Program- BTech-3rd Semester
Course Code- CSET213
Year- 2025
Date- 18/11/2025

Type- Sp. Core-I
Course Name-Linux and Shell Programming
Semester- Odd
Batch- All (Cyber Security)

Lab Assignment 18

Exp No	Name	CO1	CO 2	CO 3	CO 4
18	Process management, creation, termination and other useful commands, Parent, zombie, orphan process, Process system calls. Fork, exec, Wait and signal, Inter Process communication via pipes and shared memory, various commands.			✓	-

Objective: To understand process creation, management, termination, and other useful process system calls. Moreover, different types of processes like parent, child, orphan, and zombie can be understand by doing hands-on using fork, exec, wait and signal system calls. Further, communication between the processes can be established using pipes and shared memory.

Outcomes: The student can able to do system and network level programming.

Hands-on Learning on Linux process (60 minutes)

1. Process Management

a. **Process:** An instance of a program is called a process. In simple terms, any command that you give to your Linux machine starts a new process.

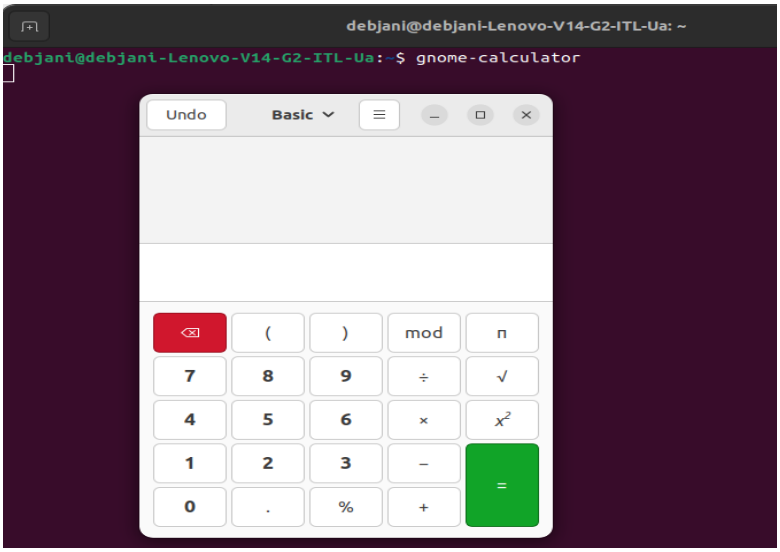
b. **Types of Processes:**

i. Foreground Processes: They run on the screen and need input from the user. For example, Office programs.

ii. Background Processes: They run in the background and usually do not need user input. For example, Antivirus.

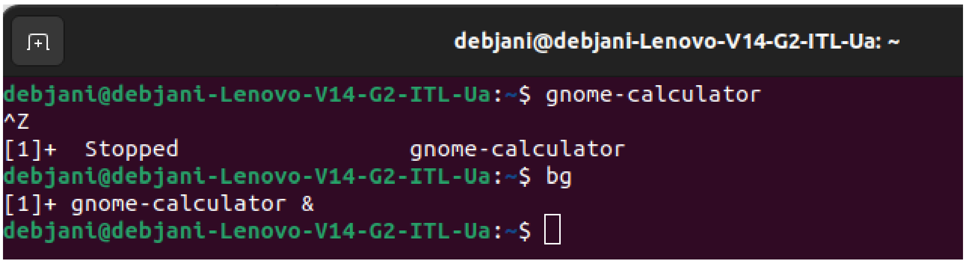
c. **Hands on example**

i. Open terminal and type gnome-calculator



ii. Stop the foreground process by pressing CTRL+Z

iii. Keep the calculator process in background by executing “bg” command on terminal.

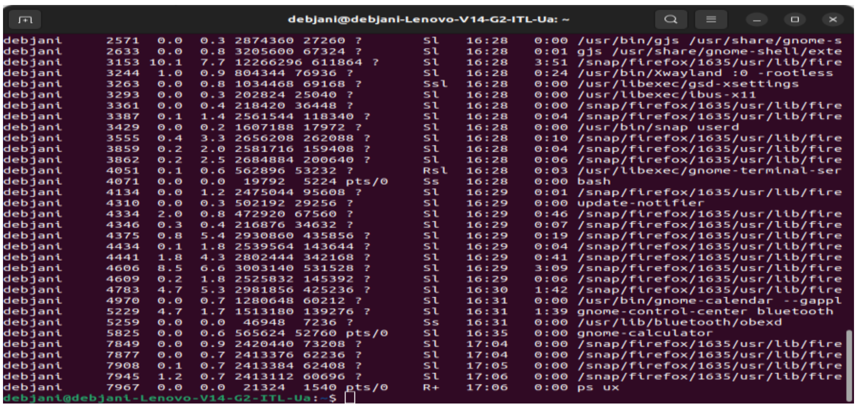


```
debjani@debjani-Lenovo-V14-G2-ITL-Ua: ~
debjani@debjani-Lenovo-V14-G2-ITL-Ua:~$ gnome-calculator
^Z
[1]+  Stopped                 gnome-calculator
debjani@debjani-Lenovo-V14-G2-ITL-Ua:~$ bg
[1]+ gnome-calculator &
debjani@debjani-Lenovo-V14-G2-ITL-Ua:~$
```

