Programming Assignment 1

Yuchen Duan

Sep 7

473

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 dynami- cally 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:

Memory Growth :

Address 1 = 0x7ffed0abc73c Address 2 = 0xf90010

Address 1 = 0x7ffed0abc6fc Address 2 = 0xf90030

Address 1 = 0x7ffed0abc6bc Address 2 = 0xf90050

Address 1 = 0x7ffed0abc67c Address 2 = 0xf90070

Address 1 = 0x7ffed0abc63c Address 2 = 0xf90090

Address 1 = 0x7ffed0abc5fc Address 2 = 0xf900b0

Address 1 = 0x7ffed0abc5bc Address 2 = 0xf900d0

Address 1 = 0x7ffed0abc57c Address 2 = 0xf900f0

Address 1 = 0x7ffed0abc53c Address 2 = 0xf90110

Address 1 = 0x7ffed0abc4fc Address 2 = 0xf90130

Address 1 = 0x7ffed0abc4bc Address 2 = 0xf90150

Press Enter to continue

(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?

Address 2 is dynamically allocated variable, and address 1 is local variable. Stack is growing from top to down and heap is growing from bottom to up. The address 1 is a large number decreasing so it is in stack and local variable always save in stack. The address 2 is a smaller number increasing so it is in heap and dynamically allocated variable always save in heap.

(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 pro- gram 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.

[stack]

Size: 84kB

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

[heap]

Size: 132 kB

(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 start- ing and ending addresses assigned to each segment of virtual memory of a process.) Confirm the variables being allocated lie within these limits.

Stack  
7ffed0aa9000-7ffed0abe000

7ffed0abe000 >7ffed0abc73c > 7ffed0aa9000

7ffed0abe000 >0x7ffed0abc4bc > 7ffed0aa9000

heap

00f90000-00fb1000

00fb1000>00f90010>00f90000

00fb1000>0xf90150>00f90000

(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.

execve

This system cell pause the current running process and start run the process that pass in with the input pass in.

brk

Set the ending of the data structure. Define the size of the allocation.

mmap

Create a virtual space for mapping, PROT\_READ it can read, PROT\_WRITE it can write, MAP\_ANONYMOUS the mapping can't be separate or break by other files

access

To check if the file is accessable under the mode, R\_OK readable, W\_OK writeable, X\_OK executeable.

open

Open the file with the define mode (read write or exec...)

fstat

Retrieved information from the file pointed at the giving name and specified by st\_mode

close

Close the file that file handler is define by open.

read

Read file by file handler to a buffer and count the size (832), return the size read

mprotect

Change the access mode of the memory (address to the address +size)

arch\_prctl

Set architecture specific, select sub-function with argument in the addr

munmap

Remove mapping of the addr space

write

Write number of bit from buffer to the file current point.

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.

I)64bit 1651B, 32bit 1419B

II)64bit 15872KB, 32bit 13816KB

III)64bit 3784KB, 32bit 1748KB

(b) Use gdb to find the program statement that caused the error. See some tips on

gdb in the Appendix if needed.

64bit: Program received signal SIGSEGV, Segmentation fault.

0x00000000004005af in allocate (count=Cannot access memory at address 0x7fffff366f2c

) at prog2.c:5

32bit: Program received signal SIGSEGV, Segmentation fault.

allocate (count=0) at prog2.c:11

11 c = malloc (30000);

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

32bit: address 0x0804a000-0x08096000=0d311296

The loop is if count>=0(…. Count-1);) so the loop go throw 11 times

The heap address space only can take 10 malloc(30000)

64bit

Can't access the address 0x7fffff366f2c, it is not in the stack. The loop 11 address is out of range

(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?

32bit: the total space of (stack) / each time need for stack

(0xffffe000-0xff48a000)/(1200048)=10.0074930336

64bit:

(0x7ffffffff000-0x7fffff48b000)/(1200048)=10.0074930336

It can only run 10 possible invocation but actually only 11 time before the bug

(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?

