L3-1 Process Synchronisation - Process Types - Race Condition

Processes

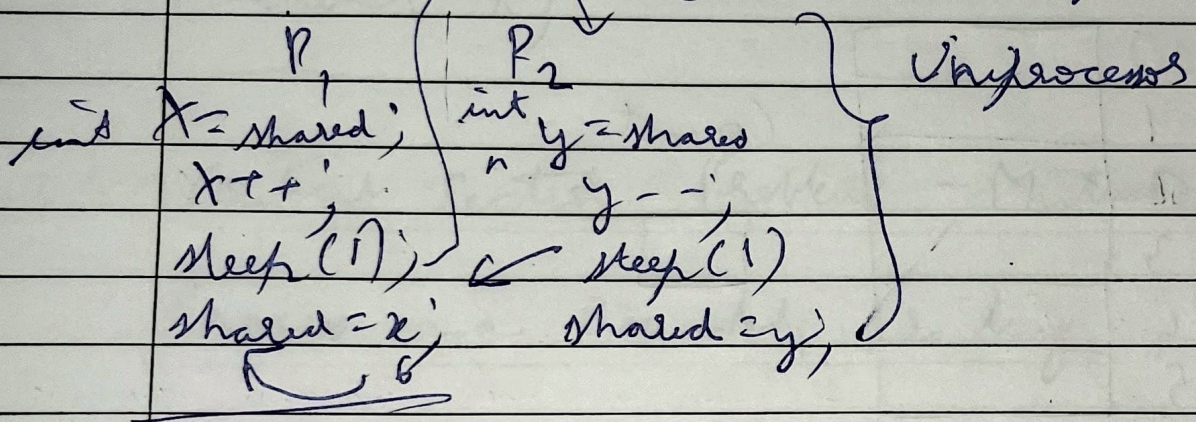
Cooperative processes — Independent processes

↓

Process can share

* Variable
* Memory
* Code
* Resources — CPU, Printer, Scanner

→ Context switching




Terminated

shared gets updated 6 times

This process is called race condition

---

## L-3.2 Producer Consumer Problem

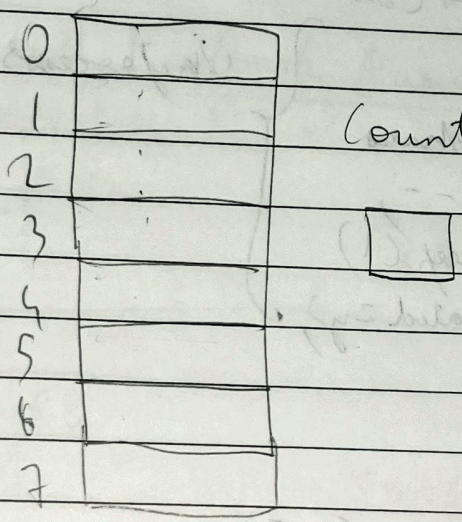
```
void consumer (void)
{
    int itemC;
    while (true)
    {
```

Buffer empty

```
while (count == 0);
item = Buffer (out)
out = (out + 1) mod n;
count = count - 1
process - item (item);
}
}
```

1) Load $R_c$, n[count]
2) DECR $R_c$
3) Store n[count], $R_c$

n = 2
Buffer [0 ... n-1]




```
int count = 0;
void producer (void)
{
  int itemp;
  while (true)
  {
    produce_item (itemp);
    while (count == n); -> Buffer overflow
    buffer [in] = itemp;
    in = (in + 1) mod n;
    count = count + 1 ——→ Load R, n[count]
                          INCR R;
                          Store n[count], R
  }
}
```

Case I: Best Case

Producer produces items and stores in the buffer

Consumer consumes the items from a particular position from the buffer.

Case (ii) - If producer preempts & next, consumer preempts

Flow: Produce $I_1$ $I_2$ Consume $I_1$ $I_2$ Produce $I_3$

Consume $I_3$

L-3-4 Critical Section Problem - Mutual Exclusion, Process and Bounded Waiting

Critical Section → it is part of the program where shared resources are accessed by various processes

cooperative

* Critical section is the place where shared variables, resources are placed

Process-1

```
main()
{
    A, B Non-critical section
    Entry section
    {
        count++;
    }
    Exit Section
}
```

Process-2

```
main()
{
    Non-critical
    C, D
    Entry Section
    {
        count--;
    }
    Exit section
}
```

Race condition occurs if critical section of 2 process are accessed simultaneously

To access critical section, the entry section code must be executed

Rules for synchronisation mechanism

Primary
1) Mutual Exclusion
2) Progress

Secondary
3) Bounded Wait
4) No assumption related to hardware of speed

1) Mutual exclusion :- If one process is using common code of critical section, another process cannot enters the critical section