iv. Execute “top” command on the terminal

Field	Description
PID	The process Id of each task
USER	The username of task owner
PR	Priority can be 20(highest) or –20(lowest)
NI	The nice value of a task
VIRT	Virtual memory used(Kb)
RES	Physical memory used (Kb)
SHR	Shared memory used (Kb)
S	Status There are five types: ‘D’ = Uninterruptible sleep ‘R’ = Running ‘S’ = Sleeping ‘T’ = Traced or Stopped ‘Z’ = Zombie
%CPU	% of CPU time
%MEM	Physical memory used
TIME+	Total CPU time

v. Execute “ps ux” on the terminal (ux stands for user)



```
debjani@debjani-Lenovo-V14-G2-ITL-Ua: ~
debjani     2571  0.0  0.3 2874360 27260 ?     Sl   16:28   0:00 /usr/bin/gjs /usr/share/gnome-s
debjani     2633  0.0  0.8 3205600 67324 ?     Sl   16:28   0:01 gjs /usr/share/gnome-shell/exte
debjani     3153 10.1  7.7 12266296 611864 ?   Sl   16:28   3:51 /snap/firefox/1635/usr/lib/fire
debjani     3244  1.0  0.9 804344 76936 ?      Sl   16:28   0:24 /usr/bin/Xwayland :0 -rootless
debjani     3263  0.0  0.8 1034468 69168 ?     Ssl  16:28   0:00 /usr/libexec/gsd-xsettings
debjani     3293  0.0  0.3 202824 25040 ?      Sl   16:28   0:00 /usr/libexec/ibus-x11
debjani     3361  0.0  0.4 218420 36448 ?      Sl   16:28   0:00 /snap/firefox/1635/usr/lib/fire
debjani     3387  0.1  1.4 2561544 118340 ?    Sl   16:28   0:04 /snap/firefox/1635/usr/lib/fire
debjani     3429  0.0  0.2 1607188 17972 ?     Sl   16:28   0:00 /usr/bin/snap userd
debjani     3555  0.4  3.3 2656208 262088 ?    Sl   16:28   0:10 /snap/firefox/1635/usr/lib/fire
debjani     3859  0.2  2.0 2581716 159408 ?    Sl   16:28   0:04 /snap/firefox/1635/usr/lib/fire
debjani     3862  0.2  2.5 2684884 200640 ?    Sl   16:28   0:06 /snap/firefox/1635/usr/lib/fire
debjani     4051  0.1  0.6 562896 53232 ?      Rsl  16:28   0:03 /usr/libexec/gnome-terminal-ser
debjani     4071  0.0  0.0  19792  5224 pts/0  Ss   16:28   0:00 bash
debjani     4134  0.0  1.2 2475044 95608 ?     Sl   16:29   0:01 /snap/firefox/1635/usr/lib/fire
debjani     4310  0.0  0.3 502192 29256 ?      Sl   16:29   0:00 update-notifier
debjani     4334  2.0  0.8 472920 67560 ?      Sl   16:29   0:46 /snap/firefox/1635/usr/lib/fire
debjani     4346  0.3  0.4 216876 34632 ?      Sl   16:29   0:07 /snap/firefox/1635/usr/lib/fire
debjani     4375  0.8  5.4 2930860 435856 ?    Sl   16:29   0:19 /snap/firefox/1635/usr/lib/fire
debjani     4434  0.1  1.8 2539564 143644 ?    Sl   16:29   0:04 /snap/firefox/1635/usr/lib/fire
debjani     4441  1.8  4.3 2802444 342168 ?    Sl   16:29   0:41 /snap/firefox/1635/usr/lib/fire
debjani     4606  8.5  6.6 3003140 531528 ?    Sl   16:29   3:09 /snap/firefox/1635/usr/lib/fire
debjani     4609  0.2  1.8 2525832 145392 ?    Sl   16:29   0:06 /snap/firefox/1635/usr/lib/fire
debjani     4783  4.7  5.3 2981856 425236 ?    Sl   16:30   1:42 /snap/firefox/1635/usr/lib/fire
debjani     4970  0.0  0.7 1280648 60212 ?     Sl   16:31   0:00 /usr/bin/gnome-calendar --gappl
debjani     5229  4.7  1.7 1513180 139276 ?    Sl   16:31   1:39 gnome-control-center bluetooth
debjani     5259  0.0  0.0  46948  7236 ?      Ss   16:31   0:00 /usr/lib/bluetooth/obexd
debjani     5825  0.0  0.6 565624 52760 pts/0  Sl   16:35   0:00 gnome-calculator
debjani     7849  0.0  0.9 2420440 73208 ?     Sl   17:04   0:00 /snap/firefox/1635/usr/lib/fire
debjani     7877  0.0  0.7 2413376 62236 ?     Sl   17:04   0:00 /snap/firefox/1635/usr/lib/fire
debjani     7908  0.1  0.7 2413384 62408 ?     Sl   17:05   0:00 /snap/firefox/1635/usr/lib/fire
debjani     7945  1.2  0.7 2413112 60696 ?     Sl   17:06   0:00 /snap/firefox/1635/usr/lib/fire
debjani     7967  0.0  0.0  21324  1540 pts/0  R+   17:06   0:00 ps ux
debjani@debjani-Lenovo-V14-G2-ITL-Ua:~$
```

