UNIT 3

PROCESS COORDINATION

Synchronization

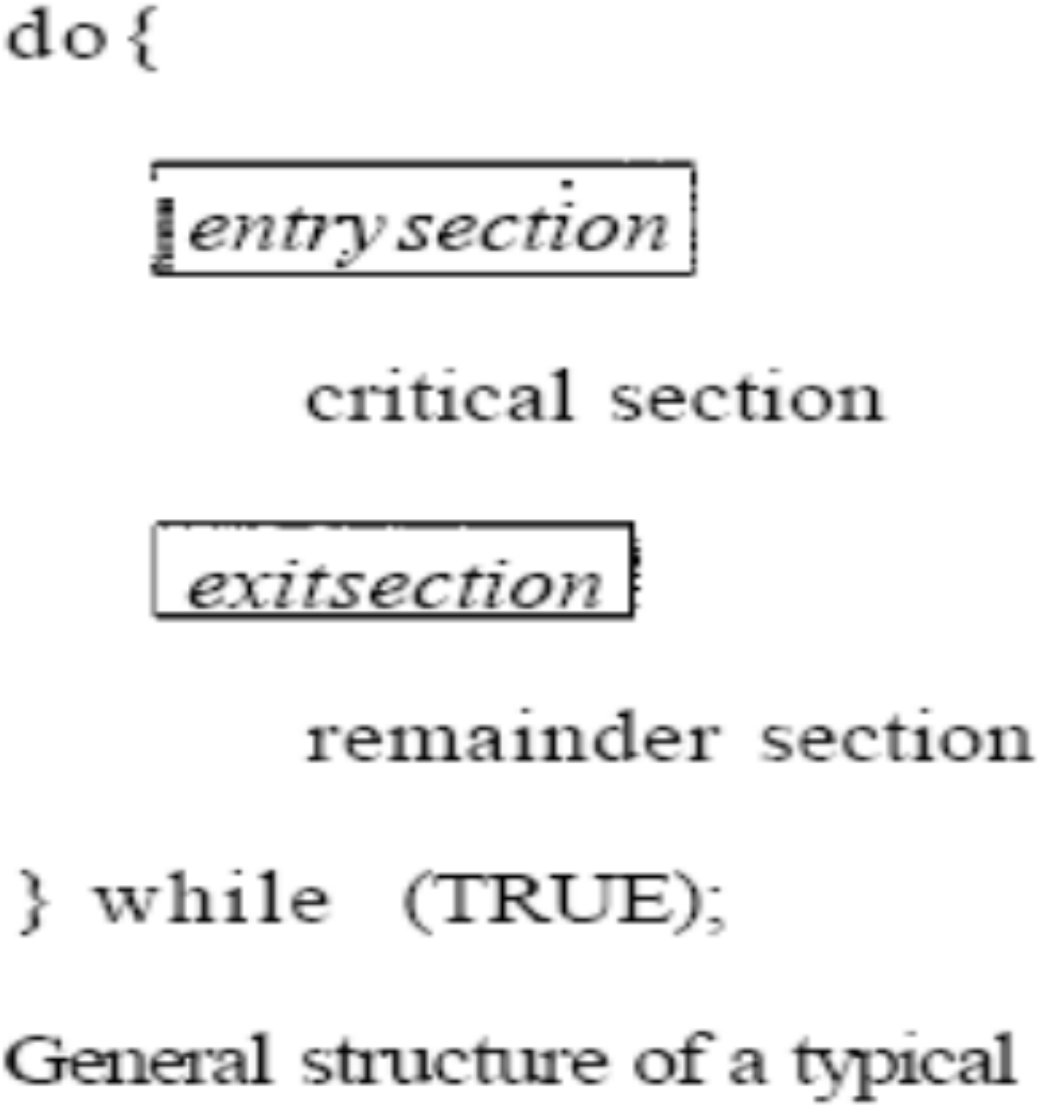
- Cooperating process is one that can affect or be affected by other processes and they share data.
- Its main drawback is that unordered execution causes data inconsistency
- P1
  - Read bal
  - bal = bal + 1000
- P2
  - Read bal
  - Bal = bal – 500
- Assume bal = 5000, execute P1 after p2 then bal = 5500

if the order is p1 : Read bal (5000), P2:Read bal(5000), P1: bal = bal + 1000 (6000), P2: bal=bal-500(4500) . So when the order of execution changes the final value of bal changes.

