Aim:

1. To Write a c program to implement Interprocess Communication using Pipes.

2. To Write a program to create Zombie process

3. To Write a Program to create orphan process

Description :

Pipes:

Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).

Zombie Process:

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 child process always first becomes a zombie before being removed from the process table.

Orphan Process:

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

1. write():
Header:
#include <unistd.h>
Syntax:
ssize_t write(int fd, const void *buf, size_t count);
Description:
write() writes up to count bytes from the buffer starting at buffer
to the file referred to by the file descriptor fd.
The number of bytes written may be less than count if, for
example, there is insufficient space on the underlying physical
medium, or the RLIMIT_FSIZE resource limit is encountered
or the call was interrupted by a signal handler
after having written less than count bytes

2. read():
Header:
#include <unistd.h>
Syntax:

ssize_t read(int fd, void *buf, size_t count);

Description:

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.
On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.

OS Lab Exercise 2 -Implementation Of System Calls -B.Ajay(21110103)
If count is zero, read() may detect the errors described below. In the absence of any errors, or if read() does not check for errors, a read() with a count of 0 returns zero and has no other effect

3. sleep(time):

Header:

#include <unistd.h>

Syntax:

unsigned int sleep(unsigned int seconds);

Description:

sleep() causes the calling thread to sleep either until the number of real-time seconds specified in seconds have elapsed or until a signal arrives which is not ignored.

4.close(int fd):

Header:

#include <unistd.h>

Syntax:

int close(int *fd*);

Description:

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see fcntl(2)) held

on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

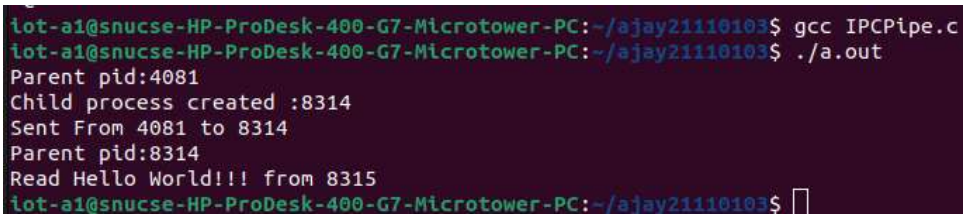
Code:

IPCPipe.c:

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
int main(){

        char* write1=(char*)malloc(sizeof(char)*30);
        char* read1=(char*) malloc(sizeof(char)*30);
        pid_t pid;
        int fd[2];
        write1= "Hello World!!!";
        if(pipe(fd)==0){
        pid=fork();
        printf("Parent pid:%d\n",getppid());
        if(pid>0){
                printf("Child process created :%d\n",getpid());
                close(fd[0]);
                write(fd[1],write1,strlen(write1)+1);
                printf("Sent From %d to %d\n",getppid(),getpid());
                close(fd[1]);
        }
        else{
                close(fd[1]);
                read(fd[0],read1,30);
```

