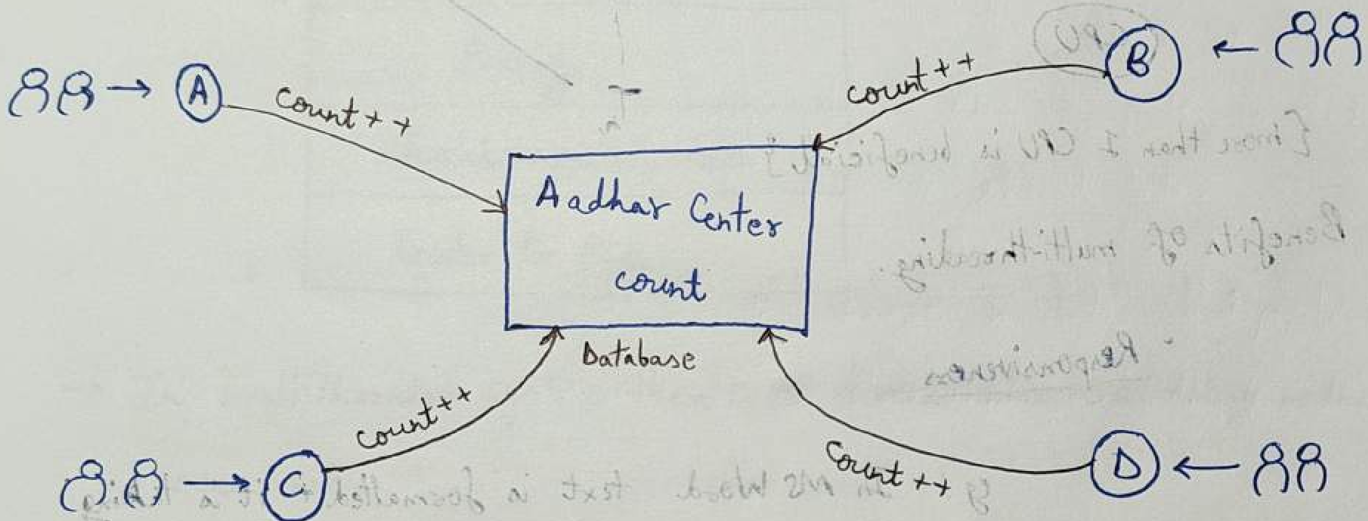


→ In above diagram Aadhar center is the central database who keeps record of people who have created aadhar card.

→ A, B, C, D are the centers where people go to create aadhar card. { Can consider A, B, C, D as threads too }

→ Count is incremented like this if we do count ++

temp = count + 1

count = temp.

→ Let's assume center A and B both simultaneously requested to increment count as someone created their aadhar card.

So if both request come at the same time when count = 11 then center A request is handled so

temp = count + 1

temp = 12

simultaneously request by center B is also handled

temp = count + 1

temp = 12

Then finally count is updated

count = temp

→ But as we can see instead of +2 we only did +1 due to simultaneous request and due to this we lost some data.

→ Therefore process synchronization is needed to handle this problem.

→ Process synchronization techniques play a key role in maintaining the consistency of shared data.

→ The aadhar center or the central database is the <u>critical section</u> and the condition that was arising due to which there was inconsistent data is called <u>race condition</u>.

Critical section (C.S)

→ The critical section refers to the segment of code where processes / threads access shared resources such as common variable, and files and perform write operations on them.

→ Since processes/threads execute concurrently any process can be interrupted mid-execution.

Race condition {Major thread scheduling issue}

→ A race condition occurs when two or more threads can access shared data and they try to change it at the same time.

→ Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data.

→ Therefore the result of the change in data is dependent on the thread scheduling algorithm i.e. both threads are "racing" to access/change the data.

eg :-

	P_1		P_2
I_1	int x = shared;	I_1	int y = shared;
I_2	x++;	I_2	y--;
I_3	sleep(1);	I_3	sleep(1);
I_4	Shared = x;	I_4	shared = y;

① P_1 comes into CPU and execute I_1 set $\boxed{x = 10}$

Now executes $I_2 \Rightarrow x{+}{+} \Rightarrow \boxed{x = 11}$

Now sleep(1) executes and P_1 lose the CPU.

(2) P_2 comes in CPU and executes I_1 set $\boxed{y = 10}$

Now executes $I_2 \Rightarrow y{-}{-} \Rightarrow \boxed{y = 9}$

Now P_2 executes sleep(1) and lose the CPU.

(3) P_1 resumes execution in CPU. and set $\boxed{shared = 11}$ and terminates after complete execution.

