Introduction to Debugging the FreeBSD Kernel

John H. Baldwin
Yahoo!, Inc.
Atlanta, GA 30327
jhb@FreeBSD.org, http://people.FreeBSD.org/~jhb

Abstract

Just like every other piece of software, the FreeBSD kernel has bugs. Debugging a kernel is a bit different from debugging a userland program as there is nothing underneath the kernel to provide debugging facilities such as ptrace() or procfs. This paper will give a brief overview of some of the tools available for investigating bugs in the FreeBSD kernel. It will cover the in-kernel debugger DDB and the external debugger kgdb which is used to perform post-mortem analysis on kernel crash dumps.

1 Introduction

When a userland application encounters a bug the operating system provides services for investigating the bug. For example, a kernel may save a copy of the a process' memory image on disk as a core dump. An operating system may also provide APIs for one process to analyze the state of another process. Using these services, debugging tools such as gdb [1] can be written.

Operating system kernels have bugs just like userland applications. A key difference, however, is that operating system kernels are not always able to rely on a separate piece of software to provide debugging services. As a result, kernels generally must provide their own specialized support for debugging services. The FreeBSD kernel provides several services such as crash messages, crash dumps, the `/dev/kmem` device, a remote GDB interface, and a self-contained in-kernel debugger called DDB [2]. These services can then be used either directly by the user or indirectly via other tools such as kgdb [3].

The *Kernel Debugging* chapter of the *FreeBSD Developer's Handbook* [4] covers several details already such as entering DDB, configuring a system to save kernel crash dumps, and invoking kgdb on a crash dump. This paper will not cover these topics. Instead, it will demonstrate some ways to use FreeBSD's kernel debugging tools to investigate bugs.

2 Kernel Crash Messages

The first debugging service the FreeBSD kernel provides is the messages the kernel prints on the console when the kernel crashes. When the kernel encounters an invalid condition (such as an assertion failure or a memory protection violation) it halts execution of the current thread and enters a "panic" state also known as a "crash". The kernel will always print a message on the console summarizing the reason for the crash. For many crashes this summary message is all the kernel outputs. Figure 1 contains the crash message for a simple assertion violation. In this case the message indicates that a thread attempted to sleep while sleeping is prohibited.

Some crashes output more detailed messages to the console. Crashes caused by a CPU exception generally output several details. For example, if a thread in the kernel dereferences a pointer into unmapped memory (such as a `NULL` pointer) then the crash messages will include the invalid address. Figure 2 contains the crash messages for an amd64 page fault. This particular page

```
panic: Trying sleep, but thread marked as sleeping prohibited
```

Figure 1: Example Assertion Failure Crash

fault was the result of a `NULL` pointer dereference in the `net.inet.tcp.pcblist` sysctl handler. It was caused by a race condition where a `struct tcpcb` was freed in one thread while another thread was in the sysctl handler.

The first key piece of data is the *fault virtual address*. It is the invalid memory address that caused the fault. In this case the fault address is indicative of a `NULL` pointer derefence since its value is very small. The *instruction pointer* indicates the program counter value where the fault occurred. This can be used either with gdb(1) or addr2line(1) to determine the corresponding source line. The *current process* lists the command name and PID of the process that was executing when the fault occurred.

3 Live Debugging with DDB

Another debugging tool provided by the FreeBSD kernel is the in-kernel debugger DDB. DDB is an interactive debugger that allows the user to execute specific commands to inspect various details of the running kernel. It is able to resolve global symbols to addresses and control execution via breakpoints and single stepping. It is also extensible since new commands may be added at compile time. Details about several of the commonly used DDB commands may be found in the ddb(4) manpage [2].

3.1 Inspecting Processes and Threads

One of the best ways to get an overview of a system's state from DDB is to examine the current state of individual processes and threads. DDB provides several commands to do this. First, the `ps` command will display a list of all the processes and threads in the system. The listing includes a summary of the state of each thread including any lock the thread is blocked on or a wait channel on which the thread is sleeping. More specific details about individual processes may be obtained via the `show proc` command. This command accepts a single argument that is either a direct pointer to a `struct proc` or a process ID (PID). Similarly, the `show thread` command provides details about an individual thread and accepts either a direct pointer to a `struct thread` or a thread ID (TID). Figure 3 shows a truncated list of processes and threads in various states. Figure 4 shows more detailed information about the first process in the list and one of its threads.

