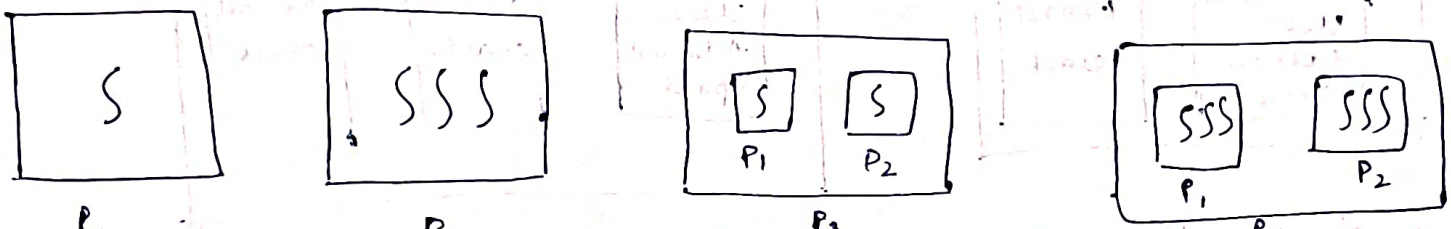
17/10/23

Threads

- Process → Program in execution
- Threads → Process is divided into multiple / single thread
  → basic unit of CPU utilization



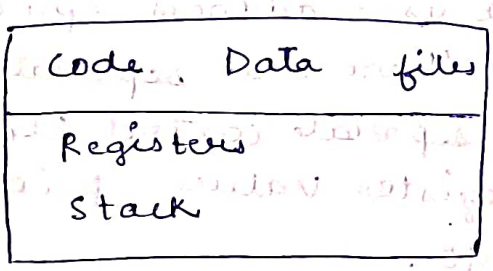
P_1 single process single thread

P_2 single process multiple thread

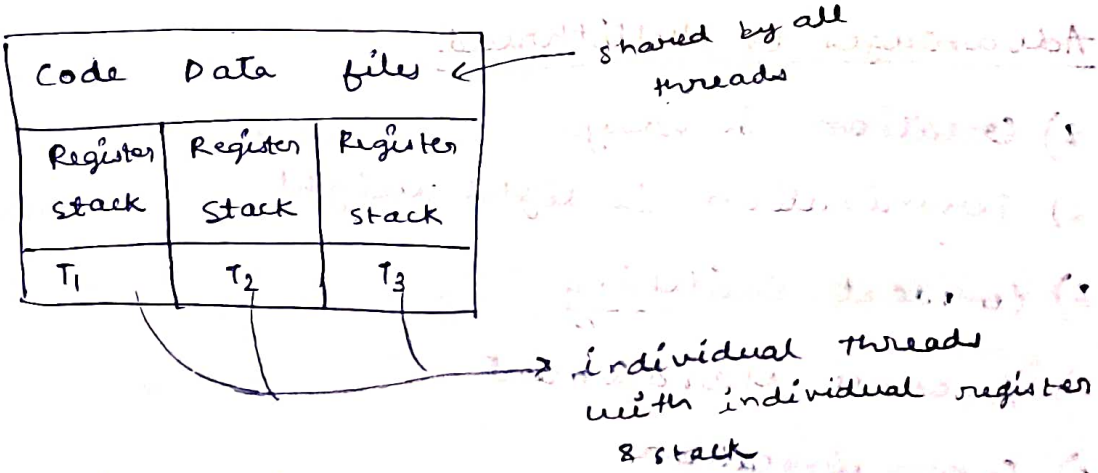
P_3 multiple process single thread

P_4 multiple process multiple thread.

Process :



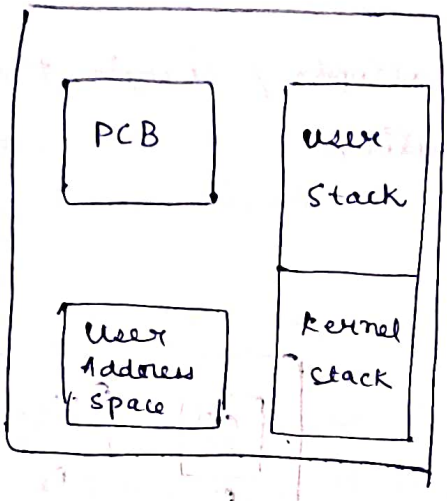
Threads (multiple threads) :



