Course	**Process Management System (PMS)**	
Module Day 3		**3 Hours**
	At the end of this module you will be able to know: ▪ What is process and process anatomy? And what is the role of PMS? ▪ What is task_struct structure? ▪ Different process states ▪ Process table and u-area ▪ Process context switching ▪ What are the different types of processes? ▪ System calls related to processes ▪ Killing a process.	

Session Plan	
1	Introduction to PMS: what is process; process anatomy, task_struct structure, process states, and different types of processes.
2	Process table, process context switch, system calls related to process and code examples.
3	Use of Exec functions

Module Overview

- ✓ Introduction to PMS
- ✓ Process Anatomy
- ✓ task_struct Structure
- ✓ Process states
- ✓ Process table
- ✓ Types of Process
- ✓ System calls related to process
- ✓ Exec system calls
- ✓ Killing a process

Introduction to Process Management System (PMS)

Process: Code under execution is a process.

Each process in a Linux system is identified by its unique *process ID* sometimes referred to as *PID*. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

The *parent process ID*, or *ppid*, is simply the process ID of the process's parent

For every running process task_struct data structure will be created, and this will be having all information about process (PCB – Process Control Block).

Each terminal window is running a shell; each running shell is another process. When you invoke a command from a shell, the corresponding program is executed in a new process; the shell process resumes when that process completes.

Process Management System: Process management system is responsible for the process synchronization , IPC , Memory management and process scheduling.
The file subsystem and process subsystem interact when loading file into memory for execution. Process management system is used to maintain and control the different processes. This will be maintaining the double linked list of task_struct structures of all processes.

Process Anatomy

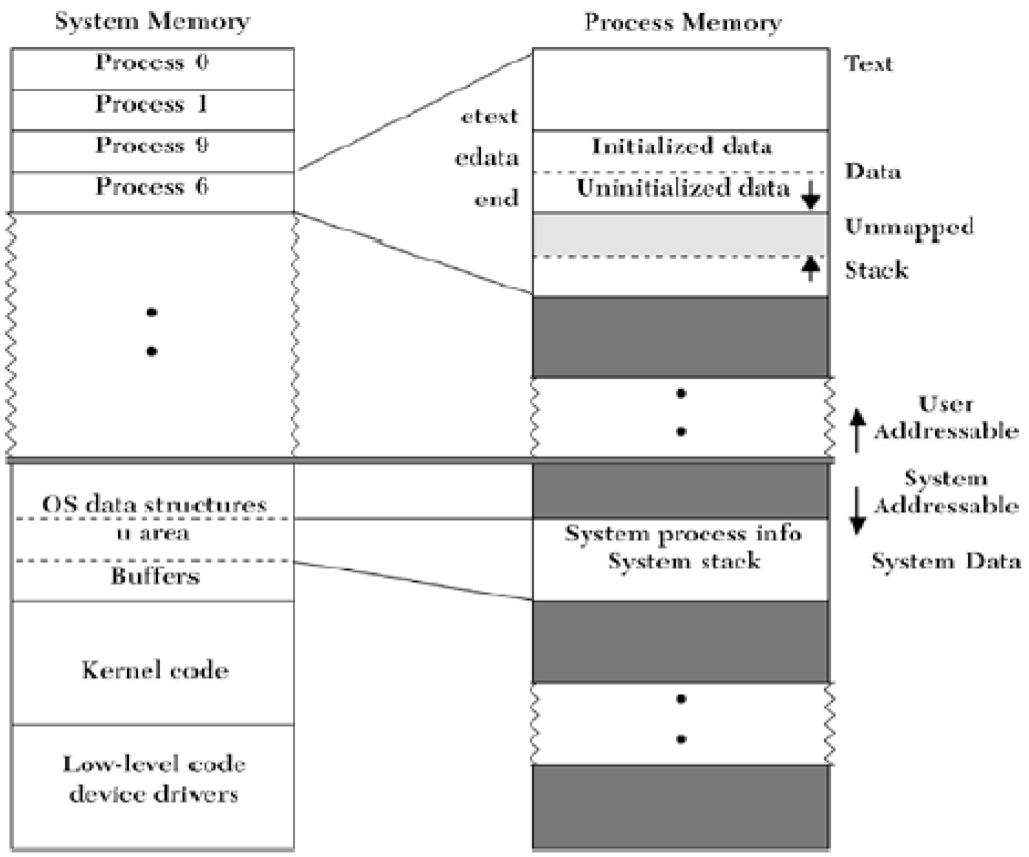
When you invoke any command in shell, that will create a process.

- Text segment— The text segment (sometimes called the instruction or code segment) contains the executable program code and constant data. The text segment is marked by the operating system as read-only and cannot be modified by the process. Multiple processes can share the same text segment.

- Data segment— The data segment, which is contiguous (in a virtual sense) with the text segment, can be subdivided into initialized data (e.g., in C/C++, variables that are declared a static) and uninitialized data. During its execution lifetime, a process may request additional data segment space. Library memory allocation routines (e.g., new, malloc, calloc, etc.) in turn make use of the system calls `brk` and `sbrk` to extend the size of the data segment. The newly allocated space is added to the end of the current uninitialized data area.

- stack segment— The stack segment is used by the process for the storage of automatic identifiers, register variables, and function call information. As needed, the stack segment grows toward the uninitialized data segment. The area beyond the stack contains the command-line arguments and environment variables for the process. The actual physical location of the stack is system-dependent.

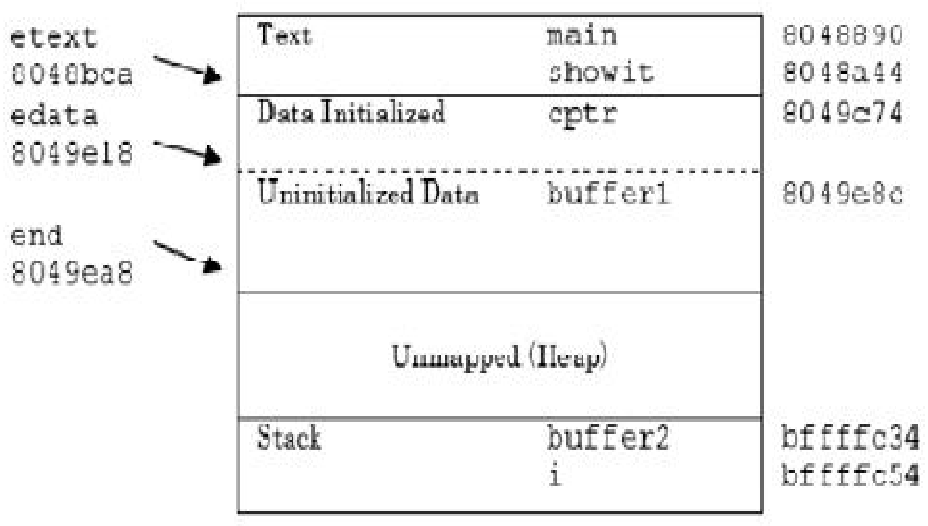
Process Memory



Process Memory Addresses

The system keeps track of the virtual addresses associated with each user process segment. This address information is available to the process and can be obtained by referencing the external variables etext, edata, and end. The addresses (not the contents) of these three variables correspond respectively to the first valid address above the text, initialized data, and uninitialized data segments.