vi. Execute “ps <pid>” to know more about that process

vii. Execute “kill <pid>” if you want to kill the process

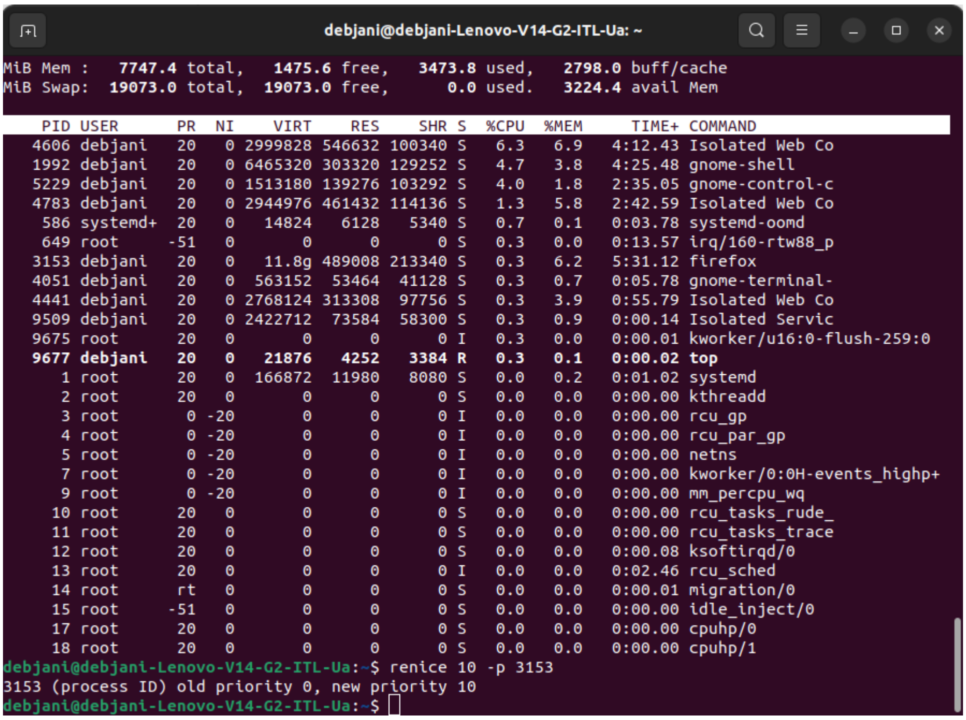
viii. To know the pid of any process, you can execute the command “pidof <process_name>”