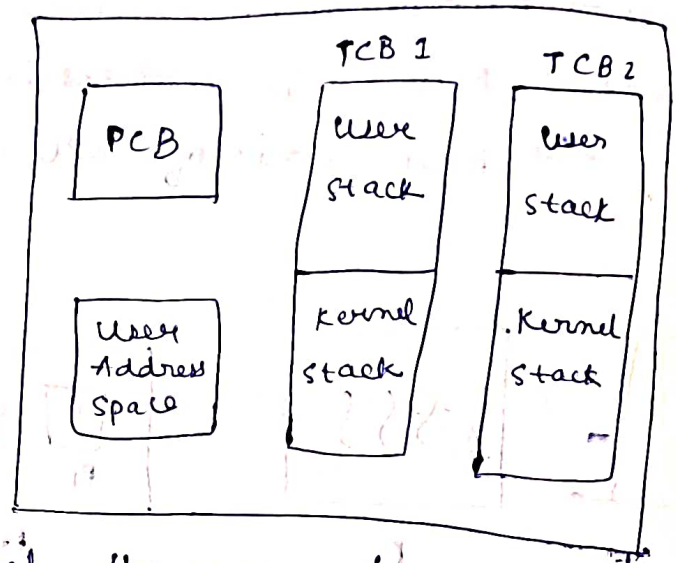
A thread is the basic unit of CPU utilization. We can have single / multiple threads in a system. Multithreading refers to the ability of OS to support multiple concurrent paths of execution within a single process. In a multithreaded environment, the thread shared code, data & files of the process & maintain their own registers & stacks

TCB = thread control block

Single thread



Multi-thread



In a single threaded model, the process includes, PCB & user address space as well as user & kernel stack to manage the behaviour of execution of processes. In a multithreaded environment, there is still a single process control flow & a user address space associated with the process but now there are separate stacks for each thread as well as separate control block for each thread that contains register values, priority & other thread related state info.

Advantages of Multithread

1) Creation is easy
2) Termination is light weight
3) Context Switching
4) Resource Management
5) Communication

1. It takes less time to create a new thread in a existing process rather than creating a brand new process
2. Less time to terminate a thread than a process
3. Less time to switch b/w 2 threads, than to switch b/w processes
4. Resource management is better
5. Communication in most OS, communication b/w independent processes requires the intervention of the kernel to provide protection & the mechanisms needed for communication. As threads in a process, share memory & files, they can communication without invoking the kernel.

States of thread

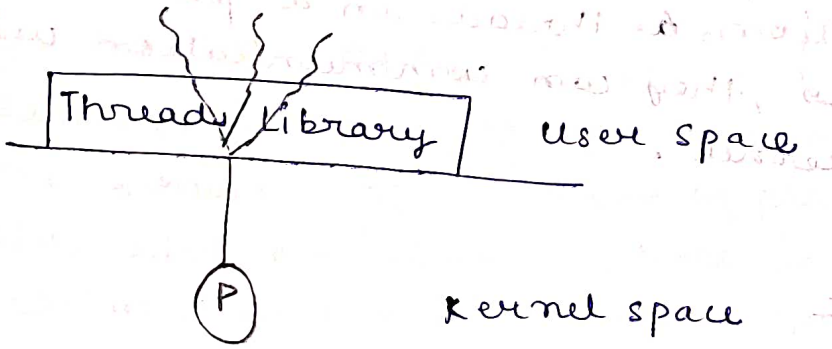
1) Ready, Running
2) Blocked
3) Unblocked
4) Finish
5) Spawned.

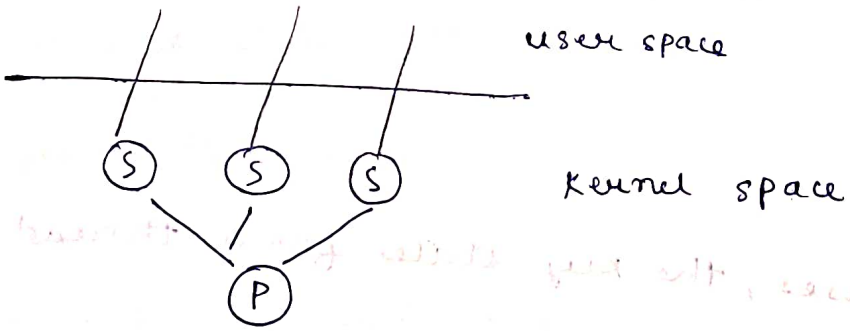
1. As like processes, the key states for a thread are running, ready
2. When a threads needs to wait for an event it will block. And the processor will run the next ready thread.
3. When the event for which a thread is blocked occurs, the thread is unblocked & sent to the ready state.
4. When a thread completes, the register & stacks are deallocated.
5. A thread within a process, may spawn another thread

providing an inst^{n} pointer, arguments for the new thread.

