Tashfeen 22i-2041, NUCES, FAST, PAKISTAN.

Thread/Process share resources
↑

• — Process Synchronization

Processes

- Cooperative Processes (Kush bhi common share hoo ksi k! ~~Execution ast say~~ pr kuch effect daal skti hy)
- independent processes.

Example:-

int shared = 5 6/4. • but wrong.

• Race Condition also

P1	P2
int x = shared	int y = shared 5
x++;	y--;
sleep(1);	sleep(1);
shared = x	shared = y;
6	4
terminated	

• — Producer - Consumer Problem (bounded buffer Problem)

(~~• Multi process~~ • Both processes are parallel and share sth)

• Count is set to 0 initially.
Producer — increment count
Consumer — decrement count

• Problem:- • Increment (count++) & decrement (count--)
• It might overright as in above example.

- Synchronization between producer & consumer
  - use three separate semaphores.

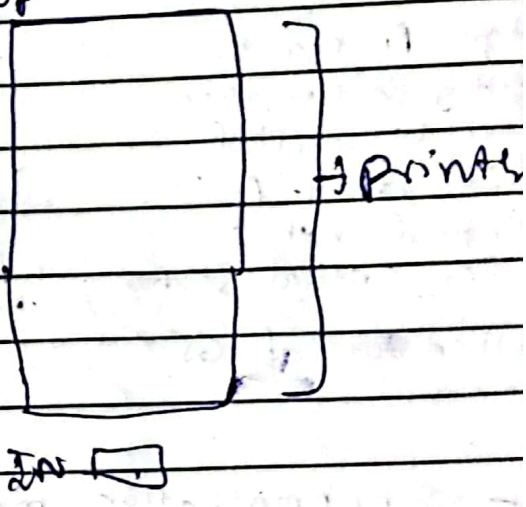
```
int count=0

Void consumer (void)
{
  int itemC;
  while (true)
  {
  while (count==0);
  itemC = Buffer (out);
  out = (out+1) mod n;
  count = count -1;
  Process item (itemC)
```

```
Void Producer(void)
{ int temp;
while (true)
{ Produce item (temp)
while (count==n);
Buffer[in] = temp;
in=(in+1) mod n;
count = count +1;
} }.
```

- Print - Spooler Problem.

1) Load Ri, m[in]
2) Store SD[Ri], "F-N"
3) INCR Ri
4) Store m[in], Ri.



Problems:-

- Process wishes to print a file so, it adds name in spooler directory.
- 2 process are in same slot data lost.

• At any time, one process in critical section (Mutual exclusion)

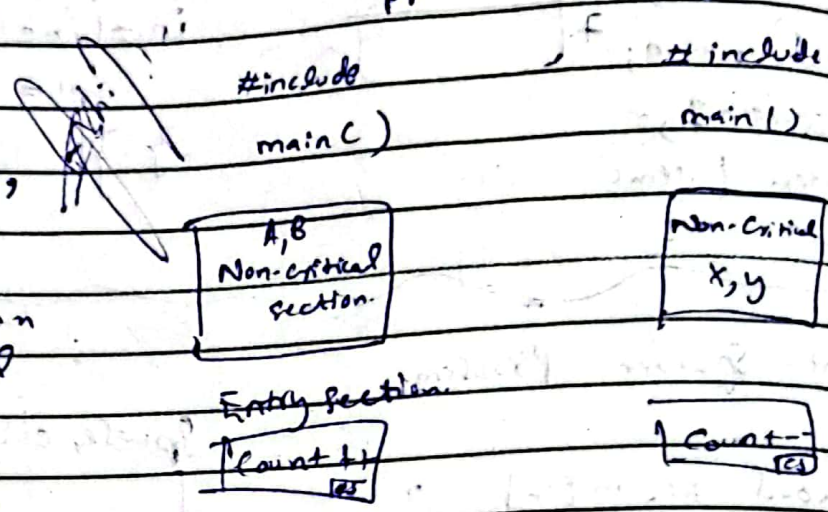
Critical region → • Part of program where shared memory is accessed

•– Race Condition

• Several process access and manipulate same directory or data.

