Course	Topic: SCHEDULER And THREADS	
Module Day 5		6 Hours
	At the end of this module you will be able to know: ▪ Scheduler ▪ Scheduling criteria ▪ Types of scheduler ▪ Scheduling algorithms ▪ nice() system call ▪ What is thread? ▪ Working with threads. ▪ System calls related to threads	

Session Plan	
1	Introduction to Scheduler
2	Scheduling Criteria and types of Scheduler
3	Different Scheduling Algorithms
4	Introduction to Threads
5	Working with threads
6	System calls related to threads

Module Overview

- ✓ Scheduler and its types
- ✓ Scheduling Criterias
- ✓ Different types of Scheduling Algorithms
- ✓ nice() system call
- ✓ Introduction to Threads
- ✓ Working with Threads
- ✓ Sytem calls related to Threads

Introduction to Scheduling techniques

The process scheduler decides which process runs, when, and for how long.

The process scheduler (or simply the *scheduler*) divides the finite resource of processor time between the runnable processes on a system.

The scheduler is the basis of a multitasking operating system such as Linux.

By deciding which process runs next, the scheduler is responsible for best utilizing the system and giving users the impression that multiple processes are executing simultaneously deciding how use the processor's time on the computer.

In general, the scheduler runs:

When a job switches states (running, waiting, etc.)
When an interrupt occurs
When a job is created or terminated.

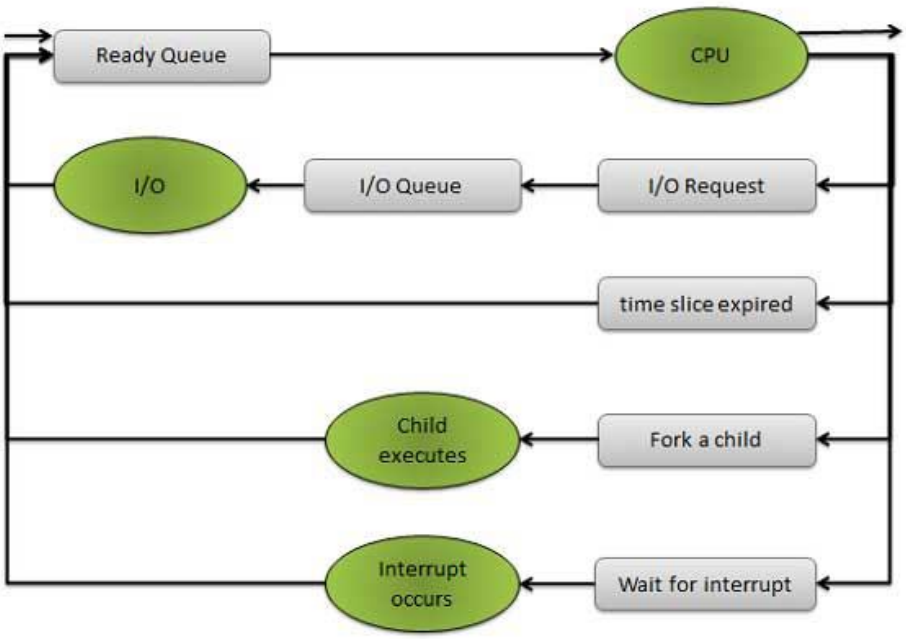


Diagram shows the process scheduling in Unix

Dispatcher:

The dispatcher in Operating System is a module that selects the process from the ready queue for allotting it the CPU (Processor). There is a switch associated during dispatching and that is the *process status changes from ready to running*. The

dispatcher is placed in between the ready queue and Processor Scheduler (i.e. short term scheduler).

