The History and Future of Core Dumps in FreeBSD

Sam W. Gwydir, Texas A&M University sam@samgwydir.com

January 13, 2017

Contents

1 Abstract

Crash dumps, also known as core dumps, have been a part of BSD since it's beginnings in Research UNIX. A core dump is "a copy of memory that is saved on secondary storage by the kernel" for debugging a system failure[1]. Though 38 years have passed since `doadump()` came about in UNIX/32V, core dumps are still needed and utilized in much the same way they were then. Given this, one might assume the core dump code changed little over time but, with some research, this assumption has proven incorrect.

What has changed over time is where core dumps are sent to and what processor architectures are supported. Previous to the advent of UNIX, core dumps were printed to a line printer or punch cards. At the birth of UNIX core dumps were made to magnetic tape and because UNIX only supported the PDP-11, it was the only architecture supported for dumps. Over time machine architecture support has evolved from different PDP-11 models to hp300, i386 up to the present day with AMD64 and ARM64. In addition the type of dump device has changed from tape, to hard disk or another machine over a LAN.

The following paper begins with a quick background on what core dumps are and why operators might need them. Following that the current state of the core dump facility and some of the more common extensions in use are examined. We conclude with a call to action for upstreaming these extensions and modularizing the core dump code such that different methods of core dump can be dynamically loaded into the kernel on demand.

In addition a complete history of core dumps in UNIX and BSD was produced as research for this paper and can be found in the appendix.

2 Introduction

The BSD core dump facility performs a simple yet vital service to the operator: preserving a copy of the contents of system memory at the time of a fatal error for later debugging.

This copy or "dump" can be a machine readable form of the complete contents of system memory, or just the set of kernel pages that are active at the time of the crash. There is also support for dumping a less complete but human readable debugger scripting output.

Throughout the history of UNIX operating systems, different methods have been used to produce a core dump. In the earliest UNIXes magnetic tape was the only supported dump device but when hard disk support matured swap space was used, obviating the need for changing out tapes before a dump[2]. Modern and embedded systems continue to introduce new constraints that have motivated the need for newer methods of ex-filtrating a core dump from a faltering kernel.

The FreeBSD variant of the BSD operating system has introduced gradual extensions to the core dumping facility. FreeBSD 6.2 introduced "minidumps", a subset of a full dump that only consists of active kernel memory. FreeBSD 7.1's `textdumps(4)` consist of the result of debugger commands input interactivly in DDB or via script[3]. FreeBSD 12-CURRENT introduced support for public-key cryptographic encryption of core dumps.

Though not in the main source tree, compressed dumps and the ability to dump to a remote network device exist and function. While promising, these extensions have been inconsistent in their integration and interoperability.

Another BSD derived OS, Mac OS X has also introduced similar compression and network dumping features into their kernel albeit with a distinct pedigree from FreeBSD[4, 5].

The following paper will provide a historical survey of the dump facility itself, from its' introduction in UNIX to its' current form in modern BSDs and BSD derived operating systems. We will also explore these core dump extensions, associated tools, and describe an active effort to fully modularize them, allowing the operator to enable one or more of them simultaneously.

[1] The Design and Implementation of the FreeBSD operating system by McKusick, Neville-Neil, and Watson

[2] crash(8) - 3BSD

[3] `https://lists.freebsd.org/pipermail/freebsd-current/2007-December/081626.html`

[4] `https://developer.apple.com/library/content/technotes/tn2004/tn2118.html`

[5] `https://github.com/opensource-apple/xnu/blob/27ffc00f33925b582391b1ef318b78b8bd3939d1/osfmk/kdp/kdp_core.c#L527`

3 Motivation

In UNIX and early BSD's core dumps were originally made to magnetic tape which was superseded by dumping to a swap partition on a hard disk since at least 3BSD. For decades since, increases in physical system memory and swap partition size have loosely tracked increases in available persistent memory, allowing for the continued use of this paradigm.