Types of Thread

1) User - level thread → created by User
   → Not recognized by OS
2) Kernel level thread → created by OS
   → Recognized by OS

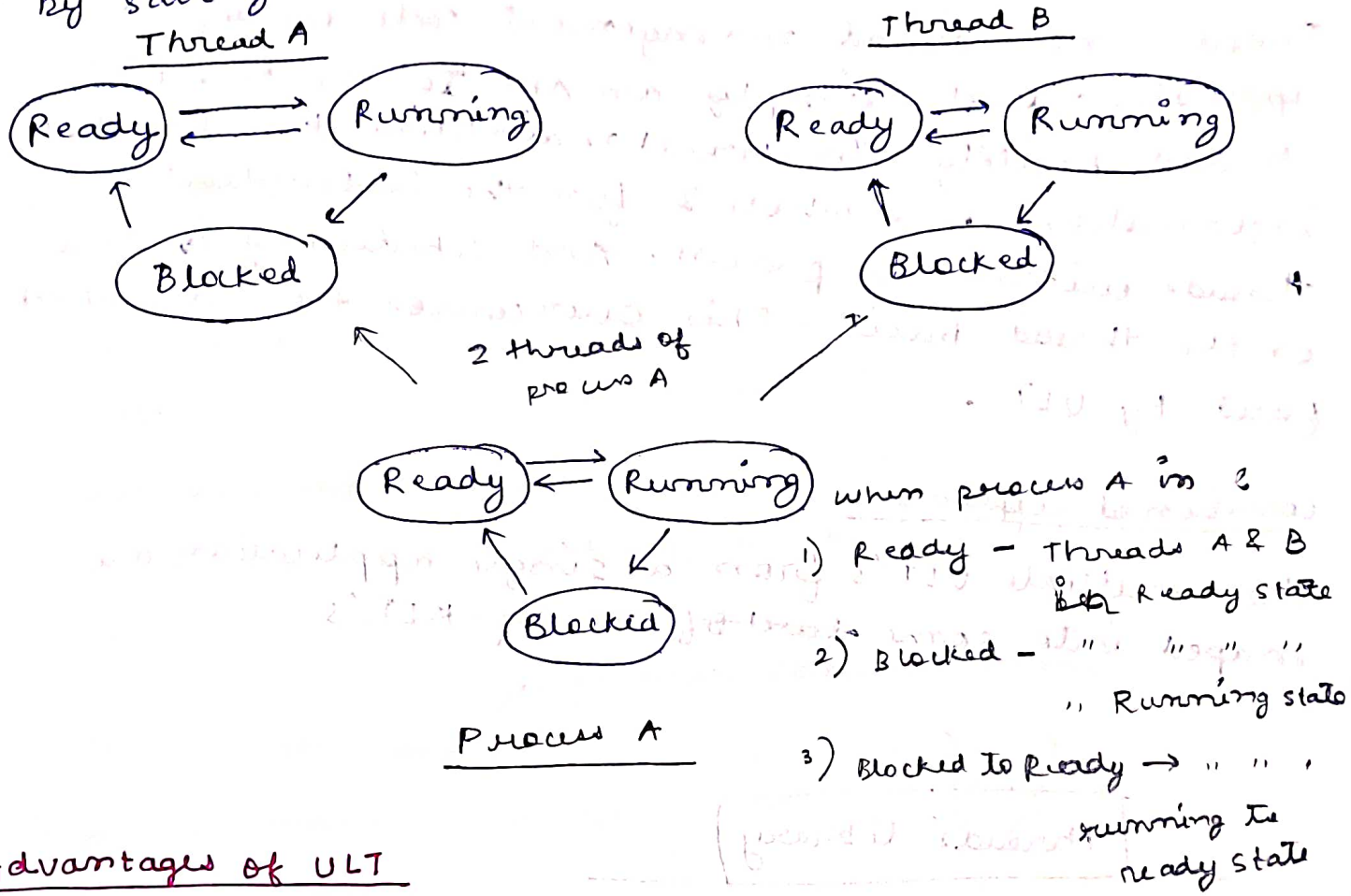




1. User - level thread

In user level thread, all the thread management work is done by the application & kernel is not aware of the existence of threads. Any application can be programmed to be multithreaded using a threads library which is a package of routines for ULT management. The thread library contains the code for creating & destroying threads. For passing msgs

& data b/w threads. For scheduling thread execution
& by saving & restoring thread context.



