SEE
TEXT
ONLY

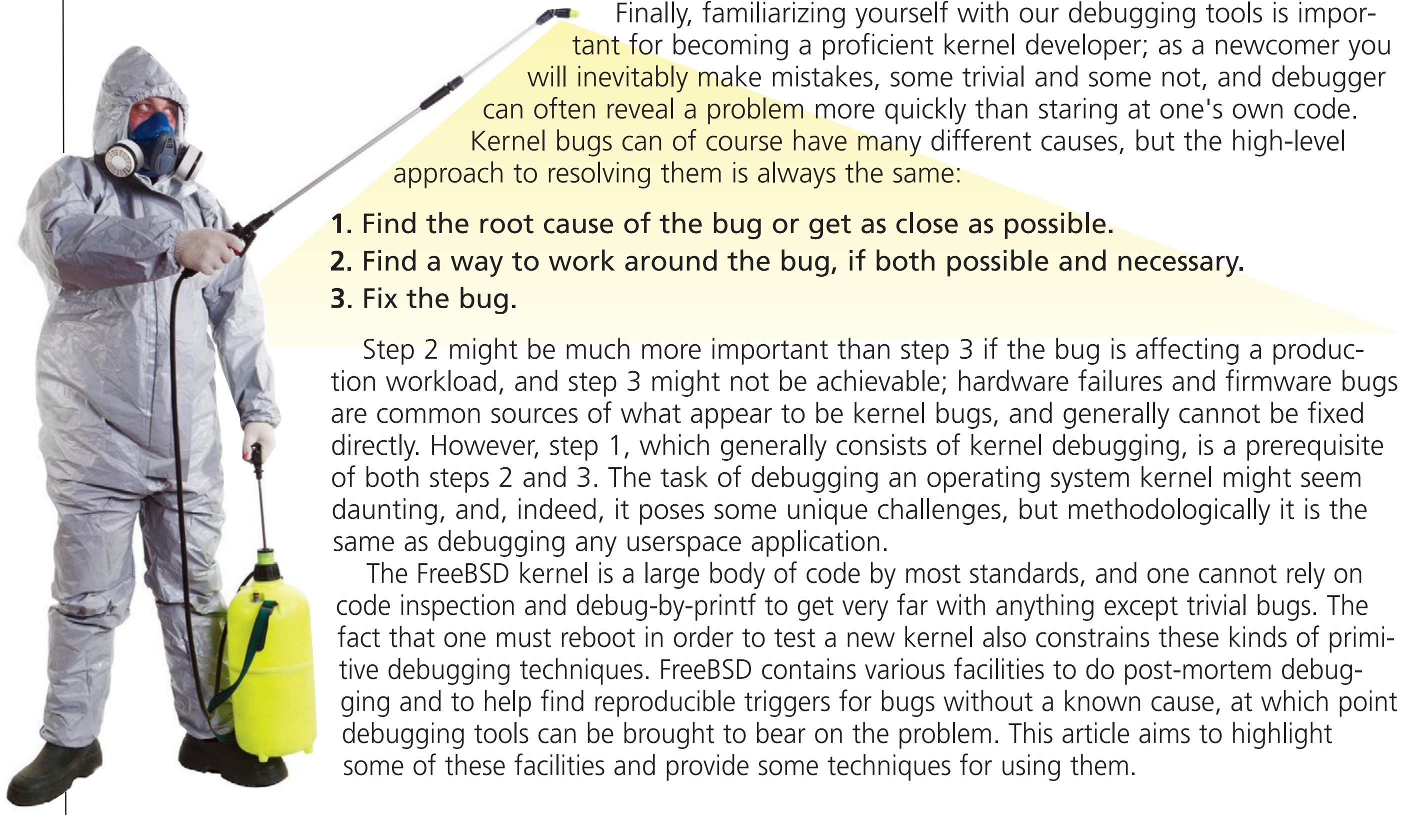
DeBUGGING the FreeBSD Kernel

By Mark **Johnston**

As I write this, the long-awaited FreeBSD 12.0 release is within days of shipping. Of course, FreeBSD developers have been meticulously polishing and testing 12.0's kernel for months and we're confident that every last bug has been squashed.

For the typical FreeBSD user, the subject of kernel debugging is just an intellectual curiosity, right? This is how it should be, at least; FreeBSD's stability is one of its main selling points and contributes a lot to its reputation. The stability of an operating system kernel is especially important since a fault in the kernel will typically bring down the entire system. In truth, however, no matter how many bugs were fixed in the lead-up to 12.0, the harsh reality is that some are still lingering and will emerge later to bite users, often with a stress-inducing kernel panic.

