Virtual Address Space - II

Fonaments dels Computadors – Grau en Intel·ligència Artificial – 2021-2022 Q1

Facultat d'Informàtica de Barcelona – Departament d'Arquitectura de Computadors

In order to complete the understanding of the process address space, in this session we focus our attention on its management, and how the application can allocate and deallocate memory. Additionally, we will learn to use a few operating system tools useful to understand the behaviour of programs.

The sample program

The matrix multiplication program used in this session is the same as in the previous one (S9), with a small change. Download the program set from:
http://docencia.ac.upc.edu/FIB/GIA/FC/documents/Lab/S10/S10-FC_GIA.tar.gz.

First, let's adapt the problem size, as we did in session S9. Look at the source code of "matmul.c". You will find this section of code at the beginning:

```
     // alternatives   1224x960   ->  40 MBytes RAM
     //                1224x9600  -> 220 MBytes RAM
     //                1224x96000 ->   2 GBytes RAM

#define ROWS_A 1224
#define COLS_A 96000    //  alternatives, set to 960, 9600, or 96000

#define ROWS_B COLS_A
#define COLS_B 412

#define ROWS_C ROWS_A
#define COLS_C COLS_B
```

Adjust the problem (matrix) size properly for your virtual machine. We have set 3 alternatives:

- If your virtual machine has 4 Gbytes or more RAM memory, you can use COLS_A set to 96000 elements
- If your virtual machine has between 1 and 4 Gbytes, you can use COLS_A set to 9600 elements
- If your virtual machine has less than 1 Gbyte, try setting COLS_A to 960 elements

  Try not to change the sizes ROWS_A or COLS_B, as the behaviour you will see in exercise 1 depends on C being of the proper size.

Compile the samples of the matrix multiplication example, and the display-maps support app:

```
        $ make all
```

This version of matmul has a small change with respect to the previous ones: it stops in specific places, and waits for you to press <Enter> before continuing. While stopped, you can follow the given instructions to display the details of its address space, and then continue with the execution, just by pressing <Enter>. As an example, execute the static version:

```
      $ ./matmul.static

      Starting the execution, please look at the memory regions, and
       record the size of the "heap" area,
      the process identifier is PID = 23793
          <<< press <Enter> to continue with the execution >>>
```

Observe that the process reports its PID (23793), so you can use another terminal to use “display-maps”, and see the memory regions of the “matmul.static” application while it is running:

```
$ ./display-maps  23793
 _________________________________________________________________________
|                                                                         |
|----0x0000000000400000---read---nowrite--noexec--private--0x0000001000----|
|       /mnt/downloads/Downloads/S10-FC_GIA/matmul.static (0x0000000000)    |
|----0x0000000000401000---read---nowrite---exec---private--0x0000096000----|
|                                                                         |
|                                                                         |
|       /mnt/downloads/Downloads/S10-FC_GIA/matmul.static (0x0000001000)    |
......
```

In the previous listing you can see each one of the regions of the process executing “matmul.static”, including the code, data, BSS, heap, and stack, as explained in the course.

There is a Linux tool showing also the memory regions of a process: “pmap”. You can also use it if you prefer. It has the inconvenient that it does not show explicitly the region with the “heap” name. It refers to it using the “anon” attribute, as with the other “anonymous” regions:

```
 $ pmap  23793
0000000000400000      4K r---- matmul.static
0000000000401000    600K r-x-- matmul.static
0000000000497000    156K r---- matmul.static
......
```

Management of the process virtual address space in C

On the matrix multiplication program (“matmul.static”), do the following exercises, and write your answers in the “*answers.txt*” file for this session:

Exercise 1

While the process is stopped for the first time as explained above, record the size of the “heap” region in your “answers.txt” file. Look at the source code and see that after this first stop, the program allocates matrix C:

```
C = matrix_allocate (“C”, ROWS_C, COLS_C);
```

This functions calls the C library function “malloc”. This request goes to the Linux operating system, and it makes the “heap” region to grow.

Exercise 2

Let the program to continue its execution by pressing “Enter”, and when it stops again, look again to the “heap” area, and determine if it has been enlarged. Record the size of the “heap” region in your “answers.txt” file. Also check that the address of C as reported by the program, belongs to the “heap” area, and explain this in your “answers.txt” file.

This is a message like the one you will get at the second stop:

```
C matrix has been allocated at address 0x4bd743a0 (check that
it corresponds to the "heap" area).
Please record the new size of the "heap" area again,
is it different? is it larger?
      <<< press <Enter> to continue with the execution >>>
```

Exercise 3

Let’s do the same experiment with the dynamic version of matmul. In this case you will need to be aware that there are a lot more memory regions, because of the dynamic libraries mapped onto the address space of the process. Nevertheless, you should only focus on the region labeled as “heap”.

```
$ ./matmul

Starting the execution, please look at the memory regions, and
 record the size of the “heap” area,
the process identifier is PID = 4592
    <<< press <Enter> to continue with the execution >>>

$ ./display-maps  4592
 __________________________________________________________________________
|                                                                          |
|----0x0000565550f07000---read---nowrite--noexec--private--0x0000001000----|
|              /mnt/downloads/Downloads/S10-FC_GIA/matmul (0x0000000000)   |
|----0x0000565550f08000---read---nowrite---exec---private--0x0000002000----|
|              /mnt/downloads/Downloads/S10-FC_GIA/matmul (0x0000001000)   |
......
```

While the process is stopped for the first time as explained above, record the size of the “heap” region in your “answers.txt” file.

