**PA1: Address Spaces and Resource Usage Monitoring CMPSC 473, Spring 2021**

**Due: Feb 16th, 2021 By: Joseph Brauckmann (jdb6064) and Taylan Unal (tuu2)**

**Server Address:**

cse-p204instXX.cse.psu.edu (I use 21 instead of XX, but doesnt matter)

[Penn State Engineering: CSE Student Lab Access During COVID-19 Response Period (psu.edu)](https://www.eecs.psu.edu/cse-student-lab-access-covid-19/index.aspx)

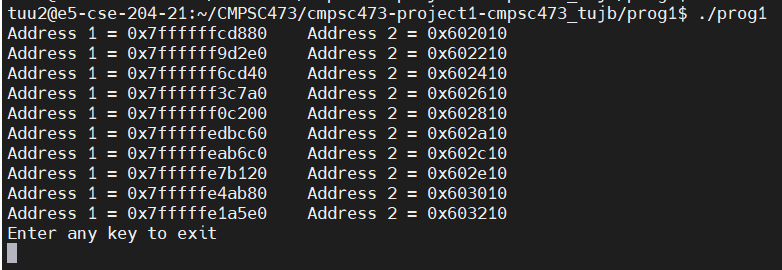
**ssh <YourPSUUserid>@cse-p204inst??.cse.psu.edu** where <YourPSUUserid> is your PSU UserName and ?? is the machine numbers (01-38) for the Linux computers.

* Example: ssh abc123@e5-cse-204-10.cse.psu.edu which would connect user abc123 to the computer with the Host Name of e5-cse-204-10.cse.psu.edu.

**Disable Address Randomization:**

* setarch `uname -m` -R /bin/bash

**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:**

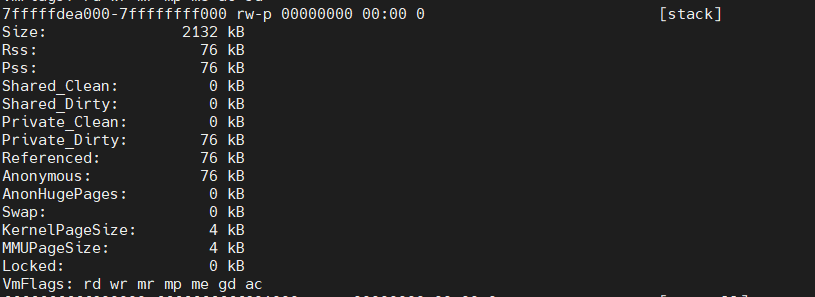
**(a) Which addresses are for the local variables and which ones are for the dynamically allocated variables? How were you able to deduce this? What are the directions in which the stack and the heap grow on your system?** 

For prog1, Address 1 represents local variables, and Address 2 values are for dynamic variables. We were able to deduce this by looking at the program, and saw that variable int ‘a’, or Address 1, was being statically allocated within the program, and variable char \*b is a pointer to a memory location that is being dynamically allocated.

**(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.**

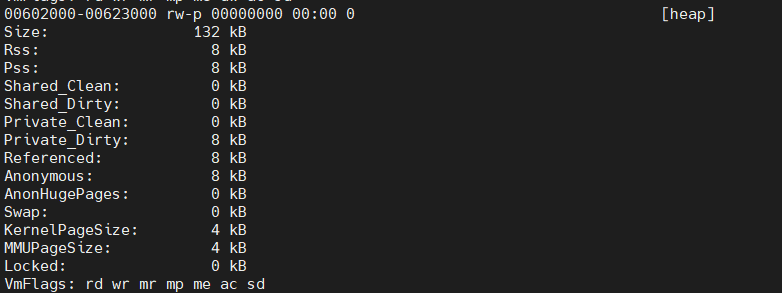


Run prog1 in a separate tab, then open proc/PID/smaps, look for [stack] to see data.



**Size of Process Stack:** 2132 KB

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



**Size of Process Heap:** 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 starting and ending addresses assigned to each segment of virtual memory of a process.) Confirm the variables being allocated lie within these limits.**

**Stack:** 7fffffdea000-7ffffffff000 rw-p 00000000 00:00 0 [stack]

**Heap:** 00602000-00623000 rw-p 00000000 00:00 0 [heap]

All variables being allocated in the stack (Address 1) are within the stack address range, and all variables being allocated in the heap (Address 2) are within the heap address range.

