UNIT- 2

# Chapter 1

# Process synchronization

# Syllabus

**Process synchronization and deadlocks:** The critical section problem, Synchronization hardware, Semaphores, Classical problems of synchronization, Critical regions, monitors, Dead locks –System model , characterization, Dead lock prevention, avoidance and detection, Recovery from dead lock.

# The Critical Section Problem

- A **critical section is a piece of code that** accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

- The goal is to provide a mechanism by which only one instance of a critical section is executing for a particular shared resource.

- Unfortunately, it is often very difficult to detect critical section bugs

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

- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

**A Critical Section Environment contains:**

– **Entry Section** Code requesting to entry into the critical section.

– **Critical Section** Code in which only one process can execute at any one time.

– **Exit Section** The end of the critical section, releasing or allowing others in.

– **Remainder Section** Rest of the code AFTER the critical section.

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

## Solution to Critical-Section Problem

1. **Mutual Exclusion -** If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress -** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting -** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

# Synchronization Hardware

- 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
        - » Have to wait for disable to propagate to all processors
        - » Operating systems using this not broadly scalable
    - Modern machines provide special atomic hardware instructions
        - » Atomic = non-interruptable

# Solution to Critical-section Problem Using Locks

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

# test_and_set Instruction

- Definition:

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

## Solution using test_and_set()

- Shared boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

# Semaphores

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

# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Known as **mutex locks**
- Can implement a counting semaphore ***S*** as a binary semaphore
- Can solve various synchronization problems
- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

# 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 ready queue

# Semaphore Implementation with no Busy waiting

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

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

# Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
- When such a state is reached, these processes are said to be deadlocked
- Consider a system consisting of two processes,P0 and P1,each accessing two semaphores, S and Q ,set to the value 1:

# Deadlocks and Starvation

| $P_0$ | $P_1$ |
| --- | --- |
| `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

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

# Bounded Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n

- The structure of the producer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
} 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
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

# Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
   wait(rw_mutex);
      ...
   /* writing is performed */
      ...
   signal(rw_mutex);
} while (true);
```

# Readers-Writers Problem (Cont.)

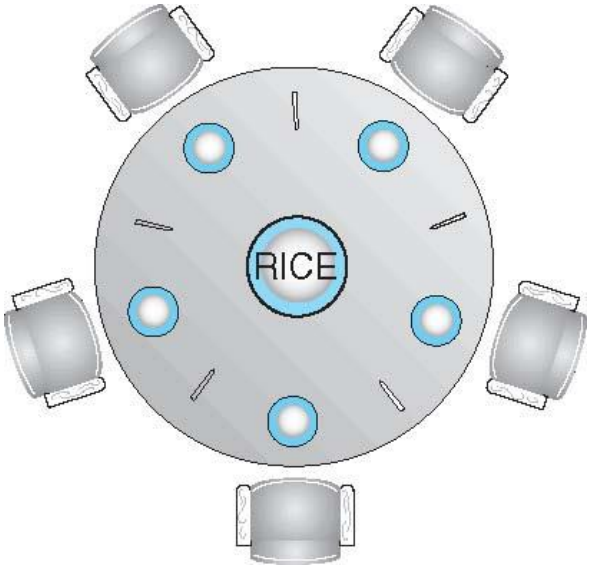
- The structure of a reader process

```
do {
   wait(mutex);
   read count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
   ...
   /* reading is performed */
   ...
   wait(mutex);
   read count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

# Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

# Dining-Philosophers Problem




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

# Dining-Philosophers Problem Algorithm

- 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);
```

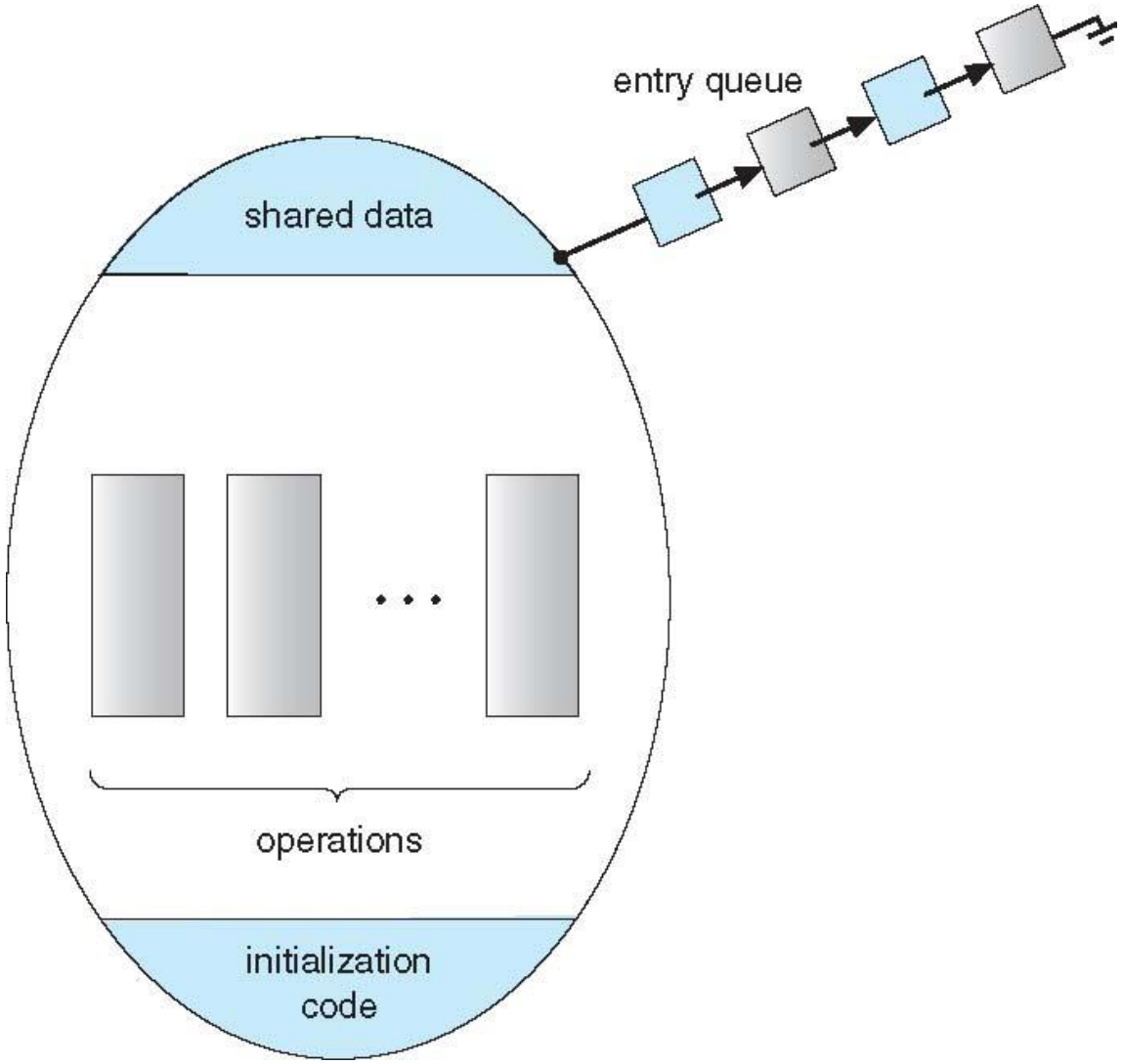
# Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only ***one*** process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  function P1(...) { ... }
  ...
  function Pn(...) { ... }

  initialization_code (...) { ... }
}
```