ix. To set the priority of the new process, use nice command like

nice –n 15 firefox

x. To set the priority of the existing process, use renice command like

renice 10 –p <pid>



```
debjani@debjani-Lenovo-V14-G2-ITL-Ua: ~
MiB Mem :   7747.4 total,   1475.6 free,   3473.8 used,   2798.0 buff/cache
MiB Swap:  19073.0 total,  19073.0 free,      0.0 used.   3224.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   4606 debjani   20   0 2999828 546632 100340 S   6.3   6.9   4:12.43 Isolated Web Co
   1992 debjani   20   0 6465320 303320 129252 S   4.7   3.8   4:25.48 gnome-shell
   5229 debjani   20   0 1513180 139276 103292 S   4.0   1.8   2:35.05 gnome-control-c
   4783 debjani   20   0 2944976 461432 114136 S   1.3   5.8   2:42.59 Isolated Web Co
    586 systemd+  20   0   14824   6128   5340 S   0.7   0.1   0:03.78 systemd-oomd
    649 root     -51   0       0      0      0 S   0.3   0.0   0:13.57 irq/160-rtw88_p
   3153 debjani   20   0   11.8g 489008 213340 S   0.3   6.2   5:31.12 firefox
   4051 debjani   20   0  563152  53464  41128 S   0.3   0.7   0:05.78 gnome-terminal-
   4441 debjani   20   0 2768124 313308  97756 S   0.3   3.9   0:55.79 Isolated Web Co
   9509 debjani   20   0 2422712  73584  58300 S   0.3   0.9   0:00.14 Isolated Servic
   9675 root      20   0       0      0      0 I   0.3   0.0   0:00.01 kworker/u16:0-flush-259:0
   9677 debjani   20   0   21876   4252   3384 R   0.3   0.1   0:00.02 top
      1 root      20   0  166872  11980   8080 S   0.0   0.2   0:01.02 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
      7 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_highp+
      9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     10 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
     11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
     12 root      20   0       0      0      0 S   0.0   0.0   0:00.08 ksoftirqd/0
     13 root      20   0       0      0      0 I   0.0   0.0   0:02.46 rcu_sched
     14 root      rt   0       0      0      0 S   0.0   0.0   0:00.01 migration/0
     15 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/0
     17 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     18 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
debjani@debjani-Lenovo-V14-G2-ITL-Ua:~$ renice 10 -p 3153
3153 (process ID) old priority 0, new priority 10
debjani@debjani-Lenovo-V14-G2-ITL-Ua:~$
```

xi. Few more useful commands: df –h, free –m, free -g

2. Process Creation

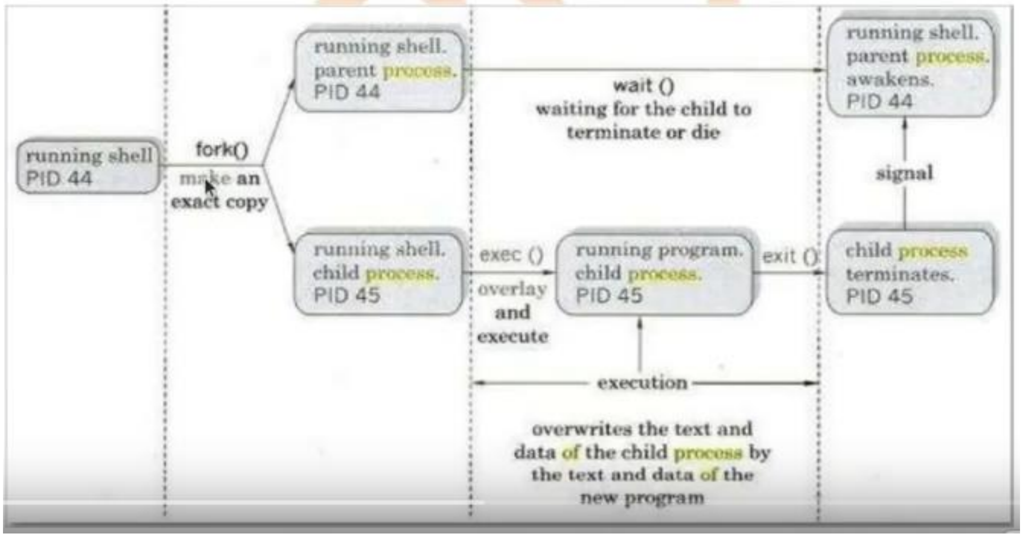
A program starts its execution ---> a process is born

As soon as the program terminates ----> a process dies

There are three distinct phases in the creation of a process

- Forking
- Overlaying and Execution
- Waiting

Mechanism of process creation is depicted as shown in below figure



- ○ Forking is the first phase in the creation of a process by a process
- ○ The calling process makes a call to the system routine fork() which then makes an exact copy of itself.
- ○ After the fork() there will be two processes.
- ○ The fork of the parent process returns the PID of the new process, that is the child process just created.
- ○ The fork of the child returns a 0 (zero).
- ○ Immediately after forking, the parent makes a system call to one of the wait() functions.
- ○ The parent keeps waiting for the child process to complete it task.
- ○ The second phase, the exec() function overwrites the text and data area of the child process.
- ○ The exit() function terminates the child process.
- ○ The parent awakens only when it receives a complete signal from the child, after which it will be free to continue with its other functions.

