Chapter-5 Shell and Process

TOPICS: Process:

Basic idea about UNIX process, Display process attributes (ps), Display System processes, Process creation cycle,

Shell creation steps (init -> getty -> login -> shell), Process state, Zombie state,

Background jobs (& operator, nohup command), Reduce priority (nice), Using signals to kill process, Sending job to background (bg) and foreground (fg), Listing jobs (jobs), Suspend job, Kill a job, Execute at specified time (at and batch)

UNIX PROCESS:

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside :main memory –

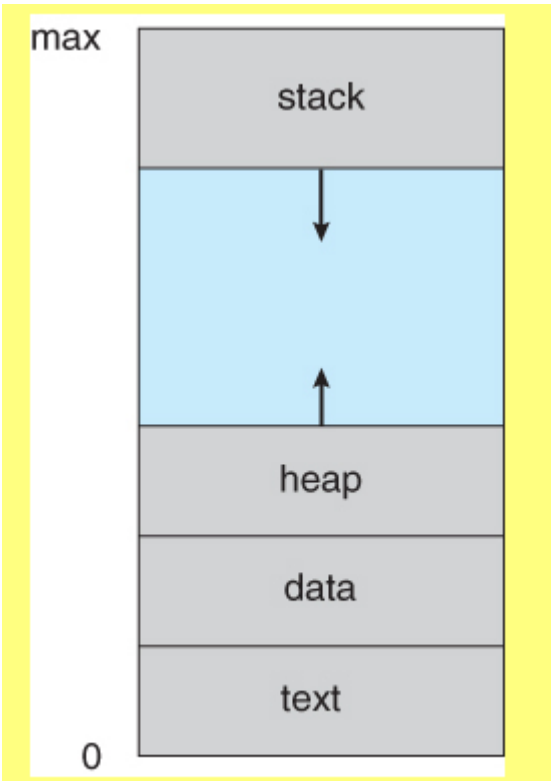


Figure 3.1 - A process in memory

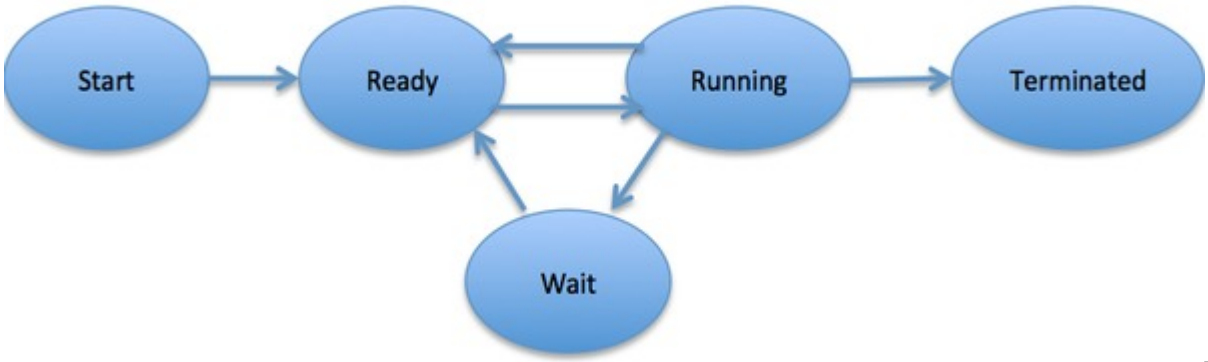
S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope.
2	**Heap** This is dynamically allocated memory to a process during its run time. heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc. o Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers. The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
4	**Data** The data contains the global and static variables., allocated and initialized prior to executing main.

Process Life Cycle/process state:

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

S.N.	State & Description
1	**New/Start** This is the initial state when a process is first started/created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.



-

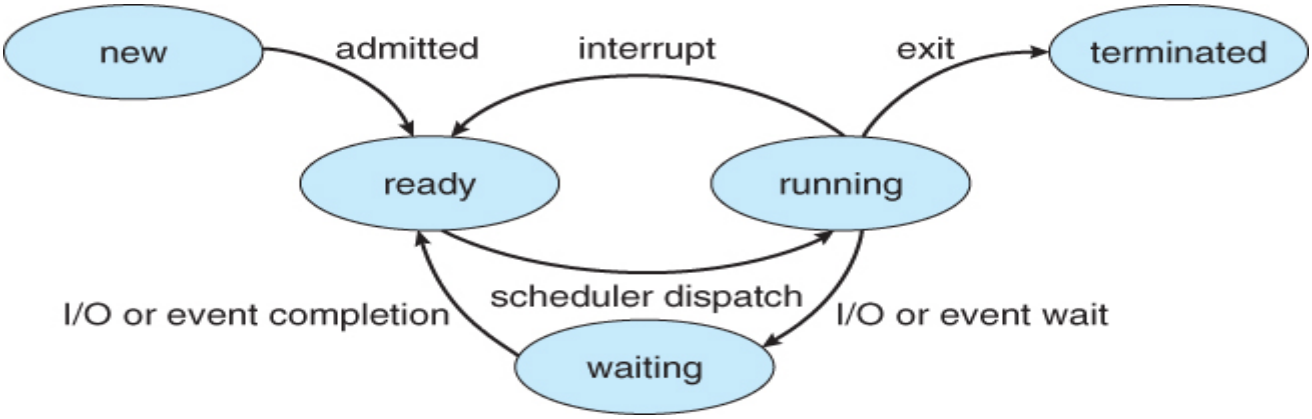


Figure - Diagram of process state

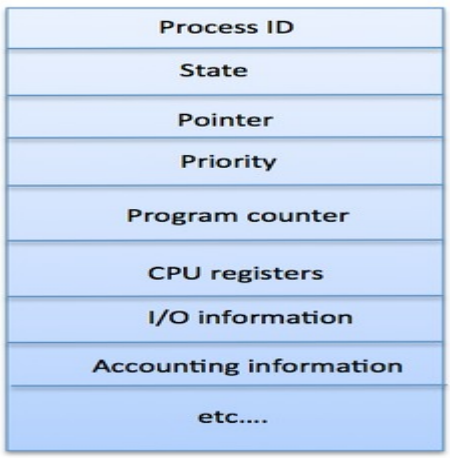
Process Control Block (PCB):

A **Process Control Block** is a data structure maintained by the Operating System for every process. The PCB is identified by **an integer process ID (PID)**. A PCB keeps all the information needed to keep track of a process as listed below in the **table −**

S.N.	Information & Description
1	**Process State** Running, waiting, etc., whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID(PID)** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.
5	**Program Counter(PC)** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for

	running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information(MMU)** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process, open file tables, etc.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



The **PCB is maintained for a process throughout its lifetim**e, and is **deleted once the process terminates.**

Init process/scripts:

Init (short for **initialization**) is the program on Unix and Unix-like systems that spawns all other processes. It runs as a daemon and typically has **PID 1 and PPID as 0.**

The **/etc/inittab** file is used to set the default run level for the system. This is the runlevel that a system will start up on upon reboot. The applications that are started by init are located in the **/etc/rc.d** folder. Within this directory there is a separate folder for each run level, e.g. *rc0.d*, *rc1.d*, and so on.

After the Linux kernel has booted, the init program reads the **/etc/inittab** file to determine the behaviour for each run level. Unless the user specifies another value as a kernel boot parameter, the system will attempt to enter (start) the default runlevel.

Standard run levels for RedHat based distributions		
Run Level ID	**Mode**	**Action**
0	**Halt**	**Shuts down system**
1 or S	**Single-User Mode**	**Does not configure network interfaces, start daemons, or allow non-root logins Only root user login.**
2	**Multi-User Mode**	**Does not configure network interfaces or start daemons.**
3	**Multi-User Mode with Networking**	**Starts the system normally.**
4	**Undefined**	**Not used/User-definable**
5	**Start the system normally with appropriate display manager (with GUI)**	**As runlevel 3 + X display manager**
6	**Reboot**	**Reboots/Restart the system**

Shell creation steps (init -> getty -> login -> shell):

Login Process in Linux:

When Linux booting process is completed it display the login prompt. There are different process running in Linux when any user login to their shell. these processes are as follows.

Login process steps:

(i)Init starts getty process

(ii)getty process initiates login prompt on terminal

(iii)login command check user credentials from /etc/passwd

(iv)getty starts user shell process

(v)shell reads the system wide files /etc/profile, /etc/bashrc

(vi)Shell reads user specific files .profile, .login

Now it reads shell specific configuration file .bashrc

(vii)Shell displays the default prompt.