- The situation in which several processes access and manipulate same data(share data) concurrently and the result depends on the order in which the shared data is accessed is known as **race condition**.
- To avoid race conditions some form of synchronization among the processes is required which ensures that only one process is manipulating the shared data at a time

Critical-Section Problem

- Critical section is a piece of code that accesses a shared resource.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section.
- CS problem is to design a protocol that a process can use to cooperate. It contains
  - Entry section – portion of code to request permission to enter its critical section
  - Exit Section – portion of the code to exit after executing the critical section
  - Remainder section – remaining code



l General structure of a typical process P_i.

- Synchronization mechanism **is required at the entry and exit of the critical section to ensure exclusive use.**
- Solution to Critical Section problem must satisfy
  - **Mutual exclusion** - When one process is executing in its critical section, no other process is allowed to execute in its critical section.
  - **Progress** - if no process is executing in its critical section, then the processes which are not executing in their remainder section can enter the Critical Section & it cannot be postponed for long time
  - **Bounded Waiting** – there is a limit on the no of times a process can enter their critical section, after a process has made a request to enter its critical section

- **General approaches to handle critical sections in OS are:**
  - **Preemptive Kernels** – allows a process to be preempted while a process is running in kernel mode. They will have race conditions and should be careful in designing.
  - **Non-preemptive Kernels** - does not allow a process running in kernel mode to be preempted. They are free from race conditions

Critical Section Problem Solution – Peterson's Algorithm

- Peterson's Algorithm is a two process solution
- Consider 2 processes p_i and p_j sharing 2 variables

```
int   turn;
boolean flag[2];
```

  - turn – indicates whose turn is next to enter its critical section ; turn=i means p_i is allowed to enter CS
  - flag array – indicates whether a process is ready to enter CS

    flag[i] = true means p_i is ready to enter CS
  - For p_i to enter, set flag[i] to true and turn = j

    in other words Flag[j] =false or turn=i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
                    doNothing();

      critical section

    flag[i] = false;

        remainder section
} while (true);
```

Entry Section

Exit Section

- Provable that
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

Any solution to the critical-section problem requires a simple tool—a lock.

Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

do {

acquire lock

critical section

release lock

remainder section

} while (TRUE);

Handling CS in Uniprocessors

- prevent interrupts from occurring while a shared variable was being modified.
- Sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is the approach taken by nonpreemptive kernels.

In multiprocessor systems

- Disabling of interrupts not feasible in a multiprocessor environment.
- Time consuming as the message is passed to all the processors.
- This message passing delays entry into each critical section, and system efficiency decreases.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically.

The TestAndSet() instruction can be defined as follows

TestAndSet() instruction Definition

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

The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

Mutual exclusion implementation with TestAndSet()

If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.

The structure of process P_i is as follows

```
do
{
        while ( TestAndSetLock (&lock) )
              ;   // do nothing (spin until lock is
                  //acquired)
              // critical section
        lock = FALSE;          //release the lock
            // remainder section
}while (TRUE);
```

- The algorithm is easy and simple to understand. Any process that is wish to enter its critical section executes the Test And Set instruction and pass the values of lock as a parameter to it.
- If the value of lock is false, which break the while loop and allow the process to enter its the critical section.
- However if the value of lock is true, the Test And Set instruction returns true , thus blocking the process in the loop.
- The algorithm satisfies mutual exclusion but does not satisfy bounded waiting requirements.

- To meet all the requirements of the solution for critical section problem, another algorithm is developed that uses Test And Set instruction.

The algorithm lets the process to share the following two variables:

```
boolean  lock;
boolean waiting[n];
```

- The variable lock and all the elements of array waiting are initialized to false. Each process also has a local Boolean variable say key. The general structure for the code segment of process say p1 is as follows:

```
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] = = false)
    j = (j + 1) % n;

if (j == i)
   lock = FALSE;
else
waiting[j] = FALSE;

// remainder section
}while (TRUE);
```

- To prove that the mutual exclusion requirement is met, suppose a process Pi attempts to enter its critical section.
- It first sets the waiting[i] and key to true,and then reaches the while loop in the entry section.
- If pi is the first process attempting to enter its critical section, it finds both the conditions in the while loop are true, then it execute Test And Set instructions which sets the lock to true and return false, since lock is initially false.

- The returned value, that is false, is assigned to key.
- This allows the process Pi to exit from the loop and enter its critical section after resetting the waiting[i] to false.

- Now the value of lock is true, thus any other process, say Pj that attempts to enter its critical section when execute the Test And Set instruction sets the key to true and is blocked in the while loop until either key or waiting[j] become false.
- Note that neither key nor waiting[j] becomes false until Pi is in its critical section. This maintains mutual exclusion requirements .

Progress requirement

- Either sets lock to false or sets waiting[j] to false.
- Both allow a process that is waiting to enter its critical section to proceed.

Bounded-waiting requirement

- When a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, *i + 2,..., n* — 1, 0, ..., *i* — 1).
- It designates the first process in this ordering that is in the entry section (waiting [j] = true) as the next one to enter the critical section.
- Any process waiting to enter its critical section will thus do so within *n* — 1 turns.

Semaphore

Semaphore is a synchronization tool for the critical section problems

Semaphore S is an integer variable that

- Accessed through 2 operations:
- Wait() and Signal() operations.
- Wait() – to test (originally termed “P”)
- Signal() – to increment (originally termed “V”)

Semaphore Definition and Busy Wait Implementation

- **Definition for wait()**

```
wait(S)
 {
     while S <= 0 ;
         // do nothing
     S--;
}
```

- **Definition for signal()**

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

- The general structure for the code segment of the process say Pi is as follows

```
Do
{
    Wait(S);

        //Critical section

 Signal (S);

                ⟶ Exit Section
        // Remainder section
}
While(1)
```

Two Types of Semaphores

- The two most common kinds of semaphores are **counting semaphores** and **binary semaphores**.
- Counting semaphores are used to control access to a given resource consisting of a finite number of instances and its value can range over an unrestricted domain. semaphore is initialized to the no of resources available
- Binary semaphores, represents two possible states (generally 0(busy or locked) or 1(free or unlocked))

- Binary semaphore also known as mutex locks

```
Do
{
  waiting(mutex);
      //critical section
Signal (mutex)
       Remainder section
}while(true);
```

Spin lock(busy wait implementation)

- While a process is in its critical section any other process that tries to enter its critical section must loop continuously in the entry code and this semaphore is called **spin lock.**
- spin lock is useful in multiprocessor system(if loops for less time).
-The advantage of spin lock is that no context switch is required

Overcoming Busy Wait

- by modifying wait() & signal() operations

- When s <=0 , the process is blocked and put into a waiting queue of semaphore(waiting state)

Then cpu can be given to any other process

When process in CS executes signal() operation, the blocked process is restarted by a wakeup() operation(waiting to ready state)

If semaphore value is negative , its magnitude is the no of processes waiting in that semaphore.

- Block() – suspends the process that invokes it
- Wakeup() – resumes the execution of a blocked process

Note : by this we can only limit the Busy wait

- **Semaphore is implemented as a ‘C’ Struct as follows :**

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

Definition of wait()

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

Definition of signal()

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

Classic Problems of Synchronization

- The Bounded-Buffer problem
- The Readers-Writers problem
- The Dining Philosophers Problem

Solution to the problems: Using Semaphores

The Bounded-Buffer Problem

- Also known as Producer-consumer problem
- Consists of two processes : Producer and consumer
- The producer process produces information that is consumed by consumer process.
- One solution to the producer consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently it must have available buffer which is filled by the producer and consumed by the consumer.

- A producer can produce one item while a consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item which is not yet been produced.

- Also a pool consists of *n buffers, each capable of holding* one item.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.

Producer process

- Generate the data and put it into the buffer

Consumer process

- Removes data from the buffer

Bounded-Buffer Producer/Consumer Problem

- Shared data:
  `semaphore, full, empty, mutex`
- Initially:
  `full = 0, empty = n, mutex = 1`
  where n is the buffer size

- Make sure that producer will not try to add data to a full buffer
- Also consumer will not remove data from an empty buffer
- Solution for producer – discard data / go to sleep when buffer is full
- Solution for consumer – go to sleep when buffer is empty

The code for the producer process is as follows

