CS342 Operating Systems – Spring 2020
Project 3: Virtual Memory of a Process

Assigned: April 21, 2020
Due date: May 5, 2020. Time: 23:55 *Document Version*: 1.1

*You will do this project **individually**. You have to program in C and Linux. We will do our tests in the following distribution of Linux: **Ubuntu 18 LTS – 64 bit**.*

In this project you will write a program **app** and see its virtual memory (VM) usage. You will write the application using three source files (three modules): module1.c, module2.c and module3.c. The module1.c will contain the main() function. The other files (modules) will contain some other functions and variables. It is up to you what to put into those, but you should achieve the requested memory usage behavior below.

You can compile the program with such a **Makefile** content (you can modify):

```
all: app module1.o module2.o module3.o

module1.o: module1.c
	gcc -c -no-pie module1.c
module2.o: module2.c
	gcc -c –no-pie module2.c
module3.o: module2.c
	gcc –c –no-pie module3.c

app: module1.o module2.o module3.o
	gcc -g –Wall –no-pie –o app module1.o module2.o module3.o
```

The program will be invoked as follows:

```
./app
```

As a result of the compilation using the Makefile, you will obtain an **executable** [**object**] **file** (app), and three [**relocabtable**] **object files** (module1.o, module2.o, module3.o). The C compiler (**cc**) compiles each source file first into a relocatable object file. Then the linker (**ld**) links those object files into a single executable object file (app). The **–c option** of gcc causes only the compiler to run to produce a relocatable object file. If we omit the -c option, gcc runs the linker as well, after compilation, to generate an executable object file. **Linking** is described very nicely in Chapter 7 [C7-CSapp] of [CSapp].

The format of the object and executable files for Unix and Linux systems is called **ELF** (**executable and linkable format**) [wElf, Elf, Elf64].

You will analyze the object files and the executable file using **objdump** utility. You will see the runtime virtual memory usage of the application using **pmap** utility. You will see the addresses of various program elements at run-time by **printing the addresses** (i.e., pointer variable values) to the screen. You will compare the various addresses and address ranges that you see in the executable file's dump, in the pmap output and the output of the program.

You will put everything, the program (source codes), its various outputs, the outputs of utilities (objdump, pmap), and your descriptions and interpretations into a report (i.e. document them) that will be submitted as a **pdf** file. You will also upload the program sources and a Makefile. We should be able to just type make and obtain the object files. All these files (pdf report, README, Makefile, program sources) will be put into a folder, tarred and gziped and submitted as a single file in Moodle.

Compile the program using **–no-pie** option of **gcc** compiler driver (so that address randomization is not used).

In the application you will define some **global** variables, some of which are **initialized**, some of which are not **initialized**. The initialized data should be at least 128 bytes long. The unutilized data size (at run time) should be at least 16 KB long. When the program is started, it will start with such data, without heap and stack yet.

You program should run in **steps**. When started, it is in step 1. In each step you will do the **required things** described below. Your program should wait between the steps for you to do these things. This waiting will be achieved in the program by waiting the user to type character **'n'** (meaning **next**). When you type 'n', the program will go to the next step. To cause the program to wait until you press 'n', you can use **sprintf**() or **getchar**() functions. Below, we will indicate when the program should go to the **next step**.

>>> **At step 1**, i.e., when the program is started, the program will just have initialized and uninitialized global variables allocated space in its **virtual memory**. OS might have also allocated some physical frames, but we are concerned in this project with the virtual memory of a program. Your program should print out (using **printf()**) the addresses of all the global variables (data) and functions (code), including the main() function, defined in your program to screen so that you see the addresses (i.e., pointer values).

In your program, you can use **printf ("%lx",** (void *) p) to print an **address** (**value of a pointer** p) to screen in hexadecimal format (pointer value is an **unsigned long integer - 64** bits long).

See the program output at this step. Also run the pmap utility, and see the virtual memory usage of the program in the output of the pmap utility. Save (document into report) these outputs (you will put it into your report). Compare the range of virtual memory used by data and code (text) segments of the program with the pointer values that you see on the screen. Explain and discuss.

Now look into the executable file with the **objdump** utility**.** Use the **-dx** option (objdump -dx app). Save the output. Find the addresses of your global variables and functions in the objdump output. Compare them with the screen output of the program and the pmaps output. Discuss.

>>> Now look into the object files (**module1.o** and **module2.o**) using the objdump utiliy. You need to type: objdump –dx module1.o to see the binary content of *relocatable* object file module1.o. You will analyze module2.o similarly. Look to the

addresses of the functions and global variables that you defined in these modules. Then look to their addresses in the executable file dump. Compare them and interpret them. What has happened. Why a module module1.o (or module2.o) is called **relocatable object module**, but the module app is called **executable** object module?

>>> Find some **references** in the code part (text segment) of the program (app) to some of those global variables and functions. For example, call instruction makes a reference to a function. Similarly, move instruction can make a reference to a global variable. See and document what these references are (i.e., the hex values). Find some references to the same global variables or functions in the relocatable object files (module1.o, module2.o, module3.o). Compare the references in the executable object file and in the relocatable object files. Document your findings.

