Debugging memory errors with Valgrind

Σέργιος - Ανέστης Κεφαλίδης
Κωνσταντίνος Νικολέτος
Κώστας Πλας

What is Valgrind?

- The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct. The most popular of these tools is called Memcheck.

- It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour.

What to expect with Valgrind

- Valgrind informs you of various memory corruptions that occur in your program. It provides various error messages regarding invalid memory reads and writes, use of uninitialized memory spaces and where they occurred in the program.

- Valgrind also provides functionality to pinpoint memory leaks in your program and where they come from.

Installing and running Valgrind (1)

- In debian based systems to install Valgrind run the following command on terminal:

  >$ sudo apt install valgrind

- To run Valgrind to debug a your program run:

  >$ valgrind [options] myprogram [args]

  -options: valgrind options

  -args: command line arguments for your program

Installing and running Valgrind (2)

- Valgrind can provide the line of code where a memory error/leak happened. To activate that functionality compile your programs with the -g flag. (It is generally considered a good practice, when developing your programs, to include -g and -Wall flags on your Makefiles). For example:

```
>$ gcc -g myprogram.o -o myprogram
```

Interpreting Valgrind Messages (1)

Let’s assume a simple c program:

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    int array* = malloc(sizeof(int)*5);
    array[5] = 10;
    return 0;
}
```

Interpreting Valgrind Messages (2)

If we run this simple c program with valgrind we will get the following messages:

```
==141== Memcheck, a memory error detector
==141== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==141== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==141== Command: ./myprogram
==141==
==141== Invalid write of size 4
==141==    at 0x10916B: main (myprogram.c:6)
==141==  Address 0x4a4a054 is 0 bytes after a block of size 20 alloc'd
==141==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==141==    by 0x10915E: main (myprogram.c:5)
==141==
==141==
==141== HEAP SUMMARY:
==141==     in use at exit: 20 bytes in 1 blocks
==141==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==141==
==141== LEAK SUMMARY:
==141==    definitely lost: 20 bytes in 1 blocks
==141==    indirectly lost: 0 bytes in 0 blocks
==141==      possibly lost: 0 bytes in 0 blocks
==141==    still reachable: 0 bytes in 0 blocks
==141==         suppressed: 0 bytes in 0 blocks
==141== Rerun with --leak-check=full to see details of leaked memory
==141==
==141== For lists of detected and suppressed errors, rerun with: -s
==141== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Interpreting Valgrind Messages (3)

==141== : The process ID. This will very rarely concern us.

Invalid write of size 4 : The first line of every message Valgrind presents to us is the type of error that occurred. In this case we incorrectly wrote 4 bytes (the size of an integer) into a memory place that we did not “own” (remember that we allocated 5 integer spaces for the array and wrote on a 6th one). There are various error types that might occur, the most common ones being:

- Invalid read/write of size n : We read/written from/to a memory space we do not “own” of a size n
- Use of uninitialised value of size n : We use a variable that we did not initialize (an array we did not initialize with 0, a pointer we did not malloc etc.)
- Conditional jump or move depends on uninitialised value(s) : Similar to use of uninitialized value, we use a variable that we did not initialize in a condition (if, while loop, for loop etc.)
- Invalid free() / delete / delete[] / realloc() : we de-allocated a value that we shouldn’t have. Usually occurs with double frees. (Freeing the same memory space twice).

It is important to keep in mind that the next line after the error, is the exact line of code where the error occurred. In this example: at 0x10916B: main (myprogram.c:6) , means that the error occurred in the function main, in the file program, at line 6. (0x10916B is the memory space in which the error occurred and very rarely concerns us.)

Interpreting Valgrind Messages (4)

HEAP SUMMARY : heap summary informs us, of any memory leaks in our program. In our case we can see that we allocated 1 time but de-allocated memory 0 times.

total heap usage: 1 allocs, 0 frees, 20 bytes allocated

LEAK SUMMARY : the kind of leaks that occur in our program. Indirectly lost and possibly lost bytes must always be 0. Keep in mind that to be certain that there are no memory leaks valgrind must present the following message:

All heap blocks were freed -- no leaks are possible

ERROR SUMMARY : the number of memory errors occurring in our program. Our program is free of memory errors only when the following message is present by valgrind:

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Debugging memory errors (1)

With the information provided by valgrind we can start fixing what errors occur in our programs. For this simple program we can start by changing the index of the array to 4 (More complex problem solving is provided in the lab exercises):

```
#include <stdio.h>
#include <stdlib.h>

int main(){
    int array* = malloc(sizeof(int)*5);
    array[4] = 10;
    return 0;
}
```

Debugging memory errors (2)

Valgrind know gives the following output. Now we must deal with the leaks:

```
==398== Memcheck, a memory error detector
==398== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==398== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==398== Command: ./myprogram
==398==
==398==
==398== HEAP SUMMARY:
==398==     in use at exit: 24 bytes in 1 blocks
==398==   total heap usage: 1 allocs, 0 frees, 24 bytes allocated
==398==
==398== LEAK SUMMARY:
==398==    definitely lost: 24 bytes in 1 blocks
==398==    indirectly lost: 0 bytes in 0 blocks
==398==      possibly lost: 0 bytes in 0 blocks
==398==    still reachable: 0 bytes in 0 blocks
==398==         suppressed: 0 bytes in 0 blocks
==398== Rerun with --leak-check=full to see details of leaked memory
==398==
==398== For lists of detected and suppressed errors, rerun with: -s
==398== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Debugging memory errors (3)

In this simple program finding where the memory leak is, is very easy. However, for larger programs with hundreds of lines of code, freeing the allocated memory correctly may prove tricky. Valgrind provides an option to help us. When calling valgrind with the option --leak-check=full, valgrind pinpoints exactly the origin of memory leaks. (In our example we run [valgrind --leak-check=full myprogram])

```
==399== Memcheck, a memory error detector
==399== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==399== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==399== Command: ./myprogram
==399==
==399==
==399== HEAP SUMMARY:
==399==     in use at exit: 24 bytes in 1 blocks
==399==   total heap usage: 1 allocs, 0 frees, 24 bytes allocated
==399==
==399== 24 bytes in 1 blocks are definitely lost in loss record 1 of 1
==399==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==399==    by 0x10915E: main (myprogram.c:5)
==399==
==399== LEAK SUMMARY:
==399==    definitely lost: 24 bytes in 1 blocks
==399==    indirectly lost: 0 bytes in 0 blocks
==399==      possibly lost: 0 bytes in 0 blocks
==399==    still reachable: 0 bytes in 0 blocks
==399==         suppressed: 0 bytes in 0 blocks
==399==
==399== For lists of detected and suppressed errors, rerun with: -s
==399== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Debugging memory errors (4)

We can see that from the orange code block that the leak comes from a memory allocation (malloc) at the line of code 5 in the main function inside the file myprogram.c .Now that we know where the leak comes from, we can free the memory space properly:

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    int array* = malloc(sizeof(int)*5);
    array[4] = 10;
    free(array);
    return 0;
}
```

Debugging memory errors (5)

Running valgrind again now shows no errors and no leaks. Our program is free of memory errors!

```
==405== Memcheck, a memory error detector
==405== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==405== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==405== Command: ./lol
==405==
==405==
==405== HEAP SUMMARY:
==405==     in use at exit: 0 bytes in 0 blocks
==405==   total heap usage: 1 allocs, 1 frees, 24 bytes allocated
==405==
==405== All heap blocks were freed -- no leaks are possible
==405==
==405== For lists of detected and suppressed errors, rerun with: -s
==405== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```