OPERATING SYSTEM LAB

Name: Thorat Amey Arun Reg No.: 23MCS1004

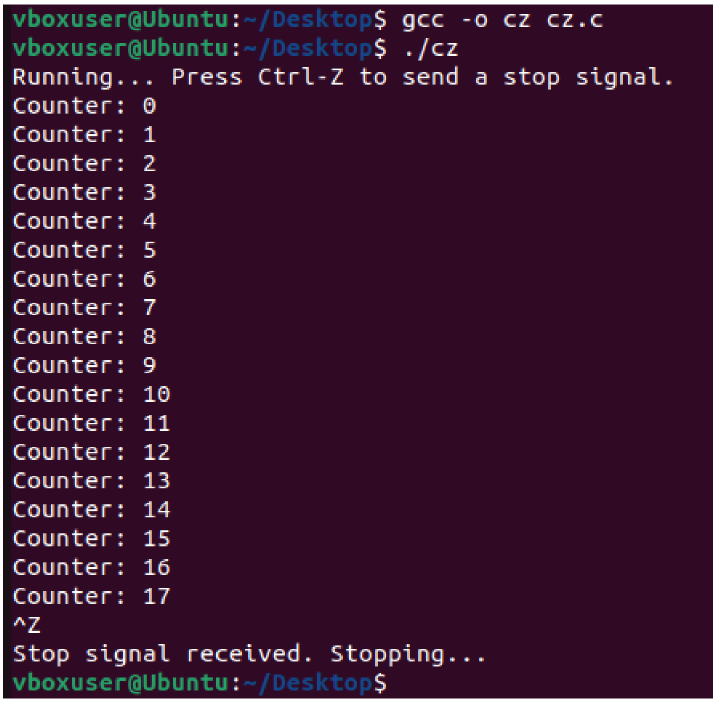
LAB EXPERIMENT 4
Signal Handling

1. Write your own C handlers to handle the following signals
   a. Send a stop signal using Ctrl-Z

      Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

// Signal handler function
void stopSignalHandler(int signum) {
   printf("\nStop signal received. Stopping...\n");
   exit(0);
}

int main() {
   // Set up the signal handler
   signal(SIGTSTP, stopSignalHandler);

   printf("Running... Press Ctrl-Z to send a stop signal.\n");
   int counter=0;
   while (1) {
      printf("Counter: %d\n", counter);
      counter++;
      // Simulate some work or processing here
      for (int i = 0; i < 100000000; i++) {

      }
   }

   return 0;
}
```

      Output:



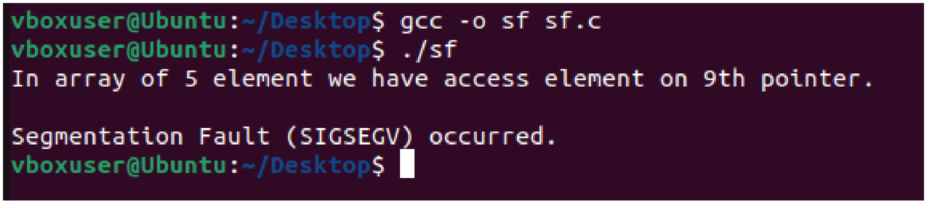
```
vboxuser@Ubuntu:~/Desktop$ gcc -o cz cz.c
vboxuser@Ubuntu:~/Desktop$ ./cz
Running... Press Ctrl-Z to send a stop signal.
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
Counter: 8
Counter: 9
Counter: 10
Counter: 11
Counter: 12
Counter: 13
Counter: 14
Counter: 15
Counter: 16
Counter: 17
^Z
Stop signal received. Stopping...
vboxuser@Ubuntu:~/Desktop$
```

b. Segmentation fault

<u>Program:</u>

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void segFaultHandler(int signal) {
   printf("\nSegmentation Fault (SIGSEGV) occurred.\n");

   exit(-1);
}


int main() {
   signal(SIGSEGV, segFaultHandler);
   int arr[5];
   arr[10] = 42; // Accessing an element outside the bounds of the
array
   printf("In array of 5 element we have access element on 9th
pointer.\n");
   raise(SIGSEGV);
   return 0;
}
```

Output:



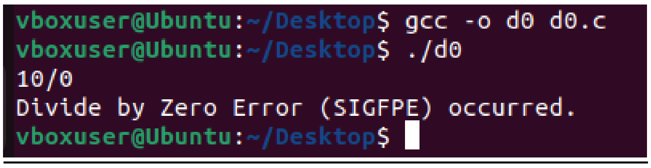
```
vboxuser@Ubuntu:~/Desktop$ gcc -o sf sf.c
vboxuser@Ubuntu:~/Desktop$ ./sf
In array of 5 element we have access element on 9th pointer.

Segmentation Fault (SIGSEGV) occurred.
vboxuser@Ubuntu:~/Desktop$ 
```

c. Divide by zero error

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void divByZeroHandler(int signal) {
   printf("\nDivide by Zero Error (SIGFPE) occurred.\n");
   exit(-1);
}

int main() {
   signal(SIGFPE, divByZeroHandler);
   int numerator = 10;
   int denominator = 0;
   printf("10/0");
   int result = numerator / denominator; // Division by zero
   raise(SIGFPE);
   return 0;
}
```

Output:



```
vboxuser@Ubuntu:~/Desktop$ gcc -o d0 d0.c
vboxuser@Ubuntu:~/Desktop$ ./d0
10/0
Divide by Zero Error (SIGFPE) occurred.
vboxuser@Ubuntu:~/Desktop$ 
```

2. Write a program which creates a child process and continues to run along with its child (choose any small task of your own). Once the child completes its task, it should send a signal to parent which in turn

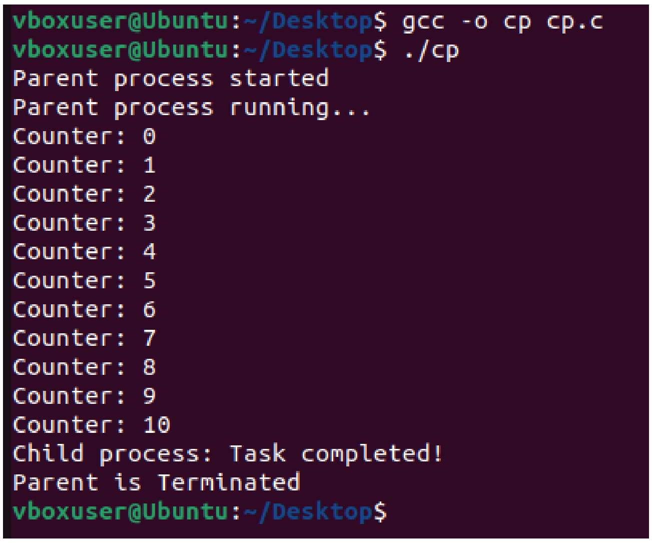
terminates the parent. (Expected output: output of the task carried out by the child process, termination of parent)

Program:

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

void Kill(int signal) {
   printf("Parent is Terminated\n");
   exit(-1);
}

void child_task() {
   int counter = 0;
   while (counter!=11) {
      printf("Counter: %d\n", counter);
      counter++;
      // Simulate some work or processing here
      for (int i = 0; i < 11; i++) {
      }
   }
   printf("Child process: Task completed!\n");
   kill(getppid(), SIGTERM); // Send termination signal to parent
}

int main() {
   signal(SIGTERM, Kill);
   printf("Parent process started\n");

   pid_t child_pid = fork();

   if (child_pid == -1) {
      perror("Fork failed");
      exit(1);
   }
```

```
    if (child_pid == 0) {
        // Child process
        child_task();
        exit(0);
    } else {
        // Parent process
        while (1) {
            printf("Parent process running...\n");
            sleep(2); // Simulate some parent activity
        }
    }

    return 0;
}
```

Output:



```
vboxuser@Ubuntu:~/Desktop$ gcc -o cp cp.c
vboxuser@Ubuntu:~/Desktop$ ./cp
Parent process started
Parent process running...
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
Counter: 8
Counter: 9
Counter: 10
Child process: Task completed!
Parent is Terminated
vboxuser@Ubuntu:~/Desktop$
```

3. Write two c programs: One displaying the PID infinitely and the other program sending a signal to terminate the first program.(Note: Execute the programs in separate terminals)

   Program:

   dp.c:

   ```
   #include <stdio.h>
   #include <unistd.h>

   int main() {
   ```

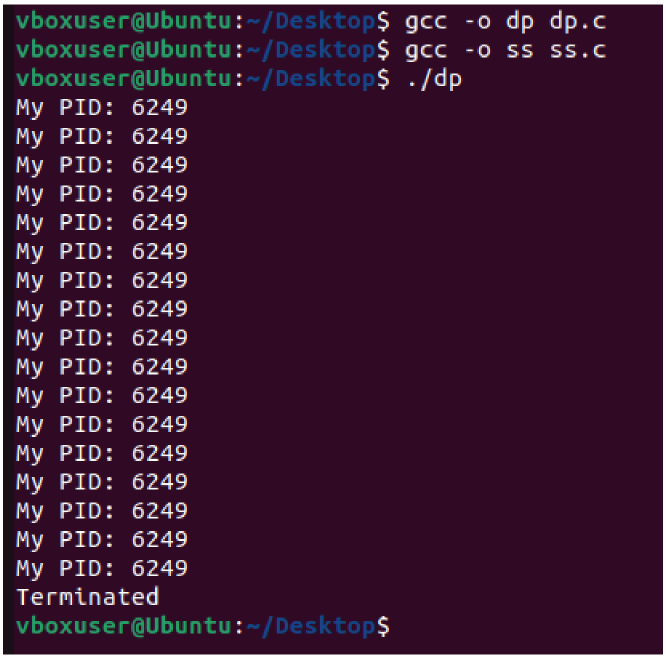
```
    while (1) {
        printf("My PID: %d\n", getpid());
        sleep(1);
    }

    return 0;
}
```

ss.c:

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        return 1;
    }

    pid_t pid = atoi(argv[1]);

    if (kill(pid, SIGTERM) == 0) {
        printf("Termination signal sent to PID %d\n", pid);
    } else {
        perror("kill");
        return 1;
    }

    return 0;
}
```

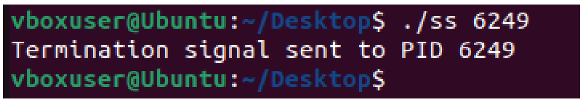
Output:

dp.c:



```
vboxuser@Ubuntu:~/Desktop$ gcc -o dp dp.c
vboxuser@Ubuntu:~/Desktop$ gcc -o ss ss.c
vboxuser@Ubuntu:~/Desktop$ ./dp
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
My PID: 6249
Terminated
vboxuser@Ubuntu:~/Desktop$
```

ss.c:



```
vboxuser@Ubuntu:~/Desktop$ ./ss 6249
Termination signal sent to PID 6249
vboxuser@Ubuntu:~/Desktop$
```

4. Execute this code and explain:

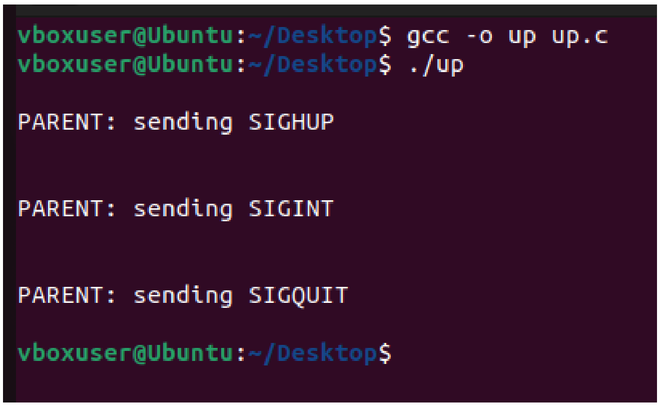
<u>Program:</u>

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

// function declaration
void sighup();
void sigint();
void sigquit();

// driver code
void main()
{
   int pid;

   /* get child process */
```

```
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) { /* child */
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for (;;)
            ; /* loop for ever */
    }

    else /* parent */
    { /* pid hold id of child */
        printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid, SIGHUP);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid, SIGINT);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid, SIGQUIT);
        sleep(3);
    }
}

// sighup() function definition
void sighup()

{
    signal(SIGHUP, sighup); /* reset signal */
    printf("CHILD: I have received a SIGHUP\n");
}

// sigint() function definition
void sigint()
```

```
{
   signal(SIGINT, sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

// sigquit() function definition
void sigquit()
{
   printf("My DADDY has Killed me!!!\n");
   exit(0);
}
```

Output:



```
vboxuser@Ubuntu:~/Desktop$ gcc -o up up.c
vboxuser@Ubuntu:~/Desktop$ ./up

PARENT: sending SIGHUP


PARENT: sending SIGINT


PARENT: sending SIGQUIT

vboxuser@Ubuntu:~/Desktop$
```

Explanation:

A child process is created using the fork() system call. If the fork() call fails, an error message is printed, and the program exits.

a. *Child Process Execution:*
   Inside the child process block (if (pid == 0)), signal handlers are set for SIGHUP, SIGINT, and SIGQUIT.
   The child process sets up three signal handlers using the signal() function. The signal() function associates a specific signal with a handler function. In this case, sighup(), sigint(), and sigquit() functions will be called when SIGHUP, SIGINT, and SIGQUIT signals are received, respectively. The child then enters an infinite loop, which ensures that it remains alive and can respond to signals.

b. *Parent Process Execution:*

In the parent process block (else), the parent sends signals to the child with a delay using the kill() function and then sleeps.
The parent process sends the SIGHUP, SIGINT, and SIGQUIT signals to the child process in sequence, and then sleeps for 3 seconds after each signal. The output shows the interaction between the parent and child processes.

c. Signal Handler Function Definitions:
These functions define the behavior when each signal is received. In this case, they print messages to indicate which signal was received. The sigquit() function also prints a message before the child process exits.