**(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.**

tuu2@e5-cse-204-21:~/CMPSC473/cmpsc473-project1-cmpsc473\_tujb/prog1$ strace ./prog1

execve("./prog1", ["./prog1"], 0x7fffffffdf10 /\* 53 vars \*/) = 0

brk(NULL) = 0x602000

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7ffff7ff9000

access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

open("/usr/local/cuda/lib64/tls/x86\_64/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

stat("/usr/local/cuda/lib64/tls/x86\_64", 0x7fffffffc5c0) = -1 ENOENT (No such file or directory)

open("/usr/local/cuda/lib64/tls/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

stat("/usr/local/cuda/lib64/tls", 0x7fffffffc5c0) = -1 ENOENT (No such file or directory)

open("/usr/local/cuda/lib64/x86\_64/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

stat("/usr/local/cuda/lib64/x86\_64", 0x7fffffffc5c0) = -1 ENOENT (No such file or directory)

open("/usr/local/cuda/lib64/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

stat("/usr/local/cuda/lib64", {st\_mode=S\_IFDIR|0755, st\_size=8192, ...}) = 0

open("tls/x86\_64/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

open("tls/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

open("x86\_64/libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

open("libc.so.6", O\_RDONLY|O\_CLOEXEC) = -1 ENOENT (No such file or directory)

open("/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

fstat(3, {st\_mode=S\_IFREG|0644, st\_size=204488, ...}) = 0

mmap(NULL, 204488, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7ffff7fc7000

close(3) = 0

open("/lib64/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0`&\2\0\0\0\0\0"..., 832) = 832

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=2156272, ...}) = 0

mmap(NULL, 3985920, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7ffff7a0d000

mprotect(0x7ffff7bd1000, 2093056, PROT\_NONE) = 0

mmap(0x7ffff7dd0000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1c3000) = 0x7ffff7dd0000

mmap(0x7ffff7dd6000, 16896, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7ffff7dd6000

close(3) = 0

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7ffff7fc6000

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7ffff7fc4000

arch\_prctl(ARCH\_SET\_FS, 0x7ffff7fc4740) = 0

mprotect(0x7ffff7dd0000, 16384, PROT\_READ) = 0

mprotect(0x600000, 4096, PROT\_READ) = 0

mprotect(0x7ffff7ffc000, 4096, PROT\_READ) = 0

munmap(0x7ffff7fc7000, 204488) = 0

brk(NULL) = 0x602000

brk(0x623000) = 0x623000

brk(NULL) = 0x623000

fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 2), ...}) = 0

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7ffff7ff8000

write(1, "Address 1 = 0x7ffffffcd880 Ad"..., 51Address 1 = 0x7ffffffcd880 Address 2 = 0x602010

) = 51

write(1, "Address 1 = 0x7ffffff9d2e0 Ad"..., 51Address 1 = 0x7ffffff9d2e0 Address 2 = 0x602210

) = 51

write(1, "Address 1 = 0x7ffffff6cd40 Ad"..., 51Address 1 = 0x7ffffff6cd40 Address 2 = 0x602410

) = 51

write(1, "Address 1 = 0x7ffffff3c7a0 Ad"..., 51Address 1 = 0x7ffffff3c7a0 Address 2 = 0x602610

) = 51

write(1, "Address 1 = 0x7ffffff0c200 Ad"..., 51Address 1 = 0x7ffffff0c200 Address 2 = 0x602810

) = 51

write(1, "Address 1 = 0x7fffffedbc60 Ad"..., 51Address 1 = 0x7fffffedbc60 Address 2 = 0x602a10

) = 51

write(1, "Address 1 = 0x7fffffeab6c0 Ad"..., 51Address 1 = 0x7fffffeab6c0 Address 2 = 0x602c10

) = 51

write(1, "Address 1 = 0x7fffffe7b120 Ad"..., 51Address 1 = 0x7fffffe7b120 Address 2 = 0x602e10