2) Progress :- There is no progress if neither of the code processes are accessing the critical section

3) Bounded wait :- Let one process access critical section as many no. of times as possible, but it should not be that one process keeps on using the critical section

L-3.5 Lock Variable in OS

Critical Section Solution using "Lock"

```
do{
    acquire lock
        CS
    release lock
}
```

Execute in user mode

* ~~Multiprocessor~~ Multiprocess solution
* No mutual exclusion
Guarantee

```
1  while(LOCK==1);      ENTRY
2  LOCK=1               CODE

3  Critical section

4  LOCK=0               EXIT CODE
```

Case 1:- $P_1$ $P_2$ processes

~~If~~ If one process has made LOCK=1, then another process can't access at the known LOCK=1.

Case 2:- If $P_1$ preempts by $P_2$ accesses the C.S. then $P_1$ comes back, and accesses the C.S. then, there is no mutual exclusion.

1-3-6 Test and set instruction

__/__/__

while (test_and_set (& lock));
CS
lock = false

{ Replace this with ① & ② instruction

boolean test_and_set (boolean * target)
{
boolean x = * target;
* target = TRUE;
return x;
}

* Initial value is 0 for lock

This method asures both mutual exclusion as well as progress

-3-6 Semaphores

Semaphore
(Counting (-ve to +ve)) (Binary (0, 1))

Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve synchronisation

Operations :- P(), Down, Wait
V(), UP, Signal, Post, Release

Entry section

```
Down (Semaphore S)
{
  S.Value = S.Value - 1;
  if (S.Value < 0)
  {
    Put Process (PCB) in
    suspended list, Sleep();
  }
  else
    return;
}
```

If $S = -4$ then 4 processes are present in the block state

If $S < 0$, 0 processes are in the suspended list

If $S = 10$ 10 processes can enter critical section

```
Up (Semaphore S)
{
  S.Value = S.value + 1;
  if (S.value <= 0)
  {
    Select a process
    from suspended list,
    Wake up();
  }
}
```

$S = 17$

5P, 3V, 7V, 1P

$|17 - 5 + 3 \cdot 2| = 14$

L-3.9 What is binary semaphore?

Binary Semaphore

0 1

```
Down (Semaphore S)
{
  if (S.value==1)
  {
    S.value=0;
  }
  else
  {
    Block this Process
    and place in suspend
    list, sleep();
  }
}
```

```
Up (Semaphore S)
{
  if (Suspend list is
  empty)
  {
    S.value=1;
  }
  else
  {
    Select a process from
    suspend list and
    wake up();
  }
}
```

Eg:- P1 P2

Down(S) Down(S)

S=0

CS CS

Nothing will execute → deadlock

up(S) up(S)

L-3.11 Solution of Producer Consumer Problem using Semaphore

Counting Semaphore

full = 0 → No. of filled slots

empty = N → No. of empty slots

Binary Semaphore S=1

IN 3 Out 0

| Producer | Consumer |
|---|---|
| Produce_item(item);<br>{<br>Wait<br>down(Empty);<br>down(S);<br>Buffer[IN]=<br>item;<br>IN=(IN+1) mod n;<br>up(S);<br>up(full); | 1) down(full);<br>2) down(S);<br>itemC = Buffer[out];<br>out=(out+1) mod n;<br>3) up(S)<br>4) up(Empty) |

N=8

| | |
|---|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

EN [1] N
see [2]

Exit [3]
see [4]

