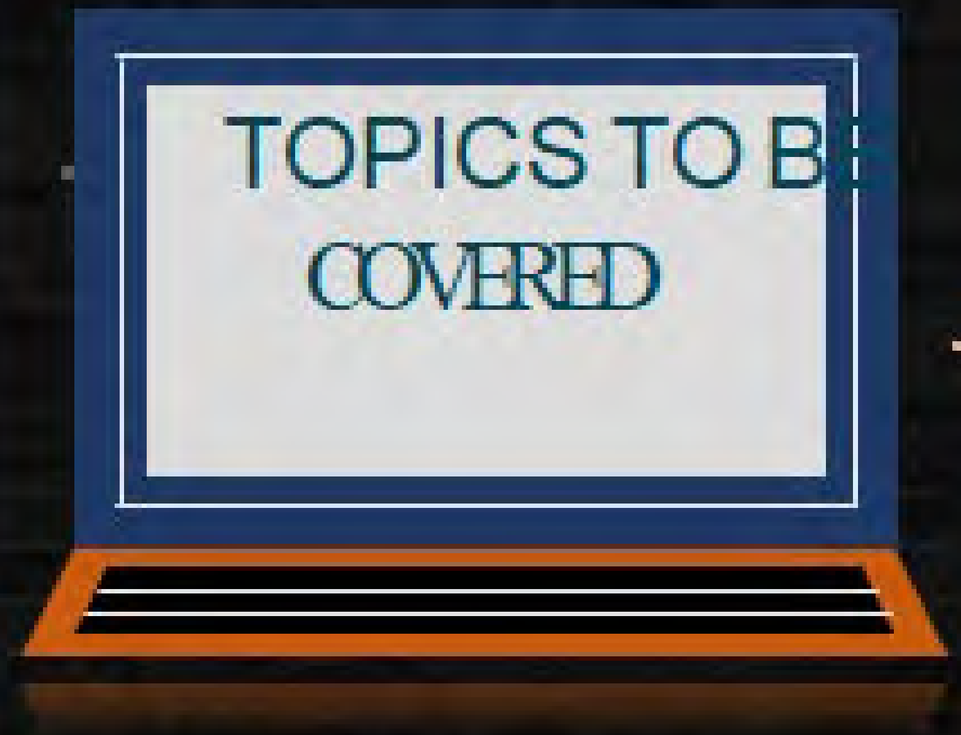
COMPUTER SCIENCE

Process Synchronization

Lecture No 04

Dr. KHALEEL KHAN SIR



01 Strict Alteration

2. Peterson Algoritm

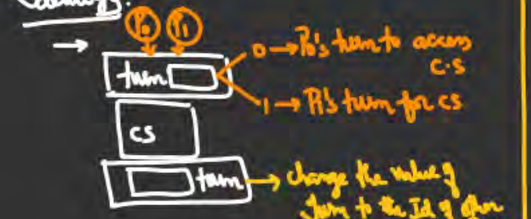
3. Lock variables

4. Test and set Instructions

2) STRICT ALTERNATION

- → 2-Process Solution (P0, P1)
- → Busy-waiting
- → S/w Soln | U-mode

Ideology:




```
int turn = rand(0,1);
```

turn = 0

```
void Process(0)
{
  while(1)
  {
    Non_CS();
    Entry: while(turn!=0);
    CS
    Exit: turn=1;
  }
}
```

```
void Process(1)
{
  while(1)
  {
    a) Non_CS();
    b) while(turn!=1);
    c) CS
    d) turn=0;
  }
}
```

(i) Mutual Exclusion: Guaranteed ✓

(ii) Progress: NOT GUARATEED

(iii) Bounded waiting: Guaranteed

Generalized Implementation

```
int turn = rand(i,j);
void Process(int i)
{
  int j = NOT(i);
  while(1)
  {
    a) Non_CS();
    b) while(turn!=i);
    c) CS
    d) turn=j;
  }
```

Q.1

Several concurrent processes are attempting to share an I / O device. In an attempt to achieve Mutual Exclusion each process is given the following structure. (Busy is a shared Boolean Variable)

```
<code unrelated to device use>
[ Repeat
  until busy = false; ]
  Busy = true;
<code to access shared >
  Busy = false;
<code unrelated to device use>
```

NCS

while (busy != FALSE);

while (lock != 0);

Waste cpy Time

LV
- ME : X
- Prog : ✓
- B.W : X

Which of the following is (are) true of this approach?