) = 51

write(1, "Address 1 = 0x7fffffe4ab80 Ad"..., 51Address 1 = 0x7fffffe4ab80 Address 2 = 0x603010

) = 51

write(1, "Address 1 = 0x7fffffe1a5e0 Ad"..., 51Address 1 = 0x7fffffe1a5e0 Address 2 = 0x603210

) = 51

write(1, "Enter any key to exit\n", 22Enter any key to exit

) = 22

fstat(0, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(136, 2), ...}) = 0

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7ffff7ff7000

read(0,

***System call definitions:***

**execve -** executes program

**mmap-** map files into memory

**brk-** modify data segment size

**access-** check if user has proper permissions

**open-** opens a file or even creates a file

**stat-** obtain information about the file

**fstat:** similar to stat except the info received is specified by the file descriptor fd

**close:** close a file

**arch\_prctl:** chose which architecture protocol you will use

**mprotect:** protect a section of memory

**munmap:** unmap files from memory

**write:** write to a file (modification)

**read:** read from a file

**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) Size of compiled code:

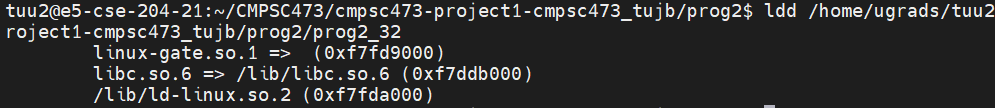
* prog2\_32 is 8536 bytes
* prog2\_64 is 9928 bytes

ii) Size of code during runtime: (use pmap PID | grep “total”)

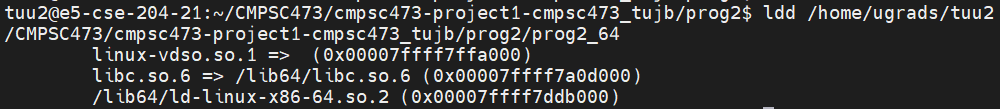
* prog2\_32 is 220928k,
* prog2\_64 is 220668K

iii) Size of linked libraries: (use ldd, full path to program)

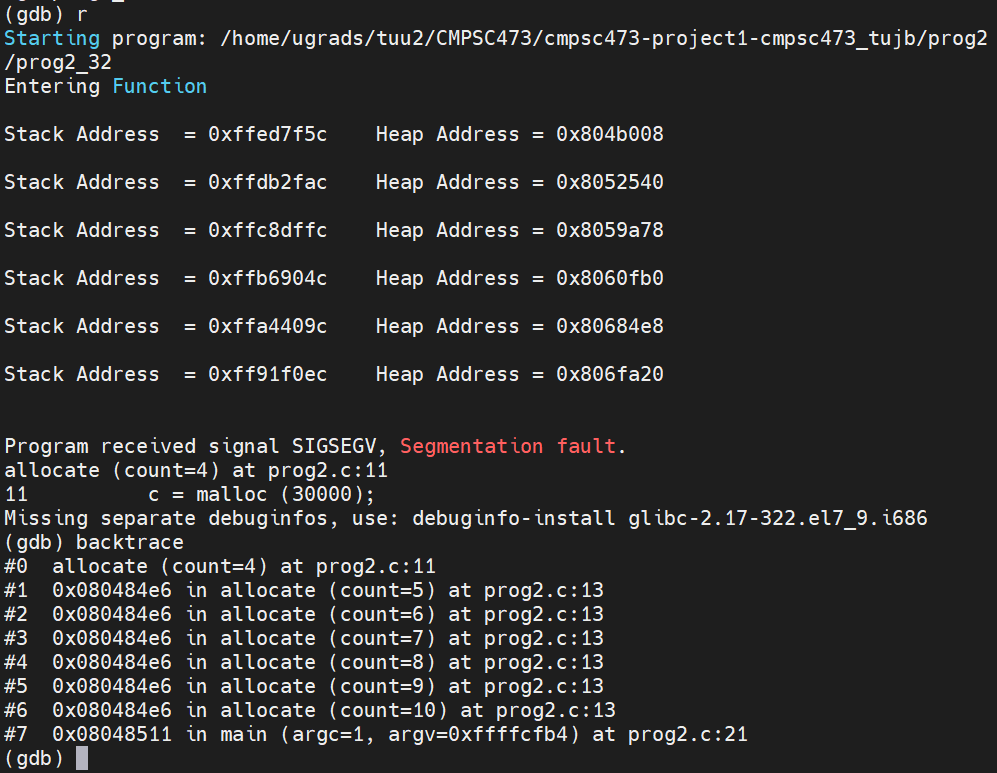
For 32Bit:



