Process Basic

A process is simply an instance of a running program. A process is said to be born when the program starts execution and remains alive as long as the program is active. After execution is complete, the process is said to die.

Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the **ls** command to list directory contents, you started a process. A process, in simple terms, is an instance of a running program.

When UNIX runs a process it gives each process a unique number - a **process ID (pid)**.

The C function **int getpid( )** will return the **pid** of process that called this function.

Processes are the primitive units for allocation of system resources. Each process has its own **address space** and (usually) one thread of control. **A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.**

Processes are organized hierarchically. Each process has a **parent** process which explicitly arranged to create it. The processes created by a given parent are called its **child** processes.

A child inherits many of its attributes from the parent process.

Every process in a UNIX system has the following attributes:

- **some code**
- **some data**
- **a stack**
- **a unique process id number (PID)**

Question2:

Process Creation Program

```
#include <stdio.h>
#include <sys/wait.h> /* contains prototype for wait */

int main(void)
{
int pid;
int status;
printf("Hello World!\n");

pid = fork( );

if (pid == -1) /* check for error in fork */
{
perror("bad fork");
exit(1);
}

if (pid == 0)
printf("I am the child process.\n");
else
{
wait(&status);  /* parent waits for child to finish */
printf("I am the parent process.\n");
}
}
```

Sample Output :

```
$gcc prog2.c
$./a.out
Hello World!
I am the child process.
I am the parent process.
```

Used ps command to List Running Processes

One of the most commonly used flags for ps is the **-f** (f for full) option, which provides more information as shown in the following example:

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fileds displayed by ps -f command:

Column	Description
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.
CMD	The command that started this process.

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.

Parent and Child Processes:

Each unix process has two ID numbers assigned to it: Process ID (pid) and Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check ps -f example where this command listed both process ID and parent process ID.

Init Process

In Unix-based computer operating systems, **init** (short for *initialization*) is the first process started during booting of the computer system. Init is a daemon process that continues running until the system is shut down. It is the direct or indirect ancestor of all other processes.

Getty Process

`getty`, short for "get teletype", is a Unix program running on a host computer that manages physical or virtual terminals (TTYs). When it detects a connection, it prompts for a username and runs the 'login' program to authenticate the user. It is usually called by init.

The getty process does nothing until a terminal is connected to the port.When the terminal gets connected getty process sends the signal of login to the terminal and waits for the feedback. After the userid is types, getty

process starts a new process called login and after password is verified the control is transferred to the shell.

Daemon Process

In Unix and other multitasking computer operating systems, a **daemon is a computer program that runs as a background process**, rather than being under the direct control of an interactive user. Typically 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.

Zombie process

A **zombie process** is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status.

A process that terminates cannot leave the system until its parent accepts its return code. If its parent process is already dead, it'll already have been adopted by the "**init**" process, which always accepts its children's return codes. However, **if a process's parent is alive but never executes a wait ( ), the process's return code will never be accepted and the process will remain a *zombie***.
The following program created a zombie process, which was indicated in the output from the **ps** utility. When the parent process is killed, the child was adopted by "**init**" and allowed to rest in peace.

```
#include <stdio.h>
main ( )
{
        int   pid ;

        pid = fork ( ) ;          /* Duplicate. Child and parent continue from here */

        if ( pid != 0 )  /* pid is non-zero, so I must be the parent */
        {
             while (1) /* Never terminate and never execute a wait ( ) */
              sleep (100) ;    /* stop executing for 100 seconds */
```

```
        }
        else                    /* pid is zero, so I must be the child */
        {
                exit (42) ;     /* exit with any number */
        }
}
```

The output is:

```
$ a.out   &   execute the program in the background
[1] 5186

$ ps  obtain process status
  PID TT STAT   TIME COMMAND
 5187 p0 Z      0:00 <exiting> the zombie child process
 5149 p0 S      0:01 -csh (csh)          the shell
 5186 p0 S      0:00 a.out               the parent process
 5188 p0 R      0:00 ps

$ kill 5186      kill the parent process
[1]    Terminated          a.out

$ ps  notice that the zombie is gone now
 PID TT STAT   TIME COMMAND
 5149 p0 S      0:01 -csh (csh)
 5189 p0 R      0:00 ps
```

orphan process

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 `wait`s on its children.

When a **parent dies before its child**, the child is automatically adopted by the original "**init**" process whose **PID** is **1**.

