# Lab 3: Processes virtual memory in Linux

Master M1 MOSIG – Université Grenoble Alpes & Grenoble INP

2018

In this lab, we are going to observe the virtual address space of processes in Linux through the implementation of simple test programs.

## 1 Instructions

**This lab is not graded. Please take the time to understand the results observed during the suggested tests, and try to answer the corresponding questions. Feel free to design your own test programs if you think additional experiments can improve your understanding. Also, do not hesitate to look for additional resources and explanations on the web. Answers to the questions (at least partial) will be published in one week.**

Some questions of the lab are marked as *Bonus*. We consider these questions difficult or less important. Focus on these questions only if you progress fast enough.

This lab is supposed to be run on a 64-bit Linux kernel (on a x86-64 machine). The command "`uname -a`" can be run to check information about the kernel.

## 2 About the proc file system

In a Linux system, the `/proc` directory contains a hierarchy of special files which represent the current state of the kernel.

Among the information that can be found in `/proc`, information about the current state of a process `PID` can be found in the `/proc/PID` directory. For a detailed description of all the information that can be found in `/proc`, we refer you to the man-pages: `man 5 proc`.

Among the virtual files available in `/proc`, we will focus on the information provided by `/proc/PID/maps`. This file describes the state of the virtual address space of a process.

## 3 Observing processes virtual address space layout

The command `cat` can be used to display the content of a file. Running "`cat /proc/self/maps`" displays information about the virtual address space of the current process.

**Question 3.1:** *Which process does the displayed information refer to in this case?*

**Question 3.2:** *We can see that each line of the `maps` file includes several fields. What does each of these fields correspond to?*

**Question 3.3:** *Observe the position of the stack and the heap in the virtual address space. Based on what you know about their internal details, explain why they are positioned this way.*

Let us now compare the virtual address space of two processes.

**Question 3.4:** *What differences do you notice between two processes that are run with the exact same parameters?*

**Question 3.5:** *There is a good reason for these differences. Explain.*

**Bonus question 1:** *We can observe that the executable file executed by a process is mapped several times at the beginning of the virtual address space. What do these different mappings correspond to (remember "Lecture 1: Introduction")?*

**Bonus question 2:** *What differences do you observe between these mappings? Does this match with the answer to the previous question? Try to explain.*

We are now going to think about the kernel memory space.

**Bonus question 3:** *Based on your observation of the address space of a process, figure out where is the virtual address space of the kernel.*

**Bonus question 4:** *Why does it belong to the virtual address space of the user although the user does not get access to it?*

## 4 Testing processes virtual address space layout

In this part of the lab, you are asked to write small programs that allocate or `mmap` some memory to observe what happens in the virtual address space.

## 4.1 Using malloc()

First, we would like to see what happens when we dynamically allocate memory using `malloc`.

We provide you with a simple program to run tests for the first question below (`test_malloc.c`). Note that this program uses `getpid()` to obtain the `pid` of the running process. It allows us to know which entry in `/proc` to look at. It also uses a call to `fread` (`man 3 fread`) on `stdin` to prevent the program from terminating before we have had time to observe the process address space.

**Question 4.6:** *In which part of the virtual address space has the memory block been allocated?*

Now, write a program that allocates many small memory blocks (few tens of KB).

**Question 4.7:** *Where are the blocks allocated in this case?*

Finally, write a program that allocates a large memory block (a few tens of MB)

**Question 4.8:** *Where is the block allocated in this case?*

**Question 4.9:** *Based on the previous observations, explain how the heap is managed by the operating system.*

## 4.2 Using mmap()

In this part, we are going to use the `mmap` system call (see "`man 2 mmap`") to map files in the virtual address space of processes. As we will see, among the capabilites offered by this system call, it allows a process to read and write into a file through virtual memory accesses.

Mappings can be *shared* or *private*. Write a program that maps portions of a file into memory: Create at least one shared and one private mapping. Note that we provide you with the code of a basic program (`test_mmap.c`) that does a *private* mapping.

**Question 4.10:** *Find the corresponding entries in `/proc/PID/maps`. How can we know if a mapping is shared or private?*

Mappings can also be anonymous. See `MAP_ANONYMOUS` in the manpage of `mmap`. Write another program that maps two anonymous regions, one private and one shared.