- **Occurrence of the event to a process**

- ○ SIGNAL ----> Mechanism used by the kernel to communicate the occurrence of the event to a process.
- ○ Signal is identified by an integer and its symbolic name
- ○ The action that a signal will take is called disposition
- ○ When the process receives the signal it can do three things:
  - ○ Ignore the signal
  - ○ Stop the process
  - ○ Terminate the process
- ○ The process is terminated using exit() system call
- ○ Exit system call returns the status to the parent process
- ○ The process can be terminated due to the following reasons:
  - ○ Not used an explicit exit or return call

Example of fork()

/* --- */
/* PROGRAM fork-01.c */
/* This program runs two processes, a parent and a child. Both of */
/* them run the same loop printing some messages. Note that printf()*/
/* is used in this program. */
/* --- */

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

#define   MAX_COUNT  200

void  ChildProcess(void);                /* child process
prototype  */
void  ParentProcess(void);               /* parent process
prototype */

void  main(void)
{
     pid_t  pid;

     pid = fork();
     if (pid == 0)
          ChildProcess();
     else
          ParentProcess();
}

void  ChildProcess(void)
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
          printf("   This line is from child, value = %d\n", i);
     printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
     int   i;

     for (i = 1; i <= MAX_COUNT; i++)
          printf("This line is from parent, value = %d\n", i);
     printf("*** Parent is done ***\n");
}
```

For more information go to the man page by executing man fork

- exit terminates the process that calls it
    - process can supply an exit status code when it exits
    - kernel records the exit status code in case another process asks for it (via waitpid)
    - waitpid lets a process wait for another to terminate, and retrieve its exit status code
- Wait System Call

In the previous example, you have observed that the execution of parent and child process are random. If you want to execute the child first and then the parent, you need to use wait system call. Wait() system calls are only used when parent needs to wait for the child to execute first.

For example: You want child process to be executed first. So, you need to add wait() system call in the parent part.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else{
wait(NULL);
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
return 0;
}
```

- **With wait status**

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else{
w1=wait(&wstatus);
printf("Status is %d\n", WIFEXITED(wstatus)); //status =1 indicates true that the child process is terminated successfully
printf("PID of child %d\n",w1);
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
return 0;
}
```

- If you have multiple child processes and want the parent to wait for any specific child then you can use waitpid() system call.

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

#include<sys/wait.h>
int main()
{
pid_t p,q;
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am first child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else{

q=fork();

if(q==0)

{

printf("I am second child having id %d\n",getpid());
printf("Second child's parent's id is %d\n",getppid());
}

else

{

waitpid(p,NULL,0);

printf("I am parent having id %d\n",getpid());

printf("My first child's PID is %d\n",p);

printf("My second child's PID is %d\n",q);

}
}
```

  - For more information go to the man page by executing man wait

- If you want child process to wait for the parent then add sleep() system call in the child process.

  - For example:

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

int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)
{sleep(3);
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else{

printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
return 0;
```

- **How to create an orphan process?**

  Orphan process: Parent process terminate before child process is called as an orphan process.

  Example:

```
#include<stdio.h
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
p=fork();

 if(p==0)
     {
        sleep(5);//child goes to sleep and in the meantime parent terminates
        printf("I am child having PID %d\n",getpid());
        printf("My parent PID is %d\n",getppid());
     }
     else
     {
         printf("I am parent having PID %d\n",getpid());
         printf("My child PID is %d\n",p);
     }
 }
```

  - Execute the program and observe
  - Again run the program by adding sleep in both the process

Example:

```
#include<stdio.h
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
p=fork();

 if(p==0)
     {
         sleep(10); //child goes to sleep and in the mean time parent
terminates
         printf("I am child having PID %d\n",getpid());
         printf("My parent PID is %d\n",getppid());
     }
     else
     {
            sleep(2);

          printf("I am parent having PID %d\n",getpid());
          printf("My child PID is %d\n",p);
     }
 }
```

- Run the program in background, for example ./a.out &
- Execute the command ps twice with a gap of 1s and observe

- **How to create Zombie process?**

Zombie process: Parent process is responsible to free the resource entries held by the child process. When the child process finishes and terminates without informing the parent process, the parent never knows the status of child process and will remain assume that the child process exists in the system. So, the child process becomes the zombie process.

Example:

```
#include<stdio.h
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
p=fork();

 if(p==0)
     {

          printf("I am child having PID %d\n",getpid());
          printf("My parent PID is %d\n",getppid());
     }
     else
     {
           sleep(3);