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

- switching context
- switching to user mode
- jumping to the proper location in the user program to restart that program

Dispatch latency – time it takes for the dispatcher to stop one process and start another running

Scheduling criteria:

- CPU utilization – keep the CPU as busy as possible
- Throughput – amount of work done by CPU per time unit
- Turnaround time (TAT) – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)
- Then, it loads the registers from the saved context of the new process
- p_addr field in the proc structure points to the page table entries of the uarea and is used by swtch to locate the new pcb

Types of schedulers:

1. **Long-term scheduler:**
   - Selects process and loads it into memory for execution
   - Decides which process to start based on order and priority
   - Not used in timesharing systems
2. **Medium-term scheduler:**
   - Schedule processes based on resources they require (memory, I/O)
   - Suspend processes for which adequate resources are not currently available
   - Commonly, main memory is the limiting resource and the memory manager acts as the medium term scheduler
3. **Short-term scheduler (CPU scheduler):**
   - Shares the processor among the ready (runnable) processes
   - Crucial the short-term scheduler be **very** fast -- a fast decision is more important than an excellent decision

- If a process requires a resource (or input) that it does not have, it is removed from the ready list (and enters the WAITING state)
- Uses a data structure called a **ready list** to identify ready processes
- Started in response to a clock interrupt or when a process is suspended or exits .

Scheduling algorithms:

First come first serve algorithm(FCFS**)**

Other names of this algorithm are:

- First-In-First-Out (FIFO)
- Run-to-Completion
- Run-Until-Done

Perhaps, First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a nonpreemptive discipline, once a process has a CPU, it runs to completion.

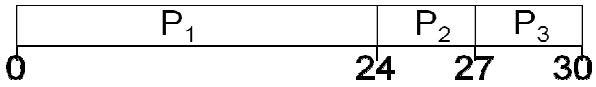
FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawback of this scheme is that the average time is often quite long.

The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.

Example:

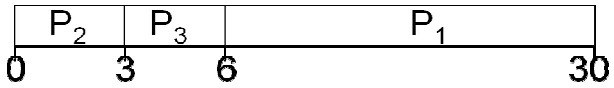
Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order: P_2 , P_3 , P_1 .



Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
Average waiting time: (6 + 0 + 3)/3 = 3

Short-Job-First-Serve Algorithm(SJFS):

Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

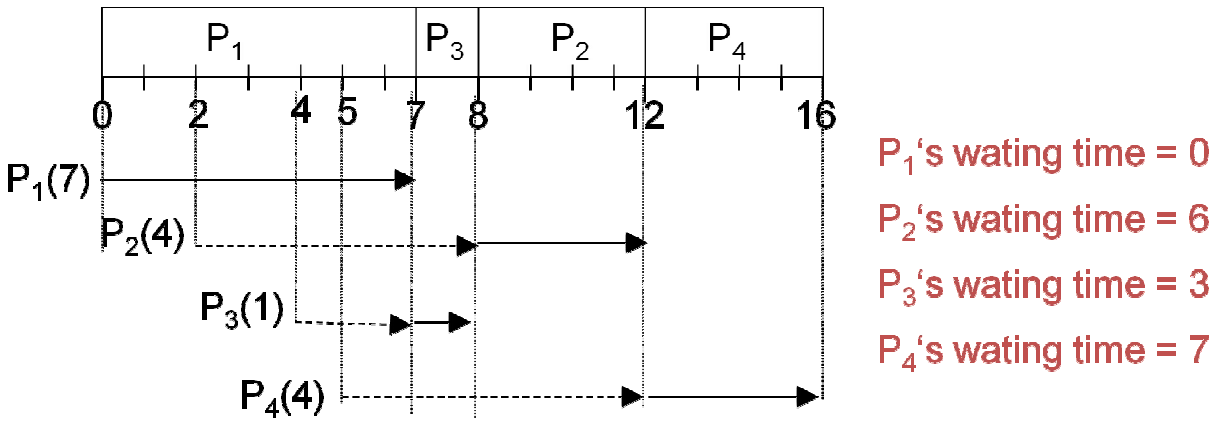
The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.

The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.

