# PA1: Address Spaces and Resource Usage Monitoring

CMPSC 473, Fall 2024

Release: Sep 10, 2024. Due: Sep 26, 2024 11:59:59pm

## 1 Goals

This programming assignment(PA1) has three main goals. First, you will learn in concrete terms what the different segments of a virtual address space(or simply address space) are. Second, you will learn to use three Linux facilities - `/proc`, `strace`, and `getrusage` - to record/measure certain aspects of process resource usage. Finally, we hope that PA1 will serve to refresh your understanding of(or make you learn in case you lack it) :(i) logging into a Linux machine(`ssh`, `VPN`, `2FA`) and using basic Linux shell and commands,(ii) compiling a C program(`gcc`, `make`) , creating/linking against libraries,(iii) debugging(`gdb`) and(v) using Linux man pages(the `man` command) . All of these are good programming and experimental analysis practices that we would like you to follow throughout this course(and beyond it).

For PA1, you are welcome to do your own research and use the internet for your research as long as you acknowledge them (in the essay type questions in the Canvas quiz). However, you are not allowed to discuss with anyone else except your project partner and the CMPSC 473 staff. You're welcome to refer to the cheatsheet at the end of this document and a real time version of it at Cheatsheet.

## 2 Getting Started

**Project group:** You are welcome to create project groups of up to 2 members. You can work solo as well. Only one member from a project group will submit the Canvas quiz. Include your project partner's psu ID in the appropriate field in the Canvas quiz.

**Necessary files:** Download the zip called "PA1" from canvas. This contains all the files needed for completing PA1. When you open this folder, you will find 4 folders, named prog1, ..., prog4. These folders contain the files described in Section 3. As mentioned in class, you may do the bulk of your work on any Linux(virtual) machine of your choosing. However, the results that you will report(which will be used for grading your work) must be carried out on CSE department's Linux-based teaching machines. These machines are named `e5-cse-135-xx.cse.psu.edu`(where xx is a machine number, xx $\in$ [01, 28]) . **(The TAs have not been able to access machine 05. Please do not use that machine.)** The reason for asking you to report results on these machines is to have relative consistency/uniformity in your measurements - this would help us grade in a consistent manner and identify possible bugs/shortcomings. To copy the project folder from your local computer to the department's Linux machine, you can run the following command:

```
$ scp -r source destination
```

where the `source` is the path to the project folder in your local machine and the `destination` might look something like this: `psuid@e5-cse-135-xx.cse.psu.edu:/destinationPathToYourProjectFolder`

**Before starting:** Run the command `./project1.sh` in the PA1 folder before doing the project. Once you run this command, your terminal will say `[Project1]$:` instead of something like `e5-cse-135-xx.cse.psu.edu 1%`. You can exit this mode by typing `exit`. You will have to repeat this step every time you start a new terminal session. **Only run your code if you see the text `[Project1]$:` on your terminal**. This ensures that Address Space Layout Randomization(ASLR) is disabled and will make your answers consistent.

**Report your answers:** You will report your answers on the Canvas quiz named Project 1 Quiz. The canvas quiz contains the same questions as this project description. Some of the questions require getting an answer by compiling and running the appropriate executable while the rest involve doing some research and reading up on `man` pages. Additionally, you can refer to the Cheatsheet at the end of the document or in this link.

## 3 Description of Tasks

1. **Stack, heap, and system calls:** The executable named `prog1` contains a function that is recursively called 10 times. This function has a local variable and a dynamically allocated variable. Upon each invocation, the function displays the addresses of the newly allocated variables on the console. After 10 invocations, the program waits for a key to be pressed on the keyboard before concluding. We would like you to observe the addresses displayed by `prog1` and answer the following:

   i. Which addresses are for the local variables and which ones are for the dynamically allocated variables? (0.5)
   ii. What are the directions in which the stack and the heap grow on your system? (0.5)
   iii. What is the size of the process stack in KB when it is waiting for user input? (1)
   iv. What is the size of the process heap in KB when it is waiting for user input? Do not include anonymous regions. (1)
   v. What are the address limits of the stack and the heap? (1)
   vi. Use the `strace` command to record the system calls invoked while `prog1` executes and use man pages to learn basic information about each of these system calls. Based on the information from the `man` pages, match each system call with the purpose they serve in this program(refer to Canvas quiz) . (1)

