
The **P** in **P**thread comes from POSIX(Portable Operating System Interface)

**https://computing.llnl.gov/tutorials/pthreads/**

https://www.youtube.com/watch?v=qkMk3k15PPA&list=PL3-wYxbt4yCjpcfUDz-TgD_ainZ2K3MUZ&index=34

**q)process vs thread??**
ans.
*.
Thread run in shared memory space but process run in separate memory space
Thread is light weight process but process is heavy weight process.
Thread is subtype of process.

**Process:** program under execution is known as process

**Thread:** థ్రెడ్,దారము, నూలుపోగు, సూత్రం

**Thread:** Thread is a functionality which is executed with the other part of the program based on the concept of "one with other"
so thread is a part of process..

We tie them around trees, wear them around necks, tie them around ankles also around the wrists.
Mangal sutram, yajnopavitam, kankanam, abhaya sutram and raksha bandhanam.

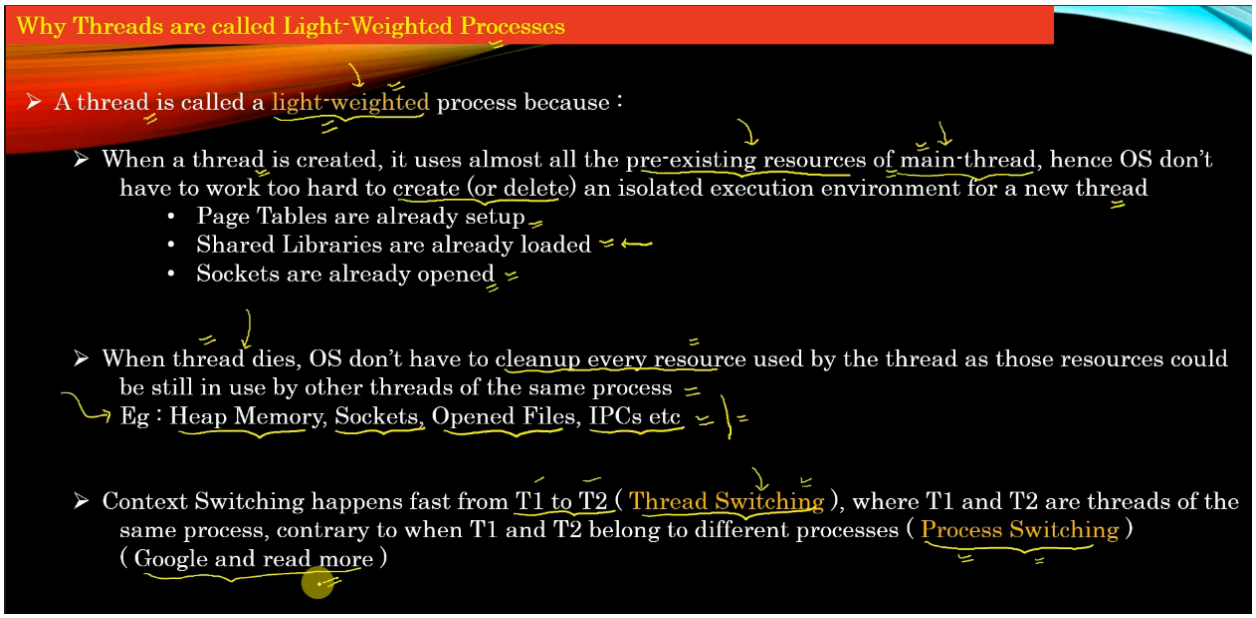
A Thread is a basic unit of *execution flow*.
A thread runs in the context of process.

**q).what do you mean main thread?**
**ans).**
***.**a process has atleast one thread -> main thread.

**q).can one thread create another thread?**
**ans).**
***.**a thread can create another threads, other threads can create more threads
And so on.

**q).why should i go for threads?**
**ans).**
*.With multiple threads of control, we can design our programs to do more than one thing at a time within a single process, with each thread handling a separate task.
*.We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type
*.Threads, in contrast, automatically have access to the same memory address space and file descriptors.
*.Some problems can be partitioned so that overall program throughput can be improved.Some problems can be partitioned so that overall program throughput can be improved.
*.

**q).why threads are light weight process?**
**ans).**
—--




—--

**q).can we use multithreaded concept in uniprocessor system?**
**ans).**
*.The benefits of a multithreaded programming model can be realized even if your program is running on a uniprocessor. A program can be simplified using threads regardless of the number of processors, because the number of processors doesn't affect the program structure. Furthermore, as long as your program has to block when serializing tasks, you can still see improvements in response time and throughput when running on a uniprocessor, because some threads might be able to run while others are blocked.

**q).what is race condition in threads?**
**ans).**
*.the race condition on thread creation is due to the fact that which thread
The kernel chooses to allocate CPU - the parent thread or new thread.
*.kernel schedules the threads on multiple CPU's as per the scheduling policy.
*.

**q).what thread context includes ?**
**ans).**
*.

a.Thread_ID within a process
B.a set of register values
C.a stack
D.a scheduling priority and policy
E.a signal mask
F.an errno variable
G.thread specific data
H.everything within a process is sharable among the threads in a process.
  .including the text of the executable program,
  .the program's global and
  .heap memory,
  .the stacks, and
  .the file descriptors
  .The newly created thread has access to the process address space and inherits the calling thread's floating-point environment and signal mask; however, the set of pending signals for the thread is cleared.

Reasons of Potential parallesim:
—------------------------------
1.Overlapping I/O
2.Asynchronous events
3.Real-time scheduling

**q).what is special about pthread_t ?**
ans).
*.*pthread_t* tid; //thread handle
*.*Opaque object*, don't bother about its internal members.
*.It is inbuilt data structure which is defined in pthread.h
*.When i say opaque, you never need to know what are internal members of This data structure.

**q). pthread_attr_init(pthread_attr_t *attr)??**

ans).
1. pthread_attr_init initialize thread attributes object

int pthread_attr_init(pthread_attr_t *attr);

2. pthread_attr_destroy destroy thread attributes object

int pthread_attr_destroy(pthread_attr_t *attr);

*.the *pthread_attr_t* data structure for threads is analogous to *open file descriptor table* data structure in linux.
*.the *pthread_attr_init()* function initializes the thread attributes object pointed to by attr with *default attribute* values.
*.after this call, individual attributes of the object can be set using various related funtions.
*.when a thread attributes object is no longer required, it should be destroyed using the pthread_attr_destroy() function. destroying a thread

attributes object has no effect on threads that were created using that object.
*.once a thread attributes object has been destroyed, it a be reinitialized using pthread_attr_init()

RETURN VALUE:
---
on success, these functions returns 0
on error ,they return a nonzero error number.

==========

**q).what if main thread/process terminates?**
**ans).**
***.**if *main()* thread dies, all other thread dies by default, but vice-versa not true.

**q)pthread_join??**
ans).
*.if we want that the main thread should wait until all the other threads are finished then there is a function pthread_join().

1.prototype:
int pthread_join(thread_t tid, void **status);

2./*waiting to join thread "tid" with status*/
int ret=pthread_join(tid,&status);

3./*waiting to join thread "tid" without status*/

```
           int ret=pthread_join(tid,NULL);

        the pthread_join() function blocks the calling thread until the
specified thread terminates.
*.when status is not NULL, it points to a location that is set to the exit
status of the terminated thread when pthread_join () returns successfully.
*.multiple threads cannot wait for the same thread to terminate. if they try
to,one thread returns successfully and the others fail with an error of
ESRCH.

        return values:
   ESRCH     :    tid is not a vaid, undetached thread in the current
                  process.
   EDEADLK   :    tid specifies the calling thread.
   EINVAL    :    the value of tid is invalid.
  returns zero, when it completes sucessfully.when any of the following
conditions.
-->any other returned value indicates that an error ocurred. when any of the
above onditions are detected, pthread_join() fails and returns the
corresponding value.
-->the function  above makes sure that its parent thread does not terminate
until it is done.
-->this function is called from within the parent thread and the first
argument is the thread ID of the thread to wait on and the second argument is
the return value of the thread on which we want to the parent thread to wait.
if we are not interested in the return value then we can set this pointer to
be NULL.
```

**q).how can thread terminates ?**
**ans).**

```
*.if we classify on a broader level, then we see that  a thread can terminate
in three ways:
1.if the thread returns from its start routine.
2.if it is canceled by some other thread. the function use here is
  pthread_cancel().
3.if its calls pthread_exit()
```

**NOTES:** after a successful call to *pthread_join()*, the caller is guaranteed that the target thread has terminated.

```
1.joining with a thread that has previouisly been joined results in undefined
behaviour.
2.all of the threads in a process are peers: any thread can join with any
other thread in the process.
```

3.failure to join with a thread that is joinable(i.e., one that is not detached), produces a "zombie thread". Avoid doing this
since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads(or processes).

**q).can parent thread get exit status of the other thread by using pthread_exit ?**
**ans).**
*.As we can see, when a thread exits by calling *pthread_exit()* or by simply returning from the start routine, the exit status can be obtained by another thread by calling *pthread_join()*.

---

**q).pthread_exit()**

prototype: void pthread_exit(void *rval_ptr);

*.so, we see that this function accepts only one argument, whih is the return from the thread that calls this function.
*.this return value is accessed by the parent thread which is waiting for this thread to terminate.
*.the return value of the thread teminated by pthread_exit() function is accessible in the second argument of the pthread_join.

**q).how can we exit from thread?**
**ans).**
*.
A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.
1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.
3. The thread can call pthread_exit.

---

**q).pthread_self()**

ans). 1.prototype: pthread_t pthread_self(void);
compile and link with -pthread.

description:
------------
the pthread_self() function returns the id of the calling thread.
ERRORS:
-------
THIS Function always succeeds, returning the calling thread's ID.
this function always succeeds.
NOTES:
------
thread IDs are guaranteed to be unique only within a process. a thread id may be reuse after a terminated thread has been joined, or detahed thread has terminated.

---

**q). gettid??**

ans).

gettid -get thread identification
prototype : pid_t gettid(void);
description : gettid() returns the caller's thread ID(TID).
in a single threaded process, the thread id is equal to the process ID (PID, as returned by getpid(2)). in a multithreaded process, all threads have the same PID, but each one has a unique TID.

RETURN VALUES:
--------------
ON sucess, return the thread ID of the calling process.
ERRORS:
-------
this call is always successful.

