IC221 Lab: Memory Leaks Spring AY2022

100 points total

Submission

- completed worksheet

- fixed memleak.c

- fixed memviolation.c

Testing ./test.sh

**Debugging Memory Errors with Valgrind**

In this lab, you will dynamically allocate memory in multiple contexts, while ensuring that your program does not have memory leaks or memory violations. Fortunately, there exists a wonderful debugging program which can capture and help you debug both: Valgrind.

**Memory Leaks**

A memory leak occurs when dynamically allocated memory, allocated using malloc() or calloc(), is not freed properly. It is vital that memory leaks are plugged because they can cause system wide performance issues as one program begins to hog all the memory, affecting access to the resources for other programs.

To understand a memory leak, let's look at perhaps the most offensive memory leaking program ever written:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/types.h>  int main(){  while(1){  malloc(1024); //memory leak!  sleep(0.5); //slow down the leak  }  } |

At the malloc(), new memory is allocated, but it is never assigned to a pointer. Thus, there is no way to keep track of the memory and no way to deallocate it; we have a memory leak. This program is even more terrible in that it loops forever leaking memory. If run, it will eventually slow down and cripple your computer. If you do decide to run this program, quickly kill it.

Ordinary memory leaks are less offensive. Here is a more common example:

|  |  |
| --- | --- |
| /\*memleak\_example.c\*/  #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \* argv[]){  int \* a;  a = malloc(sizeof(int));  \*a = 10;  printf("%d\n", \*a);  a = calloc(3, sizeof(int \*));  a[0] = 10;  a[1] = 20;  a[2] = 30;  printf("%d %d %d\n", a[0], a[1], a[2]);  } | This is a simple program that uses an integer pointer a in two different allocations.  First, it allocates a single integer and assigns the value 10 to the allocated memory.  Next, it uses a to reference an array of integers of length 3. It prints out the values for both cases. |

Here is some program output and compilation. (The -g is to compile with debugging information, which will become important later.)

|  |
| --- |
| $ gcc memleak\_example.c -g -o memleak\_example  $ ./memleak\_example  10  10 20 30 |

On its face, there doesn't seem to be anything wrong with this program in terms of its intended output. It compiles without errors, and it runs as intended. Yet, this program is wrong, and there is a memory leak in it.

Upon the second allocation and assignment to a, the previous allocation is not freed. The assignment of the second allocation from calloc() will *overwrite* the previous pointer value, which used to reference the initial allocation, the one by malloc(). As a result, the previous pointer value and the memory it referenced is lost and cannot be freed; a classic memory leak.

Ok, so we know what a memory leak is and how to recognize one by reading code, but that's hard. Why can't the compiler or something figure this out for us? Turns out that this is not something that a compiler can very easily check for. However, one way to determine if a program has a memory leak is to run it and see what happens.

The valgrind debugger is the tool designed for that. It will run a program, track the memory allocations, and check at the end if all the allocated memory has been freed. If not, some memory was lost, and it will generate a warning. Let's look at the valgrind output of running the above program.

|  |
| --- |
| $ valgrind ./memleak\_example  ==30134== Memcheck, a memory error detector  ==30134== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.  ==30134== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info  ==30134== Command: ./memleak\_example  ==30134==  10  10 20 30  ==30134==  ==30134== HEAP SUMMARY:  ==30134== in use at exit: 16 bytes in 2 blocks  ==30134== total heap usage: 2 allocs, 0 frees, 16 bytes allocated  ==30134==  ==30134== LEAK SUMMARY:  ==30134== definitely lost: 16 bytes in 2 blocks  ==30134== indirectly lost: 0 bytes in 0 blocks  ==30134== possibly lost: 0 bytes in 0 blocks  ==30134== still reachable: 0 bytes in 0 blocks  ==30134== suppressed: 0 bytes in 0 blocks  ==30134== Rerun with --leak-check=full to see details of leaked memory  ==30134==  ==30134== For counts of detected and suppressed errors, rerun with: -v  ==30134== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2) |

Check out the LEAK SUMMARY section, and you find that 16 bytes were "definitely" lost. Let's re-run the valgrind with the --leak-check option set to "full" to see more details, which additionally prints the HEAP SUMMARY

|  |
| --- |
| $ valgrind --leak-check=full ./memleak\_example  (...)  ==30148== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2  ==30148== at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==30148== by 0x4005F7: main (memleak\_example.c:6)  ==30148==  ==30148== 12 bytes in 1 blocks are definitely lost in loss record 2 of 2  ==30148== at 0x4C29DB4: calloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==30148== by 0x400636: main (memleak\_example.c:12) |

It lists the two allocations. The first call to malloc() allocated 4 bytes, the size of an integer. The second allocation allocated 3 integers, or 12 bytes, with calloc(). With this information, the programmer can track down the memory leak and fix it.