   Hints:

   (iii/iv) Use the contents of `/proc/PID/smaps` that the `/proc` file system maintains for this process where we are denoting its process ID by `PID`. While the program waits for a user input, try running `ps -ef | grep prog1`. This will give you `PID`. You can then look at the `smaps` entry for this process(`cat /proc/PID/smaps`) to see a description of the current memory allocation to each segment of the process address space.

   (v) Use the maps entry within the `/proc` filesystem for this process. This will show all the starting and ending addresses assigned to each segment of virtual memory of a process.

2. **Debugging refresher:** The program `prog2.c` calls a recursive function which has a local and a dynamically allocated variable. Unlike the last time, however, this program will crash due a bug we have introduced into it. Use the `Makefile` that we have provided to compile the program. Execute it. The program will exit with an error printed on the console. You are to compile the program with **64 bit** option and answer the following:

   i. Use `gdb` to find the program statement that caused the error. (1)
   ii. Explain the cause of this error. Support your claim with address limits found from `/proc`. (2)
   iii. Using `gdb` back trace the stack. Examine individual frames in the stack to find each frame's size to be ______ bytes. Combining this with your knowledge(or estimate) of the sizes of other address space components to determine that ______ invocations of the recursive function should be possible on your system. However, ______ invocations occur when you actually execute the program. (1)
   iv. Which of the following are generally contents of a frame? Select all that are possible(refer to Canvas quiz) . (1)

3. **More debugging:** Consider the program prog3.c. It calls a recursive function which has a local and a dynamically allocated variable. Like the last time, this program will crash due to a bug that we have introduced in it. Use the provided Makefile to compile the program. Create a **32 bit** executable. Upon executing, you will see an error on the console before the program terminates. Answer the following questions:

   i. What is the cause of the error? Use `valgrind`. (1)
   ii. Which program statement(s) is causing the error(s)? (1)
   iii. Validate this alleged cause with address space related information from using `valgrind/gdb` or `/proc`. (2)

iv. How is the error in `prog3` different than the one for `prog2`? (1)

4. **And some more:** The program prog4.c may seem to be error-free. But when executing under `valgrind`, you will see many errors. Answer the following questions:

   i. We find the following errors in prog4 by running `valgrind`(Refer to canvas quiz). (1)

   ii. How would you solve the error(s) detected by `valgrind`. Implement in code and write only the code snippets for function(s) in `prog4.c` that you edited in the following textbox. (2)

   iii. Modify the program to use `getrusage` for measuring the following. Run the program three times and report all three(comma separated) values for each one:(i) user CPU time used: ______(ii) system CPU time used: ______ (0.25)

   iv. True/False: Time taken by system calls is included in system CPU time used. (0.25)

   v. Modify the program to use `getrusage` for measuring the following. Run the program three times and report all three(comma separated) values for each one: (iv) voluntary context switches: ______,(v) involuntary context switches: ______ (0.25)

   vi. Describe the difference between voluntary and involuntary context switches. (0.25)

   vii. Modify the program to use `getrusage` and report the value of maximum resident set size in kilobytes before(or after?) the fix. (0.5)

   viii. What is the maximum resident set size? (0.5)

## 4 Submission and Grading

Submit the Canvas quiz names Project 1 Quiz with the answers found by running the appropriate code/commands. Only 1 member of the project group should submit the quiz. If you have a project partner, **include their PSU ID** in the appropriate field in the quiz. **The Canvas quiz named Project 1 Quiz is the only deliverable for this project.**

You only get one attempt to submit the quiz but you can save your draft while you work on them. PA1 is worth 20 points(amounting to 10% of your overall grade). Each of the problems is worth 5 points.

## Cheatsheet

We offer some useful hints here.

