Operating Systems 3. Memory

This practical activity aims to familiarize the student with the routines of memory management in UNIX/Linux systems and the information provided by the operating system concerning the memory usage of running processes. After its completion, the student should understand the fundamentals of the Operating System in relation to memory management and virtual memory.

Suggested Shell commands: *man, ps, top, vmstat, perf, fg, bg, &, hexdump,...*
Suggested C Functions: *malloc, free, rand, sizeof, getpagesize, sysconf, fread, fwrite...*
Include (header files): *<stdlib.h> <unistd.h> <stdio.h>*

*** Note: Find out how these commands and C functions will help you in the development of this practical activity.**

1. Memory Management

Remember that memory addresses of a running process will always be virtual. The Operating System manages and protects physical memory addresses. As far as we are concerned, the operating system makes use of paging for the virtualization of memory.

As in the previous practical activity, the student must remember that it is possible to obtain information about running processes, including the use of memory, with commands such as ps or top, as well as in the corresponding files in the **/proc/<PID>/** directory.

2. Memory Management through C API.

In this practical activity, we will design and develop a C program able create an NxM matrix dynamically, filled with random floating numbers.

The idea is to use the memory management tools (`malloc/free`), for the creation/destruction of the matrix, given the size entered in the command line (Eg "`./executable <numfiles> <numcolumns>` "). Pointers should be used to assign the value of the elements of the matrix (base pointer + offset), assigning a random floating value.

The program, after filling the matrix, must wait for an interactive command before freeing memory or exiting. The list of possible commands are:

- D (Stores the matrix to disk in a binary file with the name "matrix")*

The file will have binary format with the following structure:
[MagicNumber**][NumberOfRows][NumberOfColumns][Element1,1][Element1,2]...[Element1,M]...Element2,1][Element2,2]...[ElementN,M].

- T (Prints in the terminal the total size of the array in bytes)
- L (Frees the matrix memory)
- S (Exits the program)***

* *You can check the contents of a binary file using `hexdump`.*
** *MagicNumber is just an integer at the student's choice that marks the file as created with the program.*
*** *The program should not finish the execution until the command "S" is entered.*

1) Executing the program with a matrix size of 12000x12000, while the process is waiting for the command, check in the file **/proc/<PID>/status** the counters associated with the memory management. (**Note: VmXXXX**)
How much memory does the process use? How much does the process stack occupy? How much does the process code segment occupy? How much SWAP space does the process use?

2) Do the same execution making use of the **`perf`** tool.

```
#perf stat -e page-faults ./programa 12000 12000
(Note: you may need to install it) How many page faults occurred?
```

3) Repeat the previous execution but twice (two copies executing in background). How many page faults occurred this time? What about the SWAP memory? Explain the differences with the previous execution (single).

4) Add a new command to the program (M) that prints on the terminal one address from the code segment, one address from the heap and one address from the stack. (**Note: &**) Are these addresses physical or virtual? Try executing it several times.

The Operating System has a protection mechanism that causes the data segments (heap and stack) to have random addresses within the address space. We are going to disable this protection by executing the following command as root:

```
echo 0 | tee /proc/sys/kernel/randomize_va_space
```

If the value stored in "`randomize_va_space`" is 2 (by default), randomization of the address space occurs. With the previous command, we replace it with 0, eliminating the protection. What happens now with the addresses of the different segments? Try executing it several times, even simultaneously.

5) Write a program that can read the contents of the previously generated binary file and load it into memory showing on the screen the number of rows, the number of columns and the size of the matrix. (Note: check MagicNumber).

6) Change the writing format of the binary file to ascii and check the size in memory and on disk of both formats.

3. Obtaining the TLB size

Along with this guide, the practical activity contains the program `tlb.c`. This program allows us to calculate the number of entries available in our tlb. Read the code trying to understand its behavior and compile it.
Execute the program with input parameters that vary from 2 to 4096 in powers of 2 (2, 4, 8, 16,..., 1024, 2048, 4096). Perform the executions without any other job running in the system that may disturb the measurements and write down the average access time returned by the program. How many entries do you think the TLB has in our system? Explain why you think there are differences in the returned values for average time for the different executions.

4. Creating a Makefile

Define a basic **Makefile** file to help you with the compilation tasks.
The file should facilitate changing the compilation options, and should delete all the objects and compiled files through the clean command.