(i)init starts getty:

When Linux system boots it goes through various booting stages. In last stage it starts **init**, which reads a file called **inittab** which is located in **/etc**, where it find out in which run level it has to execute. Once init process completes run-level execution and executing commands in **/etc/rc.local**, it will start a process called **getty**. getty is the process which will take care of complete login process.

(ii)Getty shows login prompt:

getty is short for "**get terminal**". A getty program initiates login command, it opens the terminal device, initializes it, prints login: and waits for a user name to be entered.

(iii)getty starts /etc/login:

Once user enters his login name getty starts **/ets/login**, this in-turn will prompt for user password. The password what user typed will be hidden and will not be shown on screen.

(iv)getty verifies the credential and starts users shell:

In next stage **getty** checks the user credentials by verifying it with **/etc/passwd** and **/etc/shadow** file, if password matches then it initiates user properties gathering and starts users shell.

If password doesn't match then getty terminates login process and re-initiates again with new login: prompt.

In next stage the getty process reads the user properties (username, UID, GID, home directory, user shell etc.) from **/etc/passwd** file. After that it reads **/etc/motd** file for displaying content banner message.

(v)Shell reads system wide default files and specific default files:

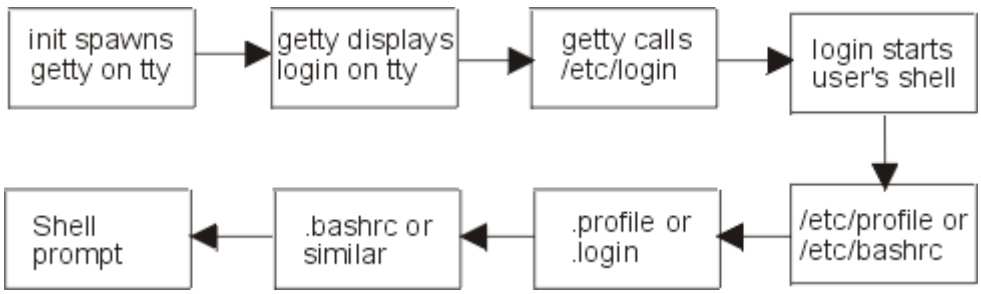
For getting shell related properties, user aliases and variables getty process reads appropriate system wide default files **/etc/profile** or **/etc/bashrc** . After the system wide default files are read the shell reads **user specific login files .profile** or **.login .**

Shell specific file read:

At last stage it reads shell specific configuration files **(.bashrc, .bash_profile** etc. for **bash shell**) of that user which it gets on the users home directory.

Shell Prompt:

When all startup files are read the shell displays the default prompt, normally PS1 prompt for user to execute their commands, here user can type any command to execute.



What is a process in UNIX / Linux?

A **process** is a program in execution in memory or in other words, an instance of a program in memory. Any program executed creates a process. A program can be a command, a shell script, or any binary executable or any application. However, not all commands end up in creating process, there are some exceptions. Similar to how a file created has properties associated with it, a process also has lots of properties associated to it.

Process attributes:

A process has some properties associated to it:

PID : Process-Id. Every process created in Unix/Linux has an identification number associated to it which is called the process-id. This process id is used by the kernel to identify the process similar to how the inode number is used for file identification. The PID is unique for a process at any given point of time. However, it gets recycled.

PPID : Parent Process Id: Every process has to be created by some other process. The process which creates a process is the parent process, and the process being created is the child process. The PID of the parent process is called the parent process id(PPID).

TTY: Terminal to which the process is associated to. Every command is run from a terminal which is associated to the process. However, not all processes are associated to a terminal. There are some processes which do not belong to any terminal. These are called daemons.

UID: User Id- The user to whom the process belongs to. And the user who is the owner of the process can only kill the process(Of course, root user can kill any process). When a process tries

to access files, the accessibility depends on the permissions the process owner has on those files.

File Descriptors: File descriptors related to the process: input, output and error file descriptors.

List the processes:

```
debasis@LAPTOP-H3N6JCNE:~$ ps
  PID TTY          TIME CMD
   10 pts/0    00:00:00 bash
   24 pts/0    00:00:00 ps
```

ps is the Unix / Linux command which lists the active processes and its status. By default, it lists the processes belonging to the current user being run from the current terminal.

The ps command output shows 4 things:
PID : The unique id of the process
TTY: The terminal from which the process or command is executed.
TIME: The amount of CPU time the process has taken
CMD: The command which is executed.

Two processes are listed in the above case:
1. **bash** : The login shell, which we are working on, is also a process which is currently running.
2. ps : The ps command which we executed to get the list also creates a process. And hence, by default, there will be at least 2 processes when executing the ps command.

Parent & Child Process:

Every process in Unix has to be created by some other process.
Hence, the ps command is also created by some other process. The 'ps' command is being run from the login shell, bash.
The bash shell is a process running in the memory right from the moment the user logged in. So, for all the commands trigerred from the login shell, the login shell will be the parent process and the process created for the command executed will be the child process.
In the same lines, the 'bash' is the parent process for the child process 'ps'.

The below command shows the process list along with the PPID.

The PID of the bash is same as the PPID of the ps command which means the bash process is the parent of the ps command. The '-o' option of the ps command allows the user to specify only the fields which he needs to display.

#Show full information about processes

```
debasis@LAPTOP-H3N6JCNE:~$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
debasis       10     9  0 15:05 pts/0    00:00:00 -bash
debasis       29    10  0 15:11 pts/0    00:00:00 ps -f
```

#Display extended system process

```
debasis@LAPTOP-H3N6JCNE:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
    8 ?        00:00:00 init
    9 ?        00:00:00 init
   10 pts/0    00:00:00 bash
   30 pts/0    00:00:00 ps
```

debasis@LAPTOP-H3N6JCNE:~/process$ **ps -e**

```
 PID  TTY          TIME CMD
   1    ?        00:00:00 init    [ ? specify as background or daemon process ]
   8  tty1     00:00:00 init
   9  tty1     00:00:01 bash
 552  tty1     00:00:00 ps
```

Init Process:
If all processes of the user are created by the login shell, who created the process for the login shell?
In other words, which is the parent process of the login shell?
When the Unix system boots, the first process to be created is the ***init* process**. This init process will have the **PID as 1** and **PPID as 0.**
All the other processes are created by the init process and gets branched from there on. Note in the below, command, the process of the login shell has the **PPID 1** which is the PID of the ***init*** process.

#Show all processes on the system:

```
debasis@LAPTOP-H3N6JCNE:~$ ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 15:05 ?        00:00:00 /init
root           8     1  0 15:05 ?        00:00:00 /init
root           9     8  0 15:05 ?        00:00:00 /init
debasis       10     9  0 15:05 pts/0    00:00:00 -bash
debasis       32    10  0 15:14 pts/0    00:00:00 ps -ef
```

The `-e` option instructs `ps` to display all processes.

The `-f` **stands full-format listing**, which provides detailed information about the proces

Q: why two /init in above output:

```
debasis@LAPTOP-H3N6JCNE:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init    [ init ID 1 and its work as background/daemon process ]
    8 tty1     00:00:00 init    [ user PPID 8 ]
    9 tty1     00:00:00 bash
   69 tty1     00:00:00 ps
debasis@LAPTOP-H3N6JCNE:~$ echo $$
9
debasis@LAPTOP-H3N6JCNE:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
debasis     9     8    0   09:45 tty1     00:00:00 -bash  [ user PID 9 and PPID 8 ]
debasis    70     9    0   09:48 tty1     00:00:00 ps -f
```

NOTES:
In the olden days before Upstart, there was just one **init** process (where PID = 1). ... But, when a user logs created for that user session. Thus, that is why you see the second **init** process (where PID != 1).

```
debasis@LAPTOP-H3N6JCNE:~$ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   8936   316 ?        Ssl  09:54   0:00 /init
root         8  0.0  0.0   8936   224 tty1     Ss   09:54   0:00 /init
debasis      9  0.0  0.0  18212  3680 tty1     S    09:54   0:00 -bash
debasis     74  0.0  0.0  17136  2556 tty1     T    09:57   0:00 man ps aux
debasis     86  0.0  0.0  15636  1324 tty1     T    09:57   0:00 pager
debasis    101  0.0  0.0  18880  2036 tty1     R    10:15   0:00 ps -aux
```

- **a** option tells `ps` to display the processes of all users.
- **x** option instructs `ps` to list the processes **without a controlling terminal.**

COMMAND NAME: top

top - display Linux processes

DESCRIPTION:

The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of processes or threads currently being managed by the Linux kernel.

```
debasis@LAPTOP-H3N6JCNE:~$ top
top - 10:43:00 up 48 min,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:   6 total,   1 running,   3 sleeping,   2 stopped,   0 zombie
%Cpu(s):  3.1 us,  3.9 sy,  0.0 ni, 93.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   4005.2 total,    371.0 free,   3410.2 used,    224.0 buff/cache
MiB Swap:  12288.0 total,  11828.2 free,    459.8 used.    464.4 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
102	debasis	20	0	18900	2132	1524	R	0.7	0.1	0:00.07	top
1	root	20	0	8936	316	272	S	0.0	0.0	0:00.20	init
8	root	20	0	8936	224	180	S	0.0	0.0	0:00.00	init
9	debasis	20	0	18212	3680	3576	S	0.0	0.1	0:00.51	bash
74	debasis	20	0	17136	2556	2476	T	0.0	0.1	0:00.29	man
86	debasis	20	0	15636	1260	968	T	0.0	0.0	0:00.	01 pager

Here,

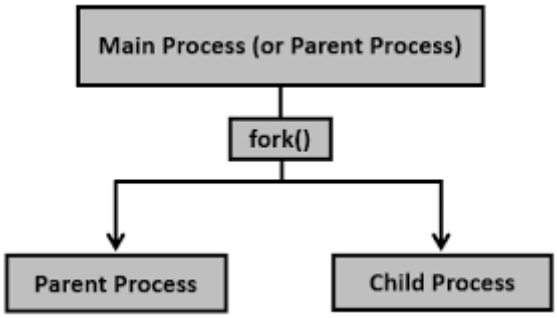
- **PID:** Shows task's unique process id.
- **PR:** Stands for priority of the task.
- **SHR:** Represents the amount of shared memory used by a task.
- **VIRT:** Total virtual memory used by the task.
- **USER:** User name of owner of task.
- **%CPU:** Represents the CPU usage.
- **TIME+:** CPU Time, the same as 'TIME', but reflecting more granularity through hundredths of a second.
- **SHR:** Represents the Shared Memory size (kb) used by a task.
- **NI:** Represents a Nice Value of task. A Negative nice value implies higher priority, and positive Nice value means lower priority.
- **%MEM:** Shows the Memory usage of task.

- `USER` - The user who runs the process.
- `%CPU` - The cpu utilization of the process.
- `%MEM` - The percentage of the process's resident set size to the physical memory on the machine.
- `VSZ` - Virtual memory size of the process in KiB.
- `RSS` - The size of the physical memory that the process is using.
- `STAT` - The the process state code, such as `Z` (zombie), `S` (sleeping), and `R` (running).
- `START` - The time when the command started

Exceptions to creating process:
Not all commands end up creating a process.

There are some exceptions.
i) Internal commands does not create a process since they are shell built-in.
ii) Any file if sourced does not create a process since it has to be run within the shel

Process creation is achieved through the fork() system call. The newly created **process** is called the child **process** and the **process** that initiated it (or the **process** when execution is started) is called the parent **process**. After the fork() system call, now we have two processes - parent and child processes



Question: Process in Unix? Mechanism of Process creation in Unix? Process states and transition with diagram?
Answer:
A process is simply a running program. A process is said to be born when the program starts execution and remains alive till the process is active. After complete the execution the process is said to be died.

A process in a Unix system is created by fork system call. Every process except process 0 is created. The process that invokes the fork system call is parent process and the newly created process is the child process.

Every process has one parent process but a parent can have many child process. The kernel identifies it process by its process identification number (PID). Process 0 is a special process created "by hand" when the system boots.

There are 3 distinct phase in mechanism of process creation and uses 3 system calls:
(i) fork(),(ii) exec() and (iii)wait().

fork(): Creates a child process. A new process is created because an existing process creates an exact copy of itself. This child process

has the same environment as its parent but only the PID is different. This procedure is known as forking.

exec(): After forking the process, the address space of the child process is overwritten by the new process data. This is done through exec call to the system.
exec() family of functions replaces the current process image with a new process image.
No new process is created over here. The PID & PPID remains unchanged.

wait(): The parent then executes wait system call to wait for the child process to complete its execution.

Process Creation:

```
#Create a process

#include<stdio.h>

#include<unistd.h>

int main()

{

	fork();

	printf("Hello world.\n");

	return(0);

}

Output:

debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out

Hello world.

Hello world.
```

```
#Create a process

#include<stdio.h>

#include<unistd.h>
```

```
int main()
{
    int id=fork();
    printf("Hello world from ID: %d\n",id);
    return(0);
}
```

Output:

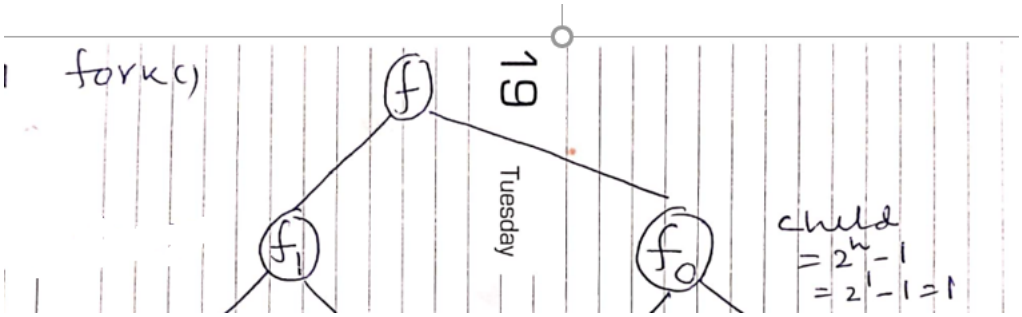
```
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
Hello world from ID: 105
Hello world from ID: 0
```

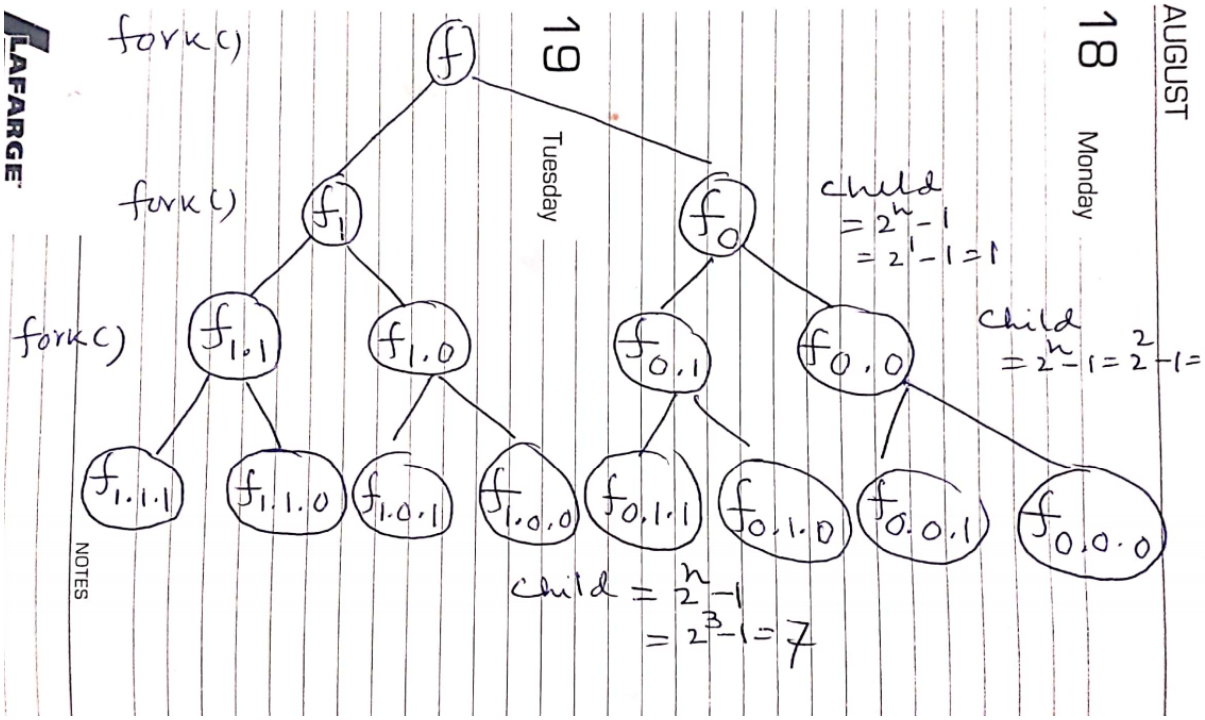
```c
//fork() system call
#include<stdio.h>
#include<unistd.h>
int main()
{
        pid_t id;

