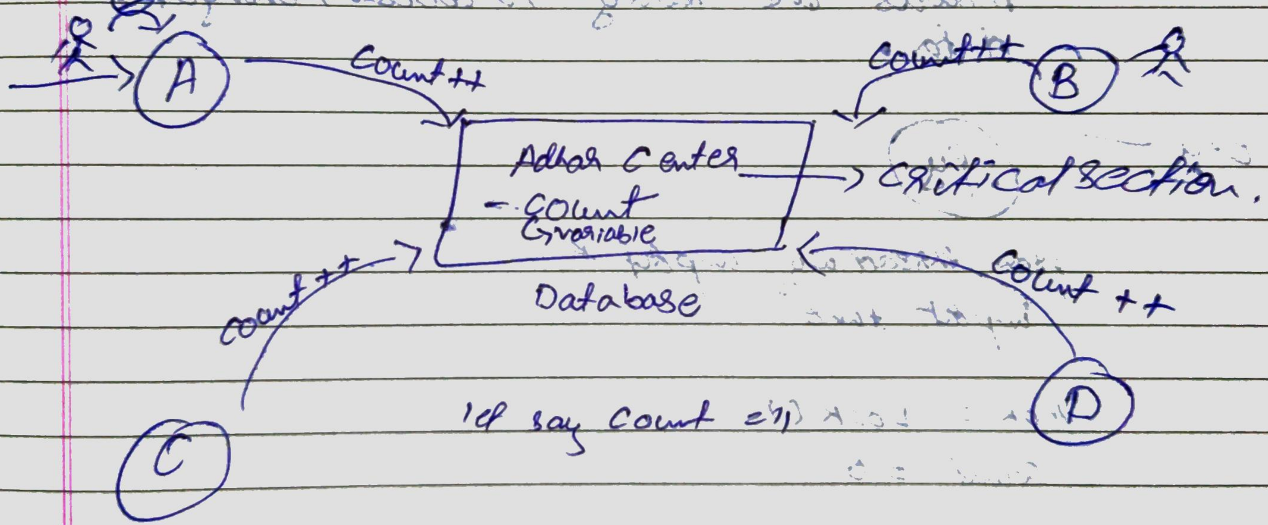
LEC-16 | critical section problem
Race condition in OS.

* Process synchronization techniques play a key role in ~~maint~~ maintaining the consistency of shared data.

* ~~Critical Section~~ (C.S)
(a) The critical section refers to the segment of code where processes/threads access shared resources, such as common variable & files and perform write operations on them. Since processes/threads execute concurrently, any process can be interrupted mid execution.

→ ~~Major thread scheduling issue~~



if say count = 11

A → count ++
↳ temp = count ++
temp = 12
Now context switch → B
Count = temp

B → Count ++
temp = count ++
temp = 11 + 1 = 12
Count = temp

A or B ke saath count ++ kiya phir bhi count = 12 hai
~~ye to sahi nahi~~ hai ~~aise~~

i'c Race cond^{n}.

* Major thread scheduling issue.

→ Race condition.

- A race cond^{n} occurs when two or more threads can access shared data & they try to change it at the same time. B/C the thread scheduling algo can swap b/w threads at any time, you don't know the order in which the threads will attempt to access the shared data.

=> The result of the change in data is dependent on the thread scheduling algo, i.e both threads are "racing" to access / change the data.

e.g —

(.py)

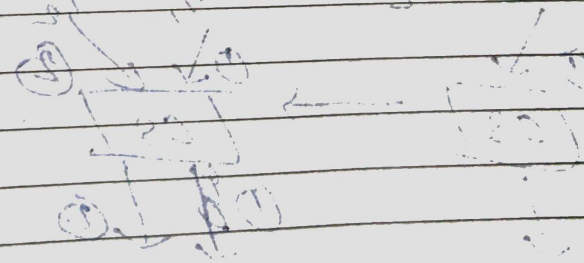
```
from threading import *
import time

lock = LOCK()
count = 0

def task():
    # lock.acquire()
    global count
    for i in range(1000000):
        count += 1
    # lock.release()
```

if __name__ == '__main__':
    t_1 = Thread(target = task)
    t_2 = Thread(target = task)

t_1.start()
t_2.start()
t_1.join
t_2.join()
print(count)

o/p → expected 2000000 ✗
but not
output is inconsistent
har baar naya value aa rha hai.

temp = count + 1
count = temp
} ise position pe context switch hogya to inconsistent data aaega
as count ka update hua nhi.



