**OPERATING SYSTEM PROJECT REPORT**

**TOPIC: Comparing performance for Interprocess communication using Filing, Pipes(both simple and named), and Semaphores.**

**GROUP MEMBERS:**Muhammad Ahmed (k15-2848)  
Bilal Salman (k15-2114)  
Bilal Shafique (k15-2152)

**Description:**

The main purpose of our project is to implement communication between processes using filing, pipes and semaphores. Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Typically, applications using IPC are categorized as clients and servers, where the client requests data and the server responds to client requests Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods. In our project we have used the following methods to implement IPC.

**1. Shared memory segments:**

Shared memory is the fastest interprocess communication mechanism. The operating system maps a memory segment in the address space of several processes, so that several processes can read and write in that memory segment without calling operating system functions. Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The page of the virtual memory is referenced by page table entries in each of the sharing processes' page tables. It does not have to be at the same address in all of the processes' virtual memory. It is usually used to create a environment of server and client.

**2. Filing:**

Filing is one way of implementing interprocess communication. Two or more processes can share a file between them and use it for reading and writing. One process can write the contents and data into the file while the other can read it. Or both can read or write depending on the requirement of the system or the user. A shared memory segment will be used in this to create a client and server. One process will act as a server and the other as a client.

**3. Pipes:**

**SIMPLE**

Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process. Neither process is aware of this redirection and behaves just as it would normally. When the writer wants to write to the pipe it uses the standard write library functions. These all pass file descriptors that are indices into the process's set of file data structures, each one representing an open file or, as in this case, an open pipe. The Linux system call uses the write routine pointed at by the file data structure describing this pipe. If there is enough room to write all of the bytes into the pipe and, so long as the pipe is not locked by its reader, Linux locks it for the writer and copies the bytes to be written from the processes address space into the shared data page. If the pipe is locked by the reader or if there is not enough room for the data then the current process is made to sleep on the pipe node's wait queue and the scheduler is called so that another process can run.

**NAMED**

The easiest way to create a FIFO file is to use the mkfifo command. This command is part of the standard Linux utilities and can simply be typed at the command prompt of your shell. This command requires an additional argument but otherwise it works pretty much the same as simple pipes. Since this named pipe looks like a file, you can use all the system calls associated with files to interact with it. In particular, you can use the open, read, write, and close system calls.

**4. Semaphores:**

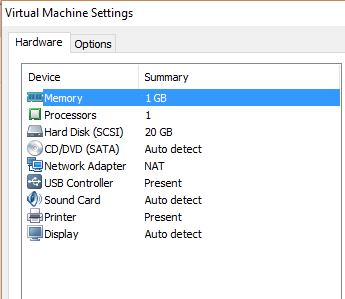
A semaphore is a variable or abstract data type that is used for controlling access, by multiple processes. In its simplest form a semaphore is a location in memory whose value can be tested and set by more than one process.a semaphore is a variable with a value that indicates the status of a common resource. It's used to lock the resource that is being used. A process needing the resource checks the semaphore to determine the resource's status and then decides how to proceed. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values.

**5. Signal Handling:**

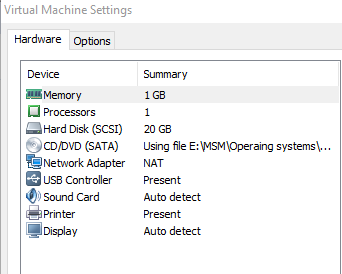
A signal is a software interrupt, a way to communicate information to a process about the state of other processes, the operating system, and the hardware. A signal is an interrupt in the sense that it can change the flow of the program. When a signal is delivered to a process, the process will stop what its doing, either handle or ignore the signal, or in some cases terminate, depending on the signal.

**CONFIGURATION OF VMWARE**

**UBUNTU**



**KALI LINUX**



**Task between the processes:**

We used insertion sort algorithm on 50000 random numbers as a task to implement interprocess communication so that we can learn about the working and handling of large data by the process and to get clear idea of the performance of each type of IPC method so that we can compare them later on and do the analysis.

**How and from where the project started?**

The project was started by first learning about shared memory segments and the methods of communicating between processes. Also learning about parent and child processes and how it works. We first created a basic server and client program so that we can know how to actually start working on the project. After learning about shared memory and the mechanism of communication we started working on filing and generating random numbers.

**Problems faced and their solution:**

As interprocess communication is not a easy task to accomplish we faced several problems which are mentioned below.

**1. Segmentation fault:**

Initially when working with the shared memory segment we get segmentation fault and core dumped problems probably because of exchanging large data between processes and greater work load.

**Solution:**

We have used the library #include<sys/ipc.h> to incease the kernel memory so that we can work on large data.

**2. Waiting time on Random numbers:**

While generating random numbers we had to wait for one second after each number because we used “sleep” command to clear the memory and to generate different number.

**Solution:**

Instead of generating random numbers on run time and waiting we saved all the 50000 numbers in another file and saved it. Now the process which is writing data will read the data one by one from that file and write it on the other file which is our main shared file between the processes.

**3. Problem with semaphores:**

Another problem was that there was an error in declaring the semaphore in shared memory so we could not communicate and signal between processes because processes could not read the declared semaphore and its values hence they could not send commands to each other

**Solution:**

To overcome this problem we did not declare semaphore in shared memory instead we made parent and child processes so that they can signal each other for waiting or to execute.

**Actual Working of the project:**

Our project is divided into three parts which is to implement interprocess communication using filing, pipes and semaphore. The working of each type is described below.

**IPC using filing:**

We have created two programs or you can say two processes. One is used for writing the data into the shared file named as “shared.txt” and the other program reads the data from it and sorts the data using insertion sort. First we run the first process which acts as server on one terminal and it goes on waiting after writing all the data. Then we open a separate terminal to run the other process it reads all the data and sorts it and shows the final contents of file. At the end total time is calculated and displayed.

**Signals:**

As through signals we can just communicate between process by simply sending a signal. We have used it to signal the filewrite.c through fileread.c by using the kill command to let it know that reading and writing is performed.

**IPC using pipes:**

In this type of inter process communication we declared and initialize the pipe data structure and creates parent and child process. After that we close the read end of the pipe and write the data into it. After that we close the write end and read all the data and sort it.At the end total time is calculated and displayed.

In name pipes we did the same but instead of creating pipes we created a file MYFIFO and later on use the commands read and write to sort the data using insertion sort.

**IPC using semaphores:**

In this type of inter process communication we declared and initialize the semaphores and creates parent and child process. When one process finished writing all the data it signals the other process to reads it which in turn sorts the data. At the end total time is calculated and displayed.

**Explanation of source code:**

Source code is explained below

**Filing:**

**FILE WRITE**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include<sys/shm.h>

#include<signal.h>

#include<sys/ipc.h>

#define KEY 9999

voidfunc(intsignalNo)

{

printf("Terminating%d \n",signalNo); //signal handlers handles the signal and terminates it

}

int main()

{

intnum,i,n=20000;

FILE \*ff,\*fp,\*fw;

ff = fopen("numbers.txt","r"); //open the file in which numbers are stored

fp = fopen("shared.txt","w"); // open the shared file and clear all data

fclose(fp); // close shared file

fw = fopen("shared.txt","w");

for(i=0;i<n;i++) // this loop writes the number one by one into shared file

{

fscanf(ff, "%d", &num);

fprintf(fw,"%d ",num);

}

fclose(fw);

fclose(ff);

printf("writing complete, waiting for second process....\n");

signal(SIGUSR1,func);//waiting for the signal of second process

while(1)

{ }

printf("Terminating\n");

}

**FILEREAD**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include<sys/shm.h>

#include<signal.h>

#include<sys/ipc.h>

#define KEY 9999

int main()

{

printf("reading and sorting the data....\n");

intarr[50000],n=10000,i,j,var,temp;

clock\_tst;

FILE \*fp,\*fr;

fp = fopen("shared.txt","r"); // opens the shared file

st = clock(); //time starts

for(i=0;i<n;i++) // loop for reading the file and storing data

{

fscanf(fp, "%d", &arr[i]);

}

fclose(fp); // close the shared file

// insertion sort algorithm starts

for(i=1;i<n;i++)

{

var=arr[i];

temp=i;

for(j=i-1;j>=0;j--)

{

if(arr[j]>var){

arr[temp]=arr[j];

arr[j]=var;

temp--;

}

}}

// insertion sort algorithm ends

fr = fopen("shared.txt","w"); // open the shared file again for writing sorted data

for(i=0;i<n;i++) // loop for writing sorted data into the shared file

{

fprintf(fr,"%d ",arr[i]);

printf("%d \t",arr[i]);

}

fclose(fr); // close the shared file

printf("sorting complete....\n");

clock\_t et = clock();

clock\_tctt = et - st;

double tis = ctt / (double) CLOCKS\_PER\_SEC;

printf("\n\ntotal time is: %lf",tis);

system("pkillfilewrite"); //sends the signal pkill to filewrite.c to terminate

}

**Pipes:**

**SIMPLE PIPES**

#include <stdio.h>

#include <unistd.h>

#include<time.h>

#include <sys/types.h>

int main(void)

{

intfd[2], nbytes;

pid\_tchildpid;

intarr[50000],sort[50000],i,var,temp,j,n=30000;

clock\_tst;

FILE \*ff;

pipe(fd); // declares pipe structure

if((childpid = fork()) == -1)

{

perror("fork");

}

if(childpid == 0)