• These can be avoided if no two processes are in critical region at some time.

•– Critical Section Problem



• If P_1 & P_2 Wanna to we ~~Critical Section,~~ ~~before it, it~~ should execute entry section, then move to critical section.

• Agr P_1 executes entry section, then ~~move to critical~~ section., thin P_2 should not enter critical section

• When out of CS exits

•– Synchronization mechanism:-

Conditions / Rules.

1) Mutual exclusive → • P_1 is in Critical section. • P_2 can not enter critical section

2) Progress → • P_1 Wanna enter critical section. P_2 Rahe hy. progress rahi hy ya. • No process should block other.

3) Bounded wait → • P_1 Critical section sa bhr aya. P_1 Critical section ko access kr rahe hy 00 times. P_2 ki turn ho

4) No assumption related to H/W speed →

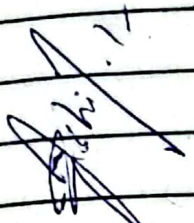
Before entering Critical section, a process should know if any other is already in critical section or not

- Lock variable (for solving critical section problem)

```
do {
    acquire lock
        or
    release lock
}
```

1. while (lock == 1);	Entry code
2. lock = 1	
3. Critical section	
4. lock = 0.	Exit Code.

- execute in user mode
- Multiprocess solution
- No mutual exclusion.
- Problem :- If P_0 executes line 1 and then there is emergency. P_0 chala gya kahin. P_2 aya and critical section enter kiya. Phir P_1 jb aya it will execute 2nd line aur wo bhi critical section mein. Mutual exclusion is finished.



- Turn Variable (strict Alteration) → turn wise.
  - 2 process solution (Only for 2 process)
  - Execute in user mode.
  - Mutual execution ✓
  - Progress ✗ b/c P_0 P_1 ko yak value and P_1 P_0 ko. after critical section.

Process P_0	Process P_1
while (turn != 0);	while (turn != 1);
critical section	critical section
turn = 1;	turn = 0;

  - Bounded wait ✓
  - taking turns is not good idea if one process is slow.
  - Process 1 turns but maybe it is not interested.

↓ Solution.

```
P0  while (true)
    interested[0] = TRUE;
    while (interested[1] != FALSE);
```

```
P1  while (True)
    { interested[1] = True;
    while (interested[0] != FALSE;
```

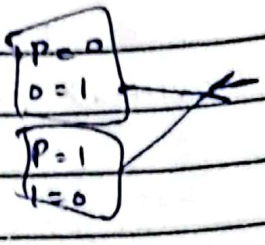
Deadlock.

- Peterson & Bakery algo solves critical section problem.
- Are not used in practice.

-- Peterson Method. (2 process)

$P_0 = 0$
$P_1 = 1$

	0	1
Interested	F	F
Turn	0	




```
Entry Section (Process)
{
1. int other;
2. other = 1 - process;
3. interested [process] = TRUE;
4. turn = process
5. while (interested [other] = TRUE
        && turn = process);
}
Critical Section
Exit Section (process)
{
6. interested [process] = FALSE;
}
```

-- Lamport's Algo | Bakery Algo. (n process)

1) n number of process.
2) Token number to processes
3) Token number in same enumeration
4) Token number. if not assigned and when finishes execution zero/false is assigned.
5) Token id ~~least~~ will enter Critical section.
6) 2 process can have same token number. Than see id.

- Boolean Choosing [N]
  - initially it is false.
  - ~~when Pi is choosed,~~ token is set to TRUE.
- int Number [N]
  - initially 0.
  - if it wants enter number should be non-zero.