When process A is :

1) Ready - Threads A & B in Ready state
2) Blocked - " " " " Running state
3) Blocked to Ready → " " , running to ready state

Advantages of ULT

1. Scheduling can be application specific
2. Thread switching does not require kernel mode priviledges, bcoz all the thread management are within the user space of a single process
3. ULT can run on any process, no changes are made to the kernel

Disadvantages of ULT

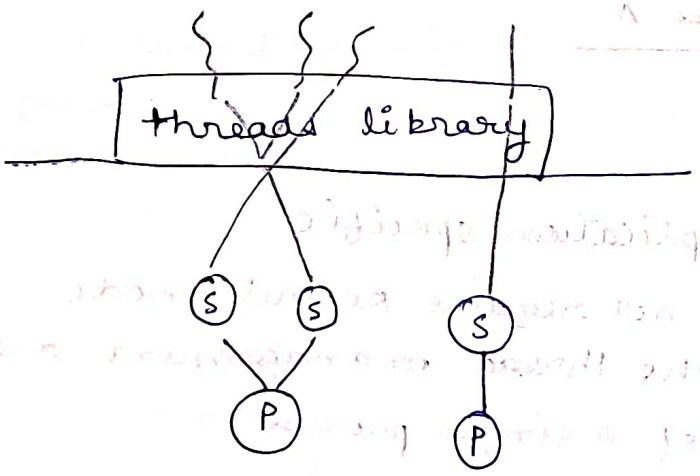
1. Many system calls are blocking in a typical process so blocking a process would block all its threads.
2. In a multithreaded application, it cannot take advantage of multiprocessing, as a kernel assigns 1 process to a processor at a time.

2. Kernel level threads

There is no thread management code in the application level, simply an API to the kernel thread facility. The kernel maintains kernel context information as a whole & for the individual threads within the process. And scheduling is done on the thread basis. This overcomes the 2 drawback faced by ULT.

Combined approach

The multiple ULT's from a single application are maped into some ~~fixed~~ no. of KLT's



18/10/23 **4 types of relationship holds for the processes**

1. 1:1 relationship (One thread one process)

Each thread of execution is a unique process with its own add. space & resources

2. M:1 relationship (Many thread one process)

A process defines an address space & dynamic resource ownership. Multiple threads may be created & executed within that process

3. 1:M relationship (one - to many)

A thread may migrate from a process environment to another. This allows a thread to be easily moved among distinct systems.

For eg - There is a main program that makes use of a i/o subprogram. The main program can spawn new process to handle the i/o, so there are 3 ways to implement this :

→ The entire program can be implemented as a single process but we have memory constraints here.

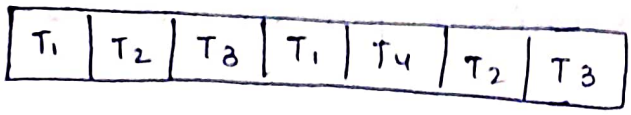
→ The main program & i/o subprogram can be implemented as 2 separate processes but if there is an overhead of creating subordinate process & context switching.

→ So the solution for these problems is, we treat the main program & i/o subprogram at a single activity & implement it as a single thread. One address space can be created for main program & one for the i/o subprogram, so that the threads can be moved b/w 2 address space as execution proceeds.

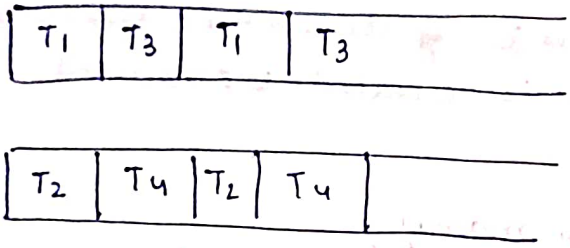
4. M:N relationship (Many threads & many processes

It combines the attributes of many threads & many processes.

