CS 342 - OPERATING SYSTEMS

PROJECT 1

ABDULLAH CAN ALPAY

21702686 - SECTION 3

27.02.2021

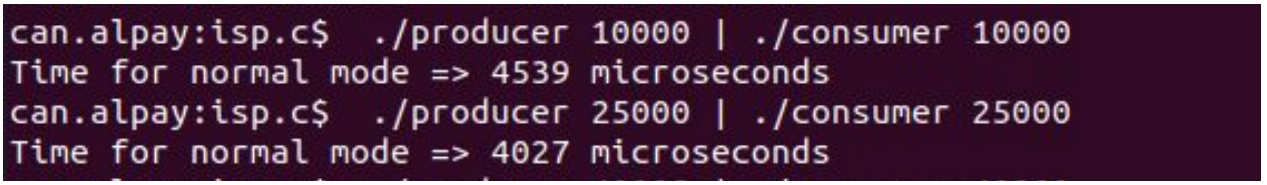
The report consists of the execution outputs, their graphs, and the conclusion. Note that a.out is the executive file derived from isp.c file. For variety, the following M values are used: 1, 100, 1000, 5000, 10000, 25000, 40000, 60000, 80000 and 99999.

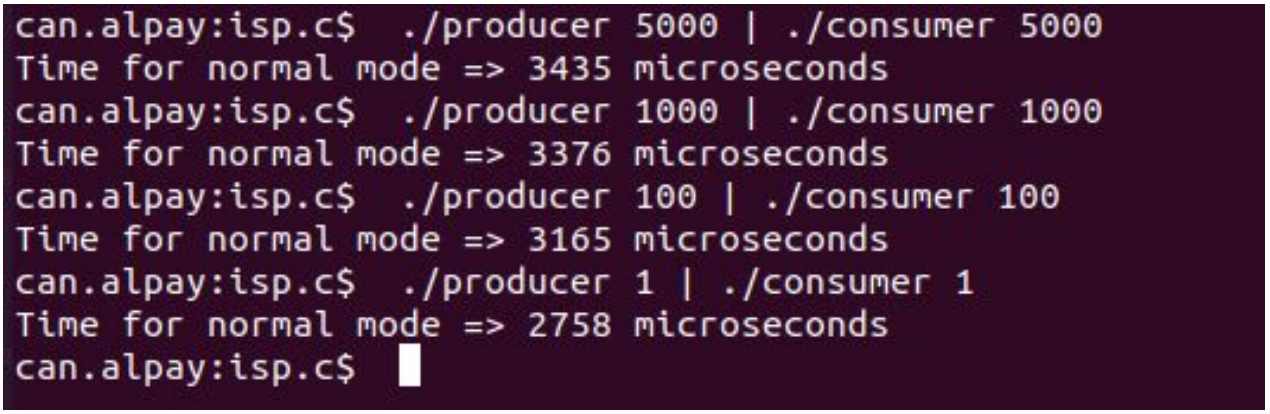
In Normal mode, the buffer size is not essential, but in the tapped mode, the used buffer sizes are:
1, 50, 1000, 2000 and 4000.

In Tapped mode, while loop condition uses read call. Therefore, while the read can read from the pipe, it will call write. However, when the pipe becomes empty, the read will check the pipe one more time and return 0. When the read returns 0, the while statement terminates the loop. Thus, when the while loop calls the read, and the read returned 0, the loop will not call the write. With that, the read is always called one more than the write.

Additionally, from time to time, the execution time can be different from the expected result. Hence, some executions run several times to meet the expected execution time. To roughly describe the expected time, the more time is required, as for bigger M value (producer and consumer's character count).

