NAME: VARSHA D KULKARNI **SRN: PES1UG19EC339**

OS -LAB- WEEK-2

1. Write a program to create a child process which calls one of the exec functions to create a file named "abc.txt"[Hint: shell command “touch filename” creates a new file]. Parent using one of the exec calls must execute the command "ls". Make the parent wait for the child to terminate and then execute "ls".

Program:

```
#include<stdio.h>

#include<sys/wait.h>

#include<unistd.h>

#include<stdlib.h>
int main()
{
int pid;
pid=fork();
if(pid<0)
{
printf("Error in creating a process\n");
exit(1);
 }
else if(pid==0)
{
printf("child process...\n");
execlp("/bin/touch","touch","abc.txt",NULL);

}
else{
wait(NULL);
printf("child processes terminated\n");
printf("parent process...\n");
execlp("/bin/ls","ls",NULL);
}
return 0;
}
```

OUTPUT:

```
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cc week2_1.c
abc@ubuntu:~/PES1UG19EC339_VARSHA$ ./a.out
child process...
child processes terminated
parent process...
abc.txt  a.txt            rec.c         rev_head.h    rev_server.o  seq.h
a.c      b.c              rev_client.c  rev.mk        seq.c         seq_server.c
a.out    binary_client.c  rev_client.o  rev_server.c  seq_client.c  varsha.txt
```

2. Write a program where in the child process initiates a new program which finds the sum of n numbers. The numbers to add are given as arguments in the exec function. Use appropriate exec function. Parent process should wait for the termination of child process.

Program:

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>

int main()
{
int pid;
pid=fork();
if(pid<0)
{
printf("Error in creating a process\n");
exit(1);
}
else if(pid==0)
{
printf("Child process...\n");
execlp("/home/abc/PES1UG19EC339_VARSHA/add","add","40","20",    NULL);
}
else
{
wait(NULL);
printf("Parent process...\n");
}
return 0;}
```

OUTPUT:

```
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cat sum.c
#include<stdio.h>
#include<stdlib.h>
void main(int argc,char* argv[])
{
 int sum=0;
 for(int i=0; i<argc; i++)
 sum+=atoi(argv[i]);

 printf("%d\n", sum);
}
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cc sum.c
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cc -o add sum.c
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cc week2_2.c
abc@ubuntu:~/PES1UG19EC339_VARSHA$ ./a.out
Child process...
60
Parent process...
```

3. WAP to create orphan and Zombie processes.

<u>Orphan:</u>

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int pid;
pid=fork();
if(pid<0)
{
printf("Error in creating a process\n");
exit(1);
}
else if(pid==0)
{
sleep(30);
printf("child process...\n");
printf("Became orphan\n");

}
else{
```

```
printf("parent process...\n");
printf("Finished execution while the child process is running\n");
}
return 0;
}
```

```
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cc  week2_3.c
abc@ubuntu:~/PES1UG19EC339_VARSHA$ ./a.out
parent process...
Finished execution while the child process is running
abc@ubuntu:~/PES1UG19EC339_VARSHA$
```

Here parent process finished execution when child process is still running now child process is called as orphan.

```
abc@ubuntu:~/PES1UG19EC339_VARSHA$ ./a.out
parent process...
Finished execution while the child process is running
abc@ubuntu:~/PES1UG19EC339_VARSHA$ child process...
Became orphan
```

<u>Zombie:</u>

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()

{
int pid;
pid=fork();
if(pid<0)
{
printf("Error in creating a process\n");
exit(1);
}
else if(pid==0)
{
printf("child process...\n");
printf("..Zombie..\n");
}
else{
sleep(30);
printf("parent process...\n");
}
return 0;
}
```

OUTPUT:

Zombie process created:

```
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cc  week2_3.c
abc@ubuntu:~/PES1UG19EC339_VARSHA$ ./a.out
child process...
..Zombie..
```

Here child process is running when parent process is not yet initiated.in this case child process becomes zombie.

