

Debugging Kernel Problems

by Greg Lehey

Edition for BSDCan 2005

Ottawa, 12 May 2005

Debugging Kernel Problems

by Greg Lehey (grog@FreeBSD.org)

Copyright © 1995-2005 Greg Lehey

Preface

Debugging kernel problems is a black art. Not many people do it, and documentation is rare, inaccurate and incomplete. This document is no exception: faced with the choice of accuracy and completeness, I chose to attempt the latter. As usual, time was the limiting factor, and this draft is still in beta status. This is a typical situation for the whole topic of kernel debugging: building debug tools and documentation is expensive, and the people who write them are also the people who use them, so there's a tendency to build as much of the tool as necessary to do the job at hand. If the tool is well-written, it will be reusable by the next person who looks at a particular area; if not, it might fall into disuse. Consider this book a starting point for your own development of debugging tools, and remember: more than anywhere else, this is an area with "some assembly required".

1

Introduction

Operating systems fail. All operating systems contain bugs, and they will sometimes cause the system to behave incorrectly. BSD kernels are no exception. Compared to most other operating systems, both free and commercial, BSD kernels offer a large number of debugging tools. This tutorial examines the options available both to the experienced end user and also to the developer.

This tutorial bases on the FreeBSD kernel, but the differences in other BSDs are small. We'll look at the following topics:

- How and why kernels fail.
- Understanding log files: *dmesg* and the files in */var/log*, notably */var/log/messages*.
- Userland tools for debugging a running system.
- Building a kernel with debugging support: the options.
- Using a serial console.
- Preparing for dumps: *dumpon*, *savecore*.
- The assembler-level view of a C program.
- Preliminary dump analysis.
- Reading code.
- Introduction to the kernel source tree.
- Analysing panic dumps with *gdb*.
- On-line kernel debuggers: *ddb*, remote serial *gdb*.
- Debugging a running system with *ddb*.
- Debugging a running system with *gdb*.
- Debug options in the kernel: *INVARIANTS* and friends.

- Debug options in the kernel: WITNESS.
- Code-based assistance: KTR.

How and why kernels fail

Good kernels should not fail. They must protect themselves against a number of external influences, including hardware failure, both deliberately and accidentally badly written user programs, and kernel programming errors. In some cases, of course, there is no way a kernel can recover, for example if the only processor fails. On the other hand, a good kernel should be able to protect itself from badly written user programs.

A kernel can fail in a number of ways:

- It can stop reacting to the outside world. This is called a *hang*.
- It can destroy itself (overwriting code). It's almost impossible to distinguish this state from a hang unless you have tools which can examine the machine state independently of the kernel.
- It can detect an inconsistency, report it and stop. In UNIX terminology, this is a *panic*.
- It can continue running incorrectly. For example, it might corrupt data on disk or breach network protocols.

By far the easiest kind of failure to diagnose is a panic. There are two basic types:

- Failed consistency checks result in a specific panic:

```
panic: Free vnode isn't
```

- Exception conditions result in a less specific panic:

```
panic: Page fault in kernel mode
```

The other cases can be very difficult to catch at the right moment.

2

Userland programs

dmesg

In normal operation, a kernel will sometimes write messages to the outside world via the “console”, `/dev/console`. Internally it writes via a circular buffer called `msgbuf`. The `dmesg` program can show the current contents of `msgbuf`. The most important use is at startup time for diagnosing configuration problems:

```
# dmesg
Copyright (c) 1992-2002 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
    The Regents of the University of California. All rights reserved.
FreeBSD 4.5-PRERELEASE #3: Sat Jan  5 13:25:02 CST 2002
  grog@echunga.lemis.com: /src/FreeBSD/4-STABLE-ECHUNGA/src/sys/compile/ECHUNGA
Timecounter "i8254" frequency 1193182 Hz
Timecounter "TSC" frequency 751708714 Hz
CPU: AMD Athlon(tm) Processor (751.71-MHz 686-class CPU)
  Origin = "AuthenticAMD" Id = 0x621 Stepping = 1
  Features=0x183f9ff<FPU,VME,DE,PSE,TSC,MSR,PAE,MCE,CX8,SEP,MTRR,PGE,MCA,CMOV,PAT,PSE3
6,MMX,FXSR>
  AMD Features=0xc0400000<AMIE,DSP,3DNow!>
...
pci0: <unknown card> (vendor=0x1039, dev=0x0009) at 1.1
...
cdl at ahc0 bus 0 target 1 lun 0
cdl: <TEAC CD-ROM CD-532S 1.0A> Removable CD-ROM SCSI-2 device
cdl: 20.000MB/s transfers (20.000MHz, offset 15)
cdl: Attempt to query device size failed: NOT READY, Medium not present
...
WARNING: / was not properly unmounted
```

Much of this information is informative, but occasionally you get messages indicating some problem. The last line in the previous example shows that the system did not shut down properly: either it crashed, or the power failed. During normal operation you might see messages like the following:

```
siol: 1 more silo overflow (total 1607)
siol: 1 more silo overflow (total 1608)
```

```
nfsd send error 64
...
nfs server wantadilla:/src: not responding
nfs server wantadilla::: not responding
nfs server wantadilla:/src: is alive again
nfs server wantadilla::: is alive again
arp info overwritten for 192.109.197.82 by 00:00:21:ca:6e:f1
```

In the course of time, the message buffer wraps around and the old contents are lost. For this reason, FreeBSD and NetBSD print the *dmesg* contents after boot to the file */var/run/dmesg.boot* for later reference. In addition, the output is piped to *syslogd*, the system log daemon, which by default writes it to */var/log/messages*.

During kernel debugging you can print *msgbuf*. For FreeBSD, enter:

```
(gdb) printf "%s", (char *)msgbuf->msg_ptr
```

For NetBSD or OpenBSD, enter:

```
(gdb) printf "%s", (char *) msgbuf->msg_bufc
```

Log files

BSD systems keep track of significant events in *log files*. They can be of great use for debugging. Most of them are kept in */var/log*, though this is not a requirement. Many of them are maintained by *syslogd*, but there is no requirement for a special program. The only requirement is to avoid having two programs maintaining the same file.

syslogd

syslogd is a standard daemon which maintains a number of the files in */var/log*. You should always run *syslogd* unless you have a very good reason not to.

Processes normally write to *syslogd* with the library function *syslog*:

```
#include <syslog.h>
#include <stdarg.h>

void syslog (int priority, const char *message, ...);
```

syslog is used in a similar manner to *printf*; only the first parameter is different. Although it's called *priority* in the man page, it's divided into two parts:

- The *level* field describes how serious the message is. It ranges from *LOG_DEBUG* (information normally suppressed and only produced for debug purposes) to *LOG_EMERG* ("machine about to self-destruct").
- The *facility* field describes what part of the system generated the message.

The *priority* field can be represented in text form as *facility.level*. For example, error messages from the mail subsystem are called *mail.err*.

In FreeBSD, as the result of security concerns, *syslogd* is started with the `-s` flag by default. This stops *syslogd* from accepting remote messages. If you specify the `-ss` flag, as suggested in the comment, you will also not be able to log to remote systems. Depending on your configuration, it's worth changing this default. For example, you might want all systems in *example.org* to log to *gw*. That way you get one set of log files for the entire network.

/etc/syslog.conf

syslogd reads the file */etc/syslog.conf*, which specifies where to log messages based on their message priority. Here's a slightly modified example:

```
# $FreeBSD: src/etc/syslog.conf,v 1.13 2000/02/08 21:57:28 rwatson Exp $
#
#       Spaces are NOT valid field separators in this file.
#       Consult the syslog.conf(5) manpage.
*. *                @echunga                log everything to system echunga
*.err;kern.debug;auth.notice;mail.crit /dev/console    log specified messages to console
*.notice;kern.debug;lpr.info;mail.crit /var/log/messages log messages to file
security.*          /var/log/security    specific subsystems
mail.info           /var/log/maillog      get their own files
lpr.info            /var/log/lpd-errs
cron.*             /var/log/cron
*.err              root                  inform logged-in root user of errors
*.notice;news.err  root
*.alert            root
*.emerg            *
# uncomment this to enable logging of all log messages to /var/log/all.log
#*. *              /var/log/all.log
# uncomment this to enable logging to a remote loghost named loghost
#*. *              @loghost
# uncomment these if you're running inn
# news.crit        /var/log/news/news.crit
# news.err         /var/log/news/news.err
# news.notice      /var/log/news/news.notice
!startslip         all messages from startslip
*. *              /var/log/slip.log
!ppp               all messages from ppp
*. *              /var/log/ppp.log
```

Note that *syslogd* does not create the files if they don't exist.

Userland programs

A number of userland programs are useful for divining what's going on in the kernel:

- *ps* shows selected fields from the process structures. With an understanding of the structures, it can give a good idea of what's going on.
- *top* is like a repetitive *ps*: it shows the most active processes at regular intervals.
- *vmstat* shows a number of parameters, including virtual memory. It can also be set up to run at regular intervals.
- *iostat* is similar to *vmstat*, and it duplicates some fields, but it concentrates more on I/O activity.
- *netstat* show network information. It can also be set up to show transfer rates for specific interfaces.

- *systat* is a curses-based program which displays a large number of parameters, including most of the parameters displayed by *vmstat*, *iostat* and *netstat*.
- *ktrace* traces system calls and their return values for a specific process. It's like a *GIGO*: you see what goes in and what comes out again.

ps

ps displays various process state. Most people use it for fields like PID, command and CPU time usage, but it can also show a number of other more subtle items of information:

- When a process is sleeping (which is the normal case), WCHAN displays a string indicating where it is sleeping. With the aid of the kernel code, you can then get a reasonably good idea what the process is doing. FreeBSD calls this field MWCHAN, since it can also show the name of a mutex on which the process is blocked.
- STAT shows current process state. There are a number of these, and they change from time to time, and they differ between the versions of BSD. They're defined in the man page.
- flags (F) show process flags. Like the state information they change from time to time and differ between the versions of BSD. They're also defined in the man page.
- There are a large number of optional fields which can also be specified with the *-O* option.

Here are some example processes, taken from a FreeBSD release 5 system:

```
$ ps lax
  UID   PID   PPID  CPU  PRI  NI   VSZ   RSS  MWCHAN  STAT  TT        TIME  COMMAND
    0     0     0    0   -16   0     0    12  sched  DLs    ??        0:15.62  (swapper)
```

The swapper, sleeping on sched. It's in a short-term wait (D status), it has pages locked in core (L) status, and it's a session leader (s status), though this isn't particularly relevant here. The name in parentheses suggests that it's swapped out, but it should have a W status for that.

```
  UID   PID   PPID  CPU  PRI  NI   VSZ   RSS  MWCHAN  STAT  TT        TIME  COMMAND
 1004     0  60226    0  -84   0     0     0  -      ZW    ??        0:00.00  (galeon-bin)
```

This process is a zombie (Z status), and what's left of it is swapped out (W status, name in parentheses).

```
  UID   PID   PPID  CPU  PRI  NI   VSZ   RSS  MWCHAN  STAT  TT        TIME  COMMAND
    0     1     0    0    8   0    708    84  wait   ILs    ??        0:14.58  /sbin/init --
```

init is waiting for longer than 20 seconds (I state). Like *swapper*, it has pages locked in core and is a session leader. A number of other system processes have similar flags.

```
  UID   PID   PPID  CPU  PRI  NI   VSZ   RSS  MWCHAN  STAT  TT        TIME  COMMAND
    0     7     0    0  171   0     0    12  -      RL    ??        80:46.00  (pagezero)
```

pagezero is waiting to run (R), and also no wait channel.

```
  UID   PID   PPID  CPU  PRI  NI   VSZ   RSS  MWCHAN  STAT  TT        TIME  COMMAND
    0     8     0    2    4   0     0    12  sbwait  DL    ??        1:44.51  (bufdaemon)
```

`sbwait` is the name of wait channel here, but it's also the name of the function that is waiting:

```
/*
 * Wait for data to arrive at/drain from a socket buffer.
 */
int
sbwait(sb)
    struct sockbuf *sb;
{
    sb->sb_flags |= SB_WAIT;
    return (tsleep(&sb->sb_cc,
        (sb->sb_flags & SB_NOINTR) ? PSOCK : PSOCK | PCATCH, "sbwait",
        sb->sb_timeo));
}
```

The name `sbwait` in the *ps* output comes from the convoluted `tsleep` call at the end of the function, not from the name of the function.

| UID | PID | PPID | CPU | PRI | NI | VSZ | RSS | MWCHAN | STAT | TT | TIME | COMMAND |
|-----|-----|------|-----|-----|----|-----|-----|--------|------|----|-------------|---------|
| 0 | 11 | 0 | 150 | -16 | 0 | 0 | 12 | - | RL | ?? | 52617:10.66 | (idle) |

The idle process (currently only present in FreeBSD release 5) uses up the remaining CPU time on the system. That explains the high CPU usage. The priority is bogus: `idle` only gets to run when nothing else is runnable.

| UID | PID | PPID | CPU | PRI | NI | VSZ | RSS | MWCHAN | STAT | TT | TIME | COMMAND |
|-----|-----|------|-----|-----|----|-----|-----|--------|------|----|----------|-----------------------|
| 0 | 12 | 0 | 0 | -44 | 0 | 0 | 12 | - | WL | ?? | 39:11.32 | (swil: net) |
| 0 | 13 | 0 | 0 | -48 | 0 | 0 | 12 | - | WL | ?? | 43:42.81 | (swi6: tty:sio clock) |

These two processes are examples of software interrupt threads. Again, they only exist in FreeBSD release 5.

| UID | PID | PPID | CPU | PRI | NI | VSZ | RSS | MWCHAN | STAT | TT | TIME | COMMAND |
|-----|-----|------|-----|-----|----|-----|-----|--------|------|----|-----------|---------------|
| 0 | 20 | 0 | 0 | -64 | 0 | 0 | 12 | - | WL | ?? | 0:00.00 | (irq11: ahc0) |
| 0 | 21 | 0 | 34 | -68 | 0 | 0 | 12 | Giant | LL | ?? | 116:10.44 | (irq12: rl0) |

These are hardware interrupts. `irq12` is waiting on the Giant mutex.

top

top is like a repetitive *ps*. It shows similar information at regular intervals. By default, the busiest processes are listed at the top of the display, and the number of processes can be limited. It also shows additional summary information about CPU and memory usage:

```
load averages:  1.42,  1.44,  1.41                                16:50:23
41 processes:  2 running, 38 idle, 1 zombie
CPU states: 81.4% user,  0.0% nice, 16.7% system,  2.0% interrupt,  0.0% idle
Memory: Real: 22M/48M act/tot  Free: 12M  Swap: 7836K/194M used/tot

  PID USERNAME PRI NICE  SIZE  RES STATE WAIT      TIME    CPU COMMAND
  336  build    64    0   12M 244K run  -       0:25 69.82% cc1
 1407  grog     28    0   176K 328K run  -       0:25  1.03% top
14928  grog      2    0 1688K 204K sleep select  0:17  0.54% xterm
 9452  grog     18    4   620K 280K idle  pause 376:06  0.00% xearth
18876  root      2    0    28K  72K sleep select 292:22  0.00% screenblank
  399  grog      2    4   636K   0K idle  select 126:37  0.00% <fvwm2>
```

```

7280 grog      2      0 9872K 124K idle  select 102:42  0.00% Xsun
8949 root      2      0 896K 104K sleep select  37:48  0.00% sendmail
10503 root     18      0 692K 248K sleep pause  24:39  0.00% ntpd

```

Here again the system is 100% busy. This machine (*flame.lemis.com*) is a SPARCstation 5 running OpenBSD and part of the Samba build farm. The CPU usage shows that over 80% of the time is spent in user mode, and less than 20% in system and interrupt mode combined. Most of the time here is being used by the C compiler, *cc1*. The CPU usage percentages are calculated dynamically and usually don't quite add up.

The distinction between system and interrupt mode is the distinction between process and non-process activities. This is a relatively easy thing to measure, but in traditional BSDs it's not clear how much of this time is due to I/O and how much due to other interrupts.

There's a big difference in the reactivity of a system with high system load and a system with high interrupt load: load-balancing doesn't work for interrupts, so a system with high interrupt times reacts very sluggishly.

Sometimes things look different. Here's a FreeBSD 5-CURRENT test system:

```

last pid: 79931;  load averages:  2.16,  2.35,  2.21          up 0+01:25:07  18:07:46
75 processes:  4 running, 51 sleeping, 20 waiting
CPU states: 18.5% user,  0.0% nice, 81.5% system,  0.0% interrupt,  0.0% idle
Mem: 17M Active, 374M Inact, 69M Wired, 22M Cache, 60M Buf, 16M Free
Swap: 512M Total, 512M Free

  PID USERNAME  PRI  NICE   SIZE   RES STATE   TIME  WCPU   CPU COMMAND
    10 root       -16    0     0K    12K RUN    18:11  1.07%  1.07% idle
  79828 root       125    0   864K   756K select  0:00  3.75%  0.83% make
     6 root        20    0     0K    12K syncer  0:35  0.20%  0.20% syncer
    19 root       -68 -187     0K    12K WAIT   0:12  0.00%  0.00% irq9: r10
    12 root       -48 -167     0K    12K WAIT   0:08  0.00%  0.00% swi6: tty:sio clock
   303 root        96    0  1052K   688K select  0:05  0.00%  0.00% rlogind

```

This example was taken during a kernel build. Again the CPU is 100% busy. Strangely, though, the busiest process is the idle process, with only a little over 1% of the total load.