Race cond^n sol^n

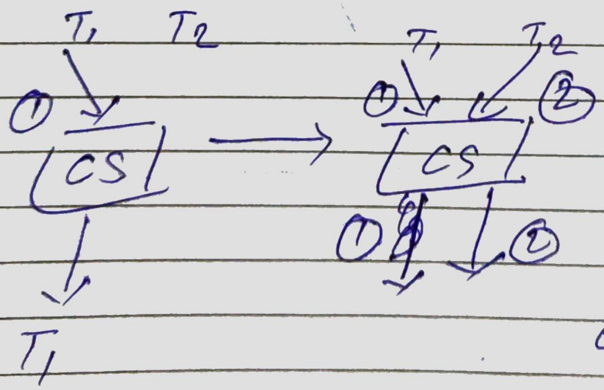
① count ++ ⟹ temp = count ++
count = temp.

isko kisi tarah atomic operation bnade i.e ki ek he CPU cycle mae execute ho jae to apna kaam ban jaega

In C++ ⟶ atomic variable banane ka code hota h
atomic <int>
jisse thread safe hota h ⟶ i.e kbhi inconsistent data degenerate nhi karega.

Mtlab manipulation pura hoga tbhi context switch hoga.

② Mutual exclusion using locks



① T_1

② T_2

or T_2 phir T_1

e.g same code phle wala but concurrent codes.

Locks ⟶ Mutex.

i.e T_1 thread lock karegi th T_2 andar nhi ja paegi & vice versa for T_2

or jb tk T_1 unlock nhi karegi tb tk T_2 andar nhi ja skti.

⑤ Semaphores.

→ Soln of CS should have 3 cond^n

① Mutual exclusion → by locks/mutex

② progress

③

T1 T2

[CS]

Agar koi bhi thread CS me andar nhi h to kisi or thread ko CS jane se na roke.

aur a koi fixed order na ho T1 vhi jaegi to T2 vhi jaegi.

④ bonded waiting

koi bhi thread h wo wo indefinite wait na kare jaise agar 3-4 threads h aisa na ho 3 thread ko mauka to mil gya h CS me jane ka lekin 4th thread ko mauka hi nhi mil gya CS jane ka matlab ek time bound hona chahiye us particular time ke baad thread h wo CS me ghus hi jaegi.

⇒ Limited waiting honi chahiye

Eg. ③ → cond^n is optional for soln.

Q. Can we use a sample flag variable to solve the problem of race cond^n?

⇒

→ turn flags → 0/1

initially turn = 0

T1

while (1)
{
    0 ≠ 0 ⇒ false
    while (turn != 0);   ← This line is false
    CS   ← T1 ghus jaega
    turn = 1
    RS
}

T2

while (1)
{
    0 ≠ 1 ⇒ True ⇒ wait
    while (turn != 1)
    CS
    Turn = 0
    RS
}

let turn = 0 ⇒ T1 —then→ T2

ab ye achieve hogya ekbaar ek Thread CS mae jaega.

then Turn = 1 ⇒ T2 —then→ T1

ab T1 wait & T2 → CS

T1 or T2 pe ek fixed order aa rha h. depend on initial turn value.

progress Cond^n ✗

=> Single flag X ... improved soln

* **Peterson's solution**

can be used to avoid race cond^n but holds good for only 2 process / threads.

- flag [i]
- turn

flag [i] → indicate if a thread is ready to enter the CS flag [i] = true => that pr is ready.

turn → indicates whose turn is to enter the C.S.

i/l.