It is useful to have some familiarity with FreeBSD kernel debugging even if you are not a FreeBSD developer. The permissiveness of the BSD license has allowed many companies to build products on top of the FreeBSD source tree, often directly extending the kernel. These extensions will, of course, contain bugs or may expose latent bugs in FreeBSD itself. Moreover, developers working on such a code base generally cannot work directly with the FreeBSD community and thus would need to do some debugging themselves, perhaps before reporting the issue upstream. Another very common scenario is the case where developers do not have direct access to the system exhibiting a problem, so live debugging is not possible. Instead, any data required for debugging must be collected by an administrator or user. Knowing what to collect and how to collect it can greatly speed up the debugging process.



Finally, familiarizing yourself with our debugging tools is important for becoming a proficient kernel developer; as a newcomer you will inevitably make mistakes, some trivial and some not, and debugger can often reveal a problem more quickly than staring at one's own code.

Kernel bugs can of course have many different causes, but the high-level approach to resolving them is always the same:

1. Find the root cause of the bug or get as close as possible.
2. Find a way to work around the bug, if both possible and necessary.
3. Fix the bug.

Step 2 might be much more important than step 3 if the bug is affecting a production workload, and step 3 might not be achievable; hardware failures and firmware bugs are common sources of what appear to be kernel bugs, and generally cannot be fixed directly. However, step 1, which generally consists of kernel debugging, is a prerequisite of both steps 2 and 3. The task of debugging an operating system kernel might seem daunting, and, indeed, it poses some unique challenges, but methodologically it is the same as debugging any userspace application.

The FreeBSD kernel is a large body of code by most standards, and one cannot rely on code inspection and debug-by-printf to get very far with anything except trivial bugs. The fact that one must reboot in order to test a new kernel also constrains these kinds of primitive debugging techniques. FreeBSD contains various facilities to do post-mortem debugging and to help find reproducible triggers for bugs without a known cause, at which point debugging tools can be brought to bear on the problem. This article aims to highlight some of these facilities and provide some techniques for using them.

Triggering Kernel Bugs

Unexpected kernel panics are never pleasant. Almost as bad, though, is trying frantically to replicate someone else's kernel panic and failing. Usually this happens when the bug is timing-dependent, as in a race condition or a use-after-free. There is no silver bullet for these situations, but the FreeBSD kernel provides several facilities that may prove useful.

INVARIANTS Kernels

INVARIANTS is a kernel option that is configured in development builds of the FreeBSD kernel—if you are not running a FreeBSD release, it is quite likely that you are using GENERIC with INVARIANTS configured. In this case, congratulations! Your kernel has thousands of integrity checks enabled, constantly testing the assumptions of the programmers who wrote it. These checks impose runtime overhead, making INVARIANTS unsuitable for many production workloads, but they should be enabled whenever it is feasible. If you report a FreeBSD kernel panic to the mailing lists, it is quite likely that you will simply be asked to test with an INVARIANTS kernel if you are not already doing so. Other kernel options, such as DIAGNOSTIC and DEBUG_VFS_LOCKS, enable further rounds of more expensive integrity checking that can catch many bugs.

Memory Trashing

The FreeBSD kernel has an implementation of malloc(9) very similar to that used in regular C applications. Additionally, a memory subsystem called UMA ("Universal Memory Allocator") provides a slab allocation API for high-performance code paths. When INVARIANTS is configured in the kernel, memory freed using either of these APIs is "trashed," meaning that its contents is overwritten with a specific pattern. Any time trashed memory is reused, its contents are verified against that pattern. If verification fails, it means that the memory was written to after it was freed, and the kernel panics.

This is a very standard technique and is implemented in most full-featured memory allocators. On FreeBSD, the trash pattern is 0xdeadc0de so that, while the allocator itself will catch a write-after-free, anything that reads a trashed memory location (i.e., a read-after-free) is also likely to crash if it treats the value as a pointer. When debugging or triaging a kernel panic, keep an eye out for 0xdeadc0de —it very often signifies a use-after-free.

Memory trashing helps use-after-frees stand out by making it clear what the use-after-free did. Consider the following memory dump:

```
0xc00000037d3c0ba0:    0                   fffffe00017f2530
0xc00000037d3c0bd0:    fffff80002268e80    ffffffff80b35c58
0xc00000037d3c0be0:    deadc0dedeadc0de    deadc0dedeadc0de
0xc00000037d3c0bf0:    deadc0dedeadc0de    deadc0dedeadc0de
0xc00000037d3c0c00:    0                   deadc0dedeadc0de
0xc00000037d3c0c10:    deadc0dedeadc0de    deadc0dedeadc0de
0xc00000037d3c0c20:    deadc0dedeadc0de    deadc0dedeadc0de
0xc00000037d3c0c30:    deadc0dedeadc0de    deadc0dedeadc0de
```

Here we can see that the write-after-free wrote 8 bytes of zeros at a specific offset within a structure; that can be a helpful clue for identifying the responsible code.