Logical addresses — calculated and used without concern as to their actual physical location.



task_struct Structure

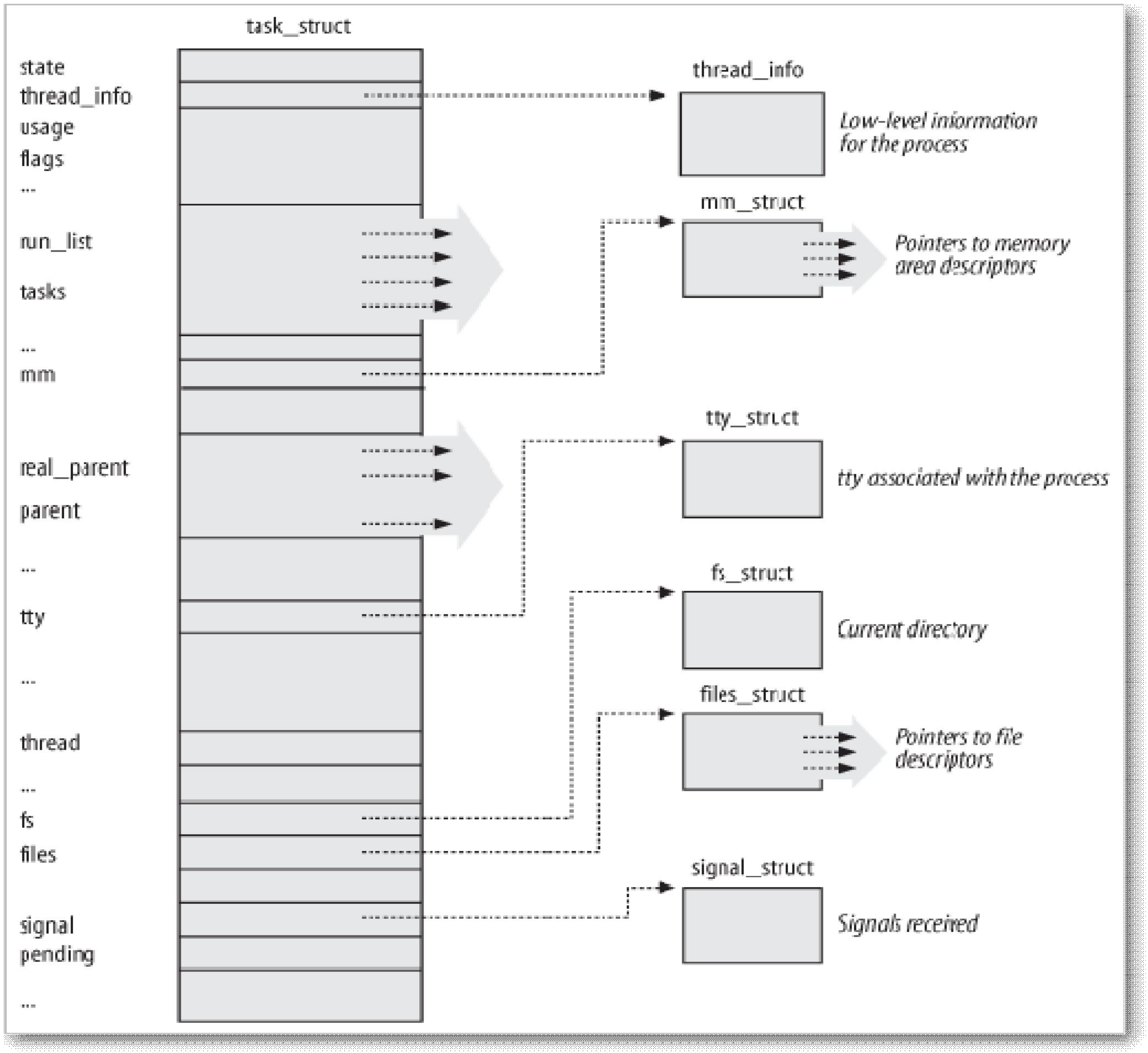
For every process in Linux, one task_struct structure will be created. All these structures will be maintaining as double linked list.

The task_struct is a relatively large data structure, at around 1.7 kilobytes on 32-bit machine. The task structure contains the data that describes the executing program: open files, the process's address space, pending signals, the process's state, and much more.

task_struct structure:

```
struct task_struct {
        volatile long state;          /* -1 unrunnable, 0 runnable, >0 stopped */
        long priority;
        int pid;
        struct tty_struct *tty;       /* NULL if no tty */
        - - -
        - - -
        struct fs_struct *fs;         /* pointer to open files Structure */
        struct signal_struct *sig;     /* signal handlers */

};
```



fs_struct structure:

```
struct fs_struct {
        int count;
        unsigned short umask;
        struct inode * root, * pwd;
};
```

For more information on this task_struct structure refer the following link:

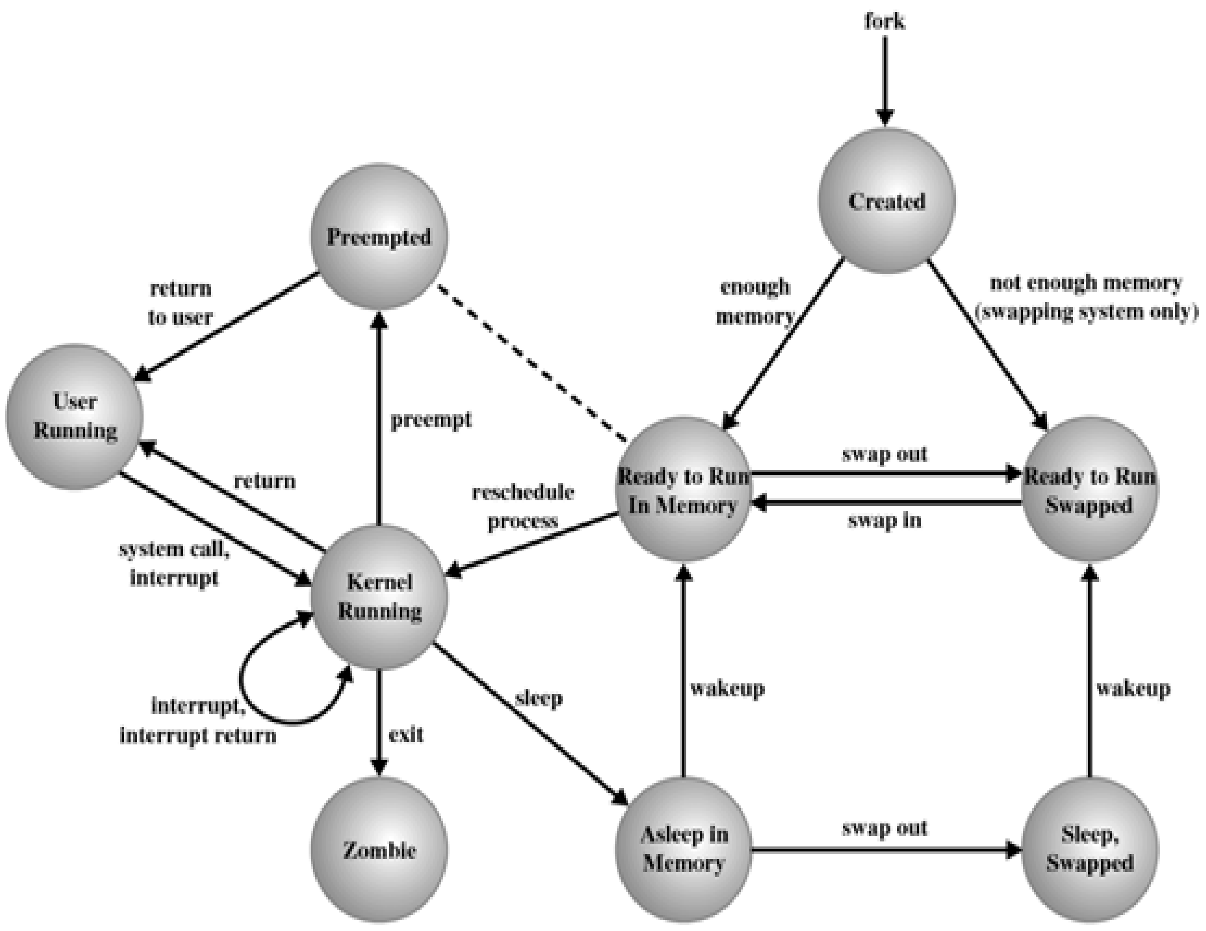
http://students.mimuw.edu.pl/SO/LabLinux/PROCESY/ZRODLA/sched.h.html

Process states

As a process executes, it changes its state according to its circumstances. In Linux a process may have the different states. The below diagram shows, how process states will be changed, and what are the different process states.

As a process executes, it changes its state according to its circumstances. In Linux a process may have the different states.

1. The above diagram shows, how process states will be changed. The different process states are:
2. Process is executing in user mode.
3. Process is executing in kernel mode, when user makes a system call.
4. Process is not executing ,it will be in ready to run as soon kernel schedules it.
5. The process is sleeping but its still in main memory.
6. Process is ready to run, but swapper must swap the process to main memory before kernel schedule it to execute.
7. The process is sleeping, The swapper must swap the process from main memory to secondary memory, to make space for other processes in main memory.
   - **Preempt:** It involves the use of interrupt mechanism, which suspends the executing process and invokes the scheduler to determine which process should execute it
8. The process is created using fork system call , and its not ready to run , not ready to sleeping.
9. Once the process executed the exit system call and is in zombie state. That no more exists, But leaves status containing exit code and some timing statistics for parent process.
10. If any interrupt signal generated to process, process has to execute the corresponding ISR and resumes back.



UNIX Process State Transition Diagram

Process table and User area

The state of the process can be described in Two kernel data structure, Those are:

<u>Process table:</u> The process table in Linux is simply a data structure in the RAM of a computer. It holds the information about the processes that are currently handled by the OS. This information includes general information about each process:

- **Process state:** The process table contains the fields that must always be accessible to the kernel.
    - o Running in user mode or kernel mode

    - Ready in memory or Ready but swapped
    - Sleep in memory or sleep and swapped
- **PID:** Process id (specify the relationship of the processes)
- **UID:** User id (Determine the various process privileges.
- Scheduling information,
- Process owner
- Process priority
- environment variables for each process
- The parent process
- Signals that is sent to the process but not yet handled

There is a single process table for the entire system. This will be having information about all processes which are running in the system.

<u>User area:</u>

In addition to the text, data, and stack segments, the operating system also maintains for each process a region called the u area (user area).

The u area contains information specific to the process (e g., open files, current directory, signal actions, accounting information) and a system stack segment for process use. If the process makes a system ,the stack frame information for the system call is stored in the system stack segment.

Again, this information is kept by the operating system in an area that the process does not normally have access to. Thus, if this information is needed, the process must use special system calls to access it.

Like the process itself, the contents of the u area for the process are paged in and out by the operating system.

- A pointer to the process table slot.
- Parameters of the current system call, return values error codes.
- File descriptors for all open files.
- Current directory and current root.
- Process and file size limits and etc.

Types of Processes

There are generally 4 types of processes that are run on Linux environment.
Namely:

1. Interactive processes
2. Orphan processes
3. Zombie processes
4. Daemon processes

1. Interactive processes:
These are the processes that are invoked by a user and can interact with the user. In Linux environment, whatever the user executing programs or executing inbuilt commands will be called as interactive processes. Again these interactive processes can be classified into foreground and background processes.

The *foreground process* is the process that you are currently interacting with, and is using the terminal as its stdin (standard input) and stdout (standard output).

Ex: < Shell >$ ./a.out

A *background process* is not interacting with the user and can be in one of two states - paused or running. If you want to execute a process in background, add '&' symbol with that process name and execute.

Ex: < Shell >$ ./a.out&

2. Orphan Process:
An Orphan Process is nearly the same thing which we see in real world. Orphan means someone whose parents are dead. The same way this is a process, whose parents are dead, that means parents are either terminated, killed or exited but the child process is still alive.

In Linux/Unix like operating systems, as soon as parents of any process are dead, re-parenting occurs, automatically. Re-parenting means processes, whose parents are dead, means Orphaned processes, and are immediately adopted by special process "init". Thing to notice here is that even after re-parenting, the process still remains Orphan as the parent which created the process is dead.

Reasons for Orphan Process: The following are some reasons for Orphan Process:

- A process can be orphaned either intentionally or unintentionally. Sometime a parent process exits/terminates or crashes leaving the child process still running, and then they become orphans.

- Also, a process can be intentionally orphaned just to keep it running. For example when you need to run a job in the background which don't need any manual intervention and going to take long time, then you detach it from user session and leave it there.
- At the same time, when a client connects to a remote server and initiated a process, and due to some reason the client crashes unexpectedly, the process on the server becomes Orphan.

3. Zombie Process:
A Zombie Process is nearly the same thing which we see in lot of horror movies. Like the dead people which don't have any life force left, Zombie processes are the dead processes sitting in the process table and doing nothing.

Zombie process or *defunct process* is a process that have completed the execution, have released all the resources (CPU, memory) but still had an entry in the process table.

Reasons for Zombie Process: Most of the time, the reason for existence of Zombie process is bad coding. Normally, when a child (sub process) finishes its task and exits, then its parent is suppose to use the "wait" system call and get the status of the process.

