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. 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 a quick summary of each of the tools from [Dev15a], 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?)

---

[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.

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 `--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. Or is it? In-class example will tell...

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.

Helgrind

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 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 pthreads 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 [Dev15c]:

- 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.

A quick example of an error message from Helgrind, also from [Dev15c]:

```
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)
```

The second category of errors should be familiar to you from your operating systems class as a source of potential deadlock.

Thread P	**Thread Q**
1. wait(a)	1. wait(b)
2. wait(b)	2. wait(a)
3. [critical section]	3. [critical section]
4. signal(a)	4. signal(b)
5. 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; in normal code they would probably be separated by some number of lines, and the mutexes will probably not be called `a` and `b` (isn't using meaningful variable names one of those things you were supposed to learn to do in introductory programming?) and there therefore will not necessarily be an obvious (alphabetical) order.

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 [Dev15c]. 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. See the example below [Dev15c]:

```
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)
```

The third category we have discussed already. Recall the earlier definition of a race condition. This is a difficult problem to find sometimes; what Helgrind will do is examine where multiple threads are accessing shared memory without the use of locks. Let's cut to the chase and see it in action:

```
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== ----------------------------------------------------------------
==10389==
==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)
==10389==
==10389==
==10389== For counts of detected and suppressed errors, rerun with: -v
==10389== Use --history-level=approx or =none to gain increased speed, at
==10389== the cost of reduced accuracy of conflicting-access information
==10389== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

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).

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 [Dev15c]. 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.

A note about Helgrind for later, since we're going to dive into OpenMP. Helgrind + OpenMP + Linux can cause the occasional problem in GCC 4.2 and 4.3 (perhaps fixed in GCC 4.6.3[4]). But the "futex" system call in Linux is not visible to Helgrind so it won't figure things out. If you have old GCC, you can rebuild it with the option -disable-linux-futex and it won't generate "futex" syscalls.

---

[4]http://valgrind-users.narkive.com/QARaSvBN/helgrind-and-openmp-on-gcc

Cachegrind

Cachegrind is another tool in the package and this one is much more performance oriented than the other two tools. Yes, Valgrind's memcheck and Helgrind look for errors in your program that are likely to lead to slowdowns (memory leaks) or make it easier to parallelize (spawn threads) without introducing errors. Cachegrind, however, does a simulation of how your program interacts with cache and evaluates how your program does on branch prediction. As we discussed earlier, cache misses and branch mispredicts have a huge impact on performance.

Recall that a miss from the fastest cache results in a small penalty (perhaps, 10 cycles); a miss that requires going to memory requires about 200 cycles. A mispredicted branch costs somewhere between 10-30 cycles. All figures & estimates from the cachegrind manual [Dev15b].

Cachegrind reports data about:

- The First Level Instruction Cache (I1) [L1 Instruction Cache]

- The First Level Data Cache (D1) [L1 Data Cache]

- The Last Level Cache (LL) [L3 Cache].

Unlike for normal Valgrind operation, you probably want to turn optimizations on (`-02` or perhaps `-03` in `gcc`). You still want debugging symbols, of course, but enabling optimizations will tell you more about what is going to happen in the released version of your program.

If I instruct cachegrind to run on the search example (same one from above), using the `-branch-sim=yes` option because by default it won't show it:

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search
==16559== Cachegrind, a cache and branch-prediction profiler
==16559== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==16559== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==16559== Command: ./search
==16559==
--16559-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16559==
==16559== I   refs:      310,670
==16559== I1  misses:      1,700
==16559== LLi misses:      1,292
==16559== I1  miss rate:    0.54%
==16559== LLi miss rate:    0.41%
==16559==
==16559== D   refs:      114,078  (77,789 rd   + 36,289 wr)
==16559== D1  misses:      4,398  ( 3,360 rd   +  1,038 wr)
==16559== LLd misses:      3,252  ( 2,337 rd   +    915 wr)
==16559== D1  miss rate:     3.8% (   4.3%    +    2.8%  )
==16559== LLd miss rate:     2.8% (   3.0%    +    2.5%  )
==16559==
==16559== LL refs:        6,098  ( 5,060 rd   +  1,038 wr)
==16559== LL misses:      4,544  ( 3,629 rd   +    915 wr)
==16559== LL miss rate:     1.0% (   0.9%    +    2.5%  )
==16559==
==16559== Branches:      66,622  (65,097 cond +  1,525 ind)
==16559== Mispredicts:    7,202  ( 6,699 cond +    503 ind)
==16559== Mispred rate:    10.8% (  10.2%    +   32.9%   )
```

So we see a breakdown of the instruction accesses, data accesses, and how well the last level of cache (L3 here) does.

