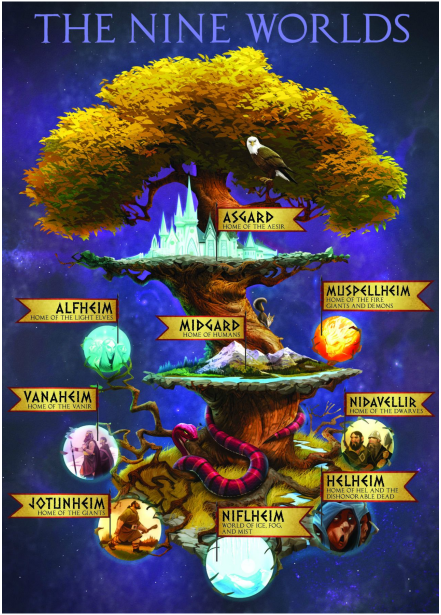
Lecture 24 — Of Asgard and Hel

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 15, 2019

Yggdrasil, the World Tree



Ginnungagap

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 clear), there is Valhalla, hall of the honoured dead.

Carry We, Who Die In Battle...

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.

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.

Across the Rainbow Bridge, To Valhalla...

We’re going to examine some very useful tools for programming called Valgrind and Helgrind (also Cachegrind).

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.

I may be interested in this subject...

Through the Bifrost

But all of these are analysis tools for your C and C++ programs.

They are absolute murder on performance, but they are wonderful for finding errors in your program.

To use them, start the tool of your choice and instruct it to invoke your program.

The target program then runs under the “supervision” of the tool.

Remember to enable debugging symbols in your compile.

Valgrind (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`.

Is Your Program Worthy?

Is Your Program Worthy?

Memcheck can find problems like:

- Accessing uninitialized memory
- Reading off the end of an array
- Memory leaks
- Incorrect freeing of memory
- 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

Valgrind Output

```
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)
```

No, It’s Because I’m Awesome

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.

Hence the word... Sabotage

```
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)
```

Send the Plumbers to the Watergate

Take the program's suggestion to use the `-leak-check=full`.

You get a bit more detail about where you made the mistake.

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.

Fixing the Leaks

```
==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)
```

Not My Problem!

But it’s also important to learn what to ignore.

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:

Ignore and Continue

```
==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.

Reading the Output

- **Definitely lost**
- **Indirectly lost**
- **Possibly lost**
- **Still reachable**
- **Suppressed**

Helheim

Credit: Norse Legends Fandom User OlaTansta

What Fresh Hel is This

The purpose of Helgrind is to detect errors in the use of POSIX pthreads.

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 single-threaded program and split it off into a multithreaded program, we may introduce a lot of errors.

Humans are not very good at parallel thinking; we are very much sequential.

What Fresh Hel is This

But a program that is fast and wrong is probably less useful than one that is slow and correct.

But can we make it faster and still have it be correct?

That's the goal of Helgrind: 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.

Hel Hath No Fury...

Helgrind will categories errors into three basic categories:

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

Misuse of the API

The first category does not require much explanation:

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

… and many more.

API Misuse Error Output

```
Thread #1 unlocked a not-locked lock at 0x7FEFFFA90
   at 0x4C2408D: pthread_mutex_unlock (hg_intercepts.c:492)
   by 0x40073A: nearly_main (tc09_bad_unlock.c:27)
   by 0x40079B: main (tc09_bad_unlock.c:50)
  Lock at 0x7FEFFFA90 was first observed
   at 0x4C25D01: pthread_mutex_init (hg_intercepts.c:326)
   by 0x40071F: nearly_main (tc09_bad_unlock.c:23)
   by 0x40079B: main (tc09_bad_unlock.c:50)
```

Type Two Errors

The second category of errors should be familiar to you:

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 )
```

Potential Deadlock!

Interleaving Leaves Locks

Risk of deadlock:thread P holds mutex a and thread Q holds mutex b.

Each waits for the mutex that the other one has.

The example is slightly silly, of course, because it’s super easy to see.

There will not necessarily be an obvious (alphabetical) order.