Multicore & Multithreading



: Single core



: Multicore

Challenges :

1. Data splitting (which thread To which core)
2. Balance (each thread given equal weightage work)
3. Data Dependency (O/P of 1 is input To another & process is sequential)
4. Testing & Concurrency

Multithreading programming provides mechanism for more efficient use of multicores. On a single core system, the execution of threads will be interleaved over time.

On the system with multicores, concurrency means the thread can run ||^{el} however this proposess few challenges :

→ Dividing activities : The application must be examined To find areas that can be divided into separate concurrent tasks

→ Balance : It must be ensured that the tasks perform equal work of equal value

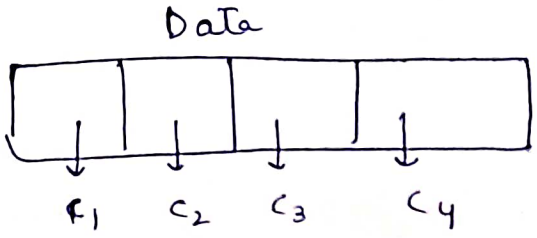
→ Data splitting : As the applications are divided into tasks, the data accessed & manipulated must be divided To run on separate cores.

→ Data dependency : The execution of tasks must be synchronized to accomodate data dependency

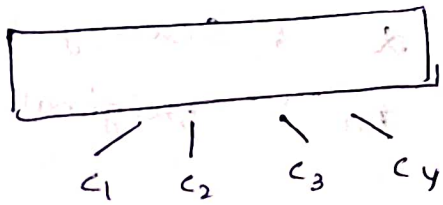
→ Testing, debugging & dependency : When a program is running parallely on multicores, there are different execution paths. Testing & debugging such programs is difficult.

(4)

(1) Data level parallelism



(2) Task level parallelism



There are 2 ways to parallelise the workload of a process — (1) Data level parallelism / Fine-grained threading
(2) Coarse grained threading / Task level parallelism

(1) Data level parallelism

It focuses on distributing subsets of same data across multiple computing course & performing the same operation on each core. It is also known as fine-grained threading

(2) Task level parallelism

It involves distributing not data but tasks across multiple computing ~~core~~ cores. An eg would be different threads performing a unique operation on the same array of elements. AKa ~~to~~ coarse grained threading

In SJF

To predict time (we can use the below methods) of next process :

(1) Simple Avg (of before processes)

(2) Exponential Avg.

$$\tau_{(n+1)} = \alpha t_n + (1-\alpha)\tau_n$$

where $\tau_{(n+1)}$: predicted time of next process

α : Scaling factor

t_n : original BT

Q① $\alpha = 0.5$; $\tau_1 = 10$

P_1	P_2	P_3	P_4
6	4	6	4

$\tau(5) = ?$

$$\tau(5) = \alpha t_4 + (1-\alpha)\tau_4 \longrightarrow (i)$$

$$\tau_2 = \frac{1}{2} \times 6 + \frac{1}{2} \times 10 = 8$$

$$\tau_3 = \frac{1}{2} \times 4 + \frac{1}{2} \times 8 = 6$$

$$\tau_4 = \frac{1}{2} \times 6 + \frac{1}{2} \times 6 = 6$$

$$\tau_5 = \frac{1}{2} \times 6 + \frac{1}{2} \times 6 = 2 + 3 = 5$$

②

P_1	P_2	P_3	P_4
5	2	6	3

$\tau(5) = ?$

$\alpha = 0.5$

$\tau(1) = 5$

Solve using simple averaging & exponential averaging

Solⁿ (1) **Simple Avg.**

$$\tau(5) = \frac{5+2+6+3}{4} = \frac{16}{4} = 4$$

(2) **Exponential Avg.**

$$\tau(2) = \frac{1}{2} \times 5 + \frac{1}{2} \times 5 = 5$$

$$\tau(3) = \frac{1}{2} \times 2 + \frac{1}{2} \times 5 = \frac{7}{2} = 3.5$$

$$\tau(4) = \frac{1}{2} \times 6 + \frac{1}{2} \times 3.5 = \frac{9.5}{2} = 4.75$$

$$\tau(5) = \frac{1}{2} \times 3 + \frac{1}{2} \times 4.75 = \frac{7.75}{2} = 3.875$$