{

// Child process closes up input side of pipe

close(fd[0]);

ff = fopen("numbers.txt","r"); //open the file in which numbers are stored

for(i=0;i<n;i++){

fscanf(ff, "%d", &arr[i]);

}

fclose(ff); // close the file

/\* Send "array" through the output side of pipe \*/

write(fd[1], &arr, sizeof(arr));

}

else

{

st = clock(); //time starts

printf("reading and sorting the data....\n");

/\* Parent process closes up output side of pipe \*/

close(fd[1]);

/\* Read in a array from the pipe \*/

nbytes = read(fd[0], &sort, sizeof(sort));

// insertion sort algorithm starts

for(i=1;i<n;i++)

{

var=sort[i];

temp=i;

for(j=i-1;j>=0;j--)

{

if(sort[j]>var){

sort[temp]=sort[j];

sort[j]=var;

temp--;

}}}

// insertion sort algorithm ends

for(i=0;i<n;i++) // prints final sorted data

printf("%d \t",sort[i]);

printf("sorting complete....\n");

clock\_t et = clock();

clock\_tctt = et - st;

double tis = ctt / (double) CLOCKS\_PER\_SEC;

printf("\n\ntotal time is: %lf",tis);

}

return(0);

}

**BY MKFIFO COMMAND**

**WRITE**

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <sys/shm.h>

#define n 50000

int main()

{

intarr[50000];

intfd;

char \* myfifo = "/tmp/myfifo"; //file path

mkfifo(myfifo, 0666); //ceates the file

printf("\nWaiting for the other process\n");

FILE \*ff;

ff = fopen("numbers.txt","r"); //open the file in which numbers are stored

for (int c = 0 ; c < n ; c++ )

{

intnum;

fscanf(ff, "%d",&num);

arr[c]=num;

}

fd = open(myfifo, O\_WRONLY);

write(fd,arr,sizeof(arr)); //writes the data

close(fd); //closes pipe

unlink(myfifo); //unlinks itself from the pipe

return 0;

}

**READ**

#include <fcntl.h>

#include <stdio.h>

#include <sys/stat.h>

#include <unistd.h>

#include<time.h>

#define n 50000

int main()

{

int pipe; //contains the address of pipe

intarr[50000];

clock\_tst = clock(); //gets current time

// open a named pipe

pipe = open("/tmp/myfifo", O\_RDONLY);

// actually read out the data and close the pipe

read(pipe,arr,sizeof(arr));

// insertion sort algorithm starts

for(int i=1;i<n;i++)

{

intvar=arr[i];

int temp=i;

for(int j=i-1;j>=0;j--)

{

if(arr[j]>var)

{

arr[temp]=arr[j];

arr[j]=var;

temp--;

}}}

// insertion sort algorithm ends

for (int c = 0 ; c < n ; c++ )

printf("%d \t",arr[c]);

clock\_t et = clock();

clock\_tctt = et - st;

double tis = ctt / (double) CLOCKS\_PER\_SEC; //converts tme in to seconds

//writing the sorted data in shared.txt file

FILE \*fp;

fp = fopen("shared.txt","w");

for(int c = 0 ; c < n ; c++)

fprintf(fp,"%d ", arr[c]);

printf("\n\ntotal time is: %lf\n\n",tis);

// close the pipe

close(pipe);

return 0;

}

**Semaphores:**

#include <unistd.h>

#include <sys/types.h>

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <semaphore.h>

#define KEY 1000

sem\_tsema;

int n=50000;

int main()

{

sem\_init(&sema,0,1);

intshm\_id;

shm\_id = shmget(KEY, 50000\*4 , IPC\_CREAT | 0666);

if (shm\_id< 0)

{

printf("shmget error\n");

}

pid\_t process=fork();

if( process == 0 ) //Child Process reads the data from shared memory and sorts the data using insertion sort

{

if(sem\_wait(&sema) == 0) //if no process is in the region then sort it

{

int \*shm\_ptr,i,j,var,temp,c;

intarr[50000];

shm\_ptr= shmat(shm\_id, NULL, 0);

clock\_tst = clock();

for ( c = 0 ; c < n ; c++ )

{

arr[c]=\*shm\_ptr; //reading the data from shared memory

shm\_ptr++;

}

// insertion sort algorithm starts

for(i=1;i<n;i++)

{

var=arr[i];

temp=i;

for(j=i-1;j>=0;j--)

{

if(arr[j]>var)

{

arr[temp]=arr[j];

arr[j]=var;

temp--;

}}}

// insertion sort algorithm ends

for( c = 0 ; c < n ; c++ )

printf("%d \t",arr[c]);

clock\_t et = clock();

clock\_tctt = et - st;

double tis = ctt / (double) CLOCKS\_PER\_SEC;

//writing the sorted data in shared.txt file

FILE \*fp;

fp = fopen("shared.txt","w");

for( c = 0 ; c < n ; c++)

fprintf(fp,"%d ", arr[c]);

printf("\n\ntotal time is: %lf\n\n",tis);

}

sem\_post(&sema);

}

else //parent prrocess reads the data from file and writes it in to shared memory

{

if(sem\_wait(&sema) == 0)

{

int \*shm\_ptr,i,c,num;

shm\_ptr= shmat(shm\_id, NULL, 0);

FILE \*ff;

ff = fopen("numbers.txt","r"); //open the file in which numbers are stored

for ( c = 0 ; c < n ; c++ )

{

fscanf(ff, "%d", &num);

\*shm\_ptr = num; //adding the data in shared memory

\*shm\_ptr++;

}

}

sem\_post(&sema);

}}

**COMPARING PERFORMANCES:**

The charts and the graphs below shows time in seconds for different data sets and types of interprocess communication. We can see the difference between them. We test the data sets on separate Linux operating system (Ubuntu) and (KALI ) on vmware.

**KALI LINUX:**

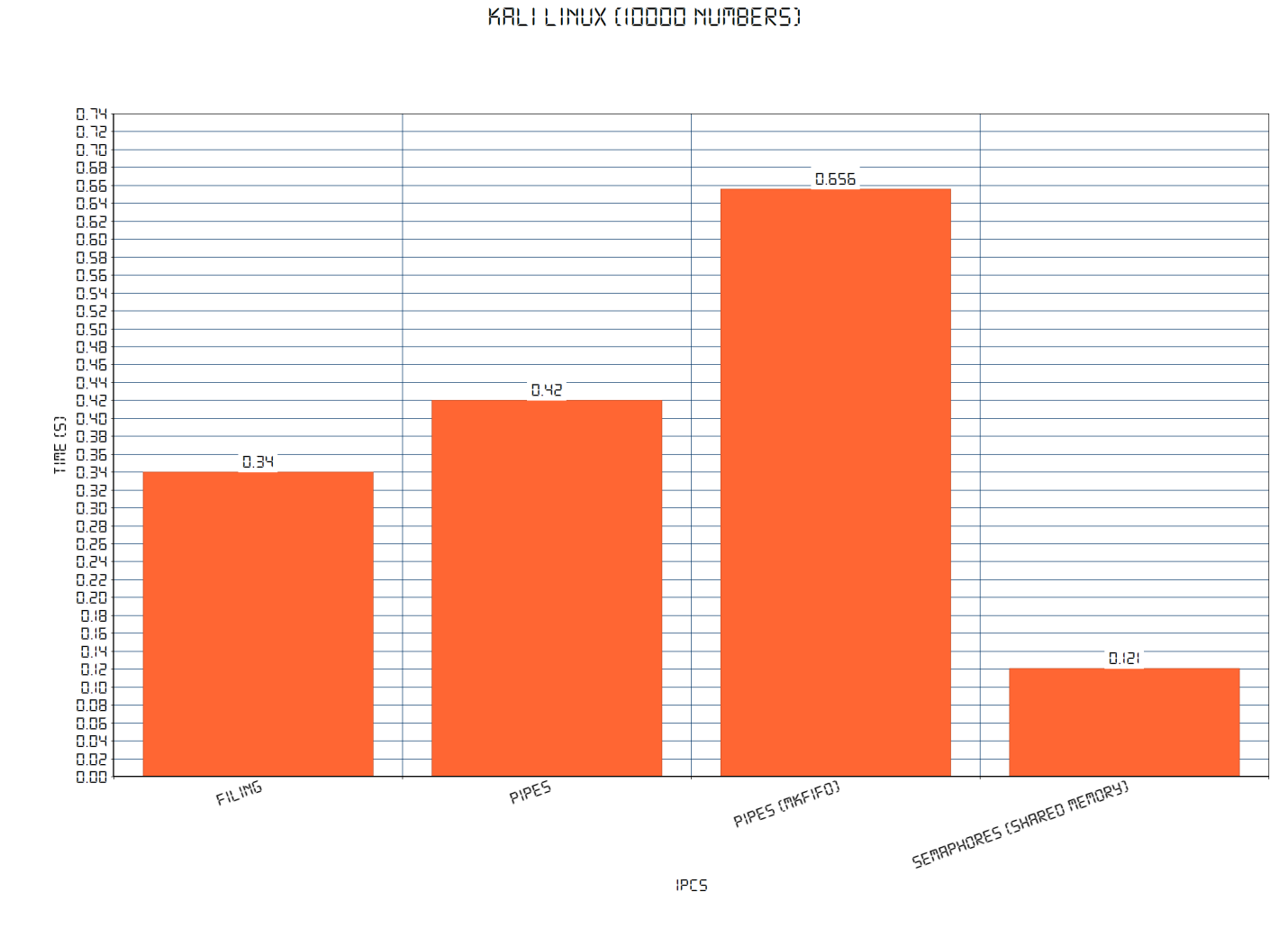
**(Time in seconds)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Numbers (data)** | **FILING** | **PIPES (SIMPLE)** | **PIPES (MKFIFO)** | **Semaphores**  **(shared memory)** |
| 10000 | 0.34 | 0.42 | 0.656 | 0.121 |
| 20000 | 1.567 | 1.879 | 2.512 | 1.234 |
| 30000 | 2.261 | 2.456 | 3.11 | 1.78 |
| 40000 | 5.014 | 5.510 | 5.987 | 2.876 |
| 50000 | 5.510 | 5.878 | 8.234 | 3.871 |

