NAME: TANYA CHANCHALANI

SRN: PES1UG19EC326

BRANCH: ECE

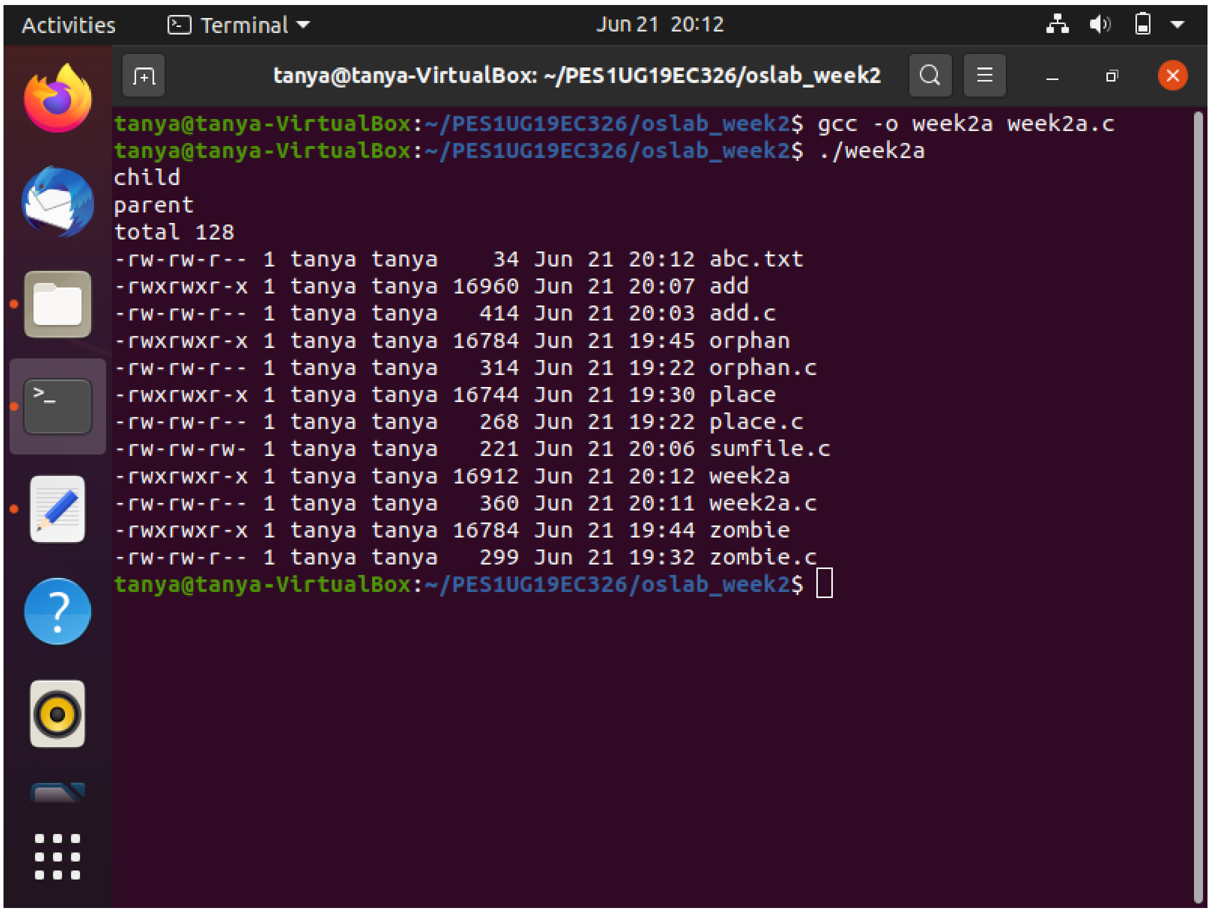
OS_LAB_WEEK: 2

DATE: 21/06/21

1)

a)