A use-after-free is an especially nasty type of bug since the cause and effect of the bug may be far removed from each other. Memory trashing helps identify that a use-after-free occurred but can't catch the offending code in the act. This is where MemGuard comes in.

MemGuard

MemGuard is a memory subsystem feature which proactively detects use-after-frees for a particular memory type. A memory type is defined by either the type argument to malloc(9), or a UMA zone. For example, the line:

```
kq = malloc(sizeof *kq, M_KQUEUE, M_WAITOK | M_ZERO);
```

references M_KQUEUE, which is defined with:

```
MALLOC_DEFINE(M_KQUEUE, "kqueue", "memory for kqueue system");
```

The M_KQUEUE malloc(9) type is thus named "kqueue". Statistics for each malloc(9) type can be listed with: vmstat -m, and similar statistics for UMA zones can be listed with vmstat -z.

MemGuard is not present in FreeBSD kernels by default and must be compiled in by adding the

DEBUG_MEMGUARD option to the kernel configuration. Then, to configure MemGuard for a particular memory type, set vm.memguard.desc to the name of the malloc(9) type or UMA zone, either by adding an entry to /boot/loader.conf or by setting the sysctl. Once MemGuard is configured, it hooks all allocations and frees of the memory type. Each allocation is padded to a full page, and when a piece of memory is freed, the entire range of allocated memory is unmapped. This means that an attempt to access the memory after it has been freed will likely result in a page fault which crashes the kernel, catching the use-after-free as it happens.

The MemGuard facility is quite powerful but is correspondingly heavy-handed. Since each memory allocation must be padded to a full page, it will waste a large amount of memory if the memory type is used for small allocations. For instance, enabling MemGuard on the mbuf zone causes 4096 - 256 = 3840 bytes to be wasted per mbuf. The MemGuard allocation and free hooks will also severely impact the performance of the allocation: unmapping kernel memory is an expensive operation that must be serialized across all CPUs in the system. Thus, on a system with many CPUs and at a high allocation rate, MemGuard can easily become a major bottleneck. If the use-after-free is timing-dependent as is often the case, MemGuard's overhead can be such that the bug stops occurring. Nonetheless, it is quite effective in practice.

Fail Points

Many bugs, both in the kernel and userspace, are the result of incorrect or insufficient error handling. These bugs lay dormant until the corresponding error occurs, so the commonly executed "happy path" is unaffected. When the error does occur, it may be the result of a hardware or firmware malfunction, or some other seemingly unlikely event. Suppose you are in the position of having to debug the error-handling problem, and against all odds, you think that you have spotted the bug and want to verify your theory. How do you go about this? How can you write a regression test to ensure that your fix remains valid into the foreseeable future?

FreeBSD's fail point subsystem (documented in fail(9)) is a useful tool here. Fail points are a mechanism that allows programmers to inject errors and modify behavior at pre-defined locations in the kernel code. Fail points do not have any effect by default, but can be activated using sysctl(8): each fail point is controlled by a unique sysctl, usually under debug.fail_point. Fail points can be used to modify stack variables (usually to inject an error), change control flow (by following a goto or returning early from a function), or sleep or pause (useful for triggering race conditions).

The fail point interface consists of a set of macros, most of which are convenience wrappers for KFAIL_POINT_CODE. For example, we might try using a fail point to simulate random failures from malloc(9). The following fail point at the end of malloc() would do the trick:

```
void *
malloc(size_t size, struct malloc_type *mtp, int flags)
{
  ...
  if ((flags & M_WAITOK) == 0)
    /* M_WAITOK allocations are not allowed to fail. */
    KFAIL_POINT_CODE(DEBUG_FP, malloc, {
      free(va, mtp);
      va = 0;
    });
  return ((void *)va);
}
```

The same fail point could be defined a bit more succinctly:

```
void *
malloc(size_t size, struct malloc_type *mtp, int flags)
{
  ...
  KFAIL_POINT_CODE_COND(DEBUG_FP, malloc, (flags & M_WAITOK) == 0, 0, {
    free(va, mtp);
    va = 0;
  });
  return ((void *)va);
}
```

Either way, we must take care to free the newly allocated memory ourselves before simulating an allocation failure: otherwise that memory would be permanently leaked. For ad-hoc debugging, this might be acceptable, but in general, one must take care to ensure that a fail point does not introduce a new bug.