A very important part of a thread's state is the stack trace. A stack trace provides a bit of history of where the thread has been in the past. It can also help explain how a thread arrived at its current state. DDB provides a `trace` command to obtain the stack trace of single thread. With no aguments it will provide a trace of the current thread. If an argument is specified then it may be either a TID or a PID. If the argument is a PID, then the first thread from the indicated process will be used. Figure 5 shows the stack trace for the thread blocked on the `def` lock. The trace indicates that the thread attempted to acquire the lock in the aptly named `mtx_deadlock` function.

3.2 Investigating Deadlocks

Debugging deadlocks requires determining which resources threads are waiting on and then analyzing those dependencies to find a cycle. One source of deadlocks is misuse of locking primitives such as mutexes. DDB provides several commands for analyzing locking primitives and the dependency relationships between threads and locks.

First, DDB provides commands to directly inspect the state of locks and the queues of

```
Fatal trap 12: page fault while in kernel mode
cpuid = 0; apic id = 00
fault virtual address   = 0x4
fault code              = supervisor read, page not present
instruction pointer     = 0x8:0xffffffff80359af8
stack pointer           = 0x10:0xffffffffa3cbb550
frame pointer           = 0x10:0xffffffffa3cbb570
code segment            = base 0x0, limit 0xfffff, type 0x1b
                        = DPL 0, pres 1, long 1, def32 0, gran 1
processor eflags        = interrupt enabled, resume, IOPL = 0
current process         = 31466 (netstat)
trap number             = 12
panic: page fault
```

Figure 2: Example amd64 Page Fault

```
db> ps
  pid  ppid  pgrp   uid   state   wmesg         wchan        cmd
  954     0     0     0  LL      (threaded)                  crash2
100144                   L      *abc      0xffffff0001288dc0 [crash2: 3]
100143                   L      *jkl      0xffffff0001288c80 [crash2: 2]
100142                   L      *ghi      0xffffff0001288be0 [crash2: 1]
100055                   L      *def      0xffffff0001288d20 [crash2: 0]
  812     0     0     0  SL      -        0xffffffff80673a20 [nfsiod 0]
  771   769   771 26840  Ss+     ttyin    0xffffff00011b9810 tcsh
  769   767   767 26840  S       select   0xffffff00018ca0d0 sshd
  767   705   767     0  Ss      sbwait   0xffffff00016ed94c sshd
...
   10     0     0     0  RL      (threaded)                  idle
100005                   Run     CPU 0                       [idle: cpu0]
100004                   Run     CPU 1                       [idle: cpu1]
100003                   Run     CPU 2                       [idle: cpu2]
100002                   Run     CPU 3                       [idle: cpu3]
```

Figure 3: Example DDB ps Output

```
db> show proc 954
Process 954 (crash2) at 0xffffff0001354000:
 state: NORMAL
 uid: 0  gids: 0
 parent: pid 0 at 0xffffffff806538e0
 ABI: null
 threads: 4
100144                   L      *abc      0xffffff0001288dc0 [crash2: 3]
100143                   L      *jkl      0xffffff0001288c80 [crash2: 2]
100142                   L      *ghi      0xffffff0001288be0 [crash2: 1]
100055                   L      *def      0xffffff0001288d20 [crash2: 0]
db> show thread 100055
Thread 100055 at 0xffffff00013869c0:
 proc (pid 954): 0xffffff0001354000
 name: crash2: 0
 stack: 0xffffffffae213000-0xffffffffae216fff
 flags: 0x4  pflags: 0x200000
 state: INHIBITED: {LOCK}
 lock: def  turnstile: 0xffffff0001288d20
 priority: 224
```

Figure 4: Inspecting a Process and Thread in DDB

```
db> tr 100055
Tracing pid 954 tid 100055 td 0xffffff00013869c0
sched_switch() at sched_switch+0x15d
mi_switch() at mi_switch+0x215
turnstile_wait() at turnstile_wait+0x24c
_mtx_lock_sleep() at _mtx_lock_sleep+0xe0
_mtx_lock_flags() at _mtx_lock_flags+0x7a
mtx_deadlock() at mtx_deadlock+0xb4
crash_thread() at crash_thread+0x138
fork_exit() at fork_exit+0x12a
fork_trampoline() at fork_trampoline+0xe
--- trap 0, rip = 0, rsp = 0xffffffffae23ed30, rbp = 0 ---
```

Figure 5: Example DDB Stack Trace

threads waiting for locks. The `show lock` command takes the address of a lock (either a mutex [5], read-mostly lock [6], reader/writer lock [7], shared/exclusive (sx) lock [8], or lockmgr lock [9]) as its argument and displays details about the lock including the current owner, if any. The `show turnstile` command takes the address of a mutex, read-mostly lock or reader/writer wlock as its argument. If there is a turnstile associated with the lock, then it will display the lists of threads waiting on the specified lock. In Figure 3, four threads from process 954 are stuck in a deadlock cycle. In Figure 6 the relationships between the threads from that process and the `def` lock are inspected.