So, until the parent process doesn't check for the child's exit status, the process is a Zombie process, but it usually is very small duration. But if due to any reason (bad programming or a bug), the parent process didn't check the status of the child and don't call "wait", the child process stays in the state of Zombie waiting for parent to check it's status.

4. Daemon Process:
A Daemon process is a process which is not associated with any terminal and hence is supposed to run in background. Since, a daemon process involves background processing, it is recommended it should not include any user interaction. Therefore, it should be clear about its continuous processing not caring to wait for any input or to display any output.

Traditionally daemon names end with the letter d: for example, syslogd is the daemon that implements the system logging facility and sshd is a daemon that services incoming SSH connections.

A daemon is usually created by a process forking a child process and then immediately exiting, thus causing init to adopt the child process.

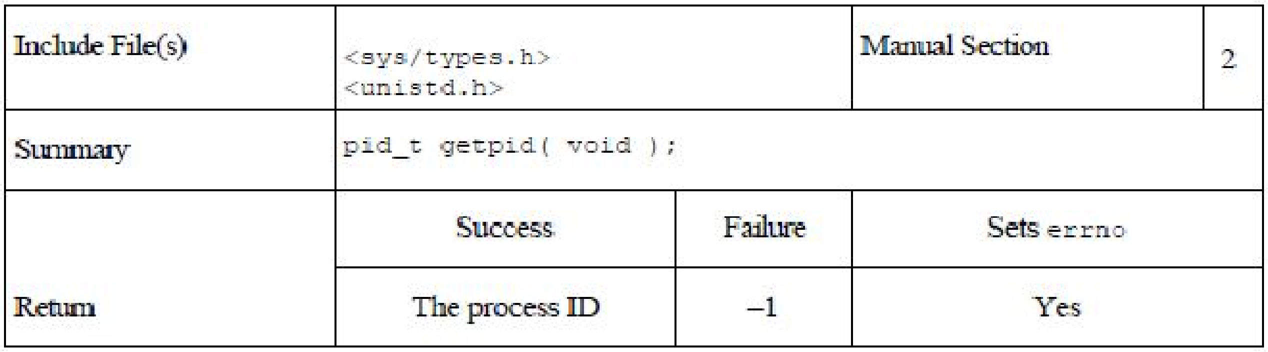
System Calls related to Process

As you have seen that, In linux there are several system calls provided to access the kernek resources. Now let us discus the system calls related to Process.

1. `getpid(), getppid(), getuid();`

- `getpid()` system call is used to get the calling process id. The prototype is as follows.

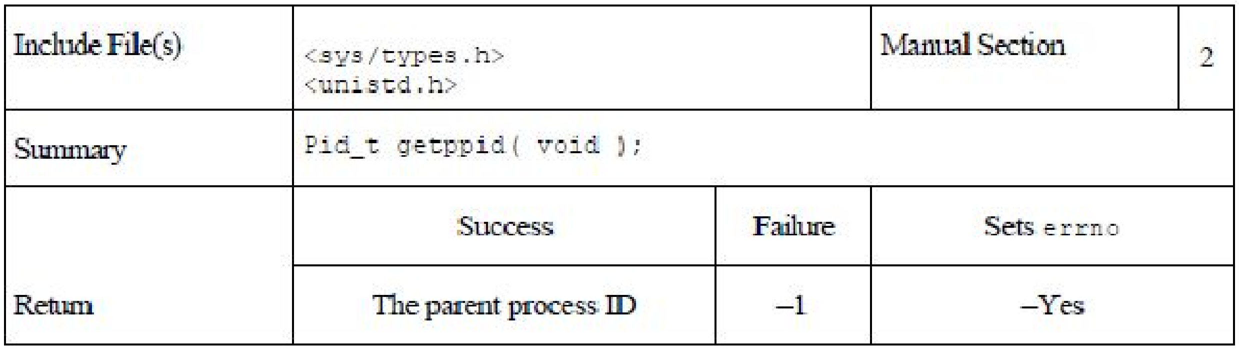
Summary of `getpid()` system call



Include File(s)	`<sys/types.h>` `<unistd.h>`		Manual Section	2
Summary	`pid_t getpid( void );`			
Return	Success	Failure	Sets `errno`	
	The process ID	–1	Yes	

- `getppid()` system call is used to get the parent process id of a calling process. The prototype is given below.

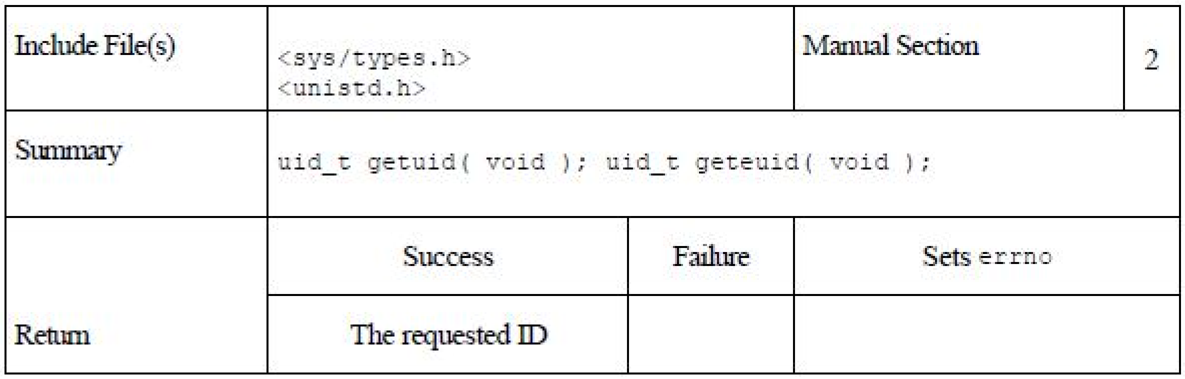
Summary of `getppid()` system call



Include File(s)	`<sys/types.h>` `<unistd.h>`		Manual Section	2
Summary	`Pid_t getppid( void );`			
Return	Success	Failure	Sets `errno`	
	The parent process ID	–1	–Yes	

- `getuid()` system call is used to get the real user id of a calling process. The prototype is given below:

Summary of `getuid()` system call

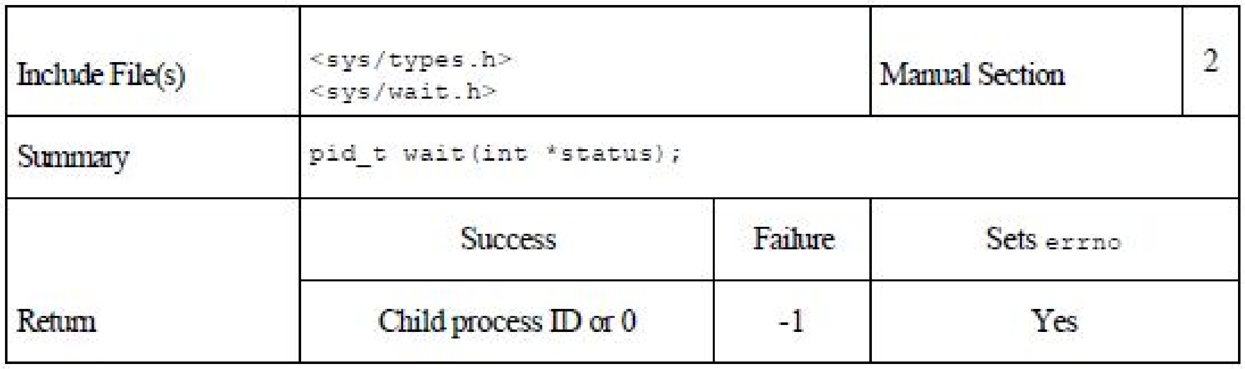


Include File(s)	`<sys/types.h>` `<unistd.h>`	Manual Section	2
Summary	`uid_t getuid( void ); uid_t geteuid( void );`		
	Success	Failure	Sets `errno`
Return	The requested ID		

