Operating Systems (2)

Computadors – Grau en Ciència i Enginyeria de Dades

Facultat d'Informàtica de Barcelona

The second Lab Session about OS is focused on memory management. In this case, it aims at getting experience with logical memory space and the required library support to dynamically allocate/deallocate memory. The session also includes an exercise to analyse the memory management performed using Python.

Firstly, to do this Session, please download the files set from the following link: http://docencia.ac.upc.edu/FIB/GCED/COM/documents/Lab/S9/FilesS9.tar.gz.

Even though there is no Makefile in this set of files, we strongly suggest you to create a Makefile to ease your compilations.

Memory Management in C

The code "mem-stack-orig.c" is developed in such a way you have to introduce some code to print the required memory addresses.

Exercise 1

*Copy the "mem-stack-orig.c" file and name the new file "**mem-stack.c**". Firstly, analyse and understand the code of the main and Recursivity functions. Then, introduce the missing code (indicated with comments in the sourcecode and labels like "AAA") to show the requested memory addresses.* ***NOTE:*** *check the Theory slides about pointers to find out how to correctly introduce the missing code.*

Exercise 2

*Compile "mem-stack.c" and execute it with a given input parameter to indicate the maximum level of recursivity (e.g. "#>./mem-stack 10", a low value to ease your analysis). Analyse the output and write in the "**answers.txt**" file your findings about the updates of the local variable "localVar" address when doing recursive function invocation. In particular, why the address is different*

In the "mem-heap-orig.c" file, you may find a code theat holds heap memory allocation. This code needs an integer as input parameter to indicate the number of integers we want to hold in the array pointed by "ptr". That is, how many integers we want to allocate in the Heap memory region.

Exercise 3

*In this exercise we deal with "mem-heap-orig.c". Analyse and understand the code of the main function. Launch multiple executions with different number as input parameter in the command line. Analyse the output, especially the size of the Heap memory region compared to the number of requested Bytes, and write in the "**answers.txt**" file your findings. In particular, why the region size is different than the requested Bytes and why the region size is the same in different cases.* ***NOTE:*** *we have setup the heap memory management to follow the explanations in Theory lectures, and prevent alternative management that goes beyond the scope of this course. To ease your analysis and understanding we* ***STRONGLY SUGGEST*** *to allocate memory for 1K, 10K, and 8M integers (i.e. 4KB, 40KB, and 32MB).*

Open different terminals to compare the memory addresses shown by the executions with the information shown by the OS. Firstly, execute the command "#>ps -a" to get the PID from the process you want to analyse. Then go to "/proc/<PID>/maps" (substitute "<PID>" with the corresponding PID shown by "ps"). "maps" is a text file, thus use a command like "more", "cat", etc to show its contents. For more information, go to "#>man proc" and, in particular, to the section "/proc/[pid]/maps". ***NOTE:*** *remember every single line is a different memory region.*

Pay special attention to:

- **First column:** start memory address of this particular region
- **Second column:** end memory address of this particular region
- **Third column:** permissions ("r" read; "w" write; "x" execute)
- **Last column:** path of the file related to the memory region.

NOTE: if the last column (the path) has no label it is difficult to know its purpose. It is beyond the scope of this course finding out its goal. Thus, you can ignore those regions.

Exercise 4

Repeat the executions from exercises 2 and 3. In every execution, and in particular in the checkpoints, find the mem address ranges for: heap, stack, code, and data. ***NOTE:*** *to identify the code and data regions of your code, analyse the permissions shown in the third column of those regions labeled with the name of the executable, and check your notes taken from the Theory lectures of this lesson. Write in the "**answers.txt**" file, the amount of memory (that is, end@ - start@) of every region. You can divide the number of bytes by 1024 to obtain the number of KBytes. Compare these address ranges with the addresses you have obtained as output in the executions. If there is any address that doesn't match to the corresponding region, check whether the code you have introduced and your calculations are correct. If you consider everything is correct, but there is no match, please, ask to the lecturer, since MAYBE there is an alternative management performed by the OS.*

Exercise 5

*Copy "mem-stack.c" to "**mem-stack-2.c**". The goal of this exercise is to understand the stack management. Execute the code, with no modifications, and measure the size (in Bytes, and KiloBytes) of the stack region in every recursivity call iteration. You can do it analysing the "maps" file. Then, modify the size of the "char buf[256]" array, declared in the function "Recursivity", up to 1000 chars. Execute again and double check whether the size varies or not. Finally, find out what size we have to use in the "buf" array to force stack region increases the size in every recursive iteration. Write in the "**answers.txt**" file what size forces to increase the stack region size, why and how you have found out this value.*