        id=fork();
        if (id < 0 )
                printf("\n fork creation failure....");
        if(id==0)
        {
                printf("\n %d CHILD Process...",id);
        }
        else if ( id > 0 )
        {
                printf("\n PARENT Process %d", id);
        }
        printf("\n %d Job Over...\n",id);
return 0;
}
```

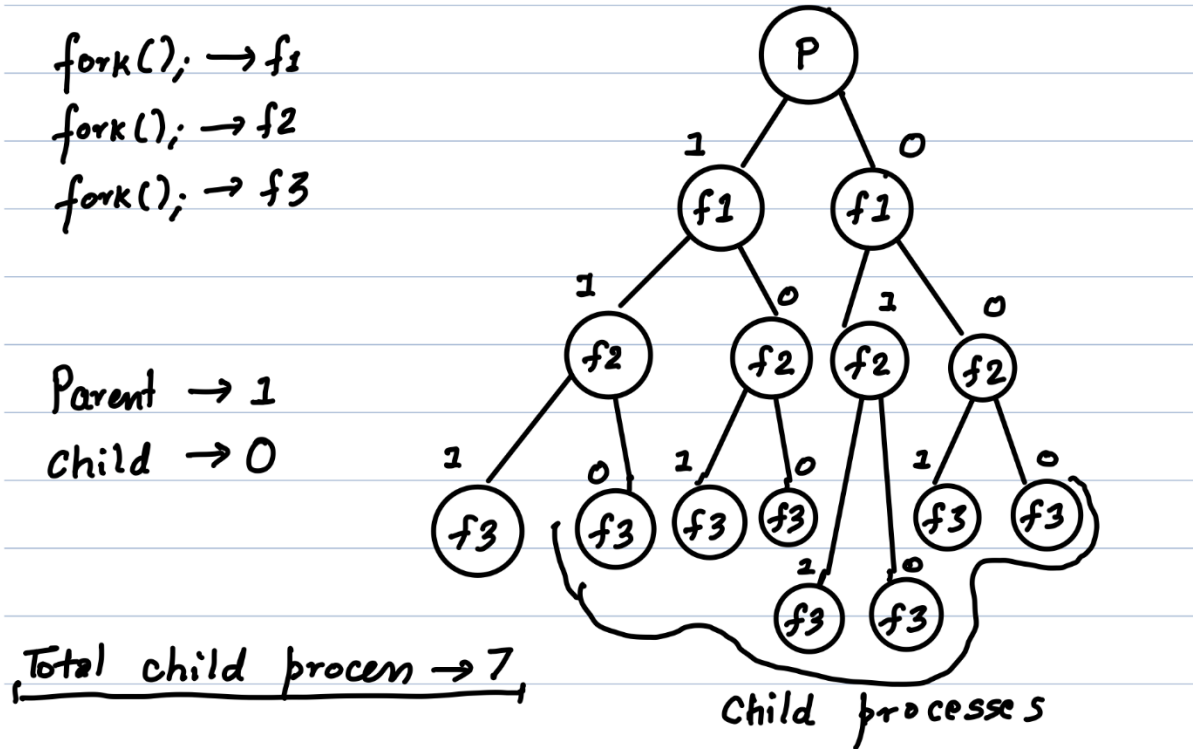
Output:

```
debasis@LAPTOP-H3N6JCNE:~$ gcc -o objfile ff.c
debasis@LAPTOP-H3N6JCNE:~$ ./objfile

 PARENT Process 61
```

```
#Create a process twice
#include<stdio.h>
#include<unistd.h>
int main()
{
	int id1=fork();
	int id2=fork();
	printf("Hello world from ID: %d \t %d\n",id1,id2);
	return(0);
}
Output:
debasis@LAPTOP-H3N6JCNE:~/process$ cc p1.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
Hello world from ID: 117	118
Hello world from ID: 117	0
Hello world from ID: 0	119
Hello world from ID: 0	0
```


fork()
f
f_1
f_0
19
Tuesday
child
$= 2^n - 1$
$= 2^1 - 1 = 1$


LAFARGE
fork()
fork()
fork()
f
f_1
f_0
$f_{1.1}$
$f_{1.0}$
$f_{0.1}$
$f_{0.0}$
$f_{1.1.1}$
$f_{1.1.0}$
$f_{1.0.1}$
$f_{1.0.0}$
$f_{0.1.1}$
$f_{0.1.0}$
$f_{0.0.1}$
$f_{0.0.0}$
19
Tuesday
18
Monday
AUGUST
child
$= 2^n - 1$
$= 2^1 - 1 = 1$
child
$= 2^n - 1 = 2^2 - 1 =$
child $= 2^n - 1$
$= 2^3 - 1 = 7$
NOTES




```
#Create a proicess 4 times

#include<stdio.h>

#include<unistd.h>

int main()

{

	int id1=fork();

	int id2=fork();

	int id3=fork();

	int id4=fork();

	printf("Hello world from ID: %d \t %d\t %d \t %d\n",id1,id2,id3,id4);

	return(0);

}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc p1.c

debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
```

```
Hello world from ID: 0   0      134     138
Hello world from ID: 128        0      132     136
Hello world from ID: 0   0      134     0
Hello world from ID: 128        129     0      140
Hello world from ID: 0   0      0      142
Hello world from ID: 128        0      132     0
Hello world from ID: 128        129     0      0
Hello world from ID: 0   130     0      137
Hello world from ID: 0   0      0      0
Hello world from ID: 0   130     133     135
Hello world from ID: 0   130     133     0
Hello world from ID: 0   130     0      0
Hello world from ID: 128        0      0      141
Hello world from ID: 128        129     131     139
Hello world from ID: 128        129     131     0
Hello world from ID: 128        0      0      0
```

```
debasis@LAPTOP-H3N6JCNE:~/process$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
debasis      9     8  0 10:29 tty1     00:00:00 -bash
debasis    213     9  4 10:59 tty1     00:00:00 ps -f
```

Q: What is zombie state?

Answer: A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.

A **zombie** process is a process in its terminated **state**. ... Until the parent function receives and acknowledges the message, the child function remains in a "**zombie**" **state**, meaning it has **executed** but not exited. A **zombie** process is also known as a **defunct process.**

Zombie processes are when a parent starts a child **process** and the child **process** ends, but the parent doesn't pick up the child's exit code.

Q:How do I see zombie processes?

Answer: Zombie processes can be found easily with the ps command. Within the ps output there is a STAT c Column which will show the **processes** current status, a **zombie process** will have Z as the status. In addition to the STAT column **zombies** commonly have the words <**defunct**> in the CMD column as well .

Also known as *"defunct"* or "*dead"* process – In simple words, a Zombie process is one that is dead but is present in the system's process table. Ideally, it should have been cleaned from the process table once it completed its job/execution but for some reason, its parent process didn't clean it up properly after the execution.

In order to kill a Zombie process, we need to identify it first. The following command can be used to find zombie processes:

```
$ ps aux | egrep 'Z|defunct'
```

Z in the STAT column and/or **[defunct]** in the last (COMMAND) column of the output would identify a Zombie process

Q: What is orphan and zombie 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.

An **orphan process** is a computer **process** whose parent **process** has finished or terminated, though it (child **process**) remains running itself.

A **zombie process** or **defunct process** is a **process** that has completed execution but still has an entry in the **process** table as its parent **process** didn't invoke an wait() system call.

Q:What is the parent process ID of an orphan process?

Answer: An **orphan process** is a **process** that is still executing, but whose **parent** has died. They do not become zombie **processes**; instead, they are **adopted by init** (**process ID** 1), which waits on its children. When a **parent** dies before its child, the child is automatically adopted by the original "init" **process** whose **PID** is 1.

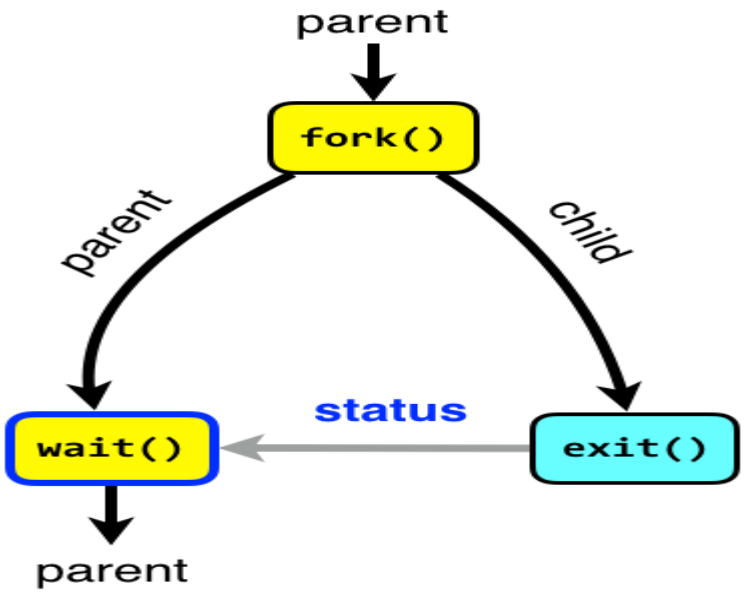
```
//defunct- zombie process : Child becomes Zombie as parent is sleeping
when child process exits.
#include<stdio.h>

