Mert Alp Taytak – 21602061

REPORT

Results

Following figures show the results of my experiments.

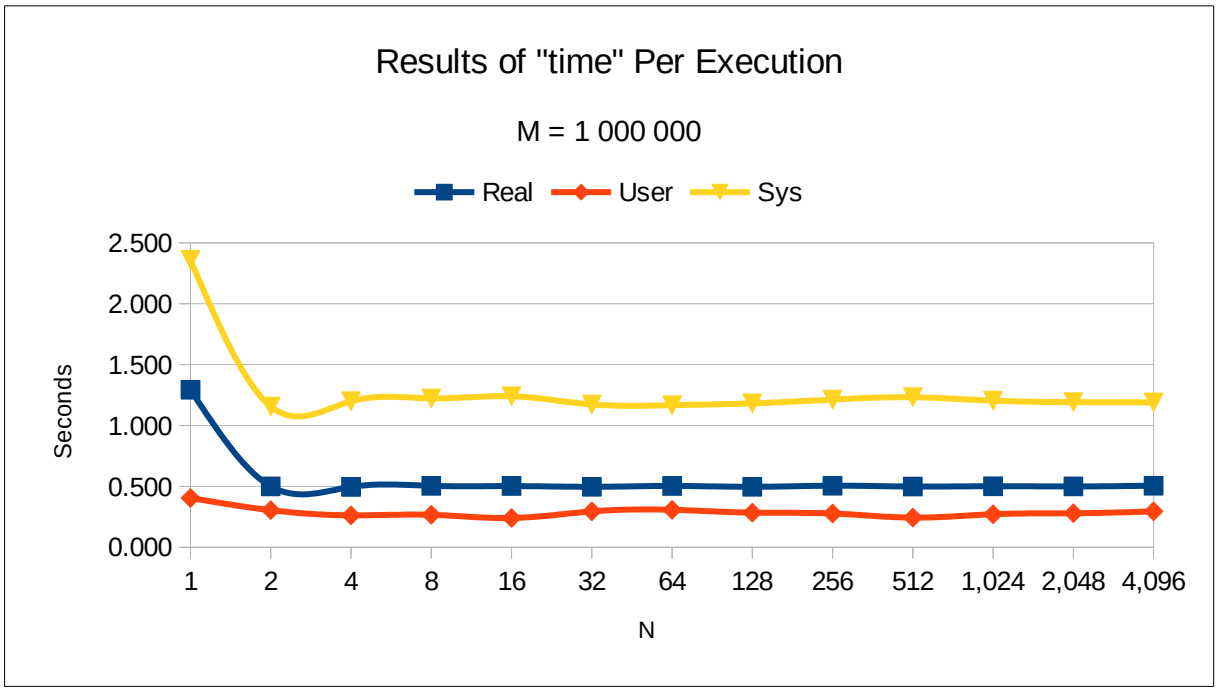

Fig. 1. Timing results from first experiment

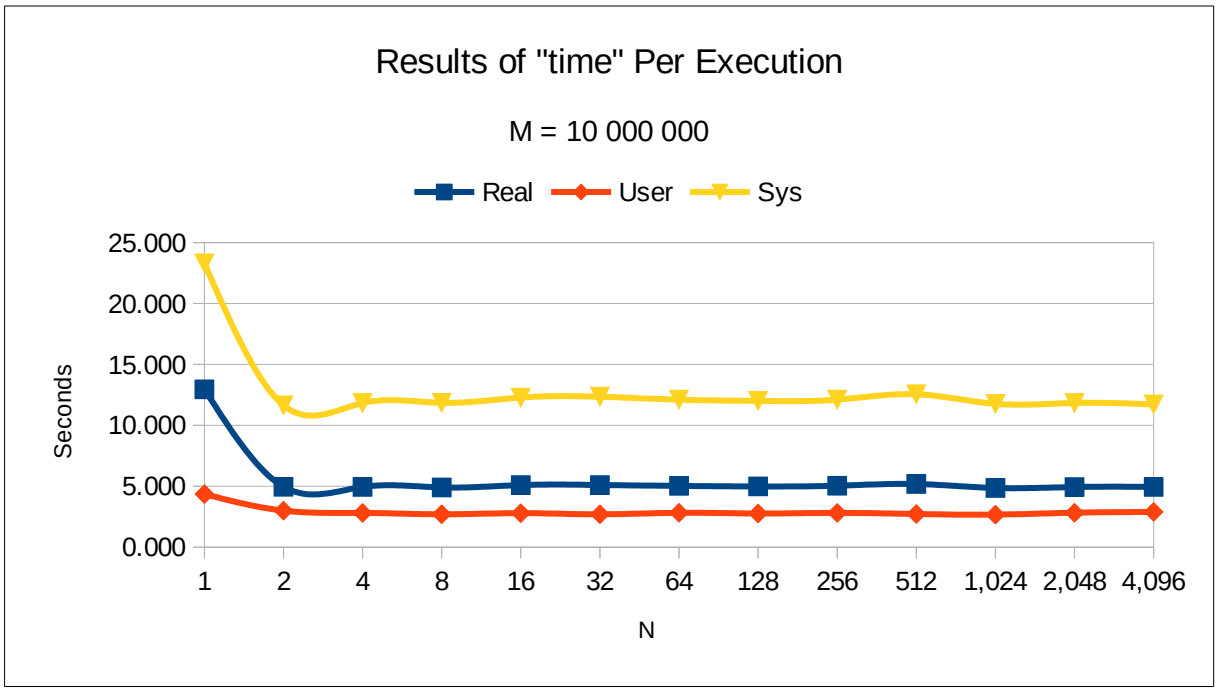

Fig. 2. Timing results from second experiment

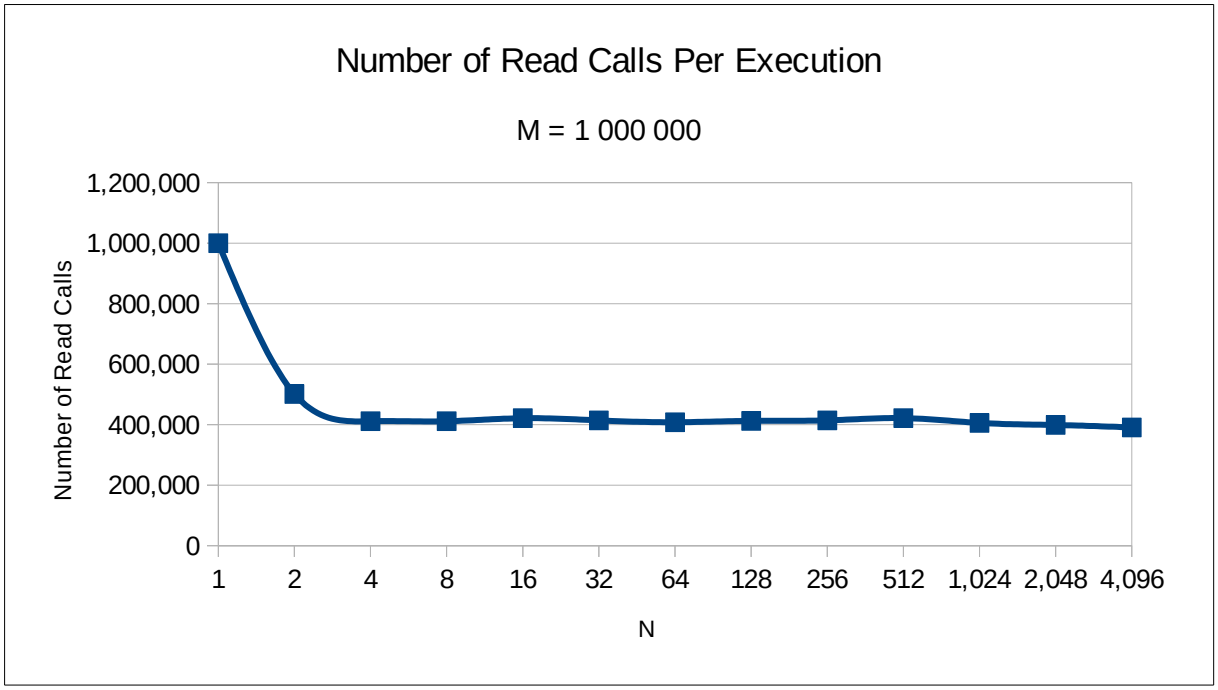

Fig. 3. Read call data from first experiment

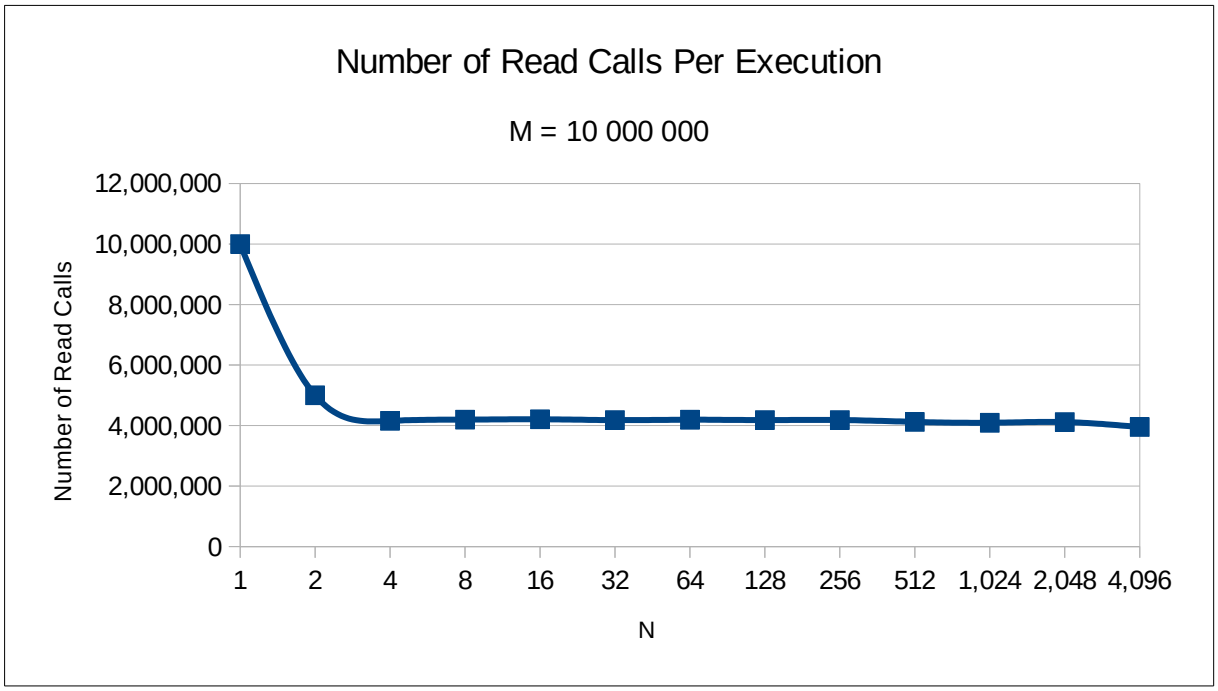

Fig. 4. Read call data from second experiment

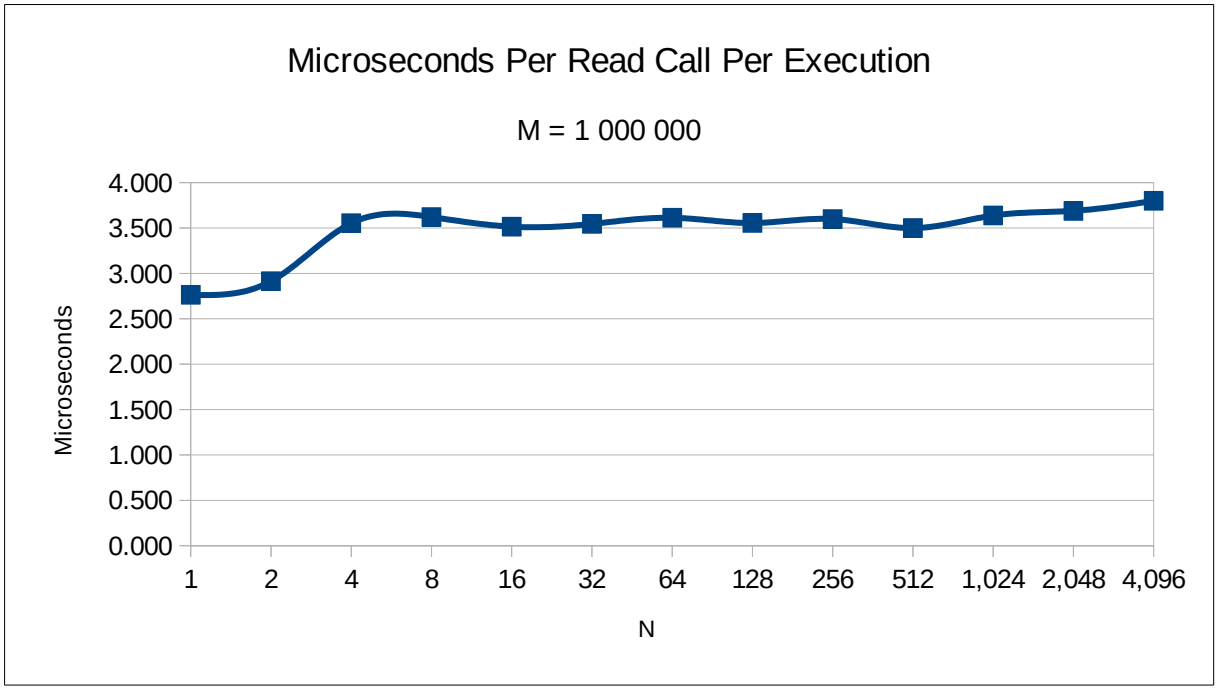

Fig. 5. Time per call data from first experiment

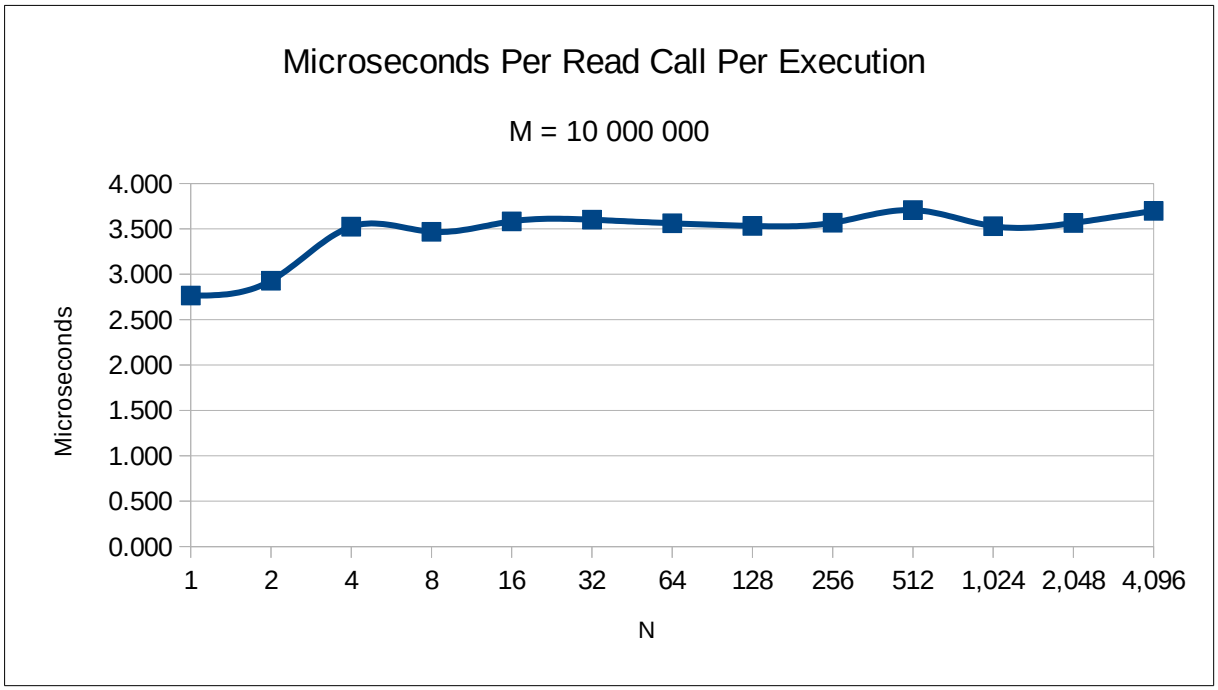

Fig. 6. Time per call data from second experiment

Discussion

First of all, notice that in figures 1 and 2 the value for *real* execution times is less than the corresponding *user* and *sys* values. That is because the program is a multiprocess application and *time* command does not take child processes into account properly. Which results in such a situation.