- Software - Based Solution of Critical Section Problem.
  Peterson & Bakery.
- Mutual Exclusion using hardware support
  - interrupt disabling → • process disables interrupts while executing its critical section.
    • disabling interrupts guarantees mutual exclusion.

```
while(true)
{ // disable interrupts
   critical section
 renables interrupts
   remaining section
}
```

- Disadvantages :- • ~~Tasks~~ processor can't switch b/w tasks when interrupt is disabled.
  - If we have more than one processor, disabling interrupts on one processor won't stop the others. So, Mutual exclusion is not fully achieved.
  - ~~restricted for user programs.~~

- These instructions perform reading & writing in one atomic (indivisible) step. No other process can interrupt.
- Uniprocessor System : These instructions run without being interrupted.
- Multiprocessor System : Executed with lock system bus.

Test & Test Instruction (Solution to critical section solution)

Mutal ✓
progress ✓.

~~while(test_set(&lock));~~

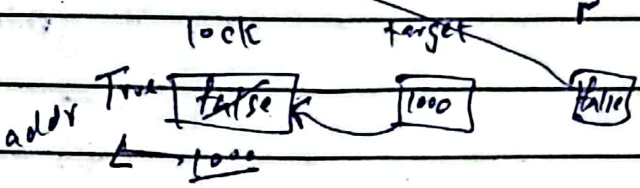
• Agr true aya jee ton loop main hee phas jae ga.

ai ko ye bana dia

lock main jo problem thi us ko solve.

```
while (lock==1);
{ lock=1
} Critical Section
  lock:=0
```

```
boolean test_set (boolean *target)
{
      boolean r = *target;
      *target = True;
}     return r;
```

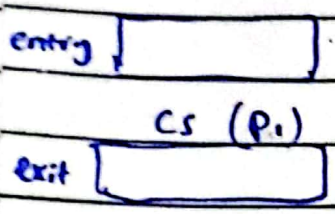


• — Mutual exclusion using machine Instructions

Advantages → • Works on any system
• Simple to implement

Disadvantages → • Busy waiting
• Starvation
• Deadlock in priority Scheduling
(If low-priority process is in critical section, high priority process will wait.
• Medium-priority can block low priority from finishing, causing high priority to wait indefinitely.)

• — Priority Inversion Problem. (Spinlock or deadlock)

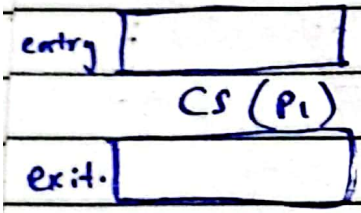


P0

But P0 >> P1
↓
priority of P0 much higher than P1

Then we take CPU from P1 but P1 remains in Critical section. So, P0 can not come in Critical section

• — Priority Inheritance (solution to Priority inversion)



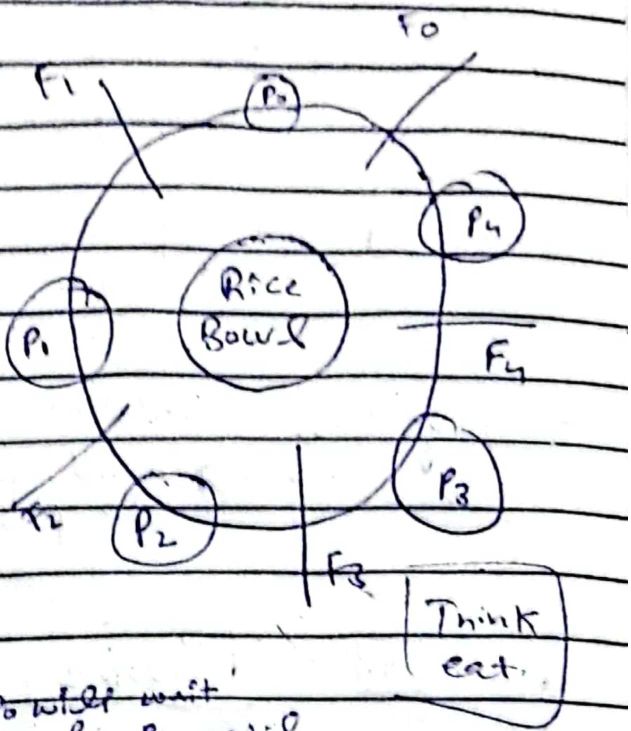
P2 P2 >>> P1

→ • for sometime, P1 apni priority high kr leta hy.
• when P1 comes out, means exit, it gains original priority which is low.

philosopher
↓
think → left fork → right fork → eat.