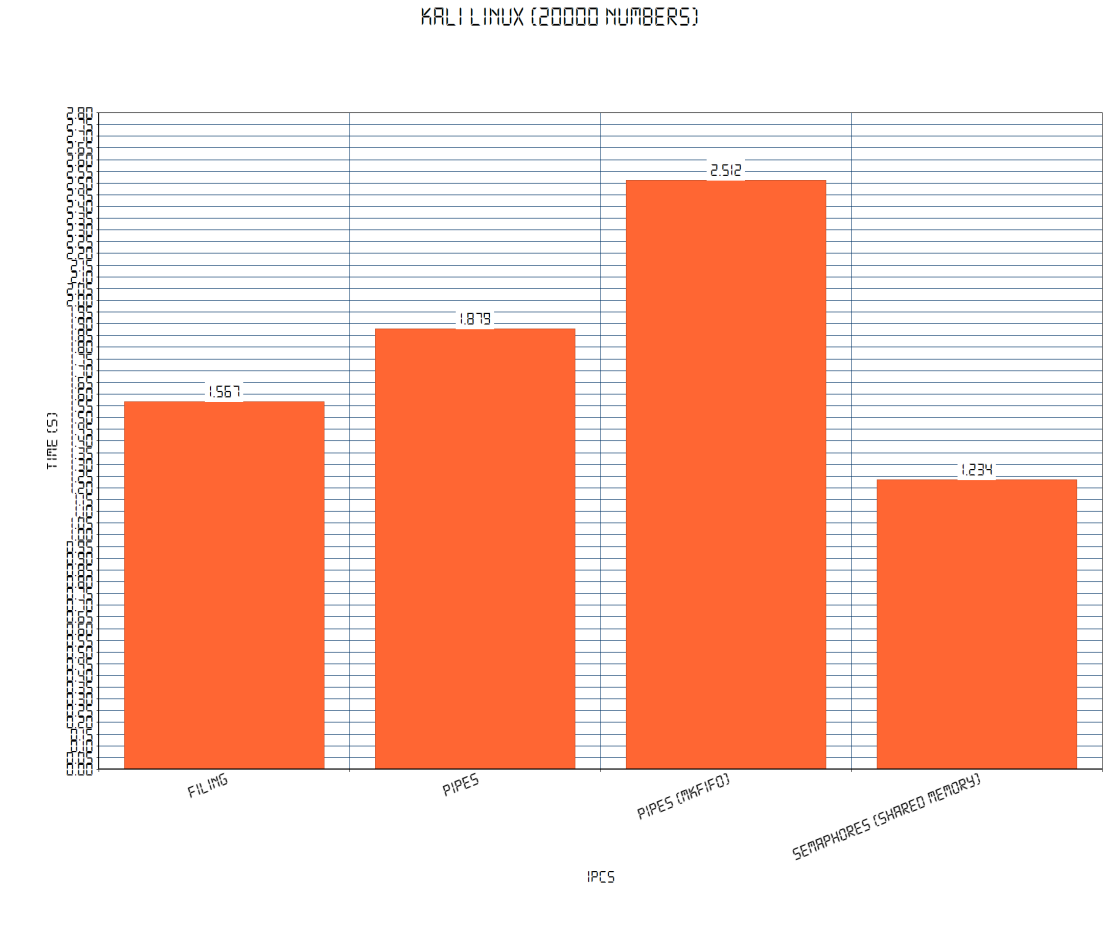
**UBUNTU:**

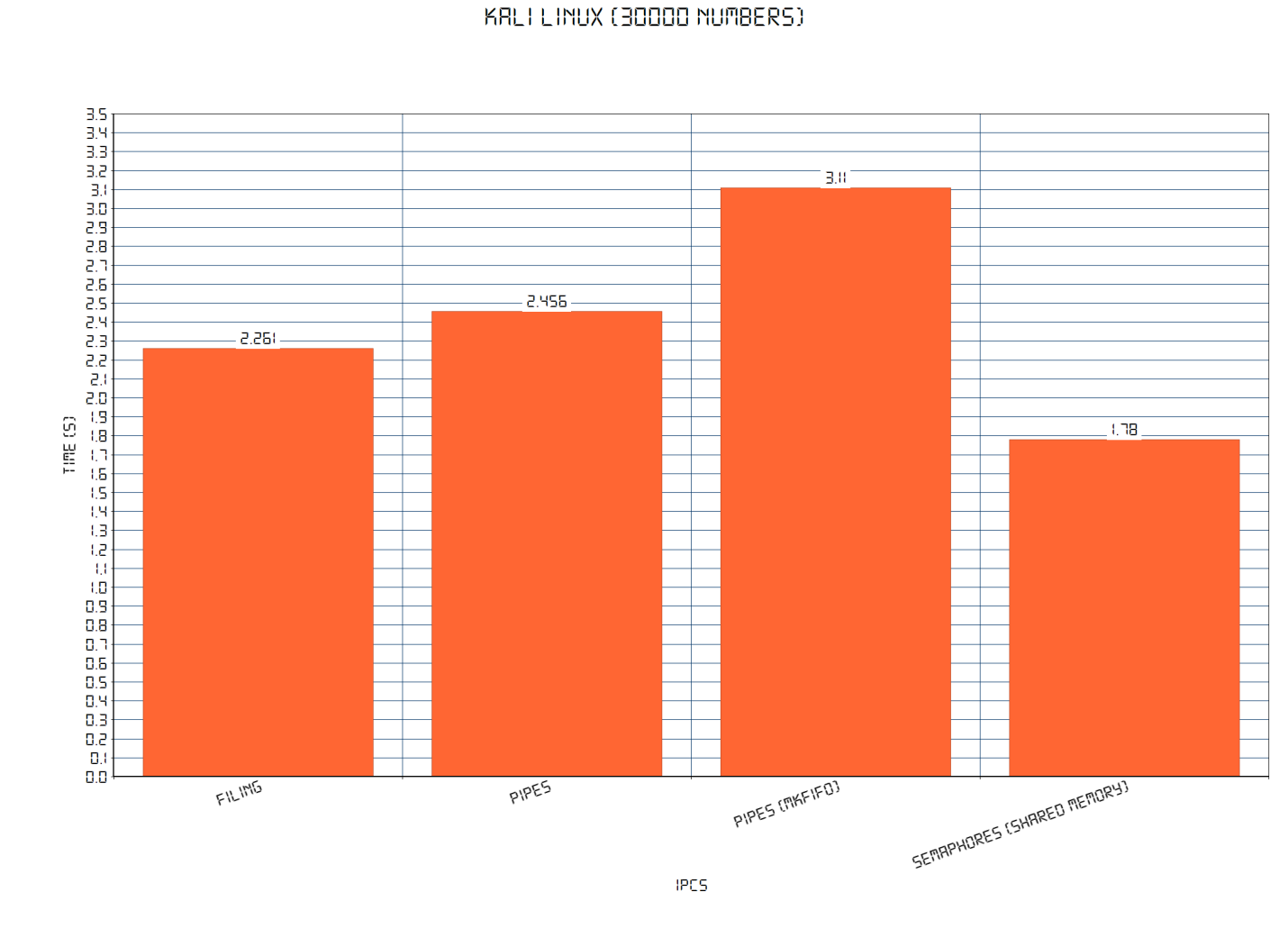
**(Time in seconds)**

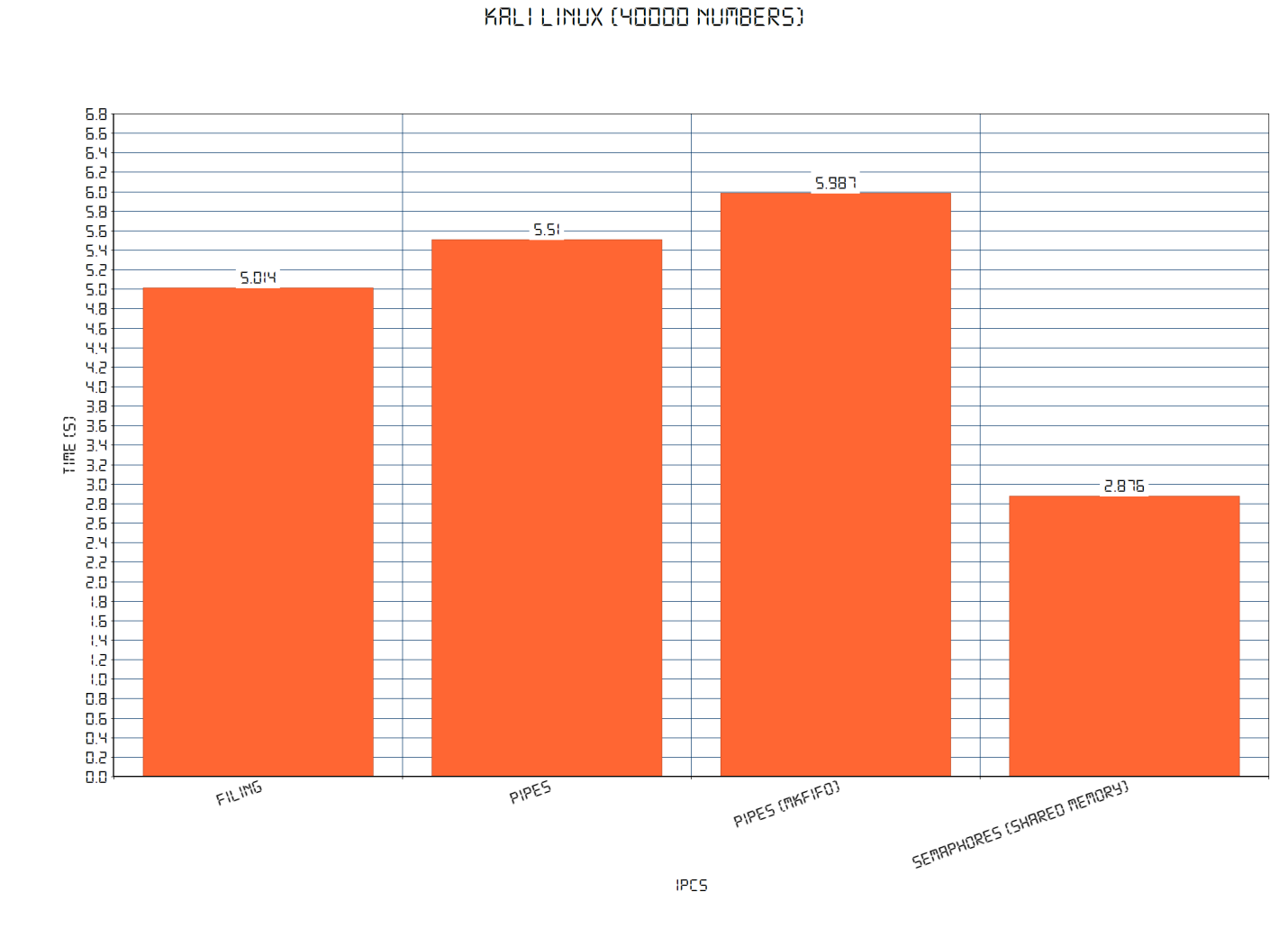
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Numbers (data)** | **FILING** | **PIPES (SIMPLE)** | **PIPES (MKFIFO)** | **Semaphores**  **(shared memory)** |
| 10000 | 0.533 | 0.536 | 0.581 | 0.511 |
| 20000 | 2.201 | 2.136 | 3.212 | 2.161 |
| 30000 | 2.562 | 2.765 | 4.321 | 2.123 |
| 40000 | 4.132 | 4.489 | 6.546 | 3.664 |
| 50000 | 6.394 | 6.845 | 9.934 | 4.445 |

