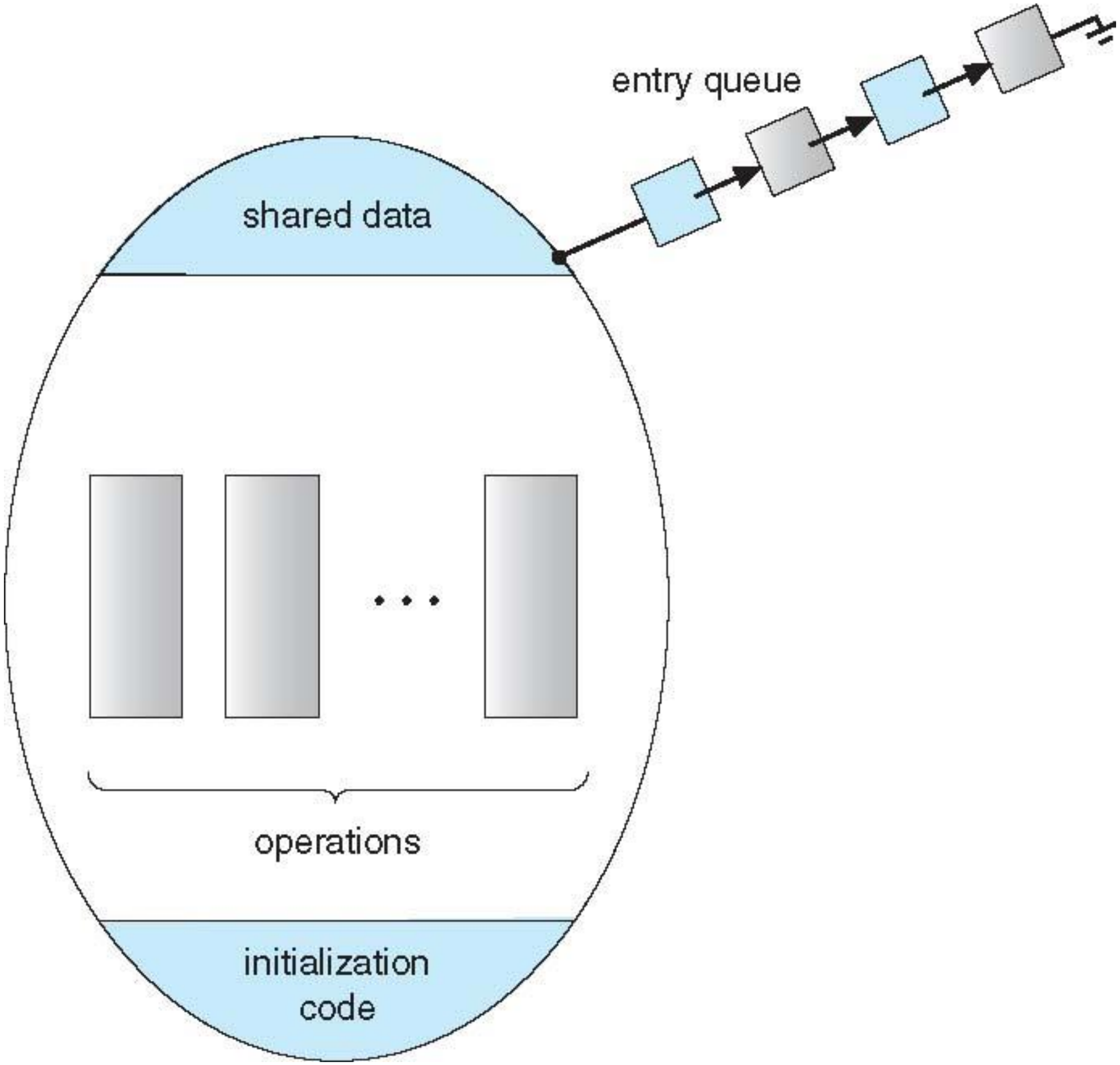
Monitors

- The syntax of a monitor is shown below
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…)
    { ….
    }
  procedure P2 (…)
    { ….
    }
          …
  procedure Pn (…) {……}
   Initialization code ( ….) { … }
          …
  }
}
```

- The representation of a monitor type cannot be used directly by the various processes.
- Thus a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local procedures.

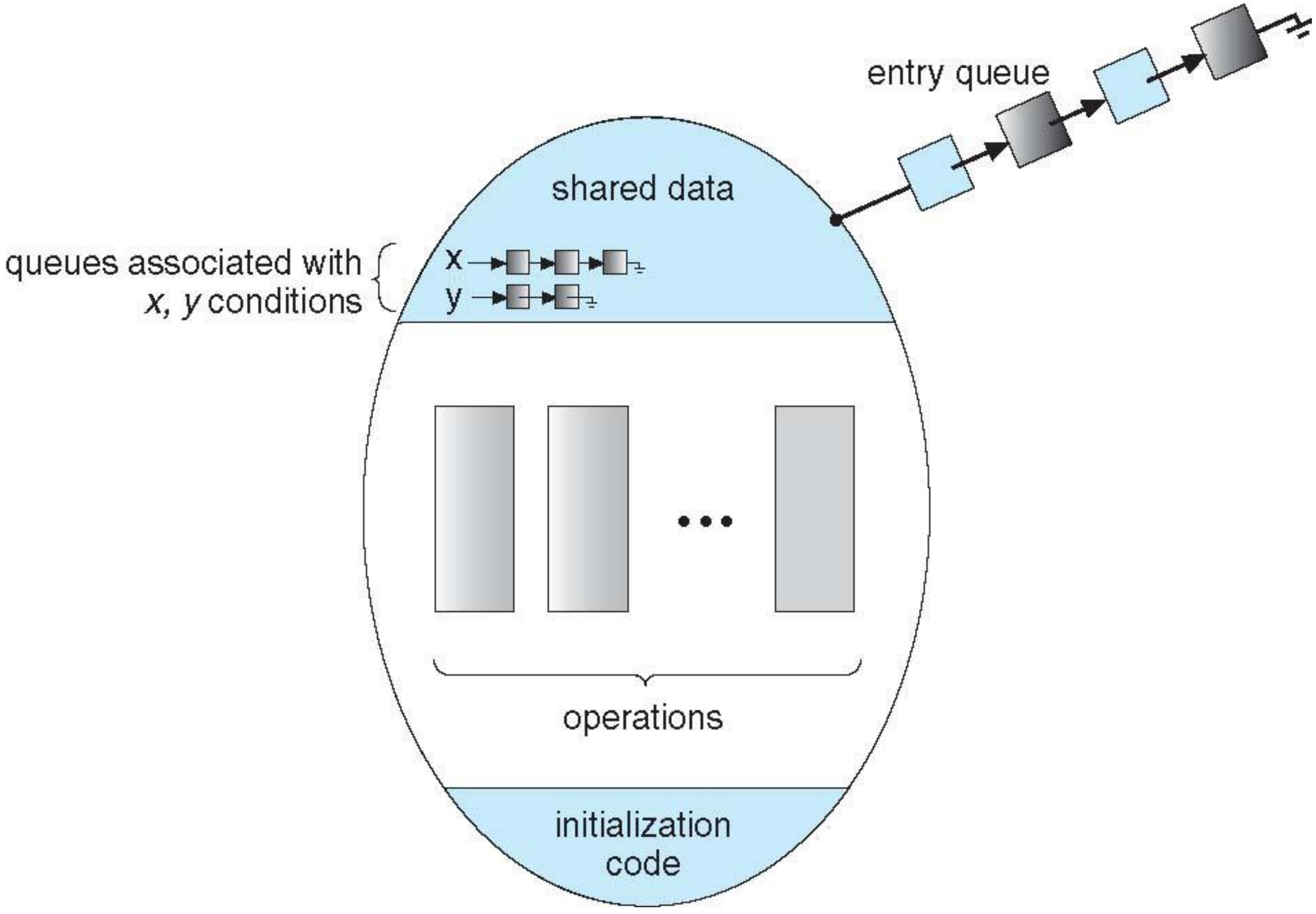
Schematic view of a Monitor



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



- The x.signal() operation resumes exactly one suspended process.
- If no process is suspended, then the signal() operation has no effect, ie, the state of x is same as if the operation had never been executed.
- Suppose that when x.signal() operation is invoked by a process P, there is a suspended process Q associated with the condition x.
- Clearly if the suspended process Q is allowed to resume its execution, the signal process P must wait.
- Otherwise both P and Q would be active simultaneously within the monitor.
- Two possibilities exist:
  - Signal and wait – P either waits until Q leaves the monitor or waits for another condition.
  - Signal and continue – Q either waits until P leaves the monitor or waits for another condition

Solution to Dining Philosophers

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

    void pickup (int i) {
           state[i] = HUNGRY;
           test(i);
           if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
           state[i] = THINKING;
                  // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
      }
```

Solution to Dining Philosophers (Cont.)

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

     initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
    }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher *I* invokes the operations pickup()

  and putdown() in the following sequence:

  DiningPhilosophters.pickup (i);

  EAT

  DiningPhilosophers.putdown (i);

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
     …
          body of F;

     …
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

Monitor Implementation

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```