In this case, thread 100142 owns the `def` lock and thread 100055 is waiting for it. Note that the turnstile information actually includes the lock owners as well as the waiters for a given lock. Also, from Figure 3 one can see that the thread information includes the turnstile that a thread is currently blocked on. From this, it is apparent that one can build a dependency graph among a group of threads. For a given thread that is blocked on a turnstile, it is waiting for the owner of the lock associated with the turnstile.

DDB provides another command, `show lockchain` that displays this dependency chain. It walks the thread dependencies via turnstiles until it finds a thread that is not blocked on a turnstile. If it encounters a deadlock it will stay stuck in the cycle until the user uses 'q' at DDB's `--More--` prompt. The `show lockchain` argument takes an optional argument specifying the starting thread as either a pointer to a `struct thread` or a TID. Figure 7 shows the dependency graph for thread 100055 which is clearly stuck in a deadlock with the other threads from the same process.

A limitation of `show lockchain` is that it only handles dependencies for locking primitives that use turnstiles such as mutexes. Other locking primitives such as sx locks use sleepqueues to hold threads waiting for locks. DDB includes a `show sleepchain` command which displays a dependency graph for threads blocked on sx locks and lockmgr locks. Figure 8 shows the dependency graph for four threads locked in a cycle of lockmgr and sx locks.

3.3 Adding New DDB Commands

DDB commands are implemented by functions in the kernel. Thus, new commands can be added simply by writing new functions. Currently new commands cannot be added at runtime via kernel modules.

3.3.1 Declaring a Command

Each DDB command is bound to a function. The `<ddb/ddb.h>` header provides helper macros to declare a command function and add it to a command table. The `DB_COMMAND` macro creates a top-level command including the function prototype. See Figure 9 for an example of a simple "foo" command. Note that there is no explicit function prototype and that the function body immediately follows the macro. To add a "show" command, use `DB_SHOW_COMMAND` instead of `DB_COMMAND`.

The command function takes four arguments which provide the command's parameters. The `addr` argument specifies the address for the command to operate on. It may either be the user-supplied address or the `dot` address as described in ddb(4) [2]. The `have_addr` argument is a boolean that is true if the user supplied an explicit address. The `count` argument indicates the count of operations to be performed. If the user did not specify one, then `count` is set to -1. Finally, the `modif` argument is a string that contains the command modifiers without the leading slash. If no modifiers were specified, then `modif` will be an empty string.

3.3.2 I/O for DDB Commands

DDB command functions are executed in an alternative environment from the rest of the kernel. One of the primary differences is that DDB uses its own I/O subsystem. DDB commands do not accept direct input from

```
db> show lock def
 class: sleep mutex
 name: def
 flags: {DEF}
 state: {OWNED, CONTESTED}
 owner: 0xffffff000155c680 (tid 100142, pid 954, "crash2: 1")
db> show turnstile def
Lock: 0xffffffffae3c6fc0 - (sleep mutex) def
Lock Owner: 0xffffff000155c680 (tid 100142, pid 954, "crash2: 1")
Shared Waiters:
        empty
Exclusive Waiters:
        0xffffff00013869c0 (tid 100055, pid 954, "crash2: 0")
Pending Threads:
        empty
```

Figure 6: Examining Relationships Between Threads and the `def` Lock

```
db> show lockchain 100055
thread 100055 (pid 954, crash2: 0) blocked on lock 0xffffffffae3c6fc0 (sleep mutex) "def"
thread 100142 (pid 954, crash2: 1) blocked on lock 0xffffffffae3c7000 (sleep mutex) "ghi"
thread 100143 (pid 954, crash2: 2) blocked on lock 0xffffffffae3c7040 (sleep mutex) "jkl"
thread 100144 (pid 954, crash2: 3) blocked on lock 0xffffffffae3c6f80 (sleep mutex) "abc"
thread 100055 (pid 954, crash2: 0) blocked on lock 0xffffffffae3c6fc0 (sleep mutex) "def"
thread 100142 (pid 954, crash2: 1) blocked on lock 0xffffffffae3c7000 (sleep mutex) "ghi"
...
```