To, illustrate this insert a **sleep** statement into the child's code. This ensured that the parent process terminated before its child.

```
#include <stdio.h>
main ( )
{
   int   pid ;
```

```
  pid = fork ( ) ;    /* Duplicate. Child and parent continue from here */
  if ( pid != 0 )     /* pid is non-zero, so I must be the parent */
  {
     printf ("I'am the parent process with PID %d and PPID %d.\n",
          getpid ( ), getppid ( ) ) ;
     printf ("My child's PID is %d\n", pid ) ;
   }
  else                    /* pid is zero, so I must be the child */
  {
     sleep (4) ;                    /* make sure that the parent terminates first */
     printf ("I'am the child process with PID %d and PPID %d.\n",
               getpid ( ), getppid ( ) ) ;
   }
        printf ("PID %d terminates.\n", getpid ( ) ) ;
}
```

The output is:

```
I'am the parent process with PID 5100 and PPID 5011.
My child's PID is 5101
PID 5100 terminates.    /* Parent dies */
I'am the child process with PID 5101 and PPID 1.
            /* Orphaned, whose parent process is "init" with pid 1 */
PID 5101 terminates.
```

---

fork() system call

To create a new process, you must use the fork() system call.

The prototype for the fork() system call is:

int fork()

fork() causes the UNIX system to create a new process, called the "child process", with a new process ID. The *contents* of the child process are identical to the *contents* of the parent process.

fork() returns zero in the child process and non-zero(the child's process ID) in the parent process.

wait() system call

- You can control the execution of child processes by calling wait() in the parent.
- wait() forces the parent to suspend execution until the child is finished.
- wait() returns the process ID of a child process that finished.

- If the child finishes before the parent gets around to calling wait(), then when wait() is called by the parent, it will return immediately with the child's process ID.

exit() system call

`void exit(int status)` -- terminates the process which calls this function and returns the exit `status` value.

What is a System Call?

In computing, a **system call** is how a program requests a service from an operating system's kernel. This may include hardware related services (e.g. accessing the hard disk), creating and executing new processes, and communicating with integral kernel services (like scheduling). System calls provide an essential interface between a process

System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls.

Use of System Calls:

UNIX system calls are used to manage the file system, control processes, and to provide inter-process communication.

Major System Calls:

GENERAL CLASS	SPECIFIC CLASS	SYSTEM CALL
File Structure Related Calls	Creating a Channel	creat( ), open( ), close( )
	Input/Output	read( ), write( )
	Random Access	lseek( )
	Channel Duplication	dup( )
	Aliasing and Removing Files	link( ), unlink( )
	File Status	stat( ), fstat( )
	Directory Information	opendir( ), readdir( )
	Access Control	access( ), chmod( ), chown( ), umask( )
	Device Control	ioctl( )
Process Related Calls	Process Creation and Termination	exec( ), fork( ), wait( ), exit( )
	Process Owner and Group	getuid( ), geteuid( ), getgid( ), getegid( )
	Process Identity	getpid( ), getppid( )
	Process Control	signal( ), kill( ), alarm( )
	Change Working Directory	chdir( )
Inter Process Communication	Pipelines	pipe( )
	Messages	msgget( ), msgsnd( ), msgrcv( ), msgctl( )
	Semaphores	semget( ), semop( )
	Shared Memory	shmget( ), shmat( ), shmdt( )

File Structure Related System Calls

The file structure related system calls available in the UNIX system let you create,open, and close files, read and write files, randomly access files, alias and remove files,get information about files, check the accessibility of files, change protections,owner, and group of files, and control devices.

(1)

creat( ) system call

The prototype for the **creat( )** system call is:

```
int creat(file_name, mode)
char *file_name;
int mode;
```

where file_name is pointer to a null terminated character string that names the file and mode defines the file's access permissions. The mode is usually specified as an octal number such as 0666 that would mean read/write permission for owner, group, and others or the mode may also be entered using manifest constants defined in the "**/usr/include/sys/stat.h**" file. If the file named by file_name does not exist, the UNIX system creates it with the specified mode permissions. However, if the file does exist, its contents are discarded and the mode value is ignored. The permissions of the existing file are retained. Following is an example of how to use **creat( )**:

```
int fd;
fd = creat("datafile.dat", S_IREAD | S_IWRITE);
```

(2) open() system call

open ( ) allows you to open or create a file for reading and/or writing.

int open (char **filename*, int *mode* [, int *permissions*])

(3) close()

To close a channel, use the **close( )** system call. The prototype for the **close( )** system call is:

```
int close(file_descriptor)
int file_descriptor;
```

where **file_descriptor** identifies a currently open channel. **close( )** fails if **file_descriptor** does not identify a currently open channel.

(4) **read() & write() system calls**

The read() system call does all input and the write() system call does all output.

Process Related System Calls

The UNIX system provides several system calls to

(1) create and end program,
(2) to send and receive software interrupts,
(3) to allocate memory, and to do other useful jobs for a process.

Four system calls are provided for creating a process, ending a process, and waiting for a process to complete.