Ex: /****** WAP to print a process pid, ppid and uid ***********/

```
int main()
{
     printf(“process id:……………. %d\n”, getpid() );
     printf(“parent process id :…… %d\n”, getppid() );
     printf(“process uid :………..… %d\n”, getuid() );
     return 0;
}
```

2. wait(); This system call suspends execution of the calling process untill one of its children terminates. The prototype is given below.



Include File(s)	`<sys/types.h>` `<sys/wait.h>`	Manual Section	2
Summary	`pid_t wait(int *status);`		
	Success	Failure	Sets `errno`
Return	Child process ID or 0	-1	Yes

Here:

Status: address to save the status of terminated child.

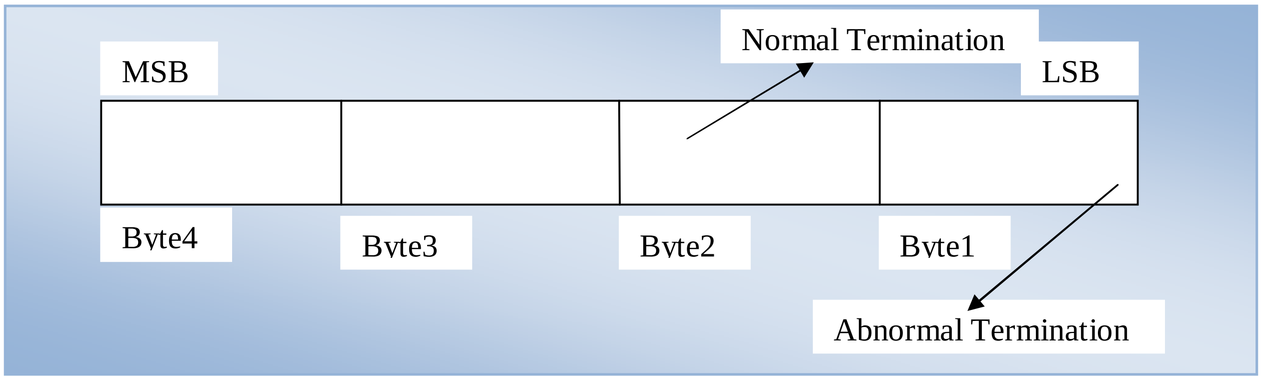
Ex: /****** WAP to show the working of wait() system call *********/

```
int main()
{
     int pid;
     if( fork() == 0)
     {
          printf("child pid: %d\n", getpid());
          exit(0);
     }
     else
     {
          pid = wait( (int *) 0);       //   waiting   to   child
exit.
          printf("%d is terminated\n", pid);      //       prints
terminated pid
     }
     return 0;
}
```

Note: Here fork() is a function to create a process. See next system calls.

Status of Child Process: Parent process can get the status of a child process by passing an address of integer to wait system call.

Wait calls fills an integer and in many cases the last two bytes have status information.



Ex: /*********** WAP to check the normal status of a child process ******/

```
int main()
{
     int integer;
     if(fork() == 0)
          exit(5);  // normal exit
     else  {
          wait(&integer);
          printf("normal  status  is:  %d\n",  (integer  >>  8)&0xff
);   // prints 5
     }
     return 0;
}
```

Explanation: in the above program the child is terminated normally. So the normal status will be in 2nd byte of integer. And we are printing 2nd byte. This will print normal status as 5.

Ex: /*********** WAP to check the abnormal status of a child process ******/
```
int main()
{
     int integer, *p=NULL;
     if(fork() == 0)
          printf("%d\n", *p); //   dereferencing   NULL   pointer
makes abnormal exit
     else  {
          wait(&integer);
          printf("normal  status  is:  %d\n",  (integer)&0xff  );
     // prints abnormal status
     }
     return 0;
}
```

3. Fork():

When a program calls fork, a duplicate process, called the *child process*, is created.The parent process continues executing the program from the point that fork was called.

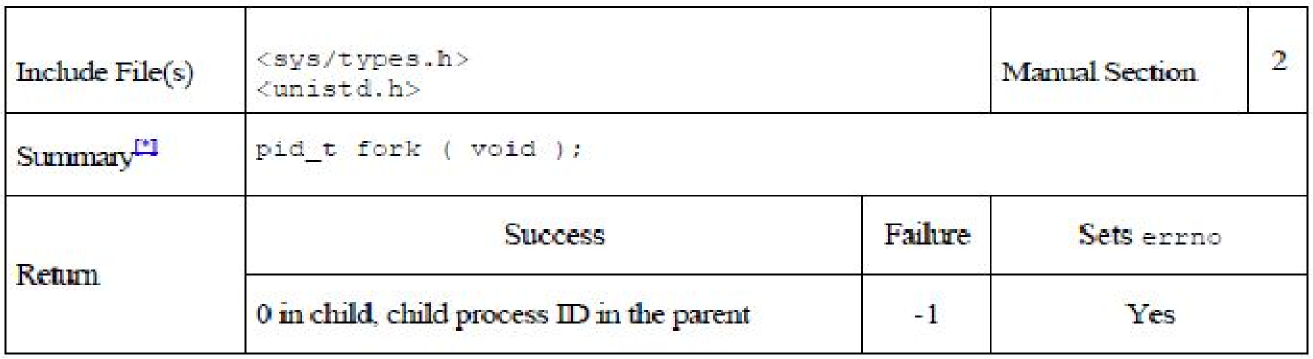
The child process, too, executes the same program from the same place.

First, the child process is a new process and therefore has a new process ID, distinct from its parent's process ID. One way for a program to distinguish whether it's in the parent process or the child process is to call `getpid`().

