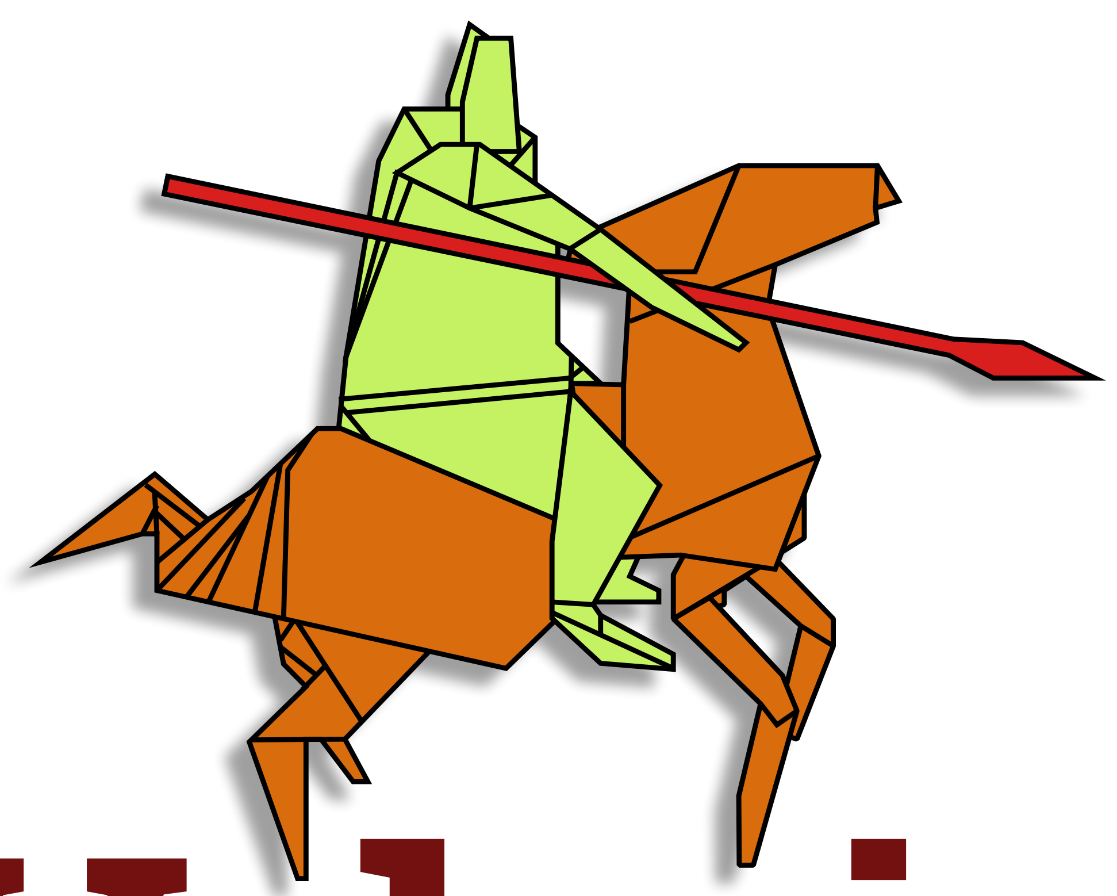


Valgrind on FreeBSD

BY PAUL FLOYD

I first started using Valgrind in the early 2000s. Previously, I had a fair bit of experience with Purify (now Unicom PuifyPlus) on Solaris/SPARC. To be honest, I wasn't that impressed with Valgrind. Sure, it didn't need a special build process, but it lacked the ability to interact with a debugger.

Switching briefly to FreeBSD, the first version I installed was 2.1 back in late 1995. Like Valgrind, at first, I was not that impressed. At least it was "a unix" on my home PC if I needed it. I continued to dabble with FreeBSD, installing new versions from time to time. My main home system(s) were OS/2 till the late 90s, Solaris for long time until it went into suspended animation with 11.4. I also got a MacBook in 2007 which is mainly for "desktop" stuff — I don't find developing on macOS to be a gratifying experience.