I. It provides a reasonable solution to the problem of guaranteeing mutual exclusion.

II. It may consume substantial CPU time accessing the Busy variable.

III. It will fail to guarantee mutual exclusion.

(A) I only (B) II only (C) III only (D) I & II (e) II & III

Q.2

Consider the following two-process synchronization solution.

Process 0

```
Entry: loop while (turn == 1);
       (critical section)
Exit: turn = 1;
```

Process 1

```
Entry: loop while (turn == 0);
       (critical section)
Exit: turn = 0;
```

The shared variable turn is initialized to zero.

Which one of the following is TRUE?

(A) This is a correct two-process synchronization solution. ✗

(B) This solution violates mutual exclusion requirement. ✗

(C) This solution violates progress requirement. ✓

(D) This solution violates bounded wait requirement. ✗

Strict Alternation

3) PETERSON'S SOLUTION

→ 2-Process Solution (P_0/P_1)
→ Busy-waiting
→ S/w Solution @ UM

Ideology: <L.V + S.A>

```
#define N 2
#define TRUE 1
#define FALSE 0
int FLAG[N]={FALSE};
int turn;
```

FLAG: indicates the current Interest of the Process (whether it is intrstd in CS or NOT)

FIFO

```
void Process(int i)
{
  int j = NOT(i);
  while(1)
  {
    a) Non_cs();
    b) FLAG[i]=TRUE;
    c) turn= i;
    d) while(FLAG[j]==TRUE && turn==i);
    e) <cs>
    f) FLAG[i]=FALSE;
  }
}
```

1. Mut. Exclusion : ✓

RQ | P_0 P_1

FLAG[0] = ~~F~~ T
FLAG[1] = ~~F~~ ~~T~~ ~~F~~ T

1 / 0 turn

CPU

CS

BW

t_1: P_0 : a; b;
i=0; j=1

PV

t_2: P_1 : a; b; c; d : P_0 has not yet updated turn
i=1; j=0

PV

t_3: P_0 : c; d
i=0; j=1

PV

t_4: P_1 : d : e: <cs>
i=1; j=0

Bounded waiting:

FLAG P_i =T ✓

P_j X

FLAG =FALSE

Progress: ✓

Limitations:

1) 2-process soln
2) Wastage of cpu time

Q.3

```
void Dekkers_Algorithm (int i)
{
    int j = ! (i);
    while (1)
    {
            (a) Non_CS( );
            (b) flag[i] = TRUE;
            (c) while (flag [j] = = TRUE)
            {
            if (turn = = j)
            {
                    flag [i] = FALSE;
                    while (turn = = j);
                    flag [i] = TRUE;
            }
            }
            (d) <CS>
            (e) flag [i] = FALSE;
            turn = j;
    }
}
```

P. T

(i) M	E

(ii) Progress

(iii) Bounded wait

II. HARDWARE MECHANISMS : Some Processors Support or Provide Special {Atomic Instns} (Privileged) to solve CS Problem [Lock-based Solns]

→ Extension to Lock variable

① → {TSL} : (Test & Set Lock)
② → {SWAP}

→ Multi-process Soln

→ Busy-waiting

→ H/w Category

1. TSL (Test and Set Lock)

Syntax: `Bool TSL(&Lock);`

Semantics: TSL when executed returns the current-value of Lock & Sets the value of lock to TRUE; Atomically

0 — FALSE
1 — TRUE

```
Bool TSL(Bool *target)
{
    Bool rv;
    {
        rv = *target;
        *target = TRUE;
        return(rv);
    }  Atomically
}
```

Pi: TSL → 0

Pi: TSL → 1

Um:

```
Bool Lock = FALSE;
Void Process(int i)
{
    while(1)
    {
        a) Non_CS();
        b) while(TSL(&lock) == TRUE);
        c) <CS>
        d) LOCK=FALSE;
    }
}
```

Logic (additional)

I. Mutual Exclusion: ✓

Whichever Process execute TSL first will enter CS & all others Busy wait;

II. Progress: ✓

III. {Bounded Waiting}:

THANK YOU!