Graph Theory Strikes Back

Helgrind builds a directed graph of lock acquisitions, so that when a thread acquires a lock, the graph is checked to see if a cycle exists.

It will report as an error the initial order (the first order seen is the one viewed as “correct”) and the the “incorrect” order that is the source of the potential problem.

Really, though, all that matters is consistency.

Order of a Mutex Lock

```
Thread #1: lock order "0x7FF0006D0 before 0x7FF0006A0" violated

Observed (incorrect) order is: acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x400825: main (tc13_laog1.c:23)

 followed by a later acquisition of lock at 0x7FF0006D0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x400853: main (tc13_laog1.c:24)

Required order was established by acquisition of lock at 0x7FF0006D0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40076D: main (tc13_laog1.c:17)

 followed by a later acquisition of lock at 0x7FF0006A0
   at 0x4C2BC62: pthread_mutex_lock (hg_intercepts.c:494)
   by 0x40079B: main (tc13_laog1.c:18)
```

Category 3

The third category we have discussed already.

Recall the earlier definition of a race condition.

Helgrind looks for when two threads access the same memory location without using locks.

It’s a Race!

```
jz@Loki:~/ece459$ valgrind --tool=helgrind ./datarace
==10389== Helgrind, a thread error detector
==10389== Copyright (C) 2007-2013, and GNU GPL’d, by OpenWorks LLP et al.
==10389== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==10389== Command: ./datarace
==10389==
==10389== ---Thread-Announcement------------------------------------------
==10389==
==10389== Thread #1 is the program’s root thread
==10389==
==10389== ---Thread-Announcement------------------------------------------
==10389==
==10389== Thread #2 was created
==10389==    at 0x515543E: clone (clone.S:74)
==10389==    by 0x4E44199: do_clone.constprop.3 (createthread.c:75)
==10389==    by 0x4E458BA: pthread_create@@GLIBC_2.2.5 (createthread.c:245)
==10389==    by 0x4C30C90: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==10389==    by 0x40068D: main (datarace.c:12)
==10389==
==10389== ----------------------------------------------------------------
```

It’s a Race

```
==10389== Possible data race during read of size 4 at 0x60104C by thread #1
==10389== Locks held: none
==10389==    at 0x40068E: main (datarace.c:13)
==10389==
==10389== This conflicts with a previous write of size 4 by thread #2
==10389== Locks held: none
==10389==    at 0x40065E: child_fn (datarace.c:6)
==10389==    by 0x4C30E26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==10389==    by 0x4E45181: start_thread (pthread_create.c:312)
==10389==    by 0x515547C: clone (clone.S:111)
==10389==
==10389== ----------------------------------------------------------------
==10389==
==10389== Possible data race during write of size 4 at 0x60104C by thread #1
==10389== Locks held: none
==10389==    at 0x400697: main (datarace.c:13)
==10389==
==10389== This conflicts with a previous write of size 4 by thread #2
==10389== Locks held: none
==10389==    at 0x40065E: child_fn (datarace.c:6)
==10389==    by 0x4C30E26: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==10389==    by 0x4E45181: start_thread (pthread_create.c:312)
==10389==    by 0x515547C: clone (clone.S:111)
```

Always Two, There Are

Note that we get two stack traces here: we have a read after write, and a write after write.

Why?

Because the operation in question is `var++` which necessitates fetching the current value of `var` (reading it) and incrementing it (then writing it back).

Behind the Scenes

It examines the use of the standard threading primitives - lock, unlock, signal/post, wait...

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 a race is reported.

Although obviously at least one of these accesses must be a write.

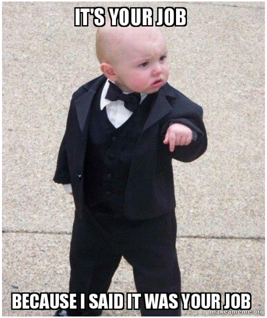
Name & Shame

You can ask Helgrind to try to tell you about variable names (if it can) with the command line option `-read-var-info=yes`.

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

Fixing it Isn't My Problem

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.