With this fail point defined, we get a sysctl named `debug.fail_point.malloc`. Fail points are activated by setting the corresponding sysctl to a string which describes the action to take when a thread executes the fail point. The default action is "off," meaning that the fail point has no effect. The "return" action causes the fail point's code to be executed; other actions are defined in the manual page. The kernel would not survive for long if all `malloc(M_NOWAIT)` allocations failed, so fail points can be configured to execute with a specified probability. The setting

```
# sysctl debug.fail_point.malloc='1%*return'
```

would cause the fail point's code to be executed roughly once for every 100 `malloc(M_NOWAIT)` calls.

Fail points are particularly useful for simulating disk errors; such errors are hopefully quite rare, but filesystems, among other things, must handle them robustly. When properly integrated, fail points can be an important part of the regression testing regimen for such code. For example, FreeBSD's RAID1 implementation `gmirror` contains fail points used to simulate errors from the underlying disks. Over time, a number of bugs in `gmirror`'s error handling have been fixed, and fail points are now used in several automated {gmirror} tests.

Kernel Stack Tracing

Occasionally a kernel bug will manifest with a thread entering an infinite loop in the kernel. No crash occurs, but a keen observer might report the issue as a task in `top(1)` consuming 100% of a CPU for no obvious reason. If the system is not being used in production, it might be reasonable to drop in to a kernel debugger to interrogate the task in question, but this is not always possible.

FreeBSD has the ability to print kernel stacks without pausing the system using the `procstat(1)` utility's `-k` option. `procstat -kk <pid>` prints kernel stacks, and `procstat -kka` prints kernel stacks for every thread in the system. While this does not directly help in finding a trigger for the bug, it provides an accessible starting point both for tasks that appear "stuck" and are unresponsive, and for tasks that appear to be spinning forever. When `procstat(1)` is asked to capture the stack for a given thread and the thread is currently running on a CPU, `procstat(1)` will raise an interrupt on that CPU and capture the thread's stack frames from the interrupt handler. While this may not provide sufficient information to root-cause a bug on the spot, it is often a useful clue, and the fact that it can be used without bringing down the entire system makes it a handy place to begin an investigation.

Kernel Dumps and `kgdb`

`kgdb` is the workhorse of many kernel debugging sessions, especially when live debugging is impossible. As the name suggests, `kgdb` is an extension of `gdb`, the GNU debugger, for FreeBSD kernels. It is actively maintained and provided as a component of the `gdb` package. Similar to `gdb`, it can be used to inspect the running kernel or to inspect the post-mortem output of a kernel panic, usually referred to as a "kernel dump" or "crash dump." If you hit a kernel panic, you may be requested to provide a kernel dump to a developer for analysis; the logistics of doing so can be complicated, so it is worth understanding the various options available for saving kernel dumps.

Kernel dumps are effectively a complete copy of the kernel's state at the time of a panic and often contain enough information to completely piece together the cause of the crash. When a kernel panic occurs, the kernel will log a message and, if so configured, jump to a special routine which attempts to save a dump and then automatically reboots the system. Upon a subsequent boot-up, `savecore(8)` will check for and recover a previously saved dump, storing it in `/var/crash`.

Kernel dumps come in two flavors: minidumps and full dumps. Full dumps are largely an anachronism and are used on architectures that do not yet have minidump support. The difference is that full dumps contain the entire contents of RAM, while minidumps contain only memory which was allocated to the kernel at the time of the crash: unallocated pages, user memory and cached files are omitted. Minidumps are correspondingly smaller but more complicated to implement and support. `kgdb` can handle both types, so the distinction is not usually important or even noticeable. It's important to note that kernel dumps will generally contain sensitive user data and should be shared cautiously.

The main consideration when configuring kernel dumps is the location at which the dump will be saved by the kernel. Because the code which actually saves the dump is running after a kernel panic, it cannot rely on the kernel to be operational and thus must be as simple as possible; we definitely don't want it to try writing to a filesystem! The simplest approach is to save the dump on the system's swap partition: the contents of swap are, by definition, useless after a reboot, so there is no harm in it being overwritten. This behavior can be configured by adding the line

```
dumpdev="AUTO"
```

to `/etc/rc.conf` —if a swap device is configured in `/etc/fstab`, it will then be used for kernel dumps. Otherwise, to save dumps to a local disk, the name of the disk device must be specified instead of "AUTO."