Example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Non preemptive SJF

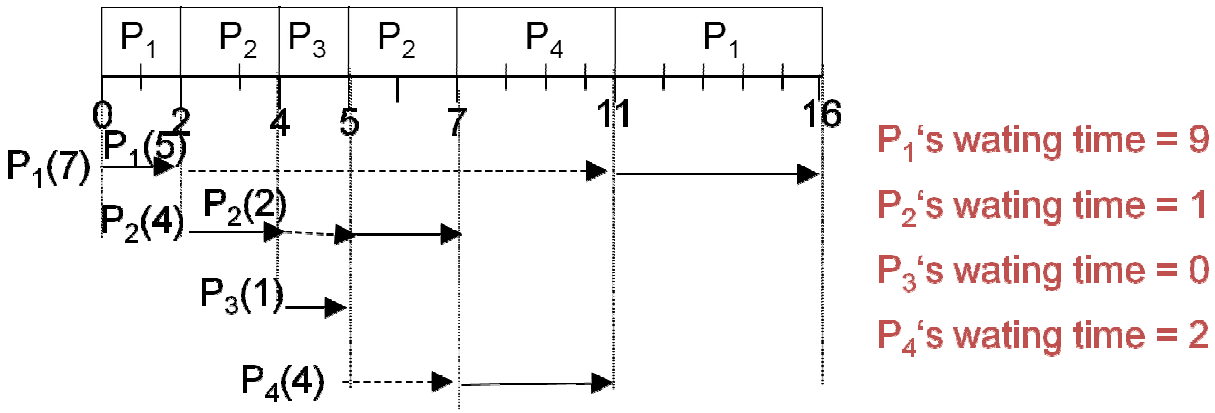


Average waiting time = (0 + 6 + 3 + 7)/4 = 4

- Preemptive SJF

Example:

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4



Average waiting time = (9 + 1 + 0 +2)/4 = 3

Like FCFS, SJF is non pre-emptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

Priority scheduling:

The basic idea is straightforward:

> Each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa.

Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process.

Examples of Internal priorities are

- Time limits.
- Memory requirements.
- File requirements,
  for example, number of open files.
- CPU Vs I/O requirements.

Externally defined priorities are set by criteria that are external to operating system such as

- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

Priority scheduling can be either preemptive or non preemptive

- A pre-emptive priority algorithm will pre-emptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-pre-emptive priority algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is *aging*.

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

Round Robin Scheduling(RR):

One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).

In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.

If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.

Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.

The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

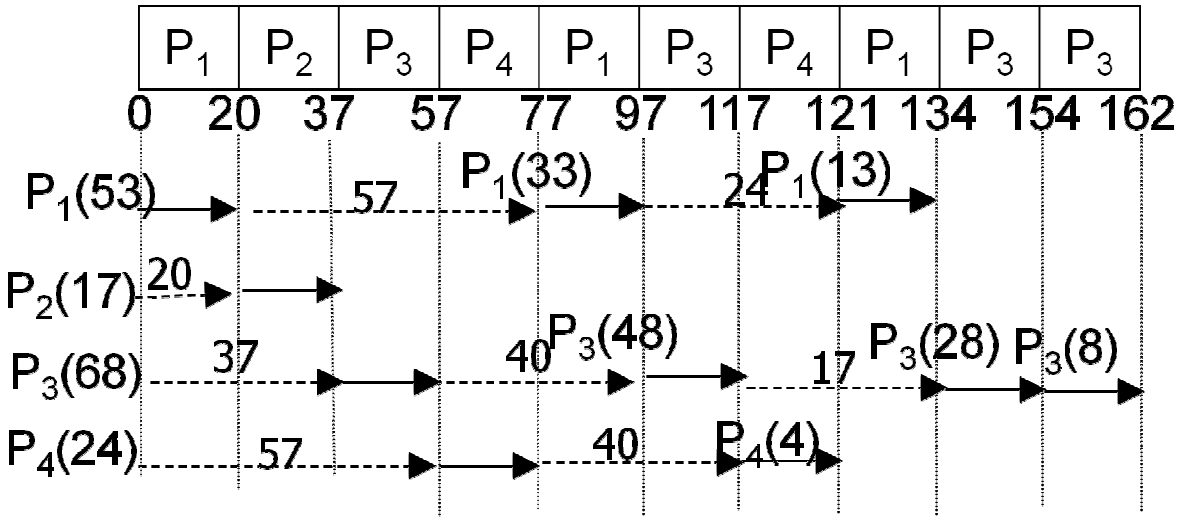
In any event, the average waiting time under round robin scheduling is often quite long.