          printf("I am parent having PID %d\n",getpid());
          printf("My child PID is %d\n",p);
     }
 }
```

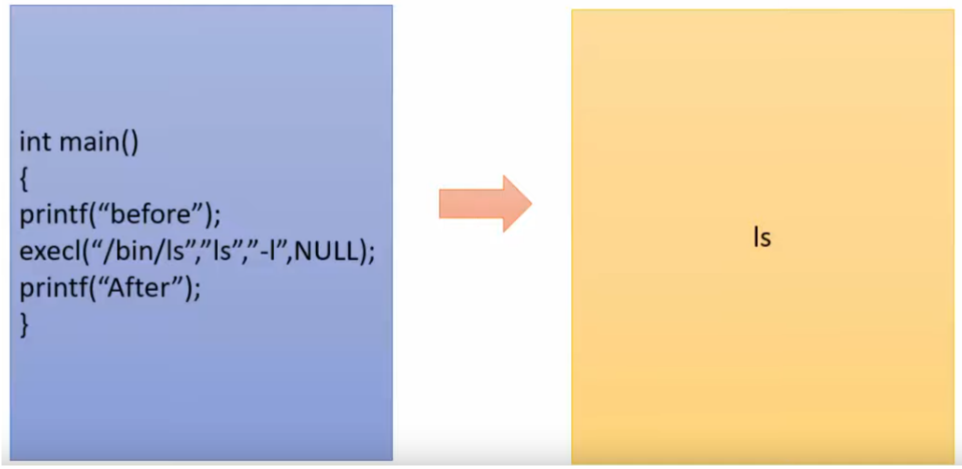
- Execute the program in backgound like ./a.out &
- Run the ps command just after the execution of the program, you will see the child process appears with the keyword “defunct” which shows that the child process is zombie process.
- How to overcome the situation to make child process to be zombie process?
- Rather than sleep() in parent process, use wait()

Example:

```
#include<stdio.h
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
p=fork();

 if(p==0)
     {

          printf("I am child having PID %d\n",getpid());
          printf("My parent PID is %d\n",getppid());
     }
     else
     {
wait(NULL);

 sleep(3);

           printf("I am parent having PID %d\n",getpid());
           printf("My child PID is %d\n",p);
     }
 }
```

- Execute the program in background and observe.

- **Exec system call**
  - exec() ----> Execute a (program) file
  - The exec() system call is also used to create processes. But there is one big difference between fork() and exec() calls. The fork() call creates a new process while preserving the parent process. But an exec() call replaces the address space, text segment, data segment, etc. Of the current process with the new process.
  - The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.
  - Exec() is used when the user wants to launch a new file or program in the same process.



  - The exec Family of Functions

There is a family of exec() functions, all of which have slightly different characteristics:

```
int execl ( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execle( const char *path, const char *arg, ..., char *const envp[] );
int execv ( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *file, char *const argv[], char *const envp[] );
```

Example:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    printf("Before execl\n");
    execl("/bin/ls","ls","-l",NULL);
    printf("After execlp\n");
}
```

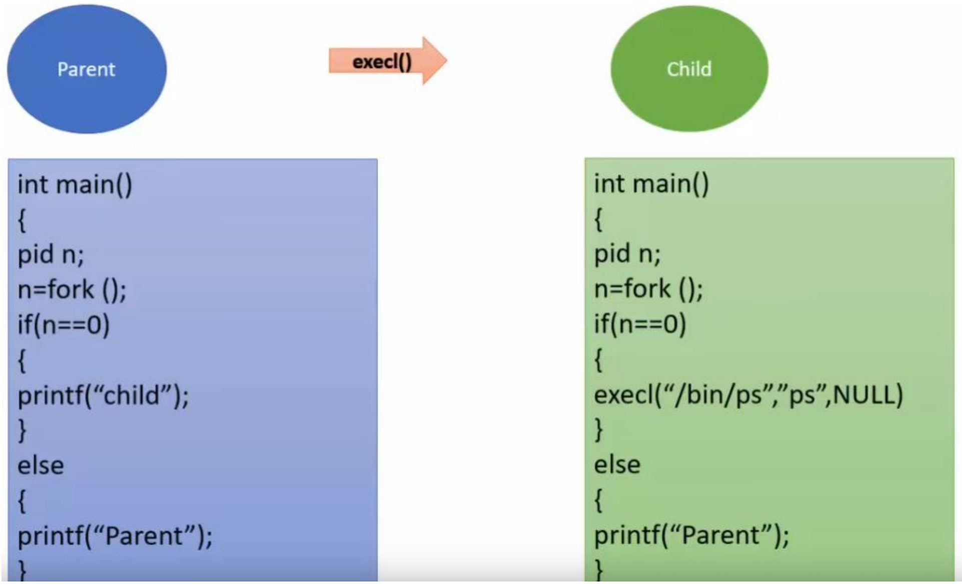
  - **Execute and observe**

Example:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    printf("Before execl\n");
    execl("/bin/ps","ps","-a",NULL);//
    printf("After execlp\n");
}
```

Execute and observe, are you able to see a.out?

Example:



Example:

```
#include<stdio.h
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
p=fork();

 if(p==0)
     {
        printf("I am child having PID %d\n",getpid());

       execl("/bin/ps","ps","-a",NULL);

        printf("My parent PID is %d\n",getppid());