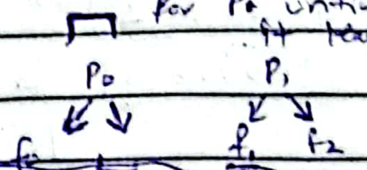
The philosopher apni left fork uthaye, phir right uthaye, say phir prompt ab right fork that beh mi utha. Sem. (deadlock in this case).

• — Dining Philosopher's Problem




```
void philospher(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);
}
}
```

P0 will wait for P1 until it leaves f1.



• We will use binary semaphores. Jitnay forks utnay hae semaphores S0 S1 S2 S3 S4

S0	S1	S2	S3	S4
1	1	1	1	1

Same code as above just some changes | Critical section main 2 bdh ake sath pr independent ho.

```
{ "
  "
  wait (take_fork(i))          } entry
  wait ( take_fork((i+1) % N)) }
  EAT()                        } critical section P0 P2
  signal ( put_fork(i)),       } exit
  signal( put_fork((i+1) % N)); }
}
```

P0 :	S0	S1
P1 :	S1	S2
P2 :	S2	S3
P3 :	S3	S4
P4 :	S4	S0

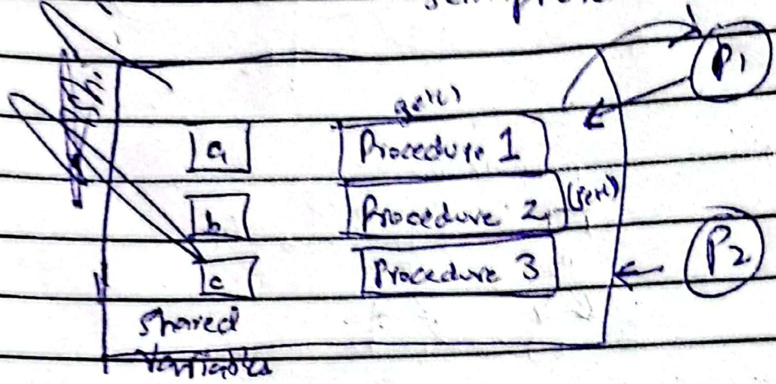
• P0 aya, S0 aur S1 ko down kr diya ge, like values 1 to 0 of both S0 and S1.
• P1 block, q kay wo down nai kr skta S1 ko aur.
• P2 down S2 and S3.

• Dead lock khatam karnay kay liye hum values hae reverse kr daaty hain like] P4 | S4 | S0] hy. ais ko [P4 | S0 | S4] kr daatey hain. P4 block ho jy ga.

• Ab S3 kay pas right fork aa gya. khaana ~~khaye aur phir fork ko release kr day 1~~ kr ~~alga ab aur ko so aur~~ use kr sleeping hain philosopher.

• Now, P4 ka sequence change in code also. phly right then left.

•— Moniters. (Solution to Critical section)
↓ class • → Avoid problems of managing multiple semaphores.



• Mutual exclusion guarantees.

•

• 1 process enters moniter at a time
• When enter, then there is a lock.
• P1 can not access shared variables directly, it will get it from procedures.
• P1 bhr aaya. P2 enters.

```
moniter
{   condition variables;   → Operate on 2 operations
                              • wait()
                              • signal().
    variables;
    Procedure 1
    {      }
    Procedure 2
    {      }
```