1. Thread overview

A thread (also referred to as a light-weight process LWP) is a basic unit of CPU utilization; it comprises

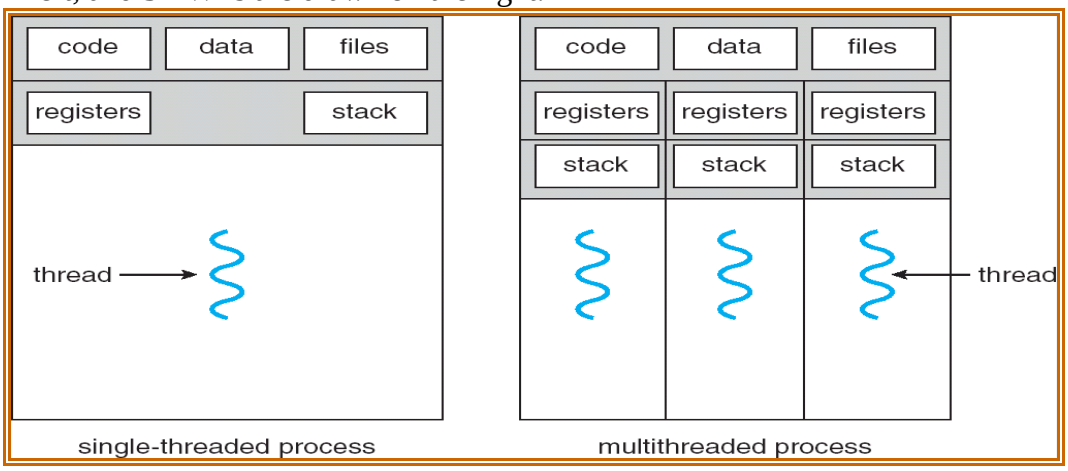
a. a thread ID,
b. a program counter,
c. a register set,
d. and a stack.

- A traditional (or heavyweight) process has a single thread of control.
- All threads in a process have exactly the same address space, which means that they also share the same global variables.
- It shares with other threads belonging to the same process its code section, data section, and other OS resources, such as open files and signals.(see the figure).

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

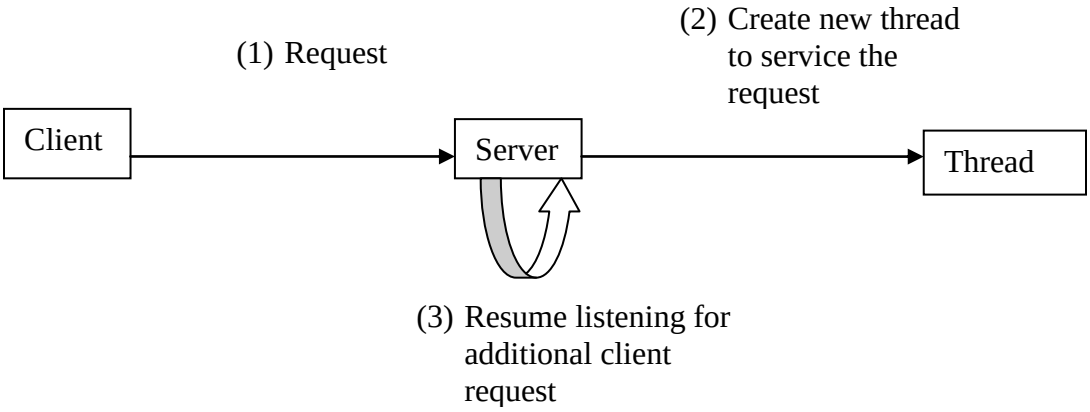
Figure 4.1: The first column lists some items shared by all threads in a process (process properties). The second one lists some items private to each thread.

- Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.
- If a process has multiple threads of control in the same address space running in quasi-parallel, as though they were separate processes (except for the shared address space).
- Multithreading works the same way as the multiple processes does. The CPU switches rapidly back and forth among the threads providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one.
- With three compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with one-third the speed of the real CPU.
- Figure illustrates the difference between a traditional single-threaded process and a multithreaded process. It shows a traditional (or heavyweight) process, on the left, and 3 LWPs are drawn on the right.



