CPE 435: OPERATING SYSTEMS LABORATORY.

Lab08

POSIX Signals.

Submitted by: Dan Otieno.

Date of Experiment: 03/03/23.

Report Deadline: 03/10/23.

Demonstration Deadline: 03/10/23.

Introduction:

The purpose of this lab was to understand the concept of signals. Signals are some of the mechanisms through which processes communicate in an operating system. Testing purposes we determine how user defined handlers for particular signals can replace the default signal handlers and also how the processes can ignore the signals.

Theory:

According to the lab manual, Signals inform processes of the occurrence of asynchronous events. Some of the inter-process communication methods we have used in this course include Pipes (piping), Message Queuing, Shared Memory techniques.

- ## Piping:
  - A pipe is an unidirectional data channel that can be used for interprocess communication. An array is used to return two file descriptors that refer to each end of the pipe, read end and write end. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.– (Source is CPE 435, Lab 3 material).

- ## Message Queuing:
  - Message queues are an example of inter-process communication in LINUX. All messages are stored in the kernel and have a queue identifier, or numeric key called msqid . The msqid identifies particular message queues, and then processes read and write messages to arbitrary message queues. – (Source is my Lab 5 report).

- ## Shared Memory Methods:
  - Shared Memory works by setting up one process to create a shared segment, whose process ID can be accessed by other processes. Other processes use a key to interact with the shared segment that was created by the first process, in which they can read and write data. When done, each processes detaches from the shared segment. – (Source is my Lab 5 report).

Results & Observation:

Assignment 1:

Description:

The goal for this assignment was to use C/C++ to write a parent and child process and simulate how signals are transmitted using the processes. If the user sends ctrl+c to the parent process before 10 seconds from start, then the parent catches the signal and sends a message to indicate that the system is protected, while the child ignores the signal. Messages are printed from both processes when ctrl+c is detected to show they cannot be killed, after 10 seconds, the parent process can be terminated, but also sends a signal to the child to terminate first.

Assignment 2:

Description:

This assignment required creating a Linux time bomb. The program starts on idle, doing nothing until ctrl+c is entered by the user, after which the program prints gibberish on the console, and then terminates after 10 seconds.

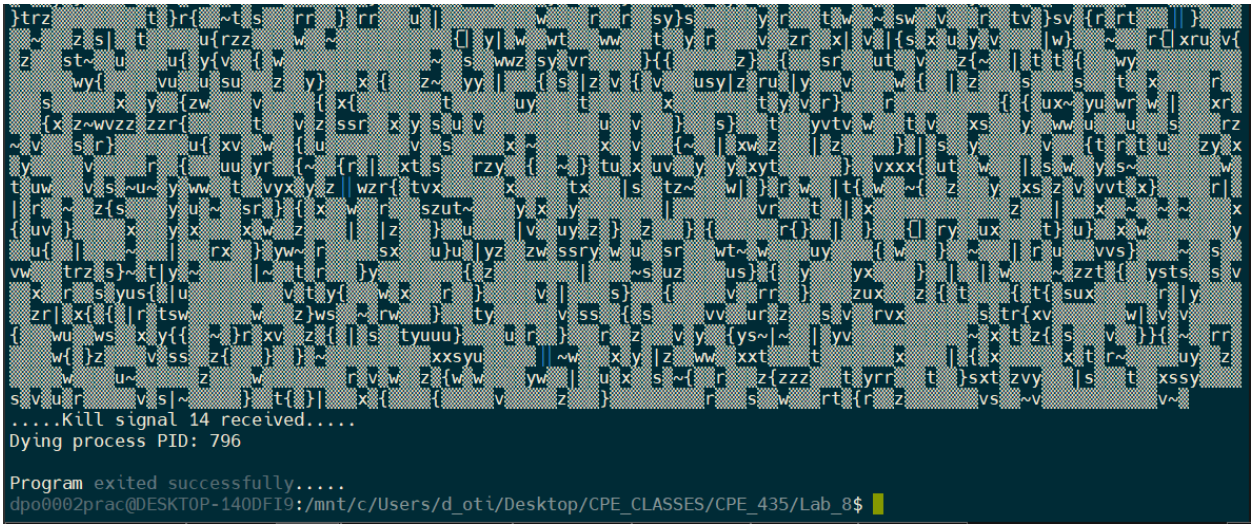
Program Outputs:

- **Assignment 1 output:**

```
dpo0002prac@DESKTOP-14ODFI9:~$ cd ../
dpo0002prac@DESKTOP-14ODFI9:/home$ cd ../
dpo0002prac@DESKTOP-14ODFI9:/$ cd mnt/c/Users/d_oti/Desktop/CPE_CLASSES/CPE_435/Lab_8
dpo0002prac@DESKTOP-14ODFI9:/mnt/c/Users/d_oti/Desktop/CPE_CLASSES/CPE_435/Lab_8$ ./l81
^C
!Please wait for process to complete. System is protected!
.....Kill signal received.....14
Now you can kill me...
^C
.....Kill signal received.....2
.....Executing child kill process.....
.....Parent has submitted child kill request.....
        Dying process 31296
.....Child kill process complete.....
        Dying process 31295
dpo0002prac@DESKTOP-14ODFI9:/mnt/c/Users/d_oti/Desktop/CPE_CLASSES/CPE_435/Lab_8$
```

- **Assignment 2 outputs:**

```
dpo0002prac@DESKTOP-14ODFI9:/mnt/c/Users/d_oti/Desktop/CPE_CLASSES/CPE_435/Lab_8$ ./l82
.....Idle time, I'm doing nothing.....
```



Conclusion:

Both programs worked as expected. This is the first lab in a while to use the parent-child relationship and forking to simulate OS processes. But it was also a great learning curve to understand the various signal functionalities built in the library.