empty=5 4
full=3 4 3
S=1 0 1
In=(3+1) mod 8=4

Case 2: With preemption

N=8

In 3

Out 0 1

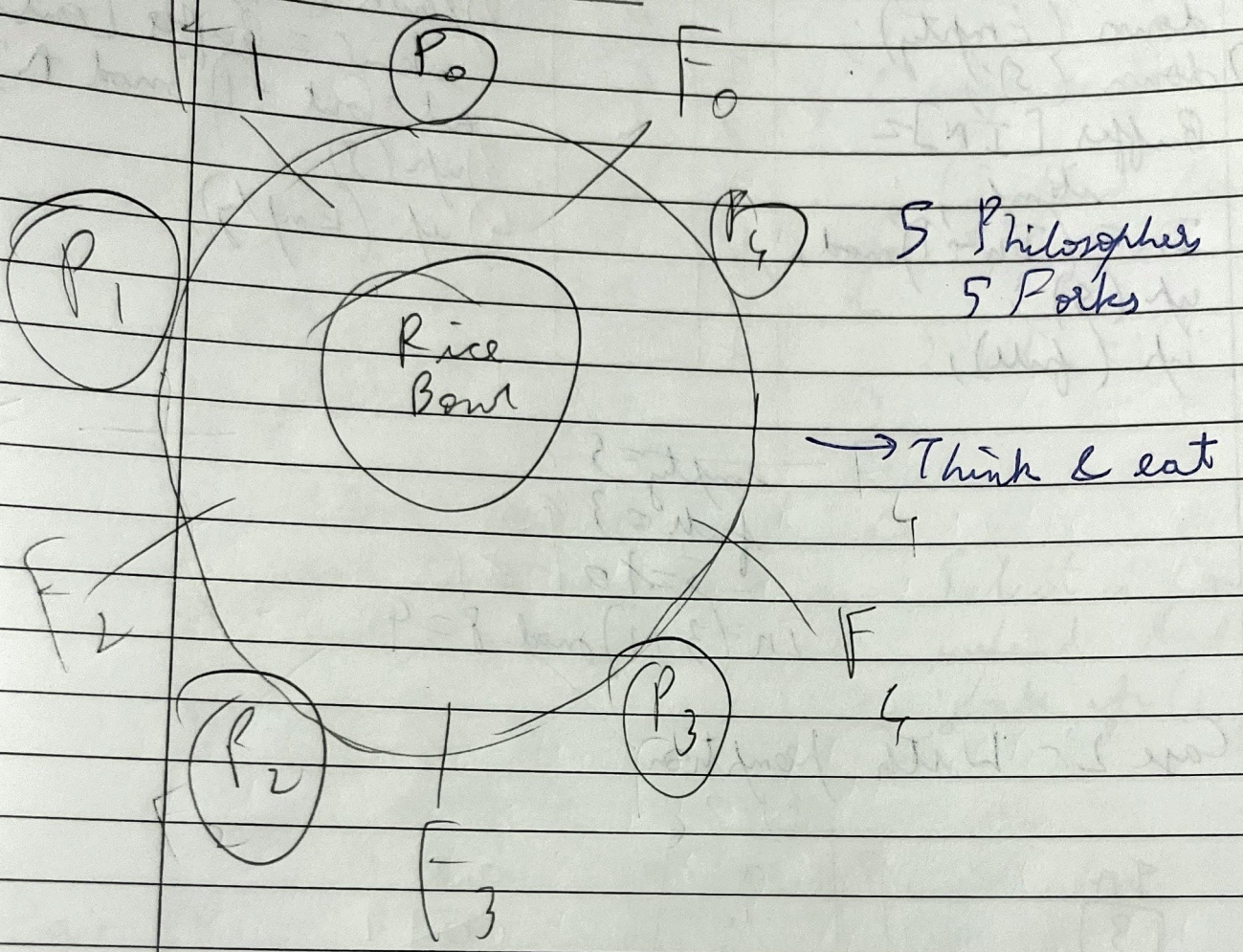
| | |
|---|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

Empty=5 4 5
full=3 2
S=1 0 0

Produced wants to sit down(S)
There is no inconsistency created, as mutex is there

L-3-13

# Dining Philosopher's problem & solution using semaphore

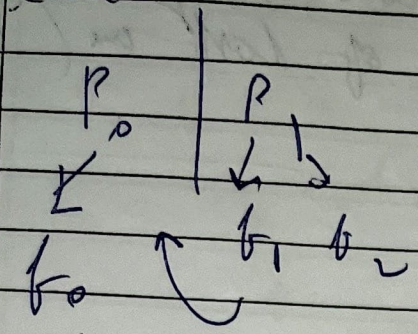



5 Philosopher
5 Forks

→ Think & eat

```
void Philosopher(void)
{
  while (true)
  {
    Thinking();
    take_fork(i);          // left fork
    take_fork((i+1)%N);    // right fork
    EAT();
    Put fork(i);
    Put fork((i+1)%N)
  }
}
```

Case 2




S[i] five semaphores

$S_0$ - $S_1$ $S_2$ $S_3$ $S_4$

Using semaphores,

void Philosophers (void)

{

while (true)

{

Thinking ();

Entry { wait (take_fork ($S_i$));

wait (take_fork ($S_{(i+1) \bmod N}$)) $P_0$

EAT ();

Exit { Signal (put fork (i));

signal (put fork (i+1, N))

$P_0 \rightarrow S_0 \quad S_1$

$P_1 \Rightarrow S_1 \quad S_2$

$P_2 \rightarrow S_2 \quad S_3$

$P_3 \rightarrow S_3 \quad S_4$

$P_4 \rightarrow S_4 \quad S_1$

When 1 philosophes is picking up second fork, there are chances of preemption. That time there is a deadlock (All the processes should preempt)

The problem can be solved by assigning any one order of the forks of last one (of the one)