FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
DEPARTAMENTO DE INFORMÁTICA

Memory Map of a (Linux) process

(Lab-02 – September 2019)

1. Lab objective

An operating system, such as Linux, defines how the regions (code, data, stack, heap) of a process are laid out in memory – something that we call the memory map of the process. If the running process corresponds to the execution of a file that originally was, say, a C program, the C development toolchain (pre-processor, compiler, assembler and linker) all "cooperate" to produce an executable program that the OS loader is able to read and store in memory.

This Lab will allow you to sketch the memory map of a Linux process. However, some lab assignments can be run in other OSs, such as Windows – the map, will, naturally, be different.

2. The Base program (C coded)

The base program is depicted in Listing 1 below. It displays the location of:a) code, using the address of the main function; b) of data, using the address of global data items, both constant (strC) and variable (strV, globalVar, globalVarInit); c) of local variables (x), stored in the stack; d) and of dynamically allocated data structures (a 100 MB character array), stored in the heap.

The addresses are taken via pointers and displayed both as hexadecimal (using the %p pointer format specification of printf) and decimal (using the %u unsigned integer format specification of printf) numbers. Casts are used to assert that the size of the pointers and the print specifiers are matched.

The base program presented in Listing 1 is for a 32-bit architecture, where a pointer is a 32-bit sized data structure, and therefore is matched by an unsigned int data type, which is also 32-bit long in Intel and AMD x86 architectures. Therefore, although we must do quite a lot of casting, our program can be compiled with **`-Wall`** without issuing any warning output statements.

The 64-bit version is not included here, but is also available in the CLIP zip file. You should also compile and execute it (on x86-64 hardware with a 64-bit version of Linux).

2.1. The 32-bit (x86) base version

```
#include <stdio.h>
#include <stdlib.h>

const char strC[]= "Memory map of a (Linux) process\n\n";

char strV[]= "Memory map of a (Linux) process\n\n";

int globalVar;
int globalVarInit= 1;

int main(int argc, char *argv[]) {

  printf("%s%s", strC, strV);

  printf("location of the code      (main):  %p (hex)  %u (dec)\n", main,
(unsigned int)main);

  printf("location of a (string) constant :  %p (hex)  %u (dec)\n", strC,
(unsigned int)strC);

  printf("location of a (string) variable :  %p (hex)  %u (dec)\n", strV,
(unsigned int)strV);

  printf("loc. of a global initialized var:  %p (hex)  %u (dec)\n",
&globalVarInit, (unsigned int)&globalVarInit);

  printf("loc. of a global non-init'ed var:  %p (hex)  %u (dec)\n",
&globalVar, (unsigned int)&globalVar);

  char *ptr = malloc(100e6);
  printf("location of the top of the heap : %p (hex) %u (dec)\n", ptr,
(unsigned int)ptr);

  int x = 3;
  printf("location of the top of the stack: %p (hex) %u (dec)\n", &x,
(unsigned int)&x);

  return 0;
}
```

Listing 1. The Lab-02a-32bit.c program

Compile the Lab-02a-32bit.c program (in a 32-bit Linux) and run it; the output should be similar to:

```
Memory map of a (Linux) process

Memory map of a (Linux) process

location of the code      (main):  0x804849b (hex)  134513819 (dec)
location of a (string) constant :  0x8048680 (hex)  134514304 (dec)
location of a (string) variable :  0x804a040 (hex)  134520896 (dec)
loc. of a global initialized var:  0x804a064 (hex)  134520932 (dec)
loc. of a global non-init'ed var:  0x804a06c (hex)  134520940 (dec)
location of the top of the heap : 0xb1e1d008 (hex) 2984366088 (dec)
location of the top of the stack: 0xbfb470e4 (hex) 3216273636 (dec)
```

Now, draw (sketch) a map **IN A PAPER SHEET** and write the pertinent information; something similar to:

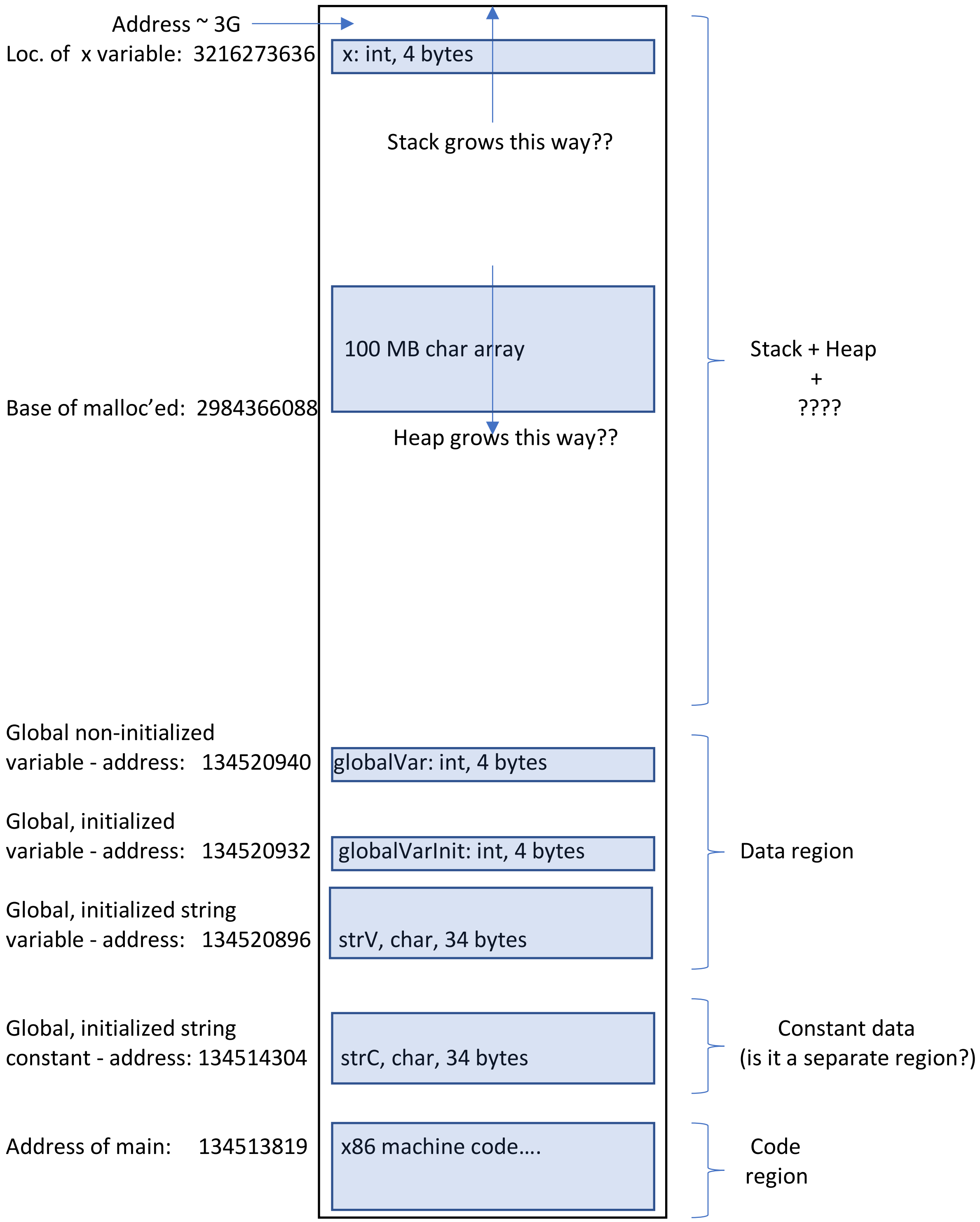
Address ~ 3G
Loc. of x variable: 3216273636
x: int, 4 bytes
Stack grows this way??
100 MB char array
Base of malloc'ed: 2984366088
Heap grows this way??
Stack + Heap
+
????
Global non-initialized
variable - address: 134520940
globalVar: int, 4 bytes
Global, initialized
variable - address: 134520932
globalVarInit: int, 4 bytes
Global, initialized string
variable - address: 134520896
strV, char, 34 bytes
Data region
Global, initialized string
constant - address: 134514304
strC, char, 34 bytes
Constant data
(is it a separate region?)
Address of main: 134513819
x86 machine code....
Code
region

Figure 1. Memory map of the Lab-02a-32bit process

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

Now add some code to the program to discover if the arrows in Fig. 1 (and your sketch) that indicate how the stack and the heap grow are correct. **If they are not, redraw the arrows.**

3. Gathering address map information from Linux

Linux provides information about a process in the **`/proc`** “directory”; several files keep information about running processes – and you will need to add just one line to prevent the **`Lab-02a-32bit`** program from terminating when it reaches the last **`return`** statement (how do you do that?!)