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int pid;
pid=fork();
if(pid<0)
{
printf("error\n");
exit(1);
}
else if(pid==0)
{
printf("child\n");
execl("/root/abc","abc","abc",(char *)0);
}
else
{wait(NULL);
printf("parent\n");
execl("/bin/ls","ls","-l",(char *)0);
execlp("ls","ls","-l", (char *)0);}

return 0;

}
```



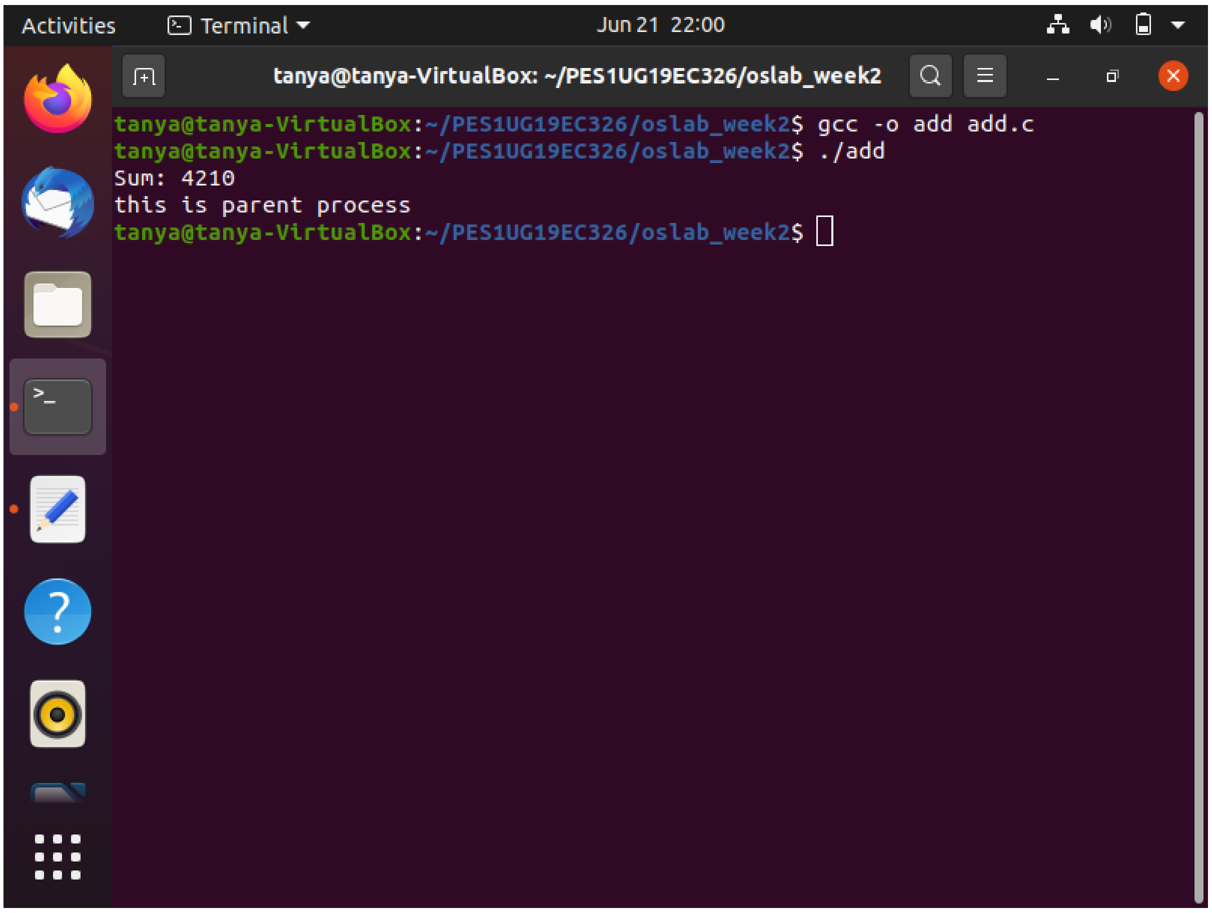
```
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ gcc -o week2a week2a.c
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ ./week2a
child
parent
total 128
-rw-rw-r-- 1 tanya tanya    34 Jun 21 20:12 abc.txt
-rwxrwxr-x 1 tanya tanya 16960 Jun 21 20:07 add
-rw-rw-r-- 1 tanya tanya   414 Jun 21 20:03 add.c
-rwxrwxr-x 1 tanya tanya 16784 Jun 21 19:45 orphan
-rw-rw-r-- 1 tanya tanya   314 Jun 21 19:22 orphan.c
-rwxrwxr-x 1 tanya tanya 16744 Jun 21 19:30 place
-rw-rw-r-- 1 tanya tanya   268 Jun 21 19:22 place.c
-rw-rw-rw- 1 tanya tanya   221 Jun 21 20:06 sumfile.c
-rwxrwxr-x 1 tanya tanya 16912 Jun 21 20:12 week2a
-rw-rw-r-- 1 tanya tanya   360 Jun 21 20:11 week2a.c
-rwxrwxr-x 1 tanya tanya 16784 Jun 21 19:44 zombie
-rw-rw-r-- 1 tanya tanya   299 Jun 21 19:32 zombie.c
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ 
```

b)