25/10/23

Process Synchronization

(1) Serial → one after the another
(2) Parallely → Running at the same time } Process running

Eg - P1 P2 shared = 5

```
{ int shared x ;        { int.
  x++ ;                  ------> same.
  sleep(1)
  w(x)m }
```

B

if parallely runs then O/P = 6 should be 7 but output

Race cond

A race condⁿ occurs when multiple processes/threads read & write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Q ① P1 ( ) P2 ( ) Initially B = 2

```
P1 ()                    P2 ()
{                        {
(1) C = B - 1            (3) D = 2 * B
(2) B = 2 * C            (4) B = D - 1
}
```

Ans Case 1 : {1 2 3 4}

C = ~~1~~ 1, B = ~~2~~ 3, D = 4

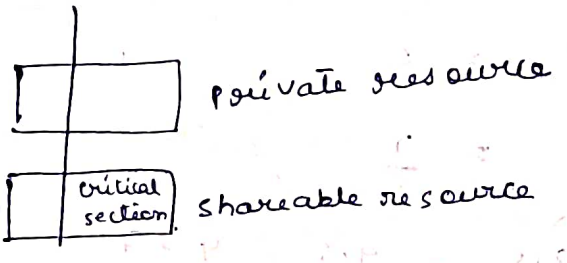
Case 2 : {3 4 1 2}

D = 4, B = ~~3~~ 4, C = 2

Case 3 : {1 3 4 2} C = 1, D = 4, B = ~~3~~ 2

Case 4 : {3 1 2 4} D = 4, C = 2, B = ~~4~~ 3,

** Process P



Critical Section

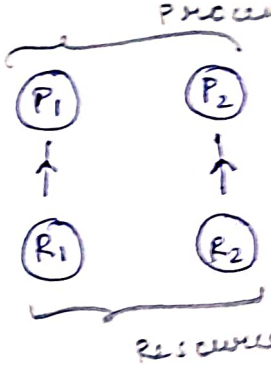
Shared resources are excessed by different processes at a time. The space where or the region where the process access this shareable resource, that region is known as critical section.

Concerns of OS

1. The OS should keep track of different procures which is done with the help of PCB.
2. The Os must allocate & deallocate various resources for each active process.
   Resources can be - Memory, i/o devices, files & processor time.
3. The Os must protect the data & physical resources of each process against unintended interference by other processes.
4. The functioning of a process must be independent of the speed at which its execution is carried out relative to the speed of another concurrent process.

Process Interaction

1. Process unaware of each other — Race cond^n
2. Process indirectly aware of each other — cooperation by sharing
3. Process directly aware of each other — cooperation by communication



Deadlock cond^n

P_1 wants R_2
P_2 " R_1

P_1 & P_2 both want resources R_1 & R_2. R_1 is given to P_1 & R_2 to P_2. But now both requires the other resource to complete, neither leave each other resources so that 1 can complete. Therefore deadlock cond^n arises.

Criteria's to enter Critical section

Mandatory
1) Mutual Exclusion – 1 resource to one process at a time
2) Progress – only wanted processes shall enter critical section
3) Bounded wait
4) Architecture Neutral – work on all systems

Optional

The criteria to access critical section is :

(1) **Mutual Exclusion** –

- Only 1 process at a time is allowed in critical section. Among all processes that have critical sections for the same resource / shared object.
- A process that halts in non-critical section must do so, without interfering with other processes.
- It must not be possible for a process requiring access to critical section to be delayed indefinitely
- When no process is in critical section, any process that request entry to its critical section must be permitted to enter without delay