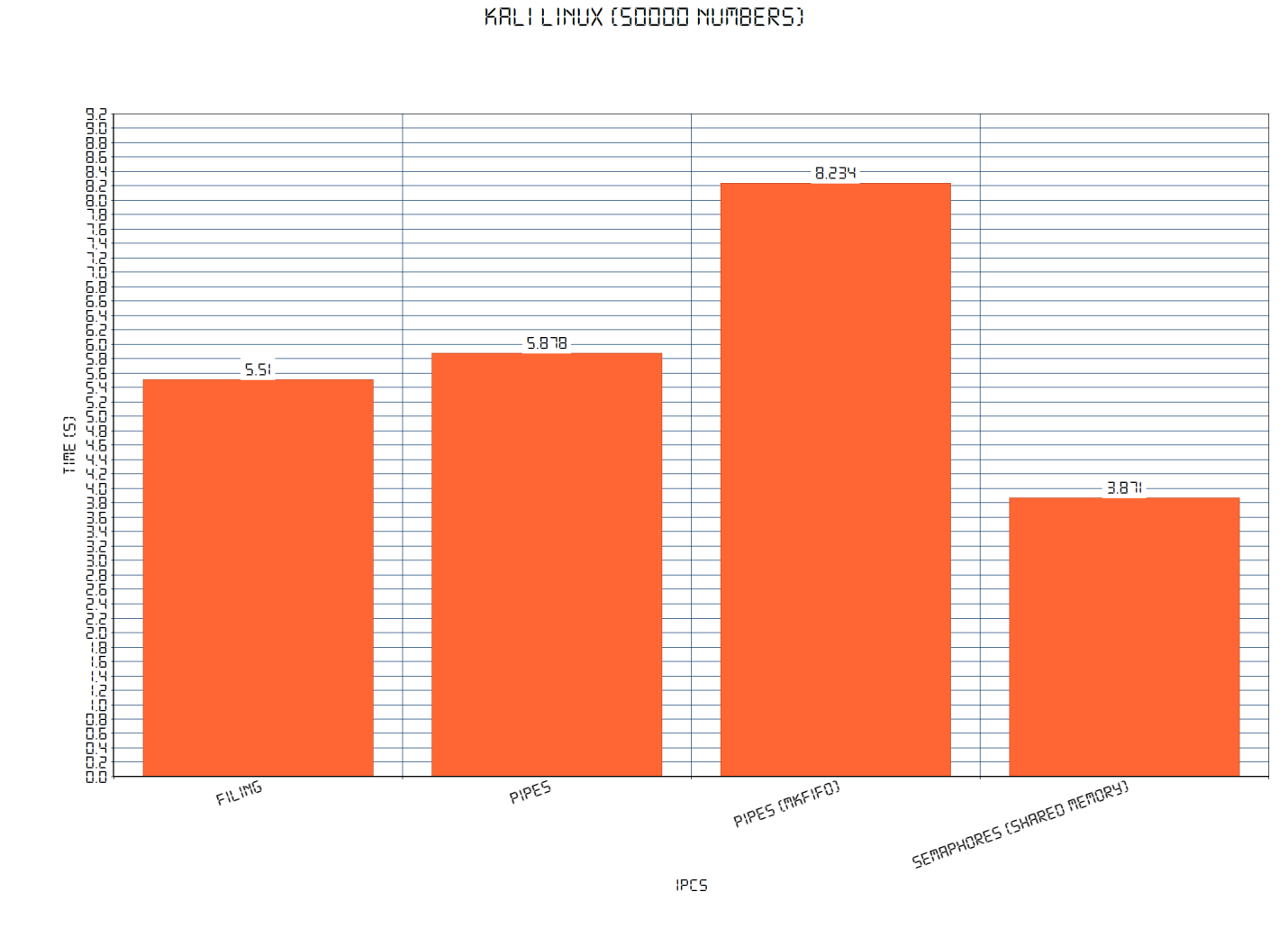


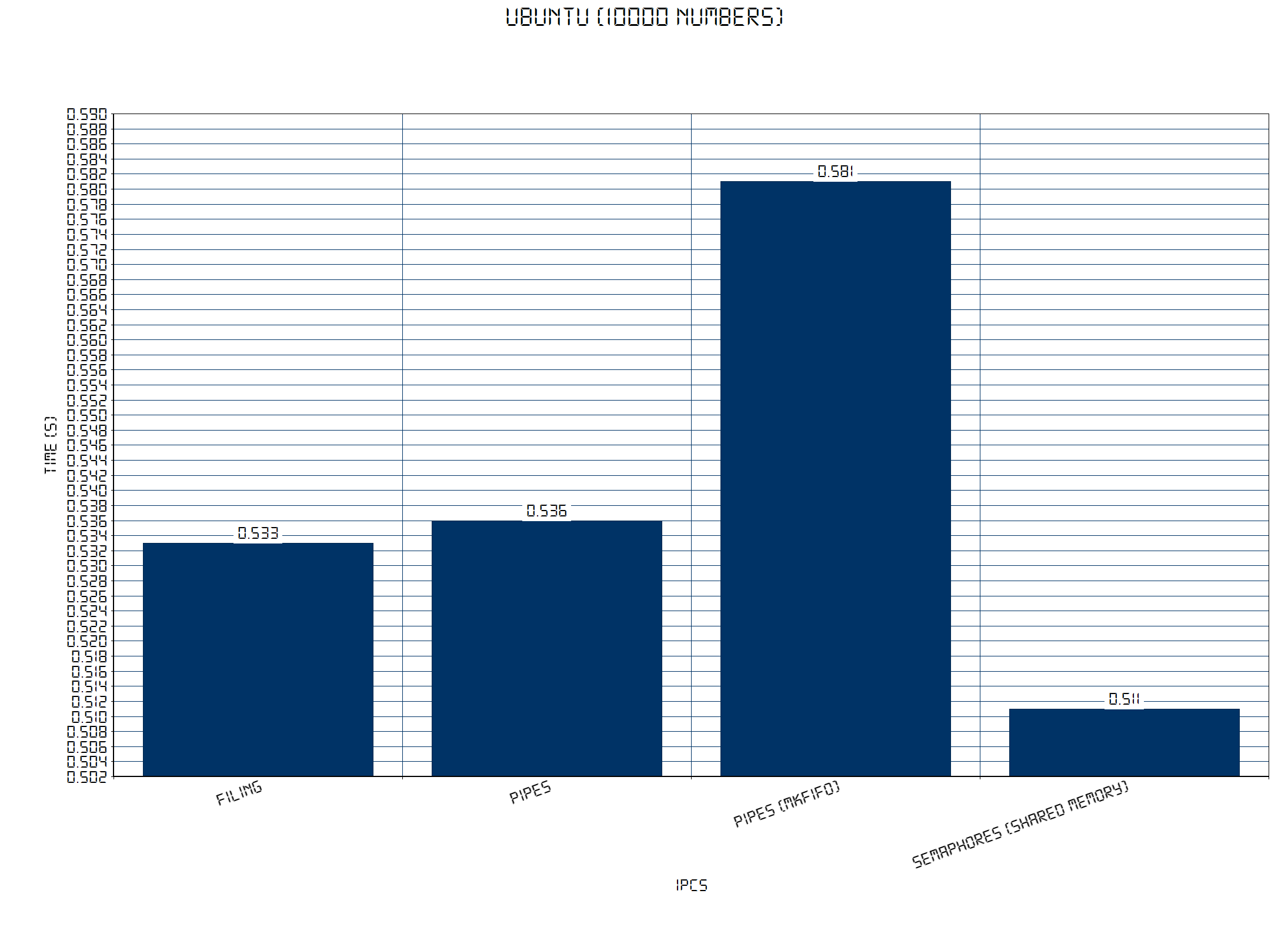
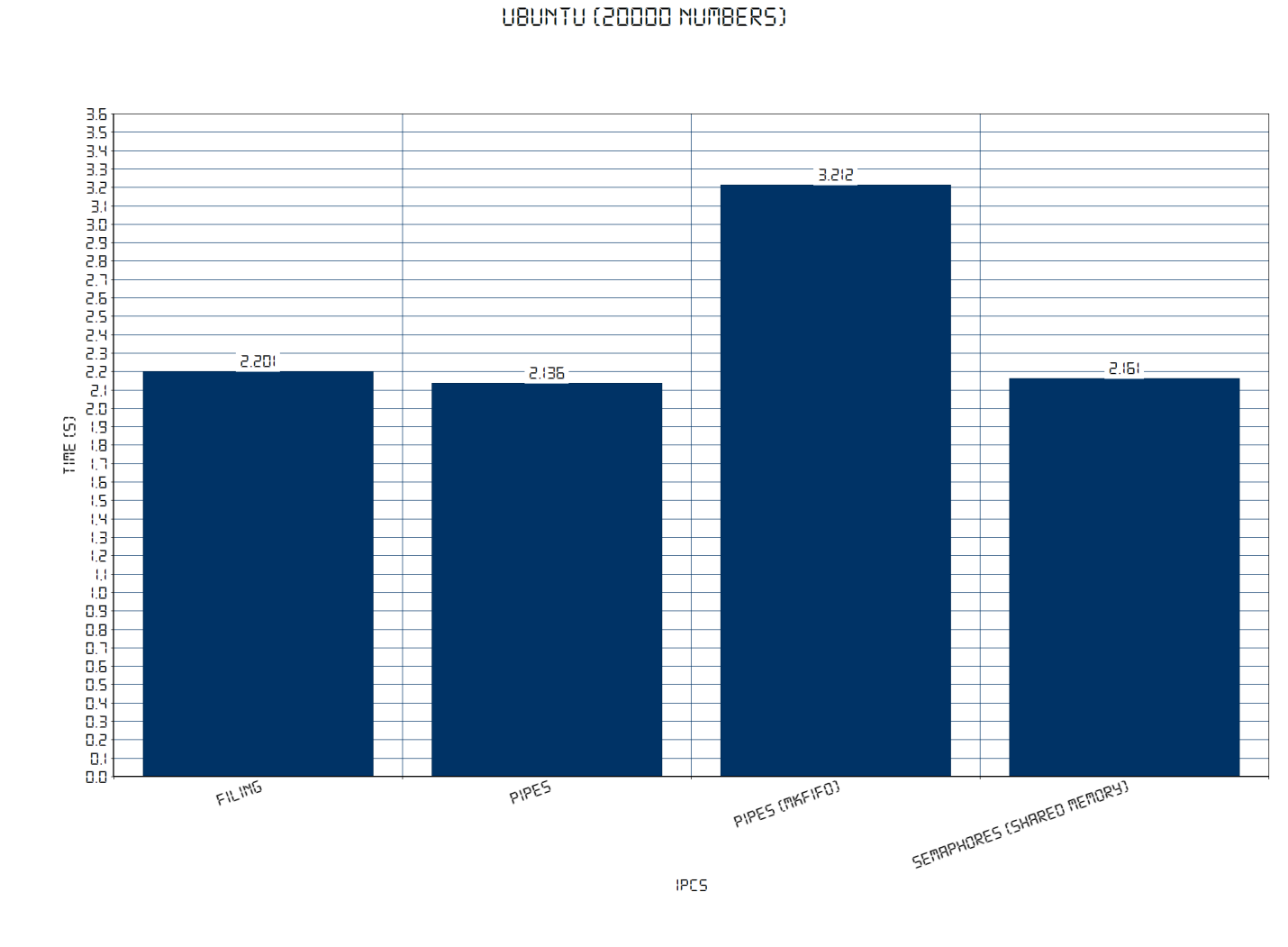