```
abc@ubuntu:~/PES1UG19EC339_VARSHA$ ./a.out
child process...
..Zombie..
parent process...
```

```
top - 17:37:04 up  3:14,  1 user,  load average: 2.42, 1.37, 0.84
Tasks: 225 total,   2 running, 182 sleeping,   1 stopped,   1 zombie
%Cpu(s): 92.6 us,  7.1 sy,  0.0 ni,  0.2 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2035248 total,   626632 free,   948228 used,   460388 buff/cache
KiB Swap:  1459804 total,  1362716 free,    97088 used.   913988 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 3796 abc       20   0    4512     72      0 R  81.6  0.0  11:52.76 a.out
 3389 abc       20   0  799428  39256  26644 S  54.9  1.9   2:34.64 gnome-terminal-
 1257 abc       20   0 3518520 256680  72800 S  41.8 12.6   6:45.05 gnome-shell
 1110 abc       20   0  733816  80984  35580 S  12.2  4.0   1:50.42 Xorg
 3868 abc       20   0    4512    800    736 S   5.9  0.0   0:02.43 a.out
 3870 abc       20   0   44216   3888   3236 R   1.0  0.2   0:00.30 top
   10 root      20   0       0      0      0 S   0.3  0.0   0:00.49 ksoftirqd/0
   11 root      20   0       0      0      0 I   0.3  0.0   0:03.26 rcu_sched
 1305 abc       20   0  428376   7532   5736 S   0.3  0.4   0:23.90 ibus-daemon
 3816 root      20   0       0      0      0 I   0.3  0.0   0:00.23 kworker/u4:0-ev
 3838 root      20   0       0      0      0 I   0.3  0.0   0:00.28 kworker/u4:2-ev
    1 root      20   0  225412   6484   4700 S   0.0  0.3   0:05.36 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.02 kthreadd
    3 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 rcu_gp
```

4. WAP to demonstrate that data segment is not shared by the parent and child process.

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>
int a=40;
int main()
{
int pid;
pid=fork();
if(pid<0)
{
printf("Error in creating a process\n");
exit(1);
}
else if(pid==0)
{
printf("child process...\n");
```

```
int b=a/4;
a=a+10;
printf("value of a and b in child process \n a=%d \n b=%d \n",a,b);

}
else{
wait(NULL);
printf("child processes terminated\n");
printf("parent process...\n");
a=a+50;
int b=10;
printf("value of a and b in parent process \n a=%d \n b=%d \n",a,b);

}
return 0;
}
```

OUTPUT:

```
abc@ubuntu:~/PES1UG19EC339_VARSHA$ nano  week2_4.c
abc@ubuntu:~/PES1UG19EC339_VARSHA$ cc  week2_4.c
week2_4.c: In function ‘main’:
week2_4.c:28:1: error: ‘b’ undeclared (first use in this function)
 b=a+15;
 ^
week2_4.c:28:1: note: each undeclared identifier is reported only once for each function it appears in
```

Here b is not declared in the parent process.which implies that data is not shared between parent and child.

Answer the following questions and submit your answers as specified in Submission #2:

- What is the role of the init process on UNIX and Linux systems in regard to process termination?

init is the first process which starts during the booting of any UNIX based computer systems .It is the parent of all processes. When exit() is called it can be normal termination ,but if the parent waits for terminated child process ,the child process becomes zombie and continues till the parent process is terminated. When the parent process terminates without invoking wait(),then child processes become orphan,which are adopted by init process.

- What causes a defunct process on the Linux system and how can you avoid it?

Defaunct process are created when a process is either completed its task or has been corrupted or killed, but its child processes are still running or these parent process is monitoring its child process. To kill this kind of process, kill -9 PID doesn't work.

To avoid a defunct process always use exit( ) in child process and wait( ) in parent process, so that the parent waits for the child to finish it's execution and once the child exits parent will start the execution of it's part.

- How can you identify zombie processes on the Linux system?

Zombie processes can be found easily with the ps command. Within the ps output there is a STAT column which will show the processes current status, a **zombie** process will have Z as the status.

- What does child process inherit from its parent?

A child process inherits most of its attributes, such as file descriptors, from **its** parent. In Unix, a child process is typically created as a copy of the parent, using the fork system call. The child process can then overlay itself with a different program (using exec) as required.