For 64Bit:



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

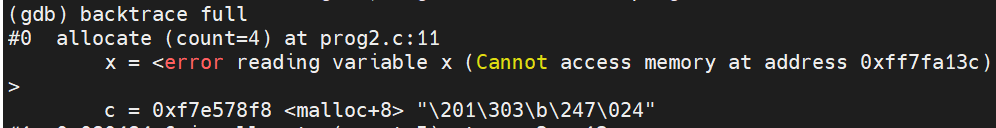


Using GDB, the program statement that caused the error was c = malloc (30000) at line 11 of the program.

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

**32bit:** PID: 917

GDB Debugging:



**Stack Address Range from /proc/917/maps:**

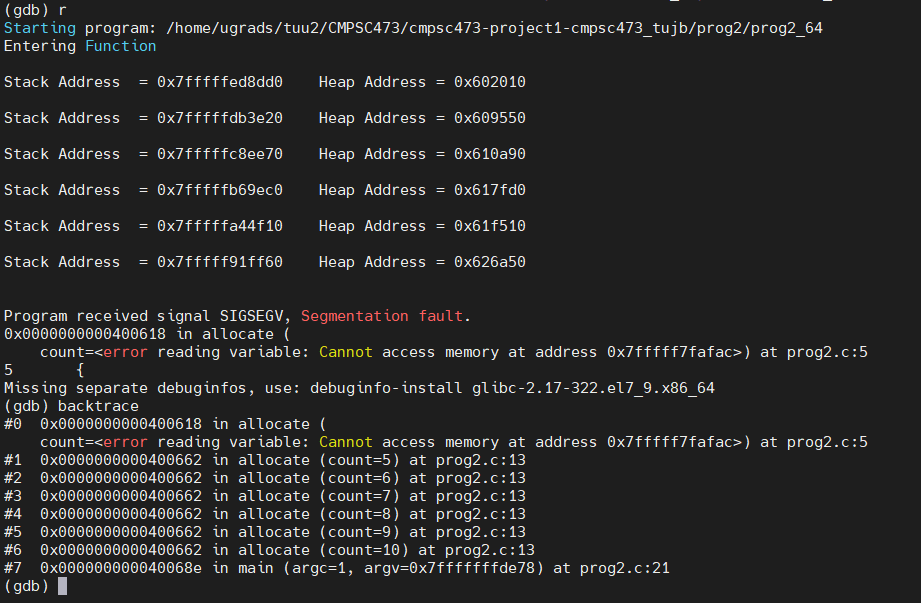
ff91e000-ffffe000 rw-p 00000000 00:00 0 [stack]

**Stack Size:** 6e0000 or 7,208,960 in decimal

The error is being thrown because the program is trying to allocate to a memory address that is not within the address range of the stack. Since 0xff7fa13c is in a region before the start address of 7ffffffde000, the program fails to allocate the memory block.

**64bit:**

PID: 2170



**Stack Address Range from /proc/2170/maps:**

7fffff91f000-7ffffffff000 rw-p 00000000 00:00 0 [stack]