For this exercise, it is easier if you have two terminal sessions; in one of them, do run the modified (b version) program,

```
$ ./Lab-02b-32bit
```

while in the other terminal you must find the PID (Process ID) of the running program; to get that information run

```
$ ps -ef | grep Lab-02b-32bit
```

The output should be similar to

```
fso       1630  1570  0 13:17 pts/0    00:00:00 ./Lab-02b-32bit
fso       1632  1596  0 13:18 pts/1    00:00:00 grep --color=…
```

we painted here the interesting PID with a green colour. Now run

```
$ pmap 1630
```

The output should be similar to:

```
1630:   ./Lab-02b-32bit
08048000      4K r-x-- Lab-02b-32bit
08049000      4K r---- Lab-02b-32bit
0804a000      4K rw--- Lab-02b-32bit
0920e000    132K rw---   [ anon ]
b1e44000  97664K rw---   [ anon ]
b7da4000   1728K r-x-- libc-2.23.so
b7f54000      8K r---- libc-2.23.so
b7f56000      4K rw--- libc-2.23.so
b7f57000     12K rw---   [ anon ]
b7f6c000      4K rw---   [ anon ]
b7f6d000     12K r----   [ anon ]
b7f70000      8K r-x--   [ anon ]
b7f72000    140K r-x-- ld-2.23.so
b7f95000      4K r---- ld-2.23.so
b7f96000      4K rw--- ld-2.23.so
bfddb000    132K rw---   [ stack ]
 total    99864K
```

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

```
08048000       4K r-x-- Lab-02b-32bit
```

starts at memory address 0x8048000 (that is, at decimal 134512640 or 128M), 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 **Lab-02b-32bit** file.

You may notice that **main** is at an address inside that region.

The second region starts at memory address 0x8049000, 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 it's address (`0x8048680`) we see that the **strC** is in the same region as the code!

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

```
0804a000       4K rw--- Lab-02b-32bit
```

region.

So why are most "structures" 4KB in size? Remember, from the AC course, that Intel/AMD architectures use a 4K pagesize by default – we are not yet studying pages in FSO, we're covering that later, but it's worth mentioning here…

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 **Lab-02b-32bit** executable file, but are dynamically created – one can see that the

```
b1e44000   97664K rw---    [ anon ]
```

region, is where the heap is located (we have allocated about 100MB – in fact, a 100e6 structure – and 97664 x 1024 = 100 007 936), a little big larger than $100x10^6$.

To finish things up, what about these **libc-2.23.so** regions, such as this one?

```
b7da4000    1728K r-x-- libc-2.23.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).

3.1. Modifying the strings…

Try to replace the first "M" character in strV and strC and print the new modified strings. If the compiler complains about trying to change strC, try to fool it, and execute the program. What happens? Why?

3.2. Discovering the addresses of C library functions

Now add some code to the program to discover the addresses of the printf and malloc functions; try to find in the pmap output where these two are located.

4. What about real addresses?

So far we have seen only the logical (also known as virtual) addresses; you have seen it in the AC course, and you know that a process' virtual address space is loaded into the computer's RAM, and accesses to virtual addresses are "transformed" into accesses to real addresses.

Program , shown in Listing 2, is able to show the real address for each virtual address.

4.1. The 32-bit (x86) base version showing virtual and real addresses

```
include <stdio.h>
#include <stdlib.h>

unsigned int v2p(unsigned int vaddr);

const char strC[]= "Memory map of a (Linux) process\n\n";

char strV[]= "Memory map of a (Linux) process\n\n";

int globalVar;
int globalVarInit= 1;

int main(int argc, char *argv[]) {

  printf("%s%s", strC, strV);

  printf("                                   virtual      real\n");

  printf("location of the code (main)       %p 0x%x\n", main, v2p( (unsigned int)main ) );
  printf("location of the code (main)       %u %u\n\n", (unsigned int)main, v2p( (unsigned int)main ) );

  printf("location of a (string) constant   %p 0x%x\n", strC, v2p( (unsigned int)strC ) );
  printf("location of a (string) constant   %u %u\n\n", (unsigned int)strC, v2p( (unsigned int)strC ) );

  printf("location of a (string) variable   %p 0x%x\n", strV, v2p( (unsigned int)strV ) );
  printf("location of a (string) variable   %u %u\n\n", (unsigned int)strV, v2p( (unsigned int)strV ) );