      }
     else
     {
        wait(NULL);

        sleep(3);

        printf("I am parent having PID %d\n",getpid());
        printf("My child PID is %d\n",p);
     }
 }
```

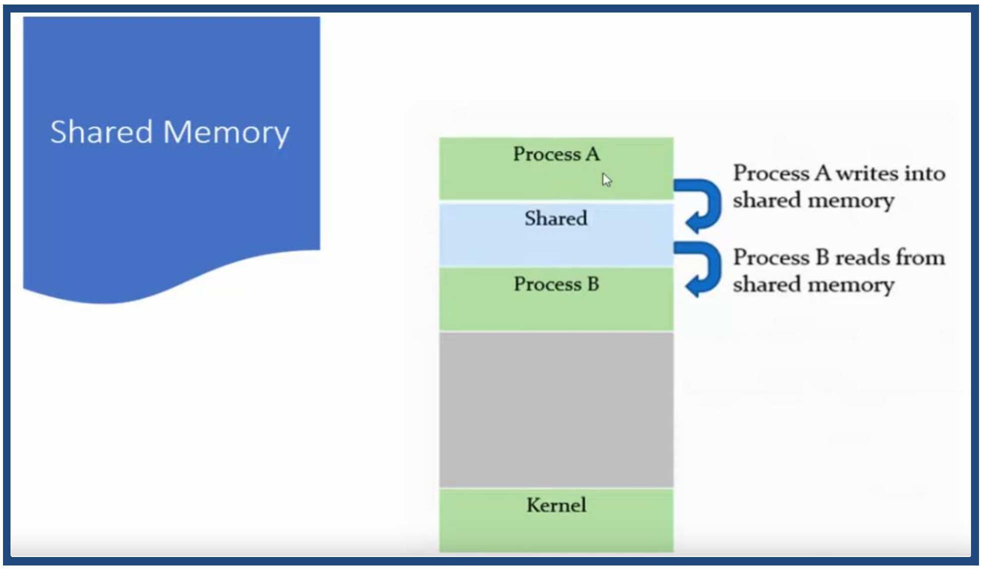
- **Interprocess communication using pipe function**

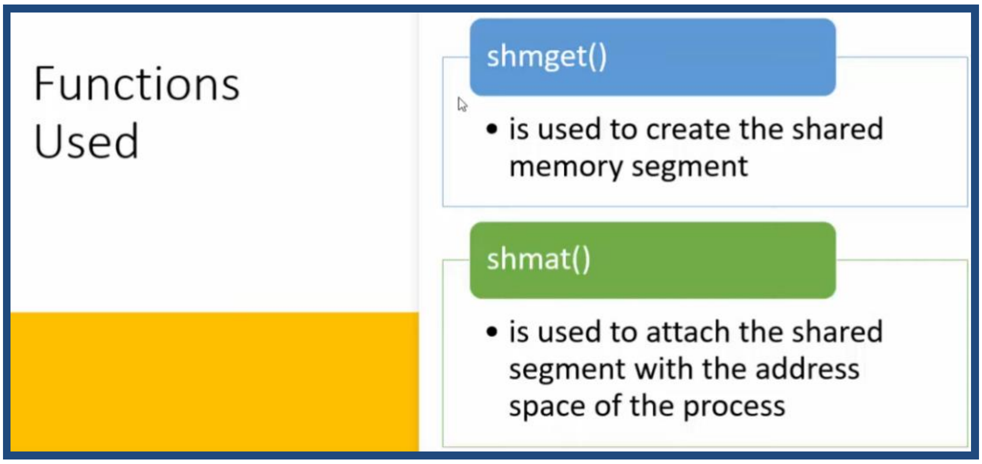
Syntax	#include<unistd.h> int pipe(int fd[2]); Creates a unidirectional pipe Writing end – fd[1] Reading end – fd[0]

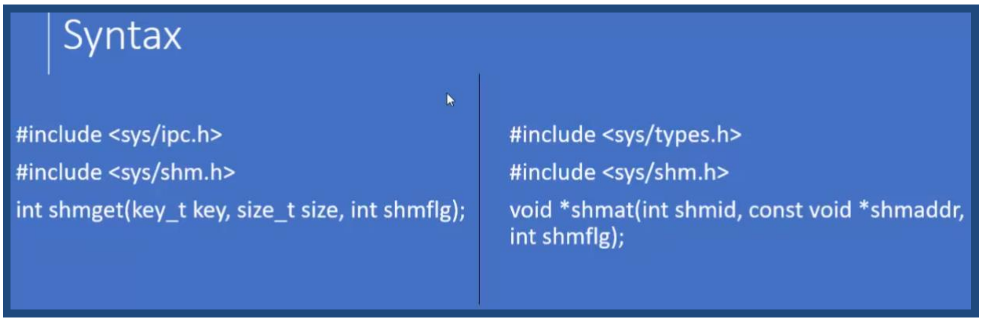
  - pipe() function creates a unidirectional pipe for IPC. On success it return two file descriptors pipefd[0] and pipefd[1]. pipefd[0] is the reading end of the pipe. So, the process which will receive the data should use this file descriptor. pipefd[1] is the writing end of the pipe. So, the process that wants to send the data should use this file descriptor.
  - The program below creates a child process. The parent process will establish a pipe and will send the data to the child using writing end of the pipe and the child will receive that data and print on the screen using the reading end of the pipe.
  - Example to send a message from parent process to child process using pipe()

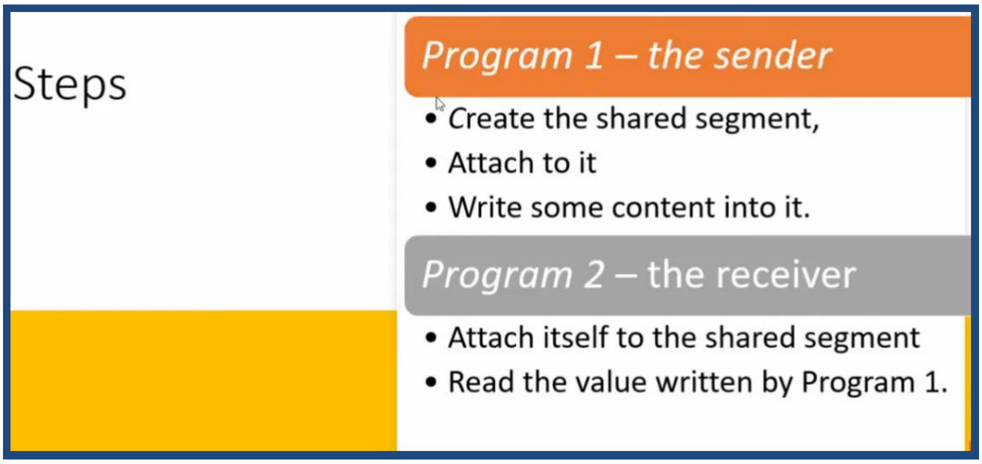
```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
int fd[2],n;
char buffer[100];
pid_t p;
pipe(fd); //creates a unidirectional pipe with two end fd[0] and fd[1]
p=fork();
if(p>0) //parent
{
printf("Parent Passing value to child\n");
write(fd[1],"hello\n",6); //fd[1] is the write end of the pipe
wait();
}
else // child
{
printf("Child printing received value\n");
n=read(fd[0],buffer,100); //fd[0] is the read end of the pipe
write(1,buffer,n);
}
}
```

- **Interprocess communication using shared memory**









- Program-1: This program creates a shared memory segment, attaches itself to it and then writes some content into the shared memory segment.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT); //creates shared memory segment with key 2345,
//having size 1024 bytes. IPC_CREAT is used to create the shared segment if it does not exist.