(4) P_2 resumes execution in CPU and set $\boxed{shared = 9}$ and terminates

→ Now in above example

if process execution starts from P_2 then $\boxed{shared = 11}$

if process execution starts from P_1 then $\boxed{shared = 9}$

⇓

But P_1 is incrementing the shared variable and P_2 is decrementing the shared variable so it's value should have been as $10 + 1 - 1 \Rightarrow 10$

→ We arrived at this incorrect state because we allowed both processes to manipulate the shared variable concurrently.

→ A situation like this when several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called a race condition.

Solution to Race Condition {or to critical section problem}

(a) Atomic operations

→ Make critical section code an atomic operation i.e. executed in one CPU cycle.

like in count++

⇓

temp = count + 1
count = temp

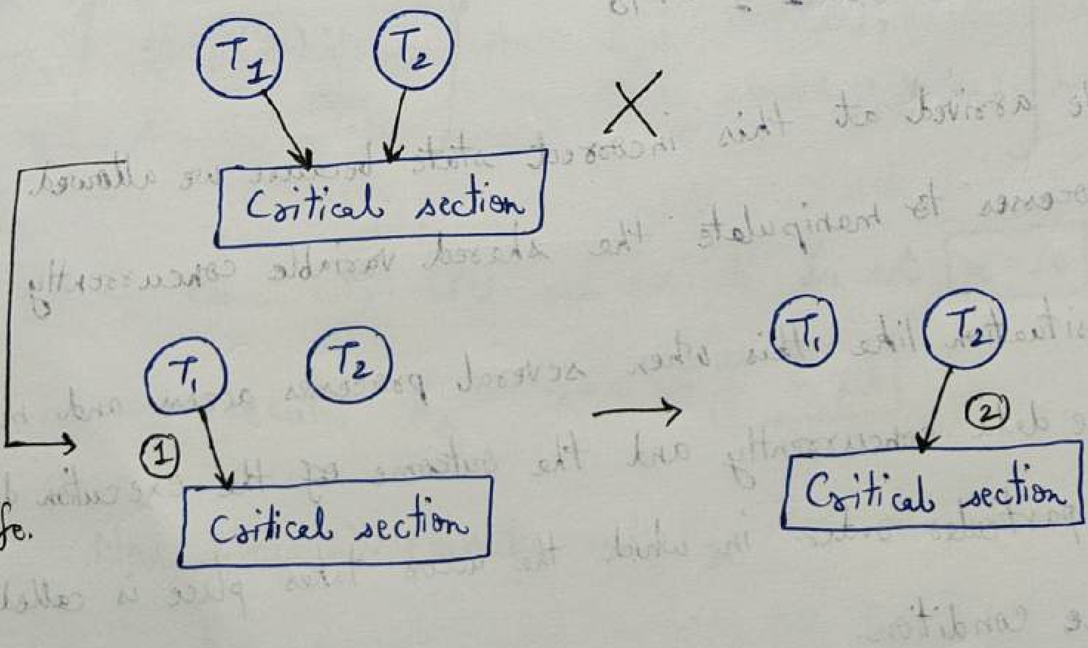
} → 2 CPU cycles so need to convert it to 1 CPU cycle

In cpp can create by atomic <int> → thread safe.

→ It mainly means that once a process has completed manipulation then only it will context switch (process or thread main task is manipulation)

(b) Mutual exclusion

converted parallel processing to sequential to make it thread safe.



→ Can use locks to bring mutual exclusion

eg :- in cpp mutex

due to this T_1 locks the critical section so that no other thread can enter then once the manipulation is complete releases the lock and then T_2 locks the critical section.

(C) Semaphores

→ A semaphore is a synchronization tool used in OS to manage concurrent access to shared resources.

→ A semaphore is a variable used to signal and control access so that multiple processes or threads don't enter critical sections at the same time.

Solution of Critical section should have 3 conditions

① Mutual exclusion

→ Only one thread/process at a time can access critical section.

{Prevents race conditions}

② Progress

→ Each thread/process should have fair chance to go in critical section.

→ There shouldn't be any fixed order like first P_1 process will go then only P_2 process will go.

→ If no process is in the critical section and some processes want to enter it one of them must be allowed to proceed without unnecessary delay.

{Prevents indefinite blocking}

③ <u>Bounded waiting</u> (not that main can or cannot be fulfilled but ① & ② should be fulfilled)

→ There must be a limit on how long a process/thread waits to enter its critical section after requesting it.

{Prevents starvation}

Can we solve problem of race condition by using single flag variable?

T_1

```
while (1)
{
    while (turn != 0);
        Critical section
        turn = 1
        remainder section
}
```

T_2

```
while (1)
{
    while (turn != 1);
    Critical section
    turn = 0
    remainder section
```

→ Here we have taken our flag variable as turn.

If turn = 0 then

T_1 condition of $0 != 0$ is false and it enters critical section

While T_2 condition of $0 != 1$ is true so it just executes while.

→ After T_1 work is done with critical section it leaves the critical section and makes turn = 1 due to which the condition of while for T_2 is false as $1 != 1$ is false

Then T_2 enters critical section and when it is done it makes the turn = 0. Till the time T_1 just executes remaining section.

