

Lecture 8 — Of Asgard & Hel

Jeff Zarnett

Norse Mythology

Everything came into creation in the gap between fire and ice, and the World Tree (Yggdrasil) connects the nine worlds. Asgard is the home of the Æsir, the Norse gods. Helheim, or simply Hel, is the underworld where the dead go upon their death. In Hel or Asgard (it's not entirely clear), there is Valhalla, hall of the honoured dead. Those who die in battle and are judged worthy will be carried to Valhalla by the Valkyries. There they will reside until they are called upon to aid in Odin's fight with the wolf Fenrir in Ragnarök¹, the doom of the gods². For the curious, humans live in the "middle realm", Midgård, surrounded by the serpent Jormungand, who will fight against Thor in Ragnarök. Thor will kill the serpent, but the serpent's poison will also finish off Thor³.

Aside from my obvious passion about the subject, why are we talking about Norse Mythology? We're going to examine some very useful tools for programming called Valgrind and Helgrind (also Cachegrind). Note that the -grind endings on those are pronounced like "grinned". Where do they take their names from? Valgrind is the gateway to Valhalla; a gate that only the worthy can pass. Helgrind is the gateway to, well, Hel. Which despite being the source of the English word "Hell", is not the place where sinners go. It's just the place where the dead go. Sadly, the authors of Cachegrind failed to choose a name that corresponds to a location in the nine worlds of Norse mythology. There are eight unused realms. They really could have.

But all of these, in program form, are analysis tools for your (usually) C and C++ programs. They are absolute murder on performance, but they are wonderful for finding errors in your program. To use them you will start the tool of your choice and instruct it to invoke your program. The target program then runs under the "supervision" of the tool. This results in running dramatically slower than normal, but you get additional checks and monitoring of the program. It's important to enable debugging symbols in your compile (-g option if using gcc) if you want stack traces to be useful.

Let's start with quick summary of each of the tools from [Dev15], followed by a more detailed explanation.

Valgrind (or Memcheck)

Valgrind is the base name of the project and by default what it's going to do is run the memcheck tool. The purpose of memcheck is to look into all memory reads, writes, and to intercept and analyze every call to malloc/free and new/delete. Thus, memcheck will check all memory accesses and allocations/deallocations, and can find problems like:

- Accessing uninitialized memory
- Reading off the end of an array
- Memory leaks (failing to free allocated memory)
- Incorrect freeing of memory (double free calls or a mismatch)
- Incorrect use of C standard functions like memcpy
- Using memory after it's been freed.
- Asking for an invalid number of bytes in an allocation (negative?!)

¹German: Götterdämmerung - "Twilight of the gods"

²Spoiler alert: this isn't going to end well for Odin.

³Sorry if I've just spoiled the plot of a Marvel movie.

These errors will be reported to the console when they occur and this will hopefully help you track the source of the problem.

I decided to run Valgrind with memcheck against the solution I wrote to the ECE 254 S15 exam question for searching an array using pthreads. I am happy to report that memcheck reports that the official solution has no memory leaks. If you do things right, you get something that looks like the example below.

```
jz@Loki:~/ece254$ valgrind ./search
==8476== Memcheck, a memory error detector
==8476== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8476== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==8476== Command: /usr/local/bin/search
==8476==
usage: search [arguments] [options]
arguments:
    for text
    in directory
options:
    -c | --case-sensitive
    -s | --show-filenames-only
==8476==
==8476== HEAP SUMMARY:
==8476==     in use at exit: 0 bytes in 0 blocks
==8476==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==8476==
==8476== All heap blocks were freed -- no leaks are possible
==8476==
==8476== For counts of detected and suppressed errors, rerun with: -v
==8476== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Okay, everything going perfectly is unlikely in anything other than a small program. The exam question I used this on is something like 62 lines (including blanks). So it's a trivial program. But I'll sabotage it a bit so we get a more interesting result. Suppose I delete from the code two of the free() calls.