//0666 are the permisions on the shared segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory); //this prints the address where the segment is
//attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

- How it works?
  - shmget() function creates a segment with key 2345, size 1024 bytes and read and write permissions for all users.
  - It returns the identifier of the segment which gets store in shmid.
  - This identifier is used in shmat() to attach the shared segment to the address space of the process.
  - NULL in shmat() means that the OS will itself attach the shared segment at a suitable address of this process.
  - Then some data is read from the user using read() system call and it is finally written to the shared segment using strcpy() function.

- Program-2: This program attaches itself to the shared memory segment created in Program 1. Finally, it reads the content of the shared memory

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to
shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char
*)shared_memory);
}
```

- How it works?
  - shmget() here generates the identifier of the same segment as created in Program-1. Remember to give the same key value. The only change is, do not write IPC_CREAT as the shared memory segment is already created.
  - Next, shmat() attaches the shared segment to the current process. After that, the data is printed from the shared segment. In the output, you will see that it is the same data that you have written while executing the Program 1.

Problems for Assessment (60 Minutes)

1. Develop an add operation using pipe and shared memory IPC mechanisms where one process writes the operand values and the other process provide the result after adding them. (15 minutes)
2. Develop a concurrent server for handling multiple clients using fork() (45 minutes)

Submission Instructions:

1. Submission requires the screen shots of all the incurred steps to execute the program.
2. All these files are in single document preferably in pdf format.
3. Use the naming convention: Prog_CourseCode_RollNo_LabNo.docx
4. Submission is through LMS only