For normal mode, the output is:





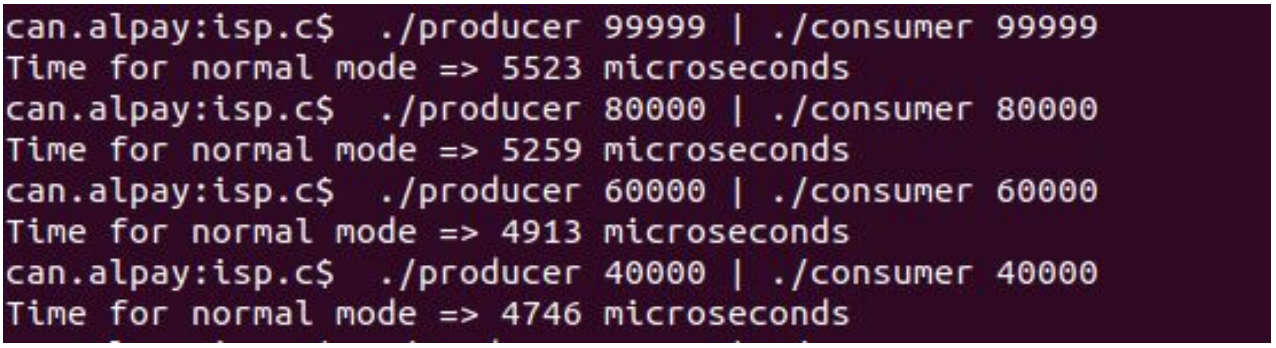


Figure - 1

For tapped mode outputs are:

When buffer size is 1, the output is:

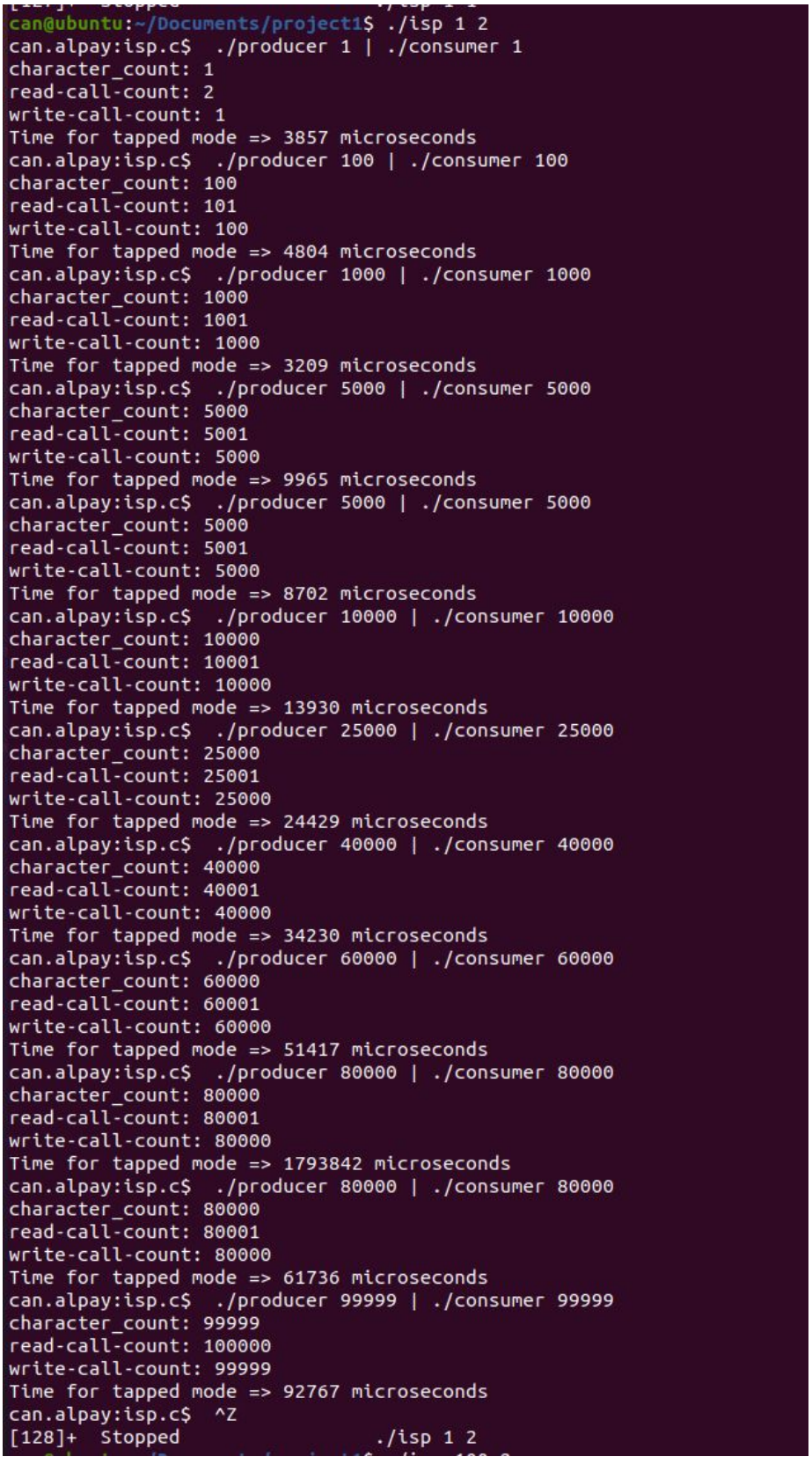


Figure - 2

When buffer size is 50, the output is:

```
can@ubuntu:~/Documents/project1$ ./isp 50 2
can.alpay:isp.c$  ./producer 1 | ./consumer 1
character_count: 1
read-call-count: 2
write-call-count: 1
Time for tapped mode => 3917 microseconds
can.alpay:isp.c$  ./producer 100 | ./consumer 100
character_count: 100
read-call-count: 3
write-call-count: 2
Time for tapped mode => 3966 microseconds
can.alpay:isp.c$  ./producer 1000 | ./consumer 1000
character_count: 1000
read-call-count: 21
write-call-count: 20
Time for tapped mode => 4915 microseconds
can.alpay:isp.c$  ./producer 5000 | ./consumer 5000
character_count: 5000
read-call-count: 101
write-call-count: 100
Time for tapped mode => 4895 microseconds
can.alpay:isp.c$  ./producer 10000 | ./consumer 10000
character_count: 10000
read-call-count: 201
write-call-count: 200
Time for tapped mode => 5066 microseconds
can.alpay:isp.c$  ./producer 25000 | ./consumer 25000
character_count: 25000
read-call-count: 501
write-call-count: 500
Time for tapped mode => 4435 microseconds
can.alpay:isp.c$  ./producer 25000 | ./consumer 25000
