Process Synchronization

Dr. SUBHASIS DASH

SCHOLE OF COMPUTER ENGINEERING.

KIIT UNIVERSITY

BHUBANESWAR

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE    n
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE is [n-1] elements.
- **(((in = (in + 1)** % **BUFFER SIZE count) == out) :- buffer full**
- **(in == out) :- buffer empty**

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
 }
```

Producer & Consumer Code

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

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

put side by side

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 (CS) PROBLEM

- Critical is a segment of code in each process where process wants to change there common variable.
- But CS allows only one process to execute in its CS –: **MUTUALLY EXCLUSIVE**.
- There are 3 different sections:-
  - ENTRY SECTION
  - EXIT SECTION
  - REMINDER SECTION

Example:-

```
repeat
        ENTRY
                critical section
        EXIT
                reminder section
until false
```

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 only those process that are not executing their reminder section can participate in the selection of the processes that which will enter the critical section next, and this selection cann't 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

1^{st} S/W Solution for CS problem

- P_i's Algorithm

repeat

var turn = i

While (turn != " i ") do no-op;

Critical section

turn=" j ";

until false

- P_j's Algorithm

repeat

var turn = j

While (turn != " j ") do no-op;

Critical section

turn=" i ";

until false

2^{nd} Solution

- P_I's Algorithm

Var flag : array [i…j] of boolen

repeat

flag [i] = true;

While (flag [j] == “ true ”) do no-op;

Critical section

Flag [i]=“ false ”;

until false

- P_J's Algorithm

Var flag : array [i…j] of boolen

repeat

flag [j] = true;

While (flag [i] == “ true ”) do no-op;

Critical section

Flag [j]=“ false ”;

until false

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!

Peterson's Solution

- P_i's Algorithm

```
repeat
   S1: flag [ i ] = TRUE;
   S2: turn = j;
   S3: while ( flag[ j ] == TRUE &&
                 turn != i ) do no-op;

       CRITICAL SECTION

    flag [ i ] = FALSE;
until false
```

- P_j's Algorithm

```
repeat
   L1: flag [ j ] = TRUE;
   L2: turn = i;
   L3: while ( flag [ i ] == TRUE &&
                 turn != j) do no-op;

       CRITICAL SECTION

    flag [ j ] = FALSE;
until false
```

Sequence of execution:

S1; S2; L1;L2;   S1;L1;L2;S2;   L1;S1;L2;S2

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
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

→ A LOCK is a object that provides the following 2 operations:-

(A) acquire ( ) → wait to enter CS

(B) Release ( ) → Allow another to enter to CS.

acquire_lock ( )

Critical Section

Release_lock ( )

Reminder Section

Ex: int withdraw(account, amount)

```
{
  acquire (lock);
  balance = get_balance(account);
  balance = balance - amount;
  Put_balance (account, balance);
  release (lock);
} Return balance ;
```

Lock implementation

```
Struct lock
  {
    int held = 0;

  }
void acquire (lock)
  {
    while (lock->held);
    lock->held = 1;
  }
void release (lock)
  {
    lock->held = 0;
  }
```

→ if instⁿ preempted after while then mutual exclusion violated. // To prevent it $\boxed{\text{disable interrupt.}}$
↳ one solution.

→ Busy waits

→ context switch here.

→ Hence this sequence must be atomic. !

TestAndndSet Instruction

- Definition:

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

Solution using TestAndSet

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

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

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

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:

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

                    //    critical section

         lock = FALSE;

                    //      remainder section

}
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore (*S)* is an integer variable in which two atomic operations P & V is defined to provide synchronization & mutual exclusion in concurrent system.
- Two standard operations modify S: wait() and signal()
  - Originally called P(S) " TO TEST" and V(S) "TO INCRIMENT"
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - P(S) : wait (S)

```
{
    while S <= 0;   // no-op
    S--;
}
```

  - V(S) : Signal (S)

```
{
   S++;
}
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutexlocks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

      Critical Section

    signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes critical due to critical section problem where the wait and signal code both are placed in the critical section.
  - Could now have **busy waiting** in critical section implementation *[which demands mutual exclusion]*
    - *Spin Lock*
    - 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 ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
      value--;
      if (value < 0) {
            add this process to waiting queue
            block();  }
}
```

- Implementation of signal:

```
Signal (S){
      value++;
      if (value <= 0) {
            remove process P from waiting queue
            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.

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.)