  printf("loc. of a global non-init'ed var  %p 0x%x\n", &globalVar, v2p( (unsigned int)&globalVar ) );
  printf("loc. of a global non-init'ed var  %u %u\n\n", (unsigned int)&globalVar, v2p( (unsigned int)&globalVar
) );

  printf("loc. of a global initialized var  %p 0x%x\n", &globalVarInit, v2p( (unsigned int)&globalVarInit ) );
  printf("loc. of a global initialized var  %u %u\n\n", (unsigned int)&globalVarInit, v2p( (unsigned
int)&globalVarInit ) );

  char *ptr = malloc(100e6);
  printf("location of the top of the heap  %p 0x%x\n", ptr, v2p( (unsigned int)ptr ) );
  printf("location of the top of the heap  %u %u\n\n", (unsigned int)ptr, v2p( (unsigned int)ptr ) );

  int x = 3;
  printf("location of the top of the stack %p 0x%x\n", &x, v2p( (unsigned int)&x ) );
  printf("location of the top of the stack %u %u\n\n", (unsigned int)&x, v2p( (unsigned int)&x ) );

  return 0;
}
```

Listing 2. The Lab-02c-32bit.c program

The "magic" of retrieving the real address from the virtual one is accomplished by the function v2p, that takes a virtual address as argument and returns the corresponding physical address. We have decided not to show the v2p code because, right now, it would be incomprehensible to you, so we supply it as a compiled object file, **v2p.o**. We will show it later when studying virtual memory through demand paging.

To compile the Lab-02c-32bit.c program (in a 32-bit Linux) you should write:

```
      $ gcc -Wall -o Lab-02c-32bit v2p.o Lab-02c-32bit.c
```

To run it, you must have a special permission to access information in the Linux kernel, and the simplest way to get it is to run as a superuser so, [Note: you can't become superuser in the DI Lab PCs]

```
      $ sudo ./Lab-02c-32bit
```

If your computer (or virtual machine 😊) has 2 GB of RAM, the output should be similar to:

```
      Memory map of a (Linux) process

      Memory map of a (Linux) process

                                         virtual      real
      location of the code (main)       0x804873b 0x6368d73b
      location of the code (main)       134514491 1667815227

      location of a (string) constant   0x8048aa0 0x6368daa0
      location of a (string) constant   134515360 1667816096

      location of a (string) variable   0x804a060 0x618eb060
      location of a (string) variable   134520928 1636741216

      loc. of a global non-init'ed var  0x804a08c 0x618eb08c
      loc. of a global non-init'ed var  134520972 1636741260

      loc. of a global initialized var  0x804a084 0x618eb084
      loc. of a global initialized var  134520964 1636741252

      location of the top of the heap  0xb1db1008 0x62572008
      location of the top of the heap  2983923720 1649877000

      location of the top of the stack 0xbff0ff94 0x62b96f94
      location of the top of the stack 3220242324 1656319892
```

As you can see, for each "interesting location" (code, variable, stack, etc.) we print the virtual and real addresses, both in hexadecimal and decimal bases. Now, try this: shutdown the VM, change it's (virtual) RAM size to 1 GB and repeat this exercise. Look at the real addresses.

4.2. Draw the RAM, record the real addresses, and connect them to the virtual addresses

Now, draw (sketch) another rectangle, **IN the same PAPER SHEET** as before, and next to the virtual memory map. The rectangle will represent the computer RAM. Record the physical addresses of main, the variables, the stack, the heap, etc. and draw arrows pointing from the virtual to the corresponding physical address.

4.3. Draw the RAM, record the real addresses, and connect them to the virtual addresses but with 2 lab-02c processes running

Run 2 terminal windows and, on each one, run an instance of lab-02c. Now, draw the RAM, the real addresses, the (virtual) address space map for each process and connect it to the RAM map.

4.4. TPC:

In your FSO study time (at home?) try to get, for each “object” (main, strC, strV, etc.) the start and end of the page(s) where the object exists, and draw them both in the logical address space map and in the physical RAM.

5. TPC: the 64-bit (x86-64) world

In your FSO study time (at home?) repeat the previous exercises – except the previous one, about real addresses – in a 64-bit Linux version; you can use the Lab PCs, or the virtual machine the FSO instructors have prepared for you.