character_count: 25000
read-call-count: 501
write-call-count: 500
Time for tapped mode => 4357 microseconds
can.alpay:isp.c$  ./producer 40000 | ./consumer 40000
character_count: 40000
read-call-count: 801
write-call-count: 800
Time for tapped mode => 5949 microseconds
can.alpay:isp.c$  ./producer 60000 | ./consumer 60000
character_count: 60000
read-call-count: 1202
write-call-count: 1201
Time for tapped mode => 5946 microseconds
can.alpay:isp.c$  ./producer 80000 | ./consumer 80000
character_count: 80000
read-call-count: 1603
write-call-count: 1602
Time for tapped mode => 18746 microseconds
can.alpay:isp.c$  ./producer 60000 | ./consumer 60000
character_count: 60000
read-call-count: 1202
write-call-count: 1201
Time for tapped mode => 5669 microseconds
can.alpay:isp.c$  ./producer 99999 | ./consumer 99999
character_count: 99999
read-call-count: 2002
write-call-count: 2001
Time for tapped mode => 10679 microseconds
can.alpay:isp.c$  ./producer 80000 | ./consumer 80000
character_count: 80000
read-call-count: 1601
write-call-count: 1600
Time for tapped mode => 8289 microseconds
can.alpay:isp.c$
```