These system calls are fork(), the "exec" family, wait(), and exit().

(1) fork() system call

To create a new process, you must use the fork() system call.

The prototype for the fork() system call is:

int fork()

fork() causes the UNIX system to create a new process, called the "child process", with a new process ID. The contents of the child process are identical to the contents of the parent process.

fork() returns zero in the child process and non-zero(the child's process ID) in the parent process.

(2) exec

Forking creates a process but is not enough to run a new program. To do that, the forked child needs to overwrite its own image with the code and data of the new program. This mechanism is called exec,and the child process is said to exec a new program.

(3) wait() system call

1. You can control the execution of child processes by calling wait() in the parent.
2. wait() forces the parent to suspend execution until the child is finished.
3. wait() returns the process ID of a child process that finished.

- If the child finishes before the parent gets around to calling wait(), then when wait() is called by the parent, it will return immediately with the child's process ID.

exit() system call

`void exit(int status)` -- terminates the process which calls this function and returns the exit `status` value.

---

ps command

ps command is used to report the process status. ps is the short name for Process Status.

The *ps* (i.e., *process status*) command is used to provide information about the currently running *processes*, including their *process identification numbers* (PIDs).

The basic syntax of ps is

```
ps [options]
```

Option	Description
-a	Displays all processes on a terminal, with the exception of group leaders.
-c	Displays scheduler data.
-d	Displays all processes with the exception of session leaders.
-e	Displays all processes.
-f	Displays a full listing.
-g*list*	Displays data for the *list* of group leader IDs.
-j	Displays the process group ID and session ID.
-l	Displays a long listing
-p*list*	Displays data for the *list* of process IDs.
-s*list*	Displays data for the *list* of session leader IDs.
-t*list*	Displays data for the *list* of terminals.
-u*list*	Displays data for the *list* of usernames.

Examples

```
$ps -ef
```

```
$ps -aux
```

What are signals.

Signals, to be short, are various notifications sent to a process in order to notify it of various "important" events. By their nature, they interrupt whatever the process is doing at this minute, and force it to handle them immediately. Each signal has an integer number that represents it (1, 2 and so on), as well as a symbolic name that is usually defined in the file /usr/include/signal.h

Sending Signals To Processes

(1) Sending Signals Using The Keyboard

The most common way of sending signals to processes is using the keyboard. There are certain key presses that are interpreted by the system as requests to send signals to the process with which we are interacting:

Ctrl-C

Pressing this key causes the system to send an `INT` signal (`SIGINT`) to the running process. By default, this signal causes the process to immediately terminate.

Ctrl-Z

Pressing this key causes the system to send a `TSTP` signal (`SIGTSTP`) to the running process. By default, this signal causes the process to suspend execution.

Sending Signals From The Command Line

Another way of sending signals to processes is done using various commands, usually internal to the shell:

`kill`

The kill command accepts two parameters: a signal name (or number), and a process ID. Usually the syntax for using it goes something like:

```
kill -<signal> <PID>
```

For example, in order to send the `INT` signal to process with PID 5342, type:

```
kill -INT 5342
```

This has the same affect as pressing Ctrl-C in the shell that runs that process.
If no signal name or number is specified, the default is to send a `TERM` signal to the process, which normally causes its termination, and hence the name of the `kill` command.

`fg`

On most shells, using the `'fg'` command will resume execution of the process (that was suspended with Ctrl-Z), by sending it a `CONT` signal.

Common UNIX Signal Names and Numbers

All available UNIX signals have different names, and are mapped to certain numbers as described below.

Number	Name	Description	Used for
0	SIGNULL	Null	Check access to pid
1	SIGHUP	Hangup	Terminate; can be trapped
2	SIGINT	Interrupt	Terminate; can be trapped
3	SIGQUIT	Quit	Terminate with core dump; can be
9	SIGKILL	Kill	Forced termination; cannot be trapped
15	SIGTERM	Terminate	Terminate; can be trapped
24	SIGSTOP	Stop	Pause the process; cannot be trapped
25	SIGTSTP	Terminal	stop Pause the process; can be
26	SIGCONT	Continue	Run a stopped process

Sending Signals Using System Calls

A third way of sending signals to processes is by using the kill system call. This is the normal way of sending a signal from one process to another. This system call is also used by the `'kill'` command or by the `'fg'` command. Here is an example code that causes a process to suspend its own execution by sending itself the `STOP` signal:

```
#include <unistd.h>     /* standard unix functions, like getpid()       */
#include <sys/types.h>  /* various type definitions, like pid_t         */
#include <signal.h>     /* signal name macros, and the kill() prototype */
pid_t my_pid = getpid();
kill(my_pid, SIGSTOP);
```

Kill command

The Kill command in unix operating system is used to send a signal to the specified process or group. If we do not specify any signal, then the kill command passes the SIGTERM signal. We mostly use the kill command for

terminating or killing a process. However we can also use the kill command for running a stopped process.