---

**q).pthread_create??**

ans).

prototype: pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

1.pthread_create() function starts a new thread in the calling process. the new thread starts execution by invoiking start_routine() arg is passed as the sole argument of start_routine().

The new thread terminates in one of the following ways:

1.it calls pthread_exit, specifying an exit status value that is available to another thread in the same process that calls pthread_join.
2.it is canceled
3.exit,the main thread performs a return from main(). this auses the termination of al threads in the process.

**pthread_attr_t:**

---

1.the attr argument points to a pthread_attr_t structure whose contents are used at thread creation time to determine attributes for the new thread;
2.this structure is initialized using phread_attr_init and related functions.
3.if attr is NULL, then the thread is created with default attributes.

4.Before returning, a successful call to phtread_create() stores the ID of the new thread in the buffer pointed to by thread;this identifier is used to refer to the thread in subsequent calls to other pthreads functions.
5.the new thread iherits a copy of the creating thread's signal mask(pthread_sigmask()). the set of pending signals for the new thread is empty (sigpending()).
6.the new thread does not inherit the creating thread's alternate signal stack.
7.the new thread inherits the calling thread's floating-point environment.
8.

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
From ipc section we are copying,...
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---

# pthread:

---

http://stackoverflow.com/questions/6420090/pthread-concepts-in-linux

**q)pthreads?**
**ans).**posix threads.
posix-portable operating system interface.

**q).what is multithreading?**
**ans).**

*.Multithreading is a technique to spawn multiple threads of a program.

*.the classic defination everywhere says is “threads are light weight process”.

**q).what is programming,process and thread?**
ans).
**programming**:programming is the simulatenous execution of computations.

**process**:process is a program under execution

**thread**:thread is a separate sequence of execution within the process context.

**q)process vs thread??**
ans.

1.Thread run in shared memory space but process run in separate memory space
2.Thread is light weight process but process is heavy weight process.
  Thread is subtype of process.
3.an application is normally considered as multiple process,that process can be breakdown into multiple threads
4.threads can be of two types:
    1.user space thread.
    2.kernal space thread.
5.**user thread**
  a.thread management done in user space
  b.user threads are supported and managed without kernal support
    *.invisible to the kernal.
    *.if one thread blocks,entire process blocks
    *.limited benefits of threading.
6.**kernal thread**
  a.kernal thread supported and directly managed by the os.
  b.kernal creates lightweight processes(**lwp’**s)(nothing but threads)
  c.most modern os’s supports kernal threads.
  d.
7.**benefits of threading:**
  1.**responsivness:**
    a.threads share code and data.
    b.thread creation and switching much more efficient than that of process.

c.creation of thread takes less cost than process, and context switching is fast than process.

8.multithreading withina single processor

a.so far,we have considered multiple threads of an application running on different processors.

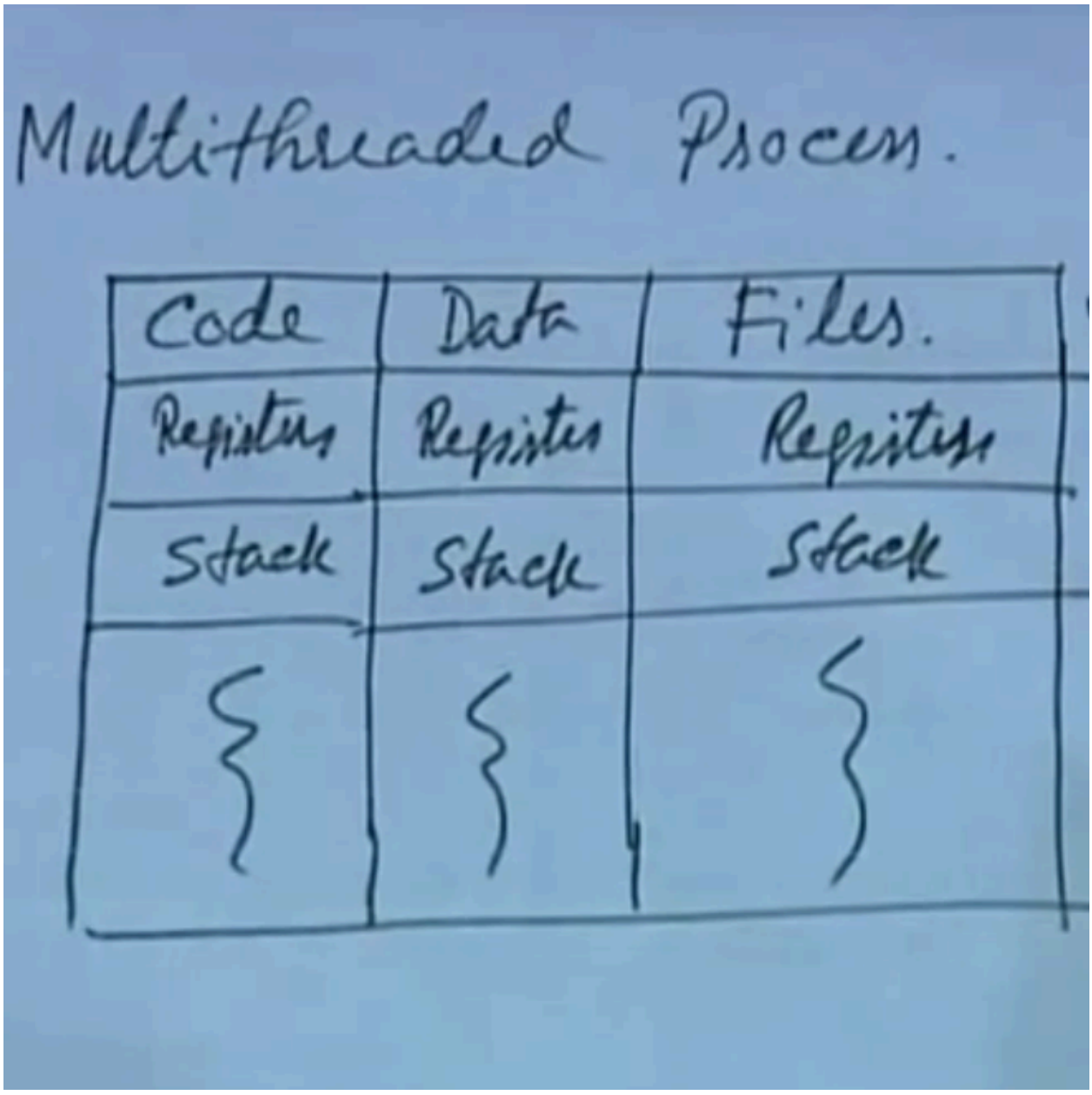
b.can multiple threads executes concurrently on the same processor? **YES.**

**c.SMT** is a technique for improving overall efficiency

d.faster communication among threads.

e.inexpensive-one processor.

| | |
|---|---|
| 1.program under execution is known as process<br>2.a process is program under execution<br><br>3.an aplication normally consists of Multiple programs.<br>4.an application has a full address space And operating environment state.<br>5.heavy weight process.<br>6.a sequence of instructions Executed Within the context of a process. | 1.thread is a functionality which is known as process<br>2.a thread can be a separate Sequence of execution within the context of process<br><br>3.a process consists of one or more threads<br>4.thread belongs to same process Shares **data** and **code space.**<br><br>5.light weight process. |

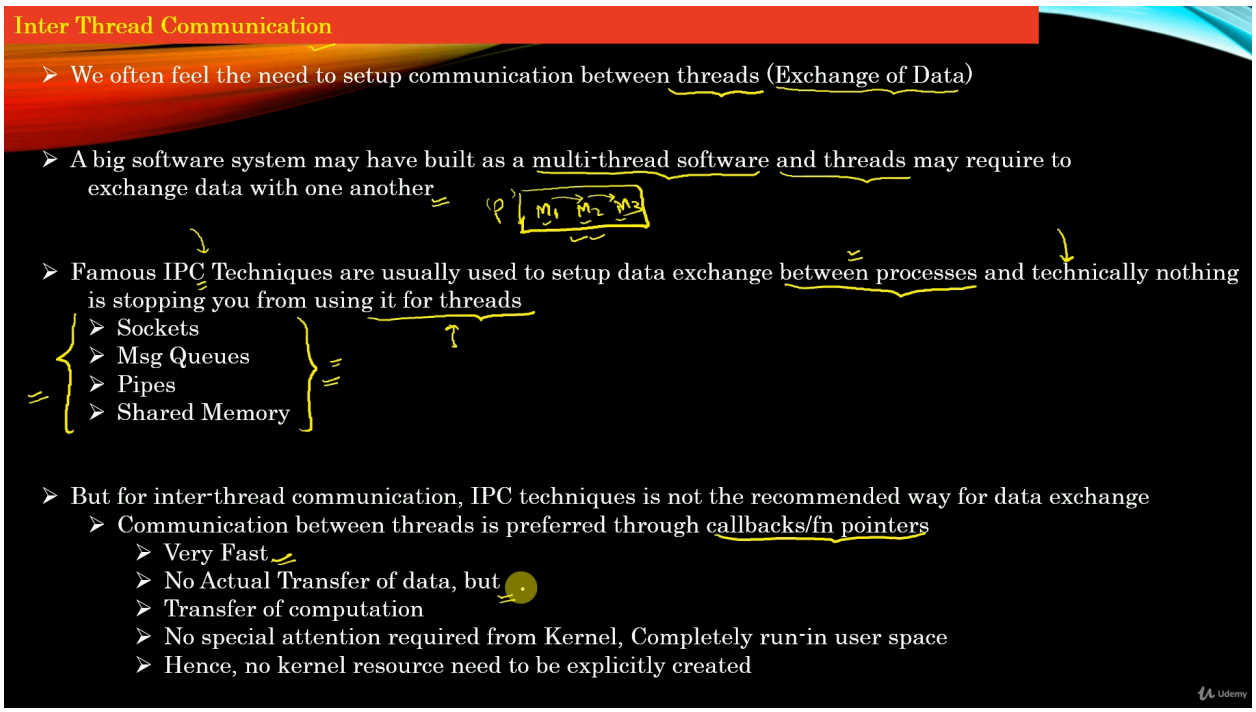



q).why use threads over processes?
q).if both the process and threads model can provide concurrent program execution, why use threads over processes?

**ans).**
**Src: pthread programming - Bradford Nichols**
*.Creating a new process can be expensive. it takes time.(A call into the operating system is needed, and if the process creation triggers process rescheduling activity, the operating system'safe context-switching mechanism will become involved.) it takes memory. (the entire process must be replicated.) add to this the cost of interprocess communication and synchronization of shared data, which also may involve calls into the operating system kernel, and threads provide an attractive alternative.