Figure - 3

When buffer size is 1000, the output is:

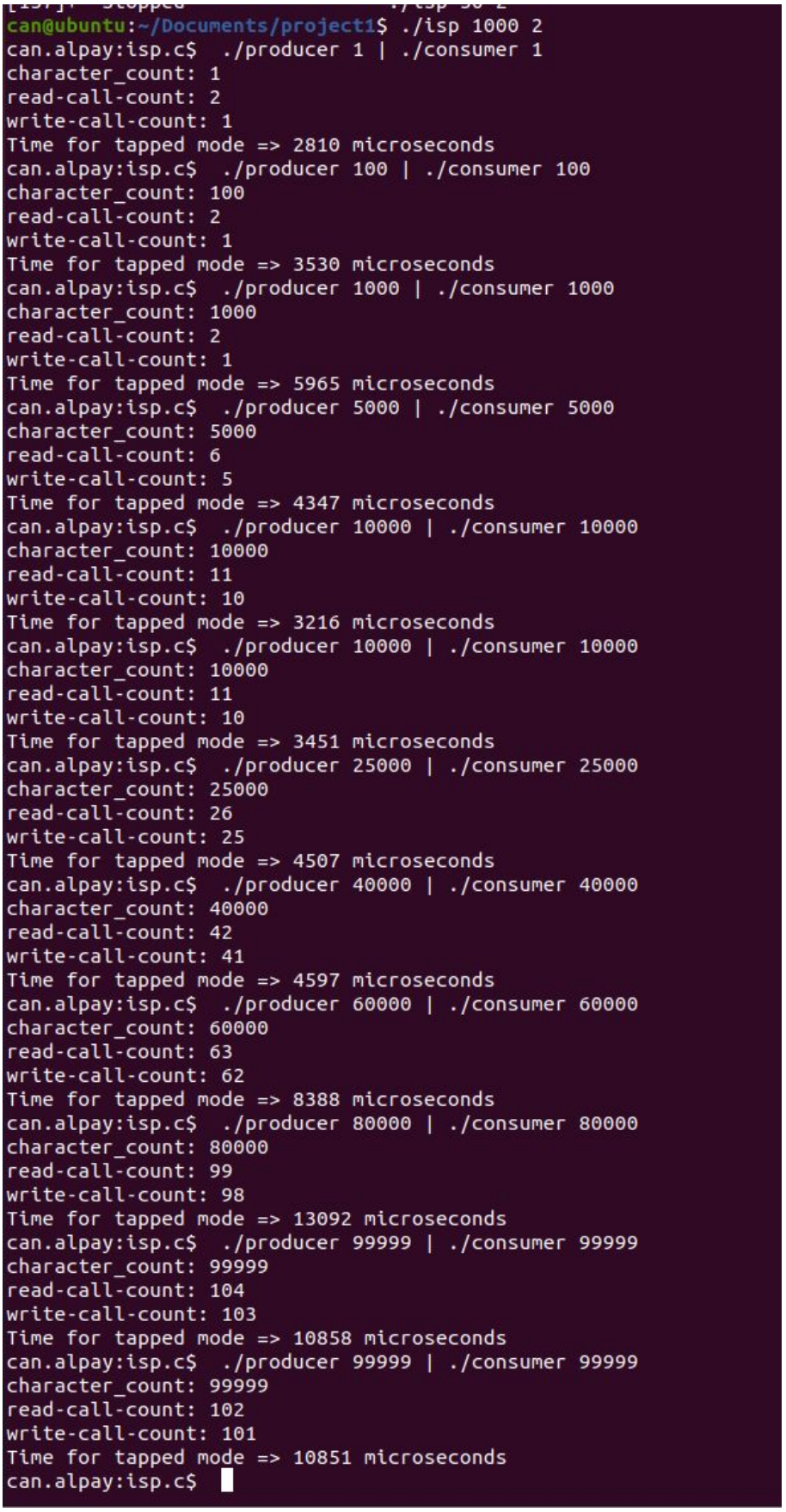


Figure - 4

When buffer size is 2000, the output is:

```
can@ubuntu:~/Documents/project1$ ./isp 2000 2
can.alpay:isp.c$  ./producer 1 | ./consumer 1
character_count: 1
read-call-count: 2
write-call-count: 1
Time for tapped mode => 3339 microseconds
can.alpay:isp.c$  ./producer 100 | ./consumer 100
character_count: 100
read-call-count: 2
write-call-count: 1
Time for tapped mode => 2665 microseconds
can.alpay:isp.c$  ./producer 1000 | ./consumer 1000
character_count: 1000
read-call-count: 2
write-call-count: 1
Time for tapped mode => 3301 microseconds
can.alpay:isp.c$  ./producer 5000 | ./consumer 5000
character_count: 5000
read-call-count: 4
write-call-count: 3
Time for tapped mode => 4891 microseconds
can.alpay:isp.c$  ./producer 10000 | ./consumer 10000
character_count: 10000
read-call-count: 6
write-call-count: 5
Time for tapped mode => 3070 microseconds
can.alpay:isp.c$  ./producer 25000 | ./consumer 25000
character_count: 25000
read-call-count: 14
write-call-count: 13
Time for tapped mode => 5752 microseconds
can.alpay:isp.c$  ./producer 40000 | ./consumer 40000
character_count: 40000
read-call-count: 22
write-call-count: 21
Time for tapped mode => 4708 microseconds
can.alpay:isp.c$  ./producer 60000 | ./consumer 60000
character_count: 60000
read-call-count: 31
write-call-count: 30
Time for tapped mode => 9563 microseconds
can.alpay:isp.c$  ./producer 80000 | ./consumer 80000
character_count: 80000
read-call-count: 45
write-call-count: 44
Time for tapped mode => 9601 microseconds
can.alpay:isp.c$  ./producer 99999 | ./consumer 99999
character_count: 99999
read-call-count: 54
write-call-count: 53
Time for tapped mode => 10219 microseconds
can.alpay:isp.c$  ./producer 60000 | ./consumer 60000
character_count: 60000
read-call-count: 34
write-call-count: 33
Time for tapped mode => 7979 microseconds
can.alpay:isp.c$
```

Figure - 5

When buffer size is 4000, the output is:

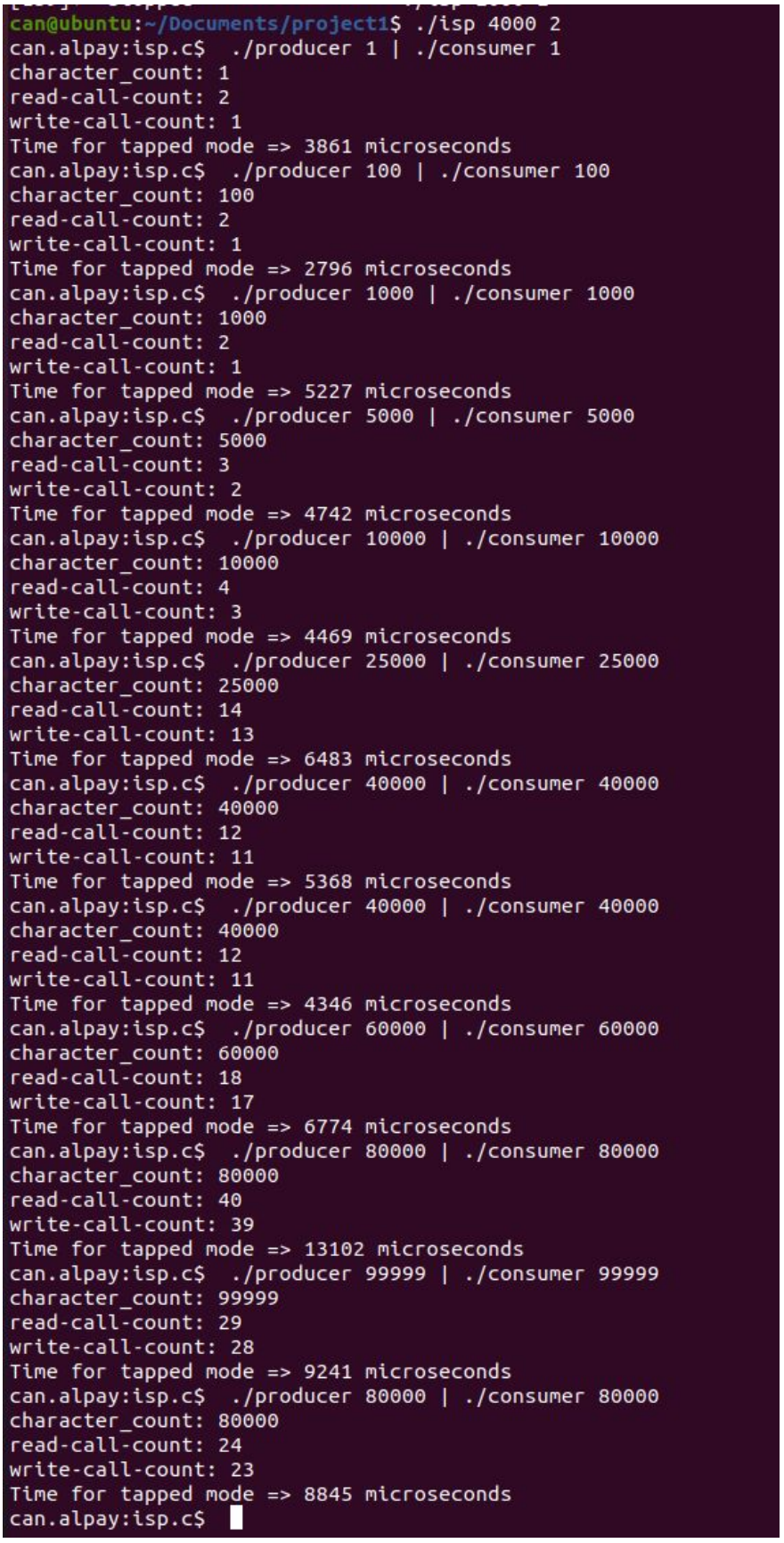


Figure - 6

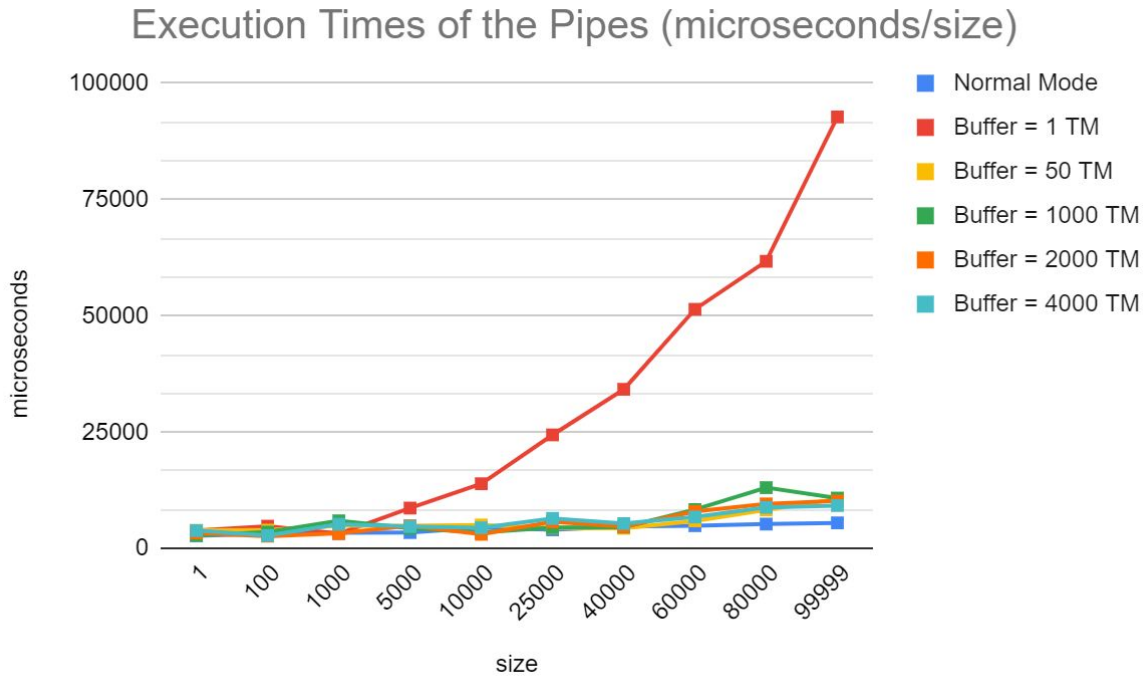


Figure - 7

The graph illustrates that most of the execution is done under 10.000 microseconds and close to each other. However, when we look at the Buffer size = 1 case, we can see that the execution time increases drastically. Thus, Buffer size = 1 is the worst case for choosing to read/write buffer.