Next, figures 1 and 2 show us that after a certain value, increasing N does not make a noticeable difference on execution speed. Considering the overhead that goes into a single read-write cycle, this is a surprising result. However, following figures help to show the reason of this.

Looking at figures 3 and 4, we see that after a certain value of N, number of read calls evens out to some particular value. Following with figures 5 and 6, we see that for all values of N the time it takes do a single read-write cycle is approximately the same. Hence, for each N with similar number of read calls we have similar execution times. Notice that the duration of a read-write cycle does not differ for different values of *M*. However, there is a linear relation between *M* and number of calls needed per execution. Resulting in the same linear relation for execution times in figures 1 and 2 for experiments with different *M*.

We now have the mechanism behind the surprising results from figures 1 and 2. However, this mechanism does not explain why increasing *N* and thus decreasing the ratio of overhead to useful work affect execution time. My proposed thesis for the reason is that periodic interrupts and scheduling done by the kernel prevents the program from utilizing increased buffer size. In the producer and consumer programs we read or write *M* bytes at a fixed size that is less than *M*. Each read or write is a system call to the kernel, causing a context switch and thus an interrupt. Therefore, there are millions of system calls made per execution. My producer and consumer programs processed bytes one by one. Hence, increasing the buffer size, *N*, did not have much effect after *N = 2*.

