Report of Project1

Yiheng Bing

April 1, 2024

1 Copy

1.1 Description

Copy file from one to another via three different methods: direct read and write, via a fork and via a pipe.

1.2 usage

```
./MyCopy <source> <destination>
./ForkCopy <source> <destination>
./PipeCopy <source> <destination>
```

In this project, the source file is *Tobecopied.txt* and the destination files are *dest (1, 2, and 3).txt* according to the copy method.

1.3 Time analysis

The time analysis is as follows(I get the time by the output of the programs):

Method	Time
Direct	0.471000s
Fork	0.478000s
Pipe	0.070000s

Table 1: Time analysis of three copy methods

We can easily find that copy with a pipe is greatly faster than the other two methods. The reason is that the pipe method can read and write simultaneously, which can save a lot of time.

2 Shell

2.1 Description

A simple shell-like server supporting multi-clients that can execute commands with arguments and commands connected by pipes.

2.2 usage

```
./shell <port>
```

2.3 Special commands

The way I implement the pipe commands is worth mentioning.

Because I have seperated the command and arguments by space, I have to ensure the pipe symbol | is surrounded by two spaces. So when I processing the command I've read, I would add spaces before and behind every |. Then I can count the index of the first appearance of | and execute the command before it. And then I would redirect the output of the first command tas the input of the command left by a recursion. Then I can implement a command with several pipes.

3 Mergesort

3.1 Description

Implement the mergesort algorithm in both serial and parallel ways.

The program will read data from a file named *data.in*, which contains the number of the data and the data, and write the sorted data to a file named *data.out* in the same format.

3.2 usage

```
./MergesortSingle
./MergesortMulti
```

3.3 Time analysis

In the *data.in* file, I have 2000000 random numbers. Due to the great amount of numbers, too many redundant pthreads are create and cause great waste of time. So I set a parameter$k$ which represents the limit that I will use serial sorting method. If the size of data is larger than the limit, I'll choose the parallel method. The time analysis is as follows(I get the system time by the command *time ./MergesortSingle* and *time ./MergesortMulti*:

Method	System Time
Serial	0.032s
Parallel ($k = 1000$)	0.043s
Parallel ($k = 10000$)	0.033s
Parallel ($k = 100000$)	0.022s

Table 2: Time analysis of two mergesort methods

We can find that the parallel method is faster than the serial method when the size of data is large enough. And the time of the parallel method is decreasing as the limit k increases because of the overhead of creating threads.