What's missing here? The processes that start and finish in the interval between successive displays. One way to check this is to look at the `last pid` field at the top left (this field is not present in the NetBSD and OpenBSD versions): if it increments rapidly, it's probable that these processes are using the CPU time.

There's another thing to note here: the CPU time is spread between user time (18.5%) and system time (81.5%). That's not a typical situation. This build was done on a test version of FreeBSD 5-CURRENT, which includes a lot of debugging code, notably the WITNESS code which will be discussed later. It would be very difficult to find this with *ps*.

Load average

It's worth looking at the load averages mentioned on the first line. These values are printed by a number of other commands, notably *w* and *uptime*. The load average is the length of the run queue averaged over three intervals: 1, 5 and 15 minutes. The run queue contains jobs ready to be scheduled, and is thus an indication of how busy the system is.

vmstat

vmstat was originally intended to show virtual memory statistics, but current versions show a number of other parameters as well. It can take a numeric argument representing the number of seconds between samples. In this case, the first line shows the average values since boot time, so it is usually noticeably different from the remaining lines.

```
$ vmstat 1
procs      memory      page      disks      faults      cpu
r  b  w      avm      fre  flt  re  pi  po  fr  sr  s0  c0      in  sy  cs  us  sy  id
1  1  0    17384    23184    200   0   0   0   0   0   9   0    236   222   35  22   7  70
2  1  0    17420    23148    2353   0   0   0   0   0   0  24   0    271  1471   94  36  45  20
1  1  0    18488    22292    2654   0   0   0   0   0   0  20   0    261  1592  102  35  51  14
```

The base form of this command is essentially identical in all BSDs. The parameters are:

- The first section (procs) shows the number of processes in different states. *r* shows the number of processes on the run queue (effectively a snapshot of the load average). *b* counts processes blocked on resources such as I/O or memory. *w* counts processes that are runnable but is swapped out. This almost never happens any more.
- The next subsection describes memory availability. *avm* is the number of “active” virtual memory pages, and *fre* is the number of free pages.
- Next come paging activity. *re* is the number of page reclaims, *pi* the number of pages paged in from disk, *po* the number of pages paged out to disk, *fr* the number of pages freed per second, and *sr* the number of pages scanned by the memory manager per second.

iostat

- Shows statistics about I/O activity.
- Can be repeated to show current activity.
- Can specify which devices or device categories to observe.

Example (OpenBSD SPARC)

```
          tty          sd0          rd0          rd1          cpu
tin tout KB/t t/s MB/s KB/t t/s MB/s KB/t t/s MB/s us ni sy in id
0      0  7.77  9 0.07  0.00  0 0.00  0.00  0 0.00  0.00  19 0  6  1 74
0    222 56.00  1 0.05  0.00  0 0.00  0.00  0 0.00  0.00  69 0  29  2  0
0      75  0.00  0 0.00  0.00  0 0.00  0.00  0 0.00  0.00  81 0  19  0  0
0      76 32.00  1 0.03  0.00  0 0.00  0.00  0 0.00  0.00  84 0  16  0  0
0      74  0.00  0 0.00  0.00  0 0.00  0.00  0 0.00  0.00  90 0  7  3  0
0      74  0.00  0 0.00  0.00  0 0.00  0.00  0 0.00  0.00  95 0  5  0  0
0      74  5.30  20 0.10  0.00  0 0.00  0.00  0 0.00  0.00  40 0  31  0 29
0      73  6.40  51 0.32  0.00  0 0.00  0.00  0 0.00  0.00  12 0  10  3 75
0      75  5.55  49 0.27  0.00  0 0.00  0.00  0 0.00  0.00  24 0  12  3 61
0      73  4.91  54 0.26  0.00  0 0.00  0.00  0 0.00  0.00  21 0  9  1 69
0      75  6.91  54 0.36  0.00  0 0.00  0.00  0 0.00  0.00  39 0  7  3 51
0      72  9.80  49 0.46  0.00  0 0.00  0.00  0 0.00  0.00  31 0  6  4 59
0      76 17.94  36 0.63  0.00  0 0.00  0.00  0 0.00  0.00  34 0  12  0 54
0      75 19.20  5 0.09  0.00  0 0.00  0.00  0 0.00  0.00  93 0  5  1  1
0      74 37.33  3 0.11  0.00  0 0.00  0.00  0 0.00  0.00  93 0  6  1  0
0      75 56.00  1 0.06  0.00  0 0.00  0.00  0 0.00  0.00  82 0  17  1  0
0      73  0.00  0 0.00  0.00  0 0.00  0.00  0 0.00  0.00  83 0  16  1  0
```

systat

- Shows a number of different parameters in graphical form.
- Includes *iostat*, *netstat* and *vmstat*.
- Ugly display.

systat example

```

Load Average  /0  /1  /2  /3  /4  /5  /6  /7  /8  /9  /10
               ||
cpu  user | /0  /10 /20 /30 /40 /50 /60 /70 /80 /90 /100
      nice|XXXXXXXXXXXXXXXXXXXX
      system|XXXXX
interrupt|
idle      |XXXXXXXXXXXXXXXXXXXX

ad0  MB/s /0  /10 /20 /30 /40 /50 /60 /70 /80 /90 /100
      tps|XXXXXXXXXXXXXXXXXXXX

```

systat vmstat, FreeBSD

```

24 users      Load  0.85  0.25  0.15                               Sun Jan 20 14:40

Mem:KB      REAL                VIRTUAL                VN PAGER  SWAP PAGER
      Tot  Share      Tot  Share      Free      in  out      in  out
Act  150180  3536      220116  10096      10404
All  252828  4808      3565340  15372      count
                                   pages

Proc:r  p  d  s  w      Csw  Trp  Sys  Int  Sof  Flt
      2    1 24      147   14  63262294  26   6   56060
                                   wire  1 ata0 irq14
                                   act   atal irq15
                                   24140  inact  ahc0 irq11
                                   9748   cache  27 mux irq10
                                   656    free   4 atkbd0 irq
                                   daefr   psm0 irq12
                                   prcfr   77 siol irq3
                                   react   ppc0 irq7
                                   pdwak   99 clk irq0
                                   pdpgs  128 rtc irq8
                                   intrn 61959 lpt0 irq7
                                   35712  buf
                                   27 dirtybuf
                                   17462  desirednodes
                                   22916  numvnodes
                                   17020  freevnodes

```

systat vmstat, NetBSD

```

1 user      Load  2.74  1.91  1.60                               Thu Jan 17 14:31:09

memory totals (in KB)          PAGING  SWAPPING  Interrupts

```

```

Active    real    virtual    free      ops      in  out      in  out      132 total
All       9868     14100     6364      pages    1
          21140    25372   658588
Proc:r    d  s  w      Csw  Trp  Sys  Int  Sof  Flt      forks
          2  1  5      40   27  193  133  20   8      fkppw
          95.9% Sy  1.4% Us  0.0% Ni  0.0% In  2.7% Id      fksvm
          |  |  |  |  |  |  |  |  |  |      pwait
          |  |  |  |  |  |  |  |  |  |      6 relck
          |  |  |  |  |  |  |  |  |  |      6 rllck
          |  |  |  |  |  |  |  |  |  |      noram
          |  |  |  |  |  |  |  |  |  |      ndcpy
          |  |  |  |  |  |  |  |  |  |      fltcp
          |  |  |  |  |  |  |  |  |  |      1 zfod
          |  |  |  |  |  |  |  |  |  |      cow
          |  |  |  |  |  |  |  |  |  |      64 fmin
          |  |  |  |  |  |  |  |  |  |      85 ftarg
          |  |  |  |  |  |  |  |  |  |      1372 itarg
          |  |  |  |  |  |  |  |  |  |      941 wired
          |  |  |  |  |  |  |  |  |  |      pdfre
          |  |  |  |  |  |  |  |  |  |      pdscn

Namei      Sys-cache      Proc-cache
Calls      hits  %      hits  %
1043      806  77      34  3

Discs  fd0  sd0  md0
seeks
xfers      14
Kbyte      164
%busy      21.2

```

systat vmstat, OpenBSD

```

3 users      Load  1.19  1.52  1.81      Thu Jan 17 14:31:48 2002

Mem:KB  REAL      VIRTUAL      PAGING      SWAPPING      Interrupts
      Tot Share    Tot Share    Free      in  out      in  out      227 total
Act  3348  1068  12940  6704  27016 count    2      5 lev1
All  35232 11888 358812 148796      pages    17 lev4
          5 lev6
Proc:r    p  d  s  w      Csw  Trp  Sys  Int  Sof  Flt      17 cow      100 clock
          2  5      29  206  184  227      374      3 objlk      lev12
          9.3% Sys  85.5% User  0.0% Nice  4.4% Idle      2 objht      100 prof
          |  |  |  |  |  |  |  |  |  |      62 zfod
          |  |  |  |  |  |  |  |  |  |      385 nzfod
          |  |  |  |  |  |  |  |  |  |      16.14 %zfod
          |  |  |  |  |  |  |  |  |  |      kern
          |  |  |  |  |  |  |  |  |  |      5408 wire
          |  |  |  |  |  |  |  |  |  |      18312 act
          |  |  |  |  |  |  |  |  |  |      11220 inact
          |  |  |  |  |  |  |  |  |  |      27016 free
          |  |  |  |  |  |  |  |  |  |      daefr
          |  |  |  |  |  |  |  |  |  |      372 prcfr
          |  |  |  |  |  |  |  |  |  |      46 react
          |  |  |  |  |  |  |  |  |  |      scan
          |  |  |  |  |  |  |  |  |  |      hdrev
          |  |  |  |  |  |  |  |  |  |      intrn

Namei      Sys-cache      Proc-cache
Calls      hits  %      hits  %
212      203  96      3  1

Discs  sd0  rd0  rd1
seeks  411
xfers  411
Kbyte  33
sec    0.1

```

ktrace

- Traces at system call interface.
- Doesn't require source code.
- Shows a limited amount of information.
- Can be useful to find which files are being opened.
- You collect a dump file with *ktrace*, and dump in with *kdump*.

ktrace example

```

71602 sh      NAMI   "/bin/url_handler.sh"
71602 sh      RET    stat -1 errno 2 No such file or directory
71602 sh      CALL   stat(0x80ec108,0xbfbff0b0)
71602 sh      NAMI   "/sbin/url_handler.sh"
71602 sh      RET    stat -1 errno 2 No such file or directory
71602 sh      CALL   stat(0x80ec108,0xbfbff0b0)
71602 sh      NAMI   "/usr/local/bin/url_handler.sh"
71602 sh      RET    stat -1 errno 2 No such file or directory
71602 sh      CALL   stat(0x80ec108,0xbfbff0b0)
71602 sh      NAMI   "/etc/url_handler.sh"
71602 sh      RET    stat -1 errno 2 No such file or directory
71602 sh      CALL   stat(0x80ec108,0xbfbff0b0)
71602 sh      NAMI   "/usr/X11R6/bin/url_handler.sh"
71602 sh      RET    stat -1 errno 2 No such file or directory
71602 sh      CALL   stat(0x80ec108,0xbfbff0b0)
71602 sh      NAMI   "/usr/monkey/url_handler.sh"
71602 sh      RET    stat -1 errno 2 No such file or directory
71602 sh      CALL   stat(0x80ec108,0xbfbff0b0)
71602 sh      NAMI   "/usr/local/sbin/url_handler.sh"
71602 sh      RET    stat -1 errno 2 No such file or directory
71602 sh      CALL   break(0x80f3000)
71602 sh      RET    break 0
71602 sh      CALL   write(0x2,0x80f2000,0x1a)
71602 sh      GIO    fd 2 wrote 26 bytes
      "url_handler.sh: not found
      "
71602 sh      RET    write 26/0x1a
71602 sh      CALL   exit(0x7f)

```


3

Hardware data structures

Stack frames

Most modern machines have a stack-oriented architecture, though the support is rather rudimentary in some cases. Everybody knows what a stack is, but here we'll use a more restrictive definition: a *stack* is a linear list of storage elements, each relating to a particular function invocation. These are called *stack frames*. Each stack frame contains

- The parameters with which the function was invoked.
- The address to which to return when the function is complete.
- Saved register contents.
- Variables local to the function.
- The address of the previous stack frame.

With the exception of the return address, any of these fields may be omitted.¹ It's possible to implement a stack in software as a linked list of elements, but most machines nowadays have significant hardware support and use a reserved area for the stack. Such stack implementations typically supply two hardware registers to address the stack:

- The *stack pointer* points to the last used word of the stack.
- The *frame pointer* points to somewhere in the middle of the stack frame.

1. Debuggers recognize stack frames by the frame pointer. If you don't save the frame pointer, it will still be pointing to the previous frame, so the debugger will report that you are in the previous function. This frequently happens in system call linkage functions, which typically do not save a stack linkage, or on the very first instruction of a function, before the linkage has been built. In addition, some optimizers remove the stack frame.

The resultant memory image looks like:

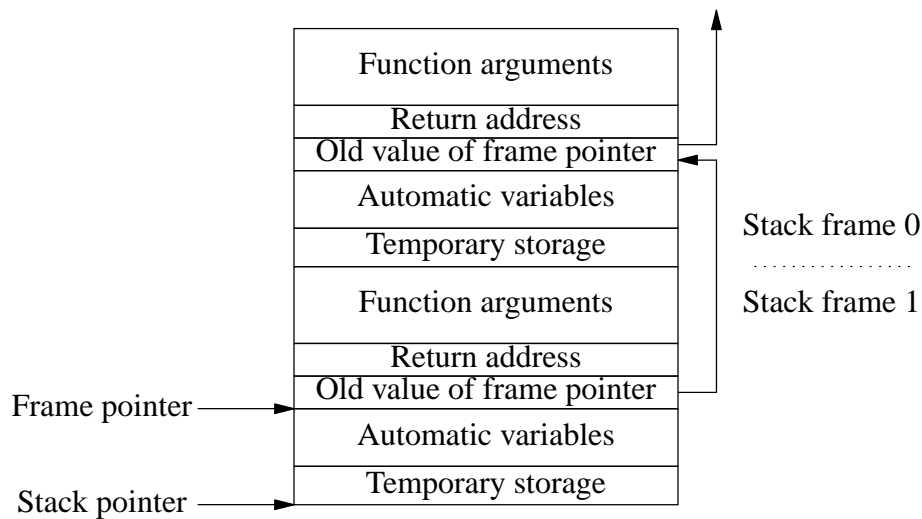


Figure 1: Function stack frame

The individual parts of the stack frames are built at various times. In the following sections, we'll use the Intel ia32 (i386) architecture as an example to see how the stack gets set up and freed. The ia32 architecture has the following registers, all 32 bits wide:

- The *Program Counter* is the traditional name for the register that points to the next instruction to be executed. Intel calls it the *Instruction Pointer* or *eip*. The *e* at the beginning of the names of most registers stands for *extended*. It's a reference to the older 8086 architecture, which has shorter registers with similar names: for example, on the 8086 this register is called *ip* and is 16 bits wide.
- The *Stack Pointer* is called *esp*.
- The *Frame Pointer* is called *ebp* (*Extended Base Pointer*), referring to the fact that it points to the stack base.
- The arithmetic and index registers are a mess on ia32. Their naming goes back to the 8 bit 8008 processor (1972). In those days, the only arithmetic register was the *Accumulator*. Nowadays some instructions can use other registers, but the name remains: *eax*, *Extended Accumulator Extended* (no joke: the first extension was from 8 to 16 bits, the second from 16 to 32).
- The other registers are *ebx*, *ecx* and *edx*. Each of them has some special function, but they can be used in many arithmetic instructions as well. *ecx* can hold a count for certain repeat instructions.
- The registers *esi* (*Extended Source Index*) and *edi* (*Extended Destination Index*) are purely index registers. Their original use was implicit in certain repeated instructions, where they are incremented automatically.

- The `eflags` register contains program status information.
- The *segment registers* contain information about memory segments. Their usage depends on the mode in which the processor is running.

Some registers can be subdivided: for example, the two halves of `eax` are called `ah` (high bits) and `al` (low bits).

Stack growth during function calls

Now that we have an initial stack, let's see how it grows and shrinks during a function call. We'll consider the following simple C program compiled on the i386 architecture:

```
foo (int a, int b)
{
    int c = a * b;
    int d = a / b;
    printf ("%d %d\n", c, d);
}

main (int argc, char *argv [])
{
    int x = 4;
    int y = 5;
    foo (y, x);
}
```

The assembler code for the calling sequence for `foo` in `main` is:

| | |
|-----------------------------|------------------------------|
| <code>pushl -4(%ebp)</code> | <i>value of x</i> |
| <code>pushl -8(%ebp)</code> | <i>value of y</i> |
| <code>call _foo</code> | <i>call the function</i> |
| <code>addl \$8,%esp</code> | <i>and remove parameters</i> |

Register `ebp` is the *base pointer*, which we call the *frame pointer*. `esp` is the *stack pointer*.

The `push` instructions decrement the stack pointer and then place the word values of `x` and `y` at the location to which the stack pointer now points.

