Author: Yuxiang Luo email:yjl5310@psu.edu

I have no teammate, this project finished by myself

Question Answers of PA1

1. Stack, heap and system calls：
2. 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?

Answer: Address 1 is for local variable and address 2 sis for dynamically allocated variable. The reason is local variable is allocated on stack and dynamically allocated variable is allocated on heap. Moreover, address 1 grows downward, and address 2 grows upward. In x86 Linux system, stack grows downward and heap grows upward.

1. What is the size of the process stack when it is waiting for user input?

Answer: The stack size is 2140KB.

1. What is the size of the process heap when it is waiting for user input?

Answer: The heap size is 132KB.

1. What are the address limits of the stack and the heap.

Answer: stack limitation 7fff32ca4000-7fff32ebb000, heap limitation 5606d76b7000-5606d76d8000. All addresses are within the limitation.

1. 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.

Answer: Following system calls are called

execve: execute an executable file

brk: this is to allocate heap memory

openat: open file

fstat: get the file information

mprotect: set the memory previlidge

open: open file

close: close file

mmap: map the file into memory

read: read file

write: write file

pread: read the value of pointer

1. Debugging refresher
2. 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.

Answer:

|  |  |  |  |
| --- | --- | --- | --- |
| Program | Compiled code size | Runtime code size | Linked library size |
| prog2\_32 | 2363 | 9504KB | 2312KB |
| prog2\_64 | 2318 | 9552KB | 2160KB |

1. Use gdb to find the program statement that caused the error.

Answer:According to gdb, code breaks at function call allocate

when count=4

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

Answer: Program breaks because of stack overflow.

The space used and limitation of the program when breaks are

stack used: about 7200kb when break

heap used: about 276kb when break

stack limitation: 8192kb

heap limitation: unlimited

1. 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?

Answer: Each recursive call stack size is about 1170kb. My system stack limitation is 8192kb, so it would support around

6 recursive calls. (main function will also grab some stack space),

the program also breaks after 6 recursive calls..

1. 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?

Answer:Each stack frame consists:

Original values of changed registers (rip, rbp in this case, 16 bytes).

Local variables. (x in this case, 1200000bytes)

Parameters that passed by stack. (count in this case, 4bytes)

1. More Debugging:
2. 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.

Answer:

|  |  |  |  |
| --- | --- | --- | --- |
| Program | Compiled Code Size | Runtime Code Size | Linked Library Size |
| prog3\_32 | 2613 | 7792KB | 3172KB |
| prog3\_64 | 2722 | 8144KB | 3500KB |

1. 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.

Answer: Valgrind shows that program breaks because it memset a NULL pointer bat the 10th recursion call of allocate. The reason should be malloc failed, because it need to allocate 4\*(10^10) bytes (around 40 gigabytes) memory once.

According to the memory map of my system, the memory space of 32bit is 4 giga bytes,the memory space of 64bits is unlimited.

For 32 bits, it is obviously that size has been over range.

For 64bits version, although the memory space is large enough, however, the highest address of heapstill can not overlap other part of memory space.

According to the memory map, my heap start from 583d6000, my linked library starts from f7c38000,the difference is around 20giga bytes, 40 gigabytes is still too large.

1. How is this error different than the one for prog2?

Answer: The prog2 failed because of stack over flow, prog3 failed because of memory allocation failed.

1. And Some More
2. Describe the cause and nature of these errors. How would you fix them?

Answer: By running valgrind, it shows that some memory allocated but not freed, it is because that func(i) in line 15 will return false from time to time,then free(p) is not called, but malloc is always called.

The fix is simple, just move malloc statement into the if block, so when func(i)return false, then malloc is not called.

1. 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)?

Answer:I wrote a prog4\_rusage.c and run it, it gives the result as follows:

i) The user time is 19.86ms.

ii) The system time is 0.00000ms. The difference is that user time is time spend in user mode,but system time is time that spend in kernel mode.

iii) The max resident set size is 26128KB, it means the largest amount of memory size this process ever used.

iv) Voluntary context switches: 6

v) Involuntary context switches: 0.

Voluntary context switch means this process gives up running by itself(for example, by calling yield()); involuntary context switch means that this process is preempted by other higher priority processes.

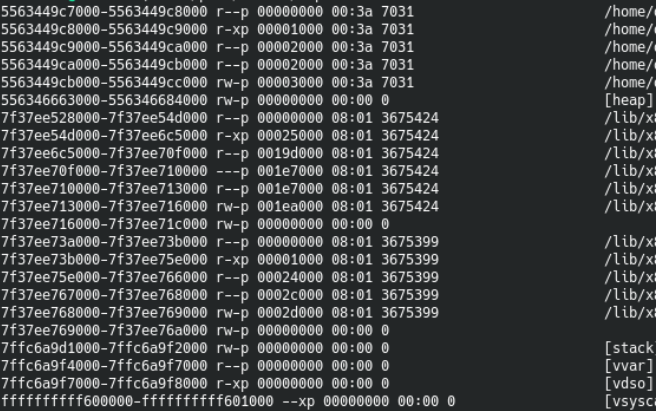
1. Multi-process Program
2. Compare the following for the parent and child processes (i) PID (ii) address of dynamic variables.