Why did I say enable optimization? Well, here's the output of the search program if I compile with the -O2 option:

```
jz@Loki:~/ece254$ valgrind --tool=cachegrind --branch-sim=yes ./search
==16618== Cachegrind, a cache and branch-prediction profiler
==16618== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==16618== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==16618== Command: ./search
==16618==
--16618-- warning: L3 cache found, using its data for the LL simulation.
Found at 11 by thread 1
Found at 22 by thread 3
==16618==
==16618== I   refs:       306,169
==16618== I1  misses:       1,652
==16618== LLi misses:       1,286
==16618== I1  miss rate:     0.53%
==16618== LLi miss rate:     0.42%
==16618==
==16618== D   refs:       112,015  (76,522 rd   + 35,493 wr)
==16618== D1  misses:       4,328  ( 3,353 rd   +    975 wr)
==16618== LLd misses:       3,201  ( 2,337 rd   +    864 wr)
==16618== D1  miss rate:      3.8% (   4.3%     +    2.7%  )
==16618== LLd miss rate:      2.8% (   3.0%     +    2.4%  )
==16618==
==16618== LL refs:          5,980  ( 5,005 rd   +    975 wr)
==16618== LL misses:        4,487  ( 3,623 rd   +    864 wr)
==16618== LL miss rate:      1.0% (   0.9%     +    2.4%  )
==16618==
==16618== Branches:        65,827  (64,352 cond +  1,475 ind)
==16618== Mispredicts:      7,109  ( 6,596 cond +    513 ind)
==16618== Mispred rate:     10.7% (  10.2%      +   34.7%   )
```

Interesting results: our data and instruction miss rates went down marginally but the branch mispredict rates went up! Well, sort of—there were fewer branches and thus fewer we got wrong as well as fewer we got right. So the total cycles lost to mispredicts went down. Is this an overall win for the code? Yes.

In some cases it's not so clear cut, and we could do a small calculation. If we just take a look at the LL misses (4 544 vs 4 487) and assume they take 200 cycles, and the branch miss penalty is 200 cycles, it went from 908 800 wasted cycles to 897 400; a decrease of 11 400 cycles. Repeat for each of the measures and sum them up to determine if things got better overall and by how much. Also be sure that you're reasoning about a realistic workload.

Cachegrind also produces a more detailed output file, titled cachegrind.out.<pid> (the PID in the example is 16618). This file is not especially human-readable, but we can ask the associated tool cg_annotate to break it down for us, and if we have the source code available, so much the better, because it will give you line by line information. That's way too much to show even in the notes, so it's the sort of thing I can show in class (or you can create for yourself) but here's a small excerpt from the search.c example:

```
--------------------------------------------------------------------------------
-- Auto-annotated source: /home/jz/ece254/search.c
--------------------------------------------------------------------------------
 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
```

```
127    1    1 96    3    0 4    0    0 23 11    0    0          for ( int i = arg->startIndex; i < arg->endIndex; ++i ) {
147    0    0 84    3    2 0    0    0 21  9    0    0              if ( array[i] == arg->searchValue ) {
  6    0    0  4    0    0 2    0    0  0  0    0    0                  *result = i;
  2    0    0  0    0    0 0    0    0  0  0    0    0                  break;
  .    .    .  .    .    . .    .    .  .  .    .    .              }
  .    .    .  .    .    . .    .    .  .  .    .    .          }
```

Cachegrind is very…verbose…and it can be very hard to come up with useful changes based on what you see…assuming your eyes don't glaze over when you see the numbers. Probably the biggest performance impact is last level cache misses (those appear as DLmr or DLmw). They have the highest penalty. You might also try to look at the Bcm and Bim (branch mispredictions) to see if you can give some better hints about what the likelihood of branch prediction is [Dev15b]. Of course, to learn more about how Cachegrind actually does what it does and how it runs the simulation, the manual is worth reading. Not that anybody reads manuals anymore…Just give it a shot, when you get stuck, google the problem, click the first stack overflow link result…

References

[Dev15a] Valgind Developers. Valgrind tool suite, 2015. Online; accessed 24-November-2015. URL: `http://valgrind.org/info/tools.html`.

[Dev15b] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler, 2015. Online; accessed 25-November-2015. URL: `http://valgrind.org/docs/manual/cg-manual.html`.

[Dev15c] Valgrind Developers. Helgrind: a thread error detector, 2015. Online; accessed 25-November-2015. URL: `http://valgrind.org/docs/manual/hg-manual.html`.