The History and Future of Core Dumps in FreeBSD

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

Abstract

Crash dumps, also known as core dumps, have been a part of BSD since its beginnings in Research UNIX. 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. However, as underlying assumptions about the ratio of swap to RAM have proven inappropriate for modern systems, several extensions have been made by those who needed core dumps on very large servers, or very small embedded systems. Frustratingly these extensions have not made it to upstream FreeBSD.

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.

1 Introduction

The BSD core dump facility performs a simple yet vital service to the operator: it preserves 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[1]. Modern and embedded systems continue to introduce new constraints that have motivated the need for newer methods of exfiltrating 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[2]. 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[3, 4].

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] crash(8) - 3BSD

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

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

[4] `https://opensource.apple.com/source/xnu/xnu-3789.31.2/osfmk/kdp/kdp_core.c.auto.html`

2 Motivation

In UNIX and early BSDs 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[5].

While dealing with the above problems the author and his colleagues became closely 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.

3 The Present

3.1 Core Dumps in FreeBSD

3.1.1 Quick Background

While reading this paper you may wish to take a crash dump on your system and play around with the features discussed. The following is a quick "crash" course.

First configure the system dump device in `/etc/rc.conf` or using `dumpon(8)` [6]. The easiest way is to use `dumpdev='AUTO'`, which will set the `dumpdev` to the first configured swap device, make sure your swap partition is large enough for a core dump, if using `dumpon(8)` use swapinfo to find a suitable partition. Next, if you are using the default `dumpdir`, make sure it exists and set permissions accordingly. Now, in order to generate a kernel dump you will need to panic your kernel, there are several ways to do this, including writing a program that calls `panic(9)`, using dtrace to call panic or the simplest using the sysctl, `sysctl debug.kdb.panic=1`. Note this will crash and reboot your system.

For those who prefer shell to English:

```
# mkdir /var/crash
# chmod 700 /var/crash
# swapinfo
# dumpon -v /dev/da0p2
# sysctl debug.kdb.panic=1
```

If your `dumpdir` is configured correctly, `savecore(8)` will run automatically upon reboot. If not, run `savecore(8)` manually.

3.1.2 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 shuts down 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. `doadump()` 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()`

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

[6] `https://www.freebsd.org/doc/en/books/developers-handbook/kerneldebug.html`

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

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

After this is done the kernel gives a zero length block to dump_write() to "Signal completion, signoff and exit stage left." And our core dump is complete.

3.1.3 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, excludes headers
dumptime	current time
blocksize	block size
hostname	hostname
versionstring	version of OS
panicstring	panic(9) message
parity	parity bits

Table 3: ehdr ELF Header Format

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

3.1.4 Minidump Procedure and Contents

FreeBSD 6.2 introduced a new form of core dump termed, "minidumps". Instead of dumping all of phsyical memory to guarantee all relevent information is archived, minidumps dump "only memory pages in use by the kernel."[8]

Minidumps use a custom format in lieu of ELF. The format of a modern minidump (version 2) can be found in table 4.

Table 4: Mini Dump Format

Field	Description
Leader	See Table 2
Minidump Header	See Table 5
Message Buffer	message buffer contents
Bitmap	map of kernel pages
Kernel Page Directory	
Memory Chunks	
Trailer	See Table 2

Table 5: `minidumphdr` Format

Field	Value
`magic`	"minidump FreeBSD/amd64"
`version`	2
`msgbufsize`	size of message buffer
`bitmapsize`	size of bitmap
`pmapsize`	size of physical memory map
`kernbase`	ptr to start of kernel mem
`dmapbase`	ptr to start of direct map
`dmapend`	ptr to end of direct map

The minidump procedure in general is similiar to that of the full dump but with the added step of creating a bitmap that indicates which pages are to become part of the dump. The minidump procedure detailed here is based on the AMD64 code as found in `sys/amd64/amd64/minidump_machdep.c`[9], but it nearly identical for other architectures.

1. Create bitmap describing pages to be dumped.
2. Calculate the dump size.
3. Determine if the dump device is large enough.
4. Fill in minidump header
5. Fill in kernel dump header
6. Begin Dump
   (a) Leader
   (b) Minidump Header
   (c) Message Buffer
   (d) Bitmap
   (e) Kernel Page Directory
   (f) Memory Chunks
   (g) Trailer
7. End Dump

   The minidump will fail for any of the reasons a full dump will and also if the dump map grows while creating it. This will cause the routine to retry up to `dump_retry_count` times, the default is 5 times but can be set with the sysctl `machdep.dump_retry_count`.

3.1.5 Textdump Procedure and Contents

FreeBSD added a new type of dump, the `textdump(4)`. "The textdump facility allows the capture of kernel debugging information to disk in a human-readable rather than the machine-readable form normally used with kernel memory dumps and minidumps."[10] If `doadump()` in `kern_shutdown.c` is given a boolean value of 'true' then a minidump or full dump is cancelled and instead `textdump_dumpsys()` is invoked in `sys/ddb/db_textdump.c`.