1.1 Example:

- Many software packages that run on modern desktop PCs are multithreaded.
- A web browser might have one thread for display images or text and another thread to collect data from network.
- A word process may have a thread for displaying graphics, another thread for responding to keystrokes from user and third one may be for spelling and grammar checking
- A busy webserver may have several clients concurrently accessing it. If the web server is multithreaded, the server will create separate thread that listen for client requests. When a request is made, rather than creating another process, the server will create a new thread to service the request and resume listening for additional request.



- Threads play a vital role in Remote Procedure Call (RPC) system. RPC servers are multithreaded. When a server receives a message, it will service the message using a separate thread. This allows the server to service several concurrent requests.
- Finally most operating system kernels are now multithreaded; several threads operate in the kernel and each thread perform specific task.

1.2 Comparison between Thread and Process

- A process can contain more than one thread.
- A process is considered as "heavyweight" while a thread is deemed as "lightweight".
- Processes are heavily dependent on system resources available while threads require minimal amounts of resource.
- Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately.
    a. The threads share an address space, open files, and other resources.
    b. The processes share physical memory, disks, printers, and other resources.
- Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states. The transitions between thread states are the same as the transitions between process states.
- Since every thread can access every memory address within the process' address space, there is no protection between threads because
    a. it is impossible,
    b. it should not be necessary. They are cooperating, not competing.
- Modifying a main thread may affect subsequent threads while changes on a parent process will not necessarily affect child processes.
- Threads within a process communicate directly while processes do not communicate so easily.
- Threads are easy to create while processes are not that straightforward.

1.3 Benefits of thread

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.
2. **Resource sharing**. By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy of Overheads**. Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.
4. **Utilization of multiprocessor architectures**. The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors (real parallelism).

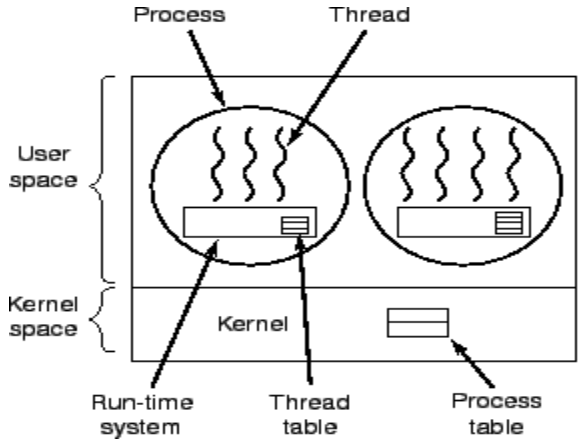
Threads also have drawbacks:

- Writing multithreaded programs requires very careful design. The potential for introducing subtle timing faults, or faults caused by the unintentional sharing of variables in a multithreaded program is considerable.
- Debugging a multithreaded program is much, much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine, unless the calculation truly allows multiple parts to be calculated simultaneously and the machine it is executing on has multiple processor cores to support true multiprocessing.

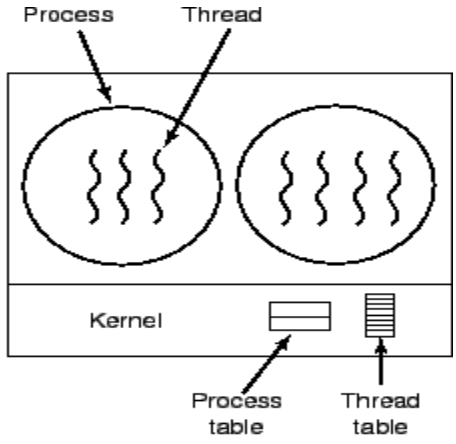
2. Multithreading Models

Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.