The `call` instruction pushes the contents of the current instruction pointer (the address of the instruction following the `call` instruction) onto the stack, thus saving the return address, and loads the instruction pointer with the address of the function. We now have:

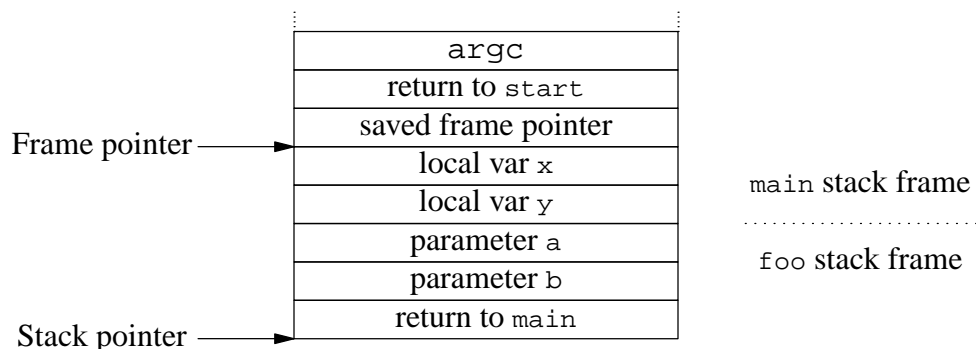


Figure 2: Stack frame after `call` instruction

The called function `foo` saves the frame pointer (in this architecture, the register is called *ebp*, for *extended base pointer*), and loads it with the current value of the stack pointer register *esp*.

```
_foo:    pushl %ebp           save ebp on stack
        movl %esp,%ebp      and load with current value of esp
```

At this point, the stack linkage is complete, and this is where most debuggers normally set a breakpoint when you request on to be placed at the entry to a function.

Next, `foo` creates local storage for `c` and `d`. They are each 4 bytes long, so it subtracts 8 from the *esp* register to make space for them. Finally, it saves the register *ebx*--the compiler has decided that it will need this register in this function.

```
        subl $8,%esp        create two words on stack
        pushl %ebx          and save ebx register
```

At this point, our stack is now complete

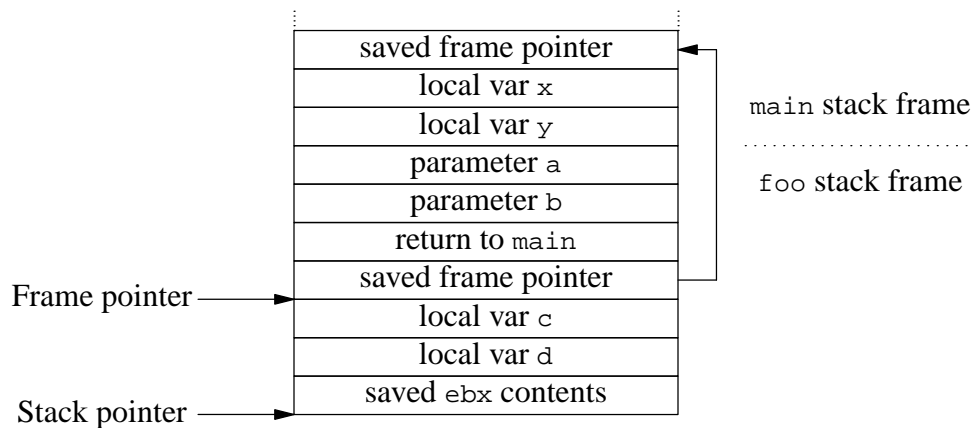


Figure 3: Complete stack frame after entering called function

The frame pointer isn't absolutely necessary: you can get by without it and refer to the stack pointer instead. The problem is that during the execution of the function, the compiler may save further temporary information on the stack, so it's difficult to keep track of the value of the stack pointer--that's why most architectures use a frame pointer, which *does* stay constant during the execution of the function. Some optimizers, including newer versions of *gcc*, give you the option of compiling without a stack frame. This makes debugging almost impossible.

On return from the function, the sequence is reversed:

```
        movl -12(%ebp),%ebx  and restore register ebx
        leave               reload ebp and esp
        ret                 and return
```

The first instruction reloads the saved register *ebx*, which could be stored anywhere in the stack. This instruction does not modify the stack.

The *leave* instruction loads the stack pointer *esp* from the frame pointer *ebp*, which effectively

discards the part stack below the saved *ebp* value. Then it loads *ebp* with the contents of the word to which it points, the saved *ebp*, effectively reversing the stack linkage. The stack now looks like it did on entry.

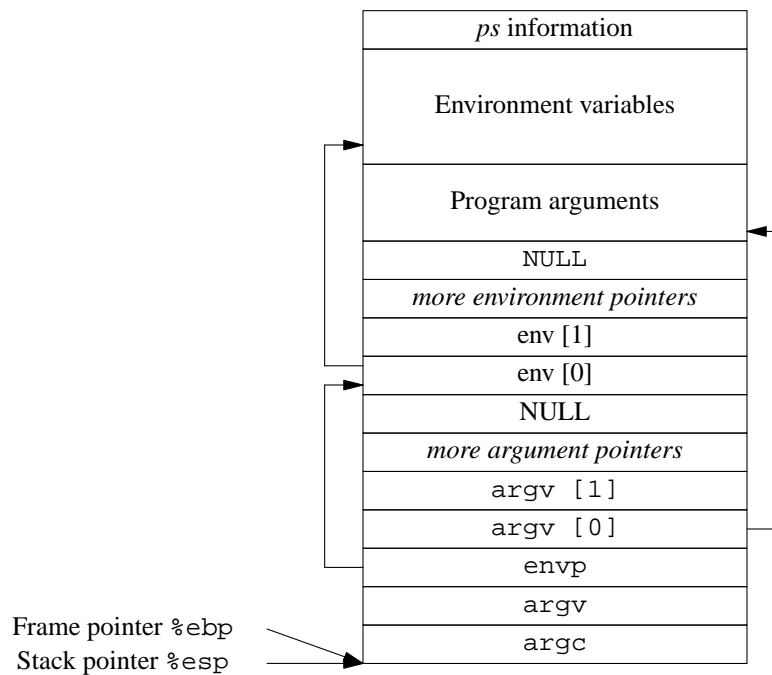
Next, the *ret* instruction pops the return address into the instruction pointer, causing the next instruction to be fetched from the address following the *call* instruction in the calling function.

The function parameters *x* and *y* are still on the stack, so the next instruction in the calling function removes them by adding to the stack pointer:

```
addl $8,%esp          and remove parameters
```

Stack frame at process start

A considerable amount of work on the stack occurs at process start, before the *main* function is called. Here's an example of what you might find on an i386 architecture at the point where you enter *main*:



Contrary to the generally accepted view, the prototype for *main* in all versions of UNIX, and also in Linux and other operating systems, is:

```
int main (int argc, char *argv [], char *env []);
```

System call stack frame

Individual processors have a number of different ways to perform a system call, but in general they're similar to a function call. In addition, though, the processor needs to manage the change of context from user to system on the call, and to save enough information to find its way back on return. Modern ELF-based ia32 systems use the `INTR` instruction (called `int` in the assembler) to perform the transition. The older *a.out* format used a form of the `CALL` instruction called `lcall` in the assembler. The entry point to the kernel ensures that the frames are the same.

The first part of the stack frame is built by the `INTR` instruction:

| | |
|----------------|--------|
| Stack segment | ss |
| Stack pointer | esp |
| Flags | eflags |
| Code segment | cs |
| Return address | eip |
| Error code | err |

Figure 4: Stack frame after `INTR` instruction

The kernel entry point for `INTR`-type system calls is `int0x80_syscall`. It saves some registers on the stack to make a standard exception trap frame and then calls `syscall`:

```
/*
 * Call gate entry for FreeBSD ELF and Linux/NetBSD syscall (int 0x80)
 *
 * Even though the name says 'int0x80', this is actually a TGT (trap gate)
 * rather than an IGT (interrupt gate). Thus interrupts are enabled on
 * entry just as they are for a normal syscall.
 */
    SUPERALIGN_TEXT
IDTVEC(int0x80_syscall)
    pushl    $2                /* sizeof "int 0x80" */
    subl     $4,%esp           /* skip over tf_trapno */
    pushal
    pushl    %ds
    pushl    %es
    pushl    %fs
    movl     $KDSEL,%eax       /* switch to kernel segments */
    movl     %eax,%ds
    movl     %eax,%es
    movl     $KPSEL,%eax
    movl     %eax,%fs
    FAKE_MCOUNT(13*4(%esp))
    call     syscall
    MEXITCOUNT
    jmp      doreti
```

At the end of this, the data on the stack is:

| | |
|---------------------------|--------|
| Stack segment | ss |
| Stack pointer | esp |
| Flags | eflags |
| Code segment | cs |
| Return address | eip |
| Error code | err |
| Trap number | trapno |
| Saved registers (pushal) | eax |
| | ecx |
| | edx |
| | ebx |
| | esp |
| | ebp |
| | esi |
| | edi |
| Data segment | ds |
| Extended segment | es |
| FS | fs |

Figure 5: Stack frame on entry to `syscall`

4

The GNU debugger

This chapter takes a look at the GNU debugger, *gdb*, as it is used in userland.

What debuggers do

gdb runs on UNIX and similar platforms. In UNIX, a debugger is a process that takes control of the execution of another process. Most versions of UNIX allow only one way for the debugger to take control: it must start the process that it debugs. Some versions, notably FreeBSD and SunOS 4, but not related systems like BSD/OS or Solaris 2, also allow the debugger to *attach* to a running process. *gdb* supports attaching on platforms which offer the facility.

Whichever debugger you use, there are a surprisingly small number of commands that you need:

- A *stack trace* command answers the question, “Where am I, and how did I get here?”, and is the most useful of all commands. It’s certainly the first thing you should do when examining a core dump or after getting a signal while debugging the program.
- *Displaying data* is the most obvious requirement: “what is the current value of the variable `bar`?”
- *Displaying register contents* is really the same thing as displaying program data. You’ll normally only look at registers if you’re debugging at the assembly code level, but it’s nice to know that most systems return values from a function in a specific register (for example, `%eax` on the Intel 386 architecture, `a0` on the MIPS architecture, or `%o0` on the SPARC architecture.¹ so you may find yourself using this command to find out the values which a function returns.²

1. In SPARC, the register names change on return from a function. The function places the return value in `%i0`, which becomes `%o0` after returning.

2. Shouldn’t the debugger volunteer this information? Yes, it should, but many don’t. No debugger that I know of even comes close to being perfect.

- *Modifying data and register contents* is an obvious way of modifying program execution.
- *breakpoints* stop execution of the process when the process attempts to execute an instruction at a certain address.
- *Single stepping* originally meant to execute a single machine instruction and then return control to the debugger. This level of control is no longer of much use: the machine could execute hundreds of millions of instructions before hitting the bug. Nowadays, there are four different kinds of single stepping. You can choose one of each of these options:
 - Instead of executing a single machine instruction, it might execute a single high-level language instruction or a single line of code.
 - Single stepping a function call instruction will normally land you in the function you're calling. Frequently, you're not interested in the function: you're pretty sure that it works correctly, and you just want to continue in the current function. Most debuggers have the ability to step "over" a function call rather than through it. You don't get the choice with a system call: you always step "over" it, since there is usually no way to trace into the kernel. To trace system calls, you use either a system call trace utility such as *ktrace*, or a kernel debugger.

In the following section, we'll look at how *gdb* implements these functions.

The *gdb* command set

In this section, we'll look at the *gdb* command set from a practical point of view: how do we use the commands that are available? This isn't meant to be an exhaustive description: if you have *gdb*, you should also have the documentation, both in GNU *info* form and also in hardcopy. Here we'll concentrate on how to use the commands.

Breakpoints and Watchpoints

As we have seen, the single biggest difference between a debugger and other forms of debugging is that a debugger can stop and restart program execution. The debugger will stop execution under two circumstances: if the process receives a signal, or if you tell it to stop at a certain point. For historical reasons, *gdb* refers to these points as *breakpoints* or *watchpoints*, depending on how you specify them:

- A *breakpoint* tells *gdb* to take control of process execution when the program would execute a certain code address.
- A *watchpoint* tells *gdb* to take control of process execution when a certain memory address is changed.

Conceptually, there is little difference between these two functions: a breakpoint checks for a certain value in the *program counter*, the register that addresses the next instruction to be executed, while a watchpoint checks for a certain value in just about anything else. The distinction is made because the implementation is very different. Most machines specify a special *breakpoint* instruction, but even on those machines that do not, it's easy enough to find an instruction which will do the job. The system replaces the instruction at the breakpoint address with a breakpoint

instruction. When the instruction is executed, the breakpoint instruction causes a trap, and the system invokes the debugger.

On the other hand, you can't use this technique for watching for changed memory contents. *gdb* solves this problem by executing the program one instruction at a time and examining the contents of memory after every instruction. This means that for every program instruction, *gdb* will execute thousands of instructions to check the memory locations. This makes program execution several orders of magnitude slower.

Many systems provide hardware support for this kind of check. For example, the Intel 386 architecture has four *breakpoint registers*. Each register can specify an address and an event for which a breakpoint interrupt should be generated. The events are instruction execution (this is the classical breakpoint we just discussed), memory write (our watchpoint), and memory read (which *gdb* can't detect at all). This support allows you to run at full speed and still perform the checks. Unfortunately, most UNIX systems don't support this hardware, so you need to run in stone-age simulation mode.

You set a breakpoint with the *breakpoint* command, which mercifully can be abbreviated to *b*. Typically, you'll set at least one breakpoint when you start the program, and possibly later you'll set further breakpoints as you explore the behaviour of the program. For example, you might start a program like this:

```
$ gdb bisdnd
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc...
(gdb) b handle_charge          set a breakpoint at handle_charge
Breakpoint 1 at 0x91e9: file msgutil.c, line 200.
```

gdb prints this political statement every time you start it. I've shown it in this case in respect of the sentiments of the people who produced it, but in the remaining examples in this book I'll omit it, since it doesn't change from one invocation to the next.

Running the program

When you start *gdb*, it's much like any other interactive program: it reads input from *stdin* and writes to *stdout*. You specify the name of the program you want to start, but initially that's all. Before you actually debug the process, you need to start it. While doing so, you specify the parameters that you would normally specify on the command line. In our case, our program *bisdnd* would normally be started as:

```
$ bisdnd -s 24 -F
```

It would be tempting (in fact, it would be a very good idea) just to put the word *gdb* in front of this command line invocation, but for historical reasons all UNIX debuggers take exactly two parameters: the first is the name of the program to start, and the second, if present, is the name of a core dump file.

Instead, the normal way to specify the parameters is when we actually run the program:

```
(gdb) r -s 24 -F          and run the program
```

```
Starting program: /usr/src/bisdn/bisdnd/bisdnd -s 24 -F
```

An alternative would be with the `set args` command:

```
(gdb) set args -s 24 -F          define the arguments
(gdb) r                          and run the program
Starting program: /usr/src/bisdn/bisdnd/bisdnd -s 24 -F
```

Stopping the process

Once you let the process run, it should run in the same way as it would do without a debugger, until it hits a breakpoint or it receives a signal. There are a few wrinkles, but they're relatively uncommon.

This could go on for hours, of course, depending on what the process does. Possibly you are concerned about the fact that the process might be looping or hanging, or you're just curious about what it's doing right now. Before you can talk to *gdb* again, you need to *stop* the process. This isn't the same thing as *termination*: the process continues to exist, but its execution is suspended until you start it again.

An obvious way to get *gdb*'s attention again is to send it a signal. That's simple: you can send a SIGINT via the keyboard, usually with the **CTRL-C** key:

```
^C
Program received signal SIGINT, Interrupt.
0x8081f31 in read ()
(gdb)
```

Alternatively, of course, you could hit a breakpoint, which also stops the execution:

```
Breakpoint 1, handle_charge (isdnfd=4, cp=0x11028, units=1, now=0xefbfd2b8, an=3,
channel=0) at msgutil.c:200
200     cp->charge = units;
(gdb)
```

Stack trace

Once we have stopped the process, the most obvious thing is to take a look around. As we have already seen, the stack trace command is probably the most useful of all. If your program bombs out, it will usually stop in a well-defined place, and your first question will be "how did I get here?". *gdb* implements this function with the `backtrace` command, which can be abbreviated to `bt`. A backtrace looks like:

```
(gdb) bt
#0  handle_charge (isdnfd=4, cp=0x11028, units=1, now=0xefbfd2b8, an=3, channel=0)
    at msgutil.c:200
#1  0x95e0 in info_dssl (isdnfd=4, s=0xefbfd504 "i") at msgutil.c:324
#2  0x7ab3 in msg_info (isdnfd=4, rbuf=0xefbfd504 "i") at msg.c:569
#3  0x1f80 in main (argc=7, argv=0xefbfde30) at main.c:441
(gdb)
```

This format looks rather unintelligible at first, so let's look at it in more detail. Each entry refers to a particular stack frame. The information is:

- The *stack frame number*, with a leading #. You can refer to this number with the `frame` command.
- The address of the next instruction in the stack frame. In frame 0, the current execution address, this is the next instruction that will be executed when you continue from the debugger. In the other stack frames, it's the next instruction that will be executed after returning control to that frame.

In frame 0, the address is specified simply as `handle_charge`. This means that the program is stopped at the beginning of the function. In the other frames, the address is specified in hexadecimal as well. It doesn't tell you how far into the function you are.

- The current values of the parameters supplied to the function. You'll only see this information if you have compiled the source file with the `-g` option. Don't rely on this information being accurate: it's common enough for programs to change the values of the parameters supplied, and the values shown here are the values which are on the stack at the time the program was stopped.
- The name of the file and the line number of the source instruction.