```
#include<stdio.h>

#include<unistd.h> //execv

#include<stdlib.h>

#include<sys/wait.h>

//pid_t data type stands for process identification and it is used to represent process ids.

//When execv() is successful, it doesn't return; otherwise, it returns -1.

//int execv(const char *pathname, char *const argv[]);

//char* arg_list[] = { "myprog", "ARG1", "ARG2", NULL };

//execv( "myprog", arg_list );
```

```
int main()

{

pid_t PID = fork();

if(PID<0)

{

printf("fork error\n");

exit(1);

}

else if(PID ==0)

{

static char *arg_list[] = {"./sumfile","435","313","235","3227", NULL};

execv(arg_list[0], arg_list);

printf("addition is done in child process\n");

exit(1);

}

if(PID>0)

{

while(wait(NULL)>0)

printf("this is parent process\n");

}

return 0;

}


//sumfile.c

//In C, the atoi() function converts a string to an integer.

#include<stdio.h>

#include<stdlib.h>

//In C, the atoi() function converts a string to an integer.

int main(int argc, char **argv)

{
```

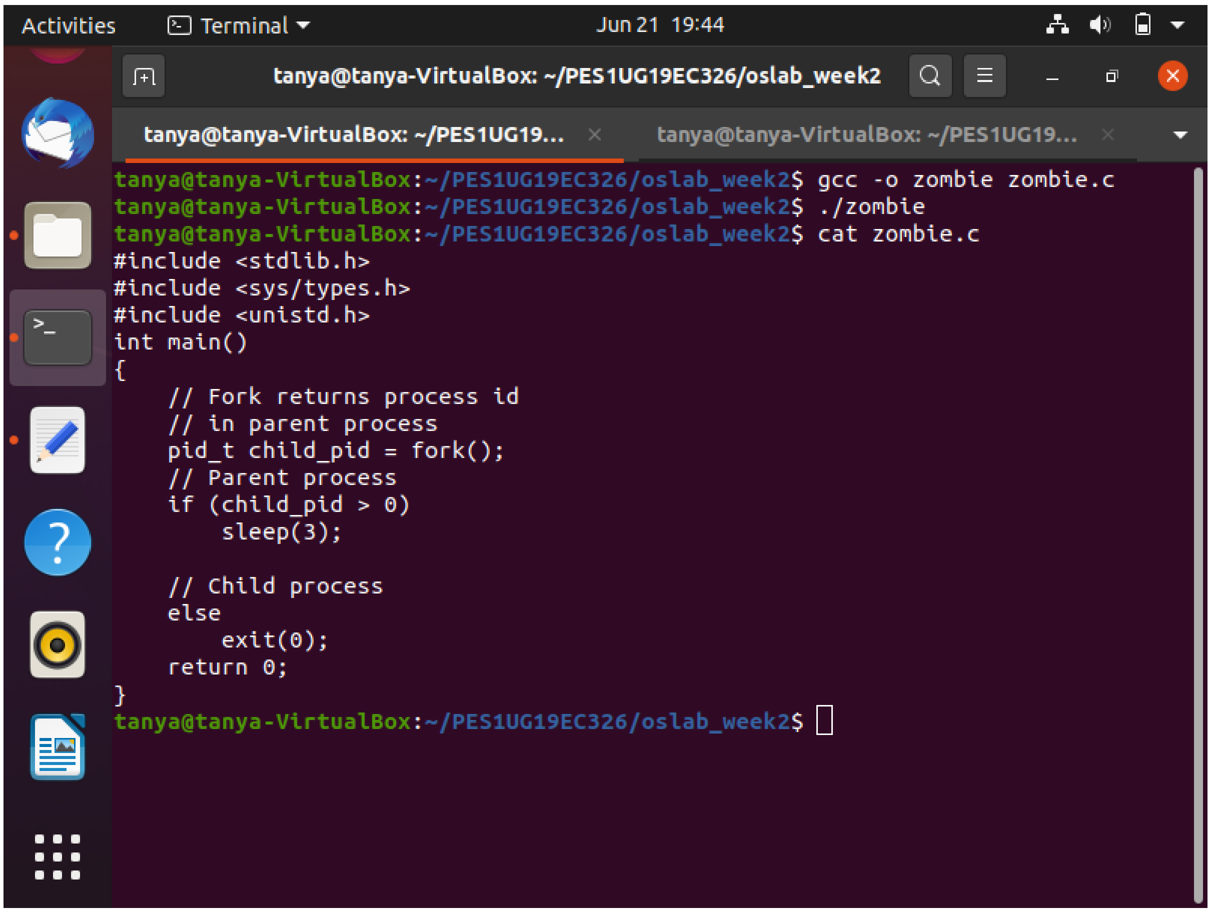
```
int num;

int sum=0;

int temp=1;

while(argv[temp]!=NULL)

{

num = atoi(argv[temp]);

sum = sum+num;

temp++;

}

printf("Sum: %d\n", sum);

return 0;