*.Threads can be created without replicating an entore process. furthermore, some, if not all, of the work of creating a thread is done in user space rather than kernel space. when processes synchronize, they usually have to issue system calls, a relatively expensive operation that involves trapping into the kernel.
But threads can synchronize by simply monitoring a variable ---- in other words, staying within the user address space of the program.

**q).can we use IPC mechanism in multithreading?**
**ans).**
*.
---




---

**q).what are suitable tasks for multi-threading?**
**ans).**
*.some properties
- potential parllalisum(It is independent of other tasks)
- Overlapping I/o (It can be blocked in potentially long waits)
  (It can use a lot of CPU cycles)
- Asynchronous events (It must be respond to a Asynchronous events)
- Real-time scheduling (It works as greater or lesser importance than Other works in application)

pthread_attr_t structure

```
typedef struct {
    int __detachstate;
    int __schedpolicy;
    struct sched_param __schedparam;
    int __inheritsched;
    int __scope;
    size_t __guardsize;
    int __stackaddr_set;
    void *__stackaddr;
    unsigned long int __stacksize;
} pthread_attr_t;
```

```
int __detachstate;
PTHREAD_CREATE_JOINABLE  0
PTHREAD_CREATE_DETACHED  1

int __schedpolicy;
SCHED_OTHER   0
SCHED_FIFO    1
SCHED_RR      2

struct sched_param __shedparam;
param.sched_priority=30;


int __inheritsched;
PTHREAD_INHERIT_SCHED  0
PTHREAD_EXPLICIT_SCHE  1

int __scope;
```

```
PTREAD_SCOPE_SYSTEM     0
PTHREAD_SCOPE_PROCESS   1

size_t __guardsize;
the defaurd size id default page size(4096bytes)
pthread_attr_setguardsize(&attr,5000);
now, the guardsize becomes 5000bytes.

int __stackaddr_set

void *_stackaddr
pthread_attr_getstackaddr(&attr,&stacksize);
base=(void *)malloc(PTHREAD_STACK_MIN +0x4000);
pthread_attr_setstackaddr(&attr,base);

unsigned long int __stacksize
PTHREAD_STACK_MIN                   16384(this is stack minmum default size)
_POSIX_THREAD_ATTR_STACKADDR        200809
_POSIX_THREAD_ATTR_STACKSIZE        200809
PAGESIZE                            4096
PAGE_SIZE                           4096(this is default minimum page size)


cancelability states

pthead_setcancelstate()       PTHREAD_CANCEL_ENABLE         0
                              PTHREAD_CANCEL_DISABLE        1

pthread_setcanceltype()       PTHREAD_CANCEL_ASYNCHRONOUS   0
                              PTHREAD_CANCEL_DEFFERED       1

 pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
 pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);


pthread_mutexattr_{get,set}pshared
pthread_condattr_{get,set}pshared

PTHREAD_PROCESS_PRIVATE       = 0,
PTHREAD_PROCESS_SHARED        = 1,

pthread_mutexattr_{get,set}robust
```

```
PTHREAD_MUTEX_STALLED         = 0,  /* Default */
PTHREAD_MUTEX_ROBUST          = 1,

pthread_barrier_wait

PTHREAD_BARRIER_SERIAL_THREAD = -1
```

```
pthread_t;
pthread_attr_t;
pthread_t pthread_self(void);
int pthread_equal(pthread_t tid1 , pthread_t tid2);
int pthread_attr_init(pthread_attr_t *attr);
int pthread_create(pthread_t *restrict tid ,
                  const pthread_attr_t *restrict attr,
                  void * (*start_routine)(void *),
                  void *restrict arg);


int pthread_getconcurrency(void);
int pthread_attr_getguardsize(const pthread_attr_t *attr,
                              size_t *restrict guardsize);
int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                         size_t *restrict stacksize);
int pthread_attr_getstack(const pthread_attr_t *restrict attr ,
                    void ** restrict stackaddr,
                    size_t *stacksize);
int pthread_attr_getdetachstate(const pthread_attr_t *restrit attr,
                                int *detachstate);
int pthread_join(pthread_t thread, void **rval_ptr);
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int  detachstate);
int pthread_attr_setstack(const pthread_attr_t *attr,
                    void *stackaddr,
                    size_t stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr ,
                              size_t *stacksize);
```

```
int pthread_setconcurrency(level);
int pthread_attr_setguardsize(pthread_attr_t *attr,size_t guardsize);
int pthread_cancel(pthread_t tid);
void pthread_cleanup_push(void (*rtn)(void *) , void *arg);
void pthread_cleanup_pop(int execute);
int pthread_detach(pthread_t tid);
int pthread_attr_destroy(pthread_attr_t *attr);
void pthread_cleanup_push(void (*routine)(void *),void *arg);
void pthread_cleanup_pop(int execute);
```

**q).multi thread programming?**

**ans).**imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or
independently by the operating system. That would describe a "multithreaded" program.

*.multithreading (MT) separates a process into many execution threads, each of which runs independently.

**q).in what way multithreading can be advantage for u?**

ans).

Multithreading your code can

1.Improve application responsiveness
2.Use multiprocessors more efficiently
3.Improve program structure
4.Use fewer system resources

**q).what are all threads contain unique for that?**

**ans).**The following state is unique to each thread.

- Thread ID
- Registers

- Stack area and stack pointer
- Signals
- Scheduling Priorities(such as policies or priorities)
- Thread-private storage (thread specific data)

**q).when we go to mulitthreading?**
**ans).**A program can be multithreaded when there are some independent **sub tasks,** each thread performing this independent task and results are collated and acted upon.

**q).what threads should share?**
**ans).**it says threads should share:

- process ID
- parent process ID
- process group ID and session ID
- controlling terminal
- user and group IDs
- open file descriptors
- record locks (see fcntl(2))
- signal dispositions
- file mode creation mask (umask(2))
- current directory (chdir(2)) and root directory (chroot(2))
- interval timers (setitimer(2)) and POSIXtimers(timer_create(2))
- nice value (setpriority(2))
- resource limits (setrlimit(2))
- measurements of the consumption of CPU time (times(2)) and resources (getrusage(2))

**q).what each thread would own?**
**ans).**

- thread ID (the pthread_t data type)
- signal mask (pthread_sigmask(3))
- the errno variable
- alternate signal stack (sigaltstack(2))
- real-time scheduling policy and priority (sched_setscheduler(2) and sched_setparam(2))
- capabilities (see capabilities(7)) , under Linux
- CPU affinity (sched_setaffinity(2)) , under Linux

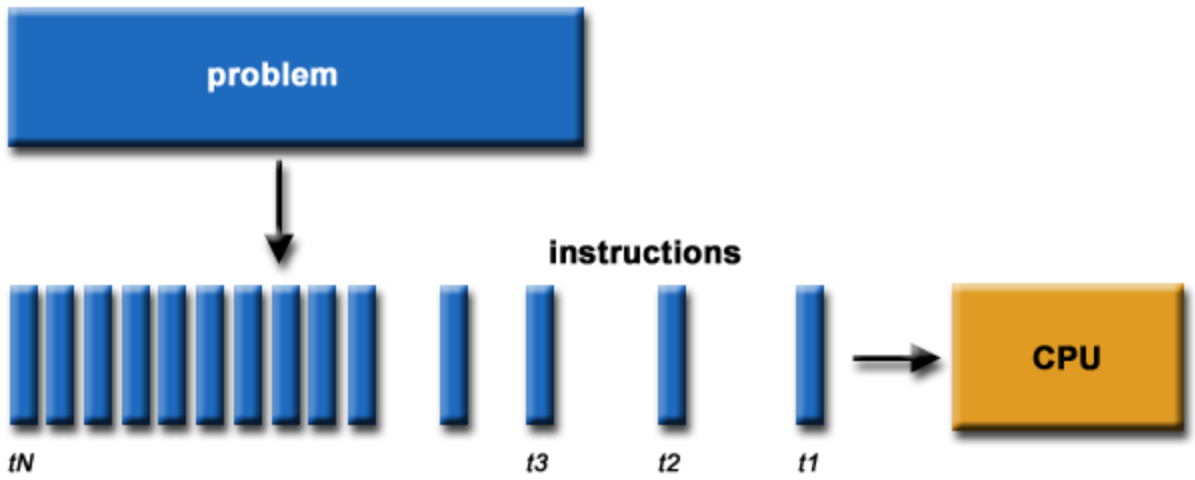
**q)what are concurrency and parllelisum related to multithreaded programming?**

**ans).**

http://concur.rspace.googlecode.com/hg/talk/concur.html#landing-slide

http://stackoverflow.com/questions/1050222/concurrency-vs-parallelism-what-is-the-difference

## concurrency:




concurrent -ఉభయ
-ఏకీభావంగా
-ఏకకాలంలో
-occurring or operating at the same time.
synonyms -simulatenously
-at the same time.
-together.
*.**Concurrency**: Interruptability

*.**Concurrency** is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. Eg. multitasking on a single-core machine.

*.In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution.

*.The term **Concurrency** refers to techniques that make program more usable.
Src:
https://stackoverflow.com/questions/1050222/what-is-the-difference-between-concurrency-and-parallelism

*.short answer: Concurrency is two lines of customers ordering from a single cashier (lines take turns ordering); Parallelism is two lines of customers ordering from two cashiers (each line gets its own cashier).

*.Mnemonic to remember this metaphor: Concurrency == same-time *Customers*; Parallelism == same-time *Payments*

***.Concurrency** is when two or more tasks can start, run, and complete in overlapping time **periods**. It doesn't necessarily mean they'll ever both be running **at the same instant**. For example, *multitasking* on a single-core machine.
**Parallelism** is when tasks *literally* run at the same time, e.g., on a multicore processor.