I have always been a bit of a believer in Quality. I had a short course on Product Quality at university which converted me to the cause. Much later, I got round to reading W. Edwards Deming. Though the writing is rough around the edges, the message is strong and clear. Since I'd studied Electronics, it was plainly obvious the benefits from quality processes that Japanese companies had reaped in the second half of the 20th century. I ended up working as a software developer in the domain of Electronics simulation. Not surprisingly, I carried on using tools like Valgrind.

I decided that it was time to start giving back to the open-source community.

About five years ago, I decided that it was time to start giving back to the open-source community. Since I was already expert in using Valgrind and had already dabbled a tiny bit with the source, it was the logical project for me. I hesitated a bit between working on the macOS and FreeBSD Valgrind ports. Two things put me off macOS — frequent major OS and userland changes that break everything, and the difficulty of getting help from within Apple. There are the XNU code source dumps and a few books, but after that you are on your own. I plumped for FreeBSD. That also suited me because I was looking to switch away from Solaris. There's been a lot of cross-fertilization between Illumos and FreeBSD so I thought that would ease the transition. In the meantime, macOS lingers in the official Valgrind repo, but it hasn't really been usable since version 10.12 in 2016.

History of Valgrind

Valgrind is now a bit over 20 years old. It started off on i386 Linux. Over the years, several other CPU architectures have been added (amd64, MIPS, ARM, PPC and s390x) as well as other operating systems (macOS, Solaris, and, most recently, FreeBSD).

The tools have continued to evolve over those 20 years. Since the initial version in 2002, the tools added were

2002 memcheck
2002 helgrind
2002 cachegrind
2004 massif
2006 callgrind
2008 drd
2009 exp-bbv
2018 DHAT

In addition, there are several tools that are maintained (or not) out-of-tree.

The development of Valgrind has been carried out by a small number of people - about twenty have made significant contributions. A few corporations have lent a hand. RedHat/IBM is probably the one that has contributed the most. Sun did contribute while Solaris was being actively developed. Apple also contributed until they suddenly became GLP averse.

History of Valgrind on FreeBSD

Valgrind on FreeBSD had a very long and checkered history. I won't mention everyone who has contributed (and I'm not even sure that I have the full list as some of the source code repos are no longer accessible). Doug Robson did a lot of the initial work in 2004. The next torch bearer was Stan Sedov who maintained the port from 2009 to 2011. There was a protracted push to get the FreeBSD source accepted upstream at that time, but it didn't quite make it. The upstream maintainers were quite strict with their quality bar, and the FreeBSD port kept getting close, but was never good enough. Secondly, someone needs to maintain the port, preferably a member of the Valgrind team. I don't know why that never happened. I've been maintaining the FreeBSD port since Apr 2021, and I've had a Valgrind commit bit for a bit over 4 years. Now, I'm the main contributor to Valgrind.

Valgrind on FreeBSD had a very long and checkered history

The most recent big change was adding support for aarch64. I added the port to this CPU in April 2024, in time for the 3.23 release of Valgrind.

The Valgrind Tools

Before I dive into the internals of Valgrind, I'll give a quick overview of the tools.

Memcheck

This is the tool that most people think of when they refer to Valgrind. It is the default tool. The main things memcheck does are validate that memory reads are from initialized memory and that reads and writes are within the bounds of blocks of allocated heap memory. The missing piece there is checking the bounds of stack memory — that requires instrumentation.

DRD and Helgrind

These two tools are both thread hazard detection tools. They will detect accesses to memory from different threads that do not use some sort of locking mechanism. They will also warn of errors in the use of the pthread functions. The difference between the two is that Helgrind will try to give the error context for all the threads involved with a hazard. DRD only gives details for one thread.

Callgrind and Cachegrind

These two tools are for CPU profiling. Callgrind profiles function calls. Cachegrind is historically used to profile CPU instructions with a basic cache and branch predictor model. These models were never very accurate and now they are quite unrealistic. On top of that, Valgrind does not do any speculative execution. For those reasons, the current version of Valgrind no longer uses cache simulation with Cachegrind by default. Some people like the precise nature of the instruction counts. Personally, I usually prefer sampling profilers like Google perftools, (port devel/google-perftools), Linux perf and gprofng, especially for large problems (runtimes in hours or days and memory use in the 100s of Gbytes).