}
```



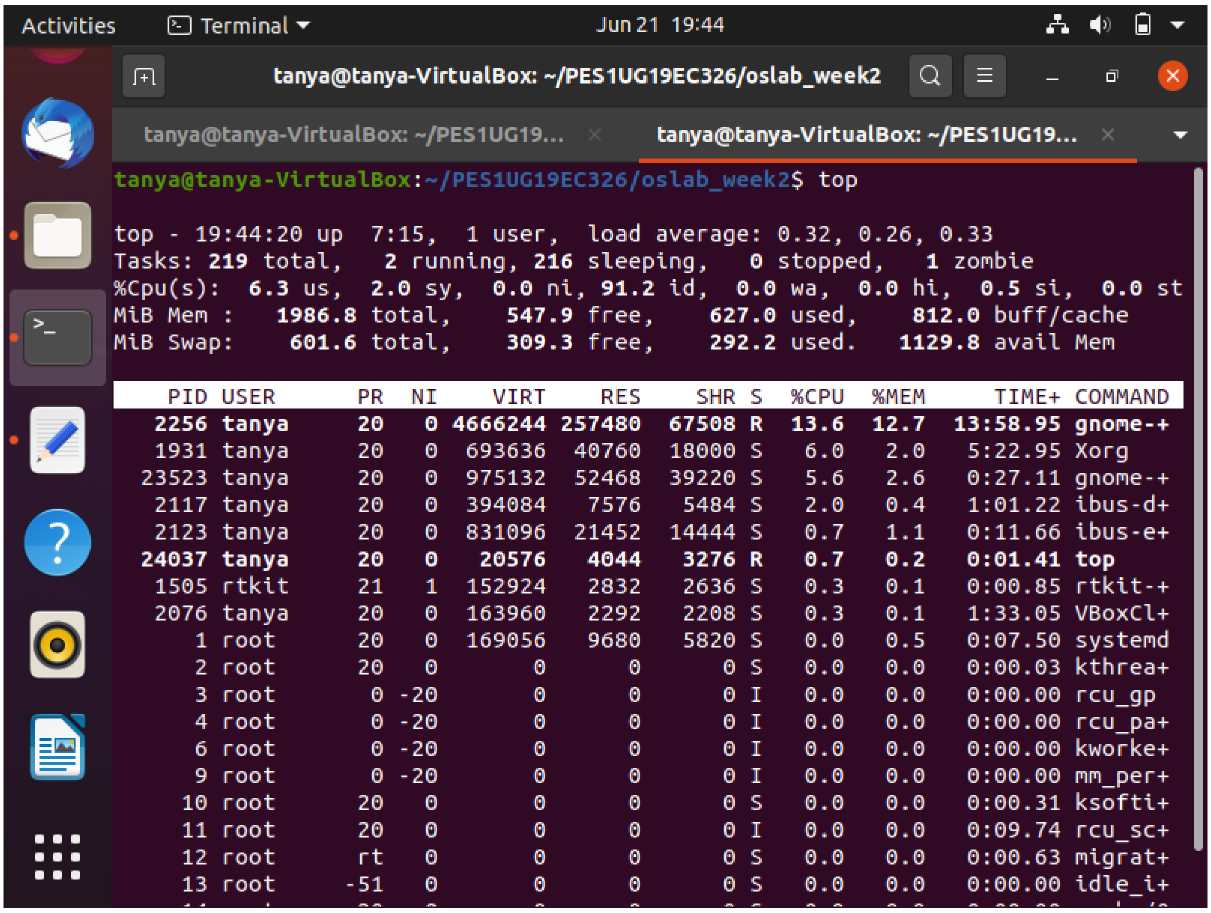
```
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ gcc -o add add.c
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ ./add
Sum: 4210
this is parent process
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$
```

c)

```
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   // Fork returns process id
   // in parent process
   pid_t child_pid = fork();
   // Parent process
   if (child_pid > 0)
      sleep(30);