- N buffers, each can hold one item
- Semaphore **mutex** = 1
- Semaphore **full** = 0
- Semaphore **empty** = N.

Structure of producer process

```
while (true)  {
      //   produce an item
        wait (empty);
        wait (mutex);
      //  add the item to
          the  buffer
         signal (mutex);
         signal (full);
}
```

Structure of consumer process

```
while (true) {
        wait (full);
        wait (mutex);
      //  remove an item
          from  buffer
        signal (mutex);
        signal (empty);
      //  consume the
          removed item
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

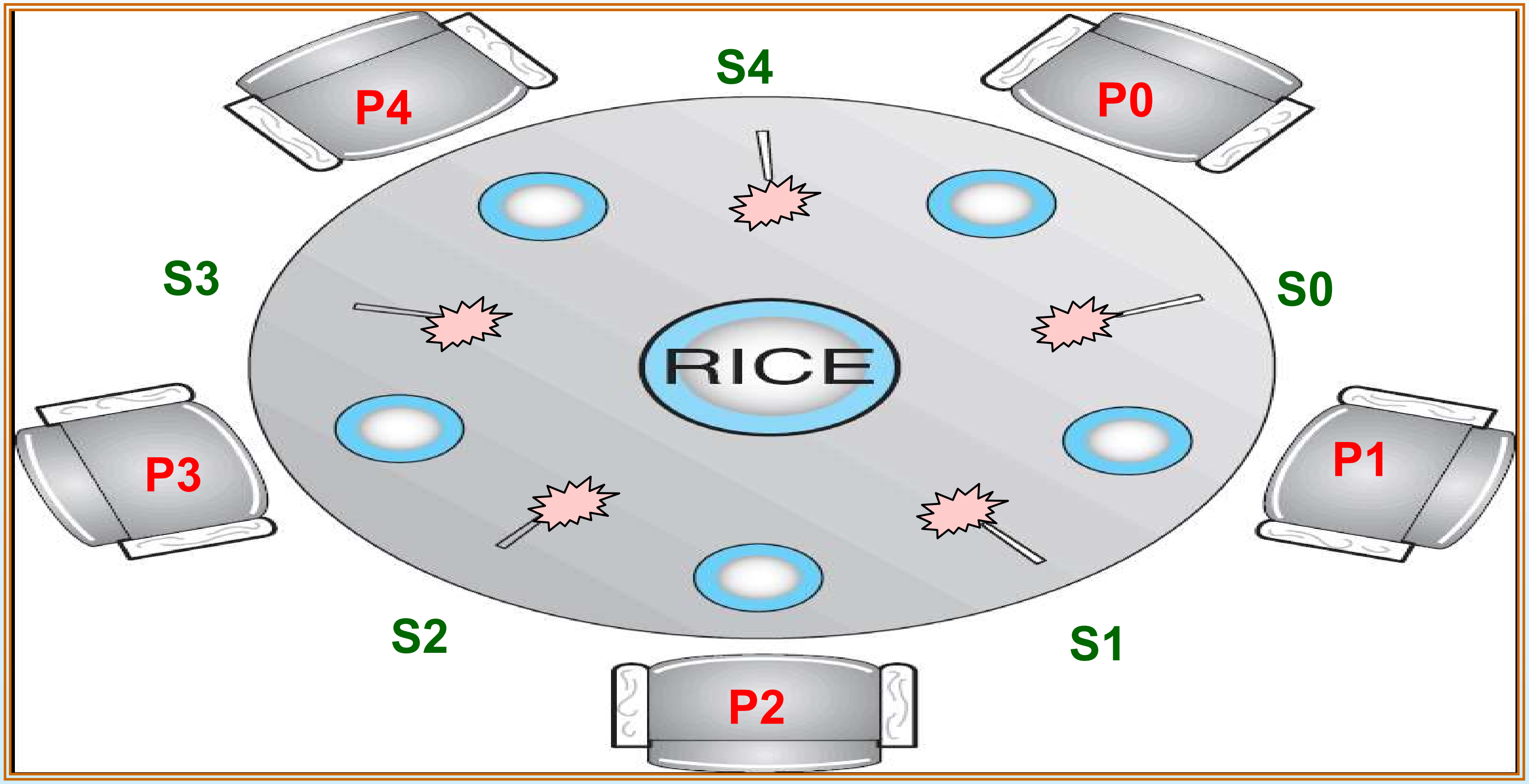
- Structure of writer process