**Question 4.11:** *How do we know in `/proc/PID/maps` that an area is anonymous?*

**Question 4.12:** *In which context can an anonymous private memory mapping be useful?*

**Question 4.13:** *In which context can an anonymous shared memory mapping be useful?*

**Bonus question 5:** *We would like to evaluate approximately the total size of the virtual address space of a process. To do so, we are going to map private anonymous regions.*

*Write a program that keeps mapping new large private anonymous regions until the system refuses to map a new region.*

*According to this test, what is the size of the virtual address space of the process?*

The flag `MAP_NORESERVE` can be useful for our evaluation to be correct.

**Bonus question 6:** *Explain what changes when `mmap` is called using the flag `MAP_NORESERVE`.*

Modify the previous program to use the flag `MAP_NORESERVE` to re-evaluate the total size of the virtual address space of the process[1].

**Bonus question 7:** *Depending on your system, you might observe the same or a different result with and without the `MAP_NORESERVE` flag. In both cases, explain the result.*

**Bonus question 8:** *Considering that we are running on a 64-bit architecture, is the maximum size you measured the one that is expected? What is the explanation?*

To finish with this part, we are going to try to map at specific memory addresses.

First write a program that tries to `mmap` a page[2] at address `0x1000`.

**Bonus question 9:** *At what address has the page actually been mapped?*

**Bonus question 10:** *In your opinion, what is the reason for the behavior you observed?*

**Bonus question 11:** *Make another try but, this time, we would like to map a page at the very beginning of the virtual address space. What address should be theoretically given as argument to `mmap` to do so?*

**Bonus question 12:** *Run the new test, and explain what you observe.*

## 5 Manipulating mapped files

In this part, we are going to read and write to a file mapped into the address space of the process.

The very first step is to create the file we are going to `mmap`. We would like this file to have a size of 1 MB and to be created in the `/tmp` directory. For this, you can use the command `dd` in the following way.

[1]Note that `NORESERVE` is the default policy on most recent operating systems. As such, setting it explicitly might not make any difference.

[2]The size of a page can be obtained thanks to function `getpagesize()` (`man 2 getpagesize`).

```
$ dd if=/dev/zero of=/tmp/my_file count=1024 bs=1024
```

This command is going to fill the file `/tmp/my_file` with the `NULL` character (`0x00`). Do not forget to run this command again before every new tests to erase modifications to the file.

To get more familiar with `mmap`, we propose you to write a program that is going to map the file previously created into memory, and then, to write to the file through this memory mapping. More precisely:

- Implement a program that writes 50 times the character `'a'` starting at offset 50 in the file.

Open the file with your favorite text editor to verify that the file has correctly been modified.

**Question 5.14:** *Which flags should be used to ensure that `mmap` propagates the modifications to the file?*

Now, we would like to better understand the differences between a shared and a private mapping.

Write a program that maps 3 times the file into the memory of a process with a shared mapping. Modify the content of the mapping through 2 of the mappings, and read the modified content through the third mapping.

**Question 5.15:** *What do you observe?*

Modify the previous program to have now one private mapping. Your program should first write using the shared mapping, then using the private mapping, and finally read the modifications through the last shared mapping.

**Question 5.16:** *What do you observe? Explain.*

**Bonus question 13:** *On the same line as the previous tests, try to design a test program that shows virtual memory management is done at the granularity of a page.*

## 6 Page faults (Bonus)

In this last part, we would like to observe the page faults generated by a program accessing a mapped file.

Write a program that maps a 1MB file into memory with a shared mapping. The program will then read and write into the file. Additionally the program should take as input a parameter that defines at which step the program terminates. The steps to consider are the following ones:

- The program has just started.
- The program opens the file to be mapped into memory.

- The program maps the file into memory.
- the program reads from the file: 1 byte, then 1 page, then 1MB.
- the program writes into the file: 1 byte, then 1 page, then 1MB.

To get the number of page faults generate by the program, you can use the tool `/usr/bin/time` (with option `-v` for a nice formatting of the output)[3].

**Question 6.17:** *How many page faults are generated at each step of the program? Try to explain the numbers.*

[3]Simply calling `time` in a terminal might not launch the right tool, use the absolute path.