**GRAPHS:**

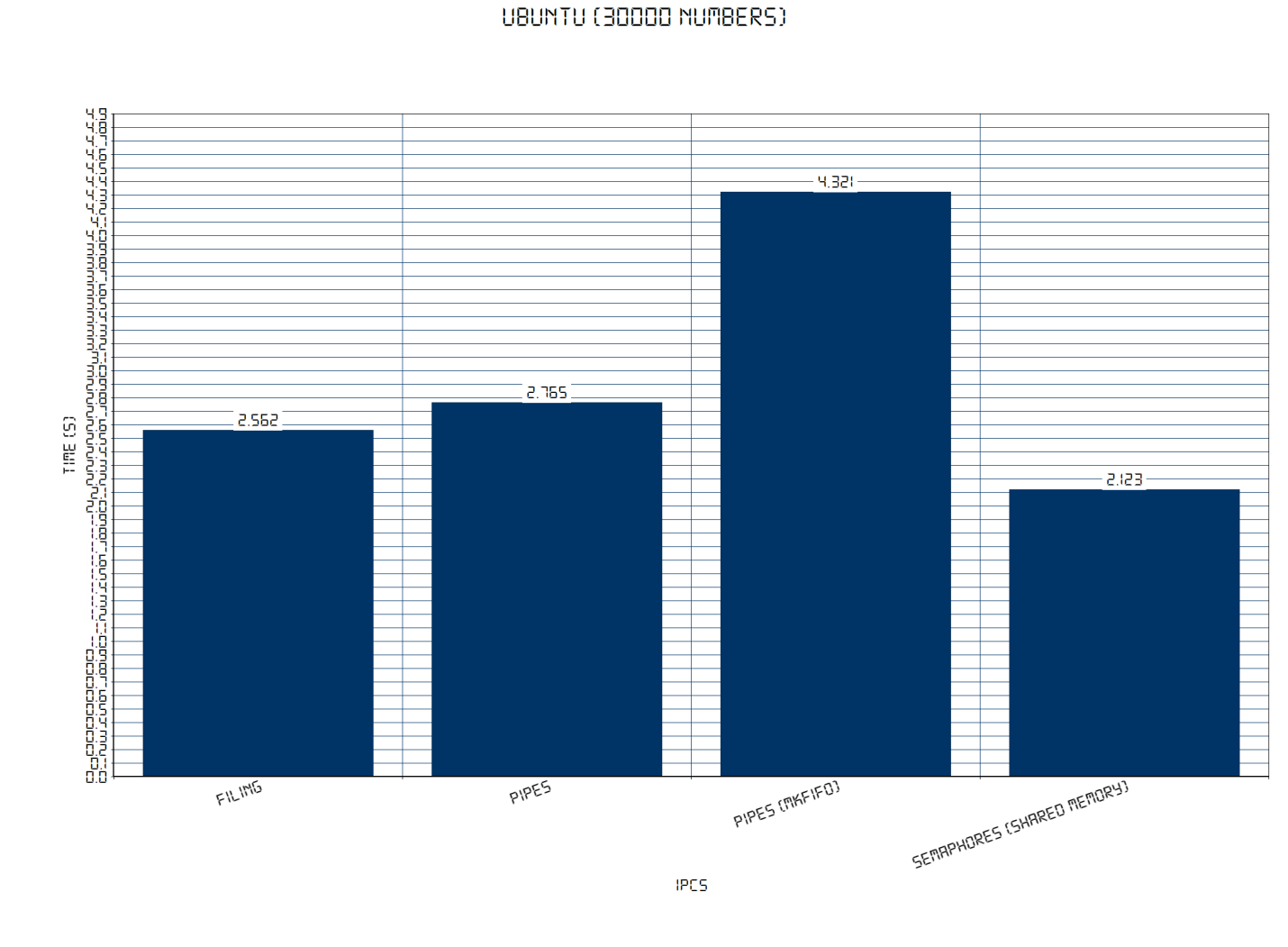
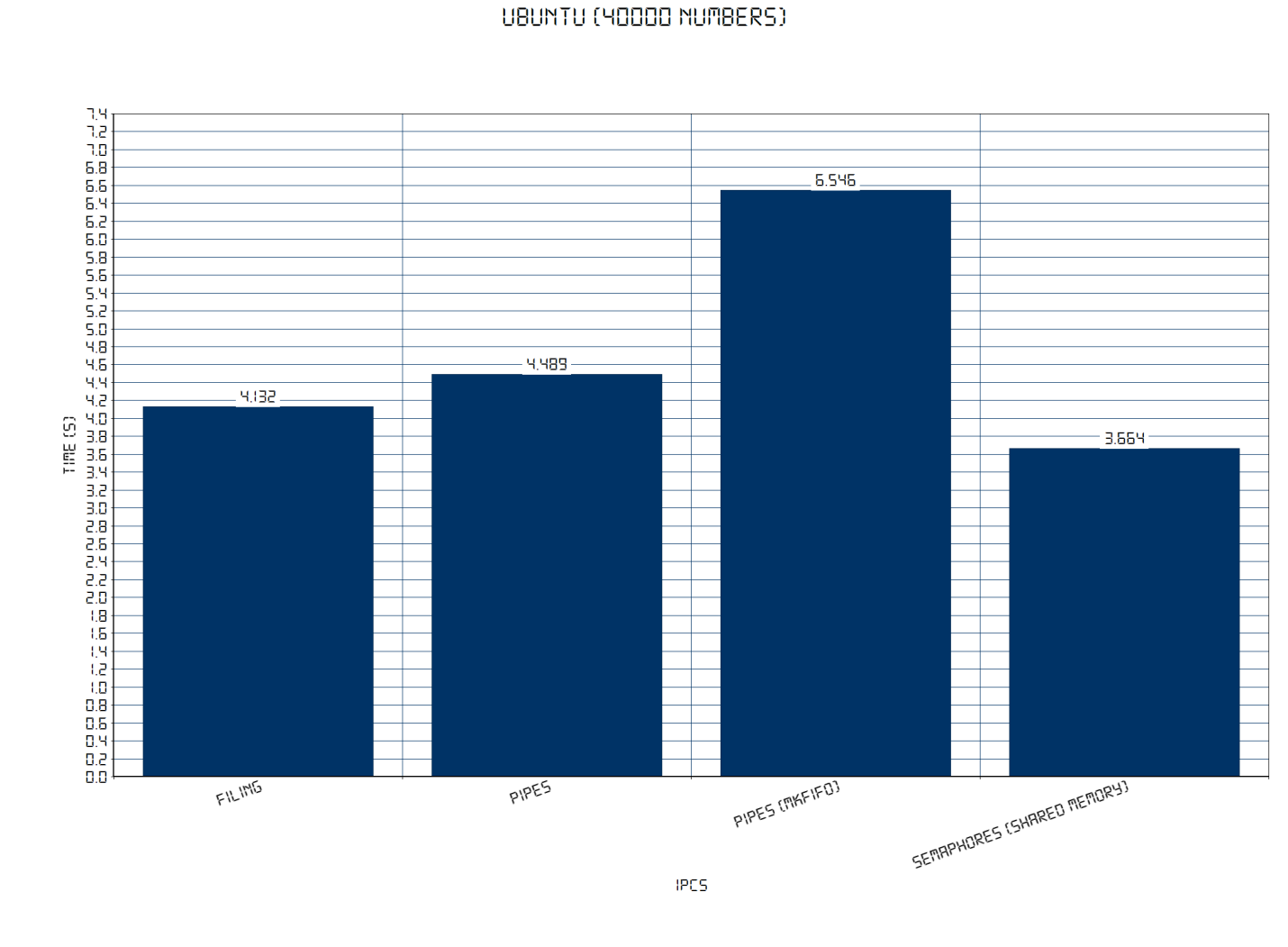


