Course: Operating Systems 28 February 2021
Name: Zeynep Cankara
Student Id: 21703381

Project 1

Intercepting Shell Program

Introduction

The project is about implementing a simple command line interpreter which resembles the UNIX shell. The program is an intercepting shell program supporting common various UNIX command together with commands such as `exit` and `help`. The shell program operates in two modes which are normal mode and the tapped mode. Both of these modes works with composed commands which are commands separated with symbol `'|'`.

Running the intercepting shell program (isp)

The shell can run via the following command where <N> is the number of bytes read and written from each system call and <mode> signifies the pipe mode to used where $mode = 1$ stands for the normal mode and $mode = 2$ stands for the tapped mode.

```
isp <N> <mode>
```

The experiments done using the commands in the following format when the shell is running where <M> is the random alphanumerical letters produced.

```
./producer <M> | ./consumer <M>
```

Experiments

I have used the `gettimeofday()` method before and after executing composed commands to display the statistics about the affects of the shell parameter changes on the program execution time. I specifically take the time statistic about `real` execution time since it is total time elapsed from start to finish.

Read and write count statistics for various values of N (M = 1000000/ tapped mode)		
N/M(=1000000)	read-count	write-count
1	1000001	1000000
256	147998	147997
512	136791	136790
1024	149468	149467

The table above displays how read count and write count changes as the number of bytes to read and write changes in one system call ($M = 1000000$) . The number of read-counts and write-counts do not change much after the `N` value of 256. One reason might be that `read` command allocating more bytes than the needed in the first place.

Time statistics for various N and M values in milliseconds (Tapped mode)				
N/M	10	100	10000	1000000
1	0.909ms	1.706ms	117.297ms	22529.848ms
256	0.869ms	1.952ms	123.507ms	7093.092ms
512	0.827ms	2.123ms	100.910ms	6472.611ms
1024	0.827ms	2.336ms	103.573ms	7007.026ms

Time statistics for various N and M values in seconds (Normal mode)				
N/M	10	100	10000	1000000
1	0.862ms	2.170ms	38.601ms	4857.360ms
256	0.999ms	1.423ms	53.287ms	5173.477ms
512	1.619ms	1.241ms	81.216ms	4767.255ms
1024	0.831ms	0.900ms	29.990ms	5058.102ms

The tables above demonstrates the affects of parameters `M` and `N` on the shell performance for both normal and tapped modes. As it can be seen from the both tables above as the `N` value increase the execution time of a composed command decreases in most cases yet there is not strong relationship we can conclude from the experiment alone. On the other hand, as the `M` value increases we can see that the time elapsed increases as well, it makes sense since we expect more bytes produced and consumed leading up to the more processing time thus more time elapsed in milliseconds. If we compare the results of the normal mode with the tapped mode version of the table above we can see that in general executing in the normal mode takes less time in general. Especially for the larger `M` values. One reason might be the overhead caused by transferring data first to the main process rather than directly transferring to the other pipe like in the normal mode.

Codes for the producer and consumer programs

I'm providing the codes of the `producer` and `consumer` programs below which I used to conduct timing experiments.

producer.c

```c
// Copyright 2021 by the Zeynep Cankara. All rights reserved.
// Program produces M random alphanum letters incrementally.

// library imports
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <string.h>

```

```c
// definition(s)
#define FD_OUT 1
#define ALPHANUMERIC_LEN 36

// constants
const char ALPHANUM[ALPHANUMERIC_LEN] = {'a', 'b', 'c', 'd', 'e', 'f',
                                         'g', 'h', 'i', 'j', 'k', 'l',
                                         'm', 'n', 'o', 'p', 'q', 'r',
                                         's', 't', 'u', 'v', 'w', 'x',
                                         'y', 'z', '0', '1', '2', '3',
                                         '4', '5', '6', '7', '8', '9'};
// Main function
int main(int argc, char *argv[])
{
    int M = 0;
    unsigned int numberOfChars = 0;
    if (argc > 1)
    {
        // M is the number of bytes for the experiments
        M = atoi(argv[1]);
    }

    while (numberOfChars < M)
    {
        int randomIndex = ALPHANUMERIC_LEN % rand();
        char charToWrite = ALPHANUM[randomIndex];
        numberOfChars += write(FD_OUT, &charToWrite, 1);
    }
    return 0;
}
```

consumer.c

```c
// Copyright 2021 by the Zeynep Cankara. All rights reserved.
// Program consumes M random alphanum letters incrementally.

// imports
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

// definition
```

```c
#define FD_IN 0

// Comsumer program
int main(int argc, char *argv[])
{
    int M = 0;
    if (argc > 1)
    {
        // M is the number of bytes for the experiments
        M = atoi(argv[1]);
    }
    unsigned int numberOfChars = 0;
    while (numberOfChars < M)
    {
        char toRead;
        numberOfChars += read(FD_IN, &toRead, 1);
    }
    return 0;
}
```

Conclusion

This assignment helped me to understand how I can implement my own command line interpreter in a simplified way. I believe after completing the project I can see the importance of how various pipe designs affects the performance of an interpreter. Additionally, I learnt a lot about the pipes in Operating Systems and how they enable the *I/O redirection*.