Since textdumps are not binary data, textdumps are written out in the ustar tar file format. This tar contains several files listed in 6[11]. There exist several sysctls to select which files an operator wishes to include. These are listed in `textdump(4)`.

Table 6: `textdump(4)` Format

File	Description
Leader	See Table 2
`version.txt`	Kernel version string
`panic.txt`	Kernel panic message
`msgbuf.txt`	Kernel message buffer
`config.txt`	Kernel configuration
`ddb.txt`	Captured DDB output
Trailer	See Table 2

The `textdump(4)` procedure is similar in its setup to the other types of dumps but has several differences in

[8] `https://www.freebsd.org/doc/en/books/developers-handbook/kerneldebug.html`

[9] `https://svnweb.freebsd.org/base/head/sys/amd64/amd64/minidump_machdep.c?revision=157908&view=markup`

[10] `https://www.freebsd.org/cgi/man.cgi?query=textdump&apropos=0&sektion=0&manpath=FreeBSD+11.0-RELEASE+and+Ports&arch=default&format=html`

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

particular because the dump is in ustar format containing several text files instead of a binary format containing kernel pages.

1. Check if minimum amount of space is available on dump device
2. Set start of dump at the end of the swap partition minus the size of the dump header
3. Fill in kernel dump header
4. Begin Dump
   (a) Trailer
   (b) ddb.txt
   (c) config.txt
   (d) msgbuf.txt
   (e) panic.txt
   (f) version.txt
   (g) Header
   (h) Re-write Trailer with correct size
5. End Dump

If an error occurs during this procedure, report said error. If not, tell `dump_write()` to write a zero-length block to signifiy the end of the dump and report that the dump suceeded and return to executing the rest of the machine independent dump code.

3.2 Core Dumps in Mac OS X

Mac OS X is capable of creating compressed core dumps and dumping them locally, or over the network using a modified `tftpd(8)` from FreeBSD called `kdumpd(8)`[12]. 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."[3] In addition dumps over FireWire are supported for situations where the kernel panic is caused by the Ethernet driver or 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 besides its features[4]. 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[13].

1. Initialize gzip
2. Determine where to write dump
   (a) If local, determine offset to place file header, panic and core log
   (b) If remote, setup buffer for compressed core and packet size
3. Traverse the pmap for dumpable pages
4. Fill in Mach-O header
5. Begin Dump Write/Transmission
   (a) Mach-O Header
   (b) Information about panicked thread's state
   (c) Information about dump output location
   (d) Pad with zeroes to page align
   (e) Kernel Pages
   (f) Signal Completion with zero length write
   (g) Print out Information about Dump
   (h) If Local, write out debug log and gzip file header
6. End Dump Write/Transmission

If an error is detected at any point, return and report the given error message.

3.3 Core Dumps in Illumos

"illumos is a free and open-source Unix operating system. It derives from OpenSolaris, which in turn derives from SVR4 UNIX and Berkeley Software Distribution (BSD)."[14] Illumos has several attractive features in its core dump routine including "live dumping", compression and support for swap on zvol as a dump device.

The Illumos dump routine, `dumpsys()` can be found in `usr/sys/uts/common/os/dumpsubr.c`. In contrast to the other dump routines explained previously, the Illumos dump routine is very complex but with that complexity comes the several features mentioned above that are not available elsewhere.

Illumos' `savecore(1M)` has the ability to "live dump", creating a dump of a running system[15]. `savecore(1M)` does note that this dump will not be entirely self consistent because the machine is not suspended while dumping.

In addition to a version of `savecore(1M)`, Illumos has a tool analogous to FreeBSD's `dumpon(8)` called `dumpadm(1M)` which primarily is used to set the current dump device. Importantly this dump device can be a swap partition in a ZFS zvol. `dumpadm(1M)` is also used

[12] `https://opensource.apple.com/source/network_cmds/network_cmds-396.6/kdumpd.tproj/kdumpd.8.auto.html`
[13] `https://en.wikipedia.org/wiki/Core_dump`
[14] `https://en.wikipedia.org/wiki/Illumos`
[15] `https://illumos.org/man/1m/savecore`

to configure save compression and is able to estimate the size of a dump on a running system[16].

3.4 Backtrace.io

"Backtrace is a company that is aiming [to improve] the post-mortem debugging process." [17] Unlike the rest of this paper, Backtrace is not an operating system's dump process or its features, but a tool for analyzing cores once they are generated.

Backtrace supports several languages for userspace core dumps, including C, C++, Go, Python. Most importantly, Backtrace supports FreeBSD kernel core dumps. This section will focus on FreeBSD kernel core dump support.

Backtrace does not replace the FreeBSD core dump procedure, but is a service that collects core dumps and helps the operator traige and fix the bugs that cause those cores to be dumped.

Backtrace is a system made up of several parts: `coresnapd`, a snapshot generator; a set of analysis modules for automated debugging; `coroner`, an object store; a web interface and `hydra` its terminal counterpart [18].