However, recent advances in commodity system hardware have upended the traditional memory to disk space ratio with systems now routinely utilizing 1TB or more physical memory whilst running on less than 256GB of solid state disk. Given that the kernel memory footprint has grown in size, the assumption that disk space would always allow for a swap partition large enough for a core dump has proved to be inaccurate. This change has spurred development of several extensions to the core dumping facility, including compressed dumping to swap and dumping over the network to a server with disk space for modern core dumps. Because dumps contain all the contents of memory any sensitive information in flight at the time of a crash appears in the dump. For this reason encrypted dumps have been recently added to FreeBSD[6].

While dealing with the above problems the author and his colleagues became intimately familiar with the state of the core dump code and its' associated documentation. As users of the core dump code they felt a need for more flexibility and extensibity in the core dump routines of FreeBSD. The author intends to provide a basis for the argument that the core dump code should be modularized for the flexibility that provides to operators.

In addition it is hoped that the information herein is of use to inform further work on core dumps, failing that we hope it is interesting.

[6] `https://svnweb.freebsd.org/base?view=revision&revision=309818`

4 The Present

4.1 Core Dumps in FreeBSD

4.1.1 Full Core Dump Procedure

When a UNIX-like system such as FreeBSD encounters an unrecoverable and unexpected error the kernel will "panic". Though the word panic has connotations of irrationality, the function `panic(9)` maintains composure while it shutsdown the running system and attempts to save a core dump to a configured dump device.

What follows is a thorough description of the FreeBSD core dump routine (as of FreeBSD 11-RELEASE) starting with `doadump()` in `sys/kern/kern_shutdown.c`.

`doadump()` is called by `kern_reboot()`, which shuts down "the system cleanly to prepare for reboot, halt, or power off." [7] `kern_reboot()` calls `doadump()` if the `RB_DUMP` flag is set and the system is not "cold" or already creating a core dump. `doadump()` takes a boolean informing it to whether or not to take a "text dump", a form of dump carried out if the online kernel debugger, `DDB`, is built into the running kernel. `doaddump()` returns an error code if the system is currently creating a dump, the dumper is NULL and returns error codes on behalf of `dumpsys()`.

`doadump(boolean_t textdump)` starts the core dump procedure by saving the current context with a call to `savectx()`. At this point if they are configured, a "text dump" can be carried out. Otherwise a core dump is invoked using `dumpsys()`, passing it a `struct dumper`. `dumpsys()` is defined on a per-architecture basis. This allows different architectures to setup their dump structure differently. `dumpsys()` calls `dumpsys_generic()` passing along the `struct dumperinfo` it was called with. `dumpsys_generic()` is defined in `sys/kern/kern_dump.c` and is the meat of the core dump procedure.

There are several main steps to the `dumpsys_generic()` procedure. The main steps are as follows. At any point if there is an error condition, goto failure cleanup at the end of the procedure.

1. Fill in the ELF header.

[7] `kern_shutdown.c - https://svnweb.freebsd.org/base/head/sys/kern/kern_shutdown.c?view=markup#l1336`

2. Calculate the dump size.
3. Determine if the dump device is large enough.
4. Begin Dump
   (a) Leader (Padding)
   (b) ELF Header
   (c) Program Headers
   (d) Memory Chunks
   (e) Trailer
5. End Dump

After this is done the kernel writes out a NULL byte to "Signal completion, signoff and exit stage left." And our core dump is complete.

Full Core Dump Contents The canonical form of core dump is the "full dump". Full dumps are created via the `doadump()` code path which starts in `sys/kern/kern_shutdown.c`. The resulting dump is an ELF formatted binary written to a configured swap partition. The following is based on amd64 code and is the result of `dumpsys_generic()`. This will be similar in format but different values for different architectures.