#include<unistd.h>
```

```
#include<sys/wait.h>
#include<stdlib.h>
int main(int argc,char *argv[])
{
        printf("My ID: %d\n",(int)getpid());

        pid_t pid=fork();
        printf("Fork returned %d\n",(int)pid);

        if(pid<0)
        {
                printf("Fork failed.\n");
                exit(1);
        }
         if(pid==0)
         {       //child process
                 printf("I am the child with my pid: %d\n",(int)getpid());
                 exit(0);
        }
         else
         {
         //parent process
         printf("I am the parent waiting for child to end.\n");
         sleep(60);
         wait(NULL);
         printf("Parent ending.\n");
         }
         printf("Job Over with ID: %d.\n",(int)getpid());
         return 0;
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc frk_zmb.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out &
[1] 245
debasis@LAPTOP-H3N6JCNE:~/process$ My ID: 245
```

```
Fork returned 246
I am the parent waiting for child to end.
Fork returned 0
I am the child with my pid: 246
ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
debasis     55    54  0 18:07 tty2     00:00:00 -bash
debasis    245    55  0 18:52 tty2     00:00:00 ./a.out
debasis    246   245  0 18:52 tty2     00:00:00 [a.out] <defunct>
debasis    247    55  0 18:52 tty2     00:00:00 ps -f
```




```
//<defunct>- zombie process

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<stdlib.h>

int main(int argc,char *argv[])

{

      printf("My ID: %d\n",(int)getpid());
```

```
    pid_t pid=fork();
    printf("Fork returned %d\n",(int)pid);
    if(pid<0)
    {
        printf("Fork failed.\n");
        exit(1);
    }
     if(pid==0)
     {    //child process
          printf("I am the child with my pid: %d\n",(int)getpid());
     }
     else
     {
     //parent process
     printf("I am the parent waiting for child to end.\n");
     wait(NULL);
     printf("Parent ending.\n");
     }
     printf("Job Over with ID: %d.\n",(int)getpid());
     return 0;
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc frk_zmb.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
My ID: 190
Fork returned 191
I am the parent waiting for child to end.
```

```
Fork returned 0

I am the child with my pid: 191

Job Over with ID: 191.

Parent ending.

Job Over with ID: 190.
```

//orphan process : A C program to demonstrate Orphan Process. Parent process finishes execution while the child process is running. The child process becomes orphan.

```
#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

int main()

{

    pid_t pid;

    pid=fork();

    if(pid==0)

    {

        printf("\nChild process ID %d Running.",getpid());

        printf("\nBefore Parent dies..My PPID=%d.",getppid());

        sleep(60);

        printf("\nSince parent dies now MY PPID=%d",getppid());

        printf("\n Child/orphan process now completed job.");

    }

    else if (pid > 0)

    {    printf("\nParent process ID =%d.",getpid());

        printf("\nEnd of parent process.");

    }

return 0;
```

```
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~$ cc orphan_process.c
debasis@LAPTOP-H3N6JCNE:~$ ./a.out
Parent process ID =164.
End of parent process.Child process ID 165 Running.
debasis@LAPTOP-H3N6JCNE:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
debasis     55    54  0 18:07 tty2     00:00:00 -bash
debasis    165     1  0 18:32 tty2     00:00:00 ./a.out
debasis    166    55  0 18:32 tty2     00:00:00 ps -f
debasis@LAPTOP-H3N6JCNE:~$ ps -aux
USER       PID      %CPU %MEM      VSZ       RSS   TTY      STAT START   TIME COMMAND
root         1       0.0              0.0   8936   312 ?                  Ssl  17:55            0:00        /init
root        54        0.0  0.0   8936   224 tty2     Ss   18:07   0:00 /init
debasis     55      0.0  0.0  18340  3904 tty2     S    18:07   0:00 -bash
debasis    165      0.0  0.0  10536   380 tty2     S    18:32   0:00 ./a.out
debasis    167      0.0  0.0  18880  2040 tty2     R    18:32   0:00 ps -aux
after 60 seconds:
debasis@LAPTOP-H3N6JCNE:~$ Before Parent dies..My PPID=164.
Since parent dies now MY PPID=1
 Child/orphan process now completed job.
```

//Unpredictable output in fork

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

```
#include<time.h>
#include<sys/wait.h>
#include<sys/types.h>
int main(int argc,char *argv[] )
{
	int id=fork();
	int n;
	if(id==0)
		n=1;
	else
		n=6;

	int i;
	for(i=n;i<n+5;i++)
	{
		printf("%d\t",i);
		fflush(stdout);
	}

	printf("\n");
	return(0);
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc p4.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
6	7	8	1	9	2	10	3	11	4
5
```

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc p4.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
6     1     7     2     8     3     9     4     10     5     11
```

```
//predictable output: 1 2 3 4 5 6 7 8 9 10
#include<stdio.h>
#include<unistd.h>
#include<time.h>
#include<sys/wait.h>
#include<sys/types.h>

int main(int argc,char *argv[] )
{
    int id=fork();
    int n;
    if(id==0)
        n=1;
    else
        n=6;

    if(id>0)
        wait(NULL);

    int i;
    for(i=n;i<n+5;i++)
    {
```

```
            printf("%d\t",i);
            fflush(stdout);
      }

      if(id>0)
      {
            printf("Job over.\n");
            printf("\n");
      }
      return(0);
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc p5.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
1     2     3     4     5     6     7     8     9     10     Job over.
```

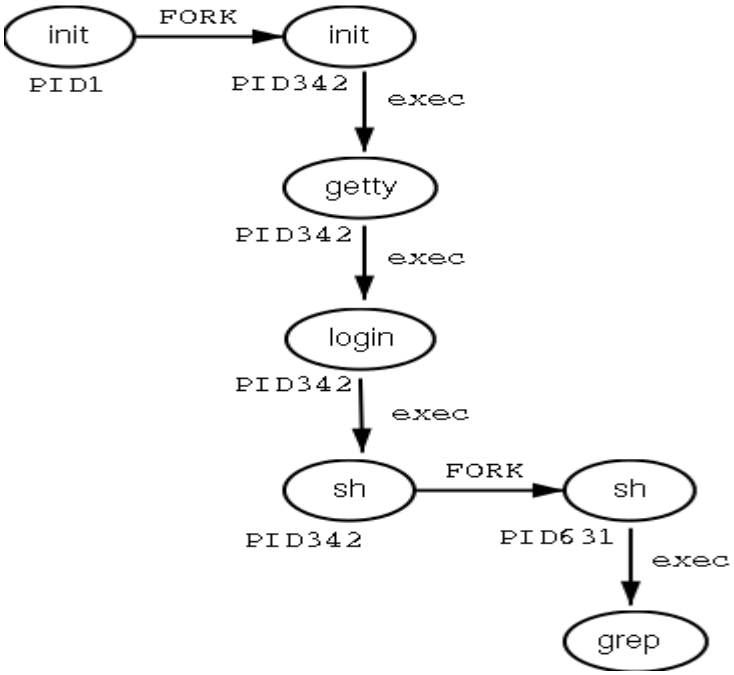
A new process is created because an existing process makes an exact copy of itself. This child process has the same environment as its parent, only the process ID number is different. This procedure is called *forking*.

After the forking process, the address space of the child process is overwritten with the new process data. This is done through an *exec* call to the system.

The *fork-and-exec* mechanism thus switches an old command with a new, while the environment in which the new program is executed remains the same, including configuration of input and output devices, environment variables and priority. This mechanism is used to create all UNIX processes, so it also applies to the Linux operating system. Even the first process, **init**, with process ID 1i, s forked during the boot procedure in the so-called ***bootstrapping* procedure**.