```
do
{
……..
 // produce an item in nextp
Item_produced=produce_item();
wait(empty);
wait(mutex);
Buffer[in]=item_produced;
In=(in+1) %size;
// add nextp to buffer
……..
signal(mutex);
signal(full));
}while (TRUE);
```

The code for the consumer process is as follows

```
do
{
wait(full);
wait(mutex);
Item_consumed=buffer[out];
Out=(out+1) % size;
// remove an item from buffer to nextc
……..
signal(mutex);
signal(empty);
……..
// consume the item in nextc
Consume_item(item_consumed);
}while (TRUE);
```

The Readers-Writers Problem

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as **readers and to the latter** as writers.

- Two classes of users:
  - Readers – never modify , only read the database
  - Writers – read and modify the database
- If two readers access the shared data simultaneously, no adverse results.
- If a writer and some other thread (reader/writer) access the data simultaneously, arise inconsistency.

- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database. This synchronization problem is referred to as the *readers-writers problem.*

Two variants of this problem

- The *first readers-writers problem*

- requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object

- The second readers-writers problem

- requires that, once a writer is ready, that writer performs its write as soon as possible.

First Readers/Writers Problem solution

- Shared data:

```
semaphore mutex, wrt
int readcount
```

- Initially:

```
mutex = 1, wrt = 1, readcount
=0
```

- The semaphore wrt – accessed by reader and writer processes. Functions as a mutual-exclusion semaphore for the writers
- readcount variable- keeps track of how many processes are currently reading the object.
- mutex semaphore - ensure mutual exclusion when the variable readcount is updated.

The structure of a writer process.

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

- **The structure of a reader process.**

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

            // reading is performed
        signal(mutex) ;
        readcount--;
        wait(mutex);
        if (readcount == 0)
           signal(wrt);
        signal(mutex);
}while (TRUE);
```

if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n — 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticksthat are between her and her left and right neighbors).

A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick[5]; where all the elements of chopstick are initialized to 1.

- The structure of philosopher i is as follows

```
do {
wait (chopstick [i] ) ,
wait(chopstick [ (i + 1) % 5] ) ;
// eat
signal(chopstick [i]);
signal(chopstick [(i + 1) % 5]);
/ / think
}while (TRUE);
```

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry simultaneously

and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next. we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Monitors

- Monitors were developed in the 1970s to make it easier to avoid deadlocks.
- A monitor is a programming language construct, that controls access to shared data .
- A monitor is a module that encapsulates
  - Shared data structures
    - Procedures that operate on the shared data structures

Characteristics of monitors

- A monitor is a collection of procedures, variables, and DS grouped together.
- Processes can call the monitor procedures but cannot access the internal DS.
- Only one process at a time may be active in a monitor.
- A monitor is a high level language construct.
- The compiler usually enforces mutual exclusion for monitors .
- Monitors are a synchronization mechanism
- The variables defined inside a monitor can only be accessed by the functions defined within the monitor

Monitor declaration format

```
monitor  monitor name;
{
 //   shared variable declarations (private data, local monitor variables)
     procedure p1( . . . .) {
          . . . . .
      }
     procedure p2( . . . .) {
          . . . . .
      }
      .
      .
   procedure pn( . . . .) {
          . . . . .
      }
  initialization code( . . . .) {
          . . . . .
      }
```