**Stack Size:** 6e0000 or 7,208,960 in decimal

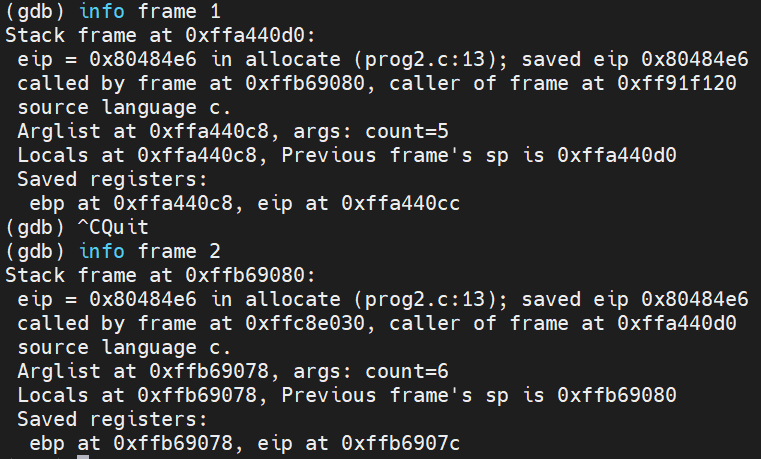
The error is being thrown because the program is trying to allocate to a memory address that is not within the range of the stack. Since 0xff7fa17c is in a region before the start address of 7ffffffde000, the program fails to allocate the memory block.

**(d) Using gdb backtrace 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?**

**Frame Size:** (next frame address) - (current frame address)

**Max. Theoretical Invocations:** (stack size) / (frame size)

**32Bit:**

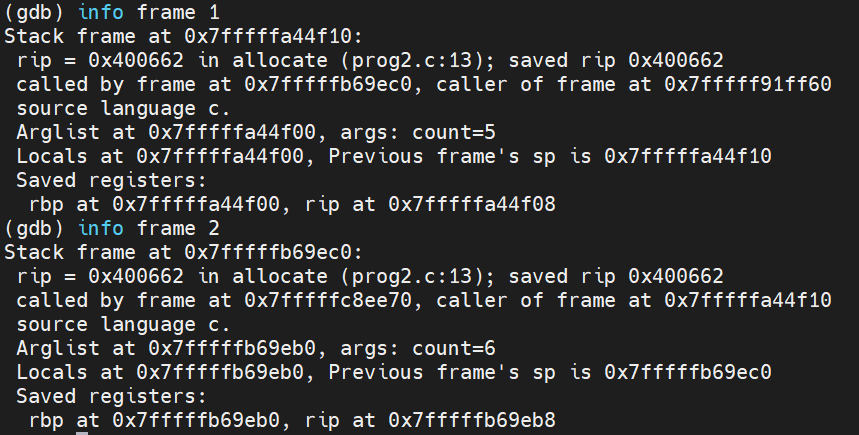
****

32Bit Frame Size: 0xffb69080 - 0xffa440d0 = 0x124fb0 or 1,200,048 in decimal

Max. Theoretical Invocations: 7,208,960 / 1,200,048 = 6.00722

Actual Execution Count: 6 invocations before segfault

**64Bit:**

****

**64Bit Frame Size:** 0x7fffffb69ec0 - 0x7fffffa44f10 = 0x124fb0 or 1,200,048 in decimal

**Max. Theoretical Invocations:** 7,208,960 / 1,200,048 = 6.00722

**Actual Execution Count:** 6 invocations before segfault

**(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?**

Generally, the contents of a stack frame are the data associated with that particular call to a function. Frame contents include the function arguments, local variables, and the address at which the function executes. The variable present for the invocation of the recursive function is “count” and its size is 524KB.

**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:**

**(Change prog3 makefile to compile for both 32, 64. Using prog2 makefile as reference)**

**(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.**

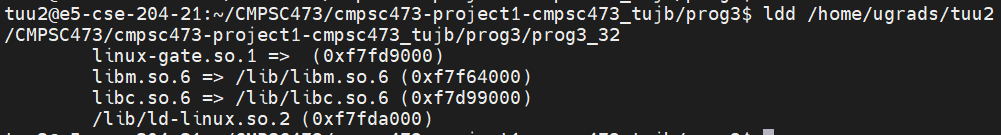
i) Size of compiled code: (use ls -l to see file sizes)

* prog3\_32 is 8724 bytes
* prog3\_64 is 10152 bytes

ii) Size of code during runtime: (use pmap PID | grep “total”)