Table 1: Full Dump Format

Field	Description
Leader	See Table 2
ELF Header	See Table 3
Program Headers	
Memory Chunks	
Trailer	See Table 2

Table 2: `kerneldumpheader` Format

Field	Value
magic	"FreeBSD Kernel Dump"
architecture	"amd64"
version	1 (kdh format version)
architectureversion	2
dumplength	varies, excluding headers
dumptime	current time
blocksize	block size
hostname	hostname
versionstring	version of OS
panicstring	message given to `panic(9)`
parity	parity bits

Table 3: ELF Header Format

`ehdr` Field	Value
`e_ident[EI_MAG0]`	`0x7f`
`e_ident[EI_MAG1]`	'E'
`e_ident[EI_MAG2]`	'L'
`e_ident[EI_MAG3]`	'F'
`e_ident[EI_CLASS]`	2 (64-bit)
`e_ident[EI_DATA]`	1 (little endian)
`e_ident[EI_VERSION]`	1 (ELF version 1)
`e_ident[EI_OSABI]`	255
`e_type`	4 (core)
`e_machine`	62 (x86-64)
`e_phoff`	size of this header
`e_flags`	`0`
`e_ehsize`	size of this header
`e_phentsize`	size of program header
`e_shentsize`	size of section header

4.1.2 Minidump Procedure and Contents

4.1.3 Textdump Procedure and Contents

4.2 Core Dumps in Mac OS X

Mac OS X is capable of creating gzipped core dumps and dumping locally, or over the network using a modified `tftpd(8)` daemon they call `kdumpd(8)`. In addition dumps over FireWire are supported for situations where the kernel panic is caused by the Ethernet driver of network code.

In `xnu/osfmk/kdp/kdp_core.c` Mac OS X gzips its' core dump before writing it out to disk, and is otherwise much like the FreeBSD "full dump" procedure with one major difference. Notably, Mac OS X uses a different executable image-format called Mach-O, as opposed to ELF, because OS X runs a hybrid Mach and BSD kernel called XNU.

Network dumping "has been present since Mac OS X 10.3 for PowerPC-based Macintosh systems, and since Mac OS X 10.4.7 for Intel-based Macintosh systems." From `kdumpd(8)`:

> Kdumpd is a server which receives kernel states in the form of a core dump from a remote Mac OS X machine.
>
> ...
>
> The kdumpd command is based on Berkeley tftpd(8) by way of FreeBSD, with several modifications.

4.3 Core Dumps in Solaris

Solaris has several features that others don't.

- `savecore(1M)` has the ability to "live dump", creating a dump of a running system. `savecore(1M)` does note that this dump will not be entirely self consistent because the machine is not halted while dumping.
- `dumpadm(1M)` allows save compression and dumping to swap on zvol(!!!)
- `dumpadm(1M)` as of Solaris 11.2 has a dump size estimation feature that will attempt to estimate the size of a dump given your current configuration.

5 The Future

There are several extensions to the FreeBSD core dump code that exist as sets of patches on mailing lists and wikis but are not found in upstream FreeBSD.

First we provide some background on several extensions and tools including dumping over the network, compressed dumps and a tool for estimating the size of a minidump. Then we will explore the benifets of modularized core dump code.

5.1 `netdump` - Network Dump

Crash dumping over the network can be especially useful in embedded systems that do not have adequately sized swap partitions.

The original netdump code was written by Darrell Anderson at Duke around 2000 in the FreeBSD 4.x era as a kernel module. This code was later ported to modern FreeBSD in 2010 at Sandvine with the intention of being part of FreeBSD 9.0, which did not succeed.

Currently there exists working netdump code at Isilon that applies cleanly to versions of FreeBSD after 11 but before encrypted dump was added in FreeBSD 12-CURRENT. Network dumps have not yet made it into upstream FreeBSD.

5.2 Compressed Dump

Modern systems often have several hundred gigabytes of RAM and will soon often have terabytes. This means full crash dumps, even minidumps, can be much larger than most sensible amounts of swap.

Though `savecore(8)` has the ability to compress core dumps with the `'-z'` option, this only compresses a core once it is copied into the main filesystem. The core dump that was written to the swap partition remains uncompressed.

Compressed dumps see a 6 to 14 compression ratio for core dumps with a slight penalty in the time require to write the dump initially[8]. However the following `savecore(8)` on the next boot is faster, resulting in a faster dump and reboot sequence.

Compressed dumps have not yet made it into upstream FreeBSD.