Once you have found where you are, the most obvious thing to do is to look around a bit. *gdb* supplies a number of options which help. Initially, they apply to frame 0, but you can change that with the *frame* command. First, let's look at the source code where we are:

Listing source code

To look at the source code in the current position, use the `list` command, which can be abbreviated to `l`:

```
(gdb) l
195             int units,
196             struct timeval *now,
197             int an,
198             int channel)
199     {
200         cp->charge = units;
201
202         if (cp->aoc_last.tv_sec == 0) /* if secs are 0, so will be usecs */
203             cp->aoc_last = *now;
204         else if (cp->aoc_now.tv_sec == 0)
```

If you look back to the stack trace, you'll see that our current position is at line 200, but *gdb* started at line 195. This is an attempt to show the area around the function. As you can see in this case, it wasn't enough: the function declaration takes up a number of lines. There are two parameters (`isdnfd` and `cp`) which are not displayed, so let's back up three lines (to 192) and have a look there:

```
(gdb) l 192
187     }
188
189     /*-----*
190     *      process the charging units
191     *-----*/
192     static void
193     handle_charge (int isdnfd,
194                  struct confdata *cp,
195                  int units,
```

```
196                                struct timeval *now,
```

How about that, *gdb* always goes back 5 lines. Still, now we have the start of our function. If we want to see more, we just press **ENTER**:

```
(gdb)                                (ENTER pressed)
197                                int an,
198                                int channel)
199    {
200        cp->charge = units;
201
202        if (cp->aoc_last.tv_sec == 0)          /* if secs are 0, so will be usecs */
203            cp->aoc_last = *now;
204        else if (cp->aoc_now.tv_sec == 0)
205            cp->aoc_now = *now;
206        else
```

In general, if you press **ENTER**, *gdb* will attempt to re-execute the last instruction, possibly with parameters it calculates (like the starting address for the `list` command).

Examining other stack frames

We've just arrived in this function, so we're probably more interested in the calling function than the function we're in. Indeed, maybe we're just wondering how we can get here at all. The stack trace has shown us where we came from, but we might want to look at it in more detail. We do that with the `frame` command, which can be abbreviated to `f`. We supply the number of the frame which we want to examine:

```
(gdb) f 1                                look at frame 1
#1  0x95e0 in info_dssl (isdnfd=4, s=0xefbfd504 "i") at msgutil.c:324
324      handle_charge (isdnfd, cp, i, &time_now, appl_no, channel);
(gdb) l 1                                and list the source code
319      gettimeofday (&time_now, NULL);
320
321      cp = getcp (appl_typ, appl_no);
322      i = decode_q932_aoc (s);
323      if (i != -1)
324          handle_charge (isdnfd, cp, i, &time_now, appl_no, channel);
325      break;
326
327      default:
328          dump_info (appl_typ, appl_no, mp->info);
```

Not surprisingly, line 324 is a call to `handle_charge`. This shows an interesting point: clearly, the return address can't be the beginning of the instruction. It must be somewhere near the end. If I stop execution on line 324, I would expect to stop before calling `handle_charge`. If I stop execution at address 0x95e0, I would expect to stop after calling `handle_charge`. We'll look into this question more further down, but it's important to bear in mind that a line number does not uniquely identify the instruction.

Displaying data

The next thing you might want to do is to look at some of the variables in the current stack environment. There are a number of ways to do this. The most obvious way is to specify a variable you want to look at. In *gdb*, you do this with the `print` command, which can be abbreviated to `p`. For example, as we have noted, the values of the parameters that `backtrace` prints are the

values at the time when process execution stopped. Maybe we have reason to think they might have changed since the call. The parameters are usually copied on to the stack, so changing the values of the parameters supplied to a function doesn't change the values used to form the call. We can find the original values in the calling frame. Looking at line 324 above, we have the values `isdnfd`, `cp`, `i`, `&time_now`, `appl_no`, and `channel`. Looking at them,

```
(gdb) p isdnfd
$1 = 6                                an int
```

The output format means 'result 1 has the value 6'. You can refer to these calculated results at a later point if you want, rather than recalculating them:

```
(gdb) p $1
$2 = 6
(gdb) p cp                                a struct pointer
$3 = (struct confdata *) 0x11028
```

Well, that seems reasonable: `cp` is a *pointer* to a `struct confdata`, so `gdb` shows us the address. That's not usually of much use, but if we want to see the contents of the struct to which it points, we need to specify that fact in the standard C manner:

```
(gdb) p *cp
$4 = {interface = "ipi3", '\000' <repeats 11 times>, atyp = 0, appl = 3,
      name = "daemon\000\000\000\000\000\000\000\000\000\000", controller = 0,
      isdnstype = 1, telnloc_ldo = "919120", '\000' <repeats 26 times>,
      telnrem_ldo = "919122", '\000' <repeats 26 times>, telnloc_rdi = "919120",
      '\000' <repeats 26 times>, telnrem_rdi = "6637919122", '\000' <repeats 22 times>,
      reaction = 0, service = 2, protocol = 0, telaction = 0, dialretries = 3,
      recoverytime = 3, callbackwait = 1,
      ...much more
```

This format is not the easiest to understand, but there is a way to make it better: the command `set print pretty` causes `gdb` to structure printouts in a more appealing manner:

```
(gdb) set print pretty
(gdb) p *cp
$5 = {
  interface = "ipi3", '\000' <repeats 11 times>,
  atyp = 0,
  appl = 3,
  name = "daemon\000\000\000\000\000\000\000\000\000\000",
  controller = 0,
  isdnstype = 1,
  telnloc_ldo = "919120", '\000' <repeats 26 times>,
  telnrem_ldo = "919122", '\000' <repeats 26 times>,
  telnloc_rdi = "919120", '\000' <repeats 26 times>,
  telnrem_rdi = "6637919122", '\000' <repeats 22 times>,
  ...much more
```

The disadvantage of this method, of course, is that it takes up much more space on the screen. It's not uncommon to find that the printout of a structure takes up several hundred lines.

The format isn't always what you'd like. For example, `time_now` is a `struct timeval`, which looks like:

```
(gdb) p time_now
$6 = {
  tv_sec = 835701726,
```

```
    tv_usec = 238536
}
```

The value 835701726 is the number of seconds since the start of the epoch, 00:00 UTC on 1 January 1970, the beginning of UNIX time. *gdb* provides no way to transform this value into a real date. On many systems, you can do it with a little-known feature of the *date* command:

```
$ date -r 835701726
Tue Jun 25 13:22:06 MET DST 1996
```

Displaying register contents

Sometimes it's not enough to look at official variables. Optimized code can store variables in registers without ever assigning them a memory location. Even when variables do have a memory location, you can't count on the compiler to store them there immediately. Sometimes you need to look at the register where the variable is currently stored.

A lot of this is deep magic, but one case is relatively frequent: after returning from a function, the return value is stored in a specific register. In this example, which was run on FreeBSD on an Intel platform, the compiler returns the value in the register *eax*. For example:

```
Breakpoint 2, 0x133f6 in isatty ()           hit the breakpoint
(gdb) fin                                   continue until the end of the function
Run till exit from #0  0x133f6 in isatty ()
0x2fe2 in main (argc=5, argv=0xefbfd4c4) at mklinks.c:777 back in the calling function
777     if (interactive = isatty (Stdin)      /* interactive */
(gdb) i reg                                look at the registers
eax                0x1          1          isatty returned 1
ecx                0xefbfd4c4    -272640828
edx                0x1          1
ebx                0xefbfd602    -272640510
esp                0xefbfd48c    0xefbfd48c
ebp                0xefbfd4a0    0xefbfd4a0
esi                0x0          0
edi                0x0          0
eip                0x2fe2        0x2fe2
eflags             0x202        514
(gdb)
```

This looks like overkill: we just wanted to see the value of the register *eax*, and we had to look at all values. An alternative in this case would have been to print out the value explicitly:

```
(gdb) p $eax
$3 = 1
```

At this point, it's worth noting that *gdb* is not overly consistent in its naming conventions. In the disassembler, it will use the standard assembler convention and display register contents with a % sign, for example *%eax*:

```
0xf011bc7c <mi_switch+116>:    movl    %edi,%eax
```

On the other hand, if you want to refer to the value of the register, we must specify it as *\$eax*. *gdb* can't make any sense of *%eax* in this context:

```
(gdb) p %eax
```

syntax error

Single stepping

Single stepping in its original form is supported in hardware by many architectures: after executing a single instruction, the machine automatically generates a hardware interrupt that ultimately causes a SIGTRAP signal to the debugger. *gdb* performs this function with the `stepi` command.

You won't want to execute individual machine instructions unless you are in deep trouble. Instead, you will execute a *single line* instruction, which effectively single steps until you leave the current line of source code. To add to the confusion, this is also frequently called *single stepping*. This command comes in two flavours, depending on how it treats function calls. One form will execute the function and stop the program at the next line after the call. The other, more thorough form will stop execution at the first executable line of the function. It's important to notice the difference between these two functions: both are extremely useful, but for different things. *gdb* performs single line execution omitting calls with the `next` command, and includes calls with the `step` command.

```
(gdb) n
203      if (cp->aoc_last.tv_sec == 0)          /* if secs are 0, so will be usecs */
(gdb)                                     (ENTER pressed)
204          cp->aoc_last = *now;
(gdb)                                     (ENTER pressed)
216      if (do_fullscreen)
(gdb)                                     (ENTER pressed)
222      if ((cp->unit_length_type == ULTYP_DYN) && (cp->aoc_valid == AOC_VALID))
(gdb)                                     (ENTER pressed)
240          if (do_debug && cp->aoc_valid)
(gdb)                                     (ENTER pressed)
243      }
(gdb)                                     (ENTER pressed)
info_dss1 (isdnfd=6, s=0xefbfcac0 "i") at msgutil.c:328
328          break;
(gdb)
```

Modifying the execution environment

In *gdb*, you do this with the `set` command.

Jumping (changing the address from which the next instruction will be read) is really a special case of modifying register contents, in this case the *program counter* (the register that contains the address of the next instruction). Some architectures, including the Intel i386 architecture, refer to this register as the *instruction pointer*, which makes more sense. In *gdb*, use the `jump` command to do this. Use this instruction with care: if the compiler expects the stack to look different at the source and at the destination, this can easily cause incorrect execution.

Using debuggers

There are two possible approaches when using a debugger. The easier one is to wait until something goes wrong, then find out where it happened. This is appropriate when the process gets a signal and does not overwrite the stack: the `backtrace` command will show you how it got there.

Sometimes this method doesn't work well: the process may end up in no-man's-land, and you see something like:

```
Program received signal SIGSEGV, Segmentation fault.  
0x0 in ?? ()  
(gdb) bt                                abbreviation for backtrace  
#0  0x0 in ?? ()                          nowhere  
(gdb)
```

Before dying, the process has mutilated itself beyond recognition. Clearly, the first approach won't work here. In this case, we can start by conceptually dividing the program into a number of parts: initially we take the function `main` and the set of functions which `main` calls. By single stepping over the function calls until something blows up, we can localize the function in which the problem occurs. Then we can restart the program and single step through this function until we find what it calls before dying. This iterative approach sounds slow and tiring, but in fact it works surprisingly well.

5

Reading Code

This section still needs to be written. It will be demonstrated.

6

Preparing to debug a kernel

When building a kernel for debug purposes, you need to know how you're going to perform the debugging. If you're using remote debugging, it's better to have the kernel sources and objects on the machine from which you perform the debugging, rather than on the machine you're debugging. That way the sources are available when the machine is frozen. On the other hand, you should always build the kernel on the machine which you are debugging. There are two ways to do this:

1. Build the kernel on the debug target machine, then copy the files to the debugging machine.
2. NFS mount the sources on the debugging machine and then build from the target machine.

Unless you're having problems with NFS, the second alternative is infinitely preferable. It's very easy to forget to copy files across, and you may not notice your error until hours of head scratching have passed. I use the following method:

- All sources are kept on a single large drive called */src* and mounted on system *echunga*.
- */src* contains subdirectories */src/FreeBSD*, */src/NetBSD*, */src/OpenBSD* and */src/Linux*.

These directories in turn contain subdirectories with source trees for specific systems. For example, */src/FreeBSD/ZAPHOD/src* is the top-level build directory for system *zaphod*.

- On *zaphod* I mount */src* under the same name and create two symbolic links:

```
# ln -s /src/FreeBSD/ZAPHOD/src /usr/src
# ln -s /src/FreeBSD/obj /usr/obj
```

In this manner, I can build the system in the “normal” way and have both sources and binaries on the remote system *echunga*. Normally the kernel build installs the kernel in the “standard” place: */boot/kernel/kernel* for FreeBSD version 5, */netbsd* for NetBSD, or */bsd* on OpenBSD. The versions installed there usually have the symbols stripped off, however, so you'll have to find where the unstripped versions are. That depends on how you build the kernel.

Kernel debuggers

Currently, two different kernel debuggers are available for BSD systems: *ddb* and *gdb*. *ddb* is a low-level debugger completely contained in the kernel, while you need a second machine to debug with *gdb*.

You can build a FreeBSD kernel with support for both debuggers, but in NetBSD and OpenBSD you must make a choice.

Building a kernel for debugging

There are three different kinds of kernel parameters for debug kernels:

- As an absolute minimum to be able to debug things easily, you need a kernel with debug symbols. This is commonly called a *debug kernel*, though in fact compiling with symbols adds no code, and the kernel is identical in size.¹

To create a debug kernel, ensure you have the following line in your kernel configuration file:

```
makeoptions      DEBUG=-g                #Build kernel with gdb(1) debug symbols
```

In most cases, this is simply a matter of removing the comment character at the beginning of the line.

- If you want to use a kernel debugger, you need additional parameters to specify which debugger and some other options. These options differ between the individual systems, so we'll look at them in the following sections.
- Finally, the kernel code offers specific consistency checking code. Often this changes as various parts of the kernel go through updates which require debugging. Again, these options differ between the individual systems, so we'll look at them in the following sections.

FreeBSD kernel

FreeBSD has recently changed the manner of building the kernel. The canonical method is now:

```
# cd /usr/src
# make kernel KERNCONF=ZAPHOD
```

Assuming that */usr/src* is not a symbolic link, this performs the following steps:

- It builds a kernel */usr/obj/sys/ZAPHOD/kernel.debug* and a stripped copy at */usr/obj/sys/ZAPHOD/kernel*.
- It also builds all modules. This can take longer than the kernel itself.
- It removes any directory */boot/kernel.old* and renames */boot/kernel* to */boot/kernel.old*.

1. On occasion the compiler generates slightly different code when compiling with symbols, but the difference is negligible. It does make it difficult to perform a direct comparison of the code with *cmp*, however.

- It installs */usr/obj/sys/ZAPHOD/kernel* and the modules in */boot/kernel*.

If you're building kernels for debugging, there's a good chance that they won't work; they may not even boot. That's why the old version is saved in */boot/kernel.old*. If the kernel doesn't boot, you boot */boot/kerne.old/kernel* and recover.

Under these circumstances, the method described above is a little heavy-handed: it's too easy to overwrite your */boot/kerne.old/kernel* and end up with two kernels, neither of which run. Also, chances are that you won't want to rebuild *every* module every time. You can speed things up a lot with the following approach:

```
# cd /usr/src
# make buildkernel KERNCONF=ZAPHOD -DNOCLEAN -DNO_MODULES -j2
# make installkernel KERNCONF=ZAPHOD -DNO_MODULES      install the kernel, renaming /boot/kernel
# make reinstallkernel KERNCONF=ZAPHOD -DNO_MODULES    install the kernel, overwriting /boot/kernel
```

The options have the following meanings:

- `-DNOCLEAN` tells the build process not to remove the old object files. This greatly speeds up a kernel build where you've only changed a file or two.
- `-DNO_MODULES` tells the build process to build only a kernel.
- `-j2` tells the build process to perform two compilations in parallel at any one time. The value 2 is right for a single processor; `-j3` tends to be slower again. If you're building on an SMP machine, multiply the number of CPUs by 2. For example, on a four-way machine you would use `-j8`.
- The `installkernel` target first renames the */boot/kernel* to */boot/kernel.old* and then installs */usr/obj/sys/ZAPHOD/kernel* and any the modules in */boot/kernel*, in the same way as the `kernel` target.
- The `reinstallkernel` target does not rename */boot/kernel*. It overwrites the old contents. Use this when the previous kernel was no good.

In the situations we're looking at, though, you're unlikely to build the kernel in */usr/src*, or if you do, it will be a symbolic link. In either case, the location of the kernel build directory changes. In the example above, if */usr/src* is a symbolic link to */src/FreeBSD/ZAPHOD/src*, the kernel binaries will be placed in */usr/obj/src/FreeBSD/ZAPHOD/src/sys/ZAPHOD*, and the debug kernel will be called */usr/obj/src/FreeBSD/ZAPHOD/src/sys/ZAPHOD/kernel.debug*.

