Threads:-

Light weight process ??
Smallest unit/part of a process ?? -- part of an application

multi threaded applications
every process runs as single thread initially

Always applicable:-
    ==> Resource sharing
    ==> Concurrent execution

Physical concurrency -- on multiple CPUs in same time slot
Logical concurrency   -- time multiplexing on CPU

Mutithreaded examples, concurrency:-
    Office suite
    Browser
    Media player

    Client -- Server
    Parallel computing, eg:- parallel sum of large array

Multithreaded apps can have better CPU utilization and efficiency

for cpu centric apps (no blocking) threading is only
beneficial on SMP only
for interactive apps(disk access,n/w access,user input)
mutithreading is beneficail even on a single CPU

Concurrency in client-server model

thread creation is faster than child process creation

shell can't be multithreaded as each unit requires independent
address space for every command

Types of threads, mapping models:-

A.user level threads -- many to one mapping

B.kernel supported threads -- one to one mapping

User level threads:-

hidden from kernel (or) kernel don't have thread support
(-) blocking prob, if one thread of app make blocking
call, other threads of same app also block
(+) user level management is faster with the help of
thread library or thread manager
In this model -- threads can be considered as part of process
Many to one mapping model
eg:- traditional UNIX

Kernel supported threads:-

visible to kernel, scheduled managed at kernel level
since they are similar to process in execution point of view
but works on resource sharing -- light weight process(LWP)
(+) no blocking prob, effective CPU utilization
(-) slight overhead on kernel for thread management

execution point of view,i.e. scheduler view all are threads
and in resource manager(eg:- mem, fs, io etc) view
thread groups are represented as processes

pool of threads running on same address space

But every thread maintains their own private stack

POSIX Threads
pthread library -- source level portability across multiple
platforms
Native coding -- Native threads

header file:- pthread.h

APIs:-

- pthread_create
- pthread_join
- pthread_exit

- pthread_self
- pthread_equal

pthread_create:-

1st param:- address of pthread_t var, thread handle

2nd param:- address of attributes variable, NULL means default behaviour

3rd param:- address(name) of entry function

4th param:- input to thread entry function

```
pthread_t pt1;
pthread_create(&pt1, NULL, entry_fun, NULL);
```

pthread_exit ==> only current thread terminates
==> shared resources are not released

exit ==> resources are released
==> all threads will be terminated

pthread_join:-

block the current thread till the completion of target thread

```
pthread_join(pt1,NULL);
```

1st param -- pthread_t variable of target thread

2nd param -- to collect exit status of target thread

```
ps -eL -o pid,ppid,lwp,nlwp,stat,cmd

pthread_create(&pt1,NULL,tentry_fun,"A1");
pthread_create(&pt2,NULL,tentry_fun,"B2");

void* tentry_fun(void* pv)
{
        char* ps=pv;
        //.......
}

int id1=101, id2=102;
pthread_create(&pt1,NULL,tentry_fun,&id1);
pthread_create(&pt2,NULL,tentry_fun,&id2);

void* entry_fun(void* pv)
{
        int* ptr=pv;
        int id=*ptr;      // * ((int*)pv)
        //......
}
```

```
pthread_t ptarr[i];
for(i=0;i<n;i++)
{
        k=100+i;
        pthread_create(&ptarr[i],NULL,tentry_fun,(void*)k);
}
for(i=0;i<n;i++)
      pthread_join(ptarr[i]);


void* tentry_fun(void* pv)
{
      int i,id=(int)pv;
      //......
}
```

Thread safe code - reentrant functions:-
----

a function can be executed safely by multiple
threads without any inconsistency or data corruption
        ==> thread safe function/reentrant code

considerations:-
==> if sharing is not the objective convert global vars to
        local variables
==> for large objects allocate on heap and keep local pointer
==> if sharing is necessary apply mutual exclusion
        by locking techniques
eg:-
        strtok, strtok_r
        rand, rand_r

Thread pool:-

create some fixed no.of threads in advanced and
keep them blocked
when request comes, select a thread from pool
unblock it for service
when thread services not required block it again

unblocking-blocking threads is faster than
creation-termination

creating threads in advance rather than on demand

Symmetric Multi Threading(SMT):-

eg:- intel hyperthreading

	no.of CPUs	no.of threads	example
Model A	2	4	core i3
Model B	4	4	core i5,quad core
Model C	4	8	core i7
Model 0	2	2	core 2 duo

one physical CPU(core) can act like two logical CPUs(cores)
two threads can be scheduled on single physical core
switching between these two threads is taken
care at h/w level, which is faster
This can be acheived with help of h/w support
to handle context of two threads -- two register sets

eg:-
xeon ==> 2 chips x 4 phy x 2 logical ==> 16 logical CPUs

cat /proc/cpuinfo