# Schematic view of a Monitor

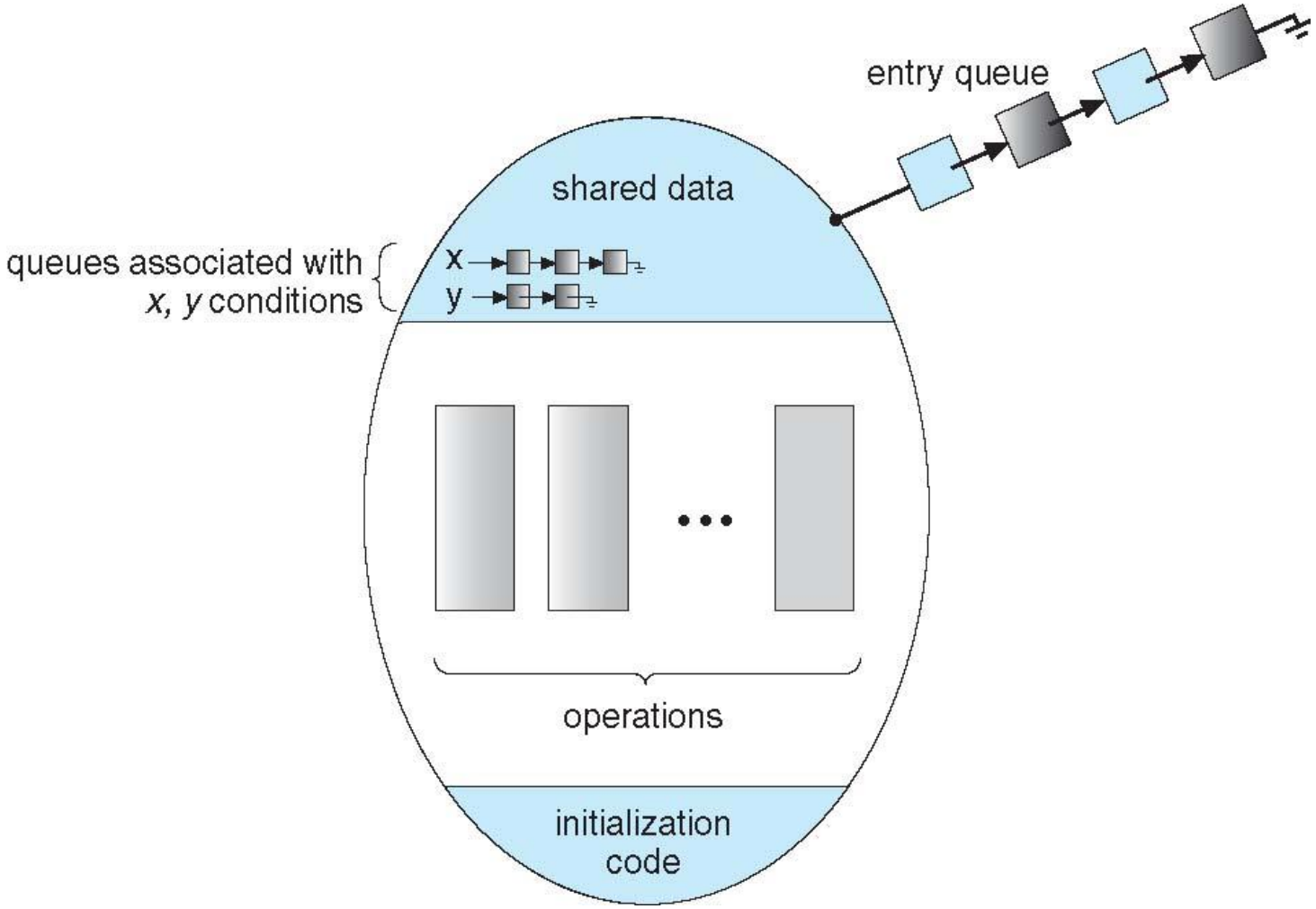


# Condition Variables

- **`condition x, y;`**

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

# Monitor with Condition Variables



# Condition Variables Choices

- If process P invokes **`x.signal()`**, with Q in **`x.wait()`** state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

# 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);
         }
```

# 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:

```
DiningPhilosophers.pickup(i);

        EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

# Monitor Implementation Using Semaphores

- Variables

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

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

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

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

- Mutual exclusion within a monitor is ensured

# Monitor Implementation – Condition Variables

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

```
semaphore x_sem; // (initially = 0)
        int x_count = 0;
```

- The operation x.wait can be implemented as:

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