When configuring a dump device, it's a good idea to test it by triggering a panic and verifying that a **`vmcore`** file appears in `/var/crash` after the reboot:

```
# sysctl debug.kdb.panic=1
```

The main difficulty with a local dump device is storage space. The size of a kernel dump is roughly proportional to the amount of RAM in the system. While minidumps greatly reduce the need for space, they can still easily be in the tens of gigabytes on a system with 128GB of RAM, larger than a reasonably sized swap partition. A system administrator would be forced to either waste disk space or potentially lose the ability to save kernel dumps. Fortunately, FreeBSD 12.0 comes with several new features to help alleviate this problem. First, the code which saves a kernel dump can now compress the data before writing it, greatly reducing space requirements. Currently, the zlib and newfangled Zstandard algorithms are supported for this purpose. Zstandard is both faster and better at compressing kernel dumps, and we frequently see compression ratios of 8:1 or better. Either algorithm can be configured by setting **`dumpon_flags`** in `/etc/rc.d`; see the **`dumpon(8)`** manual page for details. Zstandard compression can thus be easily configured:

```
dumpdev="AUTO"
dumpon_flags="-Z"
```

Compression staves off the storage requirement problem for now, but the fundamental problem remains. Moreover, FreeBSD is used on many systems that do not have a suitable dump device to begin with. For example, many embedded devices—such as routers—boot from read-only media and do not have any dynamic storage. Virtual machines are often provisioned without swap space, and diskless servers boot from an NFS mount and do not have any local storage at all. To address these scenarios, FreeBSD 12.0 will come with the ability to transmit kernel dumps over a network interface using an IPv4 UDP-based protocol called Netdump.

Netdump

Netdump provides flexibility to system administrators and developers by allowing them to use a remote server to receive kernel dumps from clients (panicking kernels). Netdump is integrated with the existing kernel dump facilities and is transparent with respect to the kernel dump features described earlier; it is possible to use kernel dump compression or encryption with Netdump, just as with dumps to a local disk. From an adminstrator's perspective, Netdump simply requires additional configuration.

Netdump clients use **`dumpon(8)`** to configure client parameters. Instead of specifying a block device, specify the name of the network interface to be used. This should preferably be a low-traffic management interface rather than one of the system's data ports: Netdump will be less reliable if the interface that it uses was busy at the time of the panic. Netdump does not automatically support every network adapter, but many commonly encountered adapters are, indeed, supported, and a list of supported drivers can be found in the **`netdump(4)`** manual page. Of course, the network interface name is not sufficient: the Netdump client code needs to know how to contact the server! Three network address parameters must be specified:

- The client IP address to use (**`-c`**). This parameter is mandatory.
- The server's hostname or IP address (**`-s`**). This parameter is mandatory.
- The IP address of the first-hop router between the client and server (**`-g`**). That is, the address of the router that forwards packets from the client to the server. This is typically just the system's default router as listed in the output of **`netstat -rn`**. This parameter is optional: if it is not specified, **`dumpon(8)`** uses the system's default route if one is configured, and, otherwise, assumes that the client and server are on the same link.

These parameters can all be specified in `/etc/rc.conf`:

```
dumpdev="em0"
dumpon_flags="-s 192.168.1.34 -c 192.168.2.154 -g 192.168.2.1"
```

At the time of writing, only IPv4 parameters are supported. We hope to support IPv6 in future releases of FreeBSD.

Because the Netdump client code runs only after the system has panicked, it cannot use any of the kernel code which normally does the work of building IPv4 packets and sending them out onto the wire. In particular, it cannot make use of the kernel's ARP cache, so as a first step, it must resolve the Ethernet address of the next hop. In fact, the Netdump client always attempts to resolve the server directly before trying to resolve the gateway. This way, if both the client and server happen to be on the same link, they can communicate directly. Otherwise, the client relies on the gateway to forward all Netdump packets to the server.

The Netdump server program is called `netdumpd` and is available as a package for FreeBSD 11 and 12. It can be configured via `/etc/rc.conf`; its main parameter specifies the root directory under which it will save all kernel dumps from clients. `netdumpd` listens on UDP port 20023 for `HERALD` messages from a client (described below). Using an ephemeral port, it replies to all client messages with an `ACK` message. When a client receives the `ACK` for its initial `HERALD`, it sends all subsequent messages to the UDP port from which the `ACK` originated. When a Netdump session completes successfully, the server saves two files under its root directory: a `vmcore` file, identical to those created by `savecore(8)`, and a text file (`info`) containing metadata describing the kernel dump. It may optionally be configured to execute a user-defined script upon completion, successful or otherwise. This can be used, for example, to send an email to the owner of a system when that system panics and transmits a kernel dump.

