NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTAMENTO DE INFORMÁTICA

Libraries, syscalls and memory map

LAB 1

Objectives

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

Identify how the memory regions (code, data, stack, heap) of a process are laid out in memory – something that we call the memory map of the process. Sketch the memory map of a Linux process. On different OS the memory layout can be, naturally, different.

Software stack and system calls

Programming languages typically offer in their runtime libraries a 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 to different OS. These function stack can introduce some overheads but try to improve a program's performance for the common cases. 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*.

For C, 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 type conversions, e.g. when printing numerical data types (int, float, etc) to text files, whereas *fgets* and *fscanf* are used to read plain 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 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 request the operating system for the real access to the computer files and devices. Moreover, they may also use buffers in order to optimize these operations.

Estimation of Call Overhead

A system call can take more time to execute than a regular function call. The execution time of each system call includes the added time for context switch from user mode to kernel mode and also 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 calls and then calculate their average value.

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 four values 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, like the existing `cp` command. After "compiling" you can run your program with:

```
java Copia 100 FILE1 FILE2
```

This should create a copy 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 version of the previous program using the standard C functions for the I/O operations (`fcopia.c`), namely `fopen`, `fread` and `fwrite`. After compiling, the program can be used like:

```
fcopia 100 FILE1 FILE2
```

to create a copy 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 file2
  real    0m0.175s
  user    0m0.001s
  sys     0m0.009s
```

Use an existing file with at least 20Mbytes (you can use any one or create a new one for testing using a command like: `dd if=/dev/random bs=1M count=20 of=file1`)

Execute your program with different block sizes (1, 64, 128, 1024, 4048, 10240)[1]. 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 differences.

[1] You can also see if there are differences around each value, like comparing results for 1024 and 1025.

Memory Map of a (Linux) process

The base program is mem.c (see source code), that displays the following locations:

a) of **code**, using the address of the main function;

b) of **data**, using the addresses of global data items, both, constant (`strC`) and variables (`strV`, `globalVar`, `globalVarInit`);

c) of **local variables**, stored in the stack (`x`);

d) and of **dynamically allocated memory**, stored in the heap (a 100 MB character array).

The addresses are taken via pointers and displayed both as hexadecimal (using the `%p` pointer format specification of `printf`) and decimal number (using the `%zu` unsigned integer format specification of `printf`). Type casts are used to enforce the compatibility of the pointers with unsigned int. When compiled in a 32-bit (**x86**) architecture, a pointer is a 32-bit sized data, and therefore is matched by the `uintptr_t` to an `unsigned int` data type, which is also 32-bit long. When compiled in a 64-bit (**x86_64**) architecture, the pointers are 64-bits and `uintptr_t` is defined accordingly to `unsigned long int`. The program is compiled with **`-Wall`** (see Makefile) should not raise warnings. You can also try to compile and execute in MacOS as this is a Unix-like system as is GNU/Linux.

Compile the mem.c program using `make`, and run it; the output should be similar to (for 32 bits architecture):

```
Memory map of a process

Architecture sizes:
int 4, long 4, long long 8, pointer 4
location of the code      (main):  0x40060d (hex)  4195853 (dec)
location of a (string) constant :  0x400860 (hex)  4196448 (dec)
location of a (string) variable :  0x402008 (hex)  4202504 (dec)
loc. of a global initialized var:  0x40201c (hex)  4202524 (dec)
loc. of a global non-init'ed var:  0x402024 (hex)  4202532 (dec)
location of the top of the heap :  0xb1e87010 (hex) 2984800272 (dec)
location of the top of the stack:  0xbffff144 (hex) 3221221700 (dec)
```

Note that those addresses will change at each execution. That is a security feature[2] that can be disabled by executing your program like[3]:

```
setarch -R ./mem
```

Now, draw (sketch) a map **IN A PAPER SHEET** and write the pertinent information. Remember that those addresses are always **virtual addresses**. For a 32 bit architecture should be something similar to (not to scale):

[2] Address space layout randomization (ASLR) - https://pt.wikipedia.org/wiki/Address_space_layout_randomization

[3] From MacOS see the Makefile.

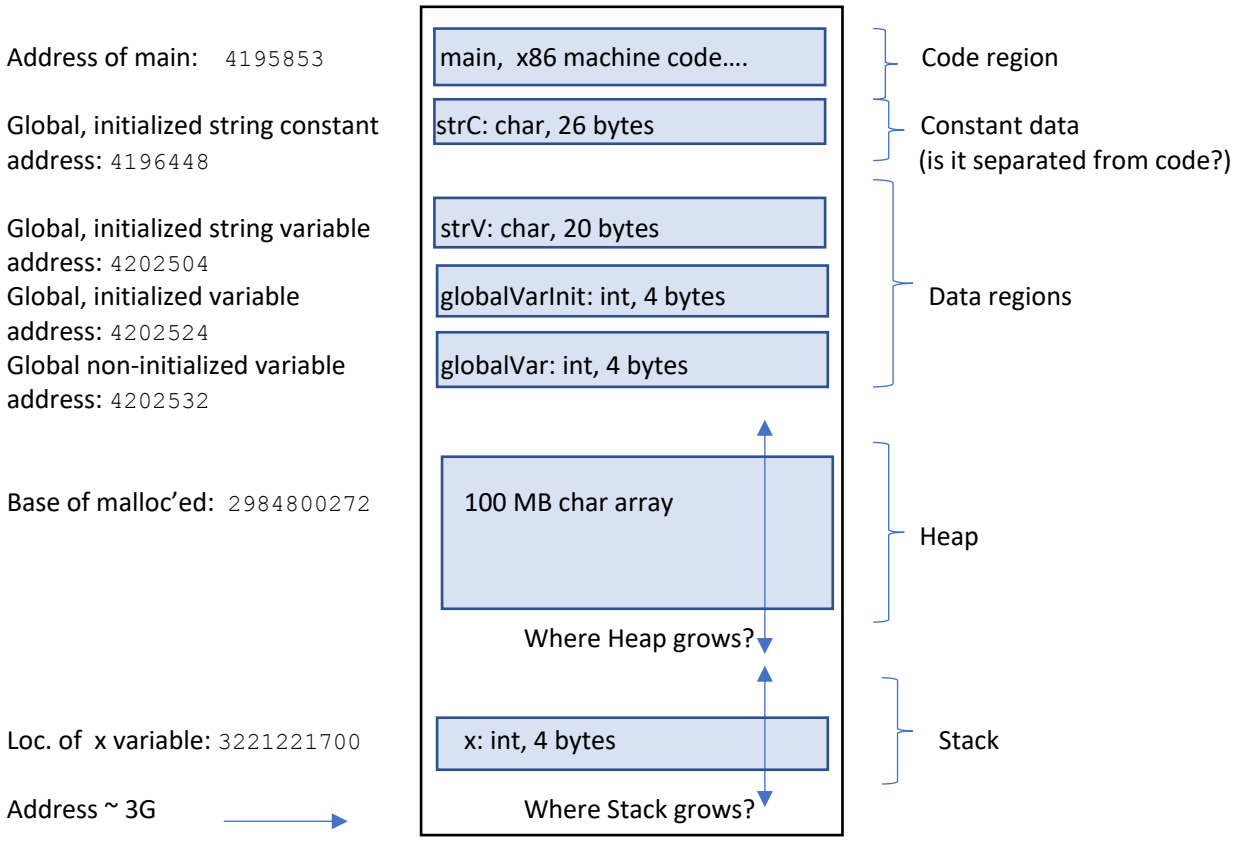

Figure 1. Memory map of the mem as a 32bit process

Discovering how the stack and heap behave (grow/shrink)

Now add some code to the program to discover how the stack and the heap grow. Suggestions: add a new function that prints the address of a local variable and call it from main; create a new variable with malloc and print the new address.

Gather information from a running process

For the next part of the exercise, it is easier if you have two terminal sessions.

Put the program running in one terminal while you give the other commands in the other terminal. First, you must find the PID (Process ID) of the running program; to get that information run

```
ps -ef | grep mem
```

If our program is running with PID 1630, now run:

```
pmap 1630
```

The output should be similar to:

```
1630:   mem
00400000      4K r-x-- mem
00401000      4K r---- mem
00402000      4K rw--- mem
00403000    136K rw---   [ anon ]
b1e87000  97660K rw---   [ anon ]
b7de6000   1876K r-x-- libc-2.27.so
b7fbb000      4K ----- libc-2.27.so
b7fbc000      8K r---- libc-2.27.so
b7fbe000      4K rw--- libc-2.27.so
b7fbf000     12K rw---   [ anon ]
b7fd2000      8K rw---   [ anon ]
b7fd4000     12K r----   [ anon ]
```

```
b7fd7000      4K r-x--   [ anon ]
b7fd8000    152K r-x-- ld-2.27.so
b7ffe000      4K r---- ld-2.27.so
b7fff000      4K rw--- ld-2.27.so
bffdf000    132K rw---   [ stack ]
 total   100028K
```

The above map shows several "regions", one per line; each region is described by its starting address, followed by the size, its permissions and the name of the file associated with that region. As an example, the first line,

```
00400000      4K r-x-- mem
```

starts at memory address 0x00400000 (that is, at decimal 4464640 or 4MB), has a size of 4KB (that is, 4096 bytes), only allows read and execute accesses during the execution of the process, and the data stored in this region can be found in the **`mem`** file. You may notice that **`main`** is at an address inside that region.

The second region starts at memory address 0x401000, has a size of 4KB, but only allows read accesses during the execution of the process – at first sight one would think that obviously that is here the string constant would be, but when we look at its address (`0x400860`) we see that the **`strC`** is in the same region as the code and associated to the same file! Why?

But, as far as variables are concerned, they are in a rw protected region, all inside the next region:

```
00402000      4K rw--- mem
```

So why are most "structures" 4KB in size? Remember, from the AC course, that Intel/AMD architectures use a 4K pagesize by default.

And what about these `anon` areas? What are they? Well, anon is the abbreviation for anonymous, and that refers to the fact that their contents do not come from the **`mem`** executable file, but are dynamically created. Example:

```
b1e87000  97660K rw---   [ anon ]
```

is where the heap is located and we have allocated a 100 000 000 bytes array (97660 x 1024 = 100 000 384, a little bit larger than 100 000 000).

To finish things up, what about these **`libc-2.27.so`** regions, such as this one?

```
b7de6000   1876K r-x-- libc-2.27.so
```

That's were code in the C library (e.g., `printf` code) is stored; as our program uses some standard C functions, like `printf` and `malloc`, and others that we do not "see" in our code, the library where those functions are implemented must be "glued" to our own code. Here, a special form of "gluing", named shared library mapping, is used (hence the `.so` suffix in the filename).

Bibliography

- Section "Laboratory: Tutorial" of recommended book: 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: *malloc*, *fopen*, *fread, fwrite* and *fclose*

  You can type the `man` command on your terminal, for example: `man cp`.