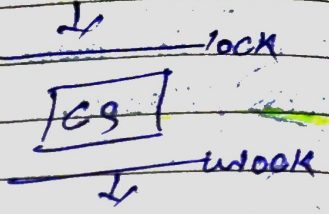
```
T1                                  T2
while (1)                           while (1)
{                                   {
  flag[0] = T                         flag[1] = T
  turn = 1                            turn = 0
  while (turn == 1 && flag[1] = T);   while (turn == 0 && flag[0] = T);
  CS                                  CS
  flag[0] = F                         flag[1] = F
}                                   }
```

=> Mutual exclusion is maintained.

=> Ib T_1 or T_2 CS mae whi hoga ye depending upon kis kaisa CPU schedule kr rha h, koi bhi thread pahle aayega

Locks / Mutex

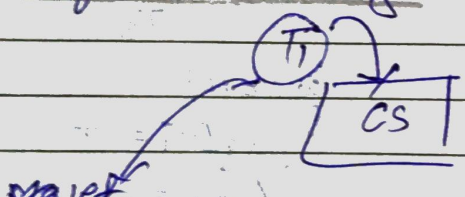
- Locks can be used to implement mutual exclusion & avoid race cond^n by allowing only one thread / process to access critical section.



→ **Disadvantages**

(a) **Contention:**

one thread has acquired the lock, other threads will be busy waiting, what if thread that has acquired the lock dies, then all other threads will be in infinite waiting



T_2 busy wait
T_3 CPU cycle khali hogi ek particular variable pe wait kate rhengi

T_1 thread kisi tarah dead hogya

$\Rightarrow$ T_2 or T_3 infinite wait pe chli jaengi

b/c kbhi bhi lock T_1 ne acquire kiya tha kbhi unlock ne nhi kiya.

Ib kbhi unlock ne nhi kiya th T_2 or T_3 infinite wait pe chli jaengi

(b) Deadlocks

(c) Debugging issue

(d) Starvation of high priority threads

LEC-17 | Conditional variable and Semaphore for thread Synchronization.

Conditional variable

(a) The cond^{n} variable is a synchronization that lets the thread wait until a certain cond^{n} occurs.

(b) works with a lock

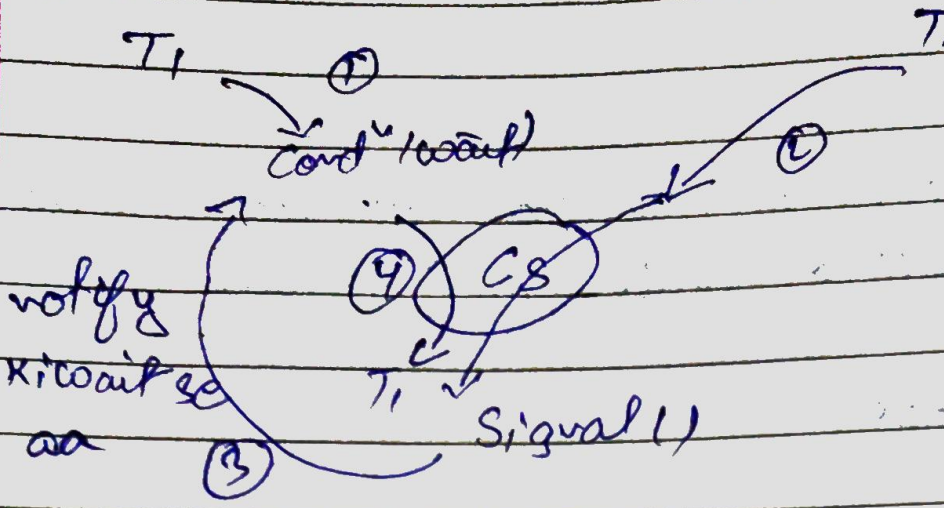
(c) Threads can enter a wait state only when it has acquired a lock. When a thread enters the wait state, it will release the lock & wait until another thread notifies that the event has occured. Once the waiting thread enters the running state, it again acquires the lock immediately & starts executing.

(d) Why to use conditional variable?
→ To avoid busy waiting

(e) Contention is not here.

(VVIP) Cond^{n} (lock)
- Waiting: wait karta rahunga or wait kate kate jo CPU cycle woo wai waste nhi karunga kyuki wai signal ka wait karunga woo signal jb aayga ek thread ayega jaise ke woo aayeg tb woo particular thread kaam karna suru karegi jb tk ke liye CPU kisi or useful job ko execute kr rha h.
- signal.