Exercise 4

Let the program to continue its execution by pressing “Enter”, and when it stops again, look again to the “heap” area, and determine if it has been enlarged. As in Exercise 3, record the new “heap” size in your “answers.txt” file, and check that the address of C as reported by the program, belongs to the “heap” area. Comment about this in your “answers.txt” file.

Management of the process virtual address space in Python

Open four terminals: in the first one, launch “python”; in the second one, launch “ps -a”, get the PID of “python” and inspect the memory, in this terminal, dumping the contents of “pmap <PID>”; in the third one, inspect the contents of “/proc/<PID>/maps”; finally, in the fourth one, execute “top -p <PID>” to just show this single process as output of the “top” command.

Exercise 5

In the terminal where you are executing python, declare a variable that holds an array of 10M items (i.e. 10000000). We will do this following different approaches. After executing every approach, check how top changes the VIRT column (number of bytes of virtual memory used by the process), and check the pmap output, in particular checking the “anon” regions (“pmap <PID> | grep anon | more”) and write, in the “answers.txt” file what region you identify it is mapped the array and what changes you are able to find after the execution of every approach. We introduce “sys.getsizeof()” to show the size in bytes of every variable. (HINT: you can check in this link https://docs.python.org/3/library/array.html the typecodes of python to define the type of data to save in the arrays):

1) *Using array module*
    a. *vec1 = array.array(‘f’, [0 for x in range(10000000)])*
       sys.getsizeof(vec1)

b. vec2 = array.array('d', [0 for x in range(10000000)])
sys.getsizeof(vec2)

2) Using numpy module

a. vec3 = numpy.array([0 for x in range(10000000)], dtype='f')
sys.getsizeof(vec3)

b. vec4 = numpy.array([0 for x in range(10000000)], dtype='d')
sys.getsizeof(vec4)

3) Using lists

*a. vec5 = [0] * 10000000*
sys.getsizeof(vec5)

*b. vec6 = [0] * 10000000*
sys.getsizeof(vec6)

We strongly suggest you to keep declaring additional arrays till the amount of free memory (shown by "top" command) is getting exhausted. At that time, keep declaring additional arrays and check what is the behaviour of Mem and Swap free amounts of bytes are updated.

Exercise 6

In the first exercises we have played with matmul. We suggest you can play with the python implementations of matmul (with and without numpy) attached to this Lab session. Remember you and Before we play with the matmul code, pretty similar to the one from Session S8, you will practice with a simple introductory example. Remember that before running the python codes, you first have to execute the C implementation with "./matmul -s".

Use of local (stack) variables

As local variables are allocated by the compiler in the stack of the process, let's observe what happens when a recursive application has a large variable on the stack. We will use a Fibonacci example, computing the Fibonacci of numbers. Make sure the "fibonacci" program is compiled:

```
$ make fibonacci
```

Exercise 7

Execute the fibonacci sample, with a number as parameter. We will track how the stack region grows at each recursive invocation of the fibonacci function:

```
$ ./fibonacci  8
PID 5056: entering fibonacci(8)... unused array is at 0x7ffc26046c50
 <<< press <Enter> to continue with the execution >>>
```

While "fibonacci" is waiting for you to press "Enter", look at the memory regions, and record the "stack" size in your "answers.txt" file. Do this until the program ends (it may end with a "segmentation fault" error, due to a stack overflow).

Exercise 8

Look at the "fibonacci.c" program, and eliminate the use of the array "taula_gairebe_no_usada", the calls to "printf", "getchar", and "fflush". Compile the resulting program, and execute it with:

```
$ ./fibonacci  8
…
$ ./fibonacci  20
…
```

Now both tests should provide the correct result of the fibonacci number. You can find the series to check here: https://ca.wikipedia.org/wiki/Successió_de_Fibonacci. Comment your observations in your "answers.txt" file. In order to check that you understand the use of the stack region, comment also about:

- *Among these ones, what type of data is allocated in the stack?*
  - *Global data?*
  - *Local data?*
  - *Function arguments?*
- *Among these ones, what control data is also allocated in the stack?*
  - *Results of conditional sentences to decide if the program should execute the "then" or the "else" part of the conditional?*
  - *Whether a program loop should keep going or control should exit from the loop?*
  - *The return address when a new function is called?*

Upload the Deliverable

To collect all contributions to this deliverable, you can use the tar command as follows:

```
# tar  czvf   session10.tar.gz    answers.txt    *.c     Makefile
```

Now go to RACO and upload this recently created file to the corresponding session slot.