```
jz@Loki:~/ece254$ valgrind ./search
==8678== Memcheck, a memory error detector
==8678== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8678== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==8678== Command: ./search
==8678==
Found at 11 by thread 1
Found at 22 by thread 3
==8678==
==8678== HEAP SUMMARY:
==8678==     in use at exit: 1,614 bytes in 4 blocks
==8678==   total heap usage: 17 allocs, 13 frees, 2,822 bytes allocated
==8678==
==8678== LEAK SUMMARY:
==8678==     definitely lost: 0 bytes in 0 blocks
==8678==     indirectly lost: 0 bytes in 0 blocks
==8678==     possibly lost: 0 bytes in 0 blocks
==8678==     still reachable: 1,614 bytes in 4 blocks
==8678==         suppressed: 0 bytes in 0 blocks
==8678== Rerun with --leak-check=full to see details of leaked memory
==8678==
```

```
==8678== For counts of detected and suppressed errors, rerun with: -v
==8678== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

If you take the program's suggestion to use the `-leak-check=full` then you end up with a bit more detail about where you made the mistake. Of course, it's important to know where to look; in the example below, lines 49 and 24 in the file `search.c` are the locations of the `malloc` calls that lack a matching call to `free`. It can't tell you where the call to `free` should go, only where the memory that isn't freed was allocated.

```
==8553== 16 bytes in 4 blocks are definitely lost in loss record 1 of 2
==8553==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8553==    by 0x40084D: search (search.c:49)
==8553==    by 0x4E3F181: start_thread (pthread_create.c:312)
==8553==    by 0x514F47C: clone (clone.S:111)
==8553==
==8553== 48 bytes in 4 blocks are definitely lost in loss record 2 of 2
==8553==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8553==    by 0x40074E: main (search.c:24)
```

But it's also important to learn what to ignore (or what's out of our hands). I decided to deploy Valgrind on the solution to the producer-consumer problem from ECE 254 and I ended up with a result that says:

```
==8734==    possibly lost: 544 bytes in 2 blocks
```

Hmm. Let's dig into that with the `-leak-check=full` option:

```
==8734== 272 bytes in 1 blocks are possibly lost in loss record 1 of 2
==8734==    at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8734==    by 0x4012E54: _dl_allocate_tls (dl-tls.c:296)
==8734==    by 0x4E3FDA0: pthread_create@@GLIBC_2.2.5 (allocatestack.c:589)
==8734==    by 0x400A57: main (mutex.c:64)
```

Looking in the file, at that line, we see a call to `pthread_create` and this is therefore probably nothing we need to do anything about.

From the Valgrind FAQ, how to read the leak summary:

- **Definitely lost** - a clear memory leak. Fix it.
- **Indirectly lost** - a problem with a pointer based structure (e.g., you've lost the head of the linked list, but the rest of the list is indirectly lost. Generally, fixing the definitely lost items should be enough to clear up the indirectly lost stuff).
- **Possibly lost** - the program is leaking memory unless weird things are going on with pointers where you're pointing them to the middle of an allocated block.
- **Still reachable** - this is memory that was still allocated that might otherwise have been freed, but references to it exist so it at least wasn't lost.
- **Suppressed** - you can configure the tool to ignore things and those will appear in the suppressed category.

Cachegrind

TODO: Fill in cache grind

Helgrind

Dynamic and static tools exist. They can help you find data races in your program. `helgrind` is one such tool. It runs your program and analyzes it (and causes a large slowdown).

Run with `valgrind -tool=helgrind <prog>`.

It will warn you of possible data races along with locations. For useful debugging information, compile your program with debugging information (`-g` flag for `gcc`).

Helgrind Output for Example.

```
==5036== Possible data race during read of size 4 at
           0x53F2040 by thread #3
==5036== Locks held: none
==5036==    at 0x400710: run2 (in datarace.c:14)
...
==5036==
==5036== This conflicts with a previous write of size 4 by
           thread #2
==5036== Locks held: none
==5036==    at 0x400700: run1 (in datarace.c:8)
...
==5036==
==5036== Address 0x53F2040 is 0 bytes inside a block of size
           4 alloc'd
...
==5036==    by 0x4005AE: main (in datarace.c:19)
```

References

[Dev15] Valgind Developers. Valgrind tool suite, 2015. Online; accessed 24-November-2015. URL: <http://valgrind.org/info/tools.html>.