Ex -



Semaphores.

(a) Synchronization method.
(b) An integer that is equal to no. of resources.
(c) Multiple threads can go & execute C.S concurrently.

(d) Allows multiple program threads to access the finite instance of resources whereas mutex allows multiple threads to access a single shared resource one at a time.

~~Let B~~

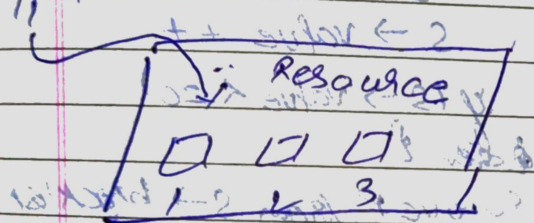
semaphores ⟶ integer

resource ⟶ single ✗

$T_1 \rightarrow T_{10}$

$\frac{?}{3}$ multiple instance

3 copies



let sem = 3 ⟶ variable

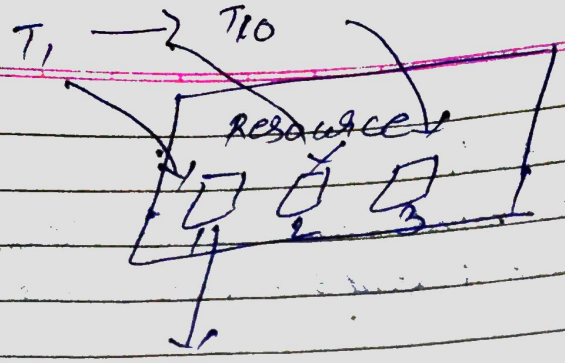
↓

< 0 ✗

① Sem = ~~3~~ ~~2~~ ~~1~~ 0

when T_1 enter, when T_2 enter, when T_3 enters

Jaise he '0' aaya, tb wait krne lg jata hu T_1 T_2 T_3 jb tk execute nhi hojati, jb tk resources ko free nhi kr deti tb tk mai koi or threads ko un resources ko lock nhi lagne dunga



T1 exist

scan = 3 2 1 0

when T1 exists

then T4 acquire T1 position then scan = 2 0

wait (S)
{
S → value --;
if (S → value < 0)
{
add to S → blocklist
block ();
}

Signal (S)
{
S → value ++;
if (S → value <= 0)
then {
remove P from S → blocklist
wake up (P)
}
}

let declare semaphore S(2);

Code
wait ()
&
signal

Matlab iski integer value jo andar 2 h yani ek baar wait & resource ke instance k liye usko assume karega.

1st

T_1 → wait() → S → value = 1
call ki then S ki value ko decrement kr degi

Is 1 < 0 ⇒ NO
A iss case mai ye resource particular mil jaega kyuki
mere paas '2' instance h aisa assume kr rha hu mai

Now,
T_2 → wait() → S → value = 0
decrement

Now, T_1, T_2 are Taken
T_3 → wait() → S → value = ~~1~~ -1
then,
jo particular thread h usko mai block kr de rha
hu

ab jitni bhi threads aaengi block ho jaengi

Now, T_1 & T_2 → After CS → signal()
~~T_4 ⇒ wait() → S~~ exist
↓
Ye btane ke liye ki
jo C.S mai maine lock
lga rkha tha ie block
kiya tha wo maine ab
free kr diya.

Jaise T_1 thread ne signal call kiya
signal call krne ke baad ... usne wake up ki
call di phir wapas se (T_3) thread aaegi
↑ or CPU mil jaegi
CPU

(e) Binary Semaphore: value can be 0 or 1.
— AKA, mutex locks.

(f) Counting Semaphore
- Can range over an unrestricted domain.
- Can be used to control access to a given resource consisting of a finite no. of instances.

(g) To overcome the need for busy waiting, we can modify the def of the wait() & signal() semaphore operations. When a process executes the wait() operation & finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.

The block-operation places a process into a waiting ~~the process can block~~ queue associated with the semaphore, & the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

(h) A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from waiting state to the ready state. The process is then placed in the ready queue.