**Task 1 (50 points)**

- Compile and execute memleak.c. Verify the output and review the program.

- Answer the following questions in your worksheet:

Run valgrind on the memleak program. How many bytes does it say have been definitely lost?

|  |
| --- |
| (answer on worksheet) |

On what line(s) of code does valgrind indicate a memory leak has occurred? Ensure you have compiled with the -g flag for debugging.

|  |
| --- |
| (answer on worksheet) |

Identify and describe at least one memory leak in memleak.c.

|  |
| --- |
| (answer on worksheet) |

Fix the memory leak you identified and verify your fix with valgrind.

Describe how you fixed the memory leak:

|  |
| --- |
| (answer on worksheet) |

**Memory Violations**

Memory leaks are not just the only kind of memory errors that valgrind can detect. It can also detect memory violations. A memory violation is when you access memory that you shouldn't, or access memory prior to it being initialized.

Let's look at a simple example of this, printing an uninitialized value:

|  |
| --- |
| int a;  printf("%d\n", a); |

The problem with this program is clear -- we're printing out the value of a without having previously assigned to it. This error can be detected by the compiler with the -Wall option:

|  |
| --- |
| $ gcc -Wall memviolation\_simple.c  memviolation\_simple.c:7:18: warning: variable 'a' is uninitialized when used here [-Wuninitialized]  printf("%d\n", a); |

But other memory violations are harder to recognize, particularly those involving arrays. Let's look at the program below. Can you spot the error?

|  |  |
| --- | --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \* argv[]){  int i, \*a;  a = calloc(10, sizeof(int));  for(i=0;i <= 10; i++){  a[i] = i;  }  for(i=0;i <= 10; i++){  printf("%d\n", a[i]);  }  } | Compiling and running this program, it's not obvious that anything is wrong:  $ ./memviolation\_array  0  1  2  3  4  5  6  7  8  9  10 |

No errors are reported and the numbers up to 10 are printed, but we know that we are actually writing out of bounds in our array! Fortunately, valgrind can detect such errors:

|  |
| --- |
| $ valgrind ./memviolation\_array  ==30588== Memcheck, a memory error detector  ==30588== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.  ==30588== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info  ==30588== Command: ./memviolation\_array  ==30588==  ==30588== Invalid write of size 4  ==30588== at 0x4005D8: main (in /home/scs/aviv/git/ic221/current/lab/04/stu/examples/memviolation\_array)  ==30588== Address 0x51f2068 is 0 bytes after a block of size 40 alloc'd  ==30588== at 0x4C29DB4: calloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==30588== by 0x4005B4: main (in /home/scs/aviv/git/ic221/current/lab/04/stu/examples/memviolation\_array)  ==30588==  0  1  2  3  4  5  6  7  8  9  **==30588== Invalid read of size 4**  ==30588== at 0x40060F: main (in /home/scs/aviv/git/ic221/current/lab/04/stu/examples/memviolation\_array)  ==30588== Address 0x51f2068 is 0 bytes after a block of size 40 alloc'd  ==30588== at 0x4C29DB4: calloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==30588== by 0x4005B4: main (in /home/scs/aviv/git/ic221/current/lab/04/stu/examples/memviolation\_array)  ==30588==  10  ==30588==  ==30588== HEAP SUMMARY:  ==30588== in use at exit: 40 bytes in 1 blocks  ==30588== total heap usage: 1 allocs, 0 frees, 40 bytes allocated  ==30588==  ==30588== LEAK SUMMARY:  ==30588== definitely lost: 40 bytes in 1 blocks  ==30588== indirectly lost: 0 bytes in 0 blocks  ==30588== possibly lost: 0 bytes in 0 blocks  ==30588== still reachable: 0 bytes in 0 blocks  ==30588== suppressed: 0 bytes in 0 blocks  ==30588== Rerun with --leak-check=full to see details of leaked memory  ==30588==  ==30588== For counts of detected and suppressed errors, rerun with: -v  ==30588== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2) |

In the execution output, there is an "Invalid read of size 4" occurring when array[10] is indexed and printed to the screen. This is a simple example, but invalid reads and writes and other memory violations can cause all sorts of problems.

**Task 2 (50 points)**

Compile and execute the memviolation.c program. Complete the following tasks and answer the questions in your worksheet.

- Describe the output and exeuction of the program. Does it seem to be consistent?

|  |
| --- |
| (answer on worksheet) |

- Run the program under valgrind, identify the line of code that is causing the memory violation and its input.

- Describe the programming bug:

|  |
| --- |
| (answer on worksheet) |

- Fix the memory violation and verify your fix with valgrind.