If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.

Example:

Time quantum = 20

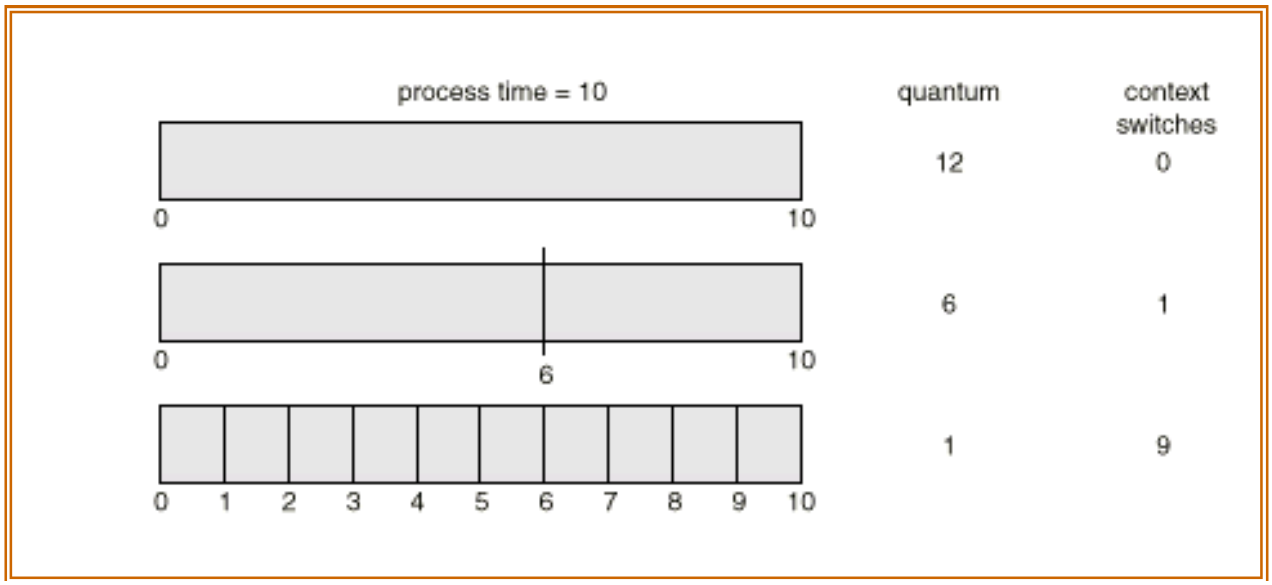
Process	Burst Time	Wait Time
P_1	53	57 +24 = 81
P_2	17	20
P_3	68	37 + 40 + 17= 94
P_4	24	57 + 40 = 97



Average wait time = (81+20+94+97)/4 = 73

Typically, higher average turnaround than SJF, but better *response*.

Performance:
q large ⇒ FCFS
q small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.



`nice()` system call:

Processes can exercise a crude control of their scheduling priority by using the nice() system call, it takes an integer value as argument which is added to the priority of the process.

For ex:

```
nice [OPTION] [COMMAND [ARG]...]
```

Run `COMMAND` with an adjusted niceness, which affects process scheduling. With no COMMAND, print the current niceness. Nicenesses range from -20 (most favorable scheduling) to 19 (least favorable).

NOTE: your shell may have its own version of nice, which usually supersedes the version described here. Please refer to your shell's documentation for details about the options it supports.