5. Create a chat application using shared memory concept.
Program:
u1.c:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

#define FILLED 0
#define Ready 1
#define NotReady -1

struct memory {
        char buff[100];
        int status, pid1, pid2;
};

struct memory* shmptr;

// handler function to print message received from user2

void handler(int signum)
```

```
        {
                // if signum is SIGUSR1, then user 1 is receiving a message from
user2

                if (signum == SIGUSR1) {
                        printf("Received User2: ");
                        puts(shmptr->buff);
                }
        }

        int main()
        {
                // process id of user1

                int pid = getpid();

                int shmid;

                // key value of shared memory
                int key = 12345;

                // shared memory create
                shmid = shmget(key, sizeof(struct memory), IPC_CREAT | 0666);

                // attaching the shared memory

                shmptr = (struct memory*)shmat(shmid, NULL, 0);

                // store the process id of user1 in shared memory
                shmptr->pid1 = pid;
                shmptr->status = NotReady;

                // calling the signal function using signal type SIGUSER1
                signal(SIGUSR1, handler);

                while (1) {
                        while (shmptr->status != Ready)
                                continue;
                        sleep(1);
        // taking input from user1

        printf("User1: ");
        fgets(shmptr->buff, 100, stdin);

        shmptr->status = FILLED;

        // sending the message to user2 using kill function

        kill(shmptr->pid2, SIGUSR2);
    }

    shmdt((void*)shmptr);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

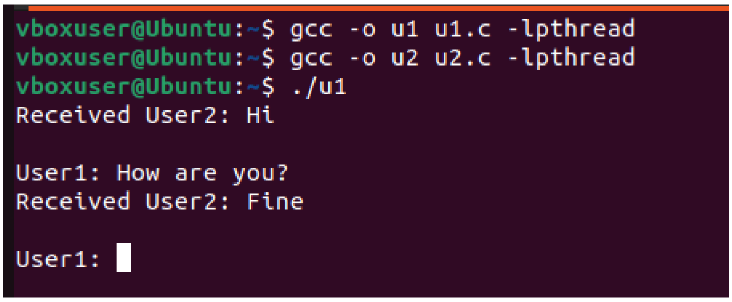
u2.c:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

#define FILLED 0
#define Ready 1
#define NotReady -1

struct memory {
    char buff[100];
    int status, pid1, pid2;
};

struct memory* shmptr;
```

```
// handler function to print message received from user1

void handler(int signum)
{
        // if signum is SIGUSR2, then user 2 is receiving a message from
user1

        if (signum == SIGUSR2) {
                printf("Received From User1: ");
                puts(shmptr->buff);
        }
}

// main function

int main()
{
        // process id of user2
        int pid = getpid();

        int shmid;

        // key value of shared memory
        int key = 12345;

        // shared memory create

        shmid = shmget(key, sizeof(struct memory), IPC_CREAT | 0666);

        // attaching the shared memory

        shmptr = (struct memory*)shmat(shmid, NULL, 0);

        // store the process id of user2 in shared memory
        shmptr->pid2 = pid;

        shmptr->status = NotReady;

        // calling the signal function using signal type SIGUSR2
        signal(SIGUSR2, handler);

        while (1) {
                sleep(1);

                // taking input from user2

                printf("User2: ");
                fgets(shmptr->buff, 100, stdin);
                shmptr->status = Ready;

                // sending the message to user1 using kill function

                kill(shmptr->pid1, SIGUSR1);

                while (shmptr->status == Ready)
                        continue;
        }

        shmdt((void*)shmptr);
        return 0;
}
```

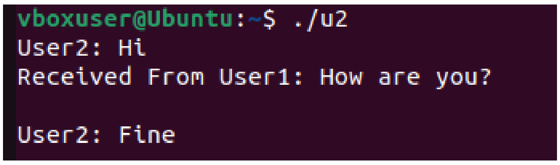
Output:

u1.c:



```
vboxuser@Ubuntu:~$ gcc -o u1 u1.c -lpthread
vboxuser@Ubuntu:~$ gcc -o u2 u2.c -lpthread
vboxuser@Ubuntu:~$ ./u1
Received User2: Hi

User1: How are you?
Received User2: Fine

User1: 
```

u2.c:



```
vboxuser@Ubuntu:~$ ./u2
User2: Hi
Received From User1: How are you?

User2: Fine
```