# *I*.Why the Confusion Exists

*Confusion exists because dictionary meanings of both these words are almost the same:*

- ***Concurrent**: existing, happening, or done at the same time(dictionary.com)*
- ***Parallel**: very similar and often happening at the same time(merriam webster).*

*Yet the way they are used in computer science and programming are quite different. Here is my interpretation:*

- ***Concurrency**: Interruptability*
- ***Parallelism**: Independentability*

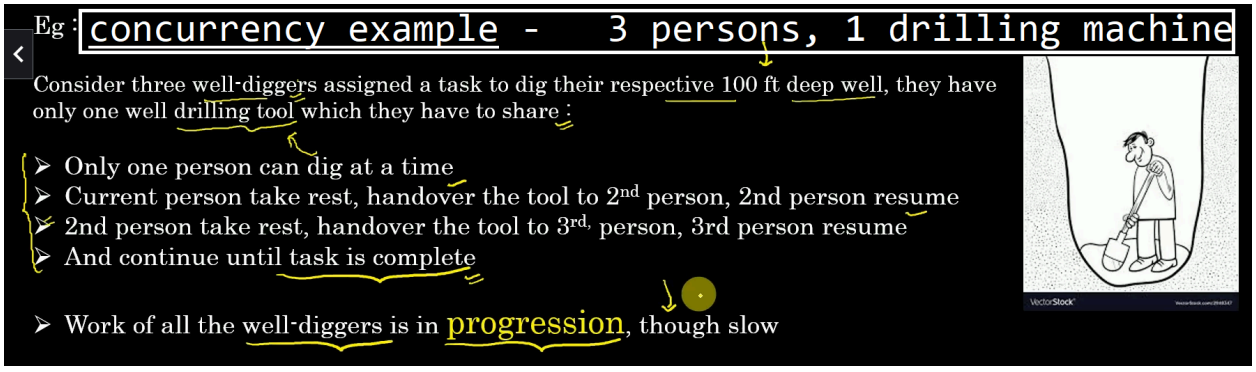
**q). Why should i choose threads for concurrency instead of processes ? is there any benefit ?**
**ans).**

*.the major benefit of multi threaded programs over non-threaded once is their ability to concurrently execute their tasks.
*.However, in providing concurrency, multi-threaded programs introduce certain amount Of overhead.
*.if you introduce in an application that can't use concurrency, you will add overhead without any performance benefit.

So, what makes concurrency possible?
First, of course, your application must consist of some independent tasks – tasks that do not depend on the completion of other tasks to proceed. Secondly, you must be confident that concurrent execution of these tasks would be faster than their serial execution.

On a uniprocessor system, the concurrent execution of independent tasks will be faster than their serial execution.

---

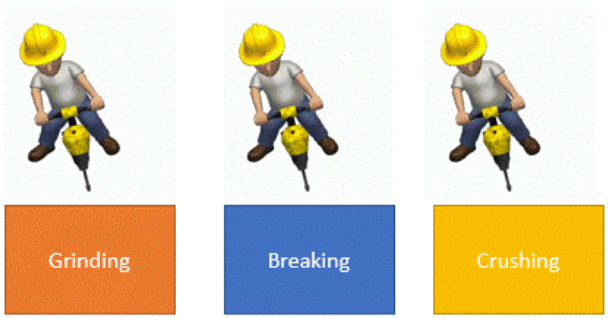



---

- For a 9 years old guy

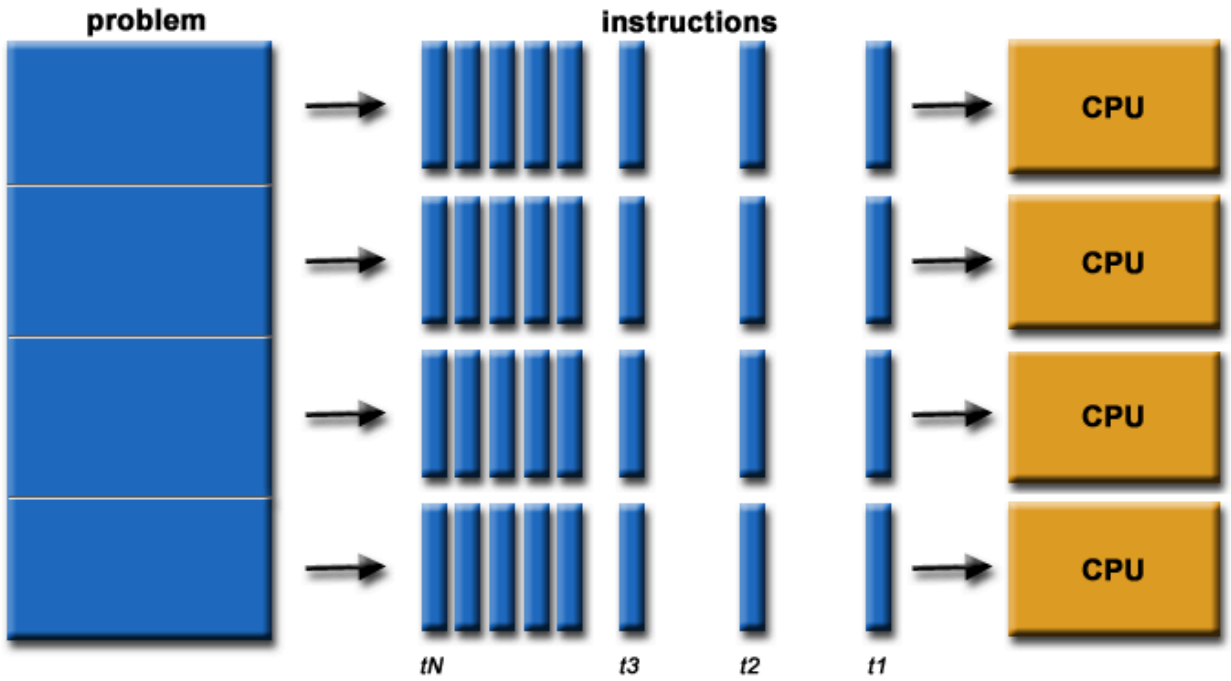
https://i.stack.imgur.com/ZqHC3.gif

—---




Parallel Processing: Doing more then one thing at a time.

## parrllelisum:




parrllelisum -సమాంతరమైన

-ఒకే పోలిక గల
-of or relating to the simultaneous performance of multiple operations.
-parllel lines or planes.

*.**Parallelism**: Independentability

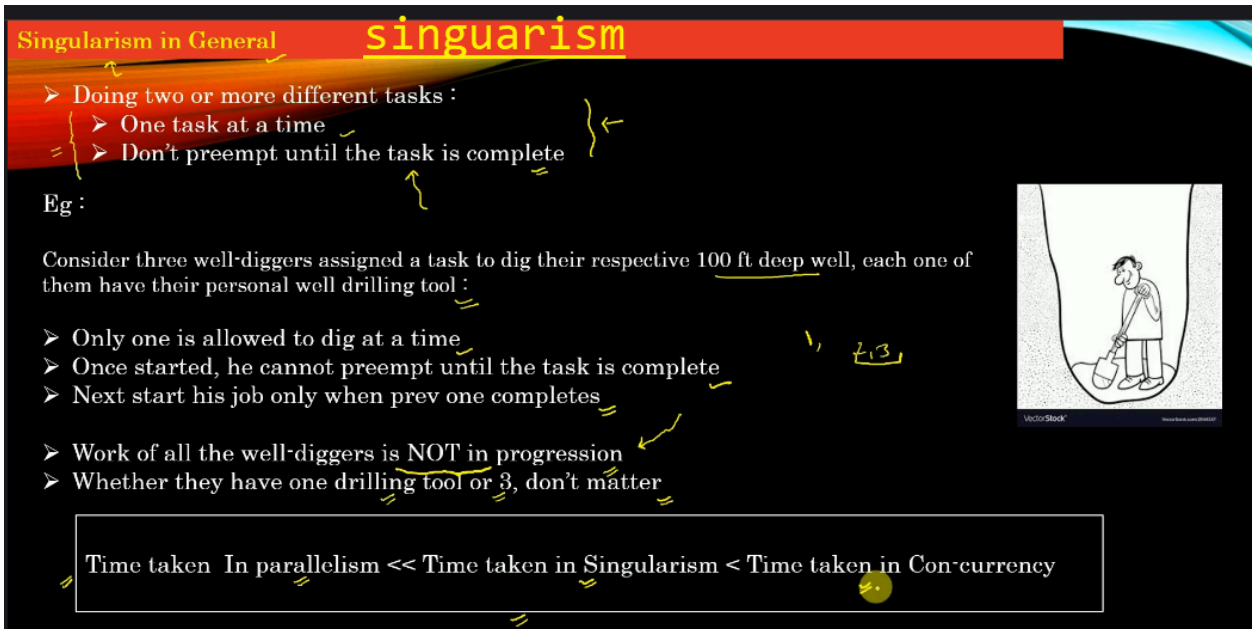
*.**Parallelism** is when tasks literally run at the same time, eg. on a multicore processor.

*.parallelism as the run-time behaviour of executing multiple tasks at the same time
*.In the same multithreaded process in a shared-memory multiprocessor environment, each thread in the process can run on a separate processor at the same time, resulting in parallel execution.

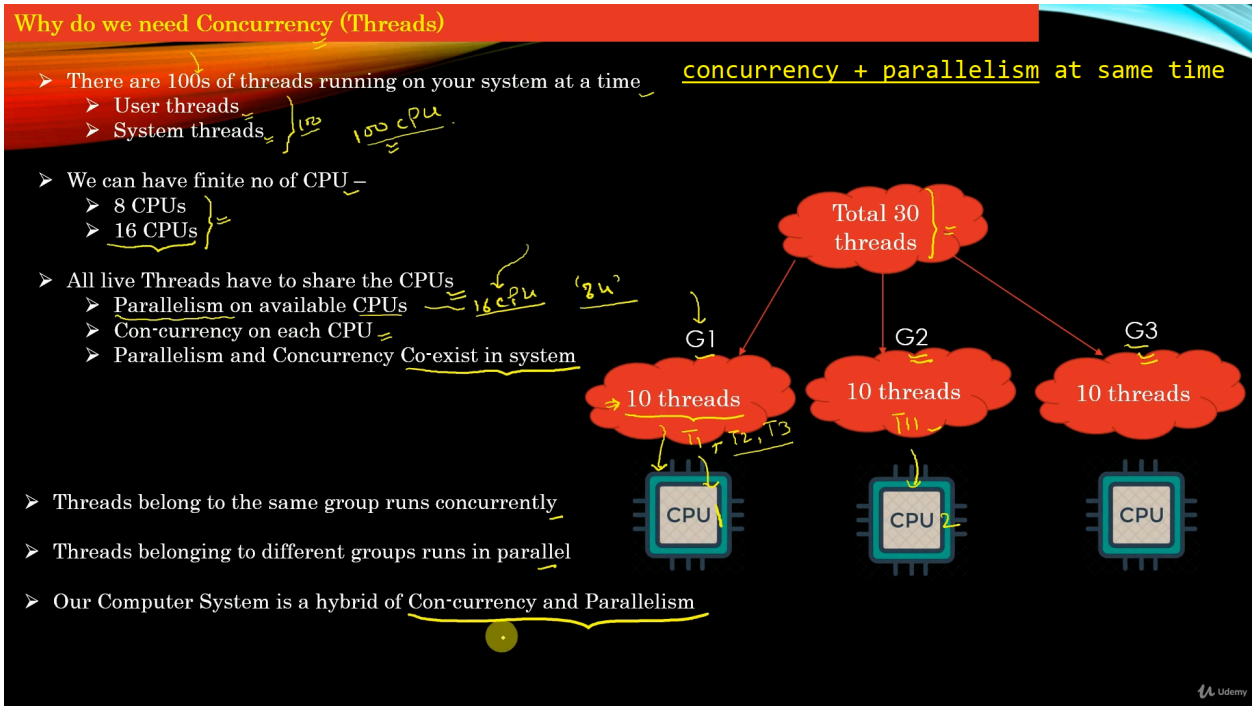
*.When the process has fewer or as many threads as there are processors, the threads support system in conjunction with the operating environment ensure that each thread runs on a different processor.

