AGENDA

1.) sync Problem

what / when

2.) solutions characteristicks

3.) Types of solutions

*) PROBLEM:

•) assume we have shared count variable

initially = 0

⇒ we do - 2 operations Parallely:

1.> Add 1 till 100

2.> Sub 1 till 100

∴

(T_1) Adding

(T_2) Subtracting

(simultaneously)

Finally → Print (count)

Expected answer:

DEMO

Observation → O/P is NOT 0 and random No. instead...

Q.) WHY ??

synchronisation Problem.

→ same data is shared b/w (2) threads...

both try to modify → causing issues.

Eg:

ADDER	SUBRACTOR
(1) X = Read (c)	(4) X = Read (c)
(2) X = X+1	(5) X = X-1
(3) X = update (c)	(6) X = update (c)

order of execution can be any thing....

*) CONDITIONS LEADING TO SYNC PROB:

1.) CRITICAL SECTION-

Part of code working on same data / var

eg:

	Adder (t_1)	Sub (t_2)
①	Print(hi)	Print(hello)
②	C++	C--
③	Print(bye)	Print(bye)

#2 → critical section

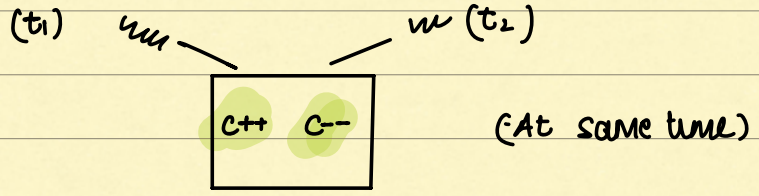
C.S. can be more than 1 line. ??

∴ Potential section of code where issues can happen.

↳ CRITICAL SECTION

2.) RACE CONDITION:

when more than ① thread enters critical section



solution:

if we can prevent more than (1) thread to enter into CS — Probably good (✓)

3.) **Preemption:**

if a thread (t_1) in **critical section** is preempted by OS, to do something else.....

COULD BE PROBLEM

(t_1)	(t_2)
Print(hi)	Print(hi)
x = Read(c)	x = Read(c)
x++	x--
count = x	count = x
Print(bye)	Print(bye)

-) t_1 starts → executes ① ② → Paused
-) t_2 starts → Completes fully
-) t_1 - Now when it Resumes - **value updated by (t_1) is NOT latest**

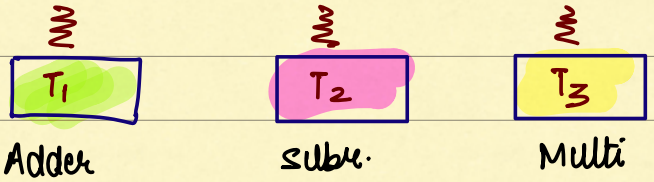
Solution: if thread has entered C.S. → No Preemption allowed...

*) PROPERTIES OF GOOD SOLUTION:

Goal: avoid above (3) Problems

1.) Offer Mutual Exclusion.

allows only (1) thread to enter C.S.



-) if adder is Allowed
-) subractor & Multi should WAIT

Prevents Race condition

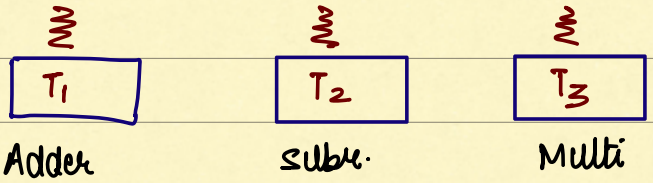
2.) Overall system should make Progress.

Never block All threads

3.) Bounded waiting:

No thread should wait only. should have some time limit

Eg:



case:

t_1 is allowed

t_2 arrives

t_3 arrives

↳

Order of execution → $t_1 \rightarrow t_2 \rightarrow t_3$.

∴ Threads will access C.S. in order of request

4.) No Busy wait:

Busy waiting - threads check continuously if they can enter C.S

solution → Notify threads when they can execute so that CPU is more productive

*] SOLUTIONS:

1.) MUTEX: (lock)

Mut - Mutual
Ex - exclusion

Nothing but a lock, that enables Mutual exclusion.

```
Adder (t1)
Print (hi)
x = Read(c)
x++
c = x
Print (bye)
```

```
Subs (t2)
Print (hi)
x = Read(c)
x--
c = x
Print (bye)
```

M.E says: Thread Must take a lock before entering into C.S.

AND →

release lock ASAP; it's out of C.S.

- Languages Provide lock objects
- it is develop. duty to ACQ and REL. locks.

*) LOCK OBJECTS:

1.) only (1) thread can unlock at a time

Others → wait

2.) Notifies other thread to execute; when lock is Released.

→ Mutual Exclusion ☑
Bounded waiting ☑
system Prog ☑
No busy wait ☑

#DEMO

2.> SYNC keyword:

(JAVA specific)

In Java, many objects (default) have implicit lock.

USAGE:

Just use 'synchronised' with Object

sync over any object is ~ taking lock.

Advantages

> ONLY 1 Object is Passed

Note: When you have (2) shared vars., always go with Mutex.

DEMO.

sync Methods:

assume: some complex piece of code

count 2

....

incrementValue();
getValue();

} Methods

∵ If you have access to Methods and Not variable...

→ doing same thing # DEMO

∴ declaring method of a class as sync

ONLY (1) thread can access

any sync Method on that Object

(T_1)	(T_2)	will execute parallelly
C_1.increament()	C_1.increament()	✗
C_1.increament()	C_1.decreament()	✗
C_1.increament()	C_1.getvalue()	✓
C_1.increament()	C_2.increament()	✓

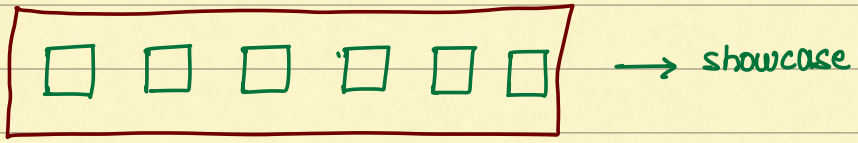
3.) SEMAPHORES:

Allows multiple threads in critical section

Eg: Producer Consumer Problem

Analogy:

store where we sell shirts



1.) Producers: (tailors)

2.) Consumers: customers

Now, usecase:

-> we only allow customer to enter when

we have shirt available

⇒ Also, we allow Producer to create shirt ONLY if space is available

At any instant t :

No. of Producers = Empty spaces
No. of Consumers = # filled spaces

similarly in code:

store = List<Object>

size_of_store = 6

Producer : Thread in system } # Add New Object
Consumer: Thread in system } # Remove Object

Now, Running 100 threads in Parallel of (P) and (C)

is that Problem ??

Yes...

1.) Consumer removing from Empty list
2.) Producer adding in full list

SOLUTION:

(i) (Naive) - add if() check ....

would that work ??

(ii) (lock) .....