32 bit:

Local address: x 4B\*30000+ 4B

Parameter: count 4B

Return address: 4B

Frame Pointer: 4B

CPU saved Reg.

Back trace info frame

64bit:

Local address: x 4B\*30000+ 8B

Parameter: count 4B

Return address: 8B

Frame Pointer: 8B

CPU saved Reg.

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 exe- cutable. 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.

I)64bit 2000B, 32bit 1700B

II)64bit 43474916KB, 32bit 437184KB

III)64bit 6360KB, 32bit 1916KB

(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. Val- idate this alleged cause with address space related information gleaned from

/proc.

64bit:

==5694== Warning: set address range perms: large range [0x7a7f040, 0x1f7f7440) (undefined)

Stack Address = 0x7feffecf0 Heap Address = 0x7a7f040

==5694== Warning: set address range perms: large range [0x3aeef040, 0x1295a1840) (undefined)

Stack Address = 0x7feffeb10 Heap Address = 0x3aeef040

==5694== Invalid write of size 1

==5694== at 0x4A09AD8: memset (mc\_replace\_strmem.c:1011)

The memset is trying to set an Invalid address ch1 which= &b[I], which malloc outside of the heap in the first place. Out of rang for both stack and heap

32bit:

==5747== Warning: set address range perms: large range [0x884a028, 0x205c2428) (undefined)

Stack Address = 0xfeb95f64 Heap Address = 0x884a028

==5747== Warning: silly arg (-294967296) to malloc()

==5747== Invalid write of size 1

==5747== at 0x400A35B: memset (mc\_replace\_strmem.c:1011)

The memset is trying to set an Invalid address ch1 which= &b[I], which malloc outside of the heap in the first place. Out of rang for heap

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

This time didn't stop at malloc. It stops at memset. This time it was trying to malloc a huge humber which return address not in heap so b's address can't be memset

4. **And some more:** The program prog4.c may seem to be error-free. But when exe- cuting 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?

==6347== definitely lost: 1,842,976,000 bytes in 921,488 blocks

There is a memory leak caused by didn't free pointers. "if(func(i)) { free (p);}"

To fix this we have to free the pointer by free every time

(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().

Program execution successfull

system Started at: 0.0s

I)system Ended at: 0.388940s

user Started at: 0.0s

II)user Ended at: 0.889864s

Different between I and II is 0.500924. ii is user usage for the pros. And kernel use

III)The maximum resident set size used are: 1814680, maximum resident set size is the peak RAM use for this process

III)Number of signals received : 0, OS mey sent this

Iv)voluntary context switch : 1, one thread explicitly yields the CPU to another

V)involuntary context switch : 164, different are 163, scheduler suspends an thread and switches to another thread.

5. **Tracking resource usage: instrumenting the program vs. using external observa- tions:** 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 fol- lowing 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

…

For sleep 1s:

4956

24687517596

85375048668

137433734608

137433734608

137433734608

137433734608

/home/ugrads/yjd5008/pa1-rainduan/prog5/memory\_utilization.png

7532

43475948

43475948

43475948

43475948

/home/ugrads/yjd5008/pa1-rainduan/prog5/prog52per1s.png

For more information sleep 0.5s:

4956

4956

29742207684

60886754880

92031302076

123617255748

137433734608

137433734608

137433734608

137433734608

137433734608

137433734608

137433734608

137433734608

7532

43475948

43475948

43475948

43475948

43475948

43475948

43475948

43475948

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 correspond- ing (included) graph. Why are you seeing this difference?

My slop is not as detail, the graph that was included is reach to top from buttom in 2 sec but to Compared two of my graph the proportionally the prog51 grow slower and prog52 grow faster to the limit.

I think the different is cause by time frames the graph is getting closer when sleep time getting smaller

And I use total size (see piazza), the graph given use heap.