```
                printf("Read %s from %d\n",read1,getpid());
                close(fd[0]);
        }
        }
}
```

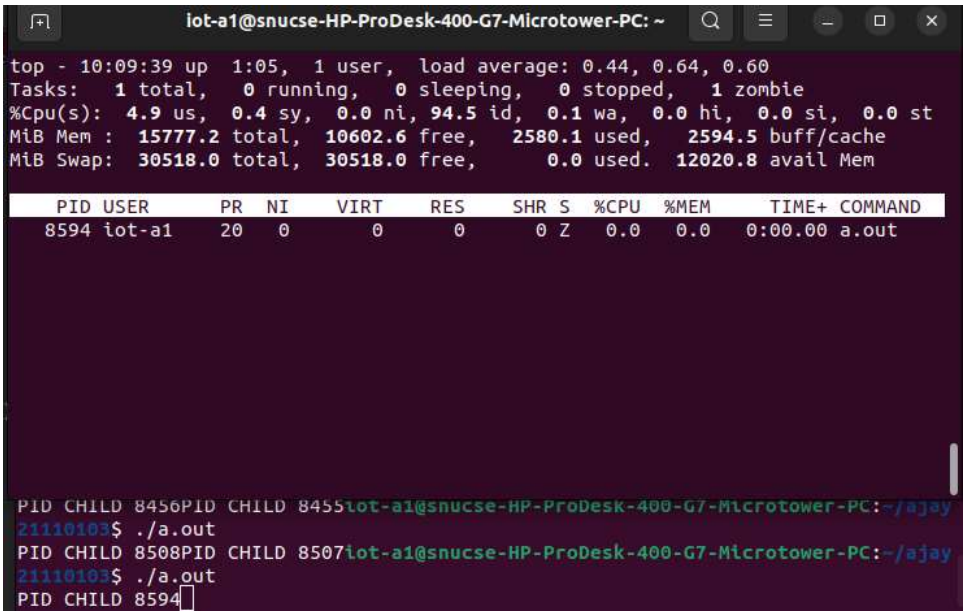
OUTPUT:



Zombie.c:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
int main(){
        pid_t pid=fork();
        printf("PID CHILD %d",getpid());
        if(pid>0){
                sleep(34);

        }else{
                exit(0);
                }

}
```

OUTPUT:

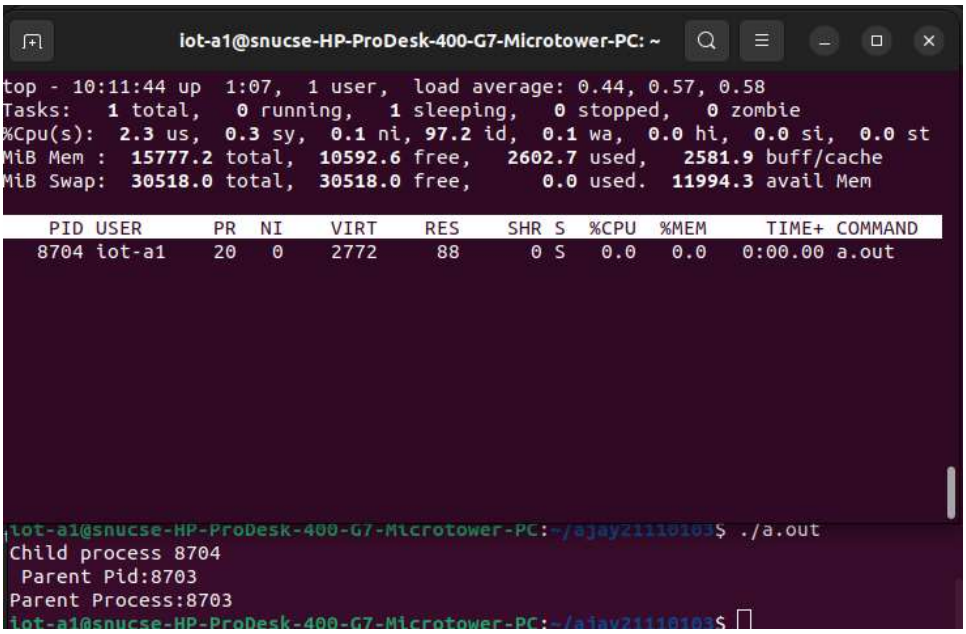


Orphan.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
int p=fork();
if(p>0){
sleep(10);
printf("Parent Process:%d\n",getpid());
//                  printf("Parent Terminating");



}
else {
printf("Child process %d\n Parent Pid:%d\n",getpid(),getppid());
sleep(30);
```

```
exit(0);


}


}
```

OUTPUT (Orphan child still exist after parent terminates)



Result:

Thus the IPC Communication was done through anonymous pipes and zombie,orphan process created using the appropriate syscalls.