The Netdump protocol is extremely simple by design and closely resembles the TFTP protocol. All messages consist of a fixed-size header. Client messages begin with:

```
struct netdump_msg_hdr {
        uint32_t  mh_type;
        uint32_t  mh_seqno;
        uint64_t  mh_offset;
        uint32_t  mh_len;
        uint32_t  mh__pad;
};
```

while server messages are always 4 bytes in length:

```
struct netdump_ack {
        uint32_t  na_seqno;
};
```

There are five Netdump client message types:

- A client sends a `HERALD` message to signal the beginning of a kernel dump. It contains an optional payload: a relative path to the server's root directory under which to save the dump. The path must exist on the server.
- A client sends a `FINISHED` message when it has completed the dump. Upon receipt of this message, the server flushes its output files and invokes the user-specified completion hook.
- `VMCORE` messages contain kernel dump data. The `mh_offset` field specifies the relative offset of the data within the output file.
- The `KDH` ("kernel dump header") message contains kernel dump metadata such as the client's version and panic message.
- The `EKCD_KEY` contains the encrypted kernel dump key if kernel dump encryption was configured on the client. This message is not used otherwise.

Server messages are very simple: they acknowledge receipt of a client message by including the corresponding sequence number.

kgdb

After going through all the work to trigger a panic and obtain a kernel dump, it's time to do some actual debugging. With `kgdb`, opening the kernel dump is just like opening a userland core dump using `gdb`: you simply point `kgdb` at the kernel binary and `vmcore` files:

```
# kgdb /boot/kernel/kernel /var/crash/vmcore.0
...
(kgdb) bt
#0  __curthread ()
#1  doadump (textdump=16777216)
#2  0xffffffff804ccdbc in db_fncall_generic (...)
#3  db_fncall (...)
#4  0xffffffff804cc8f9 in db_command (...)
#5  0xffffffff804cc674 in db_command_loop ()
#6  0xffffffff804cf91f in db_trap (...)
#7  0xffffffff8088b553 in kdb_trap (type=9, code=0, tf=<optimized out>)
#8  0xffffffff80badce1 in trap_fatal (frame=0xfffffe00188da6b0, eva=0)
#9  0xffffffff80bad1dd in trap (frame=0xfffffe00188da6b0)
#10 <signal handler called>
#11 0xffffffff808594cf in callout_process (now=1960148907297521)
#12 0xffffffff80c329b8 in handleevents (now=1960148907297521, fake=0)
#13 0xffffffff80c33231 in timercb (et=<optimized out>, arg=<optimized out>)
#14 0xffffffff80bbb289 in hpet_intr_single (arg=0xfffffe0000b590b0)
#15 0xffffffff80bbb32e in hpet_intr (arg=0xfffffe0000b59000)
#16 0xffffffff8080953d in intr_event_handle
   (ie=0xfffff80002169100, frame=0xfffffe00188da9a0)
#17 0xffffffff80c6b748 in intr_execute_handlers
   (isrc=0xfffff80002034e48, frame=0xfffffe00188da9a0)
#18 0xffffffff80c71d54 in lapic_handle_intr (vector=<optimized out>,
    frame=0x6f6bec1fb41fa)
#19 <signal handler called>
#20 acpi_cpu_c1 ()
#21 0xffffffff804f28a7 in acpi_cpu_idle (sbt=<optimized out>)
#22 0xffffffff80c684af in cpu_idle_acpi (sbt=59992010)
#23 0xffffffff80c68567 in cpu_idle (busy=0)
#24 0xffffffff80873c65 in sched_idletd (dummy=<optimized out>)
#25 0xffffffff808069a3 in fork_exit (...)
```

As with `gdb`, debug info is needed in order for `kgdb` to make sense of a kernel dump's contents. In recent versions of FreeBSD, these symbols are stored externally, in `/usr/lib/debug/boot/kernel`, assuming that the kernel being debugged is in `/boot/kernel`. When sending a kernel dump to a developer, it is important to include the contents of both `/boot/kernel` and `/usr/lib/debug/boot/kernel`.

`kgdb` is smart enough to unwind through trap frames, printed above as `<signal handler called>`. These occur when the CPU interrupted the running thread, saving its state on the stack. In the example, we have two such frames: first, a timer interrupt interrupted an idle CPU, and, while processing pending callouts, that CPU took a protection fault caused by a corrupted pointer:

```
(kgdb) frame 11
#11 0xffffffff808594cf in callout_process (now=1960148907297521)
510                                         LIST_REMOVE(tmp, c_links.le);
(kgdb) p tmp
$1 = (struct callout *)   0x11777be9162acbc1
(kgdb) x tmp
0x11777be9162acbc1:       Cannot access memory at address 0x11777be9162acbc1
```