Setting up debug macros

FreeBSD has a number of debug macros in the directory */usr/src/tools/debugscripts*. Normally you install them in the kernel build directory:

```
# cd /src/FreeBSD/obj/src/FreeBSD/ZAPHOD/src/sys/ZAPHOD/
# make gdbinit
grep -v '# XXX' /src/FreeBSD/ZAPHOD/src/sys/./tools/debugscripts/dot.gdbinit
| sed "s:MODPATH:/src/FreeBSD/obj/src/FreeBSD/ZAPHOD/src/sys/ZAPHOD/modules:" \
> .gdbinit
cp /src/FreeBSD/ZAPHOD/src/sys/./tools/debugscripts/gdbinit.kernel \
  /src/FreeBSD/ZAPHOD/src/sys/./tools/debugscripts/gdbinit.vinum \
  /src/FreeBSD/obj/src/FreeBSD/ZAPHOD/src/sys/ZAPHOD \
cp /src/FreeBSD/ZAPHOD/src/sys/./tools/debugscripts/gdbinit.i386
  /src/FreeBSD/obj/src/FreeBSD/ZAPHOD/src/sys/ZAPHOD/gdbinit.machine \
```

NetBSD kernel

NetBSD now has a do-it-all tool called *make.sh*. As the name suggests, it's a shell script front end to a bewildering number of build options. To build, say, a 1.6W kernel for *daikon*, an i386 box, you might do this:

```
# ln -s /src/NetBSD/1.6W-DAIKON/src /usr/src
# cd /usr/src
# ./build.sh tools
```

This step builds the tool chain in the directory *tools*.

Continuing,

```
# ./build.sh kernel=DAIKON
# mv /netbsd /onetbsd
# cp sys/arch/i386/compile/DAIKON/netbsd /
```

This builds a kernel file */usr/src/sys/arch/i386/compile/DAIKON/netbsd.gdb* with debug symbols, and a file */usr/src/sys/arch/i386/compile/DAIKON/netbsd* without.

ddb

The local debugger is called *ddb*. It runs entirely on debugged machine and displays on the console (including serial console if selected). There are a number of ways to enter it:

- You can configure your system to enter the debugger automatically from panic. In FreeBSD, `debugger_on_panic` needs to be set.
- `DDB_UNATTENDED` resets `debugger_on_panic`.
- Enter from keyboard with **CTRL-ALT-ESC**.

The following examples are from a FreeBSD system on the Intel ia32 platform.

ddb entry from keyboard

```
# Debugger("manual escape to debugger")
Stopped at Debugger+0x44: pushl %ebx
db> t
Debugger(c03ca5e9) at Debugger+0x44
scgetc(c16d9800,2,c16d1440,c046ac60,0) at scgetc+0x426
sckbdevent(c046ac60,0,c16d9800,c16d1440,c16d4300) at sckbdevent+0x1c9
atkbd_intr(c046ac60,0,cc04bd18,c024c79a,c046ac60) at atkbd_intr+0x22
atkbd_isa_intr(c046ac60) at atkbd_isa_intr+0x18
ithread_loop(c16d4300,cc04bd48,c16d4300,c024c670,0) at ithread_loop+0x12a
fork_exit(c024c670,c16d4300,cc04bd48) at fork_exit+0x58
fork_trampoline() at fork_trampoline+0x8db>
db>
```

ddb entry on panic

A call to panic produces a register summary:

```
Fatal trap 12: page fault while in kernel mode
fault virtual address  = 0x64
fault code             = supervisor read, page not present
instruction pointer    = 0x8:0xc02451d7
stack pointer         = 0x10:0xccd99a20
frame pointer         = 0x10:0xccd99a24
code segment          = base 0x0, limit 0xffffffff, type 0x1b
                     = DPL 0, pres 1, def32 1, gran 1
processor eflags      = interrupt enabled, resume, IOPL = 0
current process       = 107 (syslogd)
```

If you have selected it, you will then enter *ddb*

```
kernel: type 12 trap, code=0
Stopped at      devsw+0x7:      cmpl      $0,0x64(%ebx)
db> tr                                stack backtrace
devsw(0,c045cd80,cc066e04,cc066e04,0) at devsw+0x7
cn_devopen(c045cd80,cc066e04,0) at cn_devopen+0x27
cnopen(c0435ec8,6,2000,cc066e04,0) at cnopen+0x39
spec_open(ccd99b50,ccd99b24,c0320589,ccd99b50,ccd99bc4) at spec_open+0x127
spec_vnoperate(ccd99b50,ccd99bc4,c029984b,ccd99b50,ccd99d20) at spec_vnoperate+0x15
ufs_vnoperatespec(ccd99b50,ccd99d20,0,cc066e04,6) at ufs_vnoperatespec+0x15
vn_open(ccd99c2c,ccd99bf8,0,cc066f0c,cc066d00) at vn_open+0x333
open(cc066e04,ccd99d20,8054000,bfbfef64,bfbfef34) at open+0xde
syscall(2f,2f,2f,bfbfef34,bfbfef64) at syscall+0x24c
syscall_with_err_pushed() at syscall_with_err_pushed+0x1b
- syscall (5, FreeBSD ELF, open), eip = 0x280aae50, esp = 0xbfbfe960, ebp = 0xbfbfe9cc -
```

The main disadvantage of *ddb* is the limited symbol support. This backtrace shows the function names, but not the parameters, and not the file names or line numbers. It also cannot display automatic variables, and it does not know the types of global variables.

Serial console

Until about 15 years ago, the console of most UNIX machines was a terminal connected by a serial line. Nowadays, most modern machines have an integrated display. If the system fails, the display fails too. For debugging, it's often useful to fall back to the older *serial console* on machines with a serial port. Instead of a terminal, though, it's better to use a terminal emulator on another computer: that way you can save the screen output to a file.

Serial console: debugging machine

To boot a machine with a serial console, first connect the system with a serial cable to a machine with a terminal emulator running at 9600 bps. Start a terminal emulator; I run the following command inside an X window so that I can copy any interesting output:

```
# cu -s 9600 -l /dev/cuaa0
```

The device name will change depending on the system you're using and the serial port hardware. The machine doesn't need to be a BSD machine. It can even be a real terminal if you can find one, but that makes it difficult to save output.

`cu` runs `setuid` to the user `uucp`. You may need to adjust ownership or permissions of the serial port, otherwise you'll get the unlikely looking error

```
# cu -l /dev/cuaa1
cu: /dev/cuaa1: Line in use
```

Typical permissions are:

```
# ls -l /dev/cuaa0
crw-rw-rw- 1 root wheel 28, 0 Nov 3 15:23 /dev/cuaa0
# ps aux | grep cu
uucp 6828 0.0 0.5 1020 640 p0 I+ 3:21PM 0:00.01 cu -s 9600 -l /dev/cuaa0
uucp 6829 0.0 0.5 1020 640 p0 I+ 3:21PM 0:00.01 cu -s 9600 -l /dev/cuaa0
```

Boot the target machine with serial console support:

- On FreeBSD, interrupt the boot sequence at the following point:

```
Hit [Enter] to boot immediately, or any other key for command prompt.
Booting [kernel] in 6 seconds...           press space bar here

OK set console=comconsole                 select chosen serial port
the remainder appears on the serial console
OK boot                                   and continue booting normally
OK boot -d                               or boot and go into debugger
```

If you specify the `-d` flag to the `boot` command, the kernel will enter the kernel debugger as soon as it has enough context to do so.

You “choose” a serial port by setting bit 0x80 of the device flags in `/boot/loader.conf`:

```
hint.sio.0.flags="0x90"
```

In this example, bit 0x10 is also set to tell the kernel gdb stub to access remote debugging via this port.

- On NetBSD,

```
>> NetBSD BIOS Boot, revision 2.2
>> (user@buildhost, builddate)
>> Memory: 637/15360 k
Press return to boot now, any other key for boot menu
booting hd0a:netbsd - starting in 5       press space bar here

> consdev com0                           select first serial port
the remainder appears on the serial console
>> NetBSD/i386 BIOS Boot, Revision 2.12
>> (autobuild@tgm.daemon.org, Sun Sep 8 19:22:13 UTC 2002)
>> Memory: 637/129984 k
> boot                                   continue booting normally
> boot -d                               or boot and go into debugger
```

In NetBSD, you can't run the serial console and the debugger on the same interface. If the serial console is on the debugger interface, the bootstrap ignores the `-d` flag.

Problems with remote debugging

Remote debugging is a powerful technique, but it's anything but perfect. Here are some of the things which will annoy you:

- It is *slow*. Few serial ports can run at more than 115,200 bps, a mere 11 kB/s. Dumping the `msgbuf` (the equivalent of *dmesg*) can take five minutes.
- If that weren't enough, the GNU remote serial protocol is wasteful.
- The link must work when the system is not running, so you can't use the serial drivers. Instead, there's a primitive driver, called a *stub*, which handles the I/O. It's inefficient, and for reasons we don't quite understand, at least on FreeBSD it does not work reliably over 9,600 bps, further slowing things down.
- Why don't we know why the stub doesn't work reliably over 9,600 bps? How do you debug a debugger? Code reading can only get you so far.
- "Legacy" serial ports are on their way out. Modern laptops often don't have them any more, and it won't be long before they're a thing of the past.

FreeBSD also supports debugging over a firewire (IEEE 1349) interface. This eliminates the delay of the serial link (firewire is significantly faster than 100 Mb/s Ethernet), but it doesn't help much with *gdb*'s inherent slowness. Firewire also offers the possibility of accessing the target processor memory without participation of the target processor, which promises to help debug a large number of processor hangs and halts. We'll look at it in more detail below.

In addition, some other debugging interfaces are around, but they're not overly well supported. NetBSD supports debugging over Ethernet, but only on NE2000 cards. FreeBSD now supports firewire debugging, which we'll look at in the next section.

Kernel gdb

Kernel *gdb* is the same *gdb* program you know and love in userland. It provides the symbolic capability that is missing in *ddb*, and also macro language capability. It can run on serial lines (and in some cases on Ethernet and Firewire links) and post-mortem dumps. In the last case, it requires some modifications to adapt to the dump structure, so you must specify the `-k` flag when using it on kernel dumps.

gdb is not a very good fit to the kernel: it assumes that it's running in process context, and it's relatively difficult to get things like stack traces and register contents for processes other than the one (if any) currently running on the processor. There are some macros that help in this area, but it's more than a little kludgy.

Entering gdb from ddb

In FreeBSD you can build a kernel with support for both *ddb* and *gdb*. You can then change backwards and forwards between them. For example, if you're in *ddb*, you can go to *gdb* like this:

```
db> gdb
```

```

Next trap will enter GDB remote protocol mode
db> si                                step a single instruction to reenter ddb

||| $T0b08:d75124c0;05:249ad9cc;04:209ad9cc;#32~.

Disconnected.
#

```

The noise at the bottom is the prompt from the *gdb* stub on the debugged machine: the serial console and *gdb* are sharing the same line. In this case, you need to exit the terminal emulator session to be able to debug. The input sequence `~.` at the end of the line tells *cu* to exit, as shown on the following lines. Next, you need to attach from the local *gdb*, which we'll see in the next section.

Running serial gdb

On the side of the debugging ('local') machine you run *gdb* in much the same way as you would for a userland program. In the case of the panic we saw above, enter:

```

$ cd /usr/src/sys/compile/CANBERRA
$ gdbk
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-unknown-freebsd".
(kgdb) target remote /dev/cuaa1                                connect to remote machine
devsw (dev=0x0) at ../../../../kern/kern_conf.c:83
83         if (dev->si_devsw)
(kgdb)

```

The first thing you would do there would be to do a backtrace:

```

(kgdb) bt
#0  devsw (dev=0x0) at ../../../../kern/kern_conf.c:83
#1  0xc027d0c7 in cn_devopen (cnd=0xc045cd80, td=0xcc066e04, forceopen=0x0)
    at ../../../../kern/tty_cons.c:344
#2  0xc027d211 in cnopen (dev=0xc0435ec8, flag=0x6, mode=0x2000, td=0xcc066e04)
    at ../../../../kern/tty_cons.c:376
#3  0xc0230f6f in spec_open (ap=0xccd99b50) at ../../../../fs/specfs/spec_vnops.c:199
#4  0xc0230e45 in spec_vnoperate (ap=0xccd99b50) at ../../../../fs/specfs/spec_vnops.c:119
#5  0xc0320589 in ufs_vnoperatespec (ap=0xccd99b50) at ../../../../ufs/ufs/ufs_vnops.c:2676
#6  0xc029984b in vn_open (ndp=0xccd99c2c, flagp=0xccd99bf8, cmode=0x0) at vnode_if.h:159
#7  0xc0294c12 in open (td=0xcc066e04, uap=0xccd99d20) at ../../../../kern/vfs_syscalls.c:1099
#8  0xc035aedc in syscall (frame={tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f,
    tf_ei = 0xbfbfef34, tf_ei = 0xbfbfef64, tf_ebp = 0xbfbfe9cc,
    tf_esp = 0xccd99d74, tf_ebx = 0x8054000, tf_edx = 0xf7, tf_ecx = 0x805402f,
    tf_eax = 0x5, tf_trapno = 0x0, tf_err = 0x2, tf_eip = 0x280aae50,
    tf_cs = 0x1f, tf_eflags = 0x293, tf_esp = 0xbfbfe960, tf_ss = 0x2f})
    at ../../../../i386/i386/trap.c:1129
#9  0xc034c28d in syscall_with_err_pushed ()
#10 0x804b2b5 in ?? ()
#11 0x804abe9 in ?? ()
#12 0x804b6fe in ?? ()
#13 0x804b7af in ?? ()
#14 0x8049fb5 in ?? ()
#15 0x8049709 in ?? ()
(kgdb)

```

This corresponds to the *ddb* example above. As can be seen, it provides a lot more information. Stack frames 10 to 15 are userland code: on most platforms, userland and kernel share the same

address space, so it's possible to show the user call stack as well. If necessary, you can also load symbols for the process, assuming you have them available on the debugging machine.

Getting out of the debugger

How do you stop the debugger? You can hit `^C`, and you'll get a debugger prompt:

```
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
0xc5ac8378 in ?? ()
(gdb) The program is running. Exit anyway? (y or n) y
#
```

You may not realise the problem with this approach for a while: the debugged machine is still in the debugger, and it won't respond. You can reboot it, of course, but that's usually overkill. The correct way is the *detach* command:

```
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
0xc5ac8378 in ?? ()
(gdb) detach
Ending remote debugging.
(gdb)
```

You can then attach again with one of the *target remote* commands we have seen above.

Debugging running systems

For some things, you don't need to stop the kernel. If you're only looking, for example, and the data you're looking at is not very likely to change, you can use a debugger on the same system to look at its own kernel. In this case you use the special file */dev/mem* instead of dump file. You're somewhat limited in what you can do: you can't set breakpoints, you can't stop execution, and things can change while you're looking at them. You *can* change data, but you need to be particularly careful, or not care too much whether you crash the system.

Debugging a running FreeBSD system

```
# gdb -k /usr/src/sys/i386//MONORCHID/kernel.debug /dev/mem
GNU gdb 4.18
...
This GDB was configured as "i386-unknown-freebsd"...
IdlePTD at phsyical address 0x004f3000
initial pcb at physical address 0x0e5ccda0
panic messages:
---
---
#0  0xc023a6df in mi_switch () at ../../../../kern/kern_synch.c:779
779      cpu_switch();
(kgdb) bt
#0  0xc023a6df in mi_switch () at ../../../../kern/kern_synch.c:779
#1  0xffffffff in ?? ()
error reading /proc/95156/mem
```

You need the `-k` option to tell *gdb* that the “core dump” is really a kernel memory image. The line `panic messages` is somewhat misleading: the system hasn't panicked. This is also the

reason for the empty messages (between the two lines with ---).

Debugging a running NetBSD system

NetBSD's *gdb* no longer accepts the same syntax as FreeBSD, so on NetBSD you need a slightly different syntax:

```
# gdb /netbsd                                no dump
...
This GDB was configured as "i386--netbsd"...(no debugging symbols found)...
(gdb) target kcore /dev/mem                  specify the core file
#0  0xc01a78f3 in mi_switch ()
(gdb) bt                                      backtrace
#0  0xc01a78f3 in mi_switch ()
#1  0xc01a72ca in ltsleep ()
#2  0xc02d6c81 in uvm_scheduler ()
#3  0xc019a358 in check_console ()
(gdb)
```

In this case, we don't see very much of use, because we're using the standard kernel, which is stripped (thus the message above no debugging symbols found). Things look a lot better with symbols:

```
# gdb /usr/src/sys/arch/i386/compile/KIMCHI/netbsd.gdb
...
This GDB was configured as "i386--netbsd"...
(gdb) target kcore /dev/mem
#0  mi_switch (p=0xc0529be0) at ../../../../kern/kern_synch.c:834
834      microtime(&p->p_cpu->ci_schedstate.spc_runtime);
(gdb) bt
#0  mi_switch (p=0xc0529be0) at ../../../../kern/kern_synch.c:834
#1  0xc01a72ca in ltsleep (ident=0xc0529be0, priority=4, wmesg=0xc04131e4
"scheduler", timo=0, interlock=0x0) at ../../../../kern/kern_synch.c:482
#2  0xc02d6c81 in uvm_scheduler () at ../../../../uvm/uvm_glue.c:453
#3  0xc019a358 in check_console (p=0x0) at
../../../../kern/init_main.c:522
```

Debugging via firewire

Currently remote debugging via firewire is available only on FreeBSD. Firewire offers new possibilities for remote debugging:

- It provides a much faster method of remote debugging, though the speed is still limited by the inefficiencies of *gdb* processing.
- It provides a completely new method to debug systems which have crashed or hung: firewire can access the memory of the machine to be debugged without its intervention, which provides an interface similar to local memory debugging. This makes it possible to debug hangs and crashes which previously could not be debugged at all.