(2) Progress

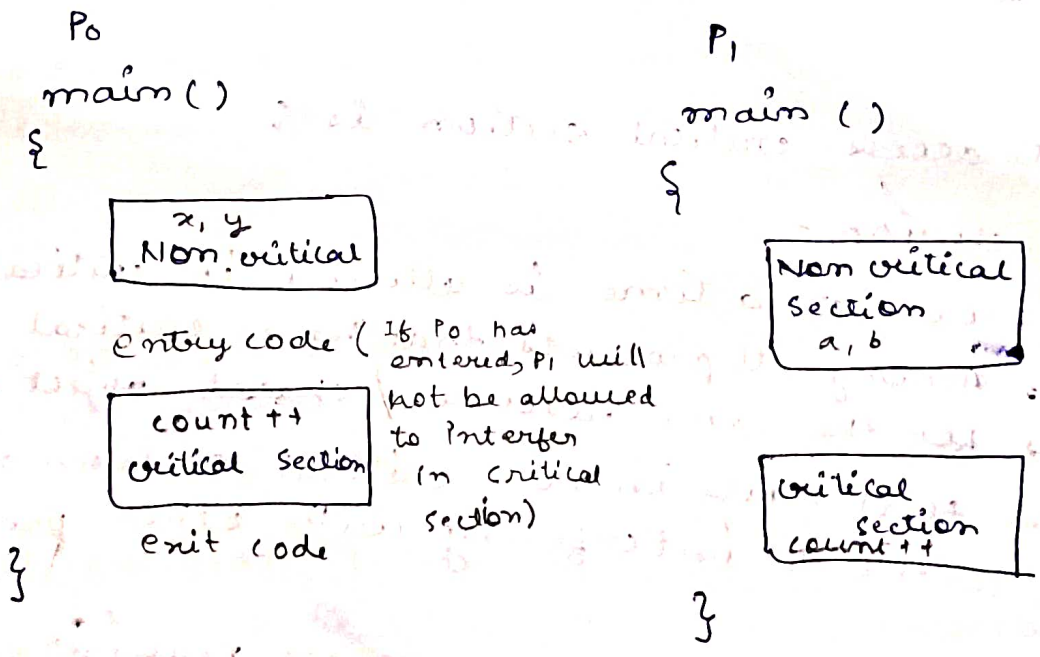
Only those process should be allowed To enter those who willingly want to enter critical section. NO assumption are To be made about relative process, speeds / no. of processes.

(3) Bounded wait

It is the max. time a process can wait A process can remain inside the critical section for a finite time only.

Mutual~~a~~ exclusion & progress are mandatory criterials while bounded wait & architecture Neutral are optional criterias.

27/10/23 ~~Hardware Solutions~~.



- Independent variables need not be added in the critical section. If any program wants to enter critical section, it will not access directly, it will execute some entry section

code. At one time only 1 process can execute entry code & entry critical section. Another process will be stopped at entry indicating mutual exclusion.

1. Lock variable

LOCK variable — Initially LOCK = 0

```
{
    while (LOCK == 1) ;   ] entry
    LOCK = 1
    CS (critical section)
    LOCK = 0              ] exit
}
```

It fails Mutual exclusion.

2. Turn variable

Turn Variable

```
{
    while (turn != 0) ;
    CS
    turn = 1.
}
```

P_0

Turn variable

```
{
    while (turn != 1);
    CS
    turn = 0
}
```

P_1

Progress don't hold.

This solution guarantees mutual exclusion but has few drawbacks -

1) It does not follow progress as processes strictly alternate in their use of critical section
2) The so choice of process has not been considered whether it wants to enter critical section or not.

Therefore P_0 can't enter even though the CS is empty.

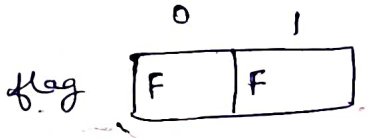
3. Flag variable

P_0

```
{
    flag(0) = T
    while ( flag(1) == T);
    CS
    flag(0) = F
}
```

P_1

```
{
    flag(1) = T
    while ( flag(0) == T);
    CS
    flag(1) = F
}
```



→ Here we have no progress

- This sol^n guarantees mutual exclusion but creates another problem if both processes set their flags to true before either has executed the while statement then each process will think that the other has entered the CS causing deadlock.

4. Using both variable (Peterson's solution) only for 2 process

P_0

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

P_1

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

→ Satisfies both mutual E & progress

- The mutual exclusion is preserved in this solution. This solves the problem of progress. ~~This~~ This also follows ~~progress as~~ problem of progress as:
  1) If P_1 has no interest in CS, as it will set
     flag [1] = F
  2) P_1 can use CS repetedly, this case is not possible as P_1 gives opportunity to P_0 by setting the turn to 0 before it enter CS. Hence we can say peterson solution satisfies our mandatory criteria i.e. Mutual exclusion & progress.