>>> Find some references in the executable object file and the in the relocatable object files to some **standard C library** routines like **printf**(). How do they look like. Document your finding. What is the address of printf() routine at run time while your program is running? You can print out the address of printf() in your program. Not that by default, standard C library is **dynamically linked** into a program. Hence **shared** version of the standard C library (**libc.so**) is linked at run time with the program.

>>> Learn the pid of the program app using the **ps** command: ps aux app. Then change into directory **/proc/**pid. What do you see there? Which files are related with memory management. See their contents using the **cat** command: cat filename. Document your findings.

Now, go the next step. That means you type 'n' at the **terminal** the program is running, and the program will get 'n' as input and proceed to the next step.

>>> At **step 2,** your program will allocate memory for dynamic data (**objects**) using **malloc**(). As a result, the **heap** segment of the program (in virtual memory of the program) will grow to store these. The program will **dynamically allocate** memory with multiple malloc() calls so that the total dynamically allocated (virtual) memory for the program is at least 2 MB (i.e., heap size become at least 2MB). The program will print the addresses of some dynamically created objects to screen. Addresses are pointer values of the pointers pointing to dynamically allocated space (i.e., to dynamically created objects). A pointer is 8 bytes long in a 64 bit machine.

Look to the output of pmap again (run pmap again).

Compare the output of pmap with the output of your app program.

>>> Find out the **physical memory** usage of your program at this moment by using tools such as **ps**, **top**, **vmstat**, or **cat /proc/pid/meminfo** (pid is the process id of the process). Compare the physical memory usage with virtual memory usage. VM usage can be found from the **pmap** output.

Now, go to the next step (i.e., type 'n').

>>> At **step 3**, your program will grow the **stack** by calling a recursive function many times. That function can be in any one of your modules. Increase in the stack should be at least 100 KB. Program will print the addresses of some local variables (pointer values) to the screen to see what they are.

Look to the pmap output again (run pmap again). Compare the pointer values (addresses) with the stack range in the pmap ouput (note that pmap has several options; you can use these options to get more info in the output).

Go to the next step.

>>> At **step 4**, the program will use **mmap**() system call to map a file of size at least 1 MB into the virtual memory of the process. Start address where the file is mapped will be printed out.

Look to the start address that is printed out. Run pmap again and see its output. Compare them. Which part in the pmap output is for memory mapped file?

Look to the physical memory usage of the program. Compare with virtual memory usage. Document it.

Go to the next step.

>> At **step 5**, program will read/write some section of the mapped memory region (i.e., file).

See the physical memory usage at this time. Document it.

>>> At this step, also look to the output of the pmap and see the segments (virtual segments) of the program. Try to explain what each is containing. Find the total data segment size (globals + heap) in number of pages. Find the total code size in number of pages. Learn the page size using a **getconf** PAGESIZE command. Do the same for other segments. Document your findings.

>>> At **step 6**, your program will print the **content of the page** corresponding to the main function address.

Compare the output with the objdump output (main function there). Document and discuss.

Stop the program.

It is totally up to you what to put into your program to see all these. You can define some functions and (global) variables in module1.c, module2.c, and module3.c. There is no restriction in this part. If programs are written independently, there is no chance that they will be similar more than the acceptable threshold.

>>> Look to the size of the program on disk. Now compile the program this time using **–static** option of the **gcc** compiler driver. Look to the size of the program again. Do objdump –dx on the executable file. What do you see. Compare with the objdump

output of the program without getting compiled with static option. Document your findings.

Include all outputs and your explanations, discussions into your report.

Submission

Put all your files into a project directory named with your ID (one of the IDs of team members for team projects), tar the directory (using **tar xvf**), zip it (using **gzip**) and upload it to Moodle. For example, a student with ID 20140013 will create a directory named 20140013, will put the files there, tar and gzip the directory and upload the file. The uploaded file will be 20140013.tar.gz. Include a **README.txt** file as part of your upload (including the name and ID of the student, at least – for team projects, names and IDs of all members will be included). Include also a **Makefile** to compile your programs. We want to type just **make** and obtain the executable and object files.

References

[CSapp] Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e). Randal E. Bryant and David R. O'Hallaron. 3rd edition. 2016. Chapter 7: Linking.

[C7-CSapp]. Chapter 7 preview of [CSapp]. URL: http://csapp.cs.cmu.edu/2e/ch7-preview.pdf.

[wElf] Executable and Linkable Format. URL: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[Elf] Executable and Linkable Format. URL: URL: https://www.cs.cmu.edu/afs/cs/academic/class/15213-f00/docs/elf.pdf.

[Elf64] ELF-64 Object File Format. URL: https://uclibc.org/docs/elf-64-gen.pdf.

Additional Information and Clarifications

- This project does not require much coding. It requires quite a bit reading, self-learning, trying, and documenting.
- Other tools related to object file analysis are **nm** and **readelf**. Objdump is the most sophisticated one.
- You can use manual pages of Linux to get information about objdump, pmap, etc. Type: **man** objdump. There is also a lot of information in Internet.