   // Child process
   else
      exit(0);
   return 0;
}
```

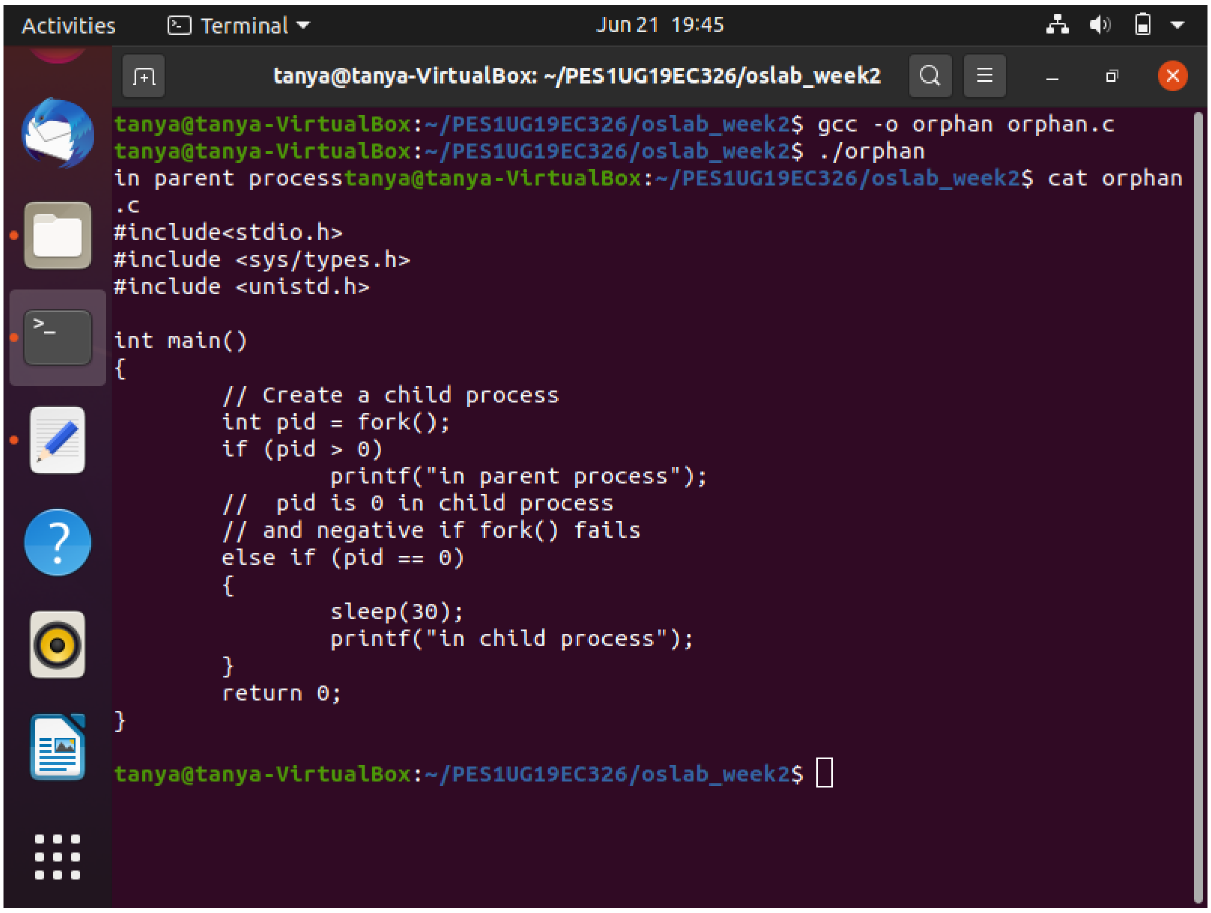

Activities
Terminal
Jun 21 19:44
tanya@tanya-VirtualBox: ~/PES1UG19EC326/oslab_week2
tanya@tanya-VirtualBox: ~/PES1UG19...
tanya@tanya-VirtualBox: ~/PES1UG19...

```
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ gcc -o zombie zombie.c
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ ./zombie
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ cat zombie.c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();
    // Parent process
    if (child_pid > 0)
        sleep(3);

    // Child process
    else
        exit(0);
    return 0;
}
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ 
```


Activities
Terminal
Jun 21 19:44
tanya@tanya-VirtualBox: ~/PES1UG19EC326/oslab_week2
tanya@tanya-VirtualBox: ~/PES1UG19...
tanya@tanya-VirtualBox: ~/PES1UG19...

```
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ top

top - 19:44:20 up  7:15,  1 user,  load average: 0.32, 0.26, 0.33
Tasks: 219 total,   2 running, 216 sleeping,   0 stopped,   1 zombie
%Cpu(s):  6.3 us,  2.0 sy,  0.0 ni, 91.2 id,  0.0 wa,  0.0 hi,  0.5 si,  0.0 st
MiB Mem :   1986.8 total,    547.9 free,    627.0 used,    812.0 buff/cache
MiB Swap:    601.6 total,    309.3 free,    292.2 used.   1129.8 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   2256 tanya     20   0 4666244 257480  67508 R  13.6  12.7  13:58.95 gnome-+
   1931 tanya     20   0  693636  40760  18000 S   6.0   2.0   5:22.95 Xorg
  23523 tanya     20   0  975132  52468  39220 S   5.6   2.6   0:27.11 gnome-+
   2117 tanya     20   0  394084   7576   5484 S   2.0   0.4   1:01.22 ibus-d+
   2123 tanya     20   0  831096  21452  14444 S   0.7   1.1   0:11.66 ibus-e+
  24037 tanya     20   0   20576   4044   3276 R   0.7   0.2   0:01.41 top