After a sucessful `savecore(8)`, `coresnapd` and a set of companion scripts create a "snapshot" of any cores generated and send it back to `coroner` [19]. A snapshot contains a stack-trace across all threads, active regions of memory, requested global variables, environment information like virtual memory and CPU statistics, custom metadata such as datacenter, and annotations created by the analysis modules such as automated checking for a double `free()` of a pointer [18]. This results in a self-contained package that is smaller than a minidump and can be analyzed on a machine with an environment differing from the machine that created the original core [20]. Once collected, Backtrace's web interface can be used to categorize and triage different faults by any metadata or by panic string, for example. After triage, the web interface or `hydra` can be used to analyze snapshots [21].

Backtrace has also sponsored work on FreeBSD itself, by improving `kvm(3)`'s libkvm physical address lookup time from a linear time lookup to a constant time lookup. This provides gains in runtime complexity and space complexity of dealing with cores via `crashinfo(8)` or `kgdb(1)` especially for those systems with large amounts of RAM. [22]

[16] `https://illumos.org/man/1m/dumpadm`

[17] `https://backtrace.io/blog/supporting-freebsd-backtrace-and-bsd-now/`

[18] `https://documentation.backtrace.io/overview/`

[19] `https://documentation.backtrace.io/coresnap_integration/`

[20] `https://backtrace.io/blog/whats-a-coredump/`

[21] `https://documentation.backtrace.io/hydra/`

[22] `https://svnweb.freebsd.org/base?view=revision&revision=302976`

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

4 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 benefits of modularized core dump code.

4.1 `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 from Isilon that can be applied with some difficulty to versions of FreeBSD after 11.0. Network dumps have not yet made it into upstream FreeBSD.

4.2 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 6:1 to 14:1 compression ratio for core dumps with a slight penalty in the time required to write the dump initially[23]. 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.

4.3 `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.

Illumos is also capable of performing an online dump size estimation using `dumpadm(1M)`'s `-e` option which estimates the size of the dump taking in account options like compression [16].

`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.

4.4 Modularizing Dump Code

Currently if one would like to implement features or fixes in the core dump code one would need to recompile their kernel and reboot. This is highly undesireable when an operator wants to upgrade or fix their production systems. Refactoring the dump code into loadable kernel modules (LKM) would yield two major benefits for operators: easier development of fixes and features and a smaller kernel for embedded systems.

There is a proof of concept modularization of the dump code working on FreeBSD 11.0p1[24]. This code has not yet made it into upstream FreeBSD.

4.5 Dump to swap on zvol

Many users of FreeBSD use ZFS extensively. Though FreeBSD supports most ZFS features it currently is not recommended to use swap on a zvol as a dump device. However Illumos distributions support this out of the box and it is often the default [16].

This would be incredibly useful for users of ZFS in enterprise settings because ZFS datasets and zvols can be created, destroyed, and modified online, while modifying standard swap partitions is not possible without taking a machine offline and may not be trivial without re-imaging a machine.

4.6 Live Dump

The ability to take a core dump on an online system can be useful when a machine is otherwise hung and a the crash or panic would be difficult if not impossible to reproduce. Illumos can force a crash dump on an online system by issuing the `savecore -L` command.

This feature is not a replacement for normal crash dumps because the system is not halted during the dump which leads to an inconsistent state stored in the core dump. However, this adds another tool for enterprise FreeBSD users that must avoid taking machines offline as much as possible.

[24] `https://people.freebsd.org/~rgrimes/`

5 Acknowledgments

The author would like to thank Michael Dexter, for his initial prompting to write this paper and his help debugging the original issues that led to our current combined knowledge of core dumps, Rodney W. Grimes, for historical knowledge and help reading code from PDP-11 assembly to modern C, and Allan Jude, Daniel Nowacki and Chris Findeisen for finding and correct the many, many spelling, grammar and syntax issues in earlier versions of this paper.

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

6 Appendix

6.1 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.

6.2 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.

6.2.1 6th Edition UNIX

In 6th Edition UNIX `crash(8)` shows 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.

6.2.2 7th Edition UNIX

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

6.2.3 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.

6.3 Core Dumps in BSD

6.3.1 1BSD & 2BSD

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

6.3.2 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!".

6.3.3 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".

6.3.4 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

6.3.5 4.2BSD

- no changes.

6.3.6 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

6.3.7 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.

6.3.8 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

6.4 Core Dumps in FreeBSD

6.4.1 FreeBSD 1.0

- i386 support from 386BSD-0.1-patchkit

6.4.2 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.

6.4.3 FreeBSD 3.0.0

- SMP support
- alpha support

6.4.4 FreeBSD 4.0.0

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

6.4.5 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

6.4.6 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.

6.4.7 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

6.4.8 FreeBSD 8.0.0

- PowerPC support added.
- mips support added.

6.4.9 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

6.4.10 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_scheduler_on_panic`

6.4.11 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

6.4.12 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.