This scheme illustrates the fork-and-exec mechanism. The process ID changes after the fork procedure:

Figure : Fork-and-exec mechanism



//Executing command in c with execlp

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main(int argc, char *argv[])
{
	printf("Welcome...\n");
	execlp("ls","ls”,NULL);
	printf("\nJob over.\n");
	return(0);
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
Welcome...
a.out  exec1.c  p1.c  p2.c  p3.c  p4.c  p5.c
```

//Executing command in c with execl or execlp

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main(int argc, char *argv[])
{
	printf("Welcome...\n");
	execl("/bin/date","date",(char*)NULL);
	execlp("ls","ls","-l",(char*)NULL);
	printf("\nJob over.\n");
	return(0);
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc exec11.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
Welcome...
Wed Dec 30 11:57:02 IST 2020
```

```
//Executing command in c with execl or execlp
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main(int argc, char *argv[])
{
	printf("Welcome...\n");
//	execl("/bin/date","date",(char*)NULL);
//	execlp("ls","ls","-l",(char*)NULL);
```

```
        execl("/bin/ls","ls","-l","exec11.c",(char*)NULL);
        printf("\nJob over.\n");
        return(0);
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc exec11.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
Welcome...
-rw-r--r-- 1 debasis debasis 343 Dec 30 12:03 exec11.c
```

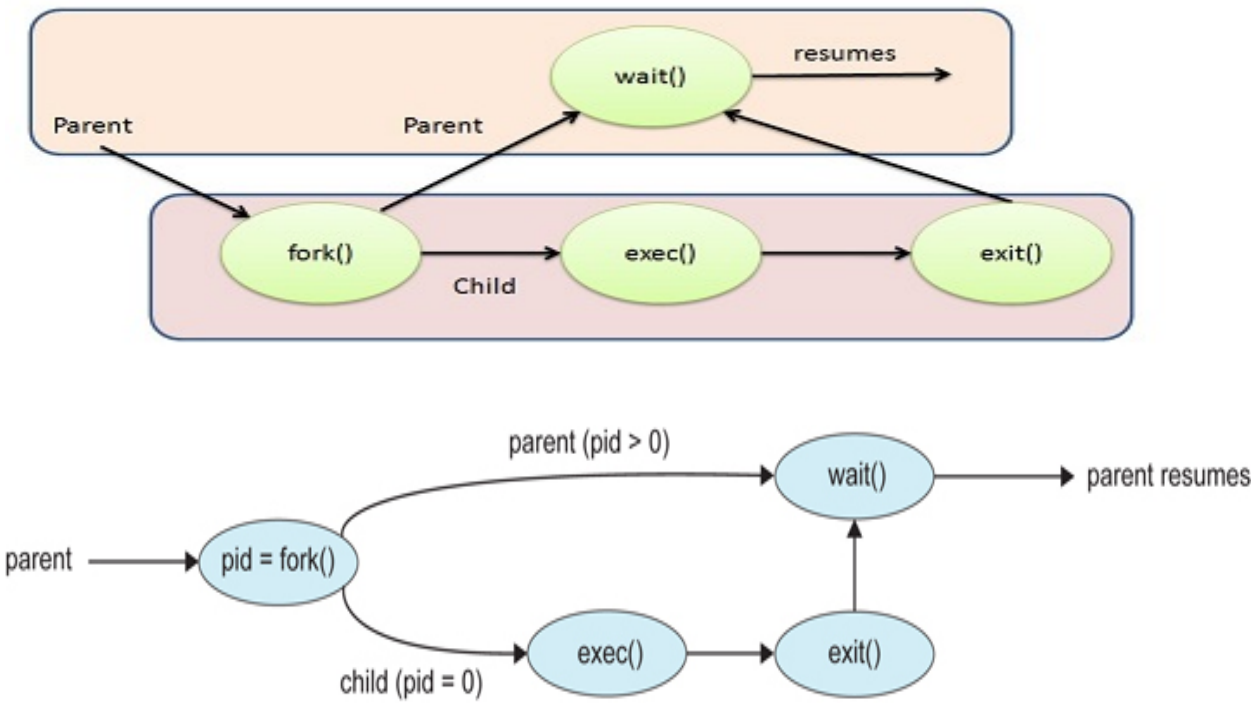


Figure - Process creation using the fork( ) system call

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-1);
    }
    else if (pid == 0) {/* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
      exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.

Figure -Creating a separate process using the UNIX fork( ) system call.

```
//Executing command in c with execlp

#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>
```

```
int main(int argc, char *argv[])
{
	int pid=fork();
	if(pid==-1)
		return(-1);
	if(pid==0)
	{
	printf("Welcome...\n");
	execlp("ls","ls", "-l",NULL);
	printf("\nJob over:No message.\n");
	}
	else
	{
		wait(NULL);
		printf("\n From main process.\n");
	}
	printf("Job over.");
	return(0);
}
```

Output:

```
debasis@LAPTOP-H3N6JCNE:~/process$ cc exec2.c
debasis@LAPTOP-H3N6JCNE:~/process$ ./a.out
Welcome...
total 52
-rw-r--r-- 1 debasis debasis   885 Dec 27 13:17 "'"
-rw-r--r-- 1 debasis debasis   664 Dec 27 19:32  MsgPassing1.c
-rw-r--r-- 1 debasis debasis  1180 Dec 27 19:36  MsgPassing2.c
```

```
-rw-r--r-- 1 debasis debasis  1944 Dec 27 19:40  MsgPassing3.c
-rwxr-xr-x 1 debasis debasis 16872 Dec 30 11:46  a.out
-rw-r--r-- 1 debasis debasis   225 Dec 30 11:45  exec1.c
-rw-r--r-- 1 debasis debasis   424 Dec 30 11:46  exec2.c
-rw-r--r-- 1 debasis debasis   611 Dec 29 15:46  exit_statusEx.c
-rw-r--r-- 1 debasis debasis   874 Dec 29 16:06  exit_statusEx1.c
-rw-r--r-- 1 debasis debasis   205 Dec 27 10:45  p1.c
-rw-r--r-- 1 debasis debasis   139 Dec 27 10:51  p2.c
-rw-r--r-- 1 debasis debasis   398 Dec 27 11:07  p3.c
-rw-r--r-- 1 debasis debasis   316 Dec 27 11:55  p4.c
-rw-r--r-- 1 debasis debasis   413 Dec 27 11:54  p5.c
-rw-r--r-- 1 debasis debasis   165 Dec 27 12:35  p6.c
-rw-r--r-- 1 debasis debasis  1181 Dec 27 22:46  pipe.c
-rw-r--r-- 1 debasis debasis   587 Dec 27 12:59  pipe1.c
-rw-r--r-- 1 debasis debasis   885 Dec 27 13:15  pipe2.c


 From main process.
Job over.
```

```
WAIT(2)                    Linux Programmer's Manual                    WAIT(2)

NAME

     wait, waitpid, waitid - wait for process to change state

SYNOPSIS

     #include <sys/types.h>

     #include <sys/wait.h>

     pid_t wait(int *wstatus);
```

pid_t waitpid(pid_t pid, int *wstatus, int options);

WIFEXITED(wstatus)

returns true if the child terminated normally, that is, by calling exit(3) or _exit(2), or by returning from main().

WEXITSTATUS(wstatus)

returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit(3) or _exit(2) or as the argument for a return statement in main(). This macro should be employed only if WIFEXITED returned true.

EXIT STATUS

0 One or more processes matched the criteria. For pkill the process must also have been successfully signalled.

1 No processes matched or none of them could be signalled.

2 Syntax error in the command line.

3 Fatal error: out of memory etc.

NOTES

The process name used for matching is limited to the 15 characters present in the output of /proc/pid/stat. Use the -f option to match against the complete command line, /proc/pid/cmdline.

Background jobs (& operator, nohup command):

Nohup Command in Linux:

The meaning of **nohup** is '**no hangup**'. Normally, when we log out from the system then all the running programs or processes are hangup or terminated. If you want to run any program after log out or exit from Linux operating system then you have to use nohup command. There are many programs that require many hours to complete. We don't need to log in for long times to complete the task of the command. We can keep these type of programs running in the background by using nohup command and check the output later.

NAME
nohup - run a command immune to hangups, with output to a non-tty

SYNOPSIS/Syntax:
nohup COMMAND [ARG]...
nohup OPTION

DESCRIPTION
Run COMMAND, ignoring hangup signals.

--help display this help and exit

--version
output version information and exit

If standard input is a terminal, redirect it from an unreadable file. If standard output is a terminal, append output to 'nohup.out' if possible, '$HOME/nohup.out' otherwise. If standard error is a terminal, redirect it to standard output. To save output to FILE, use 'nohup COMMAND > FILE'.

