Date : / /
Page No :

Unit 2
Concurrent Processes

(i) Race condition

(i) It is a situation
(ii) Occur - when cooperating processes accessing shared data at same time

(iii) Problem - (a) Data inconsistency
(b) Output depend on order of process execution

(ii) Critical section

(i) Solution of Race condition
(ii) contain - "shared Resources"
(iii) Part of program
(iv) Place where "cooperating process access shared data"
(v) do {
entry section
critical section
Exit section
Remainder section
} while(true);

entry section - (i) multiple process enter at a time
(ii) Give lock to any one process
Exit section - (i) multiple process enter at a time
(ii) Remove the lock
critical section - (i) single process enter at a time.

(iii) Requirements of a solution to CS problem

1) Mutual exclusion
Single Process enter at a time (CS)
2) Progress
Any process enter at a time, if CS is free
3 Bounded wait
"Process waiting time" is "fixed or limited"
4 No assumption related to H/w

(iv) Types of executable code, solution for entry section

(a) H/w type - Test and set operation (TSL)
(b) OS type - Semaphore
(c) Combiler type - Monitore
(d) S/w type - Peterson's solution, Bakery solution Dekker's solution, lock variable

(v) Peterson's Solution
(i) 2 Process solution
(ii) Program -

For Process Pj

```
Boolean flag[2];
int turn;
flag[0] = flag[1] = false;        // Shared variable

do {   flag[j] = true;
       true = j;
       while(flag[i] && turn==j);   // entry section

       critical section
       flag[j] = False      // exit section
       Remainder section
} while(1);
```

(vi) Dekker's solution
(i) 2 Process solution
(ii) Program -

```
int first = 0
int second = 1- first;
boolean flag[2]
int turn;                                   // Shared variable
flag[first] = flag[second] = false;
→ do {    // For Process first
flag[first] = true;
while( flag[second] )                       // entry section
{
        if ( turn == second)
    {
            flag[first] = false;
            while( turn == second);
            flag[first] = true;
    }
}
        critical section
        turn = second;                      // Exit section
        flag[first] = false
        remainder section
} while (true)
```

(vii) Bakery solution

(i) N process solution

(ii) Token mechanism

(a) Every process recieve, token
(b) Token given increasing order
(c) multi process with same token exist may
(d) If Pi token < Pj token,
Pi enter in CS

If Pi token == Pj token,
then, if Pi < j,
Pi enter in CS.

(e) Pi token = 0 (or set to zero) indicates
(i) Process execution is finished

(iii) do
{ choosing[i] = true
number[i] = max(number[0], ... number[n-1]) + 1;
choosing[i] = false;
for(j=0; j<n; j++)
{
while(choosing[j]);
while((number[j] != 0) && (number[j,j] < number[i,i]));
}
critical section
number[i] = 0;
remainder section
} while(1);

(viii) Test and Set operation (TSL)

(i) N process solution
(ii)

```
while ( test_and_set(&lock));
critical section
lock = false
```

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

(ix) Semaphore
(i) P integer variable
(ii) Used By ~~concurrent~~ cooperative process in mutual exclusion manner
(iii) Disadvantage
(a) Deadlock and starvation may occur
(b) priority Inversion
(c) Requires "Busy waiting"
(iv) operation of semaphore

(a) P() / Down() / wait() → Present in entry section
(b) V() / up() / signal() / → Present in exit section
Post()

(v) Types of semaphore

(a) counting semaphore -

→ For example - 5 printers
are used in

→ Used to control access to a resource that has multiple instance.

→ Down(semaphore S)
{
S.value = S.value - 1;
if (S.value < 0)
{
Put Process(PCB) in suspended list, sleep();
}
else
return;
}
Critical Section

up(semaphore S)
{
S.value = S.value + 1;
if (S.value ≤ 0)
{
Select a process from suspended list
wake up();
}
}

→ semaphore value → (−∞, ∞)

(b) Binary Semaphore / mutex locks

→ For example or used in -
(a) consumer Producer Problem
(b) Dining philosopher Problem
(c) Read and writer Problem
(d) sleeping barber problem

→ used to provide "mutual exclusion" (That why also know as mutex lock)

→ Down(semaphore S)
{
while (S ≤ 0)
S = S - 1;
}

up(semaphore S)
{
S = S + 1;
}

→ semaphore value → (0, 1)

(c) Program -
do {
down (S);
Critical section
up(S);
remainder section
} while(true);

(x) Consumer Producer Problem

(i) Problem Statement

(a) we have Fixed Size Buffer → (Access By Both consume / Producer)
we have Producer (Produce item, Place in Buffer)
we have consumer (Consume item, Picking from Buffer)