When any one of the process (parent or child) is trying to modify data segment, then separate copy will be given to both processes,

But the Text segment will be same for both processes,

Summary of `fork()` system call



Include File(s)	`<sys/types.h>` `<unistd.h>`		Manual Section	2
Summary[*]	`pid_t fork ( void );`			
Return	Success	Failure	Sets `errno`	
	0 in child, child process ID in the parent	-1	Yes	

Ex-1: /******* example for fork *****/

```
int main()
{
     fork();
     fork();
     printf(“this will print 4 times\n”);
}
```

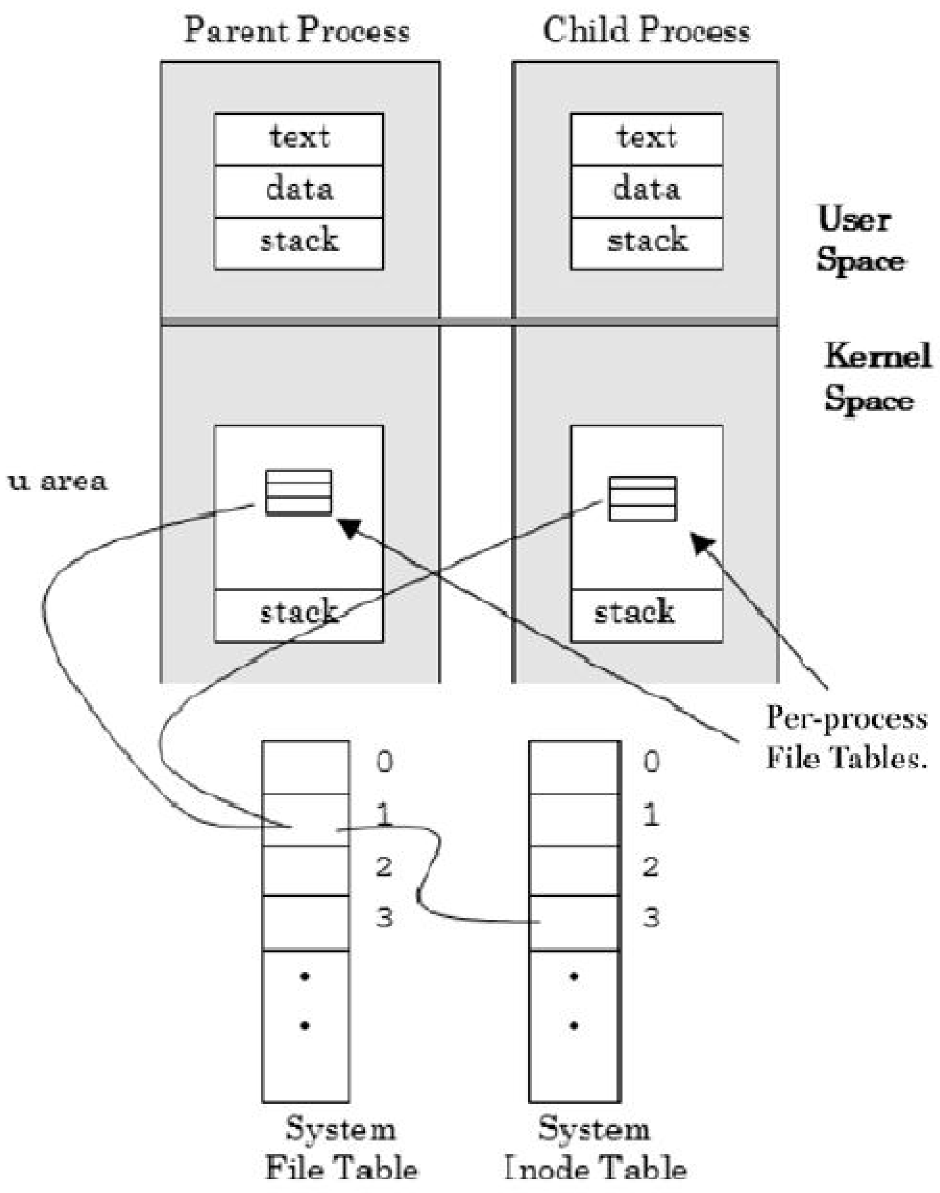
Ex-4: /********* WAP to create child process using fork call *********/

```
int main()
{
     int var;
     var = fork();
     if(var == 0) {
          printf(“this is child\n”);
          printf(“because its receiving zero\n”);
          printf(“%d\n”, var);
          printf(“%d\n”, getpid());
           printf(“%d\n”, getppid());
          printf(“end of child process\n”);
          exit(0);
     }
     else if(var > 0) {
          wait(0);
          printf(“this  is  parent,  coz  its  receiving  child
pid\n”);
          printf(“%d\n”, var);
          printf(“%d\n”, getpid());
          printf(“end of parent\n”);
          exit(1);
     }
     else {
          printf(“error in fork\n”);
          exit(2);
     }
     return 0;
}
```

Ex-3: /****** WAP to demonstrate how fork follows COW method *******/

```
int var = 10;                                    //  global variable
int main()
```

```
{
     int pid;
     pid = fork();
     if(pid == 0) {
          printf(“In child\n”);
          var = 20;
          printf(“var = %d\n”, var);         // var = 20
          exit(0);
     }
     else if(pid  > 0) {
          wait(0);                 //  parent  wait  till  child
terminates
          printf(“In parent\n”);
          printf(“var = %d\n”, var);         // var = 10
          exit(0);
     }
     else {
          printf(“error in fork\n”);
          exit(2);
     }
     return 0;
}
```



4. vfork();

- vfork() is also a function is used to create a child process,
- vfork() will not follow COW method,
- Parent and child processes will execute in same Text, Stack, and Data segments.
- Data will be shared by both parent and child processes

Ex: /************ WAP to demonstrate the working of vfork() system call ********/

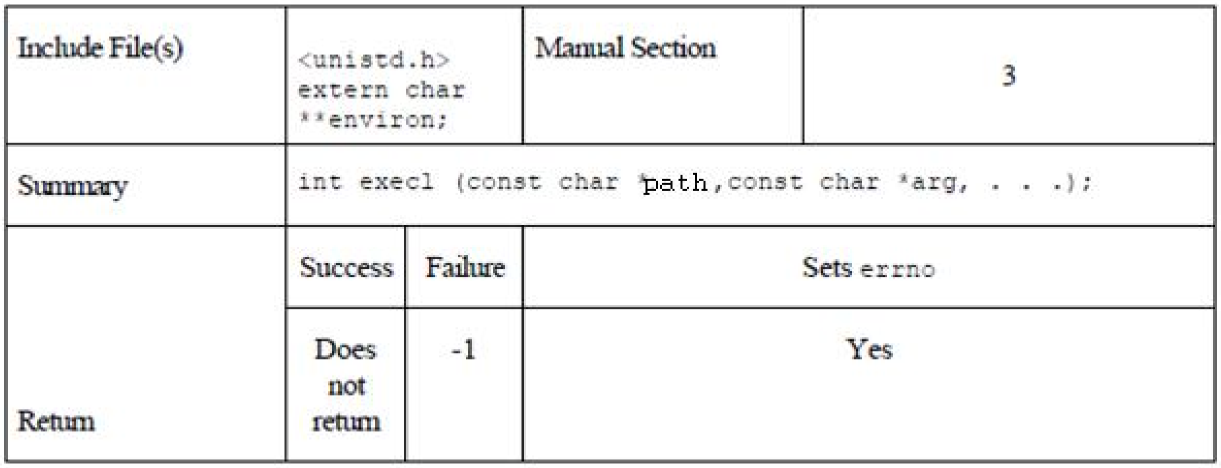
```
int var = 10;                                 //  global variable
int main()
{
     int pid;
     pid = vfork();
     if(pid == 0) {
          printf(“In child\n”);
          var = 20;
          printf(“var = %d\n”, var);         // var = 20
          exit(0);
     }
     else if(pid  > 0) {
          wait(0);                  //  parent  wait  till  child
terminates
          printf(“In parent\n”);
          printf(“var = %d\n”, var);         // var = 20
          exit(0);
     }
     else {
          printf(“error in fork\n”);
          exit(2);
     }
     return 0;
}
```

Exec System Calls

The exec family of functions will initiate a program from within a program. Or exec family is used to execute the executable file on the top of Calling Process.