Massif and DHAT

These two tools are memory profiling tools. Massif profiles memory over time. Personally, I find it is overkill. Other tools exist that can usually produce equally good profiles without the Valgrind overhead — Google perftools again, and HeapTrack (port devel/heaptrack). There is an exception to this. If your application makes heavy use of a custom allocator based on mmap or statically links with a malloc library, then those alternative tools won't work. Massif doesn't need to interpose allocation functions in a shared library, and it also has an option to profile memory at the mmap level. DHAT is the hidden gem in the Valgrind suite. This tool profiles memory accesses to heap memory. This gives you information that will allow you to see which bits of memory are heavily used, memory that remains allocated for a long time, memory that is never used. For memory blocks that aren't too large, it will also generate access histograms for that block. From that, you can see holes or unused members in structs and classes. You can also infer access patterns which might help in reordering members to get them on the same cache line.

If your application makes heavy use of a custom allocator based on mmap or statically links with a malloc library, then those alternative tools won't work.

Valgrind Basics

Non-dependencies

In order to allow Valgrind to execute all of the client code (not just from main(), but from the first instructions in the ELF file at program startup) and also to avoid any conflicts with things like stdio buffers, Valgrind does not link with libc or any external libraries. I sometimes joke that this is not so much C++ as C- -. That means Valgrind has its own implementation of a subset of libc. To keep the function names distinct, it uses macros as a kind of pseudo-namespace. The Valgrind version of *printf* is *VG_(printf)* (great fun for code navigation!). This also means that we can't just add a third-party library and use it. The library needs to be ported to use Valgrind's libc subset. An example is that there is currently a bugzilla item to add support for zstd compressed DWARF sections.

Paranoid programming

Valgrind is very cautious and makes extensive use of asserts that are enabled in release builds. That makes it a little slower, but there are just so many things that can go wrong. It's best to be honest and bomb straight away rather than try to fake it and limp on.

Valgrind has extensive verbose and debug messages. You can crank up the debug/verbosity levels by repeating -v and -d up to 4 times each. In addition to that there are several more targeted trace options like —trace-syscalls=yes. Debugging in Valgrind can be quite difficult and all these outputs can be a big aid when developing features. They are also useful for support, e.g., asking the user to upload logs to the Valgrind bugzilla.

Code complexity

Valgrind itself is a bit of a beast. One of the hardest things about working on Valgrind is that it touches on so many things. There is virtualization for four families of CPUS (Intel/AMD, ARM, MIPS and PPC with a few sub-variants). Each of those has multi-thousand-page manuals. You often need to know all about opcodes to the level of every bit that they might change. You need a good knowledge of C, C++, and POSIX. You need to be able to tell which OS syscalls need special handling. Knowing the ELF standard is important - we've had issues because lld and mold do things differently. As well as ELF there is DWARF for the debuginfo. So far, I've only covered the core of Valgrind.

Despite the complexity, I don't think that Valgrind contains a huge amount of code

Despite the complexity, I don't think that Valgrind contains a huge amount of code. A clean git clone, not counting the regression tests, is about 500kloc. With the regression tests, that goes up to about 750kloc — while there are only 1000 or so regression tests, some of them are enormous, covering vast numbers of combinations of bit patterns, using scripts to generate all combinations of inputs to test.

The tools themself take up barely 10% of the code. It's the CPU emulation and the "core" that dominate. The core consists of many things — libc replacement, syscall wrappers, memory management, gdb interface, DWARF reader, signal handling, internal data structures and function redirections.

One further complication when developing Valgrind is that, being entirely static, you can't build it with sanitizers. However, you can run Valgrind inside Valgrind! This requires a special build so you end up with an outer Valgrind and inner Valgrind which is the guest of the outer Valgrind, and a guest executable, guest of the inner Valgrind. Of course, that makes everything slower to another degree. I do use the free Coverity Scan service to run static analysis on FreeBSD builds of Valgrind. That mostly finds the usual kinds of false positives but has found a few real bugs including some that I added. I still need to do some work to provide code models for Valgrind's internal libc replacements, particularly the allocation functions.