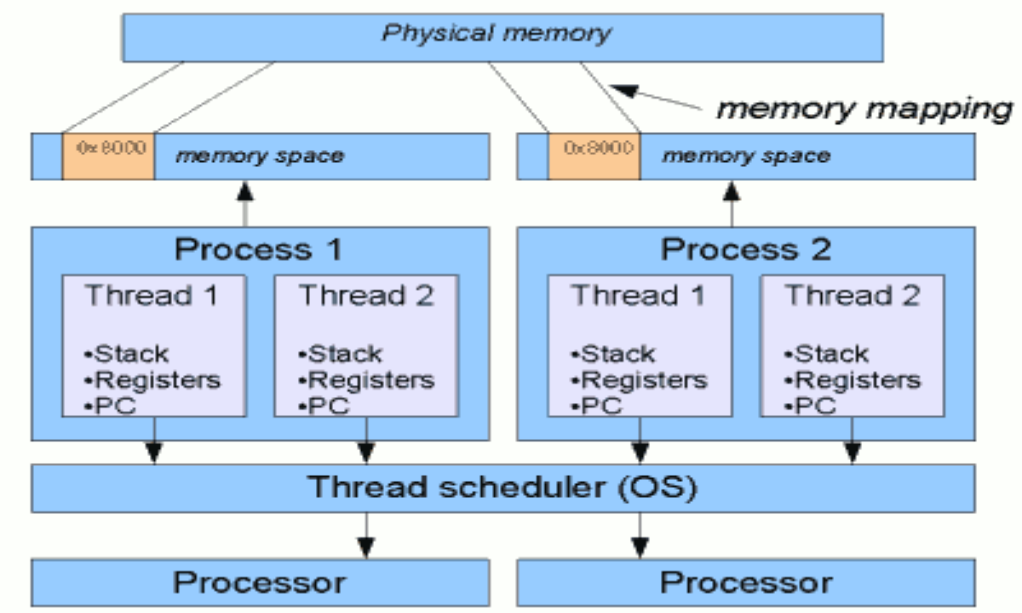
a. User threads are supported above the kernel and are managed without kernel support,
b. whereas kernel threads are supported and managed directly by the OS.



User-level thread package



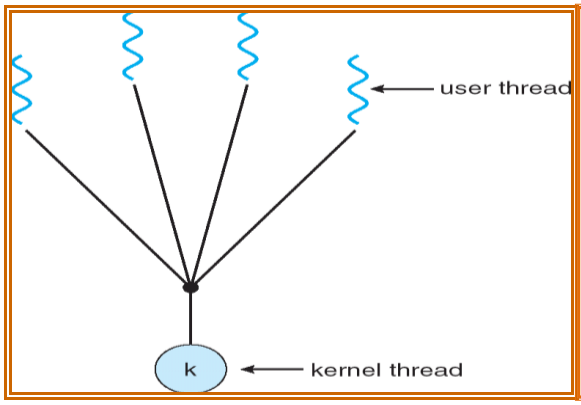
A thread package managed by the kernel



- **Implementing Threads in User Space**:
  - The threads package entirely in user space. The kernel knows nothing about them.
  - The first, and most obvious, advantage is that a user-level threads package can be implemented on an OS that does not support threads.
  - Among other issues, no trap is needed, no context switch is needed. The memory cache need not be flushed, and so on. This makes thread scheduling very fast.
  - Despite their better performance, user-level threads packages have a major problem as if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
  - While user threads usually have lower management load compared to kernel threads, one must consider this in relation to their lower functionality.
- **Implementing Threads in the Kernel**:
  - Supported by the kernel, the kernel performs all management (creation, scheduling, deletion, etc.
  - There is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system.
  - If one thread blocks, another may be run. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads.

2.1 Relationship between user threads and kernel threads

2.1.1 The many-to-one model (see Figurer) maps many user-level threads to one kernel thread.

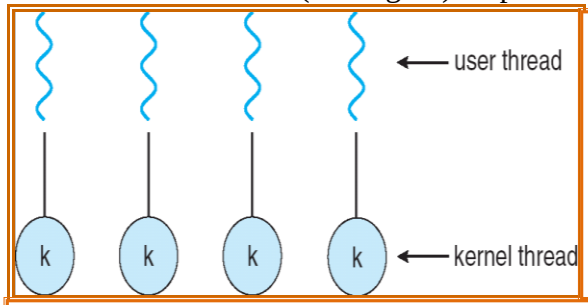


Disadvantage of many to one model:

- Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

2.1.2 One-to-One Model

The one-to-one model (see Figure) maps each user thread to a kernel thread.



