Project 1

CS342 - Operating Systems

Efe Acer
21602217

Bilkent University, CS

Contents

Introduction

The aim of this project is to develop a simple command line interpreter that is similar to the *Unix shell*, and conduct an experiment that uses it. The program is called "*bilshell*" and it is capable of executing common commands such as `cd`, `exit`, `help`; and many Unix-based system calls. *bilshell* can operate in two modes, batch mode and interactive mode. In batch mode, the commands listed inside an input file line by line are executed in order. In interactive mode, the user specifies the commands to execute until she wants to exit. *bilshell* also supports composition of two commands, where the output produced by a command is fed to the other as input.

Implementation and Experiment Description

The composed command execution is handled using the primitive Inter Process Communication (IPC) mechanism of Unix, namely *ordinary pipes*. This mechanism provides a unidirectional communication channel to processes, where the channel has a read end and a write end. In the particular case of *bilshell*, the first part of a composed command is executed separately in a child process, say *child 1*, and the second part is again executed in another child process, *child 2*. After *child 1* finishes executing the first part of the command, it writes the corresponding output to a pipe, say *pipe 1*. Then, the parent process reads from *pipe 1*, and writes the bytes it read as input to another pipe, *pipe 2*, `N` bytes at a time. Lastly, *child 2* reads from *pipe 2* and executes the second part of the command.

The implementation procedure explained above is a particular way of solving the *producer-consumer* problem. *producer-consumer* problem corresponds to the common practical case, where a *producer* process produces some information, which a *consumer* process needs to consume. [1]

In order to investigate the effect of some parameters on the performance of our particular solution to the *producer-consumer* problem, we conduct an experiment. First, we develop a *producer* program, `producer.c`, that produces `M` random alphanumeric characters and a *consumer* program, `consumer.c` that consumes `M` characters as input. The composed command execution mechanism of *bilshell* ties the output of the *producer* to the input of the *consumer* and lets the *consumer* consume `M` bytes of information, by transferring N bytes at a time. We change the values of `N` and `M`, and measure the time it takes to complete the procedure.

To collect the experiment data, we run the following command (of course after compiling the respective `.c` files):

```
time ./bilshell <value of N> infile.txt
```

When a third argument specifying a file name is given, *bilshell* runs in the batch mode and executes the commands inside the file. Hence, what we have inside `infile.txt` is:

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

`infile.txt` content and the `time` command are changed accordingly so that we cover the set of values $\{10000, 100000, 1000000\}$ for `M` and $\{1, 16, 256, 512, 2048, 4096\}$ for `N`.

Data and Interpretation

Note that the `time` command displays three different timing statistics, the only significant statistic for us is the value next to the `real` label. This value is indeed the total time elapsed from the moment you press enter until the moment the command returns.

The following table contains the execution times for different values of `N` and `M`:

N\M	10000	100000	1000000
1	0.041s	0.335s	3.883s
16	0.036s	0.193s	1.761s
256	0.025s	0.180s	1.736s
512	0.023s	0.181s	1.709s
2048	0.024s	0.188s	1.712s
4096	0.024s	0.180s	1.733s

Table 1: `N`, `M` and the execution time in seconds

As it is obvious from the table, the increase in `M` makes the execution times longer. This result agrees on our expectations; since the *producer* writes and the *consumer* reads more bytes, they run longer.

More of an interesting result is that the increase in `N` does not always make the execution times longer. In fact, for any value of `M` we see that the execution times remain more or less the same after $\texttt{N} \geq 256$. This is possibly because of the trade of between the time it takes to address a certain number of system call and the time it takes to allocate and fill a buffer of size `N`.

Because of the hypothesis that states the number of system calls affecting the execution time, the number of `read` and `write` calls for `M` = 100000 are provided in the table below:

`N` (`M` = 100000)	# `reads`	# `writes`
1	100001	100000
16	32111	32110
256	32556	32555
512	31228	31227
2048	26070	26069
4096	31197	31196

Table 2: `N`, and the number of `read` and `write` calls for `M` = 100000

Looking at this table, we see that the numbers of `read` and `write` calls do not change much after `N` $\geq$ 16. Consequently, the execution times do not change much. This behavior is not intuitive, however it is not unusual. As it is stated in the `read` command's documentation in *Linux Programmer's Manual* [2], it can be the case where `read` gets less bytes than the number of bytes requested (`N` in our case), actually this is very often the situation when *pipe*s are used.

Code for the *producer* and the *consumer*

The codes for `C` codes for the *producer* and *consumer* programs are given below for the curious reader, the codes are relatively simple and self-explanatory.

`producer.c` follows:

```
1  /**
2   * A simple program to print M random alphanumeric characters to screen one by one.
3   * @author Efe Acer
4   * @version 1.0
5   */
6
7  // Necessary imports
8  #include <unistd.h>
9  #include <stdlib.h>
10 #include <time.h>
11
12 // Definitions
13 #define STDOUT_FD 1
14
15 // Contants
```

```
const int ALPHANUMERIC_SIZE = 36;
const char ALPHANUMERIC_CHARS[ALPHANUMERIC_SIZE] = { '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'};

// Global Variables
int M = 1000;

// Main function
int main(int argc, char* argv[]) {
    if (argc > 1) {
        M = atoi(argv[1]); // get the value of M (will be used in experiments)
    }
    srand(time(NULL)); // set random seed
    int bytesWritten = 0;
    while (bytesWritten < M) {
        char randomChar = ALPHANUMERIC_CHARS[rand() % ALPHANUMERIC_SIZE];
        bytesWritten += write(STDOUT_FD, &randomChar, 1);
    }
    return 0;
}
```

`consumer.c` follows:

```
/**
 * A simple program to read M characters one by one.
 * @author Efe Acer
 * @version 1.0
 */

// Necessary imports
#include <unistd.h>
#include <stdlib.h>

// Definitions
#define STDIN_FD 0

// Global Variables
int M = 10;

// Main function
int main(int argc, char* argv[]) {
    if (argc > 1) {
        M = atoi(argv[1]); // get the value of M (will be used in experiments)
    }
    int bytesRead = 0;
    while (bytesRead < M) {
        char toRead;
        bytesRead += read(STDIN_FD, &toRead , 1);
```

```
26      }
27      return 0;
28  }
```

Conclusion

As a result of the experiment, we can confidently conclude that the total number of transferred bytes, `M` increases together with the execution time. However, the number of bytes transferred through the pipes per unit time, `N`, does not have an effect on the execution time that can be modelled using a simple linear function. The way *kernel* manages the pipes is a very important parameter that we should insert to our model, because of *kernel*'s unclear management procedure it is hard to comment on the effect of `N` on the execution time. Although, we can say that a too small value of `N` forces the program to address many `read` and `write` calls, hence makes the execution times longer than usual.

Overall, writing a *Unix shell* like program was really fun and challenging. The submitted file `bilshell.c` is well documented and explains the logical flow of the development process.

References

[1] Silbertschats, Galvin, and Gagne, *Operating System Concepts*, 9th ed. Wiley, 2014, p. 122.

[2] [Accessed: 29- Feb- 2019]. [Online]. Available: http://man7.org/linux/man-pages/man2/read.2.html