Appendix:

Assignment 1 code – Round Robin.

```
/*
DAN OTIENO
CPE 435-01
LAB 7
Round Robin.
*/

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

pid_t pid = 0;

void fnKill(int killSignal)
{
        printf("\n.....Kill signal received.....%d\n", killSignal);

        printf(".....Executing child kill process.....\n");
        kill(pid, SIGTERM);
        wait(0);
        printf(".....Child kill process complete.....\n");

        printf("\tDying process %d\n", getpid());
```

```
        exit(0);
}


void fnKillChild(int killSignal)
{
        printf(".....Parent has submitted child kill request.....\n");
        printf("\tDying process %d\n", getpid());
        exit(0);
}


void myFunction(int sigVal) // Slightly modified from demo.
{
        printf(".....Kill signal received.....%d\n", sigVal);
        printf("Now you can kill me...\n");
        signal(SIGINT, fnKill);
}


void fnProtect(int sigVal)
{
        if(pid == 0)
        {
                return;
        }
        else if (pid > 0)
        {
                printf("\n!Please wait for process to complete. System is
protected!\n");
        }
}

int main()
{
        pid = fork();

        if(pid == 0)
        {
                signal(SIGINT, fnProtect);
                signal(SIGTERM, fnKillChild);
                while(1);
        }
        else
        {
                signal(SIGINT, fnProtect);
                signal(SIGALRM, myFunction);
                alarm(10);
                while(1);
        }
        while(1);
        return(0);
}
```

Assignment 1 code.

```
/*
DAN OTIENO
CPE 435-01
LAB 8
Assignment 1.
*/
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
struct process
{
        int pid; //Process ID
        int priority;
        int bTime; //CPU Burst Time
        int wTime; //Time this process executed, if wTime==bTime, process is
complete
        int trnTime; //turn around time, time needed for the process to
complete
        int wtTime; // Wait Time = Turn around Time - Burst Time
};


/***** Function to print hyphens *****/
void dashes(int d)
{
        printf("\n");
        for(int i = 0; i < d; i++)
        {
                printf("-");
        }
        printf("\n");
}
```

```
/**********************************/
void swap(struct process *a, struct process *b)
{
        struct process tmp = *a;
        *a=*b;
        *b=tmp;
}
int main(int argc, char* argv[])
{
        int n = atoi(argv[1]);
        int quantum = atoi(argv[2]);
        int avgwt;
        struct process *prc;

        prc = malloc(sizeof(struct process) * n); // Memory allocation for n
processes.

        if (prc == 0)
        {
                exit(1);
        }

        for (int i = 0; i < n; i++)
        {
                printf("Process %d of %d\n", i+1, n);
                prc[i].pid = i+1;
                printf("\tEnter burst time: ");
                scanf("%d", &prc[i].bTime);
                printf("\tEnter Priority: ");
                scanf("%d", &prc[i].priority);
        }

        for(int i = 0; i < n-1; i++)
```

```
        {
                for(int j = 0; j < n-i-1; j++)
                {
                        if(prc[j].priority > prc[j+1].priority)
                        {
                                swap(&prc[j], &prc[j+1]);
                        }
                }
        }

        int count = 0;
        prc[0].wtTime = 0;

        for(int i = 1; i < n; i++)
        {
                prc[i].wtTime = 0;
                for(int j = 0; j < i; j++)
                {
                        prc[i].wtTime += prc[j].bTime;
                }
                count += prc[i].wtTime;
        }

        avgwt = (float)count / n;

        int count2 = 0;
        for (int i = 0; i < n; i++)
        {
                prc[i].trnTime = prc[i].bTime + prc[i].wtTime;
                count2 += prc[i].trnTime;
        }

/******************** Printing Outputs *********************/
```

```
        dashes(49);
        printf("| Time (ms) |");
        for (int i = 0; i < n; i++)
        {
                printf("     %d    |", prc[i].bTime);
        }
        dashes(49);
        printf("| PID       |");
        for (int i = 0; i < n; i++)
        {
                printf("      %d    |", prc[i].priority);
        }
        dashes(49);

        printf("\n");
        dashes(70);
        printf("| PID                    |   ");
        for (int i = 0; i < n; i++)
        {
                printf("      %d    |   ", prc[i].pid);
        }
        dashes(70);

        printf("| Wait Time (ms)         |   ");
        for (int i = 0; i < n; i++)
        {
                printf("     %d    |   ", prc[i].wtTime);
        }
        dashes(70);

        printf("| Turn Around Time (ms) |   ");
        for (int i = 0; i < n; i++)
        {
```

```
                printf("     %d     |    ", prc[i].trnTime);
        }
        dashes(70);
        printf("\n");
        dashes(38);
        printf("| Average Wait Time (ms) |     %d     |", avgwt);
        dashes(38);
/*********************************************************/
        void free(void*); // Memory deallocation.
        return 0;
}
```

Assignment 2 code.

```
/*
DAN OTIENO
CPE 435-01
LAB 8
Assignment 1.
*/

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

char ch;

void fnKill(int killSignal)
{
        printf("\n.....Kill signal %d received.....\n", killSignal);
        printf("Dying process PID: %d\n", getpid());
        printf("\nProgram exited successfully.....\n");
        exit(0);
}

void tBomb(int sigVal)
{
        printf("\n.....Signal %d received.....\n", sigVal);
        signal(SIGALRM, fnKill);
        alarm(10);
```

```
        while(1)
        {
                ch = '1' + 'A' + (random() % 36);
                printf("%c", ch);
        }
        exit(1);
}


int main()
{
        printf(".....Idle time, I'm doing nothing.....\n");
        signal(SIGINT, tBomb);
        while(1);
        return -1;
}
```