Figure 7: Example `show lockchain` Output

```
db> ps
  pid  ppid  pgrp   uid   state   wmesg         wchan        cmd
  811     0     0     0  SL      (threaded)                  crash2
100139                   D       fee      0xffffffffae3a9180 [crash2: 3]
100138                   D       four     0xffffffffae3a9140 [crash2: 2]
100137                   D       fo       0xffffffffae3a9240 [crash2: 1]
100136                   D       two      0xffffffffae3a90c0 [crash2: 0]
...
db> show lock fee
 class: lockmgr
 name: fee
 lock type: fee
 state: EXCL (count 1) 0xffffff00013079c0 (tid 100136, pid 811, "crash2: 0")
 waiters: 1
db> show sleepchain 100139
thread 100139 (pid 811, crash2: 3) blocked on lk "fee" EXCL (count 1)
thread 100136 (pid 811, crash2: 0) blocked on sx "two" XLOCK
thread 100137 (pid 811, crash2: 1) blocked on lk "fo" EXCL (count 1)
thread 100138 (pid 811, crash2: 2) blocked on sx "four" XLOCK
thread 100139 (pid 811, crash2: 3) blocked on lk "fee" EXCL (count 1)
...
```

Figure 8: Example `show sleepchain` Output

```
DB_COMMAND(foo, db_foo_cmd)
{
        struct foo *foop;
        int i;

        if (have_addr)
                foop = (struct foo *)addr;
        else
                foop = &default_foo;
        if (count == -1)
                count = 1;      /* Default count. */
        for (i = 0; i < count; i++)
                do_something(foop);
}
```

Figure 9: Sample DDB Command

the user. Instead, the input comes from the command line when the command is invoked. Commands do output various messages to the console, and DDB provides its own API for console output.

The primary routine in DDB's I/O API is `db_printf`. This function takes the same arguments as `printf(9)` and supports all of the same output formats. This includes the extended formats `%b` and `%D`. DDB command functions should use `db_printf` for all console output.

An additional detail of DDB's I/O subsystem that DDB commands may need to handle is the pager. DDB's output includes a builtin pager which will interrupt the output with a `--More--` prompt periodically. If a command does not wish to have any of its output interrupted it may disable the pager entirely by calling `db_disable_pager`. The `panic` command does this for example. A DDB command that produces a lot of output (for example, one that iterates over a list) should honor a request by the user to abort the current command at the pager prompt. If the user aborts a command, then the global variable `db_pager_quit` will be set to true. Thus, DDB command functions simply need to check the state of `db_pager_quit` periodically and gracefully exit when it is non-zero. Figure 10 contains a sample "show foos" command which walks a list of `struct foo` objects displaying information about each object. It supports a "v" flag to enable more verbose output.

3.3.3 Using DDB to Map Addresses to Symbols

Another useful debugging tool DDB provides is the ability to use its symbol tables to map addresses to symbolic names. This can be very useful for looking up the name of a function for a function pointer. This is especially true when working with facilities that work on lists of function pointers such as taskqueue tasks, callouts, or SYSINITs. Note that these routines can be used outside of DDB. However, doing so may result in races with loading kernel modules, so care should be taken.

The `db_search_symbol` function is used to map a specific address to a symbol. It accepts an address as its first argument, a strategy as its second argument, and a pointer to a `db_expr_t` variable as its third argument. The strategy argument can either by `DB_STGY_PROC` to only match functions or `DB_STGY_ANY` to match any symbol. The third argument cannot be `NULL` as `db_search_symbol` assumes it always points to valid storage. Upon successful completion, the function returns a pointer to a symbol. It also stores the offset of the address relative to the symbol in the variable pointed to by the third argument. If no appropriate symbol was found, then `db_search_symbol` returns `C_DB_SYM_NULL`.

```
DB_SHOW_COMMAND(foos, db_show_foos_cmd)
{
        struct foo *foop;
        int verbose;

        verbose = index(modif, ’v’) != NULL;
        TAILQ_FOREACH(foop, &allfoos, f_list) {
                if (verbose)
                        db_printf("%p: ", foop);
                db_printf("%s (%d)\n" foop->f_name, foop->f_count);
                if (db_pager_quit)
                        break;
        }
}
```

Figure 10: Sample DDB “show” Command

The `db_symbol_values` function is used to obtain the name and value of a symbol. The first argument is a pointer to a symbol (such as returned from `db_search_symbol`). The second argument is a pointer to a `const char *` and the third argument is a pointer to a `db_expr_t`. The second argument must point to valid storage, but the third argument can be `NULL`. On return from the function, the second argument will point to the name of the symbol or will have the value `NULL` if the first argument was an invalid symbol. The third argument will hold the value of the symbol (i.e., its address). Figure 11 shows the code from the `VERBOSE_SYSINIT` kernel option which outputs the name of each SYSINIT routine executed during boot.