Followingly, the comparison between the Normal mode and the Tapped mode with different buffer sizes is shown.

Compare Normal Mode with Buffer = 2000 and Buffer = 4000

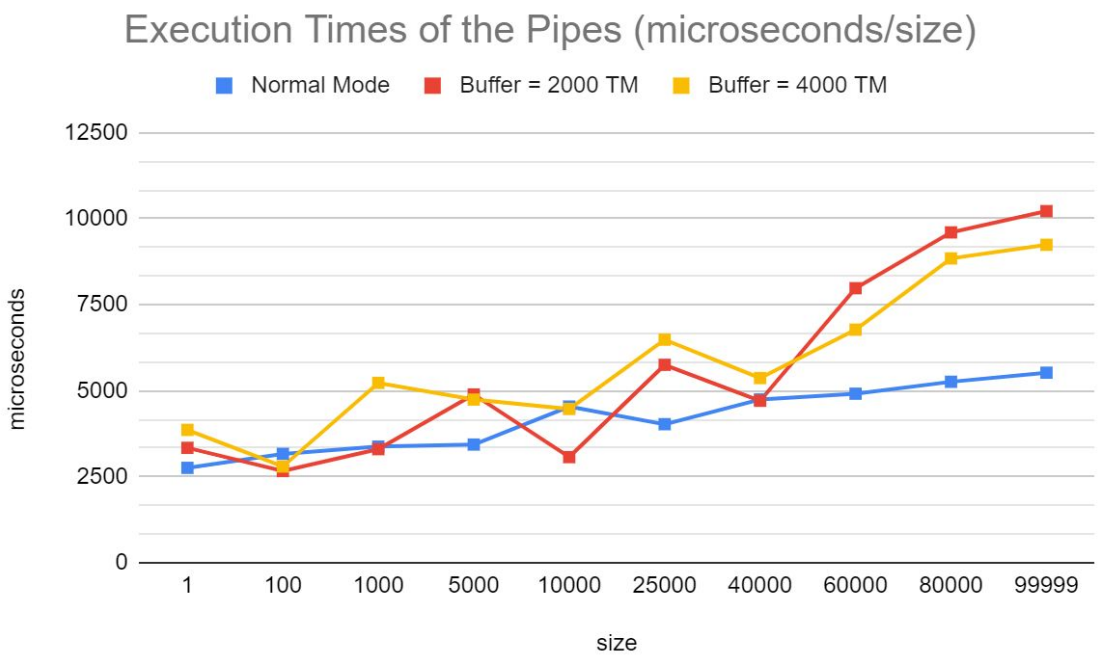


Figure - 8

When we look at the graph, normal mode and large buffer sizes perform similarly, with a slightly better performance achieved by Normal mode for a small number of characters. However, when the number of characters increased, the Tapped mode's performances decrease sharply.