Advantage of one to one model:

- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call;
- It also allows multiple threads to run in parallel on multiprocessors.

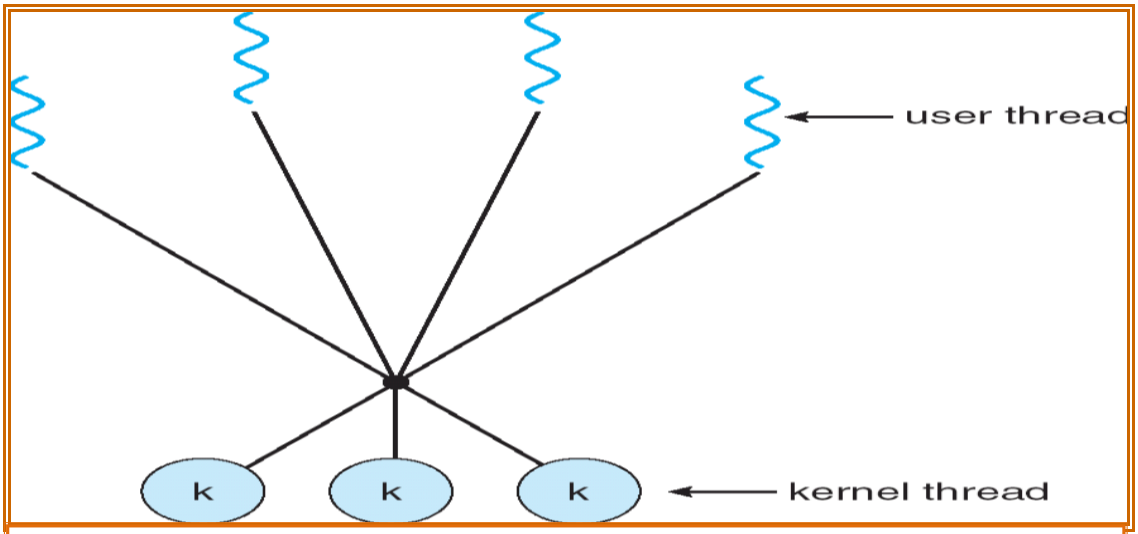
Disadvantage of one to one model:

- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

Example: Linux and family of Windows OSs implement the one-to-one model.

2.1.3 Many-to-Many Model

The many-to-many model (see figure) multiplexes many user-level threads to a smaller or equal number of kernel threads.



Pros and cons of many to many model:

- The number of kernel threads may be specific to either a particular application or a particular machine.
- Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.
- The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create).
- The many-to-many model suffers from neither of these shortcomings:
    - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
    - Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

3. Thread Libraries

- A thread library provides the programmer an API for creating and managing threads. There are two primary ways of implementing a thread library.
  - The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
  - The second approach is to implement a kernel-level library supported directly by the OS. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.
  - Three main thread libraries are in use today:
    1. **POSIX Pthreads**: Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
    2. **Win32:** The Win32 thread library is a kernel-level library available on Windows systems.
    3. **Java**: The Java thread API allows thread creation and management directly in Java programs. However, because in most instances the JVM is running on top of a host OS, the Java thread API is typically implemented using a thread library available on the host system.

4. Pthreads

In Operating system laboratory we will use mostly the Pthreads
IEEE POSIX committee published some standards which are widely available in UNIX-like operating systems, and the implementations with the advent of the POSIX 1003.1c specification, threads are not only better standardized, but are also available on most Linux distributions. It's important to be clear about the difference between the fork system call and the creation of new threads. When a process executes a fork call, a new copy of the process is created with its own variables and its own PID. This new process is scheduled independently, and (in general) executes almost independently of the process that created it. When we create a new thread in a process, in contrast, the new thread of execution gets its own stack (and hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with the process that created it.

There is a whole set of library calls associated with threads, most of whose names start with pthread. To use these library calls, you must define the macro _REENTRANT, include the file pthread.h, and link with the threads library using -lpthread.