**Singualism:**

---



---




---

**q).can parllelism possible with single core system?**
ans).parallelism (in the sense of multithreading) is not possible with single core processors.

**q).How to distinguish between Parallelism and Concurrency?**

If you need getNumCapabilities

1).in your program, then your are certainly programming parallelism.

2).If your parallelising efforts make sense on a single processor machine, too, then you are certainly programming concurrency.

q).difference between bound threads and unbound threads?
ans).

**q). What if process dies? but Threads need to do running task?**
**ans).**
*.by the way all threads expires, when the process in which they run exits .

**q).what if thread dies? Process also dies at same time?**
**ans).**
*.when thread terminates, the pthread's library reclaims any process or system resources the thread was using and stores its exit status.
*.a thread can also explicitly exit with *pthread_exit()*.
*.you can terminates another thread by calling *pthread_cancel()*.

**q).Can i know the thread exit status?**
**ans).**
*.The pthreads library may or may not save the exit status of a thread when the thread exits, depending upon when the thread is *joinable* or *detached*.
*.a *joinable* thread, the default state of thread at its creation, does have its exit status saved; a *detached* thread does not.
*.you can know the exit status by pthread_exit call or return value of the Routine.

**q).name any pthread call which always succeeded?**
**ans).**
*.a ***pthread_self()*** call always succeeds.

---

## scheduling:

SCHED_FIFO
SCHED_RR
SCHED_OTHER

## process scope:

PTHREAD_SCOPE_PROCESS

## system scope:

PTHREAD_SCOPE_SYSTEM

## synchronization:

Synchronization allows you to control program flow and access to shared data for concurrently executing threads.
The four synchronization models are mutex locks, read/write locks, condition variables, and semaphores.

- Mutex locks allow only *one thread at a time* to execute a specific section of code, or to access specific data.

- Read/write locks permit concurrent reads and exclusive writes to a protected shared resource. To modify a resource, a thread must first acquire the exclusive write lock. An exclusive write lock is not permitted until all read locks have been released.
- Condition variables block threads until a particular condition is true.
- Counting semaphores typically coordinate access to resources. The count is the limit on *how many threads* can have access to a *semaphore*. When the count is reached, the semaphore blocks.

**q).what is default thread?**
**ans).**

When an *attribute object* is not specified, it is **NULL**, and the default thread is created with the following attributes:

1.Unbound

2.Nondetached

3.With a default stack and stack size

4.With a priority of zero

5.process scope

You can also create a default attribute object with pthread_attr_init(), and then use this attribute object to create a default thread. See the section Initialize Attributesfor details.

**q).what are all default attributes for a thread?**

**q).what *attributes* means?**
**ans).**
Attributes are a way to specify behavior that is different from the default.
http://docs.oracle.com/cd/E19683-01/806-6867/6jfpgdcnc/index.html#attrib-39150

**q).can we allowable to modifiy the attr object with assingment?**
**ans).**An attribute object is ***opaque(***opaque means,invisible***)***, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type.
**q).in how many ways we can detach a thread?**
**ans).**

```
1st way:   pthread_attr_init(&attr);
           pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

2nd way:   pthread_detach(tid);
```

---

pthread_join:

---

spawn : అధిక సంతానము
synonyms : give rise to, cause, call forth

```
pthread_join(pthread_t tid, void **rval_ptr);
```

**q)pthread_join??**
**ans).**

*.It is important for the main thread to wait till all threads in the process are complete. This waiting is performed using pthread_join(). This wait is required to receive termination status of threads spawned. If main terminates before termination of joinable threads, this joinable thread will become a zombie, occupying system resources.
—--

| Attribute | Value | Result |
|---|---|---|
| *scope* | PTHREAD_SCOPE_PROCESS | New thread is unbound – not permanently attached to LWP. |
| *detachstate* | PTHREAD_CREATE_JOINABLE | Exit status and thread are preserved after the thread terminates. |
| *stackaddr* | NULL | New thread has system-allocated stack address. |
| *stacksize* | 0 | New thread has system-defined stack size. |
| *priority* | 0 | New thread has priority 0. |
| *inheritsched* | PTHREAD_EXPLICIT_SCHED | New thread does not inherit parent thread scheduling priority. |
| *schedpolicy* | SCHED_OTHER | New thread uses Solaris-defined fixed priorities for synchronization object contention; threads run until preempted or until they block or yield. |

—--
*.if we want that the main thread should wait until all the other threads are finished then there is a function pthread_join().

1.prototype:

```
int pthread_join(pthread_t tid, void **status);
```

**// here void **status is used to store termination status of thread**
**// it is equivalent to waitpid, the thread1 can be blocked untill, the thread2 terminates(change state).**

2./*waiting to join thread "tid" with status*/

```
int ret=pthread_join(tid,&status);
```

3./*waiting to join thread "tid" without status*/

```
int ret=pthread_join(tid,NULL);
```

the pthread_join() function blocks the calling thread until the specified thread terminates.
*.The specified thread must be in the current process and must not be detached.

*.when status is not NULL, it points to a location that is set to the exit status of the terminated thread when pthread_join() returns successfully.

*.multiple threads cannot wait for the same thread to terminate. if they try to,one thread returns successfully and the others fail with an error of ESRCH.

*.When the thread terminates, its return value is obtained in second parameter of pthread_join().
Any thread can perform pthread_join() on any other thread. If multiple threads simultaneously try to join with the same thread, the results are undefined.

**return values**:
ESRCH : tid is not a vaid, undetached thread in the current process.
EDEADLK : tid specifies the calling thread.
EINVAL : the value of tid is invalid.
returns zero, when it completes sucessfully.when any of the following conditions.
-->any other returned value indicates that an error ocurred. when any of the above onditions are detected, pthread_join() fails and returns the corresponding value.
-->the function above makes sure that its parent thread does not terminate until it is done.
-->this function is called from within the parent thread and the first argument is the thread ID of the thread to wait
on and the second argument is the return value of the thread on which we want to the parent thread to wait. if we are not interested in the return value then we can set this pointer to be NULL.
*.if we classify on a broader level, then we see that a thread can terminate in three ways:
1.if the thread returns from its start routine.
2.if it is canceled by some other thread. the function use here is pthread_cancel().
3.if its calls pthread_exit().

**NOTES**: after a successful call to pthread_join(), the caller is guaranteed that the target thread has terminated.

1.joining with a thread that has previouisly been joined results in undefined behaviour.

2.all of the *threads in a process are peers*: any thread can join with any other thread in the process.

3.failure to join with a thread that is joinable(i.e., one that is not detached), produces a "zombie thread". Avoid doing this

since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads(or processes).

q).what happens if multiple threads are wait for to read termination status of the particular child?

ans).If multiple threads wait for the same thread to terminate, they all wait until the target thread terminates, than one thread returns successfully and the others fail with an error of **ESRCH**.

---

**pthread_exit:**

---

```
pthread_exit(void *rval_ptr);
//which is used to terminate the thread normally exit() is used to terminate the process.
```

q).pthread_exit()

prototype: void pthread_exit(void *rval_ptr);

*.so, we see that this function accepts only one argument, whih is the return from the thread that calls this function.

*.this return value is accessed by the parent thread which is waiting for this thread to terminate.

*.the return value of the thread teminated by pthread_exit() function is accessible in the second argument of the pthread_join.

---

**pthread_self:**
---

pthread_self(void);

**q).pthread_self()**

ans). 1.prototype: pthread_t pthread_self(void);
compile and link with -pthread.
description:
---
the pthread_self() function returns the id of the calling thread.

ERRORS:
---
THIS Function always succeeds, returning the calling thread's ID.
this function always succeeds.
NOTES:
---
thread IDs are guaranteed to be unique only within a process. a thread id may be reuse after a terminated thread has been joined, or detahed thread has terminated.

---
**q). gettid??**
**ans).**
gettid -get thread identification
prototype : pid_t gettid(void);
description : gettid() returns the caller's thread ID(TID).
in a single threaded process, the thread id is equal to the process ID (PID, as returned by getpid(2)). in a multithreaded process, all threads have the same PID, but each one has a unique TID.

**RETURN VALUES:**
---
ON sucess, return the thread ID of the calling process.

ERRORS:
---
this call is always successful.

---
pthread_create:

---

**q).pthread_create??**
**ans).**

prototype: pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

```
/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

1.pthread_create() function starts a new thread in the calling process. the new thread starts execution by invoiking
start_routine()
arg is passed as the sole argument of start_routine().

The new thread **terminates** in one of the following ways:

1.it calls pthread_exit, specifying an exit status value that is available to *another thread* in the same process that calls pthread_join.

2.it is canceled.

3.exit,the main thread performs a return from main(). this causes the termination of all threads in the process.

**pthread_attr_t:**
---

1.the attr argument points to a pthread_attr_t structure whose contents are used at thread creation time to *determine attributes* for the new thread;

2.this structure is initialized using pthread_attr_init and related functions.

3.if **attr** is **NULL**, then the thread is created with default attributes.

4.Before returning, a successfull call to phtread_create() stores the ID of the new thread in the buffer pointed to by thread;
this identifier is used to refer to the thread in subsequent calls to other pthreads functions.
5.the new thread inherits a copy of the creating thread's signal mask(**pthread_sigmask()**). the set of pending signals for the new thread is empty (sigpending()).

6.the new thread does not inherit the creating thread's alternate signal stack.

7.the new thread inherits the calling thread's floating-point environment.

8.

---

**pthread_attr_init**:

---

q). pthread_attr_init(pthread_attr_t *attr)??

ans).

**Prototype**:

```
int pthread_attr_init(pthread_attr_t *attr);
#include <pthread.h>

pthread_attr_t attr;
int ret;

/* initialize an attribute to the default value */
ret = pthread_attr_init(&attr);
```

1. pthread_attr_init initialize thread attributes object

int pthread_attr_init(pthread_attr_t *attr);

2.  pthread_attr_destroy  destroy thread attributes object

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

```
/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

*.the pthread_attr_init() function initializes the thread attributes object pointed to by attr with default attribute values.