31|10|23 **Hardware solutions**

```
{
    while (Lock == 1);
        Lock = 1;
        cs;
        Lock = 0;
}
```

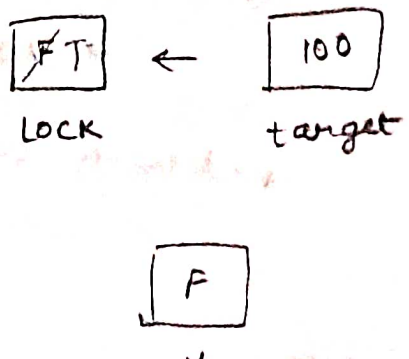
(1) Test & ~~Set~~ Set
Lock = false.

```
while (Test & Set (& Lock));
CS
Lock = false.
```

<- - - - - - - - - ->

Function → boolean Test & set (
boolean * target)
{
boolean rv = *target
*target = true
return rv;
}

Among h/w solutions, one of the way to ensure mutual exclusion is disabling interrupts but this sol^n is not feasible in a multiprocessor environment as disabling interrupts can be time consuming since the message is passed to all the processes.



- Many modern computer system therefore provide many special h/w instructions that allow us to either test / modify the content of a word / to swap the contents of 2 words atomically.

(1) Test & Set

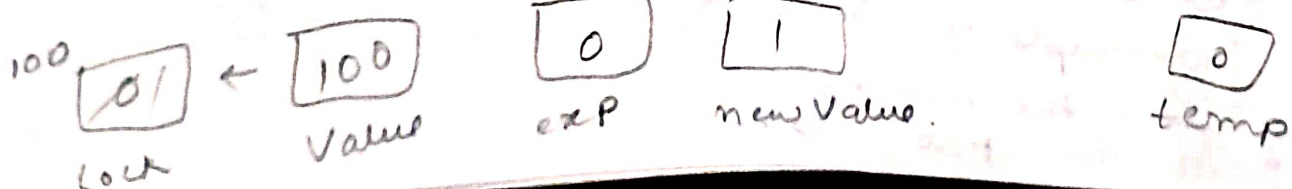
- In this sol^n, we combine line 1 & 2 from Lock variable sol^n, so that no prempt^n can occur & mutual exclusion is ensured. This sol^n satisfies mutual exclusion & progress but fails on bounded wait.
- It produces race cond^n

(2) Compare & Swap

```
{
while ( compare & swap ( & lock, 0, 1 ) != 0) ;
                          Lock = 0
  CS ;
  Lock = 0 ;
}
```

```
int compare & swap ( int *value, int expected,
                     int newValue )
{
    int temp = *value ;
                expected
    if (temp == 0)
                  newValue
        *value = 1 ;

    return temp
}
```



This solution satisfies mutual exclusion property as we take a shared variable i.e. initialized to zero. The only process that can enter in the CS is the one that finds lock = 0. All other processes attempting to enter CS go into a busy waiting mode. (↳ stuck in loop)

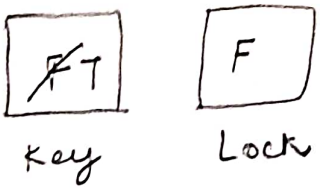
When a process leaves its CS it resets lock to 0 & at this point one of the waiting processes is granted access to CS. This process also satisfies progress but fails on bounded wait.

(3) Exchange

Atomic function → no premption in b/w

```
swap (boolean * a,
      boolean * b)
{
    temp = * a ;
    * a = * b ;
    * b = temp ;
}
```

```
do
{
    key = True
    [while (key == true){}
    swap (& lock, & key) ;]
    CS
    Lock = false
}
```



* Not satisfies bounded wait bcoz no mechanism is there for 1 process to restrict from entering again & again

In this sol^n mutual exclusion is satisfied as a shared variable "Lock" is initiazed with 0. It excludes all other process from CS by setting Lock to 1. Here we have used the swap f^n which exchanges the value of key & Lock when any process is in CS. Progress is also satisfied but this sol^n also fails on bounded wait.

Disadv. of above sol^{n}(s)

1) Bounded wait ↑

2) Starvation

3) Deadlock

Demerits of hardware solutions

1. Busy waiting is employed
   while a process is waiting for access to a CS, it continues to consume processor time

2. Starvation is possible
   When a process leaves a CS & more than 1 processes is waiting the 2nd process could indefinitely could be denied access

3. Deadlock
   Suppose P_1 executes a special inst^{n}(s) & enters the CS. P_1 is then interrupted by P_2 which is a higher priority process. If P_2 now attempts to use the CS, it will be denied access b coz of mutual exclusion.