Prepare your desktop to have four terminals openned: in the first one, launch the required program indicated in the exercise; in the second one, launch "ps -a", get the PID of the process to inspect the memory dumping the contents of "pmap <PID>" (it is a command that simplifies the output of "maps" file, but the "heap" region is not properly identified in the reduced output shown by default); 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. ***NOTE:*** *from the "top" command analyse the VIRT column, that indicates the number of bytes of virtual memory used by the process.*

Exercise 6

*Copy the file "mem-heap-orig.c" and name it "**mem-heap-loop1.c**". Modify the code to introduce a loop that iterates 10 times to allocate memory, but without free memory inside the loop. It doesn't matter malloc overwrites the ptr in every iteration. **IMPORTANT:** include the invocation to "read" function to temporary pause the execution after every checkpoint to analyse the memory. Launch 2 executions with two different input parameter values: one with a very low value; and one with a very large value. Analyse the output and write in the "**answers.txt**" how the Heap memory region changes and how it matches with the explanations done in the Theory lectures of this lesson.*

Exercise 7

*Copy the file "mem-heap-loop1.c" and name it "**mem-heap-loop2.c**". Modify the code to introduce the free inside the loop. Repeat the experiment with the same input parameter values of the previous exercise. Analyse the output and write in the "**answers.txt**" the difference you may see in the Heap memory region size changes.*

Exercise 8

*Recompile the file "mem-heap-orig.c" in order to statically link the libraries. Check past lab sessions to find out what flag you have to introduce in the command line to properly compile the binary. Write in the "**answers.txt**" file the command line you have used to do such compilation. Besides, indicate in the "**answers.txt**" file the differences you may identify if you compare the output of the "/proc/<PID>/maps", in the first checkpoint, between the execution of both binary versions (statically vs dynamically linked). **NOTE:** check the slides from the Theory about this.*

Double check Python environment

Open a terminal and double check your python interpreter is ready to fulfill the following exercise. To do so, firstly, perform a test executing the following command lines:

a.1) #>python3
a.2) #>import numpy

If you get an error message like "*ImportError: No module named numpy*", you need to install few items to Python. Otherwise, congrats! your Python environment is ready and you can directly go to the next Section, called *"Management of the process virtual address space in Python"*.

To install the required components, please, execute the following command lines (*NOTE: before doing so, exit from python with "exit()"*):

1) #> sudo apt-get update
   NOTE: it updates repositories to download and install software
2) #>sudo apt-get install python3-pip
   NOTE: pip is Python's Package manager
3) #>pip install numpy
   NOTE: it installs the Package "numpy" for Python

If you have successfully executed (i.e. with no errors) these command lines, Python should be ready. To double check it, execute the two command lines of the beggining of this Section (i.e. "a.1" and "a.2"). If you get again the error mentioned before, please, check whether you got an error executing the required command lines to install the "numpy" module, just in case you need something else, and do it again.

Memory Management in Python

Prepare the desktop to use the four terminals indicated before. In this case get the PID of the process that is running "./python".

Exercise 9

In the terminal where you are executing python, declare a variable that holds an array of 8M items (i.e. 8000000). We will do this following different approaches to analyse the impact on memory management. In every approach check the memory when indicated, as follows: check how top changes the VIRT column (number of bytes of virtual memory used by the process), the pmap output as well as "/proc/<PID>/maps" file. In the latter, ***pay special attention to the "heap" region and any region that has significantly changed after creating the vector, even though it is an anonymous region (no label) in "maps" file or "anon" region in the pmap output.***

Please, execute "#>python" at the beggining and "exit()" at the end of every approach to guarantee doing the analysis in a new clean environment.

*Write, in the "****answers.txt****" file, what region/s has/have significantly increased the size, whether the size is the one you expect from the "sys.getsizeof" execution (it shows the size in bytes of every variable).* ***NOTE:*** *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*

```
import array
import sys
//CHECK MEMORY DATA
vec1 = array.array('f', [0 for x in range(8000000)])
sys.getsizeof(vec1)
//CHECK MEMORY DATA
vec2 = array.array('d', [0 for x in range(8000000)])
sys.getsizeof(vec2)
//CHECK MEMORY DATA
```

2) *Using numpy module*

```
import numpy
import sys
//CHECK MEMORY DATA
vec3 = numpy.array( [0 for x in range(8000000)], dtype='f')
sys.getsizeof(vec3)
//CHECK MEMORY DATA
vec4 = numpy.array( [0 for x in range(8000000)], dtype='d')
sys.getsizeof(vec4)
//CHECK MEMORY DATA
```

3) *Using lists*

```
import sys
//CHECK MEMORY DATA
vec5 = [0] * 8000000
sys.getsizeof(vec5)
//CHECK MEMORY DATA
vec6 = [0] * 8000000
sys.getsizeof(vec6)
//CHECK MEMORY DATA
```

Upload the Deliverable

To save the changes you can use the tar command as follows:

```
#tar czvf session9.tar.gz answers.txt  *.c  Makefile
```

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