CASE1: Example of Multithreaded process:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/types.h>
#include<pthread.h>
void *string( void *arg);
int sum(int arg1);
int system(const char *string);
char msg[ ]="Techno India";
int sum1=0;
int main()
{
int res1,res2;
pthread_t tid1,tid2;
```

```
void *result1,*result2;
res1=pthread_create(&tid1,NULL,string,(void *)msg);
res2=pthread_create(&tid2,NULL,sum,(int )sum1);
if(res1!=0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
printf("\nWaiting for thread to finish.....\n");
res1=pthread_join(tid1,&result1);
if(res1!=0)
{
perror("Thread join failed\n");
exit(EXIT_FAILURE);
}
printf("Thread1 joined, it returned %s\n",(char *)result1);
printf("\nMessage is now %s\n",msg);
if(res2!=0)
{
perror("Thread joined fail");
exit(EXIT_FAILURE);
}
printf("\n waiting for thread to finish.........\n");
res2=pthread_join(tid2,&result2);
if(res2!=0)
{
perror("Thread joined failed");
exit(EXIT_FAILURE);
}
printf("\nThread2 jouned, it returned %s",(char *)result2);
printf(" %d\n",sum1);
system("ps -ax");
exit(EXIT_SUCCESS);
}
void *string(void *arg)
{
printf("Thread funtion is running.Argument was %s\n",(char *)arg);
sleep(3);
strcpy(msg,"Soumitra is good boy");
printf("\nThread ID=%u",(unsigned int)pthread_self());
pthread_exit("Thank you for the CPU time");
}
int sum(int arg1)
{
int i=6;
int j=8;
sum1=i+j;
printf("\nThread ID=%u",(unsigned int)pthread_self());
pthread_exit("The addition of two number is:");
}
```

To run a thread programme:

]$ gcc –D_REENTRANT thread1.c –o thread1 –lpthread

]$ ./thread1

How it work:

You declare a prototype for the function that the thread will call when you create it:
void *thread_function(void *arg);
As required by pthread_create, it takes a pointer to void as its only argument and returns a pointer to void.

In main, you declare some variables and then call pthread_create to start running your new thread.
pthread_t tid1,tid2;
void *result1, *result2;
res = pthread_create(tid1, NULL, string, (void *)msg);
You pass the address of a pthread_t object that you can use to refer to the thread afterward. You don't want to modify the default thread attributes, so you pass NULL as the second parameter. The final two parameters are the function to call and a parameter to pass to it.
If the call succeeds, two threads will now be running. The original thread (main) continues and executes the code after pthread_create, and a new thread starts executing in the imaginatively named string.
The original thread checks that the new thread has started and then calls pthread_join.
Res1 = pthread_join(tid1, &result1);
Here you pass the identifier of the thread that you are waiting to join and a pointer to a result. This function will wait until the other thread terminates before it returns. It then prints the return value from the thread and the contents of a variable, and exits.
The new thread starts executing at the start of string, which prints out its arguments, sleeps for a short period, updates global variables, and then exits, returning a string to the main thread. The new thread writes to the same array, message, to which the original thread has access. If you had called fork rather than pthread_create, the array would have been a copy of the original message, rather than the same array.
The same way we have created another thread with thread identifier tid1 and calling the sum function.
Another system call pthread_self() will print the thread ID.
All processes in the same thread group have the same PID, but each one has a unique TID.

CASE2: Simultaneous Execution of two thread

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<pthread.h>
void *string(void *arg1);
int run=1;
int main()
{
int res;
pthread_t tid1;
void *res1;
int j=0;
res=pthread_create(&tid1,NULL,string,(int )run);
printf("Waiting for the thread to complete..........\n");
if(res!=0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
while(j++<10)
{
if(run==1)
{
printf("The main program running=%d",run);
run=2;
}
else
```

```
sleep(1);
}
res=pthread_join(tid1,&res1);
printf("\n\nThread joind, it returned%s",(char *)res1);
exit(EXIT_SUCCESS);
}
void *string(void *arg)
{
int i=0;
while(i++<10)
{
if(run==2)
{
printf("\tThe thread is running=%d\n",run);
run=1;
}
else
sleep(1);
}
pthread_exit("Thank you for the CPU time\n\n");
}
```

You may find that it takes a few seconds for the program to produce output, particularly on a single-core CPU machine.)Each thread tells the other one to run by setting the run variable and then waits till the other thread has changed its value before running again. This shows that execution passes between the two threads automatically and again illustrates the point that both threads are sharing the run variable.