The `db_printsym` routine is a wrapper around the previous two routines. It accepts an address as its first argument and a strategy as its second argument. It looks up the symbol for the address and prints the name using `db_printf`. If the offset of the address is non-zero, then it appends a “+” character followed by the offset to the output. This is the routine used by DDB’s stack trace command the print the return address for each stack frame.

4 Debugging with kgdb

The kgdb program is a wrapper around gdb that is used for analyzing a kernel. Unlike DDB which is integrated into the kernel and self-contained, kgdb is an external program. As a result, it requires more setup work. However, it also can target several different environments. DDB can only be used to debug the currently running kernel on the same machine and only by halting the kernel. The kgdb debugger can be used to analyze a kernel crash, inspect the currently running kernel, or debug a halted kernel on another machine. In addition, it provides a much richer debugging environment than DDB including source-level debugging, access to local symbols, and scripting that supports control flow.

4.1 Inspecting Processes and Threads

In general, kgdb treats the kernel as if one were using gdb to analyze a single multithreaded process. Each kernel thread is mapped to a single gdb thread. Thus, the usual gdb commands for managing threads (e.g. `info threads` and `thread`) can be used with kernel threads as well. One slight annoyance, however, is that the thread IDs kgdb uses have no relation to the PIDs and TIDs the FreeBSD kernel uses to identify processes

```
#if defined(DDB)
        const char *name;
        c_db_sym_t sym;
        db_expr_t  offset;

        sym = db_search_symbol((vm_offset_t)(*sipp)->func,
            DB_STGY_PROC, &offset);
        db_symbol_values(sym, &name, NULL);
        if (name != NULL)
                printf("   %s(%p)... ", name, (*sipp)->udata);
        else
#endif
                printf("   %p(%p)... ", (*sipp)->func,
                    (*sipp)->udata);
```

Figure 11: Mapping Function Pointers to Names for `VERBOSE_SYSINIT`

and threads. Thus, to switch to a thread with a specific TID or PID one has to examine the thread list from `info threads` to map a TID or PID to a gdb thread ID.

To alleviate this inconvenience, kgdb provides `proc` and `tid` commands. The `proc` command accepts a PID and switches to the thread context of the first thread for the specified process. The `tid` command accepts a TID and switches to the corresponding thread. Note that the `proc` command does not work with remote debugging.

4.2 Debugging Kernel Modules

Kernel modules (also called "klds") are separate object files that can be loaded into the kernel's address space at runtime. Each kernel module contains its own symbols that are separate from the kernel's symbols. DDB uses a merged symbol table that is updated by the kernel linker when modules are loaded and unloaded. The kgdb debugger, on the other hand, has to explicitly load symbols for each kernel module from an appropriate symbol file.

An arbitrary symbol file can be loaded in kgdb using the `add-symbol-file` command. This command requires the relocated addresses of each section as command arguments. Doing this by hand is a bit tedious. It involves extracting the base address of the kernel module from the kernel (e.g. using kldstat(8)), and the relative addresses of each section from the kernel module (e.g. using objdump(8)). The relocated address of each section is then computed by adding its relative address to the base address of the module. Thankfully, there are ways to automate this process.

4.2.1 kgdb KLD Support

Recent versions of kgdb provide integrated support for managing kernel modules. First, the `add-kld` command can be used to manually load the symbols for a single module. Second, kgdb uses gdb's support for shared libraries to automatically load symbols for modules. Note that both of these features only work for a kernel with debug symbols.

The `add-kld` command accepts as its sole argument a pathname of a kernel module and loads the symbols for that module. The path can either be an absolute path or a relative path. If it is a relative path, then kgdb will look for the module in several directories: the current working directory, the directory of the current kernel executable, and each directory in the target kernel's module path. If a kernel module is found, then its filename is matched to one of the target kernel's loaded modules. The base address for the loaded module is read from the target kernel and used to relocate the section addresses in the kernel module symbol file. Basically, `add-kld` is a wrapper around the gdb command `add-symbol-file` that does all the math internally. As with `add-symbol-file`, the only way to unload symbols added via `add-kld` is to clear all symbols via the `file` or `symbol-file` commands.