Answer: in parent the PID is 160581, the address of pointer is 0x55824d30d6b0; in child, the PID is 160583, the address of pointer is 0x55824d30d6b0. PIDs are different, but address for same variable are the same

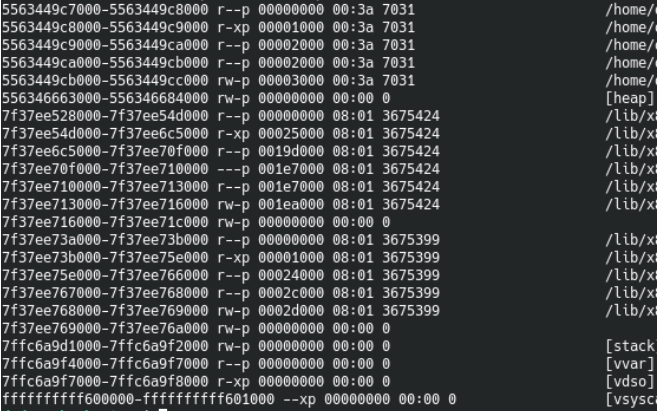
1. Compare the address space of the parent and child before and after the ”exec” command.

Answer:

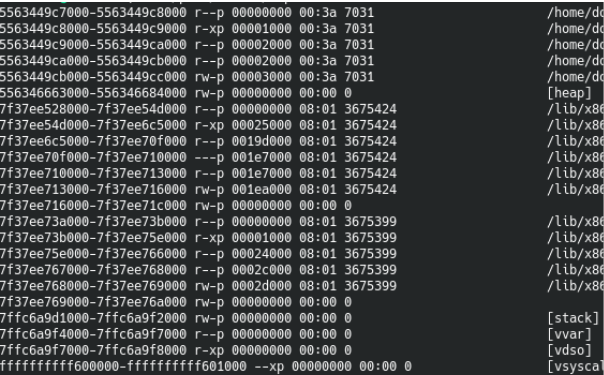
Space of parent before exec:



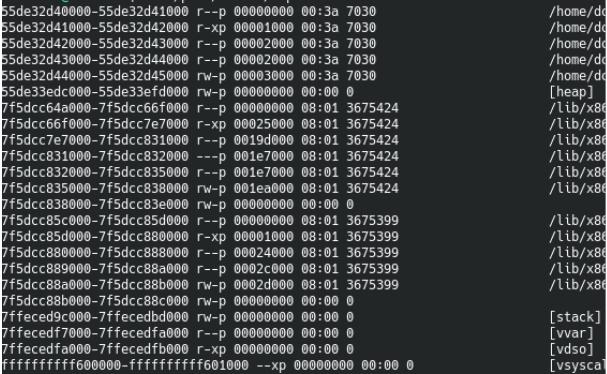
Space of child before exec:



Space of parent after exec:



Space of childafter exec:

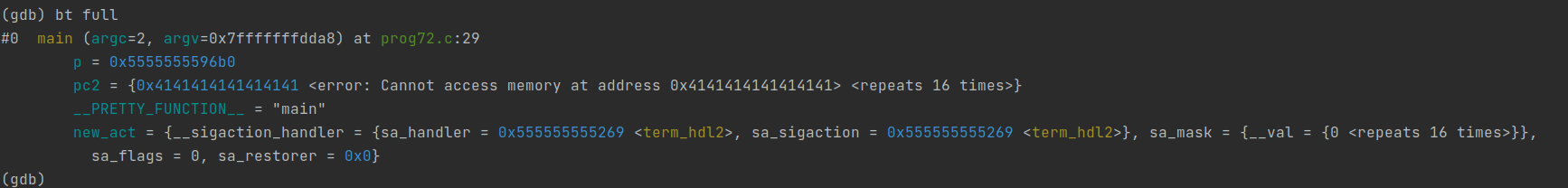


It can be seen that before exec, the memory space of parent and child are exactly the same, after exec, the parent space does not changed while the child space has changed.

1. Compare the stack of prog72 before and after signal handling.

Answer:

The stack of prog72 before signal handling showed in GDB:



The stack of prog72 after signal handling showed in GDB:



It seems that signal handler does not have its own stack, but uses the process’s stack.