As with serial debugging, to debug a live system with a firewire link, compile the kernel with the option

```
options DDB
```

options GDB_REMOTE_CHAT is not necessary, since the firewire implementation uses sepa-

rate ports for the console and debug connection.

A number of steps must be performed to set up a fi rewire link:

- Ensure that both systems have fi rewire support, and that the kernel of the system to be debugged includes the *dcons* and *dcons_crom* drivers. At the time of writing, the kernel *gdb* infrastructure in FreeBSD is broken, and remote debugging will not work unless the fi rewire driver is compiled into the kernel. Add the following lines to your kernel configuration and build a new kernel:

```
device      firewire      # FireWire bus code
device      dcons         # dumb console driver
device      dcons_crom    # FireWire attachment
```

It's probably not necessary to include the *dcons* support, but since this is a bug, it's better to play it safe.

If fi rewire is loaded in the kernel (and if your machine has a fi rewire interface), you will should see something like this in the *dmesg* output:

```
fwohci0: OHCI version 1.10 (ROM=0)
fwohci0: No. of Isochronous channels is 4.
fwohci0: EUI64 43:4f:c0:00:1d:b0:a8:38
fwohci0: Phy 1394a available S400, 2 ports.
fwohci0: Link S400, max_rec 2048 bytes.
firewire0: <IEEE1394(FireWire) bus> on fwohci0
fwe0: <Ethernet over FireWire> on firewire0
if_fwe0: Fake Ethernet address: 42:4f:c0:b0:a8:38
fwe0: Ethernet address: 42:4f:c0:b0:a8:38
sbp0: <SBP-2/SCSI over FireWire> on firewire0
fwohci0: Initiate bus reset
fwohci0: node_id=0xc800ffc0, gen=1, CYCLEMASTER mode
firewire0: 1 nodes, maxhop <= 0, cable IRM = 0 (me)
firewire0: bus manager 0 (me)
```

When the *gdb* bug has been fixed, you won't need to have the driver in the kernel. Instead, load the KLDs:

```
# kldload firewire
```

- On the system to be debugged only, you need *dcons* and *dcons_crom*. If they have been loaded, you'll see the following in the *dmesg* output:

```
dcons_crom0: <dcons configuration ROM> on firewire0
dcons_crom0: bus_addr 0x13d3000
```

Otherwise load them:

```
# kldload dcons
# kldload dcons_crom
```

It is a good idea to load these modules at boot time with the following entry in */boot/loader.conf*:

```
dcons_crom_enable="YES"
```

This ensures that all three modules are loaded. There is no harm in loading *dcons* and *dcons_crom* on the debugging system, but if you only want to load the *fi rewired* module, include the following in */boot/loader.conf*:

```
firewire_enable="YES"
```

- Next, use *fwcontrol* to find the firewire node corresponding to the machine to be debugged. On the debugging machine you might see:

```
# fwcontrol
2 devices (info_len=2)
node      EUI64      status
  0  43-4f-c0-00-1d-b0-a8-38      0
  1  00-c0-4f-32-26-e8-80-61      1
```

The first node is always the local system, so in this case, node 1 is the machine to be debugged. If there are more than two systems, check from the other end to find which node corresponds to the remote system. On the machine to be debugged, it looks like this:

```
# fwcontrol
2 devices (info_len=2)
node      EUI64      status
  1  00-c0-4f-32-26-e8-80-61      0
  0  43-4f-c0-00-1d-b0-a8-38      1
```

- Next, on the debugging system, establish a firewire connection with *dconschat*:

```
# dconschat -br -G 5556 -t 00-c0-4f-32-26-e8-80-61
[dcons connected]
dcons_crom0: <dcons configuration ROM> on firewire0
dcons_crom0: bus_addr 0x13d300
fwohci0: BUS reset
fwohci0: node_id=0xc000ffc0, gen=3, CYCLEMASTER mode
firewire0: 1 nodes, maxhop <= 0, cable IRM = 0 (me)
firewire0: bus manager 0 (me)
fwohci0: BUS reset
fwohci0: node_id=0xc000ffc1, gen=4, CYCLEMASTER mode
firewire0: 2 nodes, maxhop <= 1, cable IRM = 1 (me)
firewire0: bus manager 1 (me)

FreeBSD/i386 (adelaide.lemis.com) (dcons)

login: root
Password:
Last login: Fri Aug 13 07:59:37 on ttyv0
Copyright (c) 1992-2004 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
    The Regents of the University of California. All rights reserved.

FreeBSD 5.2-CURRENT (ADELAIDE) #0: Fri Jan  2 16:29:05 CST 2004
You have mail.
erase ^H, kill ^U, intr ^C status ^T
Could not open a connection to your authentication agent.
=== root@adelaide (/dev/dcons) ~ 1 ->
```

00-c0-4f-32-26-e8-80-61 is the EUI64 address of the remote node, as determined from the output of *fwcontrol* above. When started in this manner, *dconschat* establishes a local tunnel connection from port `localhost:5556` to the remote debugger. You can also establish a console port connection with the `-C` option to the same invocation *dconschat*. See the *dconschat* manpage for further details.

Currently, it's still possible that this may not work. Instead, you may see:

```
# dconschat -br -G 5556 -t 00-c0-4f-32-26-e8-80-61
[dcons disconnected (get crom failed)]
```

crom is the abbreviation for *Control ROM*, and it's the purpose of the *dcons_crom* module. If it fails, it's probably due to incompatibilities in the version of *dcons_crom*. To solve the problem, specify the crom address manually using the *a* flag:

```
# dconschat -br -a 0x13d300 -G 5556 -t 00-c0-4f-32-26-e8-80-61
[dcons connected]
dcons_crom0: <dcons configuration ROM> on firewire0
dcons_crom0: bus_addr 0x13d300

FreeBSD/i386 (zaphod.lemis.com) (dcons)
(etc)
```

Get the crom address from the *dmesg* output from the machine to be debugged. As we have seen, it is:

```
dcons_crom0: bus_addr 0x13d300
```

The *dconschat* utility does not return control to the user. It displays error messages and console output for the remote system, and (as shown above) you can put a *getty* on the port */dev/dcons*, so it is a good idea to start it in its own window.

To start the *getty*, add the following line to */etc/ttys*:

```
dcons    "/usr/libexec/getty std.9600"    vt100    on    secure
```

If *dcons* was loaded after the system was booted, you'll also need to HUP *init*:

```
# kill -HUP 1
```

- Find the location of the kernel objects for the machine to be debugged. These need to be on a different machine. If you're using method recommended above, do the following on the machine to be debugged:

```
# ls -l /usr/src /usr/obj
lrwxr-xr-x  1 root  wheel   16 Jan  2  2004 /usr/obj -> /src/FreeBSD/obj
lrwxr-xr-x  1 root  wheel   25 Aug  1  16:55 /usr/src -> /src/FreeBSD/ADELAIDE/src
# ls -l /boot/kernel/kernel
-r-xr-xr-x  1 root  wheel  6034055 Jan  2  2004 /boot/kernel/kernel
```

On the debugging machine (assuming the same mount points),

```
# cd /src/FreeBSD/obj/src/FreeBSD/ADELAIDE/src/sys/ADELAIDE/
# ls -l kernel*
-rwxr-xr-x  1 grog  lemis   6034055 Jan  2  2004 kernel
-rwxr-xr-x  1 grog  lemis  31883941 Jan  2  2004 kernel.debug
# make gdbinit
grep -v '# XXX' /src/FreeBSD/ADELAIDE/src/sys/./tools/debugscripts/dot.gdbinit
| sed "s:MODPATH:/src/FreeBSD/obj/src/FreeBSD/ADELAIDE/src/sys/ADELAIDE/modules:" \
> .gdbinit
cp /src/FreeBSD/ADELAIDE/src/sys/./tools/debugscripts/gdbinit.kernel \
```

```

/src/FreeBSD/ADELAIDE/src/sys/./tools/debugscripts/gdbinit.vinum \
/src/FreeBSD/obj/src/FreeBSD/ADELAIDE/src/sys/ADELAIDE \
cp /src/FreeBSD/ADELAIDE/src/sys/./tools/debugscripts/gdbinit.i386
/src/FreeBSD/obj/src/FreeBSD/ADELAIDE/src/sys/ADELAIDE/gdbinit.machine \
# ls -l gdbinit* .gdbinit
-rw-r--r--  1 grog  lemis   3828 Aug 23 13:26 .gdbinit
-rw-r--r--  1 grog  lemis  10293 Aug 23 13:26 gdbinit.kernel
-rw-r--r--  1 grog  lemis   8913 Aug 23 13:26 gdbinit.machine
-rw-r--r--  1 grog  lemis  10018 Aug 23 13:26 gdbinit.vinum

```

The purpose of these entries is to:

1. First, find our object file. In this example, the directory `/usr/src` is a symbolic link pointing to an NFS mounted file system. The corresponding directory `/usr/obj` points several levels higher; effectively you need to add the path name of the symbolic link `/usr/src` to the end of the path name. After that, the directory with the kernel objects is in the subdirectory `sys` and has the name of the kernel. In more detail:
 - Object directory name: **/src/FreeBSD/obj**.
 - Source directory name, without the initial `/`: **src/FreeBSD/ADELAIDE/src**.
 - Directory **sys**.
 - Kernel name: **ADELAIDE**.

So the final name of the directory is **/src/FreeBSD/obj/src/FreeBSD/ADELAIDE/src/sys/ADELAIDE**.
 2. Ensure that we have the correct kernel. The file `kernel` should be exactly the same size, and normally it will be a few minutes older than the file `/boot/kernel/kernel` on the machine to be debugged. This difference represents the time between when the file was linked and when it was copied to `/boot`.
 3. Ensure that we have a file `kernel.debug`. It should have approximately the same modification timestamp as the kernel file, and it will be a lot bigger.
 4. Ensure that we have the debugging macros in place.
- Put the machine to be debugged into the debugger. On the console of the machine, you can enter:

```

# sync                                just in case
(press ctrl-alt-esc)
Stopped at      Debugger +0x54:  xchgl  %ebx,in_Debugger.0
db> gdb          select gdb mode
Next trap will enter GDB remote protocol mode
db> s            single step to get the next trap

```

This doesn't work from a console connected via `dconschat`.

At this point the system will appear to hang.

Alternatively, with FreeBSD, you can enter the following from any root shell:

```
# sysctl -w debug.kdb.enter=1
```


- Finally, on the debugging machine, establish connection:

```
# gdb kernel.debug
GNU gdb 6.1.1 [FreeBSD]
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-marcel-freebsd"...
Ready to go. Enter 'tr' to connect to the remote target
with /dev/cuaa0, 'tr /dev/cuaal' to connect to a different port
or 'trf portno' to connect to the remote target with the firewire
interface. portno defaults to 5556.

Type 'getsyms' after connection to load kld symbols.

If you're debugging a local system, you can use 'kldsyms' instead
to load the kld symbols. That's a less obnoxious interface.
(gdb) trf
0xc07c6bba in Debugger (msg=0x26 <Address 0x26 out of bounds>) at machine/atomic.h:263
263      machine/atomic.h: No such file or directory.
      in machine/atomic.h
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
warning: shared library handler failed to enable breakpoint
```

The *trf* macro assumes a connection on port 5556. If you want to use a different port (by changing the invocation of *dconschat* above), use the *tr* macro instead. For example, if you want to use port 4711, run *dconschat* like this:

```
# dconschat -br -G 4711 -t 0x000199000003622b
```

Then establish connection with:

```
(gdb) tr localhost:4711
0xc21bd378 in ?? ()
```

Non-cooperative debugging a live system with a remote firewire link

In addition to the conventional debugging via firewire described in the previous section, it is possible to debug a remote system without its cooperation, once an initial connection has been established. This corresponds to debugging a local machine using */dev/mem*. It can be very useful if a system crashes and the debugger no longer responds. To use this method, set the *sysctl* variables *hw.firewire.fwmem.eui64_hi* and *hw.firewire.fwmem.eui64_lo* to the upper and lower halves of the EUI64 ID of the remote system, respectively. From the previous example, the machine to be debugged shows:

```
# fwcontrol
2 devices (info_len=2)
node      EUI64      status
  1  00-c0-4f-32-26-e8-80-61      0
  0  43-4f-c0-00-1d-b0-a8-38      1
```

Enter:

```
# sysctl -w hw.firewire.fwmem.eui64_hi=0x434fc000
hw.firewire.fwmem.eui64_hi: 0 -> 1129299968
# sysctl -w hw.firewire.fwmem.eui64_lo=0x1db0a838
hw.firewire.fwmem.eui64_lo: 0 -> 498116664
```

Note that the variables must be explicitly stated in hexadecimal. After this, you can examine the state of the machine to be debugged with the following input:

```
# gdb -k kernel.debug /dev/fwmem0.0
GNU gdb 5.2.1 (FreeBSD)
(messages omitted)
Reading symbols from /boot/kernel/dcons.ko...done.
Loaded symbols for /boot/kernel/dcons.ko
Reading symbols from /boot/kernel/dcons_crom.ko...done.
Loaded symbols for /boot/kernel/dcons_crom.ko
#0 sched_switch (td=0xc0922fe0) at /usr/src/sys/kern/sched_4bsd.c:621
0xc21bd378 in ?? ()
```

In this case, it is not necessary to load the symbols explicitly. The remote system continues to run.

Currently this feature appears to be broken. Depending on the version of FreeBSD, it may be necessary to load the *mem* module to use it.

7

Debugging a processor dump

Probably the most common way of debugging is the processor *post-mortem dump*. After a panic you can save the contents of memory to disk. At boot time you can then save this image to a disk file and use a debugger to find out what has gone on.

Compared to on-line serial debugging, post-mortem debugging has the disadvantage that you can't continue with the execution when you have seen what you can from the present view of the system: it's dead. On the other hand, post-mortem debugging eliminates the long delays frequently associated with serial debugging.

There are two configuration steps to prepare for dumps:

- You must tell the kernel where to write the dump when it panics. By convention it's the swap partition, though theoretically you could dedicate a separate partition for this purpose. This would make sense if there were a post-mortem tool which could analyse the contents of swap: in this case you wouldn't want to overwrite it. Sadly, we currently don't have such a tool.

The dump partition needs to be the size of main memory with a little bit extra for a header. It needs to be in one piece: you can't spread a dump over multiple swap partitions, even if there's enough space.

We tell the system where to write the dump with the *dumpon* command:

```
# dumpon /dev/ad0s1b
```

- On reboot, the startup scripts run *savecore*, which checks the dump partition for a core dump and saves it to disk if it does. Obviously it needs to know where to put the resultant dump. By convention, it's */var/crash*. There's seldom a good reason to change that. If there's not enough space on the partition, it can be a symbolic link to somewhere where there is.

In */etc/rc.conf*, set:

```
dumpdev=/dev/ad0b
```

Saving the dump

When you reboot after a panic, *savecore* saves the dump to disk. By convention they're stored in */var/crash*. There you might see:

```
# ls -l
total 661
-rw-r--r--  1 root  wheel           3 Sep 20 11:12 bounds
-rw-r--r--  1 root  wheel    3464574 Sep 16 06:13 kernel.10
-rw-r--r--  1 root  wheel    3589033 Sep 18 09:08 kernel.11
-rw-r--r--  1 root  wheel    3589033 Sep 19 03:13 kernel.12
-rw-r--r--  1 root  wheel    3589033 Sep 20 10:50 kernel.13
-rw-r--r--  1 root  wheel    3589033 Sep 20 11:03 kernel.14
-rw-r--r--  1 root  wheel    3589033 Sep 20 11:12 kernel.15
lrwxr-xr-x  1 root  wheel           61 Sep 20 16:13 kernel.debug ->
    /src/FreeBSD/4.4-RELEASE/src/sys/compile/ECHUNGA/kernel.debug
-rw-r--r--  1 root  wheel           5 Sep 17  1999 minfree
-rw-----  1 root  wheel   134152192 Sep 18 09:08 vmcore.11
-rw-----  1 root  wheel   134152192 Sep 19 03:13 vmcore.12
-rw-----  1 root  wheel   134152192 Sep 20 10:50 vmcore.13
-rw-----  1 root  wheel   134152192 Sep 20 11:03 vmcore.14
-rw-----  1 root  wheel   134152192 Sep 20 11:12 vmcore.15
```

These files have the following purpose:

- *vmcore.11* and friends are the individual core images. This directory contains five dumps, numbered 11 to 15.
- *kernel.11* and friends are corresponding copies of the kernel on reboot. Normally they're the kernel which crashed, but it's possible that they might not be. For example, you might have replaced the kernel in single-user mode after the crash and before rebooting to multi-user mode. They're also normally stripped, so they're not much use for debugging. Recent versions of FreeBSD no longer include this file; see the next entry.
- Recent versions of FreeBSD include files with names like *info.15*. As the name suggests, the file contains information about the dump. For example:

```
Good dump found on device /dev/ad0s4b
Architecture: i386
Architecture version: 1
Dump length: 134217728B (128 MB)
Blocksize: 512
Dumptime: Thu Aug  7 11:01:23 2003
Hostname: zaphod.lemis.com
Versionstring: FreeBSD 5.1-BETA #7: Tue Jun  3 18:10:59 CST 2003
grog@zaphod.lemis.com:/src/FreeBSD/obj/src/FreeBSD/ZAPHOD/src/sys/ZAPHOD
Panicstring: from debugger
Bounds: 0
```

- *kernel.debug* is a symbolic link to a real debug kernel in the kernel build directory. This is one way to do it, and it has the advantage that *gdb* then finds the source files with no further problem. If you're debugging multiple kernels, there's no reason why you shouldn't remove the saved kernels and create symlinks with names like *kernel.11* etc.