Introduction to Threads

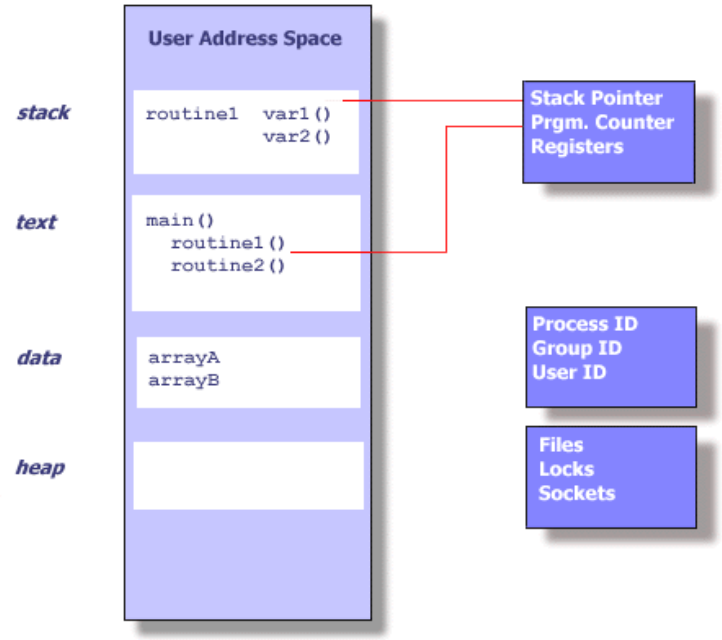
- Threads are like processes.
- It is a mechanism to allow a program to do more than one thing at a time.
- As with processes, threads appear to run concurrently. The Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute.
- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
- Thread exists within a process and uses the process resources.
- A thread has its own independent flow of control as long as its parent process exists and the OS supports it.
- It may share the process resources with other threads that act equally independently.
- A thread dies if the parent process dies.
- A process can have multiple threads, all of which share the resource within a process and all of which execute within the same address space.
- Threads in the same process share:
    - Process instructions
    - open files (descriptors)
    - signals and signal handlers

- current working directory
- User and group id

- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority

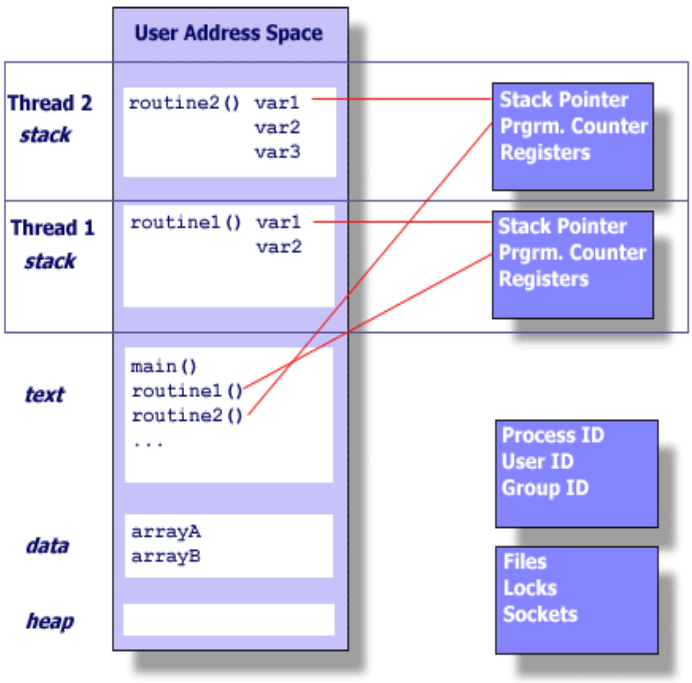
- **Process Vs Threads:**

Process:



As shown in the above diagram, assumes that in a process there are two routines called routine1() and routine2() one by one. Now routine2() has to execute only when routine1() is completed.

Threads:



As shown in above diagram, by using threads we can execute two subroutines in a process at a time. Internally these two will be switching from one thread to another. So by using threads in a process, execution will become faster, when you compare a process without threads.

System Calls related to Threads

Required Header files:

#include<pthread.h>

Creating a Thread: Threads can be created, by using "pthread_create()" system call. You have to pass a function address to this call. The prototype is given below.

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

- This function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(). 'arg' is passed as the sole argument of start_routine().
- The new thread terminates in one of the following ways:
  - It calls pthread_exit(), specifying an exit status that is available to another thread in the same process that calls pthread_join().
  - It returns from the start_routine(). This is equivalent to calling pthread_exit() with the values supplied in the return statement.

- It is cancelled by calling pthread_cancel() function.
- The main thread performs a return from main(). This causes the termination of all threads in the process.