```
 while (true)
{
  wait (wrt) ;

  //writing is
  performed

  signal (wrt) ;
 }
```

- Structure of reader process

```
while (true)
{
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  wait (wrt) ;
        signal (mutex)
             // reading is performed
         wait (mutex) ;
          readcount  - - ;
          if (readcount  == 0)  signal (wrt) ;
                signal (mutex)
    }
```

Dining-Philosophers Problem



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

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
While (true)
{
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
                //  eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
                //  think

}
```

Problems with Semaphores

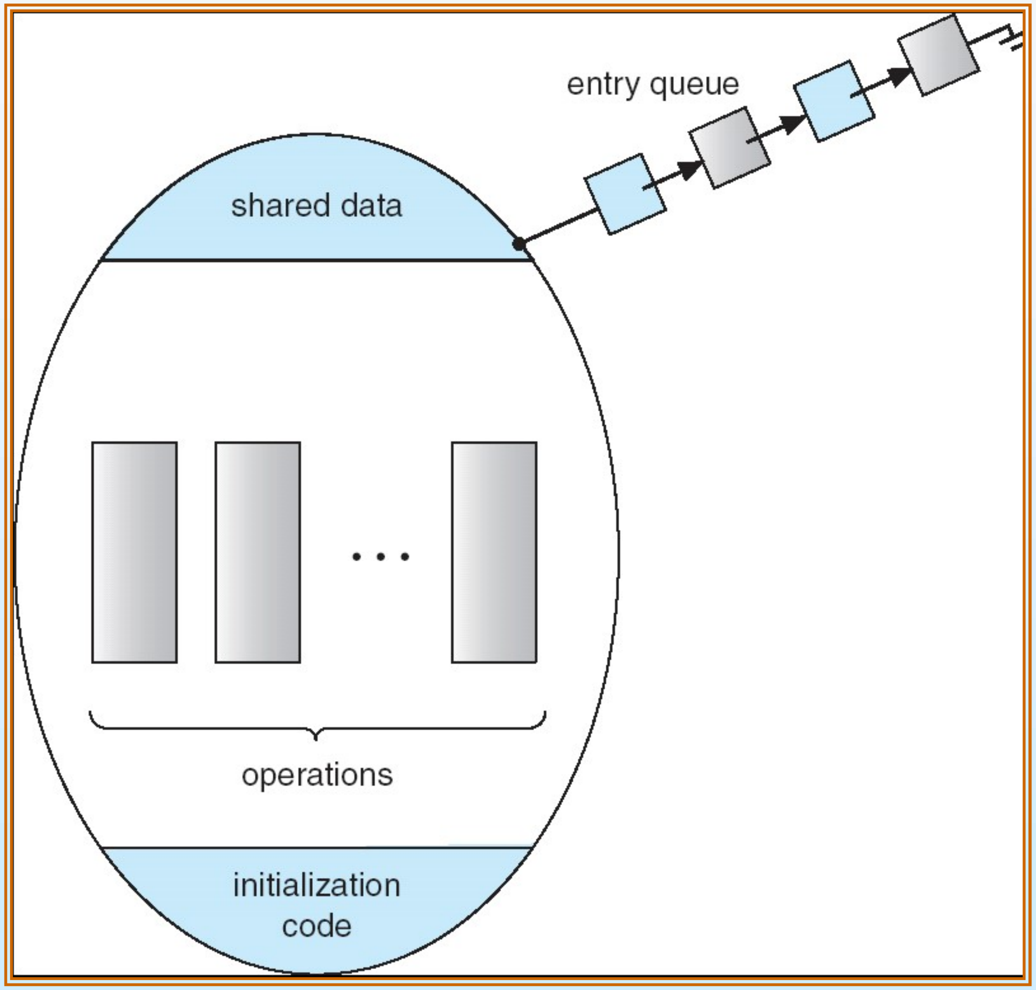
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