Appendixes

Producer Program

```c
/*
   CS342 - Project 1
   Author: Mert Alp Taytak
   ID: 21602061
*/

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

/*
   MACROS, DEFINITIONS and GLOBALS
*/

long M;

/*
   FUNCTION PROTOTYPES
*/

int process_m(const char *str);

int main(int argc, char **argv)
{
   if (argc != 2) {
      return -1;
   }

   process_m(argv[1]);

   char *buff = (char*)calloc(M, sizeof(char));
   for (long i = 0; i < M; i++) {
      buff[i] = 97 + (random() % 26); /* Random [a-z] */
      write(STDOUT_FILENO, &buff[i], 1);
   }

   return 0;
}

/*
   Reads argument for M
```

```
   Makes sure it is valid
*/
int process_m(const char* str)
{
   if (*str == '\0') {
      return -1;
   }

   char *endptr;
   long int res = strtol(str, &endptr, 10);

   if (*endptr == '\0') { /* Success */
      M = res;
      return 0;
   }
   return -1;
}
```

Consumer Program

```
/*
   CS342 - Project 1
   Author: Mert Alp Taytak
   ID: 21602061
*/

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

/*
   MACROS, DEFINITIONS and GLOBALS
*/

long M;

/*
   FUNCTION PROTOTYPES
*/

int process_m(const char *str);

int main(int argc, char **argv)
{
   if (argc != 2) {
```

```
        return -1;
    }

    process_m(argv[1]);

    char *buff = (char*)calloc(M, sizeof(char));
    for (long i = 0; i < M; i++) {
        read(STDIN_FILENO, &buff[i], 1);
    }

    return 0;
}

/*
    Reads argument for M
    Makes sure it is valid
*/
int process_m(const char* str)
{
    if (*str == '\0') {
        return -1;
    }

    char *endptr;
    long int res = strtol(str, &endptr, 10);

    if (*endptr == '\0') { /* Success */
        M = res;
        return 0;
    }
    return -1;
}
```