Most of the usual `gdb` commands work in `kgdb`. For instance, `info threads` will enumerate all of the system's threads, including kernel-only threads, and one can switch between threads. As an extension, kgdb can identify threads based on FreeBSD thread ID. This is handy when a data structure points to a particular thread and you want to switch to that thread; for example, when the current thread is blocked on a lock, we may wish to look at the thread holding that lock:

```
(kgdb) p kq
$5 = (struct kqueue *) 0xfffff801cb5a4a00
(kgdb) p/x kq->kq_lock
$6 = {
  lock_object = {
    lo_name = 0xffffffff814e8f66,
    lo_flags = 0x1430000,
    lo_data = 0x0,
    lo_witness = 0x0
  },
  mtx_lock = 0xfffff801e3b34000
}
(kgdb) p ((struct thread *)kq->kq_lock.mtx_lock)->td_tid
$7 = 101322
(kgdb) tid 101322 # Switch to the thread with TID 101322.
(kgdb)
```

kgdb offers scripting facilities: GDB script, which is the native GDB command language with some standard programming constructs like if statements and while loops, and a Python API. The GDB scripting language is rather primitive and limited, so we will focus on the more powerful Python integration and provide several examples. The API documentation is quite thorough, so the examples merely portray some possible uses for the API without explaining each detail of how they work.

The kernel's struct thread contains many fields specific to one subsystem; kernel developers effectively allocate thread-local storage by adding fields to this rather large structure. When debugging, a common need is to find the pointer to the currently selected thread. This can often be done in an ad-hoc way, by searching various frames of the stack for a copy of the pointer, but this is tedious and not scriptable. Instead, let's use Python to add a "convenience function" to behave like the magic curthread variable in the kernel:

```
import gdb

def _queue_foreach(head, field, headf, nextf):
    elm = head[headf]
    while elm != 0:
        yield elm
        elm = elm[field][nextf]
def list_foreach(head, field):
    return _queue_foreach(head, field, "lh_first", "le_next")
def tailq_foreach(head, field):
    return _queue_foreach(head, field, "tqh_first", "tqe_next")

def tdfind(tid, pid=-1):
    td = tdfind.cached_threads.get(int(tid))
    if td is not None:
         return td

    allproc = gdb.lookup_global_symbol("allproc").value()
    for p in list_foreach(allproc, "p_list"):
        if pid != -1 and pid != p['p_pid']:
            continue
        for td in tailq_foreach(p['p_threads'], "td_plist"):
            ntid = td['td_tid']
            tdfind.cached_threads[int(ntid)] = td
            if ntid == tid:
```

Continues next page

Continued

```
                return td
tdfind.cached_threads = dict()

class curthread(gdb.Function):
    def __init__(self):
        super(curthread, self).__init__("curthread")
    def invoke(self):
        return tdfind(gdb.selected_thread().ptid[2])
curthread() # Register the function with kgdb.
```

The API provides a way to get the selected thread's thread ID, but we must traverse the kernel's data structures ourselves to get the corresponding `struct thread` pointer. This is done by the `tdfind()` routine, which emulates a kernel function of the same name to search the kernel's global list of processes (`allproc`) and find the thread with the specified thread ID. The global list can be quite large, so we cache all threads encountered during the search to speed up future lookups.

The `curthread` class provides the requisite glue: when `$curthread()` is invoked it simply returns the result of calling `tdfind()` on the selected thread's thread ID:

```
(kgdb) p $curthread()
$1 = (struct thread *) 0xfffff801e3b34000
```

Now we need not grovel through the stack to find the current thread pointer, and the lookup code can be reused for other purposes.

With the addition of `VIMAGE` to the standard kernel configuration, many global data structures in the network are virtualized: every reference to such a global structure is actually relative to the current `vnet`, so there is some implicit context to each reference. The kernel uses a set of macros in `vnet.h` to handle this, but `kgdb` doesn't know about them; the upshot of all this is that the following doesn't work:

```
502          if (pcbinfo == &V_tcbinfo) {
503                  INP_INFO_RLOCK_ASSERT(pcbinfo);
(kgdb) p V_tcbinfo
No symbol "V_tcbinfo" in current context.
```