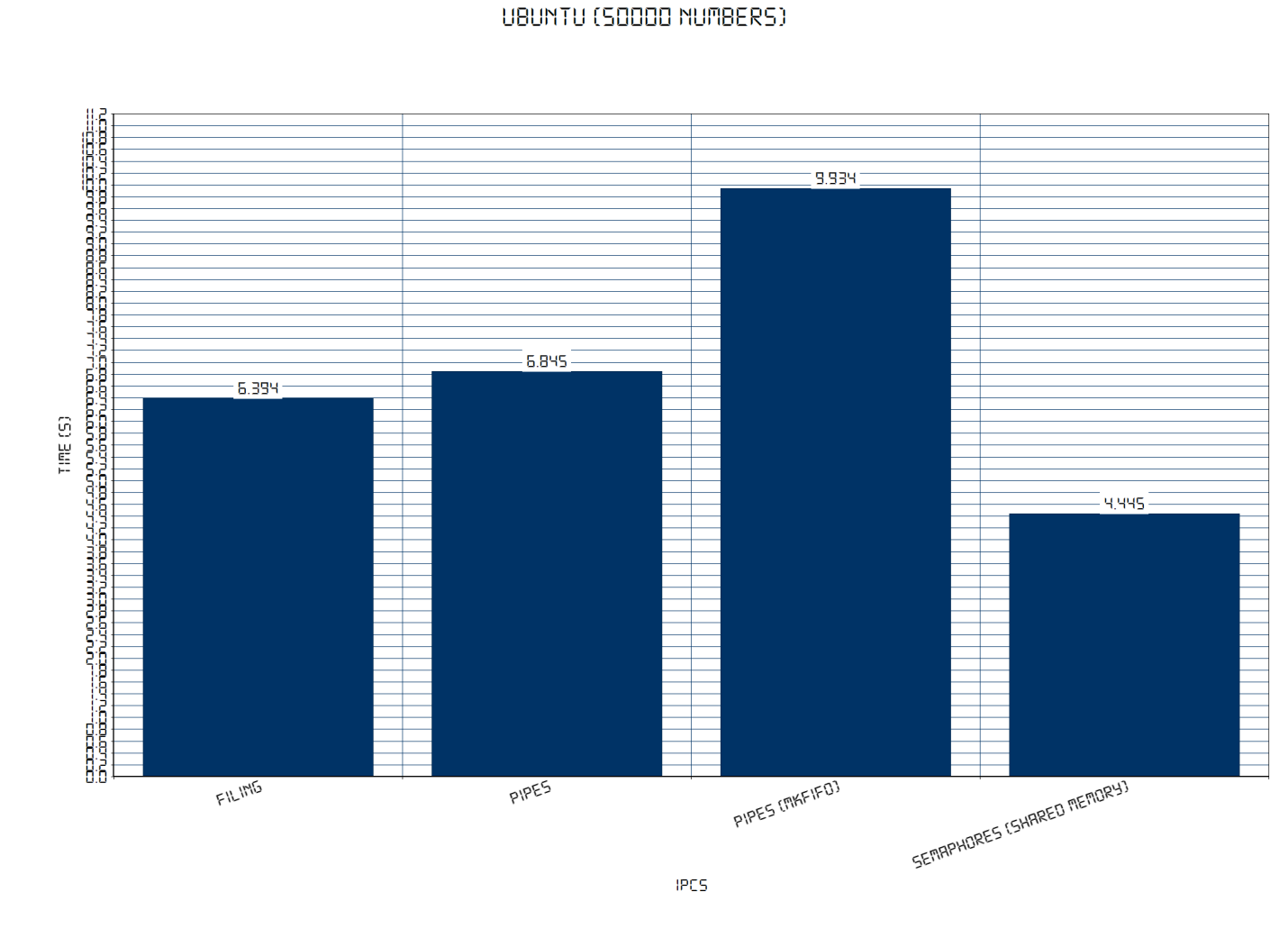
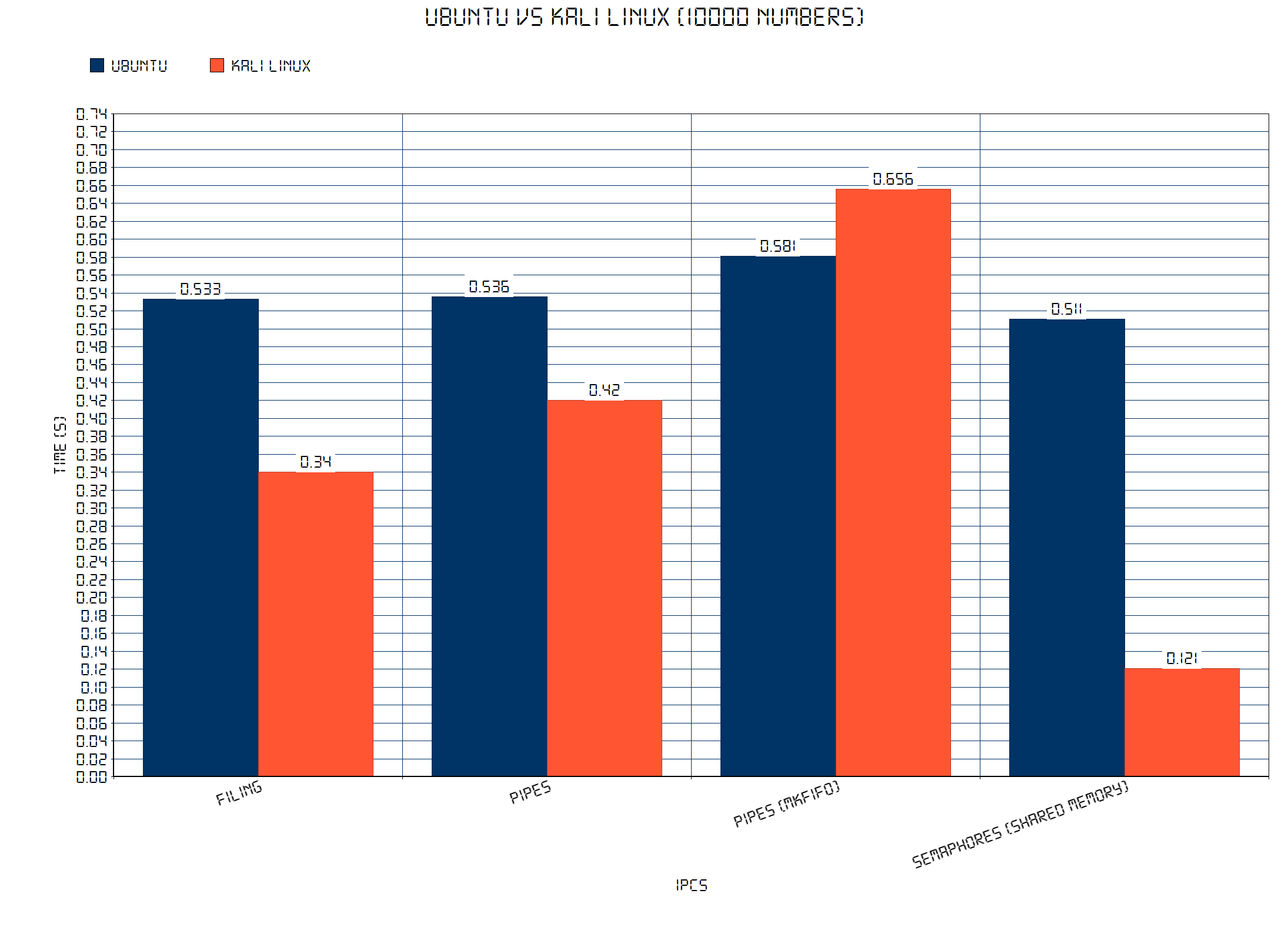


