Chapter : Process Synchronization

Synchronization Hardware

Hardware Solution to C.S.

- Many systems provide hardware support for critical section code

- Uni-processors – could disable interrupts
  - Currently running code would execute without preemption

- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Hardware Solution to C.S.

1. Interrupt Disabling

1. Process leaves control of CPU when it is interrupted.
2. Solution is:
    1. To have each process disable all interrupts just after entering to the critical section.
    2. Re-enable interrupts after leaving critical section

Hardware Solution to C.S.

- **Interrupt Disabling**

Repeat

Disable interrupts

C.S

Enable interrupts

Remainder section

Hardware Solution to C.S.

2. Hardware instructions

1. Machines provide instructions that can read, modify and store memory word
2. Common inst. are:
    1. **Test and Set**

This inst. Provides action of testing a variable and set its value

It executes atomically

Test and Set instructions operates on single Boolean variable.

Hardware Solution to C.S.

//initially lock is FALSE

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

        /* critical section */

    lock = false;

        /* remainder section */
} while (true);
```

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

Swap Instruction

- Definition:

```
void Swap (boolean a, boolean b)
 {
        boolean temp = a;
        a = b;
        b = temp:
 }
```

Hardware Solution to C.S. Using Swap

```
Shared data : lock initialized to false
Boolean:lock;

do{
flag=true;
while(flag)
swap(lock,flag);
C.S.
lock=false;
Remainder Section
} While(True);
```

```
swap(lock,flag)
{
    temp = lock;
    lock=flag;
    flag=temp;

    }
```

Semaphore

- Synchronization tool that maintains concurrency using variables
- Semaphore S is a integer variable which can take positive values including 0. It is accessed from 2 operations only.

Operations On Semaphore:

- wait() and signal()

1. Wait Operation is also known as P() which means to test
2. Signal Operation is also known as V() which means to increment

- Entry to C.S. is controlled by wait()
- Exit from C.S. is signaled by signal()

Semaphore

- Can only be accessed via two indivisible (atomic) operations and S=1
- <u>For critical Section problem semaphore value is always 1</u>
- P(S) and V(S):

```
wait (S) {
        while (S <= 0)
         do skip ; // no-op}
           S- -;
C.S
signal (S) {
      S++;      }
```

Semaphore as General Synchronization Tool

Types of Semaphores:

- Counting semaphore – when integer value can be any non-negative value
- Binary semaphore – integer value can range only between 0 and 1
  - Also known as mutex locks

Semaphore and Busy Waiting

Disadvantage of Semaphore: Busy Waiting

- When a process is in C.S. and any other process that wants to enter C.S. loops continuously in entry section.

- Wastes CPU cycles

- Semaphore that implements busy waiting is known as: **Spin Lock**

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

Semaphore Implementation with no Busy waiting

Instead of waiting, a process blocks itself.

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

Semaphore Implementation with no Busy waiting

- Implementation of wait:

S=3

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

Semaphore Implementation with no Busy waiting

- Implementation of signal:

```
  Signal (S){
  value++;        //"value" has -ve value i.e -1
here
  if (value <= 0) {
remove a process P from the 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

- 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.

- mutex= 1
- full= 0
- empty= N

Bounded Buffer Problem (Cont.)