<u>execl() and execlp():</u>

Summary of `execl()` system calls



Include File(s)	`<unistd.h>` `extern char **environ;`	Manual Section	3
Summary	`int execl (const char *path,const char *arg, . . .);`		
Return	Success	Failure	Sets `errno`
	Does not return	-1	Yes

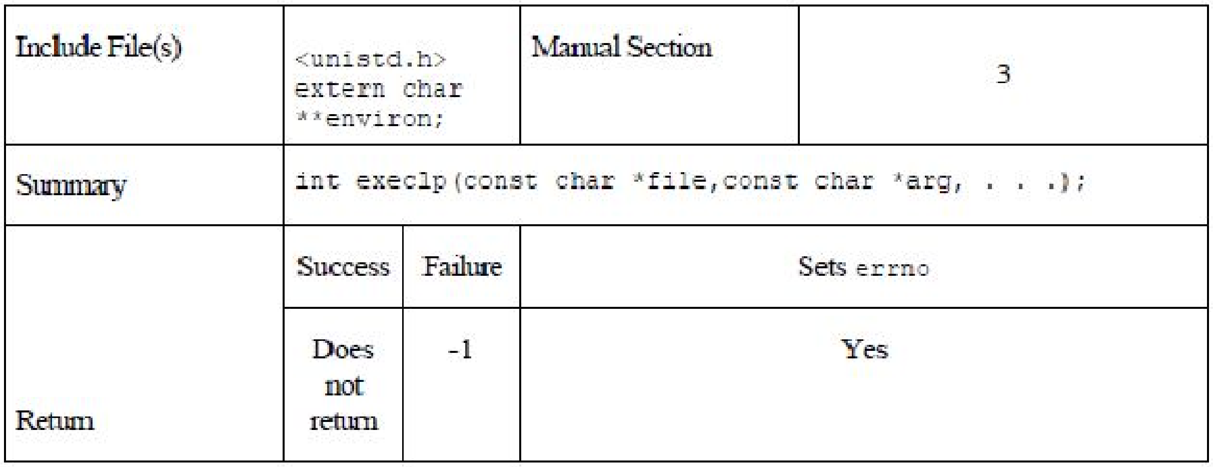
- The function call "execl()" initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. /bin/ls) and arguments are passed to the function. Note that 1st argument is the command/file name to execute.

Ex: /********** WAP to execute ls command using execl call ***********/

```
int main()
{
  printf(“before\n”);
  execl(“/bin/ls”, “ls”, “-1”, NULL);
  printf(“this statement never prints this statement\n”);
  return 0;
}
```

- The function call execlp() does the same operation as execl(). Here no need to give full path.
  The prototype is given below:

Summary of `execlp()` system calls



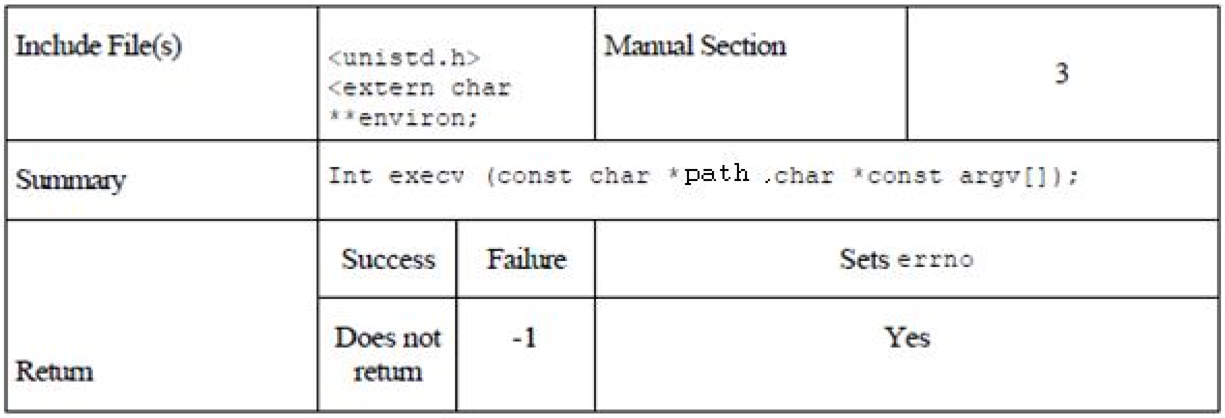
Include File(s)	`<unistd.h>` `extern char **environ;`		Manual Section	3
Summary	`int execlp(const char *file,const char *arg, . . .);`			
Return	Success	Failure	Sets `errno`	
	Does not return	-1	Yes	

Ex: /********** WAP to execute ls command using execlp call ***********/

```
int main()
{
  printf("before\n");
  execlp("ls", "ls", "-1", NULL);
  printf("this statement never prints this statement\n");
  return 0;
}
```

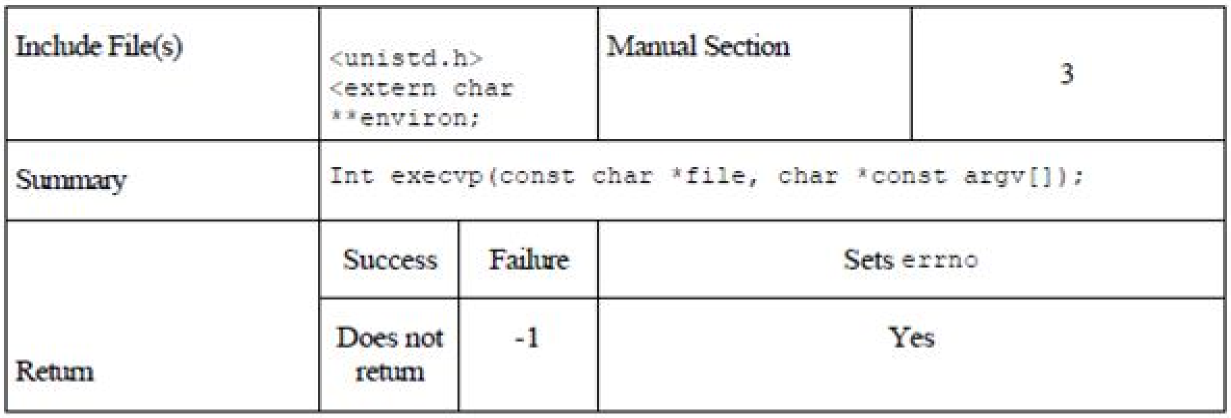
execv() and execvp():