   1505 rtkit     21   1  152924   2832   2636 S   0.3   0.1   0:00.85 rtkit-+
   2076 tanya     20   0  163960   2292   2208 S   0.3   0.1   1:33.05 VBoxCl+
      1 root      20   0  169056   9680   5820 S   0.0   0.5   0:07.50 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.03 kthrea+
      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_pa+
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworke+
      9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_per+
     10 root      20   0       0      0      0 S   0.0   0.0   0:00.31 ksofti+
     11 root      20   0       0      0      0 I   0.0   0.0   0:09.74 rcu_sc+
     12 root      rt   0       0      0      0 S   0.0   0.0   0:00.63 migrat+
     13 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_i+
```

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

int main()
{
	// Create a child process
	int pid = fork();
	if (pid > 0)
		printf("in parent process");
	//  pid is 0 in child process
	// and negative if fork() fails
	else if (pid == 0)
	{
		sleep(30);
		printf("in child process");
	}
	return 0;
}
```



```
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ gcc -o orphan orphan.c
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ ./orphan
in parent processtanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ cat orphan
.c
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
        // Create a child process
        int pid = fork();
        if (pid > 0)
                printf("in parent process");
        //  pid is 0 in child process
        // and negative if fork() fails
        else if (pid == 0)
        {
                sleep(30);
                printf("in child process");
        }
        return 0;
}

tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ 
```