- The structure of the producer process

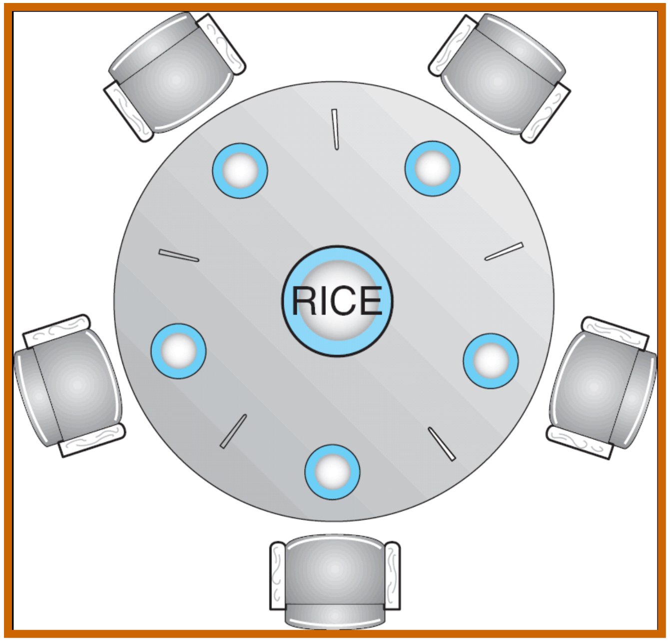
```
while (true)  {

        //   produce an item

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
}
```

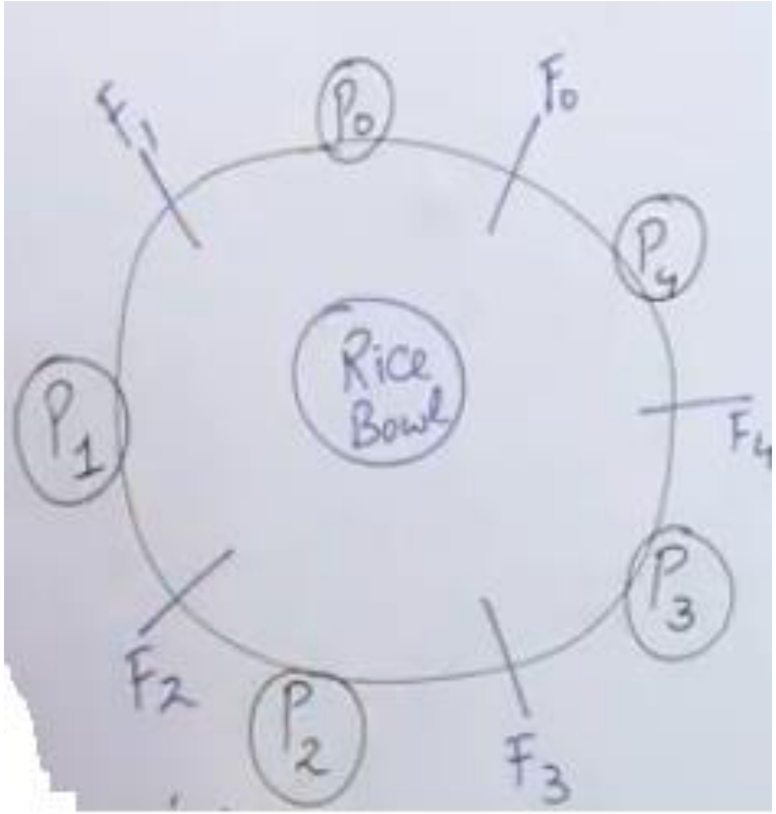
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
        wait (full);
        wait (mutex);  // mutex=1

              //  remove an item from  buffer

        signal (mutex);     // mutex=0
        signal (empty);

              //  consume the removed item
}
```

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] ); // =1

             //  eat

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

             //  think

}
```