We can get close, though, by providing a convenience function to resolve symbols in the context of the current `vnet`, or in a user-specified `vnet` if that happens to be useful:

```
class vimage(gdb.Function):
    def __init__(self):
        super(vimage, self).__init__("V")

    def invoke(self, sym, vnet=None):
        sym = sym.string()
        if sym.startswith("V_"):
            sym = sym[len("V_"):]
    if gdb.lookup_symbol("sysctl___kern_features_vimage")[0] is None:
        return gdb.lookup_global_symbol(sym).value()

    if vnet is None:
        vnet = tdfind(gdb.selected_thread().ptid[2])['td_vnet']
        if not vnet:
            # If curthread->td_vnet == NULL, vnet0 is the current vnet.
            vnet = gdb.lookup_global_symbol("vnet0").value()
    base = vnet['vnet_data_base']
    entry = gdb.lookup_global_symbol("vnet_entry_" + sym).value()
    entry_addr = entry.address.cast(gdb.lookup_type("uintptr_t"))
    ptr = gdb.Value(base + entry_addr).cast(entry.type.pointer())
    return ptr.dereference()
vimage() # Register the function with kgdb.
```

This snippet effectively reimplements what the real `VIMAGE` implementation does: get a pointer to the current `vnet` from the current thread and do a bit of math to find that `vnet`'s copy of the desired structure. It also tries to provide sane behavior if the kernel does not support `VIMAGE` by simply returning the global structure. Additionally, the desired `vnet` may be explicitly specified if needed. While we cannot simply print `V_tcbinfo`, we can now get pretty close:

```
(kgdb) p $V("tcbinfo")
$1 = {
   ipi_lock = {
     lock_object = {
       lo_name = 0xffffffff8125b55e "tcp",
       ...
```

As a final example, we will implement the `acttrace` command from DDB, which prints the backtraces from all threads that were actively executing on a CPU at the time of the panic. This is often a very important piece of the picture, as many bugs are the result of errant interactions between multiple threads. With luck, the threads that caused the panic will have been on-CPU at the time of the panic.

This example adds a new top-level command, mimicking DDB's interface. It uses the per-CPU (`pcpu`) structures which store information about the execution state of each CPU in the system.

```
import math

def cpu_foreach():
     all_cpus = gdb.lookup_global_symbol("all_cpus").value()
     bitsz = gdb.lookup_type("long").sizeof * 8
     maxid = gdb.lookup_global_symbol("mp_maxid").value()

     cpu = 0
     while cpu <= maxid:
         upper = cpu >> int(math.log(bitsz, 2))
         lower = 1 << (cpu & (bitsz - 1))
         if (all_cpus['__bits'][upper] & lower) != 0:
             yield cpu
         cpu = cpu + 1

     class acttrace(gdb.Command):
         def __init__(self):
               super(acttrace, self).__init__("acttrace", gdb.COMMAND_USER)

         def _inferior_thread_by_tid(self, tid):
              threads = gdb.inferiors()[0].threads()
              for td in threads:
                  if td.ptid[2] == tid:
                      return td

         def invoke(self, arg, from_tty):
             # Save the currently selected thread.
             curthread = gdb.selected_thread()

             # Note: not all platforms have a __pcpu array.
             pcpu = gdb.lookup_global_symbol("__pcpu").value()
             for cpu in cpu_foreach():
                 td = pcpu[cpu]['pc_curthread']
                 p = td['td_proc']
                 self._inferior_thread_by_tid(td['td_tid']).switch()
```

Continues next page

Continued

```
                print("Tracing command {} pid {} tid {} (CPU {})".format(
                    p['p_comm'].string(), p['p_pid'], td['td_tid'], cpu))
            gdb.execute("bt")
            print

        # Switch back to the starting thread.
        curthread.switch()
acttrace() # Register the command with kgdb.
```

The idea is to iterate over all online CPUs in the system, looking up the per-CPU structure and using that to find the thread that is on the CPU. We use the thread's ID to look up `kgdb`'s representation of that thread, switch to it, and print the backtrace. Once this is all done, we switch back to the starting thread to avoid confusing the user.

```
(kgdb) acttrace
Tracing command idle pid 10 tid 100002 (CPU 0)
#0  cpustop_handler ()
#1  0xffffffff80928074 in ipi_nmi_handler ()
#2  0xffffffff808c0e19 in trap (frame=0xffffffff8104b160 <nmi0_stack+3888>)
#3  <signal handler called>
#4  acpi_cpu_idle_mwait (mwait_hint=0)
...
```

Conclusion

Hopefully this article has helped illuminate several stages of the typical FreeBSD kernel developer's debugging workflow. Of course, experience and intuition can get you pretty far, but good tooling is tremendously helpful in tracking down the more slippery bugs that arise, especially when they lie outside one's domain of expertise. Debuggers also help attune one to various patterns and details that underlie the core functionality of the kernel, so with experience it becomes easier to spot things that "just don't look right": clues which might lead to a root cause. *Happy debugging!* ●

Mark Johnston is a freelance FreeBSD developer living in Toronto, Canada. In his spare time he enjoys attempting to make nice sounds on a cello, running, travelling, and playing dodgeball on a team with his friends.