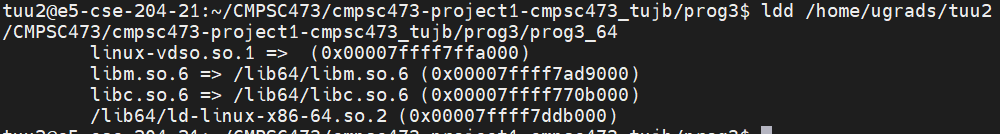
* prog3\_32 is 222076K,
* prog3\_64 is 221552K

iii) Size of linked libraries: (use ldd /full/path/to/program)

**For 32Bit:**

Total Size: 0xf7fd9000 + 0xf7f64000 + 0xf7d99000 + 0xf7fda000 = 0x3dfcb0000

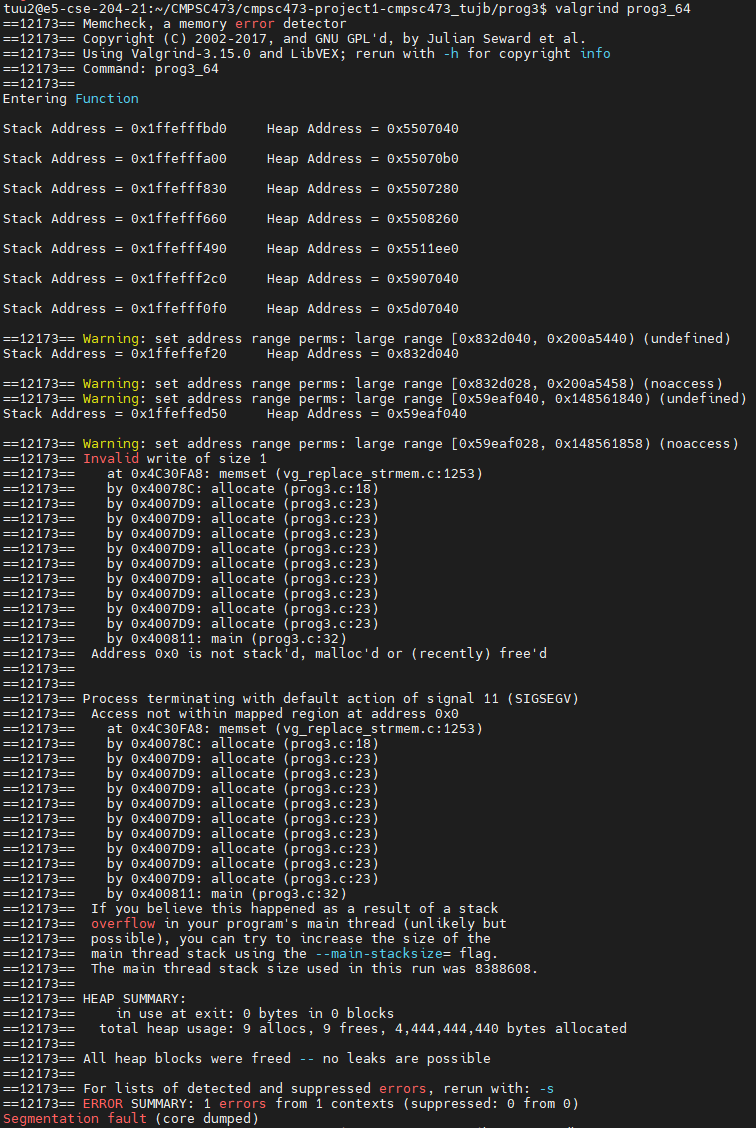
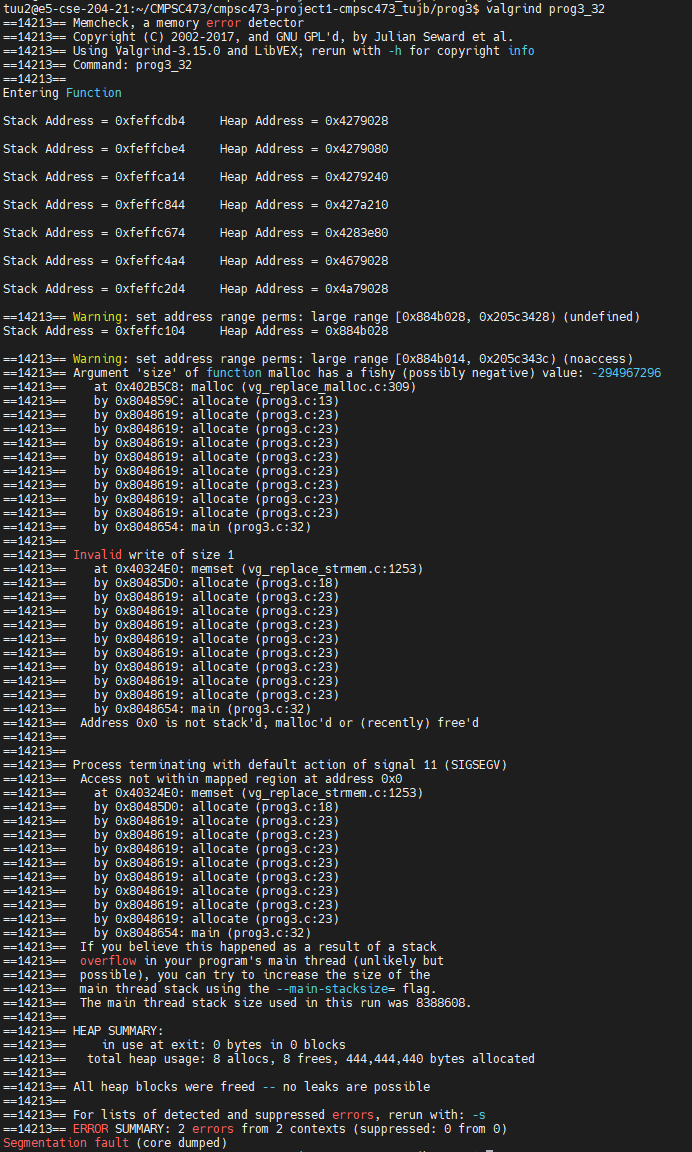
**For 64Bit:**