Problems with Semaphores

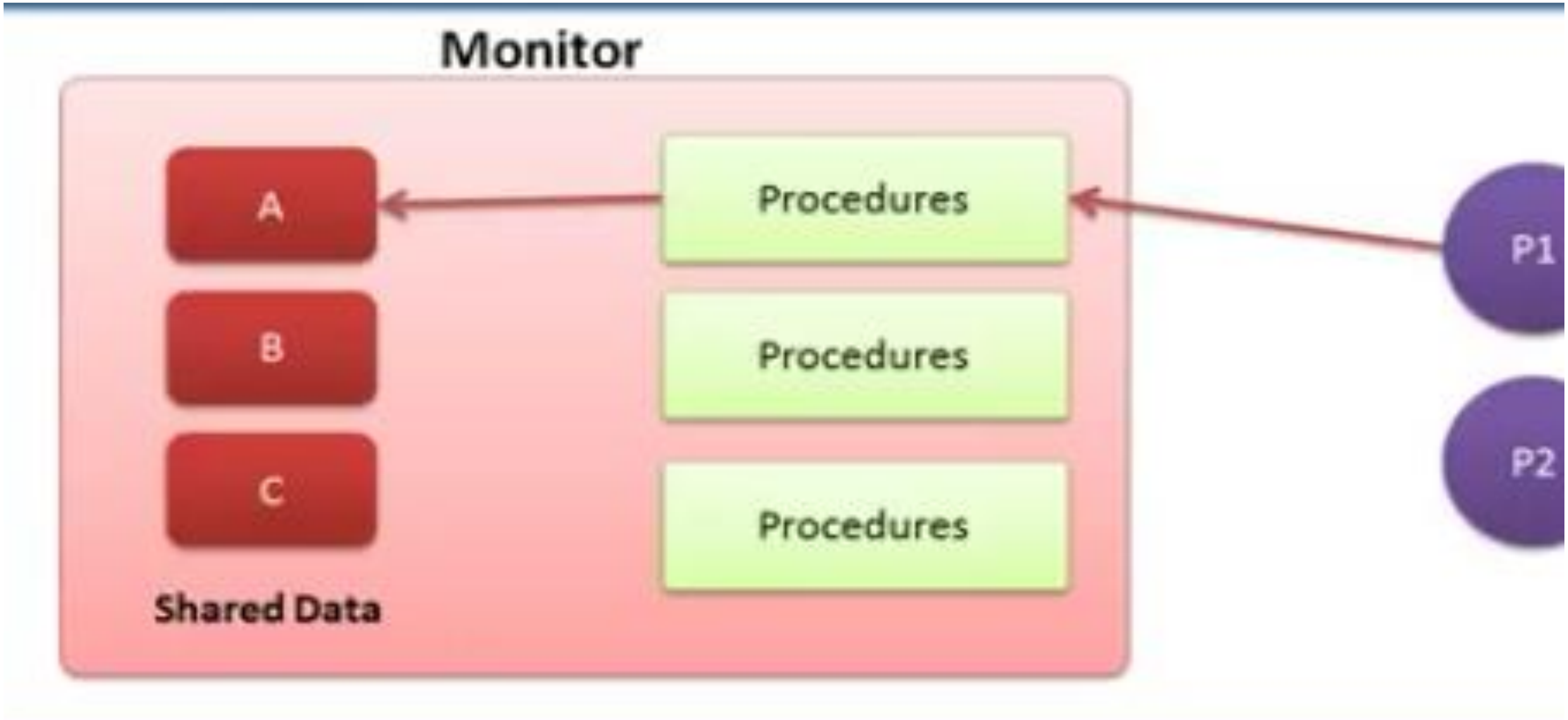
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A way to encapsulate the Critical Section by making class around the critical section and allowing only one process to be active in that class at one time.
- Only one process may be active within the monitor at a time
  - Name of Monitor
  - Initialization Code Section
  - Procedure to request the Critical Data
  - Procedure to release the Critical Data