Example-1: Using nohup command without '&'

When you run nohup command without '**&**' then it returns to shell command prompt immediately after running that particular command in the background.

```
debasis@LAPTOP-H3N6JCNE:~/process$ cat sleep5.sh
#!/bin/bash
echo "waiting for 5 seconds"
sleep 5
echo job over
```

```
debasis@LAPTOP-H3N6JCNE:~/process$ nohup bash sleep5.sh
output:
nohup: ignoring input and appending output to 'nohup.out'

NOTE:
```
The output of the **nohup** command will write in **nohup.out** the file if any redirecting filename is not mentioned in **nohup** command.

```
debasis@LAPTOP-H3N6JCNE:~/process$ cat nohup.out
output:
waiting for 5 seconds
job over
```

You can execute the command in the following way to redirect the output to the **outputfile.txt** file:

```
debasis@LAPTOP-H3N6JCNE:~/process$ nohup bash sleep5.sh > outputfile.txt
nohup: ignoring input and redirecting stderr to stdout

debasis@LAPTOP-H3N6JCNE:~/process$ cat outputfile.txt
waiting for 5 seconds
job over
```

Example-2: Using nohup command with '&'

When **nohup** command use with '**&**' then it doesn't return to shell command prompt after running the command in the background.

```
debasis@LAPTOP-H3N6JCNE:~/process$ nohup bash sleep5.sh &

[1] 177

debasis@LAPTOP-H3N6JCNE:~/process$ nohup: ignoring input and appending output to
'nohup.out'
```

But if you want you can return to shell command prompt by typing '**fg**'

```
debasis@LAPTOP-H3N6JCNE:~/process$ nohup: ignoring input and appending output to
'nohup.out'
fg
-bash: fg: job has terminated
[1]+  Done                    nohup bash sleep5.sh
```

Example-3: Using nohup command to run multiple commands in the background

You can run multiple commands in the background by using nohup command. In the following command, mkdir and ls -l command are executed in the background by using nohup and bash commands. You can get the output of the commands by checking outputfile.txt file.

```
debasis@LAPTOP-H3N6JCNE:~/process$ nohup bash -c "mkdir
DG && ls -l" > >outputfile.txt
output:
debasis@LAPTOP-H3N6JCNE:~/process$ cat outputfile.txt | more
total 68
-rw-r--r-- 1 debasis debasis   885 Dec 27 13:17 signal
drwxr-xr-x 1 debasis debasis  4096 Jan 22 12:30 DG
-rw-r--r-- 1 debasis debasis   664 Dec 27 19:32 MsgPassing1.c
-rw-r--r-- 1 debasis debasis  1180 Dec 27 19:36 MsgPassing2.c
```

Example-4: Start any process in the background by using nohup

When any process starts and the user closes the terminal before completing the task of the running process then the process stops normally. If run the process with nohup then it will able to run the process in the background without any issue.

debasis@LAPTOP-H3N6JCNE:~/process$ **nohup ping -i 15 www.google.com &**
[2] 253

For example, if you run the **ping** command normally then it will terminate the process when you close the terminal. Example:

debasis@LAPTOP-H3N6JCNE:~/process$ **ping -i 15** www.google.com

```
COMMAND NAME : ping
      ping - send ICMP ECHO_REQUEST to network
hosts
options:
(i)-c count :
         Stop after sending count ECHO_REQUEST
         packets. With deadline option, ping wait for
```

count ECHO_REPLY packets, until the timeout expires.

(ii) -i interval :

Wait interval seconds between sending each packet. The default is to wait for one second between each packet normally, or not to wait in flood mode. Only super-user may set interval to values less than 0.2 seconds.

debasis@LAPTOP-H3N6JCNE:~/process$ ping -i 10 -c 2 www.google.com

output:

PING www.google.com(maa03s31-in-x04.1e100.net (2404:6800:4007:812::2004)) 56 data bytes
64 bytes from maa03s31-in-x04.1e100.net (2404:6800:4007:812::2004): icmp_seq=2 ttl=117 time=98.8 ms

--- www.google.com ping statistics ---
2 packets transmitted, 1 received, 50% packet loss, time 12433ms
rtt min/avg/max/mdev = 98.752/98.752/98.752/0.000 ms

debasis@LAPTOP-H3N6JCNE:~/process$ pgrep -a ping

output: [No process are there in the background]

debasis@LAPTOP-H3N6JCNE:~/process$ nohup ping -i 15 www.google.com &
[2] 253

Close the ubuntu ,again open:

debasis@LAPTOP-H3N6JCNE:~$ pgrep -a ping

253 ping -i 15 www.google.com [Process running in the background]

Command NAME: **pgrep** -a : List the full command line as well as the process ID.

debasis@LAPTOP-H3N6JCNE:~$ **pgrep -a ping**
output:
253 ping -i 15 www.google.com

OR : debasis@LAPTOP-H3N6JCNE:~$ **ps -aux**
output:

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.0 8936 312 ? Ssl 11:02 0:00 /init
debasis 253 0.0 0.0 17008 1472 ? S 12:48 0:00 ping -i 15 www.google.com
root 256 0.0 0.0 8936 224 tty3 Ss 12:50 0:00 /init
debasis 258 0.4 0.0 18212 3660 tty3 S 12:50 0:00 -bash
debasis 298 0.0 0.0 18880 2036 tty3 R 12:51 0:00 ps -aux

You can stop any background process by running kill command. Just run kill command with particular process id which is running. Here, the process id of the running process is 253. **Run kill command in 253 to terminate the process.**
debasis@LAPTOP-H3N6JCNE:~$ **kill 253**

Set/Reduce priority (nice):

Every process on Linux has a nice value. The value of nice determines which process has higher priorities and which has lower. Nice value can be between -20 to 19. A process with the nice value of -20 will have the highest priority and will use the most CPU cycles.

There are two ways to set the nice value of a process. You can either start a process with the **nice** command to set a nice value while starting the process. Or you can use the **renice** command to set a nice value after a process has started.
To set a nice value when you start a process, run the process as follows:

```
$ nice -n NICE_VALUE COMMAND_TO_RUN
```

```
debasis@LAPTOP-H3N6JCNE:~$ nice -n 10 sleep 400 &
output:
[1] 102
debasis@LAPTOP-H3N6JCNE:~$ ps -fl
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S debasis     9     8     0  80   0    -     4586 -      16:21          tty1     00:00:00 -bash
0 S debasis    102    9     0  70   42   94967286 - 3819 - 16:59          tty1     00:00:00 sleep
400
0 R debasis    103    9     0  80   0 -  4662 -      16:59                tty1     00:00:00 ps -fl
```

```
debasis@LAPTOP-H3N6JCNE:~$ top
top - 17:00:47 up 39 min,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:   5 total,   1 running,   4 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.0 us,  6.2 sy,  0.0 ni, 90.0 id,  0.0 wa,  0.7 hi,  0.0 si,  0.0 st
MiB Mem :   4005.2 total,    713.9 free,   3067.3 used,    224.0 buff/cache
MiB Swap:  12288.0 total,  11759.4 free,    528.6 used.    807.3 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+
COMMAND
    1 root      20   0    8936    312    268 S   0.0   0.0   0:00.28 init
    8 root      20   0    8936    224    180 S   0.0   0.0   0:00.01 init
    9 debasis   20   0   18344   3844   3736 S   0.0   0.1   0:00.62 bash
  102 debasis   10 42+   15276    824    684 S   0.0   0.0   0:00.01 sleep
  104 debasis   20   0   18900   2132   1524 R   0.0   0.1   0:00.03 top
```

Now if you wish to change the nice value of your existing processes, then all you need is the process ID (PID) of the process of which you want to change the nice value. You can use the **ps -aux** command or the **top** command to find the process ID or PID.

Then you can run **renice** command as follows to change the nice value of an existing process:

```
$ sudo renice -n NEW_NICE_VALUE -p PROCESS_PID
```

```
debasis@LAPTOP-H3N6JCNE:~$ sudo renice -n 5 -p 102
102 (process ID) old priority 10, new priority 5
debasis@LAPTOP-H3N6JCNE:~$ ps -fl
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY
```

```
TIME CMD
0 S debasis      9     8  0  80   0 -  4586 -      16:21 tty1     00:00:00 -bash
0 S debasis    102     9  0  75 4294967291 - 3819 - 16:59 tty1    00:00:00 sleep
400
0 R debasis    109     9  0  80   0 -  4662 -      17:02 tty1     00:00:00 ps -fl
```

Using signals to kill process:

COMMAND NAME: kill

kill - send a signal to a process / terminate a process.

SYNOPSIS

kill [options] <pid> [...]

DESCRIPTION

The default signal for kill is TERM. Use -l or -L to list available signals. Particularly useful signals include HUP, INT, KILL, STOP, CONT, and 0. Alternate signals may be specified in three ways: -9, -SIGKILL or -KILL.