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

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

   procedure Pn (…) {……}

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

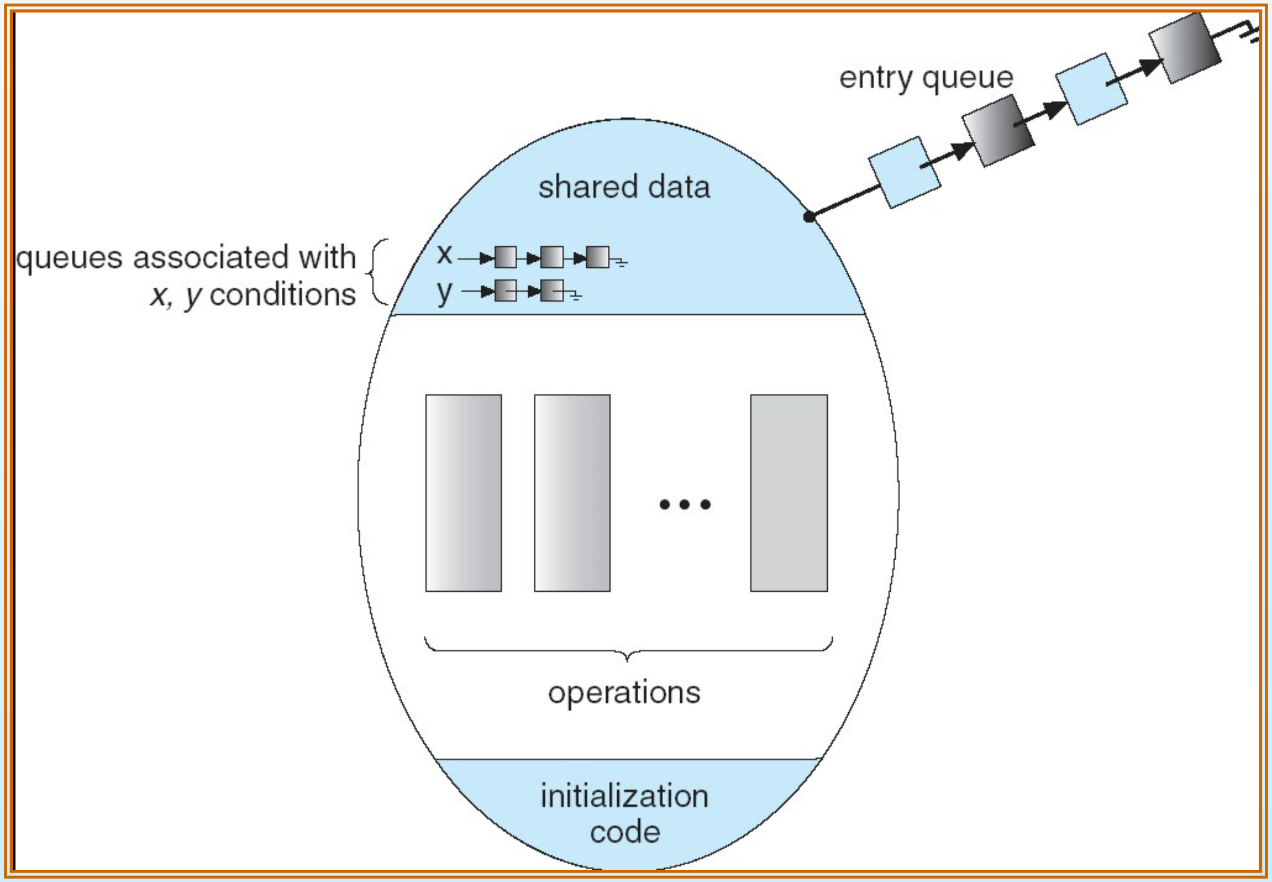
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it 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

Monitor 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);
}
```

Monitor Solution to Dining Philosophers

```
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;
     }
}
```

Monitor Solution to Dining Philosophers

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

  DiningPhilosophers.pickup(i);

  EAT

  DiningPhilosophers.putdown(i);

- No deadlock, but starvation is possible

Bounded buffer Solution using monitor

- Bounded buffer using monitors and signals
  - Shared State data[10] - a buffer holding produced data.
    num - tells how many produced data items there are in the buffer.
  - Atomic Operations Produce(v) called when producer produces data item v.
    Consume(v) called when consumer is ready to consume a data item. Consumed item put into v.
  - Condition Variables bufferAvail - signalled when a buffer becomes available.
    dataAvail - signalled when data becomes available.

```
monitor PC {
    Condition *bufferAvail, *dataAvail;
    int num = 0;
    int data[10];

    Produce(v) {
        while (num == 10) {
            bufferAvail→Wait();
        }
        put v into data array
        num++;
        dataAvail→Signal();
    }
    Consume(v) {
        while (num == 0) {
            dataAvail→Wait();
        }
        put next data array value into v
        num–;
        bufferAvail→Signal();
    }
```

End of Chapter 4