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

Monitors

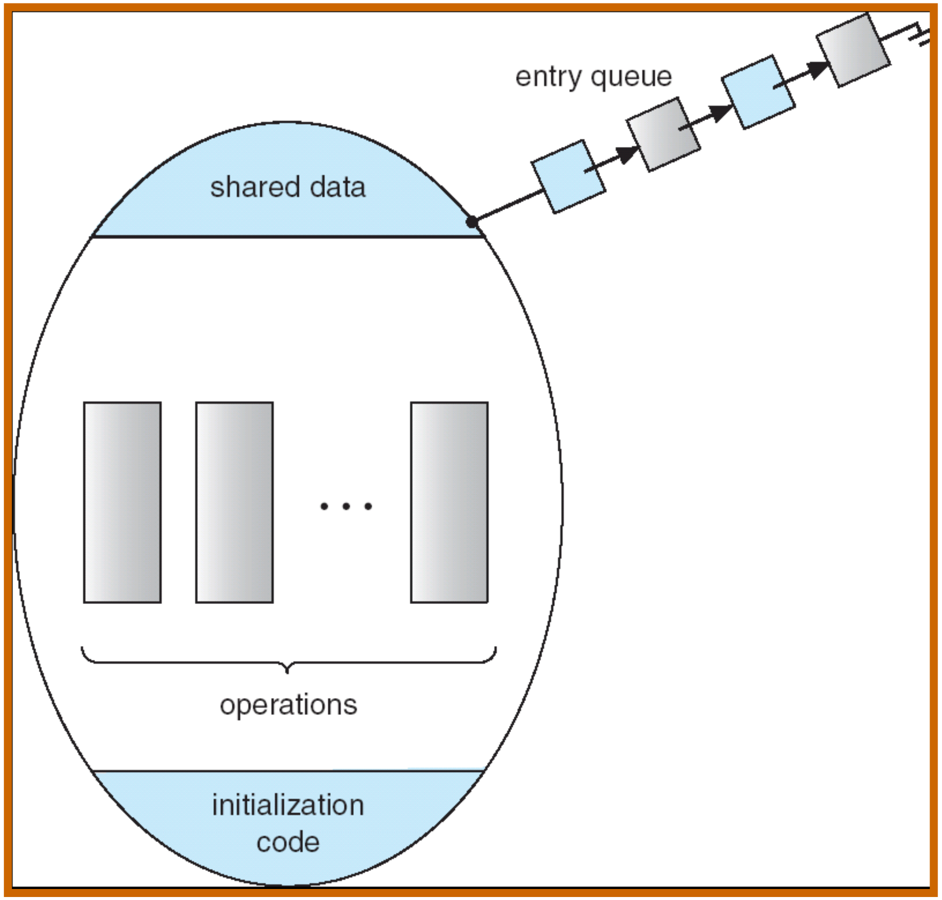


A monitor is a module that encapsulates

- Shared data structures
- Procedures that operates on the shared data
- Synchronization between concurrent procedure invocation

Schematic view of a Monitor

Only one process at a time can be in monitor



Condition Variables

Conditional Variable

- Conditional Variable provides synchronization inside the monitor.
- If a process wants to sleep inside the monitor or it allows a waiting process to continue, in that case conditional variables are used in monitors.