For more automated handling of kernel modules, kgdb hooks into gdb's shared library support and treats kernel modules as shared libraries. As a result, the standard commands for manipulating shared libraries in gdb such as `info sharedlibrary`, `sharedlibrary`, and `nosharedlibrary` can be used to manage kernel module symbols. In addition, sections from kernel modules loaded via the shared library mechanism are listed in the `info files` output. Figure 12 shows the kernel modules loaded on my laptop.

To locate the corresponding file for a kernel module, kgdb will first use the absolute path stored in the kernel image for 8.0 and later. Note that you can use `set solib-absolute-prefix` to force a prefix for the absolute paths. If the absolute path is not present (or the corresponding file is not present), then kgdb will first search for the file in paths set via `set solib-search-path`. If that fails, then kgdb will search the same set of paths as the `add-kld` command.

Using this facility, symbols for kernel modules are automatically loaded when a vmcore file is used as the target. When debugging a remote target, on the other hand, symbols for kernel modules are not automatically loaded when attaching to the target. However, invoking the `info sharedlibrary` command will cause kgdb to query the list of kernel modules from the remote kernel. Afterward the `sharedlibrary` command can be used to load symbols for the modules.

4.2.2 Using asf(8)

For older versions of kgdb, the asf(8) [10] tool can be used to automate the loading of kld symbols. Specifically, asf(8) searches for kernel modules corresponding to a set of loaded modules and then generates a text file containing `add-symbol-file` commands to load the symbols for each module. Note that by default, asf(8) expects to parse output from kldstat(8) on its standard input to obtain the list of kernel modules. However, the `-M` and `-N` options can be used to make asf(8) read the list of kernel modules directly from a vmcore similar to kgdb. Also, asf(8) assumes that it is invoked from a kernel build directory. If you wish it to load symbols from the modules in the installed location you will need to use the `-s` flag and specify an explicit kernel module path. Once asf(8) has generated a gdb command file, the symbols can be loaded by using the `source` command from kgdb to execute the commands in the generated file. Figure 13 shows the command file generated by asf(8) for the modules loaded on my laptop. Note that the addresses of the various named sections in the command for `iwi_bss.ko` match the addresses in the `info files` output from Figure 12.

4.3 Extending kgdb via Scripts

Similar to DDB, kgdb can be extended by adding new commands. Rather than requiring a recompile of the kernel, new commands can be added on the fly using gdb's scripting language. GDB scripts are evaluated at runtime and are not pre-compiled. On the one hand this provides several benefits. For example, the physical layout of structures are not hardcoded into the scripts when writing them. Instead, gdb uses symbols from the kernel and modules to compute the offsets of member names as well as the addresses of global symbols. Also, gdb does not evaluate statements that are not executed. Thus, one can use members of structures that are not always present (e.g. when a new member is added) by using conditional execution. The downside is that gdb scripts require a kernel built with debug symbols for all but the simplest tasks. The gdb info documentation covers the basics of scripts, or user defined commands, but there are several quirks that are worth mentioning.

First, while gdb scripts do support control flow via while loops and if-then-else statements, there are a few limitations. For

```
> sudo kgdb -q
Reading symbols from /boot/kernel/iwi_bss.ko...
Reading symbols from /boot/kernel/iwi_bss.ko.symbols...done.
done.
Loaded symbols for /boot/kernel/iwi_bss.ko
Reading symbols from /boot/kernel/logo_saver.ko...
Reading symbols from /boot/kernel/logo_saver.ko.symbols...done.
done.
Loaded symbols for /boot/kernel/logo_saver.ko
...
(kgdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0xc3e8e5a0  0xc3e8e63b  Yes         /boot/kernel/iwi_bss.ko
0xc41037a0  0xc4103c28  Yes         /boot/kernel/logo_saver.ko
(kgdb) info files
Symbols from "/boot/kernel/kernel".
kernel core dump file:
        ‘/dev/mem’, file type FreeBSD kernel vmcore.
Local exec file:
        ‘/boot/kernel/kernel’, file type elf32-i386-freebsd.
        Entry point: 0xc04513c0
...
        0xc3e8e5a0 - 0xc3e8e63b is .text in /boot/kernel/iwi_bss.ko
        0xc3e8e63b - 0xc3e8e724 is .rodata in /boot/kernel/iwi_bss.ko
        0xc3e8f000 - 0xc3ebdb04 is .data in /boot/kernel/iwi_bss.ko
        0xc3ebdb04 - 0xc3ebdb7c is .dynamic in /boot/kernel/iwi_bss.ko
        0xc3ebdb7c - 0xc3ebdb88 is .got in /boot/kernel/iwi_bss.ko
        0xc3ebdb88 - 0xc3ebdb8c is .bss in /boot/kernel/iwi_bss.ko
...
```