- *minfree* specifies the minimum amount of space to leave on the file system after saving the dump. This avoids running out of space on the file system.
- *bounds* is a rather misleading name: it contains the number of the next kernel dump, followed by a `\n` character.

Analyzing the dump

When you start kernel *gdb* against a processor dump, you'll see something like this:

```
# gdb -k kernel.debug vmcore.11
panicstr: general protection fault
panic messages:
---
Fatal trap 9: general protection fault while in kernel mode
instruction pointer   = 0x8:0xc01c434b
stack pointer        = 0x10:0xc99f8d0c
frame pointer        = 0x10:0xc99f8d28
code segment         = base 0x0, limit 0xffffffff, type 0x1b
                     = DPL 0, pres 1, def32 1, gran 1
processor eflags      = interrupt enabled, resume, IOPL = 0
current process       = 2638 (find)
interrupt mask        = net tty bio cam
trap number          = 9
panic: general protection fault

syncing disks... 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
giving up on 6 buffers
Uptime: 17h53m13s
dumping to dev #ad/1, offset 786560
dump ata0: resetting devices .. done

---
#0  dumpsys () at ../../kern/kern_shutdown.c:473
473      if (dumping++) {
(kgdb)
```

With the exception of the last three lines, this is the same as what the system prints on the screen when it panics. The last three lines show what the processor was executing at the time of the dump. This information is of marginal importance: it shows the functions which create the core dump. They work, or you wouldn't have the dump. To find out what really happened, start with a stack backtrace:

```
(kgdb) bt
#0  dumpsys () at ../../kern/kern_shutdown.c:473
#1  0xc01c88bf in boot (howto=256) at ../../kern/kern_shutdown.c:313
#2  0xc01c8ca5 in panic (fmt=0xc03a8cac "%s") at ../../kern/kern_shutdown.c:581
#3  0xc033ab03 in trap_fatal (frame=0xc99f8ccc, eva=0)
    at ../../i386/i386/trap.c:956
#4  0xc033a4ba in trap (frame={tf_fs = 16, tf_es = 16, tf_ds = 16,
    tf_ei = -1069794208, tf_esi = -1069630360, tf_ebp = -912290520,
    tf_isp = -912290568, tf_ebx = -1069794208, tf_edx = 10, tf_ecx = 10,
    tf_eax = -1, tf_trapno = 9, tf_err = 0, tf_eip = -1071889589, tf_cs = 8,
    tf_eflags = 66182, tf_esp = 1024, tf_ss = 6864992})
    at ../../i386/i386/trap.c:618
#5  0xc01c434b in malloc (size=1024, type=0xc03c3c60, flags=0)
    at ../../kern/kern_malloc.c:233
#6  0xc01f015c in allocbuf (bp=0xc3a6f7cc, size=1024)
    at ../../kern/vfs_bio.c:2380
#7  0xc01effa6 in getblk (vp=0xc9642f00, blkno=0, size=1024, slpflag=0,
    slptimeo=0) at ../../kern/vfs_bio.c:2271
```

```

#8 0xc01eded2 in bread (vp=0xc9642f00, blkno=0, size=1024, cred=0x0,
    bpp=0xc99f8e3c) at ../../kern/vfs_bio.c:504
#9 0xc02d0634 in ffs_read (ap=0xc99f8ea0) at ../../ufs/ufs/ufs_readwrite.c:273
#10 0xc02d734e in ufs_readdir (ap=0xc99f8ef0) at vnode_if.h:334
#11 0xc02d7cd1 in ufs_vnoperate (ap=0xc99f8ef0)
    at ../../ufs/ufs/ufs_vnops.c:2382
#12 0xc01fbc3b in getdirentries (p=0xc9a53ac0, uap=0xc99f8f80)
    at vnode_if.h:769
#13 0xc033adb5 in syscall2 (frame={tf_fs = 47, tf_es = 47, tf_ds = 47,
    tf_ei = 134567680, tf_esi = 134554336, tf_ebp = -1077937404,
    tf_isp = -912289836, tf_ebx = 672064612, tf_edx = 134554336,
    tf_ecx = 672137600, tf_eax = 196, tf_trapno = 7, tf_err = 2,
    tf_eip = 671767876, tf_cs = 31, tf_eflags = 582, tf_esp = -1077937448,
    tf_ss = 47}) at ../../i386/i386/trap.c:1155
#14 0xc032b825 in Xint0x80_syscall ()
#15 0x280aleee in ?? ()
#16 0x280a173a in ?? ()
#17 0x804969e in ?? ()
#18 0x804b550 in ?? ()
#19 0x804935d in ?? ()
(kgdb)

```

The most important stack frame is the one below trap. Select it with the frame command, which you can abbreviate to `f`, and list the code with `list` (or `l`):

```

(kgdb) f 5
#5 0xc01c434b in malloc (size=1024, type=0xc03c3c60, flags=0)
    at ../../kern/kern_malloc.c:233
233         va = kbp->kb_next;
(kgdb) l
228     }
229     freep->next = savedlist;
230     if (kbp->kb_last == NULL)
231         kbp->kb_last = (caddr_t)freep;
232 }
233 va = kbp->kb_next;
234 kbp->kb_next = ((struct freelist *)va)->next;
235 #ifdef INVARIANTS
236     freep = (struct freelist *)va;
237     savedtype = (const char *) freep->type->ks_shortdesc;
(kgdb)

```

You might want to look at the local (automatic) variables. Use `info local`, which you can abbreviate to `i loc`:

```

(kgdb) i loc
type = (struct malloc_type *) 0xc03c3c60
kbp = (struct kmembuckets *) 0xc03ebc68
kup = (struct kmemusage *) 0x0
freep = (struct freelist *) 0x0
indx = 10
npg = -1071714292
allocsize = -1069794208
s = 6864992
va = 0xffffffff <Address 0xffffffff out of bounds>
cp = 0x0
savedlist = 0x0
ksp = (struct malloc_type *) 0xffffffff
(kgdb)

```

As `gdb` shows, the line where the problem occurs is 233:

```

233         va = kbp->kb_next;

```

Look at the structure `kbp`:

```
(kgdb) p *kbp
$2 = {
  kb_next = 0xffffffff <Address 0xffffffff out of bounds>,
  kb_last = 0xc1a31000 "",
  kb_calls = 83299,
  kb_total = 1164,
  kb_elmpercl = 4,
  kb_totalfree = 178,
  kb_highwat = 20,
  kb_couldfree = 3812
}
```

With this relatively mechanical method, we have found that the crash was in `malloc`. `malloc` gets called many times every second. There's every reason to believe that it works correctly, so it's probably not a bug in `malloc`. More likely it's the result of a client of `malloc` either writing beyond the end of the allocated area, or writing to it after calling `free`.

Finding this kind of problem is particularly difficult: there's no reason to believe that the process or function which trips over this problem has anything to do with the process or function which caused it. In the following sections we'll look at variants on the problem.

A panic in Vinum

It's more interesting to look at bugs which happen when developing code. I wrote *Vinum*, so I have a plethora of bugs to look at. In the following sections we'll look at some of them.

In the first example, our Vinum test system panics during boot:

```
Mounting root from ufs:/dev/ad0s2a
swapon: adding /dev/ad0s4b as swap device
Automatic boot in progress...
/dev/ad0s2a: 38440 files, 381933 used, 1165992 free (21752 frags, 143030 blocks, 1.4%
fragmentation)
/dev/ad0s3a: FILESYSTEM CLEAN; SKIPPING CHECKS
/dev/ad0s3a: clean, 1653026 free (46890 frags, 200767 blocks, 1.5% fragmentation)
/dev/ad0s1a: FILESYSTEM CLEAN; SKIPPING CHECKS
/dev/ad0s1a: clean, 181000 free (5352 frags, 21956 blocks, 0.3% fragmentation)
Memory modified at 0xc199657c after free 0xc1996000(2044): deafc0de
panic: Most recently used by devbuf
```

This system is set up with remote debugging, so next we see:

```
Debugger("panic")
Stopped at Debugger+0x54 xchgl %ebx, in_Debugger.0
db> gdb
Next trap will enter GDB remote protocol mode
db> s
(nothing more appears here)
```

At this point, the system is trying to access the remote debugger. On the system connected to the other end of the debugger cable, we enter:

```
# cd /src/FreeBSD/obj/src/FreeBSD/ZAPHOD/src/sys/GENERIC
# gdb
...
Ready to go. Enter 'tr' to connect to the remote target
with /dev/cuaa0, 'tr /dev/cuaa1' to connect to a different port
or 'trf portno' to connect to the remote target with the firewire
interface. portno defaults to 5556.
```

Type 'getsyms' after connection to load kld symbols.

If you're debugging a local system, you can use 'kldsyms' instead to load the kld symbols. That's a less obnoxious interface.

```
(gdb) tr
Debugger (msg=0x12 <Address 0x12 out of bounds>) at /src/FreeBSD/ZAPHOD/src
/sys/i386/i386/db_interface.c:330
330      }
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
warning: shared library handler failed to enable breakpoint
```

The messages above come from this particular version of the kernel. In a development kernel, you're likely to see things like this. Unless they stop you debugging, they're probably not worth worrying about.

The "Ready to go" messages come from the debugging macros created by make gdbinit as described on page 36. We use the getsyms macro to load the symbols:

```
(gdb) getsyms
add symbol table from file "/src/FreeBSD/obj/src/FreeBSD/ZAPHOD/src/sys/GEN
ERIC/modules/src/FreeBSD/ZAPHOD/src/sys/modules/vinum/vinum.ko.debug" at
.text_addr = 0xc06a4920
.data_addr = 0xc06b5000
.bss_addr = 0xc06b5400
```

Traditionally, the first thing you do with a panic is to see where it happens. Do that with the backtrace (bt) command:

```
(gdb) bt
#0  Debugger (msg=0x12 <Address 0x12 out of bounds>)
   at /src/FreeBSD/ZAPHOD/src/sys/i386/i386/db_interface.c:330
#1  0xc031294b in panic (fmt=0x1 <Address 0x1 out of bounds>)
   at /src/FreeBSD/ZAPHOD/src/sys/kern/kern_shutdown.c:527
#2  0xc0462137 in mtrash_ctor (mem=0xc1996000, size=0x20, arg=0x0)
   at /src/FreeBSD/ZAPHOD/src/sys/vm/uma_dbg.c:138
#3  0xc04609ff in uma_zalloc_arg (zone=0xc0b65240, udata=0x0, flags=0x2)
   at /src/FreeBSD/ZAPHOD/src/sys/vm/uma_core.c:1366
#4  0xc0307614 in malloc (size=0xc0b65240, type=0xc0557300, flags=0x2) at uma.h:229
#5  0xc035a1ff in allocbuf (bp=0xc3f0a420, size=0x800) at /src/FreeBSD/ZAPH
OD/src/sys/kern/vfs_bio.c:2723
#6  0xc0359f0c in getblk (vp=0xc1a1936c, blkno=0x0, size=0x800, slpflag=0x0, slptimeo
=0x0, flags=0x0)
   at /src/FreeBSD/ZAPHOD/src/sys/kern/vfs_bio.c:2606
#7  0xc0356732 in breadn (vp=0xc1a1936c, blkno=0x2000000012, size=0x12, rablkno=0x0,
rabsz=0x0, cnt=0x0, cred=0x0,
   bpp=0x12) at /src/FreeBSD/ZAPHOD/src/sys/kern/vfs_bio.c:701
#8  0xc03566dc in bread (vp=0x12, blkno=0x2000000012, size=0x12, cred=0x12, bpp=0x12)
   at /src/FreeBSD/ZAPHOD/src/sys/kern/vfs_bio.c:683
#9  0xc043586f in ffs_blkatoff (vp=0xc1a1936c, offset=0x0, res=0x0, bpp=0xcccb3988)
   at /src/FreeBSD/ZAPHOD/src/sys/ufs/ufs/ufs_subr.c:91
#10 0xc043f5a7 in ufs_lookup (ap=0xcccb3ab8) at /src/FreeBSD/ZAPHOD/src/sys
/ufs/ufs/ufs_lookup.c:266
#11 0xc0446dd8 in ufs_vnoperate (ap=0x0) at /src/FreeBSD/ZAPHOD/src/sys/ufs
/ufs/ufs_vnops.c:2787
#12 0xc035d19c in vfs_cache_lookup (ap=0x12) at vnode_if.h:82
#13 0xc0446dd8 in ufs_vnoperate (ap=0x0) at /src/FreeBSD/ZAPHOD/src/sys/ufs
/ufs/ufs_vnops.c:2787
#14 0xc0361e92 in lookup (ndp=0xcccb3c24) at vnode_if.h:52
#15 0xc036188e in namei (ndp=0xcccb3c24) at /src/FreeBSD/ZAPHOD/src/sys/ker
n/vfs_lookup.c:181
#16 0xc036ee32 in lstat (td=0xc199b980, uap=0xcccb3d10)
   at /src/FreeBSD/ZAPHOD/src/sys/kern/vfs_syscalls.c:1719
#17 0xc0497d7e in syscall (frame=
   {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf edi = 0xbfbffda8, tf_esi = 0xbfbf
```



```

fda0, tf_ebp = 0xbfbffd48, tf_isp = 0xcccb3d74, tf_ebx = 0xbfbffe49, tf_edx = 0xffffffff,
tf_ecx = 0x2, tf_eax = 0xbe, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0x804ac0b,
tf_cs = 0x1f, tf_eflags = 0x282, tf_esp = 0xbfbffc0, tf_ss = 0x2f})
    at /src/FreeBSD/ZAPHOD/src/sys/i386/i386/trap.c:1025
#18 0xc048724d in Xint0x80_syscall () at {standard input}:138
#19 0x080483b6 in ?? ()
#20 0x08048145 in ?? ()

```

In this case, about all we can see is that the backtrace has nothing to do with Vinum. The first frame is always in Debugger, and since this is a panic, the second frame is panic. The third frame is the frame which called panic. We can look at it in more detail:

```

(gdb) f 2                                select frame 2
#2  0xc0462137 in mtrash_ctor (mem=0xc1996000, size=0x20, arg=0x0)
    at /src/FreeBSD/ZAPHOD/src/sys/vm/uma_dbg.c:138
138      panic("Most recently used by %s\n", (*ksp == NULL)?
(gdb) l 1                                list code
133
134      for (p = mem; cnt > 0; cnt--, p++)
135          if (*p != uma_junk) {
136              printf("Memory modified at %p after free %p(%d): %x\n",
137                  p, mem, size, *p);
138              panic("Most recently used by %s\n", (*ksp == NULL)?
139                  "none" : (*ksp)->ks_shortdesc);
140          }
141      }
142

```

Looking for the definition of `uma_junk` leads us to:

```

51      static const u_int32_t uma_junk = 0xdead0de;

```

This code is part of the `INVARIANTS` code to check memory allocations. When `INVARIANTS` are set, `free` writes `uma_junk` (`0xdead0de`) to every word of the freed memory. `malloc` then checks if it's still that way when it's taken off the free list. If anything is changed in the meantime, it will show up with this panic. In our example, one word has changed from `0xdead0de` to `0xdeafc0de`. The obvious question is where. Looking at the local variables, we see:

```

(gdb) i loc                                show local variables
ksp = (struct malloc_type **)
0xc19967fc p = (u_int32_t *) 0x0 cnt = 0x12

```

The value of the pointer `p` is important. But how can it be 0? We just printed the message of line 136:

```

Memory modified at 0xc199657c after free 0xc1996000(2044): deafc0de

```

This is a problem with the optimizer. On line 138, the call to `panic`, the pointer `p` is no longer needed, and the optimizer has used the register for something else. This is one of the reasons why the message prints out the value of `p`.

