Fundamentos de Sistemas de Operação

MIEI 2017/2018

Laboratorial Session 1

Objectives

Remember C programming. Exercise and compare Input/Output (I/O) programming at different levels: systems calls in *UNIX*/*Linux* systems, *standard C library* and Java. Obtain a program's execution time and estimate a system call overhead.

Introduction

Programming languages typically offer in their runtime libraries a more or less comprehensive set of commonly used *abstractions* and *operations*. Some of these operations are based on (i.e. use) services offered by the underlying operating system in the form of *system calls*. Such *library functions* aim to enhance programming by hiding some low-level details of system calls, also improving portability. Those functions can introduce some overheads but try to improve a program's performance for the common cases. It is up to the programmer to either use the system calls directly or to resort to the available runtime libraries. For instance, the *C language standard* offers the type *FILE* that can be created to access a file by using the operation *fopen*, and it is destroyed with *fclose*. In Java several classes are offered, like *FileInputStream* and *FileOutputStream*.

Subsequently to a *fopen*, the operations *fwrite* and *fread* allow writing and reading raw blocks of bytes to/from a file. In turn, the operations *printf* and *fprintf* are used to print text and perform any necessary conversions, e.g. when printing numerical data types (int, float, etc) to text files, whereas *fgets* and *fscanf* are used to read formatted text. In Java, after creating a stream object to access a file, the methods read and write can be used to read or write blocks of bytes. From those objects, *PrintStream*, *InputStream* and *Scanner* objects can be created allowing reading and writing of typed data types from/to text files with the necessary conversions.

The above I/O libraries and classes offer several operations with different functionalities and data processing and conversion but must request the operating system for the real access to the computer files and devices. Namely, those runtime libraries use an interface with the operating system — i.e. make calls to the system's kernel via the system calls interface functions offered in *libc*. The system calls for file I/O, namely, *open*, *close*, *write* and *read,* only allow unformatted/raw data writes and reads. The abstraction offered to the programmer is of a "*channel to a file*" that is created with the *open* system call and destroyed/freed by *close*.

The C library functions *fwrite*, *fread*, *fprintf*, *fgets*, *fscanf*, etc, and Java methods *read*, *write*, *println*, *nextLine*, *nextInt*, *nextFloat*, etc. use the *write* and *read* system calls to write and read the needed data. Moreover, they may also use internal buffers in order to e.g. read extra data that is available for use in a faster way.

Estimation of a System Call Overhead

A system call usually takes more time to execute than a regular function call. The execution time of each system call depends on the actions the kernel must complete internally before returning a reply to the calling process.

Start by measuring the average time it takes to call a regular function by using the following program (`timing.c`). For precision we must measure the time to execute several function calls and then calculate their average value.

```
#include <sys/time.h>
#include <stdio.h>
#define NTRIES  1000

int do_something(void) { return 1; }

int main (int argc, char *argv[]) {
    int i, p;
    long elapsed;
    struct timeval t1,t2;

    gettimeofday(&t1, NULL);
    for (i = 0;  i < NTRIES; i++)
        p = do_something();  // code to evaluate
```

```
    gettimeofday(&t2, NULL);
    elapsed = ((long)t2.tv_sec - t1.tv_sec) * 1000000L + (t2.tv_usec - t1.tv_usec);
    printf ("Elapsed time = %6li us (%g us/call)\n", elapsed, (double)elapsed/NTRIES);
    return 0;
}
```

Afterwards, evaluate the times for the following cases:

simple system call – replace the *do_something* function call in the for loop in order to measure one of the simplest system calls, *getuid* (that returns the identifier of the user that launched the process).

I/O operations – replace the *do_something* function in the for loop with the functions in the following two cases, in order to evaluate the time it takes to write some text to the screen:

1. `printf("writing");`
2. `printf("writing"); fflush(stdout);`

Compare the times and explain the differences.

Copying a file

I/O programming with Java classes

Look at the Java program `Copia.java`. It will copy an existing file, similar to the existing **`cp`** command. The command:

```
java Copia 100 FILE1 FILE2
```

should create a replica of file `FILE1` with name `FILE2`, copying 100 bytes at a time. To check if the files are indeed identical you may use the **`cmp`** command as follows:

```
cmp FILE1 FILE2
```

I/O programming with the C Standard Library

Look at the new version of the previous program using the standard C functions for the I/O operations (`fcopia.c`), namely **`fopen`**, **`fread`** and **`fwrite`**. The command can be used like:

```
fcopia 100 FILE1 FILE2
```

to create a replica of file `FILE1` with name `FILE2` (copying 100 bytes at a time).

I/O programming with System Calls

Look at the other version of the previous command using just Unix's system calls C interface, **`open`**, **`read`**, **`write`** and **`close`** for the I/O operations: **`copia.c`**. This can be used like the previous one.

Evaluation

Use the **`time`** command to obtain the time that each program takes to copy a big file. For that purpose, place the **`time`** command before your own. Example:

```
time fcopia 100 file1 newfile
  real    0m0.175s
  user    0m0.001s
  sys     0m0.009s
```

Use an existing file with at least 10Mbytes (you can use any one or create a new one for testing using a command like: `dd if=/dev/zero bs=1M count=10 of=file1`)

Execute your program with different block sizes (1, 128, 1024, 10240). Also count the number of system calls using *strace* command for the `fcopia` and `copia` programs:

```
strace -c fcopia 100 file1 newfile
```

Compare the performance of each version, with each block size and its use of system calls. Justify the results and performance discrepancies.

Bibliography

- Sections 101.1 to 101.5 of recommended book's "Lab Tutorial": http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf
- On-line manual pages for
    - the *cp* and *dd* commands
    - the system call functions *open*, *read*, *write* and *close*
    - the functions of the C library: *fopen*, *fread, fwrite* and *fclose*

  You can type the `man` command on your terminal or Internet browser, for example: `man cp`