PA1: Address Spaces and Resource Usage Monitoring

CMPSC 473, FALL 2017

Due: September 7, 2017, 11:59:59pm

Aman Jain and Bhuvan Urgaonkar

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`), (iv) working with a code repository (`github`), (v) using Linux man pages (the `man` command), and (vi) plotting experimental data for easy visualization. All of these are good programming and experimental analysis practices that we would like you to follow throughout this course (and beyond it).

2 Getting Started

After accepting the invitational link to the Github Classroom @CMPSC473, you will be given access to your own private repository and another repository named "PA1" containing all the files needed for completing PA1. When you open this repository, you will find 5 folders, named `prog1`, ..., `prog5`. These folders contain the files described in Section 3. On the web-page of this repository, you will find a link to download it. To download copy the link and type on the command line:

```
$ git clone <link_of_the_repository>
```

You will find additional information on using GitHub and other important documents uploaded in a separate document on canvas.

As mentioned in class, you may do the bulk of your work on any Linux (virtual) machine of your choosing. However, the results in your 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 `cse-p204instxx.cse.psu.edu` (where `xx` is a machine number). 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.

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:

   (a) Which addresses are for the local variables and which ones are for the dynamically allocated variables? How were you able to deduce this? What are the directions in which the stack and the heap grow on your system?

   (b) What is the size of the process stack when it is waiting for user input? (Hint: 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.

   (c) What is the size of the process heap when it is waiting for user input?

   (d) What are the address limits of the stack and the heap. (Hint: 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.) Confirm the variables being allocated lie within these limits.

   (e) Use the `strace` command to record the system calls invoked while `prog1` executes. For this, simply run `strace prog1` on the command line. Look at the man page of `strace` to learn more about it. Similarly, use man pages to learn basic information about each of these system calls. For each unique system call, write *in your own words* (just one sentence should do) what purpose this system call serves for this program.

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 32 bit and 64 bit options and carry out the following tasks separately for each:

   (a) Observe and report the differences in the following for the 32 bit and 64 bit executables: (i) size of compiled code, (ii) size of code during run time, (iii) size of linked libraries.

   (b) Use `gdb` to find the program statement that caused the error. See some tips on `gdb` in the Appendix if needed.

(c) Explain the cause of this error. Support your claim with address limits found from `/proc`.

(d) Using `gdb` back trace the stack. Examine individual frames in the stack to find each frame's size. Combine this with your knowledge (or estimate) of the sizes of other address space components to determine how many invocations of the recursive function should be possible on your system. How many invocations occur when you actually execute the program?

(e) What are the contents of a frame in general? Which of these are present in a frame corresponding to an invocation of the recursive function and what are their sizes?

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. Again, create both a 32 bit and a 64 bit executable. For each of these, execute it. Upon executing, you will see an error on the console before the program terminates. You are to carry out the following tasks:

   (a) Observe and report the differences in the following for the 32 bit and 64 bit executables: (i) size of compiled code, (ii) size of code during run time, (iii) size of linked libraries.

   (b) Use `valgrind` to find the cause of the error including the program statement causing it. For this, simply run `valgrind prog3` on the command line. Validate this alleged cause with address space related information gleaned from `/proc`.

   (c) How is this error different than the one for `prog2`?

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

   (a) Describe the cause and nature of these errors. How would you fix them?

   (b) Modify the program to use `getrusage` for measuring the following: (i) user CPU time used, (ii) system CPU time used - what is the difference between (i) and (ii)?, (iii) maximum resident set size - what is this?, (iii) signals received - who may have sent these?, (iv) voluntary context switches, (v) involuntary context switches - what is the difference between (iv) and (v)? Look at the sample code in the Appendix for an example on how to use `getrusage()`.

5. **Tracking resource usage: instrumenting the program vs. using external observations:** You are given executables for two programs (named `prog51` and `prog52`) that follow different regimes of dynamic memory allocations. Figures 1(a) and (b) depict fine-grained heap size evolution for these two programs, respectively. These were generated using a tool called Massif visualizer which instruments a program

to record resource allocation information at run-time. You are to carry out the following tasks:

(a) Using appropriate system calls and scripting come up with your own solution for recording the evolution of the size of virtual memory allocated to a specified process. Your solution should record this for the process of interest once every second and output this into a text file. Here is an example of what such a file might look like:
7092
8134
12234
54874
345355
4347712
...
Use the plotting tool supplied by us (called `plot_script.py`) to create your own graphs of virtual memory allocation evolution for `prog51` and `prog52`. To run the plotting script:
`$ python plot_script.py <path_to_your_file>`

(b) Give a brief description of the difference between your graph and corresponding (included) graph. Why are you seeing this difference?

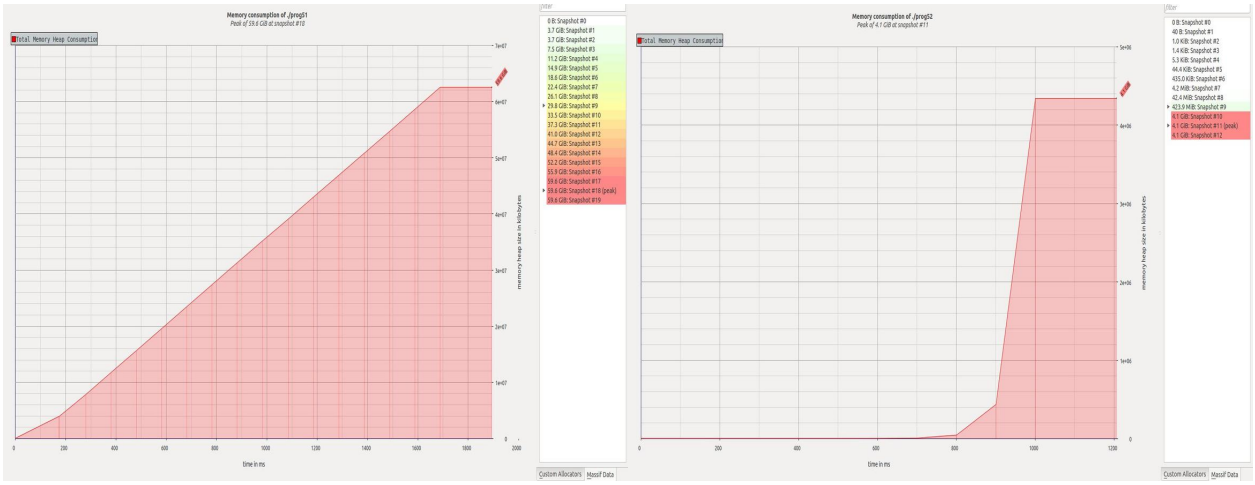

Figure 1: Heap allocations for `prog51` (left) and `prog52`.

4 Submission and Grading

You will submit all your source files, Makefiles, and READMEs (the last to convey something non-trivial we may need to know to use your code). You are to submit a report

answering all the questions posed above into your github repo. PA1 is worth 10 points (amounting to 10% of your overall grade). Each of the 5 tasks outlined above is worth 2 points. The TAs will evaluate your submission based on (i) the quality and correctness of your report and code, and (ii) examining, compiling, and running code modified by you.

Appendix

We offer some useful hints here.

- **Quick notes on** `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 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.ru_stime;
  for (i = 0; i < 100000; i++) {
    for (j = 0; j < 100000; j++) {
      k += 1;
    }
  }
  getrusage(RUSAGE_SELF, &usage);
  end = usage.ru_stime;

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