5.3 `minidumpsz` - Minidump Size Estimation

`minidumpsz` is a kernel module that can do an online estimation of the size of a minidump if it were to occur at the time `sysctl debug.mini_dump_size` is called.

`minidumpsz` performs an inactive version of the minidump routine, `minidumpsys()`, to estimate the size of a dump if it were to take place at the time of the sysctl's calling.

`minidumpsz` was created by Rodney W. Grimes for the author's work at Groupon and applies to FreeBSD 10.1 and FreeBSD 11. `minidumpsz` has not yet made it into upstream FreeBSD.

5.4 Modularizing Dump Code

- Backporting features and fixes added to dump code becomes trivial
- Development becomes easier because LKMs are easier to work with
- Embedded systems benifet from a smaller kernel

[8] `https://lists.freebsd.org/pipermail/freebsd-arch/2014-November/016231.html`

5.5 Live Dump

5.6 Dump to swap on zvol

6 Conclusion

Though it may seem like core dumps are a solved problem from the past, it turns out the core dump code is an ever changing routine that is constantly being adapted to the times.

6.1 Recommendations

- textdumps by defaults, but with better defaults?
- documentation should include reccomendations on swap size for different amounts of ram
  - include amounts for fulldump, minidump and textdump at certain RAM sizes

7 Acknowledgments

The author would like thank Michael Dexter for his help debugging the original issues that led to our current combined knowledge of core dumps. In addition Rodney W. Grimes' help reading code, from PDP-11 assembly to modern C, along with his historical knowledge was invaluable.

The author thanks Deb Goodkin of the FreeBSD Foundation for her help bringing the author into the FreeBSD community and lastly thanks the FreeBSD community in general for making this day and paper possible.

8 Appendix

8.1 The Past: A Complete History of Core Dumps

The following sections list when different features of the core dump code were introduced starting with the core dump code itself. First the dump facility will be followed through the later versions of Research UNIX and then BSD through to present versions of FreeBSD.

8.2 Core Dumps in UNIX

Core dumping was initially a manual process. As documented in Version 6 AT&T UNIX's `crash(8)`, an operator could take a core dump "if [they felt] up to debugging". Though 6th Edition is not the first appearance of dump code in UNIX, it is the first complete repository of code the public has access to.

8.2.1 6th Edition UNIX

In 6th Edition UNIX `crash(8)` teaches us how to manually take a core dump:

> If the reason for the crash is not evident (see below for guidance on 'evident') you may want to try to dump the system if you feel up to debugging. At the moment a dump can be taken only on magtape. With a tape mounted and ready, stop the machine, load address 44, and start. This should write a copy of all of core on the tape with an EOF mark.

6th Edition UNIX's core dump procedure is defined in `m40.s` and `m45.s` give UNIX support for the PDP-11/40 and PDP-11/45.

8.2.2 7th Edition UNIX

7th Edition UNIX adds support for the PDP-11/70.

8.2.3 UNIX/32V

UNIX/32V was an early port of UNIX to the DEC VAX architecture making use of the C programming language to decouple the code from the PDP-11. `/usr/src/sys/sys/locore.s` contains the first appearance of `doadump()`, the same function name used today, written in VAX assembly.

8.3 Core Dumps in BSD

8.3.1 1BSD & 2BSD

1BSD and 2BSD inherited their dump code directly from 6th Edition UNIX and is therefore supports the PDP-11/40 and PDP-11/45.

8.3.2 3BSD

3BSD imports its' dump code from UNIX/32V maintaining the name `doadump()`. Because of this pedigree, `doadump()` is written in VAX assembly.

A "todo" list found in `usr/src/sys/sys/TODO` notes that "large core dumps are awful and even uninterruptible!".

8.3.3 4BSD

4BSD introduces a new feature to `doadump`, printing tracing information with `dumptrc`.

In addition, `usr/src/sys/sys/TODO` is the first mention of writing dumps to swap: "Support automatic dumps to paging area".

8.3.4 4.1BSD

Beginning in 4.1BSD `doadump()` is relegated to setting up the machine for `dumpsys()` which is written in C and found in `sys/vax/vax/machdep.c`.