Total Size: 0x00007ffff7ffa000 + 0x00007ffff7ad9000 + 0x00007ffff770b000 + 0x00007ffff7ddb000 = 0x1ffffdefb9000

**(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.**

**32Bit: 64Bit:**



Using valgrind, found that while running the program, after 7 iterations, and 7 malloc operations, that there was an invalid write of size 1. This is because the function recursively called the malloc operation to allocate address sizes that grew by 10^r, where r++ each iteration. After 7 iterations, the space that was trying to be addressed simply didn't exist, and threw a stack overflow error.

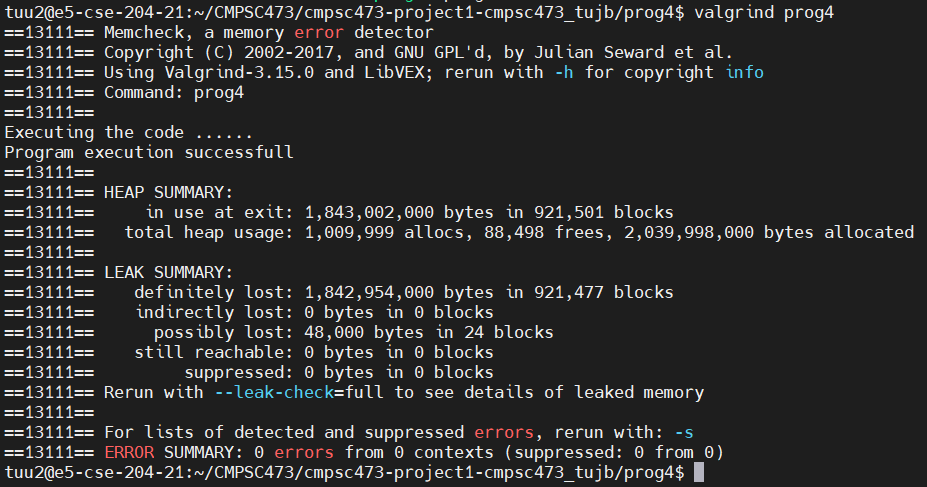
**(c) How is this error different from the one for prog2?**

Asked Avik: Don’t know how different from program2.

Avik says that they are very similar. Both are address space related. In the right direction, just think a bit more. Think about who’s address space is being violated.