The syntax of kill command is

```
kill [-s signal] pid
```

The options to the kill command are:

- **pid** : list of process that kill command should send a signal
- **-s signal** : send the specified signal to the process
- **-l** : list all the available signals.

Kill Command Examples:

(1) Listing all the signal names.

Run the kill command with -l option to list all the available signal names.

```
$ kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM
USR1 USR2 CLD PWR WINCH URG POLL STOP TSTP CONT TTIN TTOU VTALRM PROF
XCPU XFSZ WAITING LWP FREEZE THAW CANCEL LOST RTMIN RTMIN+1 RTMIN+2
RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1 RTMAX
```

```
$ps –ef
```

(2) Killing a process.

To kill processes simply pass the process id to the kill command. This is shown below:

```
$kill 4529
```

(3) Forcefully killing a process.

Use the -9 option with the kill command to kill a process force fully. The following kill command terminates the process forcefully:

```
$kill -9 1567
$kill -SIGKILL 1567
$kill -KILL 1567
$kill -s SIGKILL 1567
$kill -s KILL 1567
```

(4)kill command to kill multiple process

```
$ps –ef

$kill -9 3420 6352
```

(5) kill command in unix to find Signal name

$kill –l 3

QUIT

Running jobs in background

To run a command in the background , you end it with an ampersand (&)
The advantage of running a command in the background is that you can go on to run other commands without waiting for the background job to finish.

$find / –name “myfile*” -print 1>outputfile 2>errfile &

Job Controls in unix

(i) Suspend the foreground job

(ii) Move a suspended job to the background.

(iii) Bring back a suspended job to the foreground

Sending the current foreground job to the background using CTRL-Z and bg command

You can send an already running foreground job to background as explained below:

- Press 'CTRL+Z' which will suspend the current foreground job.
- Execute bg to make that command to execute in background.

For example, if you've forgot to execute a job in a background, you don't need to kill the current job and start a new background job. Instead, suspend the current job and put it in the background as shown below.

```
$find / –name "myfile*"  -print 1>outputfile 2>errfile
$[CTRL-Z]

$ bg
```

View all the background jobs using jobs command

You can list out the background jobs with the command **jobs**. Sample output of jobs command is

```
$jobs
[1]   Running                 bash download-file.sh &
[2]-  Running                 evolution &
[3]+  Done                    nautilus .
```

Taking a job from the background to the foreground using fg command

You can bring a background job to the foreground using **fg command**. When executed without arguments, it will take the most recent background job to the foreground.

```
$ fg
```

If you have multiple background ground jobs, and would want to bring a certain job to the foreground, execute jobs command which will show the job id and command.

In the following example, fg %1 will bring the job #1 (i.e download-file.sh) to the foreground.

```
$ jobs
[1]   Running                 bash download-file.sh &
[2]-  Running                 evolution &
[3]+  Done                    nautilus .

$ fg %1
```

5. Kill a specific background job using kill %

If you want to kill a specific background job use, kill %job-number. For example, to kill the job 2 use

```
$ kill %2
```

---

nohup command

Most of the time you login into remote server via ssh. If you start a shell script or command and you exit (abort remote connection), the process / command will get killed. Sometime job or command takes a long time. If you are not sure when the job will finish, then it is better to leave job running in background. However, if you logout the system, the job will be stopped. What do you do?

Answer is simple, use nohup utility which allows to run command./process or shell script that can continue running in the background after you log out from a shell:

nohup Syntax:

```
nohup command-name &
```

Where,

- **command-name** : is name of shell script or command name. You can pass argument to command or a shell script.

- **&** : nohup does not automatically put the command it runs in the background; you must do that explicitly, by ending the command line with an & symbol.

Examples

1. To run a command in the background after you log off, enter:

   $nohup find / –name "myfile*" -print &

   After you enter this command, the following is displayed:

   ```
   670
   $ Sending output to nohup.out
   ```

   The process ID number changes to that of the background process started by & (ampersand). The message `Sending output to nohup.out` informs you that the output from the **find** command is in the **nohup.out** file. You can log off after you see these messages, even if the **find** command is still running.

Umask Command

The default permissions assigned to all of your files and directories at the time you create them. You can list or change the default permission by using the **umask** command.

Default permissions are assigned by the system whenever you create a new file or directory, and these are governed by the umask setting.

```
$umask 002
```

umask 002 - Assigns permissions so that only you and members of your group have read/write access to files, and read/write/search access to directories you own. All others have read access only to your files, and read/search to your directories.