*.after this call, individual attributes of the object can be set using various related funtions.

*.when a thread attributes object is no longer required, it should be destroyed using the pthread_attr_destroy() function.
destroying a thread attributes object has no effect on threads that were created using that object.

*.once a thread attributes object has been destroyed, it a be reinitialized using pthread_attr_init()

**RETURN VALUE:**
---
on success, these functions returns 0
on error ,they return a nonzero error number.

---

**pthread_attr_getdetachstate:**

**Prototype:**

```
int   pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                                        int
*detachstate);
#include <pthread.h>

pthread_attr_t attr;
int detachstate;
int ret;
```

```
/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&attr, &detachstate);
```

---

**pthread_attr_setdetachstate:**

When a thread is created detached (PTHREAD_CREATE_DETACHED), its thread ID and other resources can be reused as soon as the thread terminates. Use pthread_attr_setdetachstate() when the calling thread does not want to wait for the thread to terminate.

When a thread is created **nondetached** (PTHREAD_CREATE_JOINABLE), it is assumed that you will be waiting for it. That is, it is assumed that you will be executing a **pthread_join()** on the thread.

Whether a thread is created detached or nondetached, the process does not exit until all threads have exited.

**creating detached thread:**

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init()(&tattr);
ret = pthread_attr_setdetachstate()(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create()(&tid, &tattr, start_routine, arg);
```

---

**pthread_attr_getguardsize:**

```
int pthread_attr_getguardsize(const pthread_attr_t *attr,
                                                size_t  *guardsize);
```

we can say get stack guard size.

---

**pthread_attr_setguardsize:**

```
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t  guardsize);
```

The **guardsize** argument provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates **extra memory** at the **overflow** end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a **SIGSEGV** signal being delivered to the thread).
The guardsize attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and knows its threads will never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on stack, a large guard area may be needed to detect stack overflow.

If **guardsize** is zero, a guard area will not be provided for threads created with **attr**. If **guardsize** is greater than zero, a guard area of at least size **guardsize** bytes is provided for each thread created with **attr**. By default, a thread has an implementation-defined, non-zero guard area.
A conforming implementation is permitted to round up the value contained in guardsize to a multiple of the configurable system variable **PAGESIZE** (see PAGESIZE in sys/mman.h). If an implementation rounds up the value of **guardsize** to a multiple of PAGESIZE, a call to **pthread_attr_getguardsize()** specifying **attr** will store, in **guardsize**, the guard size specified in the previous call to **pthread_attr_setguardsize()**.

---

**pthread_attr_getscope:**

**note**: Both thread types are accessible only within a given process.
Use pthread_attr_getscope() to retrieve the thread scope, which indicates whether the thread is bound or unbound.
**Prototype**:

```
int   pthread_attr_getscope(pthread_attr_t * attr, int *scope);
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

---

**pthread_attr_setscope:**

**note:  Both thread types are accessible only within a given process.**

Use pthread_attr_setscope() to create a bound thread (PTHREAD_SCOPE_SYSTEM) or an unbound thread (PTHREAD_SCOPE_PROCESS).

**Prototype**:

```
int   pthread_attr_setscope(pthread_attr_t *attr, int scope);
#include <pthread.h>

pthread_attr_t attr;
int ret;

/* bound thread */
ret = pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* unbound thread */
ret = pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
```

Notice that there are three function calls in this example: one to initialize the attributes, one to set any variations from the default attributes, and one to create the pthreads.

```
#include <pthread.h>

pthread_attr_t attr;
```

```
pthread_t tid;
void start_routine;
void arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&attr);

/* BOUND behavior */
ret =  pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &attr, start_routine, arg);
```

---

**pthread_attr_getschedpolicy:**

Use pthread_attr_getschedpolicy() to retrieve the scheduling policy.
**Prototype**:

```
int   pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);
#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;

/* get scheduling policy of thread */
ret = pthread_attr_getschedpolicy (&tattr, &policy);
```

---

**pthread_attr_setschedpolicy:**

Use pthread_attr_setschedpolicy() to set the scheduling policy. The POSIX standard specifies scheduling policy attributes of SCHED_FIFO (first-in-first-out), SCHED_RR (round-robin), or SCHED_OTHER (an implementation-defined method).

- SCHED_FIFO
- First-In-First-Out; threads whose contention scope is system (PTHREAD_SCOPE_SYSTEM) are in real-time (RT) scheduling class if the calling process has an effective user id of 0. These threads, if not preempted by a higher priority thread, will proceed until they yield or block. SCHED_FIFO for threads that have a contention scope of process

(PTHREAD_SCOPE_PROCESS) or whose calling process does not have an effective user id of 0 is based on the TS scheduling class.

- SCHED_RR
- Round-Robin; threads whose contention scope is system (PTHREAD_SCOPE_SYSTEM) are in real-time (RT) scheduling class if the calling process has an effective user id of 0. These threads, if not preempted by a higher priority thread, and if they do not yield or block, will execute for a time period determined by the system. SCHED_RR for threads that have a contention scope of process (PTHREAD_SCOPE_PROCESS) or whose calling process does not have an effective user id of 0 is based on the TS scheduling class.

SCHED_FIFO and SCHED_RR are optional in POSIX, and are supported for real time bound threads only.
For a discussion of scheduling, see the section Scheduling.
**Prototype**:

```
int   pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

---

**pthread_attr_getinheritssched:**

pthread_attr_getinheritsched() returns the scheduling policy set by **pthread_attr_setinheritsched()**.
**Prototype**:

```
int   pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit);
#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* get scheduling policy and priority of the creating thread */
ret = pthread_attr_getinheritsched (&tattr, &inherit);
```

---

**pthread_attr_setinheritsched:**

Use pthread_attr_setinheritsched(3THR) to set the inherited scheduling policy.
An inherit value of PTHREAD_INHERIT_SCHED means that the scheduling policies defined in the creating thread are to be used, and any scheduling attributes defined in the **pthread_create()** call are to be ignored. If PTHREAD_EXPLICIT_SCHED (the default) is used, the attributes from the **pthread_create()** call are to be used.
**Prototype**:

```
int   pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);
#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* use the current scheduling policy */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

---

**pthread_attr_getschedparam:**

pthread_attr_getschedparam(3THR) returns the scheduling parameters defined by **pthread_attr_setschedparam()**.
**Prototype**:

```
int   pthread_attr_getschedparam(pthread_attr_t *tattr,
    const struct sched_param *param);
#include <pthread.h>

pthread_attr_t attr;
struct sched_param param;
```

```
int ret;

/* get the existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);
```

---

**pthread_attr_setschedparam:**

pthread_attr_setschedparam() sets the scheduling parameters.
Scheduling parameters are defined in the param structure; only priority is supported. Newly created threads run with this priority.
**Prototype**:

```
int   pthread_attr_setschedparam(pthread_attr_t *tattr,
                                 const struct sched_param *param);
#include <pthread.h>

pthread_attr_t tattr;
int newprio;
sched_param param;
newprio = 30;

/* set the priority; others are unchanged */
param.sched_priority = newprio;

/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

---

**pthread_attr_getstacksize:**

pthread_attr_getstacksize() returns the stack size set by **pthread_attr_setstacksize()**.
**Prototype**:

```
int   pthread_attr_getstacksize(pthread_attr_t *attr, size_t *size);
#include <pthread.h>

pthread_attr_t attr;
size_t size;
int ret;
```

```
/* getting the stack size */
ret = pthread_attr_getstacksize(&attr, &size);
```

---

**pthread_attr_setstacksize:**

pthread_attr_setstacksize() sets the thread stack size.
The **stacksize** attribute defines the size of the stack (in bytes) that the system will allocate. The size should not be less than the system-defined minimum stack size. See About Stacksfor more information.
**Prototype**:

```
int   pthread_attr_setstacksize(pthread_attr_t *attr, size_t size);
#include <pthread.h>

pthread_attr_t tattr;
size_t size;
int ret;

size = (PTHREAD_STACK_MIN + 0x4000);

/* setting a new size */
ret = pthread_attr_setstacksize(&tattr, size);
```

In the example above, size contains the number of bytes for the stack that the new thread uses. If **size** is zero, a default size is used. In most cases, a zero value works best.
**PTHREAD_STACK_MIN** is the amount of stack space required to start a thread. This does not take into consideration the threads routine requirements that are needed to execute application code.

---

**pthread_cleanup_push:**
---

```
void pthread_cleanup_push(void (*rtn)(void *) , void *arg );
//it is like atexit function to cleanup process on termination
```

---

**pthread_cancel:**
---

```
pthread_cancel(pthread_t thread);
```

*.The pthread_cancel() function sends a cancellation request to the thread *thread*.
*.using pthread_cancel funtion, a thread can abnormally terminate another Thread.
*.

**q).Wow, so, I can literally/abnormally Cancel any thread with *pthread_cancel()* right?**
**ans).**
*.Hold on, It merely makes the request a thread can elect to ignore or otherwise control how it is canceled.