The error found in prog3 is quite similar to prog2 as it again deals with address space allocation errors due to addresses being allocated out of range. The address space allocation error in prog2 dealt with memory addresses that were just out of range of the space provided by the stack, causing a segmentation fault. Similarly, prog3 also causes a segmentation fault, but in this case, it’s because the address space that is being requested to be malloc’d is simply too large, causing 32bit overflows in the case of prog3\_32, and the address spaces that are being requested are simply undefined in the space of prog3\_32 and prog3\_64.

**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?**Although the program ran and completed correctly, there are clearly many issues with this program. First, we can see that on exit, the heap still contained ~1.8 billion bytes and after the program allocated 1 million times, there were only 88,598 frees. Additionally, there is a clearly massive memory leak with this program, with ~1.8billion bytes in 921,501 blocks being definitely lost.

To fix the issue of allocated memory that was never freed, the code in prog4 has to be changed to ensure that frees don’t only occur once i%j==0. Additionally, to fix the memory leak, we must ensure that there are deallocations for each allocation of memory.

**(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().**[seems easier]

**5. Multi-process program that uses fork, exec, wait, kill, exit calls: Compile the multi-process program prog71 using the makefile or using the following commands.**

**$ gcc -g prog72.c -o prog72**

**$ gcc -g prog71.c -o prog71**

**Execute prog71 as follows: $ ./prog71.**

**a) Compare the following for the parent and child processes (i) PID (ii) address of dynamic variables.**

**i) PID:**

Parent PID: 2980

Child PID: 2981

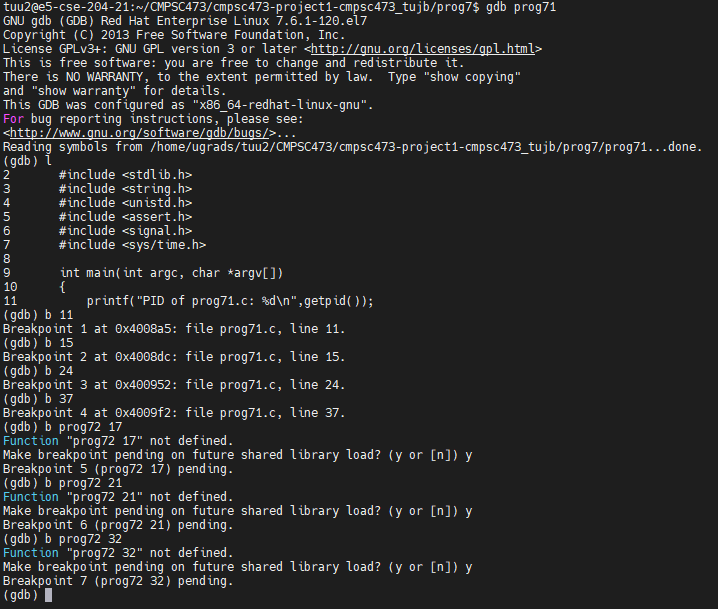
**ii) Address of the dynamic variables:**

Parent dynamic variable ‘p’ address: 0x602010

Child dynamic variable ‘p’ address: 0x602010

Both the parent and the child processes have a dynamic variable that appears to point to the same memory address. Since there was no error in runtime, it can be assumed that this is a property of parent-child processes, that dynamic variables can be shared between them.

**b) Compare the address space of the parent and child before and after the ”exec” command. (Hint: Run gdb prog71 and add breakpoints at lines 11, 15, 24, 37 in the file ”prog71” and add additional breakpoints in the file ”prog72” at line numbers 17, 21, 32.**

****

**Before ‘exec’:**

* Parent address space: 7ffffffde000-7ffffffff000 [stack]
* Child address space: 7ffffffde000-7ffffffff000 [stack]

After ‘exec’:

* Parent address space: 7ffffffde000-7ffffffff000 [stack]
* Child address space: 7ffffffde000-7ffffffff000 [stack]

**c) Compare the stack of prog72 before and after signal handling. (Hint: you may try increasing the sleep time at line 33 in prog71, if that is helpful). Attach screenshots of your findings for questions a, b, and c.**