Valgrind at Runtime

Guest execution

The CPU emulation in Valgrind is called VEX (not to be confused with Intel Vector EXtensions). I'm not sure of the origins of VEX, possibly "Valgrind Emulation."

When Valgrind runs, there is just one process — the host. Ptrace (as used by debuggers such as lldb and gdb) is not used. The guest (sometimes referred to as the client) executable runs within the host using Dynamic Binary Instrumentation (DBI). To perform the instrumentation, it performs dynamic recompilation using Just-In-Time (JIT) compilation. That proceeds as follows:

- Read a bunch of machine code.
- Translate these into Valgrind Intermediary Representation (IR) — this is the same sort of representation that compilers use, and by no coincidence Julian Seward also once worked on the Glasgow Haskell Compiler
- Instrument the IR depending on the needs of the tool
- Perform optimization and rewriting on the IR
- Store the JITted opcodes in a cache and execute them

Memory separation

Valgrind has its own memory manager. It maintains a strict separation of memory that is used by the host and memory that is used for the guest. Many of the tools replace the C and C++ allocation and deallocation functions. For these tools, it is the Valgrind memory manager that handles everything. Tools like cachegrind and callgrind do not replace the memory allocators (and thus, they include the allocators in their performance profiling).

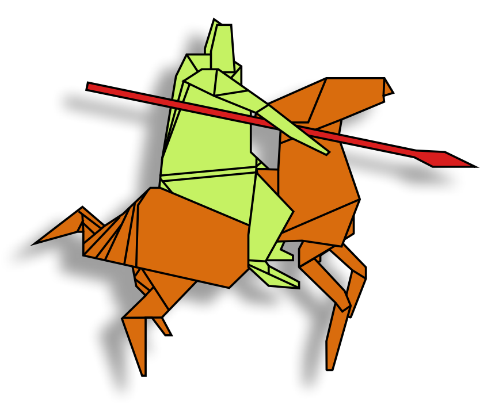
Valgrind startup

Valgrind starts off in assembler in its own _start routine (no libc, remember?), and the first things that it does is create a temporary stack for itself, set up logging, and set up the heap allocator. The point I want to make here is that there is little room for mistakes. If something goes wrong, if you are lucky, you just won't get filenames and line numbers in error messages. If you're not lucky, then all you will get is a load of hex addresses in a stack trace. As you can imagine, you fail pretty quickly if you don't have a stack. Once Valgrind has done all its internal setup, it is ready to start the guest executable on the synthetic CPU. It creates another stack for itself that has a configurable size, and it starts the guest executable. From the perspective of the guest executable, it is just like it were running natively.

Handling syscalls, threads, and signals

Valgrind intercepts all system calls. Fortunately, most of them do nothing or just have a few checks (do the registers contain initialized memory?) and then get forwarded to the kernel. More complicated syscalls will have a behavior that depends on some operation code (like umtx_op and ioctl). Finally, there are syscalls that do not get forwarded to the kernel that need to be implemented by Valgrind. An example of that is 'getcontext' where Valgrind needs to fill the context from its synthetic CPU rather than letting the kernel fill it from the context of the Valgrind host.

One tricky thing is that the code running on the virtual CPU needs to stay on the virtual CPU. While Valgrind executes some guest code natively on the physical CPU, that's usually extremely limited in scope. If the control flow of the guest escapes back to the physical CPU, things will go horribly wrong. I'll give two examples of the contortions that are needed to ensure Valgrind stays in control. Firstly, thread creation. When there are calls to 'pthread_create' Valgrind needs to make sure that the OS doesn't run the function passed in the third argument. Instead, it needs to hook the third argument with a "run_thread_in_valgrind" function. Similarly, for signals Valgrind needs to ensure that guest signal handlers run under



Valgrind, and then that the return from the signal handler goes back to running under Valgrind. These things require some very hacky code. Valgrind also must do a lot of juggling of signal masks. When the guest is running, signals are blocked with the host polling and handling signals itself. When there is a syscall, signals are unmasked, the syscall performed, and signals masked again. Without this little dance, blocking syscalls would not be interruptible.