Compare Normal Mode with Buffer = 50 and Buffer = 1000

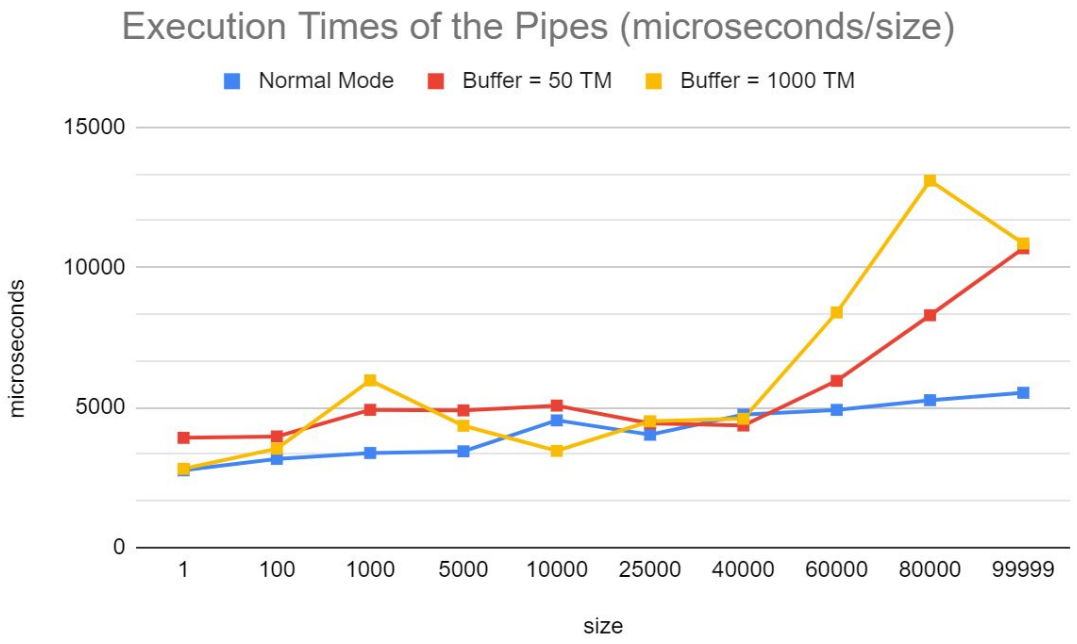


Figure - 9

In Figure - 9, we can see that the Normal mode performs better than the Tapped modes.

In conclusion, when we look at Figure - 8 and Figure - 9, the Normal mode performs better than the Tapped mode, regardless of buffer size. When evaluating the code, the Normal mode creates one pipe that transfers information from child1 to child2. However, in the Tapped mode, the code creates two pipes between child1 to parent and parent to child2. Additionally, with the buffer size, we limit the transaction speed from pipe1 to pipe2, which results in low performance. The Normal mode executes two-child creation, one pipe creation, and concurrent unlimited transaction between the children. The Tapped mode code executes two-child creation, two pipe creation, writing a file to the pipe1, limited reading from the pipe1, limited writing to the pipe2 and reading a file from the pipe2. The Normal mode has fewer tasks. Thus, the Normal mode performs better than the Tapped mode.

Additionally, when we look at the buffer sizes, the smaller buffer size decreases the performances. Thus, when the buffer size increases, the performance increases. When measuring performances, some unexpected results occurred, which are the time jumps in the graphs. These jumps of the execution time occur because of the processing speed and memory allocations. From time to time, computers can spend more time allocating memory or forking a child. Thus, the same executions can require different times to execute.