ECE459: Programming for Performance Winter 2021

Lecture 9 — Of Asgard & Hel

Jeff Zarnett *2021-01-02*

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 Ragnarök[1], the doom of the gods[2]. 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 Thor[3].

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.

But all of these, in program form, are analysis tools for your programs. Running your program with one of these tools is something like 100× slower than normal execution, 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 if you want stack traces to be useful.

Valgrind (or Memcheck)

Valgrind is the base name of the project and by default it runs the memcheck tool. The purpose of memcheck is to look into all memory reads, writes, and to intercept and analyze every call to allocate and free memory. 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?)

[1]German: Götterdämmerung - "Twilight of the gods"
[2]Spoiler alert: this isn't going to end well for Odin.
[3]Sorry if I've just spoiled the plot of a Marvel movie.

These errors will be reported to the console when they occur. Ideally, this will help you find the source of the problem. Now you might be thinking that these problems cannot occur in Rust! Well, not quite. Here are some ways that you can end up with memory leaks in Rust:

- Explicitly calling the `leak` and `forget` functions
- Unsafe code and calling functions from other languages
- Reference-count cycles

So bad things can still happen. Ideally, you're going to see this:

```
==4209== HEAP SUMMARY:
==4209==     in use at exit: 0 bytes in 0 blocks
==4209==   total heap usage: 19 allocs, 19 frees, 3,457 bytes allocated
==4209==
==4209== All heap blocks were freed -- no leaks are possible
```

Okay, everything going perfectly is unlikely in anything other than a small program.

If you take the program's suggestion to use `--leak-check=full` then you end up with a bit more detail about where memory was allocated. It can't tell you where the call to `free` should go, only where the memory that isn't freed was allocated.

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.

Still, it's also important to learn what to ignore (or what's out of our hands). The stack trace that we see will point us at the cause of the problem. Sometimes, though, we end up with something where there's very little involvement of our own program: it's thread creation or a library or similar. What do we do?

Well—we have to consider carefully if there's anything we can do about it. In a library, there might be an associated cleanup call that we have forgotten to use. Or maybe not, and the problem is beyond our ability to fix. You will have to consider carefully and use your judgement! Sorry. I know it's much better when there's a clear yes or no, but software is complex.

We'll take some time to do some examples that show the kind of error that Valgrind reports; both those we can do something with and those that are outside our control.

Helgrind

The purpose of Helgrind is to detect errors in the use of threads. In a way, Helgrind is a pretty neat tool for improving performance, even though it doesn't actually directly speed anything up. When we take a single-threaded program and split it off into a multithreaded program, we may introduce a lot of errors (or at least, introduce the possibility of a lot of errors). Truthfully, humans are not very good at parallel thinking; we are very much sequential. But a program that is fast and wrong is probably less useful than one that is slow and correct. Can we make it faster and still have it be correct? That's the goal of Helgrind: after you parallelize your code, it will do some automatic checking of the code to determine where, if anywhere, there are concurrency problems. It can't prove that your program is correct (if only) but it can at least catch some of the common problems you might introduce when writing a parallel program. Helgrind classifies errors into three basic categories:

1. Misuses of the API;
2. Lock ordering problems; and
3. Data races.

The first category does not require much explanation. These are just some programming errors related to the pthread API calls. Some examples from [Dev15]:

- Unlocking a mutex that is unlocked;
- Deallocation of memory with a locked mutex in it; or
- Thread exit while holding a lock.

. . . and many more.

The second category of errors should be familiar to you from earlier as a source of potential deadlock.

Thread P
```
 1. wait( a )
 2. wait( b )
 3. [critical section]
 4. signal( a )
 5. signal( b )
```

Thread Q
```
 1. wait( b )
 2. wait( a )
 3. [critical section]
 4. signal( b )
 5. signal( a )
```

In this case, if the interleaving of these happens to work out in a couple of particular ways, then we get deadlock because thread P holds mutex `a` and thread Q holds mutex `b` and each waits for the mutex that the other one has. The example is slightly silly, of course, because it's super easy to see.

Helgrind builds a directed graph of lock acquisitions. When a thread acquires a lock, Helgrind checks to see whether a cycle exists. If so, then there is potential for a deadlock [Dev15]. Helgrind will report as an error the initial order (the first order seen is the one viewed as "correct") and the "incorrect" order that is the source of the potential problem. Really, though, all that matters is consistency—following the same order. You may change either of the acquisition orders to match the other.

How does Helgrind work? It examines the use of the standard threading primitives—lock, unlock, signal/post, wait, etc. Anything that implies there might be an ordering between events is taken and added to a directed acyclic graph that represents these dependencies. If memory is accessed from two different threads and there is no path through this directed acyclic graph that indicates an ordering, then Helgrind reports a race [Dev15]. Obviously, at least one of these accesses must be a write. (Recall: there is no read after read dependency).

Also cool: you can ask Helgrind to try to tell you about variable names (if it can) with the command line option `-read-var-info=yes`. Then it will tell you something interesting like:

```
==10454== Location 0x60104c is 0 bytes inside global var "var"
==10454== declared at datarace.c:3
```

These will give you indications of where you need to introduce synchronization of some kind (semaphore, mutex, condition variable, etc). The authors of Helgrind assume that if it tells you where the problem is, you will figure out what variables are affected and how to properly prevent data races. You might find this frustrating, in the sense of a serial complainer who thinks that he or she can just moan about what's wrong without bringing forward any suggestions about how to fix the problems.

We'll again take some time to examine some examples of each category of problem (as time allows).

References

[Dev15] Valgrind Developers. Helgrind: a thread error detector, 2015. Online; accessed 25-November-2015. URL: `http://valgrind.org/docs/manual/hg-manual.html`.