As of 4.1c2BSD `doadump()` now fulfills the "todo" listed in 4BSD and dumps to the "paging area", or swap. `savecore(8)` is introduced to extract the core from the swap partition and place it in the filesystem.

- Support for VAX750, VAX780, VAX7ZZ (VAX730)
- In 4.1c2BSD changes VAX7ZZ references to VAX730

8.3.5 4.2BSD

- no changes.

8.3.6 4.3BSD

4.3 BSD-Tahoe

- Initial support is added for the "tahoe" processor and and `doadump` is ported to the tahoe.

4.3 BSD Net/1

- Same as 4.3-Tahoe

4.3 BSD-Reno

- hp300 and i386 core dump support is added in `usr/src/sys/hp300/locore.s` and `usr/src/sys/i386/locore.s`, respectively.

4.3 BSD Net/2

- Same as Reno

8.3.7 4.4BSD

- luna68k support added
- news3400 support added
- pmax support added
- sparc support added

4.4-BSD Lite1 & 4.4-BSD Lite2

- Same as 4.4BSD – changes made due to AT&T UNIX System Laboratories (USL) lawsuit.

8.3.8 386BSD

386BSD 0.0

- Reduce support to i386 and hp300 support

386BSD 0.1

- hp300 code removed

386BSD 0.1-patchkit

- Same as 386BSD 0.1

8.4 Core Dumps in FreeBSD

8.4.1 FreeBSD 1.0

- i386 support, hp300 support from 386BSD-0.1-patchkit

8.4.2 FreeBSD 2.0.0

FreeBSD 2.0.0

- `doadump()` no longer exists, though is mentioned in comments.

FreeBSD 2.2.0

- `dumpsys()` is placed inside `boot()` and `dumpsys()` in `kern_shutdown.c` because code was not seen as machine dependent.

8.4.3 FreeBSD 3.0.0

- SMP support
- alpha support

8.4.4 FreeBSD 4.0.0

- Added print uptime before rebooting.
- Better error message when dumps are not supported

8.4.5 FreeBSD 5.0.0

- Added IA64, sparc64, and pc98 support.
- New kernel dump infrastructure. Broken out to individual architectures again. `doadump()` is back!
- Crash dumps can now be obtained in the late stages of kernel initialisation before single user mode

8.4.6 FreeBSD 6.0.0

FreeBSD 6.0.0

- AMD64 and arm support added.
- AMD64 and i386 switch to ELF as their crash dump format.
- AMD64 and i386 bump their dump format to version 2.

FreeBSD 6.2.0

- minidump code added.

8.4.7 FreeBSD 7.0.0

FreeBSD 7.0.0

- sun4v support added
- minidumps are now default
- alpha support is removed

FreeBSD 7.1.0

- textdump code is added

8.4.8 FreeBSD 8.0.0

- PowerPC support added.
- mips support added.

8.4.9 FreeBSD 9.0.0

- Merge common amd64/i386 dump code under `sys/x86` subtree.
- Only dump at first panic in the event of a double panic
- Add dump command for DDB
- Minidump v2

8.4.10 FreeBSD 10.0.0

- On systems with SMP, CPUs other than the one processing the panic are stopped. This behavior is tunable with the sysctl 'kern.stop$_{\text{scheduleronpanic}}$'

8.4.11 FreeBSD 11.0.0

- RISC-V support added.
- arm64 support added.
- Factored out duplicated code from `dumpsys()` on each each architecture into `sys/kern/kern_dump.c`
- A 'show panic' command was added to DDB
- "4Kn" kernel dump support. Dumps are now written out in the native block size. `savecore(1)` updated accordingly.
- "4Kn" minidump support for AMD64 only
- `strlcpy(3)` is used to properly null-terminate strings in kernel dump header

8.4.12 FreeBSD 12-CURRENT

- Support for encrypted kernel crash dumps added. `dumpon(8)` and `savecore(8)` updated accordingly. New tool for decrypting cores added, `decryptcore(8)`. Tested on amd64, i386, mipsel and sparc64. Untested on arm and arm64. Encrypted textdump is not yet implemented.