The Valgrind Port

When I started looking at the Valgrind port, it was in a bad state. As mentioned earlier, there was a push from 2009 to 2011 to get the port upstreamed. From 2011 to 2018 it slipped back to minimal maintenance.

Valgrind on amd64 was broken due to a change to add large file support to the 'stat' family of functions. A couple of people had found patches for that. I386 was broken in several ways. There were no FreeBSD-specific regression tests. Valgrind contains many tests that run on all platforms, and then all combinations of OS and CPU architecture (e.g., amd64, freebsd and amd64-freebsd). There are 600 or so of these common tests. Linux amd64 has about 200 or so tests on top of those common tests. I don't remember how many of those common tests were passing and failing, probably not much more than half. Fortunately, there was a large amount of low hanging fruit. After sorting some serious issues on i386, after about six months I had about 90% of the regression tests working. That may sound good, but there were still some serious limitations. Slogging through the remaining 10% really was a case of the last 10% taking 90% of the time

When I started looking at the Valgrind port, it was in a bad state.

War Stories

Signals leading to asserts

Signals. Oh my, I did have a hard time at first understanding all this. When running natively, signals will do the following:

- The kernel synthesizes a ucontext block which contains the address where the signal occurred and a call frame on the stack (or the alt stack), with the call frame return address set to the 'retpoline' (a small asm function for returning from signal handlers)
- The kernel transfers the running exe to the signal handler
- The signal handler does its stuff and returns
- The retpoline calls the sigreturn syscall
- The kernel gets the original address before the signal from the content and transfers execution there

On Linux, that picture holds for both non-threaded and threaded applications. On FreeBSD, once you link with libthr, the picture changes. 'thr_sighandler' replaces the user signal handler. This does some things like signal masking. It calls the user signal handler and calls sigreturn itself.

Valgrind can't let guest code execute. So, it handles all possible signals. It synthesizes its own context with a bit more information. It replaces the guest signal handler with its own *run_signal_handler_in_valgrind* function. The return address has set its own retpoline that will call a *valgrind_sigreturn* that will transfer execution of the guest back to where came from. What could possibly go wrong? As it turns out, almost everything. There have been at least two things that were broken in this flow that I've dealt with.

The first was a very small code change. Valgrind crashed when returning from guest signal handler functions on i386. After a lot of debugging, I narrowed this down to the assembly retpoline function VG_(*x86_freebsd_SUBST_FOR_sigreturn*). At some point, there must have been some change to the size of the ucontext structure. VG_(*x86_freebsd_SUBST_FOR_sigreturn*) was looking for the return address at the wrong offset - 0x14 instead of 0x1c. That meant the virtual CPU was resuming execution at some rubbish address. Boom! That soon hit an assert.

My second big battle with signals was intermittent. If a signal arrives when Valgrind is executing "ordinary" guest code on the virtual CPU, that is great because it knows exactly where to resume from. But what happens if a signal arrives during a syscall? Things are a lot more complicated because syscalls are one of the places where Valgrind is sort-of letting the guest run on the physical CPU. Valgrind can't make guest syscalls within its global lock. The syscall might block and that would cause multi-threaded processes to hang. Instead, it releases the lock and then makes the syscall. Now, if an interrupt happens in the window when the lock is down, Valgrind needs to try to figure out exactly where it happened so that it can decide whether it needs to be restarted or not. To do that, the machine code function that does the guest syscall, ML(*do_syscall_for_client_WRK*), has an associate table of addresses that correspond to setup, restart, complete, committed and finished. That worked well, but occasionally would fail with an assert. The problem was with how the syscall status gets set. On Linux, it's just in the RAX register, and that gets returned from the small assembly function, so nothing special needs to be done. On FreeBSD (and Darwin), it's saved in the carry flag. That needs a function call to set the carry flag in the synthetic CPU. And if a signal arrives when Valgrind is in the *LibVEX_GuestAMD64_put_rflag_c* function call? That case wasn't handled — resulting in the assert. Unfortunately, in C there's no easy way to tell which function the instruction pointer is executing in. You can take the address of the start of the function easily enough. But where is the end? I did consider using the Valgrind DWARF debuginfo (which should always be present and Valgrind has DWARF reading code built in). In the end, I went for an ugly and non-standard way. I took the address of a dummy function just after *LibVEX_GuestAMD64_put_rflag_c*. It happened to work on i386 and amd64 even though there is no guarantee that the compiler and linker will lay out functions in the same way that they appear in source files. Later, when I worked on the aarch64 port this did not work because the carry flag setting function uses several helper functions, and they aren't all laid out in the same order. So, I switched to setting a global variable from the assembler routine that makes guest system calls.