→ Then T_2 executes their remainder section.

→ Hence we have achieved mutual exclusion.

→ But there is a fixed priority or order as if turn = 0 then T_1 will be executed first then T_2, if turn = 1 then T_2 will be executed first then T_1. (as if T_1 doesn't want to go in critical section when turn = 0 then T_2 will be waiting until T_1 goes)

→ Hence it doesn't fulfill progress condition therefore we cannot solve race condition by using single flag variable.

→ So this single flag method was improved and is called Peterson's solution

Single flag method $\longrightarrow$ improvement $\longrightarrow$ Peterson's solution

Peterson's solution

flag[2] $\rightarrow$ indicate if a thread is ready to enter the critical section, flag[i] = true implies that P_i is ready.

turn $\rightarrow$ indicates whose turn is to enter the critical section.

T_1

```
while(1)
{
    flag[0] = T
    turn = 1
    while ( turn == 1 && flag [1] == T);
    Critical section
    flag[0] = F
}
```

T_2

```
while(1)
{
    flag[1] = T
    turn = 0
    while(turn == 0 && flag[0] == T);
    Critical section
    flag[1] = F
}
```

→ In the above code if we execute T_1 first then

$flag[0] = T$ & $turn = 1$

then if it context switches to T_2 then executes it's first line then

$flag[1] = T$ then it again context switches to T_1.

→ When it resumes T_1 execution, the condition for while is true so it loops then when it context switches back to T_2 and executes it $turn = 0$ and when it checks while condition both are true so it loops around.

→ Now when it is context switched to T_1 back the condition for while is false as $turn != 1$ so it enters critical section and when the work is completed it makes $flag[0] = false$.

→ Due to this while condition of T_2 is also false and it enters critical section.

→ As there is no specific order and mutual exclusion is there so it can be used as a solution for race condition.

→ In other case if T_2 is not ready to enter critical section so whole T_1 will be executed first as $flag[1] != true$ hence when T_1 is fully executed it makes $flag[0] = false$

→ Initially the values are false in flag array.

In above example used 2 shared variables

(i) flag array (flag[0], flag[1])

(ii) turn

→ Peterson's solution can be used to avoid race condition but holds good for only 2 process/threads.

Extra

(Similar to Peterson's Solution)

Dekker's Algorithm

initially flag[0] = False
flag[1] = False
turn = random (0/1)

P_0

```
while (1)
{
  flag[0] = true;
  while( flag [1] == true)
  {
     if (turn == 1)
     {
        flag[0] = false ;
        while (turn == 1);
        flag[0] = true ;
     }
  }
  critical section
  turn = 1
  flag[0] = false
  remainder section }
```

P_1

```
while (1)
{
  flag [1] = true
  while (flag[0] == true)
  {
     if (turn == 0)
     {
        flag [1] = false;
        while (turn == 0);
        flag [1] = true
     }
  }
  critical section
  turn = 0
  flag [1] = false
  remainder section
}
```

→ In Peterson's ~~alg~~ solution, the two processes seem to be dominent. A process seems to force his way into the critical section unless it's the other one's turn.

→ In Dekker's algorithm, the two processes seem to be submissive and polite. If both processes want to enter the critical section and it's the other one's turn, the process decides to no longer want to enter.

→ Difference

Mutex/Locks

→ Locks can be used to implement mutual exclusion and avoid race condition by allowing only one thread/process to access critical section

· Disadvantages

(i) Contention

→ One thread has acquired the lock, other threads will be busy waiting.

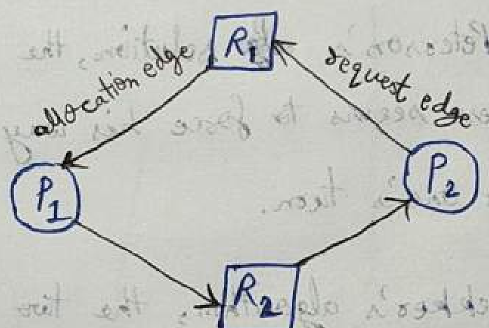
↳ In this threads just consumes or eat CPU cycle and just wait for access to critical section.

→ What if thread that had acquired the lock dies then all other threads will be in infinite waiting.

(ii) Deadlocks

→ A deadlock is a situation where two or more processes are blocked forever, each waiting for a resource that the other is holding.

eg :-



- Process P_1 holds resource R_1 and waits for resource R_2.
- Process P_2 holds resource R_2 and waits for resource R_1.

(iii) Debugging

→ Debugging in multi-threaded systems with locks is difficult because thread execution is unpredictable.

(iv) Starvation of high priority threads

→ If a low priority thread acquires the lock, then high priority thread comes it cannot execute leading to starvation.