A PID of -1(one) is special; it indicates all processes except the kill process itself and init.

EXAMPLES:

kill -9 -1 # Kill all processes you can kill.

kill 123 543 2341 3453 # Send the default signal, SIGTERM, to all those processes 123,543,2341,3453.

kill -L # List the available signal choices in a nice table.

```
#To kill running user processes:
debasis@LAPTOP-H3N6JCNE:~$ kill -9 -1          #  [ -1 →one ]
[1]-  Killed                 nice -n 14 sleep 400
[2]+  Killed                 top
debasis@LAPTOP-H3N6JCNE:~$
```

The process can be sent a SIGKILL signal in three ways:

```
kill -s KILL 590
kill -KILL 590
kill -9 590
```

```
$sleep 300 &
```

```
debasis@LAPTOP-H3N6JCNE:~$ kill -KILL 176
[1]+  Killed                  sleep 300

debasis@LAPTOP-H3N6JCNE:~$ kill -9 178
[1]+  Killed                  sleep 300

debasis@LAPTOP-H3N6JCNE:~$ kill -s kill 173
```

A process can be sent a SIGTERM signal in four ways (the process ID is '590' in this case):

```
kill 590
kill -s TERM 590
kill -TERM 590
kill -15 590
```

The `kill` command sends a signal to specified processes or process groups, causing them to act according to the signal. When the signal is not specified, it defaults to `-15` (-TERM).

The most commonly used signals are:

- `1` (`HUP`) - Reload a process.
- `9` (`KILL`) - Kill a process.
- `15` (`TERM`) - Gracefully stop a process.

To get a list of all available signals, invoke the command with the `-l` option:

```
kill -l
```

```
debasis@LAPTOP-H3N6JCNE:~$ sleep 300 &
[1] 204
debasis@LAPTOP-H3N6JCNE:~$ kill -15 204
[1]+  Terminated              sleep 300     [ #stop a process ]

debasis@LAPTOP-H3N6JCNE:~$ sudo kill -1 207
[1]+  Hangup                  sleep 300 #Reload a process
```

Q: **SIGTERM vs SIGKILL: Why should you prefer using SIGTERM over SIGKILL?**

Answer:

- SIGTERM(15) gracefully kills the process whereas SIGKILL(9) kills the process immediately or `kill -9` is a *sure kill*.
- SIGTERM signal can be handled, ignored and blocked but SIGKILL cannot be handled or blocked.
- SIGTERM **doesn't kill the child processes**. SIGKILL kills the child processes as well.

#To display LAST background process ID:

- debasis@LAPTOP-H3N6JCNE:~$ echo $!
  Output: 210

Sending job to background (bg) and foreground (fg):

Whats a job in Linux

A job is a process that the shell manages. Each job is assigned a sequential job ID. Because a job is a process, each job has an associated PID. There are three types of job statuses:
1. **Foreground**: When you enter a command in a terminal window, the command occupies that terminal window until it completes. This is a foreground job.
2. **Background**: When you enter an ampersand (&) symbol at the end of a command line, the command runs without occupying the terminal window. The shell prompt is displayed immediately after you press Return. This is an example of a background job.
3. **Stopped**: If you press Control + Z for a foreground job, or enter the stop command for a background job, the job stops. This job is called a stopped job.

Listing jobs (jobs), Suspend job, Kill a job:

Job Control Commands

Job control commands enable you to place jobs in the foreground or background, and to start or stop jobs. The table describes the job control commands.

Option	Description
jobs	Lists all jobs
bg %n	Places the current or specified job in the background, where n is the job ID

fg %n	Brings the current or specified job into the foreground, where n is the job ID
Ctrl-Z	Stops the foreground job and places it in the background as a stopped job

```
debasis@LAPTOP-H3N6JCNE:~$ cat>a.txt
this is my file
bg
^Z
[1]+  Stopped                 cat > a.txt
debasis@LAPTOP-H3N6JCNE:~$ ps -fl
F  S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
0 S debasis      9     8  0  80   0 -  4586 -      16:21 tty1     00:00:01 -bash
0 T debasis    255     9  0  80   0 -  3855 -      18:25 tty1     00:00:00 cat
0 R debasis    256     9  0  80   0 -  4662 -      18:26 tty1     00:00:00 ps -fl

debasis@LAPTOP-H3N6JCNE:~$ jobs
[1]+  Stopped                 cat > a.txt

debasis@LAPTOP-H3N6JCNE:~$ fg %1
cat > a.txt

This is my jobs operation test.
debasis@LAPTOP-H3N6JCNE:~$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
debasis      9     8  0 16:21 tty1     00:00:01 -bash
debasis    257     9  0 18:27 tty1     00:00:00 ps -f

debasis@LAPTOP-H3N6JCNE:~$ cat a.txt
this is my file
bg
This is my jobs operation test.
```

```
debasis@LAPTOP-H3N6JCNE:~$ bg %2
[2]+ cat > b.txt &
[2]+  Stopped                 cat > b.txt
```

```
debasis@LAPTOP-H3N6JCNE:~$ jobs
[1]-  Stopped                 cat > a.txt
[2]+  Stopped                 cat > b.txt
```

```
$sleep 100&
$sleep 200&
$sleep 200&

debasis@LAPTOP-H3N6JCNE:~$ ps fx
  PID TTY      STAT   TIME COMMAND
    9 tty1     S      0:01 -bash
  259 tty1     T      0:00  \_ cat
  260 tty1     T      0:00  \_ cat
  270 tty1     S      0:00  \_ sleep 100
  271 tty1     S      0:00  \_ sleep 200
  272 tty1     T      0:00  \_ sleep 300
  275 tty1     R      0:00  \_ ps fx
```

To suspend the process running in the background, use:

kill -STOP %job_id

```
debasis@LAPTOP-H3N6JCNE:~$ sleep 300 &
[6] 276
debasis@LAPTOP-H3N6JCNE:~$ jobs
[1]   Stopped                 cat > a.txt
[2]-  Stopped                 cat > b.txt
[5]+  Stopped                 sleep 300
[6]   Running                 sleep 300 &
```

To suspend the process running in the background, use:

kill -STOP %job_id

```
debasis@LAPTOP-H3N6JCNE:~$ kill -STOP %6

[6]+  Stopped                 sleep 300
debasis@LAPTOP-H3N6JCNE:~$ jobs
[1]   Stopped                 cat > a.txt
[2]   Stopped                 cat > b.txt
[5]-  Stopped                 sleep 300
[6]+  Stopped                 sleep 300

debasis@LAPTOP-H3N6JCNE:~$ kill -CONT %6 [ #Resume/continue process ]
```

Interactive:

- `Ctrl`+`z` will suspend the currently foregrounded program
- `bg` will background the most recently suspended program (use `bg %2` with the job number, which you can check with `jobs`)
- `fg` will foreground the most recently suspended program

Non-interactive:

If you're in a different shell instance (or a different user, sometimes including `sudo` commands), you likely won't be able to use job numbers.

```
kill -STOP $PID  # suspend
kill -CONT $PID  # continue (resume)
```

Difference between Process and Thread:

A process is an active program i.e. a program that is under execution. It is more than the program code as it includes the program counter, process stack, registers, program code etc. Compared to this, the program code is only the text section.

A thread is a lightweight process that can be managed independently by a scheduler. It improves the application performance using parallelism. A thread shares information like data segment, code segment, files etc. with its peer threads while it contains its own registers, stack, counter etc.

The major differences between a process and a thread are given as follows:

Comparison Basis	Process	Thread
Definition	A process is a program under execution i.e an active program	A thread is a lightweight process that can be managed independently by a scheduler
Context switching time	Processes require more time for context switching as they are more heavy	Threads require less time for context switching as they are lighter than processes.
Memory Sharing	Processes are totally	A thread may share some

	independent and don’t share memory.	memory with its peer threads.
Communication	Communication between processes requires more time than between threads	Communication between threads requires less time than between processes .
Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation	Threads require less time for creation.
Time for termination	Processes require more time for termination.	Threads require less time for termination.

Deadlock	**Starvation**
The deadlock situation occurs when one of the processes got blocked.	Starvation is a situation where all the low priority processes got blocked, and the high priority processes execute.

Deadlock is an infinite process.	Starvation is a long waiting but not an infinite process.
Every Deadlock always has starvation.	Every starvation does n't necessarily have a deadlock.
Deadlock happens then Mutual exclusion, hold and wait. Here, preemption and circular wait do not occur simultaneously.	It happens due to uncontrolled priority and resource management.