d)

```
#include <stdio.h>

#include <sys/types.h>

#include<unistd.h>

int main()

{

int a = 5;

pid_t PID;

printf("Original value: %d\n", a);

PID= fork();

if(PID == 0)

{

printf("Child has a value as: %d\n",--a);}
```

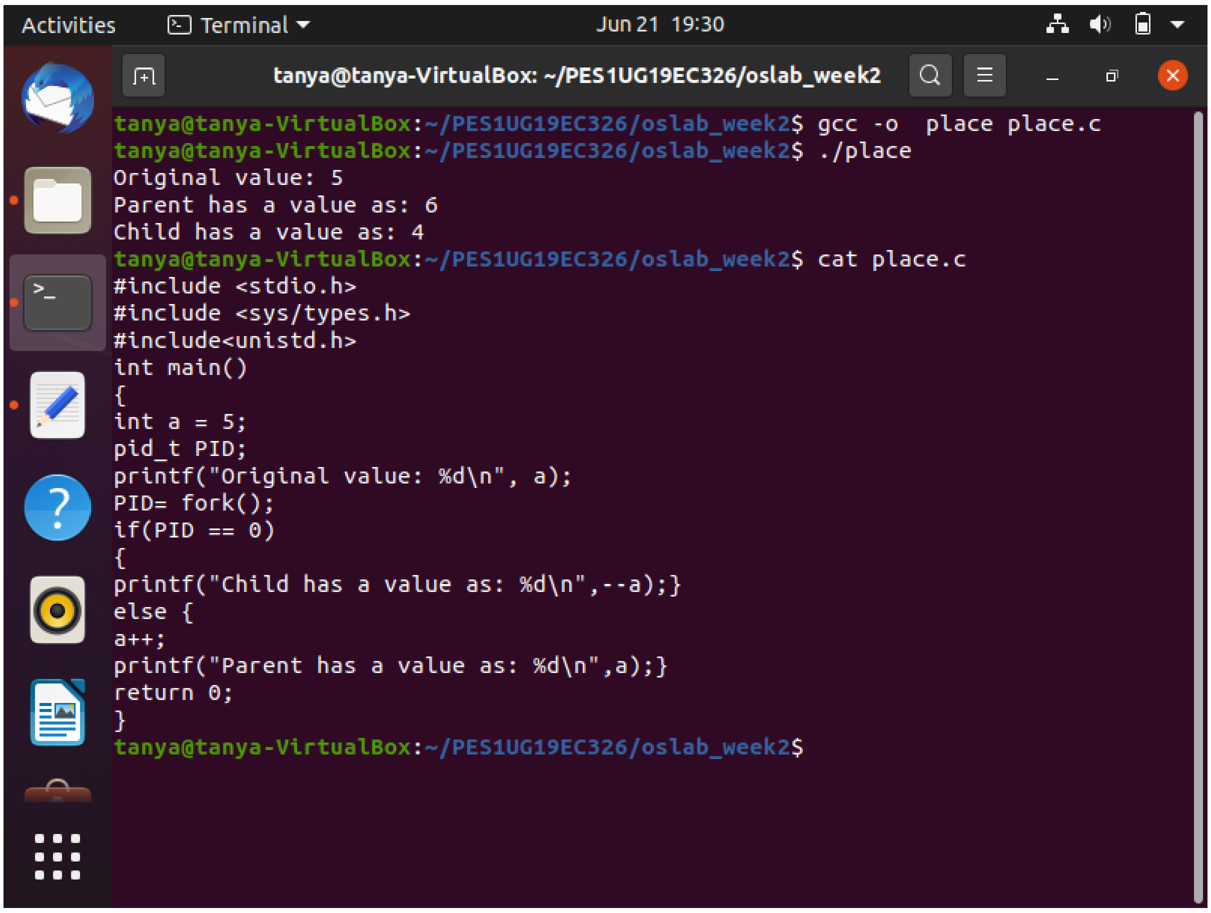
else {

a++;

printf("Parent has a value as: %d\n",a);}

return 0;

}



```
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ gcc -o  place place.c
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ ./place
Original value: 5
Parent has a value as: 6
Child has a value as: 4
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$ cat place.c
#include <stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
int a = 5;
pid_t PID;
printf("Original value: %d\n", a);
PID= fork();
if(PID == 0)
{
printf("Child has a value as: %d\n",--a);}
else {
a++;
printf("Parent has a value as: %d\n",a);}
return 0;
}
tanya@tanya-VirtualBox:~/PES1UG19EC326/oslab_week2$
```

2)

a)

Init is the parent of all processes, executed by the kernel during the booting of a system. Its principle role is to create processes from a script stored in the file. It controls autonomous processes required by any particular system. It is a daemon process which runs till the system is shutdown. That is why, it is the parent of all the processes. Init reads the script stored in the file /etc/inittab**.** Command init reads the initial configuration script which basically take care of everything that a system do at the time of system initialization like setting the clock, initializing the serial port and so on. After determining default runlevel for the system, init starts all background processes required to run the system.

Suppose we have a process P1 is terminated, now during its time when this P1 will move to the zombie state. P1 remains in zombie state until the parent invokes a system call to wait(). When this happened, the process id of P1, as well as the P1 entry in the process table will be released.

But supposedly, if a parent does not perform the wait() system call, now the child process will remain in zombie state till the life of parent. After the death of the parent process, the init process becomes the new parent of the zombie process.

b)

