Unit 1 IPC Part -2

Topics to Be Covered:

- **Semaphore**
- Classical problems of Synchronization
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Semaphore

- Synchronization tool that does not require busy waiting n 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; // no-op
S--;
 }
 signal (S) {
 S++;
 }
                                                    // remainder section
```

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 mutex locksn Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusionSemaphore mutex; // initialized to do {
  wait (mutex);

// Critical Section

Signal (mutex);

} while (TRUE);

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 crtical 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.

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.

The structure of the producer process

```
do {// produce an item
      in next p wait
      (empty);
      wait (mutex);
          // add
       the item 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 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 writenProblem – 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 read count initial 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 writenProblem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time

Shared Data

- **Readers-Writers Problem**
- The structure of a writer process

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

- 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);
```

Dining-Philosophers Problem

- Philosophers spend their lives alternating 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

- The key question is can we write a program for each philospher that does what it is supposed to do and never gets stuck?

The obvious solution :

- The philosopher wait until the specified fork is available and then seizes it, unfortunately the obvious solution is wrong

Suppose that all five philosophers take their left forks simultaneously ,none will be able to take their right forks and there will be a **deadlock**

- We could modify the prog so that after taking the left fork ,the program check to see if the right fork is available , If it is not the philospher put down the left one ,wait for some time and then repeat the whole process. This solution is too fail let all philosopher could start the algo simultaneously ,picking up their left fork seeing that their right for were not available ,putting down their left forks again simultaneously and so on
- A condition in which all the program continue to run indefinitely but fail to make any progress is called **starvation**
- The solution presented is deadlock free and allow the maximum parallelism for an arbitrary no of philosophers
- It uses an array ,state to keep track of whether a philosopher is eating, thinking or hungry (trying to acquire fork)
- A philosopher may move only into eating state if neither neighbour is eating
- Philosopher's i neighbour are defined by the macros LEFT and RIGHT (If i is 2 LEFT is 1 and RIGHT is 3)
- The program uses an array of Semaphore one per philosopher, so hungry philosopher can block if the needed forks are busy

  Note : that each process run the procedure philosopher as its main code, but the other procedures take forks, put forks and test are ordinary procedure

Dining-Philosophers Problem Algorithm

```
# define N 5      / no of philosopher

# define LEFT (i+N-1)5% N     /no of  i ‘s Left neighbour

# define RIGHT (i+1)% N    / no of I ‘s Right neighbour

# define  Thinking 0

# define Hungry 1

# define Eating 2

Type def  int Semaphore;

int State [N];

Semaphore mutex = 1;

Semaphore S[N];

Void philosopher (int i)

{

While (true) {

Think();

take_ forks(i);

eat();

Put _forks(i);

}

Void take_forks( int i)

{

   down (& mutex); / enter CS
```

```
State[i] = Hungry;
Test (i);              / Try to acquire 2 fork
Up (& mutex);     / Exit CS
Down (& S [i]);   / Block if forks were not acquired
}
Void put_forks (i)
{
 down (& mutex);
State [i] = Thinking;
Test (LEFT);
Test ( RIGHT);
Up (& mutex);
}
Void Test (i)
{
 If (State [i] = = Hungry && State [Left ! = Eating && State [Right] ! = Eating )
{ State [i] = Eating;
   Up (&S[i]) ;
}
```

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

procedure P1 (…) { …. }

procedure Pn (…) {……} Initialization code (…) { … }

}