- Bash commands

  1. Secure copy:
     (a) Source file to destination:
     `$ scp source destination` where the remote destination will be of the format:
     `ubuntu@hostname:/home/ubuntu`
     For department Linux machines, the format might be similar to
     `psuid@e5-cse-135-xx.cse.psu.edu:/home/ugrads/psuid/path`
     (b) Source folder to destination:
     `$ scp -r source destination`
  2. Unzip file:
     `$ unzip filename.zip`
  3. Create directory:
     `$ mkdir foo`
  4. Create file:
     `$ touch foo.txt`
  5. Give execute permission to everyone:
     `$ chmod +x foo.sh`
  6. Compile C program using `gcc` compiler, creating an executable named `a.out`:
     `$ gcc foo.c`

7. Compile C program using `gcc` compiler, creating an executable named `foo`:

   `$ gcc foo.c -o foo`

   *Note that in most (if not all) of the course projects, we provide `Makefile(s)` – these contain the compilation command. With the `Makefile`, you can just type `make` to compile the code into an executable.*

8. Run executable named `foo`:

   `$ ./foo`

9. Redirect program output (i.e., `STDOUT`) to a file:

   `$ ./foo > output.txt`

10. Redirect file to program input (i.e., `STDIN`):

    `$ ./foo < input.txt`

11. Redirect `STDOUT` of one program to `STDIN` of another program:

    `$ ./foo1 | ./foo2`

- gdb

  1. To run a program `prog1` under `gdb`, simply execute:

     `$ gdb prog1`

  2. While running under `gdb`'s control, you can add breakpoints in the program to ease the debugging process. To add a breakpoint, type:

     `$ break <linenumber>`

  3. To run the code, type:

     `$ r`

  4. To continue running the program after a breakpoint is hit, type:

     `$ c`

  5. To inspect the stack using `gdb`, type:

     `$ backtrace` or

     `$ backtrace full` (to display contents of local variables) .

  6. To get information about individual frames, type:

     `$ info frame <frame number>`

     E.g., if you want to see information about frame 5(assuming your program has made 6 recursive function calls, since frame number starts from 0) , then the command would look like:

     `$ info frame 5`

  7. To get the size of a frame, subtract frame addresses of two consecutive frames.

- To access virtual address space related information for a process with OS-assigned identifier `PID`, follow these steps:

  1. To find `PID` of, say, prog2, type,

     `$ ps -ef | grep prog2` or,

     `$ pgrep prog2`

  2. Inspect the contents of the files `/proc/PID/maps` and `/proc/PID/smaps` for a variety of useful information. A simple web search will offer details should you find something unclear(or ask us) .

- Often a system's administrator will set an upper bound on the stack size. To find this limit:

  `$ ulimit -s`

- Alternatively, you can use the following command to get both "soft" and "hard" limits set for a process:

  `$ cat /proc/PID/limits`

- To compile the code using 32/64 bit options, add the `-m<architecture>` flag to the compile command in the Makefile. E.g., to compile with the 32 bit option:

  `$ gcc -g -m32 prog.c -o prog`

- To find the size of an executable(including its code vs. data segments) , consider using the `size` command. Look at its man pages.
- To find the size of code during run time, type the following while the code is in execution:

  ```
  $ pmap PID | grep "total"
  ```
- To see memory allocated to each section of the process, type:

  ```
  $ pmap PID
  ```
- Sample code for using `getrusage()` :

  ```
  # include <sys/time.h>
  # include <sys/resource.h>
  # include <unistd.h>
  # include <stdio.h>

  int main() {
      struct rusage usage;
      struct timeval start, end;
      int i, j, k = 0;

      getrusage(RUSAGE SELF, &usage) ;
      start = usage.rustime;
      for(i = 0; i < 100000; i++) {
          for(j = 0; j < 100000; j++) {
              k += 1;
          }
      }
      getrusage(RUSAGE SELF, &usage) ;
      end = usage.rustime;

      printf(" Started at : %ld.% lds \n", start.tvsec, start.tvusec) ;
      printf("Ended at : %ld.% lds \n", end.tvsec, end.tvusec) ;
      return 0;
  }
  ```