A defunct process, also known as a zombie, is simply a process that is no longer running, but remains in the process table to allow the parent to collect its exit status information before removing it from the process table. Because a zombie is no longer running, it does not use any system resources such as CPU or disk, and it only uses a small amount of memory for storing the exit status and other process-related information in the slot where it resides in the process table. Child processes remain in the process table as defunct processes because many programs are designed to create child processes and then perform various tasks after the child terminates, including restarting the child process. These programs must be able to read the exit status of their child processes, and for this reason, defunct child processes are not removed from the process table as long as their parent process is still running, and has not yet read the exit status information, or has indicated it does not intend to read the exit status information from its children.

Defunct processes are processes that have terminated normally, but they remain visible to the Unix/Linux operating system until the parent process reads their status. Once the status of the process has been read, the operating system removes the process entries.

Hence, "defunct", which means the process has 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 avoid a defunct process we should use exit( ) in child process and wait( ) in parent process so that the parent waits for the child to finish its execution and once the child exits parent will start the execution of its part. In this way, we are doing a clean exit to the child.

c)

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. The process object has to stay around until this happens - it consumes no resources and is dead, but it still exists - hence, 'zombie'.

Zombie processes can be found easily with the ps command. Within the ps output, there is a STAT column that displays 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.

Pipe the output of the ps command through grep to list out any process whose STAT is Z (for zombie).

```
ps aux | grep 'Z'
```

d)

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.