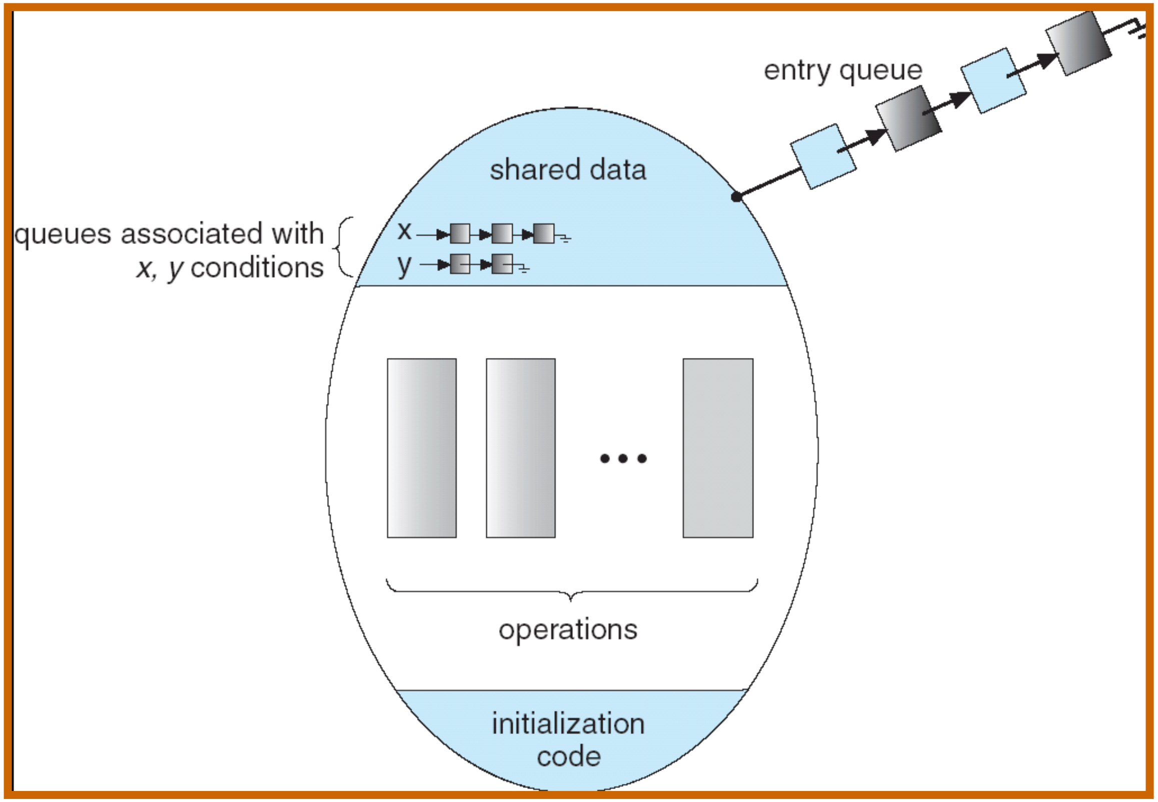
Condition Variables

- 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 ()

There could be different conditions for which a process could be waiting

Let a process made the request to C.S. but no critical data is available, So we would not go back to check all the conditions again to run the request, As we have checked some conditions so processes will be lined up for excess to C.S.

Monitor with Condition Variables



MCQ Questions

Which process can be affected by other processes executing in the system?

a) cooperating process

b) child process

c) parent process

d) init process

MCQ Questions

1 a

MCQ Questions

When several processes access the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called

a) dynamic condition

b) race condition

c) essential condition

d) critical condition

MCQ Questions

2 b

MCQ Questions

If a process is executing in its critical section, then no other processes can be executing in their critical section. This condition is called

a) mutual exclusion

b) critical exclusion

c) synchronous exclusion

d) asynchronous exclusion

MCQ Questions

3 a

MCQ Questions

Which one of the following is a synchronization tool?

a) thread

b) pipe

c) semaphore

d) socket

MCQ Questions

4 c

MCQ Questions

A semaphore is a shared integer variable

a) that can not drop below zero

b) that can not be more than zero

c) that can not drop below one

d) that can not be more than one

MCQ Questions

5 a

MCQ Questions

Mutual exclusion can be provided by the

a) mutex locks

b) binary semaphores

c) both mutex locks and binary semaphores

d) none of the mentioned

MCQ Questions

6 c

Explanation: Binary Semaphores are known as mutex locks.

MCQ Questions

When high priority task is indirectly preempted by medium priority task effectively inverting the relative priority of the two tasks, the scenario is called

a) priority inversion

b) priority removal

c) priority exchange

d) priority modification

MCQ Questions

7 a

MCQ Questions

8. Process synchronization can be done on

a) hardware level

b) software level

c) both hardware and software level

d) none of the mentioned

MCQ Questions

8 c

MCQ Questions

9. A monitor is a module that encapsulates

a) shared data structures

b) procedures that operate on shared data structure

c) synchronization between concurrent procedure invocation

d) all of the mentioned

MCQ Questions

9 d

MCQ Questions

To enable a process to wait within the monitor,

a) a condition variable must be declared as condition

b) condition variables must be used as boolean objects

c) semaphore must be used

d) all of the mentioned

MCQ Questions

10 a

End of Chapter