Figure 12: Examining Kernel Modules from kgdb

```
> sudo asf -o - -N /boot/kernel/kernel -M /dev/mem -s /boot/kernel
add-symbol-file /boot/kernel/iwi_bss.ko.symbols 0xc3e8e5a0
 -s .data 0xc3e8f000 -s .bss 0xc3ebdb88
add-symbol-file /boot/kernel/logo_saver.ko.symbols 0xc41037a0
 -s .data 0xc4104c80 -s .bss 0xc4106ee8
```

Figure 13: Sample kgdb Command File Generated by asf(8)

example, there is no direct “else-if” construct. Instead, one must include a nested `if` statement inside an else block. Figure 14 shows a simple example of this. In addition, there are no equivalents to the C statements `break`, `continue`, or `return`. There are gdb commands which have those names, but they affect the execution of the program being debugged (e.g. setting a breakpoint). Newer versions of gdb do add `loop_break` and `loop_continue` but FreeBSD’s gdb does not have those commands.

Second, the implementation of arguments to user-defined commands has several subtle implications. First, there is no easy way for a command to figure out how many arguments the user passed to it. However, if the command references an argument the user did not define, then gdb will halt execution of the command with an error. Second, as described in the documentation, gdb replaces the argument variables with the text of the user-supplied argument before evaluating expressions rather than evaluating the user-supplied expression and creating a new variable with that value. This means that you cannot treat the arguments as local variables with local scope. However, it does mean that any variables passed as arguments to user-defined commands are effectively passed by reference. This provides a way to return values from user-defined commands by assigning values to argument variables.

Third, working with string literals can be awkward. Specifically, one cannot assign a string literal to a convenience variable or index a string literal unless gdb is attached to a live process, and core dumps do not count as live processes. As a result, to compare a variable in a core to a known string one has to explicitly compare invidual characters. While this is tedious, this can be useful. In Figure 15 the contents of the `machine_arch` global variable are used to determine the current architecture and include another command file with architecture-specific commands.

Finally, there is no way to abort execution of a user-defined command. If a user-defined command gets stuck in an infinite loop, for example, the sole recourse is to kill the kgdb process. A command can be aborted at the pager prompt if it emits a full page of output. However, one cannot use `Ctrl-C` or something similar to abort execution of a command.

5 Examining Crashdumps with System Utilities

Several system utilities can examine crash dumps instead of the running kernel. In general, these utilities accept two optional arguments: `-M` and `-N`. These arguments specify an alternate core file and kernel image, respectively. Some of the utilities which support this feature include ddb(8), dmesg(8), fstat(1), iostat(8), ipcs(1), netstat(1), nfsstat(1), ps(1), pstat(8), and vmstat(8).

6 Debugging Strategies

Kernel bugs manifest in several different ways. Some bugs trigger a panic, but other bugs may result in a hang or a partial loss of function. For example, if several threads are locked in a deadlock, they may not hang the entire machine but only impair certain operations. These differing consequences require different strategies for finding the bug.

6.1 Kernel Crashes

Kernel crashes can often be investigated with a very straightforward approach. Often, the panic message itself points to the problem. For those crashes, the context of the panic in the source is sufficient to determine the cause of the crash.

Some crashes are an indirect result of a bug, however. For example, a corrupted data structure will usually result in a memory protection exception such as a page fault. For these crashes, simply examining the source line where the crash occurred will usually lead to the data structure that is in an invalid state. Inspecting the data structure more

```
def foo
        if ($arg0 > 10)
                print "big number"
        else
                if ($arg0 > 5)
                        print "medium number"
                else
                        print "small number"
                end
        end
end
```

Figure 14: Sample GDB Script Else-If Construct

```
if (machine_arch[0] == ’a’ && machine_arch[1] == ’m’ && machine_arch[2] == ’d’)
    source gdb6.amd64
    set $__amd64__ = 1
end
if (machine_arch[0] == ’i’ && machine_arch[1] == ’3’ && machine_arch[2] == ’8’)
    source gdb6.i386
    set $__i386__ = 1
end
```

Figure 15: Including a Machine Dependent kgdb Command File

closely as well as the code around the crash point is often sufficient to determine the cause of the bug.