- 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. This structure is initialized using pthread_attr_init() and related functions. If attr is NULL, then the thread is created with default attributes.
- Before using pthread_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 pthread functions.
- On success, this call returns 0, on error it returns –ve number.

Ex: /******** WAP to create a thread **********/

```
void *function(void *unused)
{
     while(1)
          fputc('x', stderr);
}

int main()
{
     pthread_t thread_id;
     pthread_create(&thread_id, NULL, &function, NULL);
     while(1)
          fputc('y', stderr);
     return 0;
}
```

OUTPUT:

```
xxxxxxxxyyyyyyyyyxxxxxxxyyyyyyyxxxxxyyyyyyxxxx…….continuously
```

Explanation: In this program, we are creating thread and assigning work as function(). Now both main and function are executed at a time. To wait main process for thread to exit, then we have to use pthread_join().

pthread_join(): This function waits for the thread specified by thread to terminate. If that thread has already terminated, this returns immediately. The prototype is given below:

```
int pthread_join( pthread_t thread, void **retval);
```

- If the retval is not NULL, then pthread_join() copies the exit status of the target thread into the location pointed to by *retval. If the target location was cancelled, then PTHREAD_CANCELED (-1) is placed in *retval.
- On success this system call returns 0, and on error it returns –ve number.

pthread_exit(): This system call terminates the calling thread and returns a value via retval that is available to another thread in the same process that calls pthread_join(). The prototype is given below:

void pthread_exit(void *retval);

This function will never fail and never return back to calling function.

Ex: /******** WAP to create a thread and join that thread ************/

```
void *function(void *unused)
{
     static int a = 0;
     while(a < 10) {
          a++;
          fputc('x', stderr);
     }
     pthread_exit(&a);
}

int main()
{
     pthread_t thread_id;
     int *ptr;

     pthread_create(&thread_id, NULL, &function, NULL);
     pthread_join(thread_id,  (void *)&ptr);

     printf("\n*ptr = %d\n", *ptr);

     while( i < 10) {
          fputc('y', stderr);
          i++;
     }
     return 0;
}
```

OUTPUT:

```
xxxxxxxxxx
*ptr = 10
yyyyyyyyyy
```

pthread_setcancelstate(): The prototype is given below:

int pthread_setcancelstate(int state, int *oldstate);

- This system call sets the cancelability state of the calling thread to the value given in state.

- The previous cancelability state of the thread is returned in the buffer pointed to by oldstate. The state argument must have one of the following values:
  - PTHREAD_CANCEL_ENABLE
    The thread is cancelable. This is the default cancelability state in all new threads.
  - PTHREAD_CANCEL_DISABLE
    The thread is not cancelable. If a cancellation request is received, it is blocked until cancelation is enabled.
- This system call returns 0 on success and non zero (error number) on failure.

pthread_cancel(): The prototype is given below:

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

- This system call sends a cancellation request to the thread specified by thread argument.
- If a thread has disabled cancellation, then a cancellation request remains queued until the thread enables cancellation.
- If a thread has enabled cancellation, then thread will be cancelled.
- This system call returns 0 on success, and non zero (error number) on failure.

Ex: /******** WAP to show thread cancellation **********/

```
#include<stdio.h>
#include<pthread.h>

pthread_t thread, thread1;

void function(void *unused)
{
        while(1)  {
                sleep(1);
                printf("\none\n");
        }
}
void func(void *waste)
{
        int i = 0;
        while(i < 3)  {
                i++;
                sleep(1);
                printf("\ntwo\n");
        }
        pthread_cancel(thread);

}
```

```
int main()
{
        int i = 0, *ptr;

        pthread_create(&thread, NULL, function, NULL);
        while(i < 3)  {
                i++;
                sleep(1);
                printf("\nmain\n");
        }
        pthread_create(&thread1, NULL, func, NULL);
        printf("\n wait till threads exits \n");
        pthread_join(thread, (void *)&ptr);
        pthread_join(thread1, (void *)&ptr);
        return 0;
}
```

OUTPUT: **/* may vary */**

```
main
one
main
one
main
one
one
wait till threads exits
two
one
two
one
two
```