What happens if a signal arrives during a syscall?

GlusterFS swapcontext crashes

One more war story. This was one of the first bug reports I got after I released the rebooted FreeBSD Valgrind. A user running GlusterFS was getting crashes in Valgrind. After quite a bit of toing and froing, asking for log files and traces, I narrowed it down to the swapcontext syscall. It turned out that switched-to context has two pointers to the signal mask in the thread state that Valgrind saves. Only the first of them was getting set. Another case of several days of debugging for a one-line code change.

FreeBSD issues

The work I've done on Valgrind has also revealed a few bugs in FreeBSD. I had to debug one of those early on when I was working with i386 binaries running on amd64. I didn't have problems with i386 on i386 or amd64 on amd64 but i386 on amd64 was crashing early in the guest startup, in the link loader (lib rtld). Eventually I discovered that this was a problem with the detection of the pagesize. Normal standalone applications have this information in their auxiliary vector (auxv) as AT_PAGESZ (the actual page size) and AT_PAGESIZES (a pointer to table of possible page sizes). Valgrind synthesizes the aux for the guest, but at the time, it ignored AT_PAGESIZES. No problem, rtld has a fallback to use the HW_PAGESIZE sysctl. I386 has two possible page sizes, but amd64 has three possible page sizes. Unfortunately, what was happening was that rtld running on amd64 was using the size of three for PAGESIZES, but the i386 kernel component was using a size of two. The result was that the sysctl was returning ENOMEM.

The Elephant in the Room — Sanitizers

Why bother using Valgrind now that we have the sanitizers? I'll also turn that question around and ask why use sanitizers when we have Valgrind? Roughly Address Sanitizer and Memory Sanitizer are equivalent to Memcheck, and Thread Sanitizer is equivalent to DRD and Helgrind. UB sanitizer has no Valgrind equivalent.

There's one case when using Valgrind is simply out of the question. That is, if you are using an unsupported CPU architecture. Valgrind on FreeBSD only supports amd64, i386, and aarch64. If you are using another architecture, then Valgrind is out of the question. Next, Valgrind is lagging CPU development. That means if your application relies on using AVX512 then you can't use Valgrind.

If both sanitizers and Valgrind work on your system, which should you choose? As ever, it depends.

	Valgrind	Sanitizer
Speed	Very slow, to the point of being sometimes unusable	Slow
Stack bounds checking	No	Yes
Instrumentation required	No	Yes
Availability and support	amd64 i386 aarch64	amd64 i386 aarch64 risc-v

When I said Valgrind doesn't need instrumentation, that was a white lie. If you are using custom allocators, then you need to write some annotation for either Valgrind or the sanitizers to work correctly. Similarly, if you use custom thread locking routines like a spin lock, you need to annotate them again in both cases. Thread sanitizer does have the advantage of having built-in annotation for standard library mechanisms that don't rely on pthreads such as std::atomic.