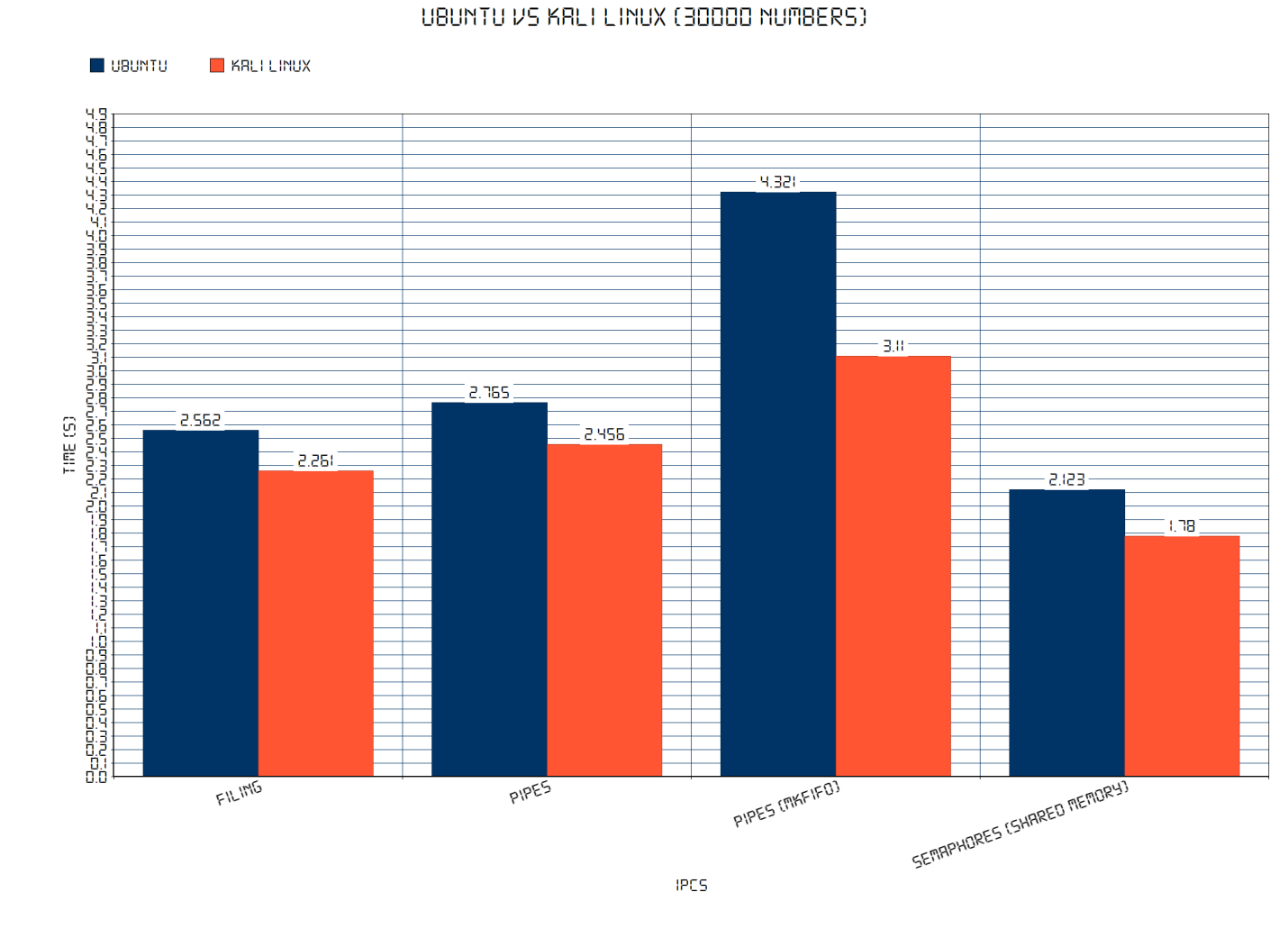
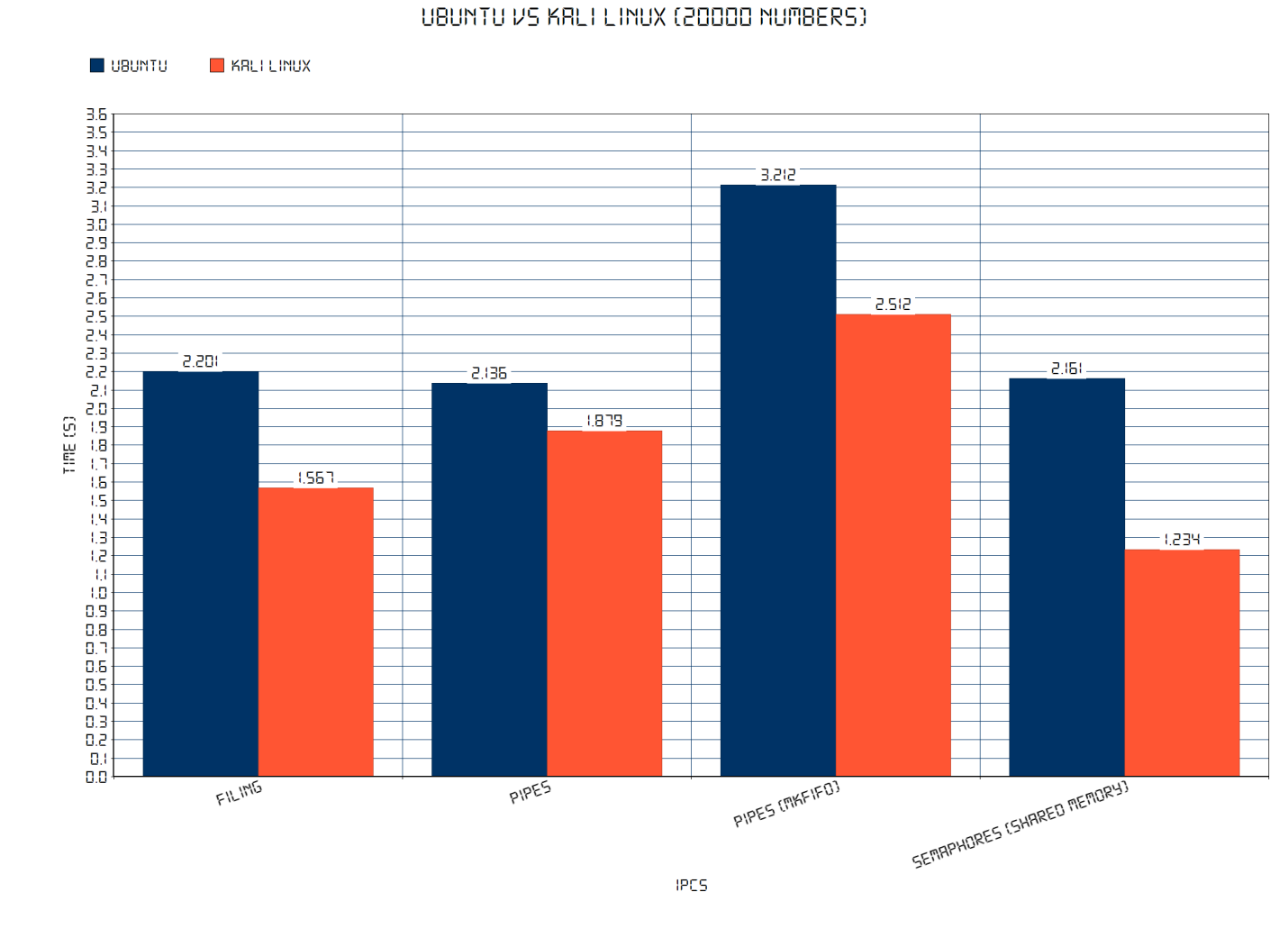