```
Void pthred_cleanup_push(void (*rtn)(void *), void *arg);
Void pthread_cleanup_pop(int execute);
```

**q).How the *thread* reacts when we call pthread_cancel(*thread*) ?**
**ans).**
*.Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability **state** and **type**.

**q).who can determine thread cancelability state and type?**
**ans).**
*.A thread's cancelability state, determined by *pthread_setcancelstate(3)*
*.A thread's cancellation type, determined by *pthread_setcanceltype(3)*

NOTE:
*.for new theads by default

**state** is enabled  (**PTHREAD_CANCEL_ENABLE**)
**type**  is deferred (**PTHREAD_CANCEL_DEFERRED**)

**q).what happen, if a thread is disabled a cancelation state?**
**ans).**If a thread has disabled cancellation, then a cancellation request remains queued until the thread enables cancellation. If a thread has enabled cancellation, then its cancelability type determines when cancellation occurs.

**q).asynchronous cancelability means?**
**ans).**
*.Asynchronous cancelability means that the thread can be canceled at any time (usually immediately, but the system does not guarantee this)

**q).dereferred canelability means?**
ans).Deferred cancelability means that cancellation will be delayed until the thread next calls a function that is a *cancellation point*.

**q).what are these cancellation points?**
ans).
*.A list of functions that are or may be cancellation points is provided in pthreads(7).
*.posix standards required following are the cancellation points http://www.questionscompiled.com/pthreads.html may be cancellation points

**q).what happens, if a thread is already canceled but we are using pthread_join?**
ans).
*.After a canceled thread has terminated, a join with that thread using pthread_join(3) obtains **PTHREAD_CANCELED** as the thread's exit status. (Joining with a thread is the only way to know that cancellation has completed.)

**q).when cancelation request is acted on?**
**ans).**

1. Cancellation clean-up handlers are popped (in the reverse of the order in which they were pushed) and called.  (See pthread_cleanup_push(3).)

2. Thread-specific data destructors are called, in an unspecified
order. (See **pthread_key_create**(3).)

3. The thread is terminated. (See pthread_exit(3).)

**q).what about the return status of pthread_cancel?**

ans). The above 1,2 and 3 steps happen asynchronously with respect to the **pthread_cancel**() call; the return status of **pthread_cancel**() merely informs the caller whether the cancellation request was successfully queued.

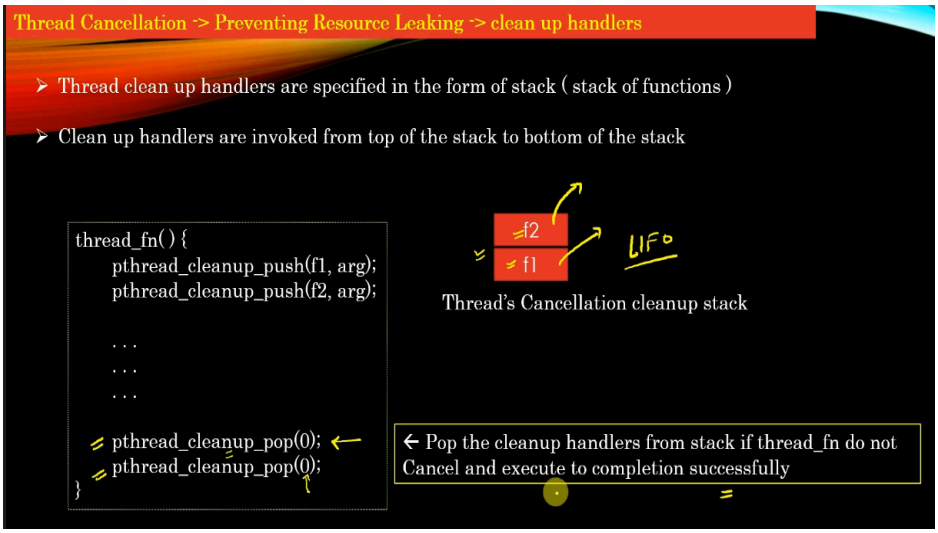
**q).when clean up handlers are called?**

ans).When type is asynchronous, threads immediately execute cleanup handlers from cleanup stack. If it were deffered, cleanup handlers are invoked once a cancellation point is reached in the thread.

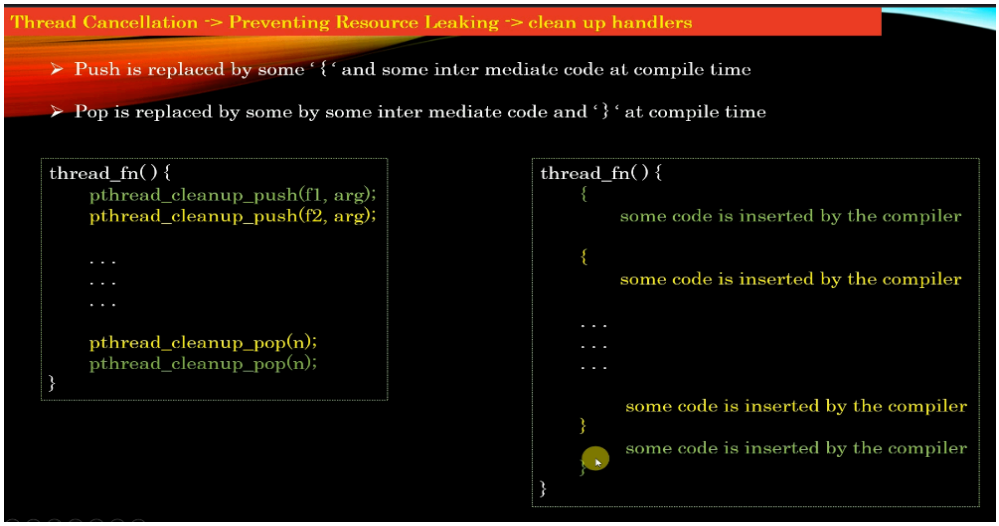
**q).how many cleanup handlers are registered in program?**

ans).Any number of cleanup handlers can be registered within a thread at any point. Each handler performs actions necessary to safely deallocate dynamic memory and free up other system resources. Use pthread_cleanup_pop() to pop the cleanup handler from the cleanup stack once those action are taken care by the thread itself

—--




—--




—--

**pthread_setcancelstate**
**pthread_setcanceltype**
---

```
#include <pthread.h>

       int pthread_setcancelstate(int state, int *oldstate);
       int pthread_setcanceltype(int type, int *oldtype);
```

***pthread_setcancelstate():***

*.A thread's cancelability state, determined by *pthread_setcancelstate(3)*, can be **enabled** (the default for new threads) or disabled.

*.If a thread has disabled cancellation, then a cancellation request remains queued until the thread enables cancellation. If a thread has enabled cancellation, then its cancelability type determines when cancellation occurs.

***pthread_setcanceltype():***

*.A thread's cancellation type, determined by *pthread_setcanceltype(3)*, may be either *asynchronous* or *deferred* (the default for new threads).

*.Asynchronous cancelability means that the thread can be canceled at any time (usually immediately, but the system does not guarantee this).

*.Deferred cancelability means that cancellation will be delayed until the thread next calls a function that is a cancellation point.

*.A list of functions that are or may be cancellation points is provided in *pthreads(7)*.

**PTHREAD_CANCEL_ENABLE**
The thread is cancelable. This is the default cancelability state in all new threads, including the initial thread. The thread's cancelability type determines when a cancelable thread will respond to a cancellation request.

**PTHREAD_CANCEL_DISABLE**
The thread is not cancelable. If a cancellation request is received, it is blocked until cancelability is enabled.

**q).why we go for PTHREAD_CANCEL_DISABLE ?**
**ans).**
***.**Briefly disabling cancelability is useful if a thread performs some critical action that must not be interrupted by a cancellation request.

**q).so, can i put PTHREAD_CANCEL_DISABLE for long run?**
**ans).**
***.**Beware of disabling cancelability for long periods, or around operations that may block for long periods, since that will render the thread unresponsive to cancellation requests.

**PTHREAD_CANCEL_DEFERRED**
(Deferred - వాయిదా)
*.A cancellation request is deferred until the thread next calls a function that is a cancellation point (see *pthreads(7)*). This is the default cancelability type in all new threads, including the initial thread.

**PTHREAD_CANCEL_ASYNCHRONOUS**
*.The thread can be canceled at any time. (Typically, it will be canceled immediately upon receiving a cancellation request, but the system doesn’t guarantee this.)
*.The set-and-get operation performed by each of these functions is atomic with respect to other threads in the process calling the same function.

**q).what about PTHREAD_CANCEL_ASYNCHRONOUS ?**
**ans).**

*.Setting the cancelability type to PTHREAD_CANCEL_ASYNCHRONOUS is rarely useful.
*.Since the thread could be canceled at any time, it cannot safely reserve resources (e.g., allocating memory with *malloc*(3)), acquire mutexes, semaphores, or locks, and so on. Reserving resources is unsafe because the application has no way of knowing what the state of these resources is when the thread is canceled; that is, did cancellation occur before the resources were reserved, while they were reserved, or after they were released? Further-more, some internal data structures (e.g., the linked list of free blocks managed by the malloc(3) family of functions) may be left in an inconsistent state if cancellation occurs in the middle of the function call. Consequently, clean-up handlers cease to be useful.
*.Functions that can be safely asynchronously canceled are called async-cancel-safe functions.

**q).what are async-cancel-safe functions?**
**ans).**
*.POSIX.1-2001 only requires that
pthread_cancel(3),
pthread_setcancelstate(), and
pthread_setcanceltype() be *async-cancel-safe*.

*.In general, other library functions can't be safely called from an asynchronously cancelable thread.

*.One of the few circumstances in which asynchronous cancelability is useful is for cancellation of a thread that is in a pure compute-bound loop.

**q).can i put second argument as NULL in pthread_setcancelstate(state,*old_state*)?**
**ans).**
*.The Linux threading implementations permit the oldstate argument of *pthread_setcancelstate()* to be NULL, in which case the information about the previous cancelability state is not returned to the caller.

---

**pthread_detach:**

---

int pthread_detach(pthread_t tid);

*.**pthread_detach()** does not cause it to terminate.

*.by default, a thread termination status retained untill pthread_join is called for that thread.

*.a thread underlying storage can be reclaimed immediately on termination.

*.when a thread is detached the pthread_join function can’t be used to wait for termination status.

*.a call to pthread_join for a detached thread will fail, returning EINVAL. we can detach a thread a thread by calling pthread_detach.

*.when a thread is detached, the pthread_join function can’t be used to wait for its termination status.

*.once a thread created with detached state, upon termination nothing will be there.

---

**pthread_key_create:**

*.Single-threaded C programs have two basic classes of data—local data and global data. For multithreaded C programs a third class is added—**thread-specific data** (TSD). This is very much like global data, except that it is private to a thread.

*.Thread-specific data is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a key that is *global to all threads in the process*. Using the key, a thread can access a pointer (void *) that is maintained per-thread.

*.Use pthread_key_create(3THR) to allocate a key that is used to identify thread-specific data in a process. The key is global to all threads in the process, and all threads initially have the value NULL associated with the key when it is created.
Call **pthread_key_create()** once for each key before using the key. There is no implicit synchronization.

Once a key has been created, each thread can bind a value to the key. The values are specific to the threads and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the key was created with a destructor function.
Prototype:

```
int   pthread_key_create(pthread_key_t *key,
    void (*destructor) (void *));
#include <pthread.h>

pthread_key_t key;
int ret;

/* key create without destructor */
ret = pthread_key_create(&key, NULL);

/* key create with destructor */
ret = pthread_key_create(&key, destructor);
```

When **pthread_key_create()** returns successfully, the allocated key is stored in the location pointed to by key. The caller must ensure that the storage and access to this key are properly synchronized.
An optional destructor function, destructor, can be used to free stale storage. When a key has a non-NULL destructor function and the thread has a non-NULL value associated with that key, the destructor function is called with the current associated value when the thread exits. The order in which the destructor functions are called is unspecified.
---

**pthread_key_delete:**

Use pthread key delete() to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated and will return an error if ever referenced. There is no comparable function in Solaris threads.
Prototype:

```
int   pthread_key_delete(pthread_key_t key);
#include <pthread.h>

pthread_key_t key;
int ret;

/* key previously created */
ret = pthread_key_delete(key);
```

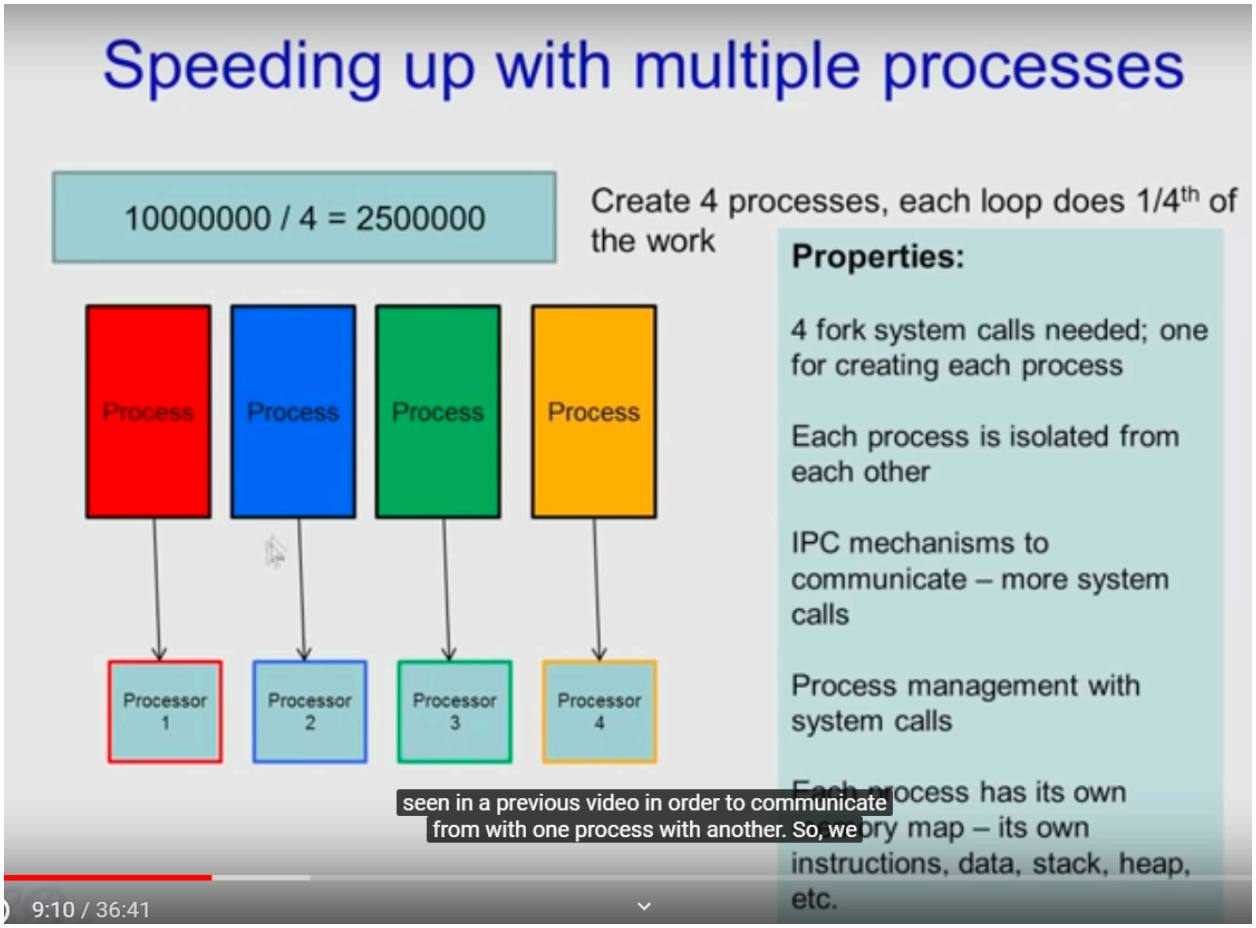
Once a key has been deleted, any reference to it with the **pthread_setspecific()** or **pthread_getspecific()** call yields undefined results. It is the responsibility of the programmer to free any thread-specific resources before calling the delete function. This function does not invoke any of the destructors.

---

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

https://www.youtube.com/watch?v=qkMk3k15PPA&list=PL3-wYxbt4yCjpcfUDz-TgD_ainZ2K3MUZ&index=34

**Process:**




*.Each of these process(multi processes) can execute on different processor in
the system
*.Each of the fork system calls that we invoke would create one of these processes.
*.each process has its own isloated memory map - its own instructions, data,stack,heap,etc.
*.required IPC to communicate between two processes.
*.each process has its own set of instructions,data,heap and the stack. Due to
this isolation we require IPC techniques.

**OVERHEAD-1:**

*.every time one of these processes invokes a system call it results in the Opeating system being executed,being triggered due to a software trap and The operating sytem then determines what system call has invoked and the

Corresponging system call handler will be executed.

**OVERHEAD-2:**

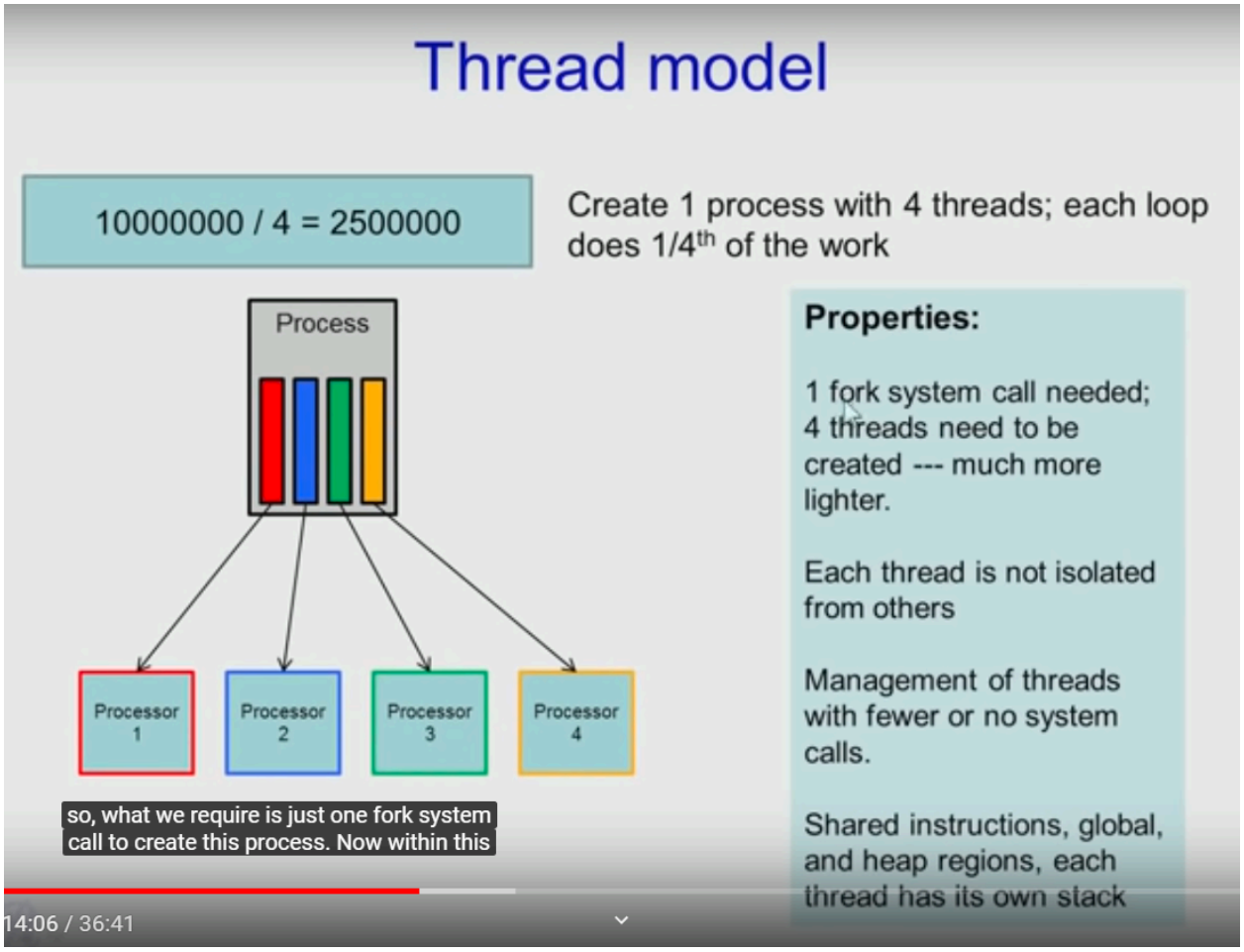
*.for example, all these 4 processes would have the same set of instructions
Which they operate upon and also there could be the same array which they
Are accessing, same global data which they are accessing and therefore,
What we see is that there are a lot of duplication of instructions and
Data which is happening due to having multiple processes whcih execute
A job in parallel.so.

q).Is there a way we can do better than this? so,amount of overhead In
parallelization can be reduced?

ans).

*. We used the concept of *Threading*.

**Thread:**




*.instead of creating 4 processes and whtin this process we create execution
Entity which we call a thread. So, each of these threads quite like the

Previous idea where of using parallelized processes.
*.within this process we have 4 executing context are the 4 threads and they
Execute within the particular process.And each of these threads are executed
In a different processor or typically executed in a different processor.thus
We are able to achieve parallelization by using theads.So, this
Parallelization is quite similar with respect to the process parallelization
However it has a lower overheads
*.