(b) Synchronize Producer, Consumer Process, other wise → Problem

→ data inconsistence
→ consumer wait → (High speed of consumer, Buffer underflow)
→ Data loss → (High speed of Producer, Buffer overflow)

(ii) Solution - Binary Semaphore

Problem code

Producer
while(true)
{
while(counter == Buffersize);
Buffer(in) = nextProduced;
in = (in+1) % Buffer size;
counter++;
}

consumer
while(true)
{
while (counter == 0)
nextconsumed = buffer(out);
out = (out+1) % Buffer Size;
count--;
}

Date : / /
Page No :

Solution code

Producer	Consumer
Semaphore empty = n Semaphore full = 0 Semaphore mutex = 1	Semaphore empty = n Semaphore full = 0 Semaphore mutex = 1
do{	do{
Produce an item in nextp	wait(full)
wait(empty);	wait(mutex)
wait(mutex);	Remove an item from Buffer
— — —	Signal(mutex);
add nextp to buffer	Signal(empty);
— — —	consume the item
Signal(mutex);	}
signal(full);	while(1)
} while(1)	

(x) Dining philosopher problem

(i) Problem statement

(a) we have N Philosopher (seated in circular table)
(b) we have 1 chopstick B/w each pair of philosopher
(c) we have 1 Bowl of rice (At center)
(d) Note
Total number of Philosopher = Total number of chopstick
(e) Condition to eat - Pick 2 adjacent chopstick

Date : / /
Page No :

(f) Note → critical section = Bowl

But, In this case 2 independent process can access critical section at same time.

(g) philosopher states → ① eat ② Thinking

(ii) Solution - Binary semaphore

Problem code

```
void philosopher (void)
{ while (true)
   { Thinking();
     take_fork(i);
     take_fork((i+1) % N);
     EAT();
     Put_fork(i)
     Put_fork((i+1)) % N);
   }}
```

Solution code

```
For Philosopher (i to N-1)
semaphore S[i] = 0;
void philosopher (void)
{ while (true)
   { Thinking();
     wait(take_fork(Si));
     wait(take_fork(S(i+1)%N);
     EAT();
     Signal(Put_fork(Si));
     Signal(Put_fork(S(i+1)%N);
   }
}
```

```
For philosopher (N)
semaphore S[i] = 0;
void philosopher (void)
{   // same
    // same
    wait(take_fork(S(i+1)%N);
    wait(take_fork(Si);
    //same
    Signal(Put_fork(Si));
    Signal(Put_fork(S(i+1)%N);
}
```

(iii) Note - If we use only (1 to N-1) Function FOR All philosopher. →

Deadlock Problem may occur.

(a) Some other Theoritical solution → using (1 to N-1) Function only

→ Allow at most N-1 philosopher Sitting simultaneously
→ Allow N+1 chopstick
→ Allow philosopher to pick up chopsticks → only if both chopstick available.

(xii) Reader and write problem

(i) Problem Statement

(a) we have a data base
(b) we have 2 process

→ Read (only Read data)
→ write (can Read and write data)

(c) Problem when simultaneously

W → R	→ occur	on same data
W - W	→ occur	
R - R	→ No Problem	
R - W	→ occur	

(ii) Solution - Semaphore

```
int rc = 0
semaphore mutex = 1
Semaphore db = 1

void Read (void)
{
  while(true)
  { down(mutex);
    rc = rc+1;
    if(rc==1) then down(db);
    up(mutex);
    DB / Critical section
    down(mutex);
    rc = rc-1
    if(rc==0) then up(db);
      up(mutex)
    Process_data
  }
}

void write()
{
  while(true)
  {
    down(db);
    DB
    up(db);
  }
}
```

(xiii) Sleeping Barbar Problem

(i) Problem Statement

(a) we have a Barber shop
(i) 1 Barbar (ii) 1 Barber chair (iii) N waiting chair
(b) when customer arrive

(i) If no customer → Barbar sleep (on his chair)
(ii) [illegible] → customer wait on "n" chair
[illegible] Barbar busy
(iii) no waiting chair → customer left the shop

(ii) Solution -

```
Semaphore customer = 0;
Semaphore barber = 0;
access seats mutex = 1;
int Number of free seats = N;
```

// Shared variable

```
Barber { while (true)
         { wait (customer);
           wait (mutex);
           Number of free seats ++;
           sem-post (Barber);
           sem-post (Mutex);
         }
```

```
customer  while (1)
          { sem-wait (access seats);
            if (Number of free seats > 0)
            {   Number of free seats --;
              sem-post (customers);
              sem-post (accessseats);
              sem-wait (barber);
            }
            else
              sem-post (access seats);
          }
```