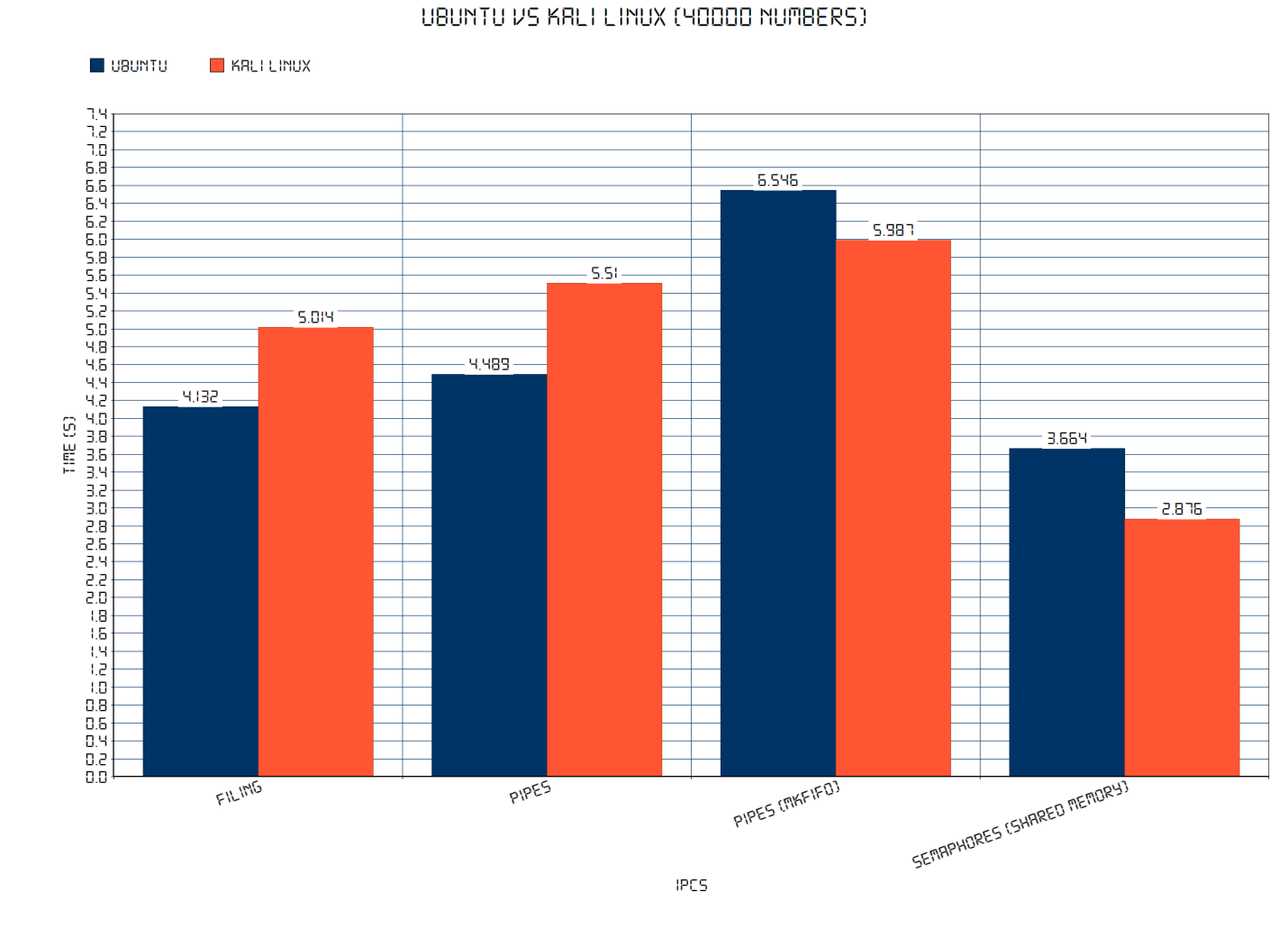
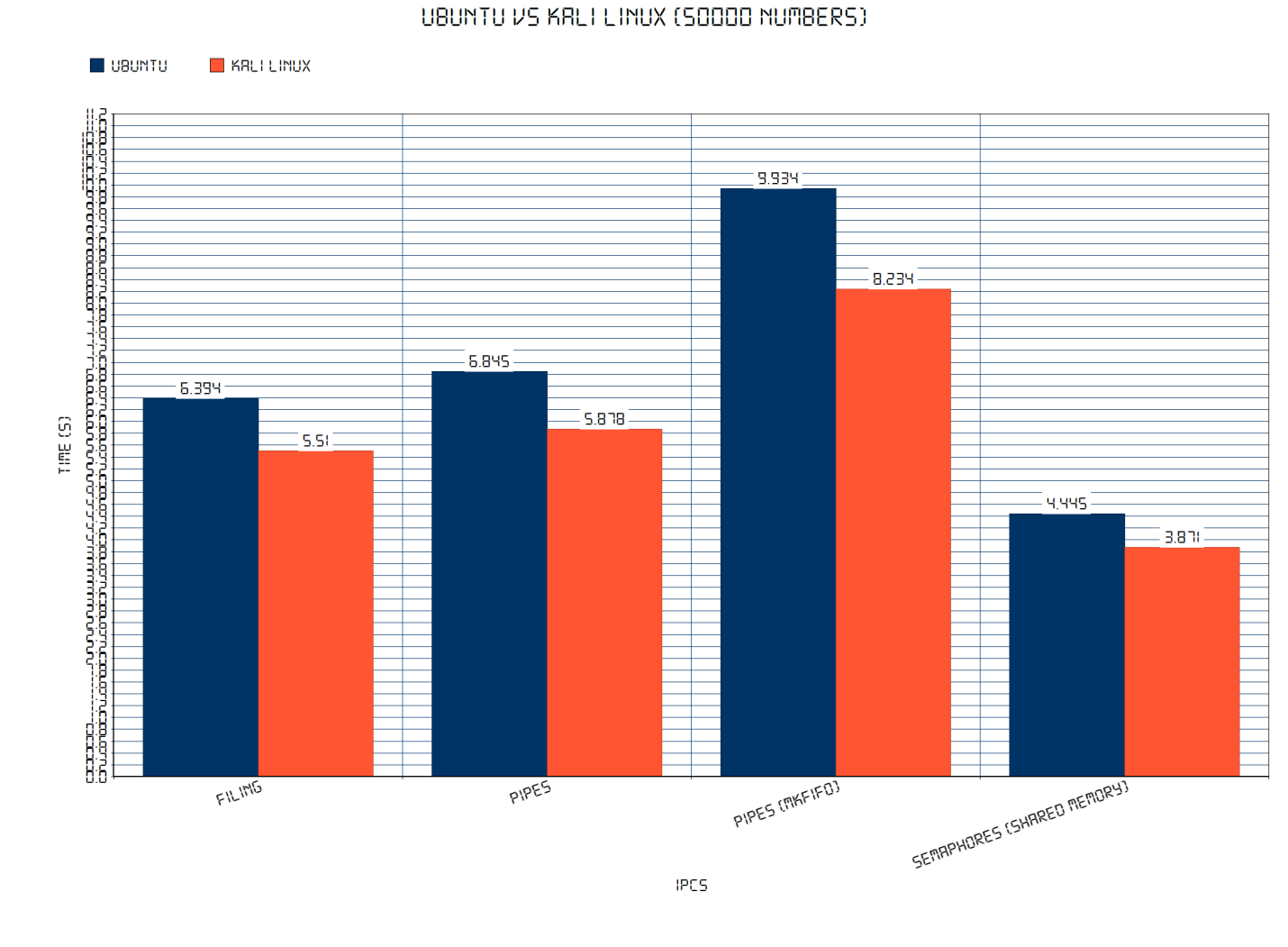




****

****

****

****

**CONCLUSION:**

To conclude, we have noticed that even though both the operating systems are working on the same number of processors and ram, the Kali Linux is the fastest for interprocess communication, be it using filing, semaphores or pipes. However, only once Kali Linux has taken a little more time than Ubuntu while sorting 40000 numbers via filing and pipes.

Now, it is noticeable that using Semaphore has turned out to be the fastest way of communication whether it is being used on Ubuntu or Kali Linux. And then follows filing and pipes. However, it is seen that pipes using mkfifo command takes the longest time to sort in both the operating systems.

Essentially, pipes - whether named or simple, are used like message passing. Someone sends a piece of information to the recipient and the recipient can receive it. Shared memory is more like publishing data - someone puts data in shared memory and the readers must use synchronization. With shared memory, it is easier to work asynchronously and check for new data only once in a while - but at the cost of much more complex code.

Differentiating between pipes, it is known that named pipe can handle two way communication as well as one way while simple pipe can handle only one way of communication, however simple pipe consume less time for IPC as it is just a simple two end pipe with one receiving and one sending end. And named pipe does not allow multiple processes to run on a single named pipe, hence taking more time than simple pipe to sort the numbers.