5. The fork ( ) System Calls

- If one thread in a program calls fork( ),
  - does the new process duplicate all threads,
  - or is the new process single-threaded?
- Some UNIX systems have chosen to have two versions of fork( ),
  - one that duplicates all threads
  - and another that duplicates only the thread that invoked the fork( )system call.
- Which of the two versions of fork( )to use depends on the application.

6. Cancellation of Thread

- **Thread cancellation** is the task of terminating a thread before it has completed.
  - For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
  - Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.
- A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.
   - The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.
   - Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.
2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

- With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not.
- This allows a thread to check whether it should be canceled at a point when it can be canceled safely. Pthreads refers to such points as cancellation points.

Example of deferred cancelation:

int pthread_cancel(pthread_t thread);

This is pretty straightforward: Given a thread identifier, you can request that it be canceled.

int pthread_setcancelstate(int state, int *oldstate);

The first parameter is either PTHREAD_CANCEL_ENABLE, which allows it to receive cancel requests. The oldstate pointer allows the previous state to be retrieved. If you are not interested, you can simply pass NULL.

int pthread_setcanceltype(int type, int *oldtype);

The type can take one of two values, PTHREAD_CANCEL_ASYNCHRONOUS, which causes cancellation requests to be acted upon immediately, and PTHREAD_CANCEL_DEFERRED, which makes cancellation requests wait until the thread executes one of these functions: pthread_join.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
void *cancel(void *arg);
int main()
{
pthread_t tid1;
void *result;
int res;
res=pthread_create(&tid1,NULL,cancel,NULL);
if(res!=0)
{
perror("Thread creation failed");
exit(EXIT_FAILURE);
}
sleep(5);
printf("Cancelling the thread.............\n");
res=pthread_cancel(tid1);
if(res!=0)
{
perror("Thread cancelation failed");
exit(EXIT_FAILURE);
}
printf("Waiting for the thread to finish.........\n");
res=pthread_join(tid1,&result);
if(res!=0)
{
perror("Thread join failed");
exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
void *cancel(void *arg)
{
int i,res;
res=pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,NULL);
if(res!=0)
{
```

```
perror("PThread cancelsate failed");
exit(EXIT_FAILURE);
}
res=pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
if(res!=0)
{
perror("PThread cancelsate failed");
exit(EXIT_FAILURE);
}
printf("Thread is running\n");
for(i=0;i<=10;i++)
{
printf("Thread is still running (%d)...\n",i);
sleep(2);
}
pthread_exit(0);
}
```

Up until now, we have always had the normal thread of execution of a program create just one other thread. Many threads can also be created.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<sys/types.h>
#include<math.h>
void *multithread(void *arg);
int main()
{
int i,res,n;
void *result;
printf("\n enter the no of thread you want create");
scanf("%d",&n);
pthread_t tid[n];
for(i=0;i<n;i++)
{
res=pthread_create(&tid[i],NULL,multithread,(void *)i);
if(res!=0)
{
perror("Thread creation failed");
exit("EXIT_FAILURE");
}
//sleep(1);
}
printf("\nWaiting for thread to finish.......");
for(i=n-1;i>=0;i--)
{
res=pthread_join(tid[i],&result);
if(res==0)
{
printf("Picked up a thread %d\n",(int *)result);
}
else
{
perror("Thread join failed");
}
}
printf("All done\n");
exit(EXIT_SUCCESS);
}
void *multithread(void *arg)
```

```
{
int j=(int)arg;
//int l;
printf("\nThread funtion is running argument was %d\n",j);
//l=1+(int)(9.0*rand()/(RAND_MAX+1.0));
//sleep(l);
printf("Bye from %d\n",j);
pthread_exit(j);
}
```

Try this program also removing the block portions.