Another crash that can be a secondary effect is a crash due to exhausting the space in the “kmem” virtual memory map. The “kmem” virtual memory map is used to provide virtual address space for memory allocated via malloc(9) or uma(9) in the kernel. On architectures with a direct map such as amd64, “kmem” is only used for allocations larger than a page. On other architectures “kmem” is used for all allocations. If the amount of virtual address space in the “kmem” map is exhausted, then the kernel will crash. This can sometimes be the result of resource exhaustion. For example, if `kern.ipc.nmbclusters` is set to a high value and a `m_getcl(M_WAIT)` invocation causes the “kmem” map to be exhausted before the `nmbclusters` limit is reached, then the kernel will panic.

Sometimes the “bug” can actually be faulty hardware. For example, a pointer might have a bit error. This can result in a page fault for a `NULL` pointer. One way to verify if a crash on an x86 machine was the result of a hardware error is to check the system event log. This can usually be examined from the BIOS setup. For systems with a BMC, the ipmitool [11] utility can be used to examine the system event log at runtime. Lack of a corresponding entry in the system event log doesn’t necessarily disprove a hardware failure, but if an entry is present it can confirm failing hardware as the panic’s cause.

6.2 Kernel Hangs

Kernel hangs tend to require a bit more sleuthing. One reason for this is that it can sometimes take a bit of investigating to figure out the true extent of the hang. Here are a few things to try to start the investigation of a hang.

First, check for resource starvation. For example, check for messages on the console about the `kern.maxfiles` or `maxproc` limits being exceeded. Sometimes a machine that is overloaded will appear to be hung because it is unable to fork a new process for a remote login, for example. Login to the box on the console if possible and check for other resource exhaustion issues using commands like netstat(1) and vmstat(1).

The next step is generally to break into DDB. The `ps` command in DDB can give a very useful overview of the system. For example, if all of the CPUs are idle, then there may be a deadlock. The `ps` command can be used to look for suspect threads which can then be investigated further. On the other hand, if all of the CPUs are busy, then that may indicate a livelock condition (or an overloaded box).

If the hang's cause is still unknown, then the `panic` command can be used from DDB to explicitly panic the machine. If the machine is configured for crashdumps, then it will write out a crash. After the machine has rebooted the crashdump can be used to examine the hang further. For example, if logging into the box to run `netstat` was not possible, then `netstat` can be run against the crashdump.

7 Conclusion

The FreeBSD kernel has bugs just like any other piece of software. To aid in the investigation and fixing of bugs, FreeBSD provides several kernel debugging tools. Some of the tools are services within the kernel itself such as DDB. Other tools are outside of the kernel such as kgdb. As with other tools, skilled use is obtained from practice and a bit of trial and error.

8 Availability

The sources of both DDB and kgdb are present in the FreeBSD source tree. While the core DDB sources are present in `src/sys/ddb`, the source to several DDB commands are present in different parts of the kernel sources. The kgdb sources are all found in `src/gnu/usr.bin/gdb/kgdb`.

The `show lock` DDB command was added in FreeBSD 6.1. The `show proc`, `show thread`, `show turnstile`, `show lockchain`, and `show sleepchain` commands were added in FreeBSD 6.2.

Several recent changes to kgdb will first appear in FreeBSD 6.4 and 7.1. These include the integrated kernel module support as well as the 'tid' command supporting remote targets. Also, while the 'proc' command has been present since 6.0, the 'tid' command first appeared in 7.0.

There are several existing sets of kgdb command files containing various user-defined commands. Some scripts are present in the FreeBSD source tree under the `src/tools/debugscripts` directory. The scripts at `http://www.FreeBSD.org/~jhb/gdb` include user-defined commands that provide similar functionality to many DDB commands such as `ps`, `lockchain`, and `sleepchain`.

References

[1] The GNU Project Debugger, `http://www.gnu.org/software/gdb`

[2] *DDB*, FreeBSD Kernel Interfaces Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[3] *kgdb*, FreeBSD General Commands Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[4] *Kernel Debugging*, FreeBSD Developers' Handbook, `http://www.FreeBSD.org/doc/en/books/developers-handbook`

[5] *Mutex*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[6] *RMLock*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[7] *RWLock*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[8] *SX*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[9] *Lock*, FreeBSD Kernel Developer's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[10] *ASF*, FreeBSD System Manager's Manual, `http://www.FreeBSD.org/cgi/man.cgi`

[11] IPMItool, `http://ipmitool.sourceforge.net`