FreeBSD is lucky to have its toolchain based on LLVM. That means memory sanitizer is easily available. GCC doesn't have memory sanitizer, making it a lot more difficult to use on Linux. Don't underestimate how big a task "instrumentation required" is. For the best results that means you should instrument all your dependent libraries. If you are a KDE application developer, that means at least the following sets of libraries: KDE, Qt, libc++. There are dozens of other dependencies (libfontconfig, libjpeg, etc.). As we Valgrind developers like to say, "good luck with that!" If you are working for a big company and you have a dedicated devops team that can set it all up, then it's not so bad. I'd be interested in hearing from anyone who has experience in using poudriere for sanitizer builds. I've also read about people with large unit test suites complaining about the excessive build time and disk space requirements when building with sanitizers, particularly as you can't do a "one stop shop" sanitizer build (address and memory sanitizers are incompatible).

My conclusion here is that you should use whichever best suits your needs.

Future Work

Unfortunately, Valgrind is a tool that bitrots very quickly. New versions of FreeBSD keep coming out with new and changed syscalls. Extra items keep getting added to the auxiliary vector. _umtx_op gets more commands. libc++ keeps finding stranger ways of using pthreads. Compilers optimize things in ways that look like they are unsafe. That means that work on Valgrind is never finished.

You can't do a "one stop shop" sanitizer build.

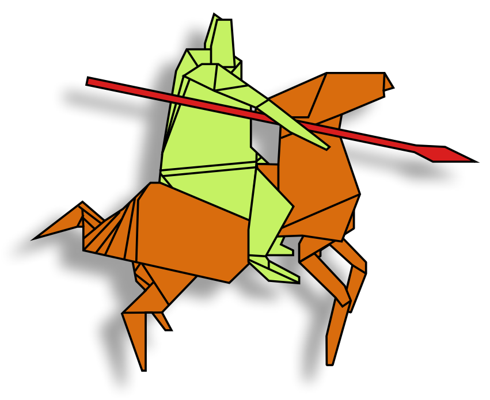
CPU architectures

Valgrind on FreeBSD runs on amd64, i386, and aarch64. I can't see myself adding MIPS or PPC support. RISC-V hasn't yet been added to the official Valgrind source — a port is on the way, but currently it is being held up by discussions over the implementation of vector instructions.

Bug list

The Valgrind Bugzilla has around 1000 open bugs in it. While many of these only affect Linux/macOS/Solaris, there are a good number that do affect FreeBSD.

- Helgrind produces false positives in thread local storage when there is a lot of thread creation/destruction. That is because there is a cache for the pthread stacks that include TLS. Valgrind doesn't see the recycled TLS as having different memory addresses. Linux works around this by deactivating the pthread stack cache via a GNU libc environment variable. I haven't found a way to do the same thing with FreeBSD libc.
- When the guest coredumps, it is Valgrind that generates the core file. Currently, the core file is pretty much with the same layout as a Linux core dump. That means lldb and gdb can't do much with the core file. I don't think that is a big issue as not many people use core files these days.
- The thread scheduler. Valgrind has a very rudimentary thread scheduler. Thread context switches occur at system call boundaries or every 100000 basic blocks. The default scheduler simply releases the global lock, and it's a question of luck as to which thread gets the lock. That could well be the previous thread if it is hot in the CPU cache. Linux has an optional fair scheduler based on futexes. Whilst that can't be ported directly to FreeBSD, it shouldn't be too difficult to post it using _umtx_op.
- On aarch64 there are occasional DRD false positives related to accesses in thread local storage



- The code that verifies ioctls is very limited. Almost all ioctls only get basic size checking done on their arguments. This needs to be extended, ideally also with testcases.

Conclusions

Working on Valgrind is quite a challenge. Debugging can be extremely difficult – I've often found myself doing things like debugging the guest in parallel with debugging Valgrind running the guest in parallel with using vgdb to debug the guest running in Valgrind. I've learned a lot about ELF, signals, and syscalls as well, of course, as about Valgrind itself. There's always much to learn — the nuances of aarch64 and amd64 opcodes and the multitude of tricks used in the dynamic recompilation.

PAUL FLOYD has been using FreeBSD intermittently since 2.1 and in earnest since 10.0. He's been a member of the Valgrind development team for four years. He has a PhD in Electronics and lives near Grenoble, on the edge of the French Alps working for Siemens EDA developing tools for analog electronic circuit simulation.