Summary of `execv()` System call summary



Include File(s)	`<unistd.h>` `<extern char **environ;`		Manual Section	3
Summary	`Int execv (const char *path ,char *const argv[]);`			
Return	Success	Failure	Sets `errno`	
	Does not return	-1	Yes	

Summary of execvp() System call summary



Include File(s)	`<unistd.h>` `<extern char **environ;`	Manual Section	3
Summary	`Int execvp(const char *file, char *const argv[]);`		
Return	Success	Failure	Sets errno
	Does not return	-1	Yes

- This is the same as **execl**() except that the arguments are passed as null terminated array of pointers to char. The first element "argv[0]" is the command name.

path:	command name in full path,	Ex:/bin/ls
args:	address of arguments	

Ex: /********** WAP to execute ls command using execv call ***********/
```
int main()
{
   char *p[3] = { “ls”, “-1”, NULL};
   printf(“before\n”);
   execv(“/bin/ls”, p);
   printf(“this statement never prints this statement\n”);
   return 0;
}
```

Ex: /********** WAP to execute ls command using execvp call ***********/
```
int main()
{
   char *p[3] = { “ls”, “-1”, NULL};
   printf(“before\n”);
   execvp(“ls”, p);
   printf(“this statement never prints this statement\n”);
   return 0;
}
```

system():

- As exec functions, this system call is also used to replace the executable code in calling process. But this will not ends the calling process. After completion of executing command, it resumes the calling process.
  The prototype is given below:

  `int system(const char *command);`

  On failure returns -1, and on success returns integer value which is returned by command.

  This system call executes a command specified in command by calling shell with /bin/sh –c command and returns after the command has been completed. During execution of the command SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored.

  Ex: /********* WAP to execute ls command using system *************/

```
int main()
{
  printf("\n before system\n");
  system("ls  -a");
  printf("\n This statement will execute\n");
  return 0;
}
```

Killing a process

Eventually all things must come to an end. Now that we have generated processes, we should take a closer look at how to end a process. Under its own power (assuming the process does not receive a terminating signal and the system has not crashed) a process normally terminates in one of three ways.

In order of preference, these are

1. It issues (at any point in its code) a call to either exit or _exit.
2. It issues a return in the function main.
3. It falls off the end of the function main ending implicitly.

The kill command will send the specified signal to the indicated PID.

When terminating a process, the system performs a number of housekeeping operations:

- All open file descriptors are closed.

- The parent of the process is notified (via a SIGCHLD signal) that the process is terminating.
- Status information is returned to the parent process (if it is waiting for it). If the parent process is not waiting, the system stores the status information until a wait by the parent process is
- affected.
- All child processes of the terminating process have their parent process ID (PPID) set to 1—they are inherited by init.
- If the process was a group leader, process group members will be sent SIGHUP/ SIGCONT
- signals.
- Shared memory segments and semaphore references are readjusted.
- If the process was running accounting, the accounting record is written out to the accounting file.

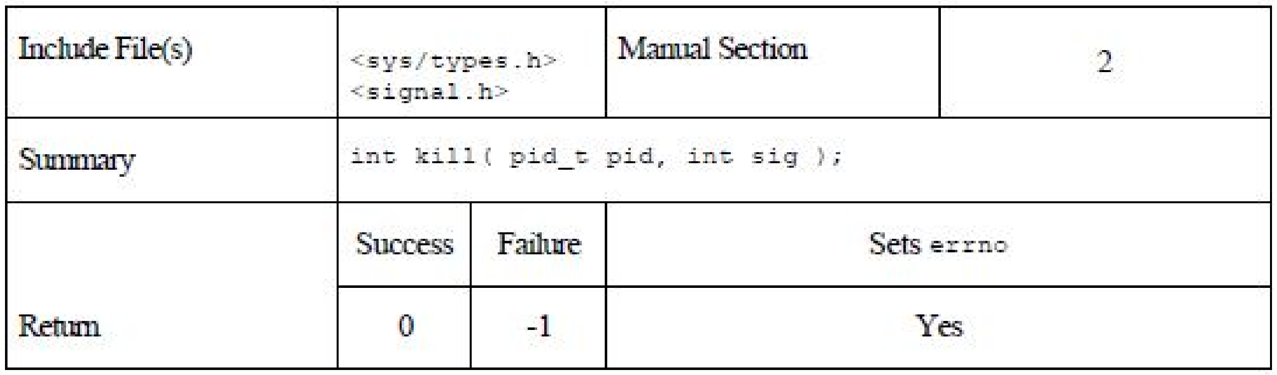
- To list all running processes in terminal, we have to use ps command. This will display, all processes, with their ids.

```
<Shell>$ ps
```

- **kill command:**

```
<Shell    >$  kill  [signal name]  process_id
```

Summary of `kill()` system call



Include File(s)	`<sys/types.h>` `<signal.h>`	Manual Section	2
Summary	`int kill( pid_t pid, int sig );`		
Return	Success	Failure	Sets `errno`
	0	-1	Yes

Here the command kill sends the specified signal to the specified process. If no signal is specified, the SIGTERM signal is sent. The TERM signal will kill the process and can be ignored by the process. If you are sending SIGKILL signal, the process can't be ignored and it will be killed.

Ex: /**** kill a process with SIGTERM signal **********/

```
<Shell>$  kill process_id
```

Ex: /**** kill a process with SIGKILL signal **********/

```
<Shell>$  kill –SIGKILL process_id
```

- **`pkill` Command:** this command will kill the specified process name.
  ```
  <Shell>$ pkill process_name
  ```

- **`killall` command:** This command is used to kill the multiple processes having same name.
  ```
  <Shell>$ killall  sample
  ```

  The above command will kill all sample processes, which running in terminal.

Key learning:

1. What is a process?
2. What are the components of a UNIX process context?
3. What are the kernel data structures maintained for process management?
4. What are the different states of a process?
5. Explain briefly the two modes of execution of a unix process?
6. When is context switch possible in unix OS?
7. Why is kernel mode of execution used by UNIX processes?
8. What are various IDs associated with a process?
9. What will be the output of the following program code?
   ```
   main()
   {
      fork();
      fork();
      fork();
      printf("Hello World!");

   }
   ```
10. Enlist the attributes which child process doesn’t inherit from its parent.
11. List the system calls used for process management.
12. Differentiate between fork() and vfork() system call?
13. Predict the output of following code.
    ```
    main()
    {
       execlp( “ls”, “ls”, (char *) 0);
       fork();
       execlp( “ls”, “ls”, (char *) 0);

    }
    ```
14. How can a process be killed?