So where did the problem happen? We're hacking on Vinum, so it's reasonable to assume that it's related to Vinum, and we know from the panic message and the backtrace that it's related to memory allocation. When compiled with the `VINUMDEBUG` option, Vinum includes a number of kernel debug tools. There are also some macros in `/usr/src/tools/debugtools/`. Two are *mem-*

info, which keeps track of currently allocated memory, and *fi nfo*, which keeps track of recently freed memory areas. They're only enabled on request—see the debug subcommand of *vinum(8)* for more details. Here we have enabled them before booting, and we see:

```
(gdb) meminfo
Block      Time          Sequence      size  address      line      file
0          18.987686         3         3136 0xc1958000    160      vinum.c
1          19.491101         7          256 0xc1991d00    117      vinumio.c
2          19.504050         9          256 0xc1991c00    117      vinumio.c
3          19.507847        11          256 0xc1991b00    117      vinumio.c
4          19.523213        13          256 0xc1991a00    117      vinumio.c
5          19.530848        16          256 0xc1991900    117      vinumio.c
6          19.537997        18          256 0xc1991800    117      vinumio.c
7          19.565260        31        2048 0xc1995800    902      vinumio.c
8          19.599982        32       1536 0xc1995000    841      vinumconfig.c
9          19.600115        33          16 0xc19885a0    768      vinumconfig.c
10         19.600170        34          16 0xc19885c0    768      vinumconfig.c
11         19.600215        35          16 0xc19885e0    768      vinumconfig.c
12         19.600263        36          16 0xc1988610    768      vinumconfig.c
13         19.600307        37          16 0xc1988620    768      vinumconfig.c
14         19.600368        38       3072 0xc1954000   1450      vinumconfig.c
15         19.600408        39          16 0xc18d93a0    768      vinumconfig.c
16         19.600453        40          16 0xc1988600    768      vinumconfig.c
17         19.600508        41       3072 0xc1953000   1450      vinumconfig.c
18         19.600546        42          16 0xc1988690    768      vinumconfig.c
19         19.600601        43       3072 0xc1952000   1450      vinumconfig.c
20         19.601170        44       3072 0xc1951000    468      vinumconfig.c
21         19.637070        45      3520 0xc1950000    763      vinumconfig.c
22         19.637122        46          16 0xc1988640    768      vinumconfig.c
23         19.637145        47          16 0xc1988670    768      vinumconfig.c
24         19.637166        48          16 0xc19886a0    768      vinumconfig.c
25         19.637186        49          16 0xc19886f0    768      vinumconfig.c
26         19.637207        50          16 0xc19886b0    768      vinumconfig.c
27         19.637227        51          16 0xc1988710    768      vinumconfig.c
28         19.637247        52          16 0xc1988730    768      vinumconfig.c
29         19.637268        53          16 0xc1988750    768      vinumconfig.c
30         19.673860        54          16 0xc1988780    768      vinumconfig.c
31         19.673884        55          16 0xc19882d0    768      vinumconfig.c
32         19.673905        56          16 0xc19887d0    768      vinumconfig.c
33         19.673925        57          16 0xc19887a0    768      vinumconfig.c
34         19.673946        58          16 0xc1988800    768      vinumconfig.c
35         19.673966        59          16 0xc1988810    768      vinumconfig.c
36         19.673988        60          16 0xc19887e0    768      vinumconfig.c
37         19.674009        61          16 0xc1988840    768      vinumconfig.c
38         19.710319        62          16 0xc1988860    768      vinumconfig.c
39         19.710343        63          16 0xc18d9ab0    768      vinumconfig.c
40         19.710364        64          16 0xc18d95c0    768      vinumconfig.c
41         19.710385        65          16 0xc18d9e40    768      vinumconfig.c
42         19.710406        66          16 0xc0b877d0    768      vinumconfig.c
43         19.710427        67          16 0xc18d99c0    768      vinumconfig.c
44         19.710448        68          16 0xc18d9b40    768      vinumconfig.c
45         19.710469        69          16 0xc19888c0    768      vinumconfig.c
46         19.740424        70          16 0xc19888e0    768      vinumconfig.c
47         19.740448        71          16 0xc18d9d00    768      vinumconfig.c
48         19.740469        72          16 0xc1988100    768      vinumconfig.c
49         19.740490        73          16 0xc18d9eb0    768      vinumconfig.c
50         19.740511        74          16 0xc1988190    768      vinumconfig.c
51         19.740532        75          16 0xc18d9a30    768      vinumconfig.c
52         19.740554        76          16 0xc1988580    768      vinumconfig.c
53         19.740576        77          16 0xc1988560    768      vinumconfig.c
54         19.778006        78          16 0xc1988570    768      vinumconfig.c
55         19.778031        79          16 0xc18d9360    768      vinumconfig.c
56         19.778052        80          16 0xc1988500    768      vinumconfig.c
57         19.778074        81          16 0xc19884c0    768      vinumconfig.c
58         19.778095        82          16 0xc1988520    768      vinumconfig.c
59         19.778116        83          16 0xc19884e0    768      vinumconfig.c
60         19.778138        84          16 0xc19884b0    768      vinumconfig.c
61         19.778159        85          16 0xc19884d0    768      vinumconfig.c
62         19.780088        86          16 0xc19884a0    224      vinumdaemon.c
```

| (gdb) finfo | | <i>look at already freed memory</i> | | | | | |
|--------------------|-----------|-------------------------------------|--------|------------|------|---------------|--|
| Block | Time | Sequence | size | address | line | file | |
| 0 | 19.501059 | 8 | 512 | 0xc1975c00 | 318 | vinumio.c | |
| 1 | 19.505499 | 10 | 512 | 0xc1975e00 | 318 | vinumio.c | |
| 2 | 19.519560 | 12 | 512 | 0xc197ac00 | 318 | vinumio.c | |
| 3 | 19.527459 | 14 | 512 | 0xc18dac00 | 318 | vinumio.c | |
| 4 | 19.527834 | 0 | 1024 | 0xc1981c00 | 468 | vinumconfig.c | |
| 5 | 19.534994 | 17 | 512 | 0xc197a400 | 318 | vinumio.c | |
| 6 | 19.542243 | 19 | 512 | 0xc197a000 | 318 | vinumio.c | |
| 7 | 19.543044 | 21 | 512 | 0xc18dac00 | 318 | vinumio.c | |
| 8 | 19.546529 | 20 | 256 | 0xc1991700 | 596 | vinumconfig.c | |
| 9 | 19.547444 | 23 | 512 | 0xc1975e00 | 318 | vinumio.c | |
| 10 | 19.550881 | 22 | 256 | 0xc1991400 | 596 | vinumconfig.c | |
| 11 | 19.551790 | 25 | 512 | 0xc1975c00 | 318 | vinumio.c | |
| 12 | 19.555305 | 24 | 256 | 0xc1991100 | 596 | vinumconfig.c | |
| 13 | 19.556213 | 27 | 512 | 0xc1975c00 | 318 | vinumio.c | |
| 14 | 19.559655 | 26 | 256 | 0xc198dd00 | 596 | vinumconfig.c | |
| 15 | 19.560516 | 29 | 512 | 0xc1975c00 | 318 | vinumio.c | |
| 16 | 19.564290 | 28 | 256 | 0xc198da00 | 596 | vinumconfig.c | |
| 17 | 19.564687 | 5 | 1024 | 0xc197ec00 | 882 | vinumio.c | |
| 18 | 19.600004 | 1 | 768 | 0xc1981400 | 841 | vinumconfig.c | |
| 19 | 19.601196 | 15 | 2048 | 0xc1996000 | 468 | vinumconfig.c | |
| 20 | 19.637102 | 2 | 1760 | 0xc18e5000 | 763 | vinumconfig.c | |
| 21 | 19.779320 | 30 | 131072 | 0xc1998000 | 966 | vinumio.c | |
| 22 | 19.779366 | 6 | 1024 | 0xc197f000 | 967 | vinumio.c | |
| 23 | 19.780113 | 4 | 28 | 0xc18d68a0 | 974 | vinumio.c | |

The time in the second column is in `time_t` format. Normally it would be a very large number, the number of seconds and microseconds since 1 January 1970 0:0 UTC, but at this point during booting the system doesn't know the time yet, and it is in fact the time since starting the kernel.

Looking at the free info table, it's clear that yes, indeed, the block starting at `0xc1996000` memory was allocated to Vinum until time `19.601196`:

```
19      19.601196      15      2048 0xc1996000      468      vinumconfig.c
```

It looks as if something was left pointing into the block of memory after it's freed. The obvious thing to do is to check what it was used for. Looking at line 468 of *vinumconfi g.c*, we see:

```
if (driveno >= vinum_conf.drives_allocated) /* we've used all our allocation */
    EXPAND(DRIVE, struct drive, vinum_conf.drives_allocated, INITIAL_DRIVES);

/* got a drive entry. Make it pretty */
drive = &DRIVE[driveno];
```

The `EXPAND` macro is effectively the same as `realloc`. It allocates `INITIAL_DRIVES * sizeof (struct drive)` more memory and copies the old data to it, then frees the old data; that's the `free` call we saw. In the *meminfo* output, we see at time `19.601170` ($26\ \mu\text{s}$ earlier) an allocation of 3072 bytes, which is the replacement area:

```
20      19.601170      44      3072 0xc1951000      468      vinumconfig.c
```

Looking at the code, though, you'll see that the pointer to the drive is not allocated until after the call to `EXPAND`. So maybe it's from a function which calls it.

How do we find which functions call it? We could go through manually and check, but that can rapidly become a problem. It could be worthwhile finding out what has changed. The word which has been modified has only a single bit changed: `0xdead0de` became `0xdeafc0de`, so we're probably looking at a logical bit set operation which *ors* `0x20000` with the previous val-


```
(gdb) p &((struct drive *) 0xc1996000)[5].writes
$4 = (u_int64_t *) 0xc19965b0
(gdb) p &((struct drive *) 0xc1996000)[5].state
$5 = (enum drivestate *) 0xc1996578
(gdb) p &((struct drive *) 0xc1996000)[5].flags
$6 = (int *) 0xc199657c
(gdb) p ((struct drive *) 0xc1996000)[5].flags
$7 = 0xdeafc0de
```

So the field is `flags`. Looking back shows that yes, this value is shown in hex, so we didn't need to do this search. In fact, though, after a few hours of this sort of stuff, it's easier to do the search than run through output which may or may not contain the information you're looking for.

It makes sense that the problem is in `flags`: it's a collection of bits, so setting or resetting individual bits is a fairly typical access mode. What's `0x20000`? The bits are defined in *vinumobj.h*:

```
/*
 * Flags for all objects. Most of them only apply
 * to specific objects, but we currently have
 * space for all in any 32 bit flags word.
 */
enum objflags {
    VF_LOCKED = 1,           /* somebody has locked access to this object */
    VF_LOCKING = 2,         /* we want access to this object */
    VF_OPEN = 4,            /* object has openers */
    VF_WRITETHROUGH = 8,    /* volume: write through */
    VF_INITED = 0x10,       /* unit has been initialized */
    VF_WLABEL = 0x20,       /* label area is writable */
    VF_LABELLING = 0x40,    /* unit is currently being labelled */
    VF_WANTED = 0x80,       /* someone is waiting to obtain a lock */
    VF_RAW = 0x100,        /* raw volume (no file system) */
    VF_LOADED = 0x200,      /* module is loaded */
    VF_CONFIGURING = 0x400, /* somebody is changing the config */
    VF_WILL_CONFIGURE = 0x800, /* somebody wants to change the config */
    VF_CONFIG_INCOMPLETE = 0x1000, /* haven't finished changing the config */
    VF_CONFIG_SETUPSTATE = 0x2000, /* set a volume up if all plexes are empty */
    VF_READING_CONFIG = 0x4000, /* we're reading config database from disk */
    VF_FORCECONFIG = 0x8000, /* configure drives even with different names */
    VF_NEWBORN = 0x10000,   /* for objects: we've just created it */
    VF_CONFIGURED = 0x20000, /* for drives: we read the config */
    VF_STOPPING = 0x40000, /* for vinum_conf: stop on last close */
    VF_DAEMONOPEN = 0x80000, /* the daemon has us open (only superdev) */
    VF_CREATED = 0x100000, /* for volumes: freshly created, more than new */
    VF_HOTSPARE = 0x200000, /* for drives: use as hot spare */
    VF_RETRYERRORS = 0x400000, /* don't down subdisks on I/O errors */
    VF_HASDEBUG = 0x800000, /* set if we support debug */
};
```

So our bit is `VF_CONFIGURED`. Where does it get set?

```
$ grep -n VF_CONFIGURED *.c
vinumio.c:843:             else if (drive->flags & VF_CONFIGURED)
vinumio.c:868:             else if (drive->flags & VF_CONFIGURED)
vinumio.c:963:     drive->flags |= VF_CONFIGURED;
```

The last line is the only place which modifies the flags. Line 963 of *vinumio.c* is in the function `vinum_scandisk`. This function first builds up the drive list, a drive at a time, paying great attention to not assign any pointers. Once the list is complete and not going to change, it goes through a second loop and reads the configuration from the drives. Here's the second loop:

```
for (driveno = 0; driveno < gooddrives; driveno++) { /* now include the config */
```

```

drive = &DRIVE[drivelist[driveno]]; /* point to the drive */

if (firsttime && (driveno == 0)) /* we've never configured before, */
    log(LOG_INFO, "vinum: reading configuration from %s\n", drive->devicename);
else
    log(LOG_INFO, "vinum: updating configuration from %s\n", drive->devicename);

if (drive->state == drive_up)
    /* Read in both copies of the configuration information */
    error = read_drive(drive, config_text, MAXCONFIG * 2, VINUM_CONFIG_OFFSET);
else {
    error = EIO;
    printf("vinum_scandisk: %s is %s\n",
           drive->devicename, drive_state(drive->state));
}

if (error != 0) {
    log(LOG_ERR, "vinum: Can't read device %s, error %d\n", drive->devicename, error);
    free_drive(drive); /* give it back */
    status = error;
}
/*
 * At this point, check that the two copies
 * are the same, and do something useful if
 * not. In particular, consider which is
 * newer, and what this means for the
 * integrity of the data on the drive.
 */
else {
    vinum_conf.drives_used++; /* another drive in use */
    /* Parse the configuration, and add it to the global configuration */
    for (cptr = config_text; *cptr != '\0';) { /* love this style(9) */
        volatile int parse_status; /* return value from parse_config */

        for (eptr = config_line; (*cptr != '\n') && (*cptr != '\0');)
            *eptr++ = *cptr++; /* until the end of the line */
        *eptr = '\0'; /* and delimit */
        if (setjmp(command_fail) == 0) { /* come back here on error and continue */
            /* parse the config line */
            parse_status = parse_config(config_line, &keyword_set, 1);
            if (parse_status < 0) { /* error in config */
                /*
                 * This config should have been parsed
                 * in user space. If we run into
                 * problems here, something serious is
                 * afoot. Complain and let the user
                 * snarf the config to see what's
                 * wrong.
                 */
                log(LOG_ERR,
                    "vinum: Config error on %s, aborting integration\n",
                    drive->devicename);
                free_drive(drive); /* give it back */
                status = EINVAL;
            }
        }
        while (*cptr == '\n')
            cptr++; /* skip to next line */
    }
}
drive->flags |= VF_CONFIGURED; /* this drive's configuration is complete */
}

```

There's nothing there which reaches out and grabs you. You could read the code and find out what's going on (probably the better choice in this particular case), but you could also find out where `get_empty_drive` is being called from. To do this, reboot the machine and go into *ddb* before Vinum starts. To do this, interrupt the boot sequence and enter:

OK **boot -d**

As soon as the system has enough context, it goes into the debugger. Look for a place to put a breakpoint:

```
(gdb) 1 get_empty_drive
452 }
453
454 /* Get an empty drive entry from the drive table */
455 int
456 get_empty_drive(void)
457 {
458     int driveno;
459     struct drive *drive;
460
461     /* first see if we have one which has been deallocated */
462     for (driveno = 0; driveno < vinum_conf.drives_allocated; driveno++) {
463         if (DRIVE[driveno].state == drive_unallocated) /* bingo */
464             break;
465     }
466
467     if (driveno >= vinum_conf.drives_allocated) /* we've used all our allocation */
468         EXPAND(DRIVE, struct drive, vinum_conf.drives_allocated, INITIAL_DRIVES);
469
470     /* got a drive entry. Make it pretty */
471     drive = &DRIVE[driveno];
```

This function gets called many times. In FreeBSD it's 35 times for every disk (four slices and compatibility slice, seven partitions per slice). This code is meticulously careful not to assign any pointers:

```
for (slice = 1; slice < 5; slice++)
    for (part = 'a'; part < 'i'; part++) {
        if (part != 'c') { /* don't do the c partition */
            snprintf(np,
                partnamelen,
                "s%d%c",
                slice,
                part);
            drive = check_drive(partname); /* try to open it */
            if (drive) { /* got something, */
                if (drive->flags & VF_CONFIGURED) /* already read this config, */
                    log(LOG_WARNING,
                        "vinum: already read config from %s\n", /* say so */
                        drive->label.name);
                else {
                    if (gooddrives == drives) /* ran out of entries */
                        EXPAND(drivelist, int, drives, drives); /* double the size */
                    drivelist[gooddrives] = drive->driveno; /* keep the drive index */
                    drive->flags &= ~VF_NEWBORN; /* which is no longer newly born */
                    gooddrives++;
                }
            }
        }
    }
}
```

After lots of code reading, it's still not clear how this could cause the kind of corruption we're looking for. The problem is obviously related to expanding the table, so the obvious place to put the breakpoint on the macro EXPAND on line 468:

```
(gdb) b 468 set a breakpoint on the EXPAND call
Breakpoint 1 at 0xc06a600f: file /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinum
config.c, line 468.
(gdb) c
```

Continuing.

```

Program received signal SIGTRAP, Trace/breakpoint trap.
Debugger (msg=0x12 <Address 0x12 out of bounds>) at atomic.h:260
260     ATOMIC_STORE_LOAD(int, "cmpxchgl %0,%1", "xchgl %1,%0");
(gdb) bt
#0  0xc06a6010 in get_empty_drive () at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinumconfig.c:468
468     EXPAND(DRIVE, struct drive, vinum_conf.drives_allocated, INITIAL_DRIVES);
(gdb) bt
#0  0xc06a6010 in get_empty_drive () at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinumconfig.c:468
#1  0xc06a60f9 in find_drive (name=0xc199581a "virtual", create=0x1)
    at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinumconfig.c:505
#2  0xc06a7217 in config_subdisk (update=0x1) at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinumconfig.c:1157
#3  0xc06a7ebe in parse_config (cptr=0x700 <Address 0x700 out of bounds>, keyset=0x700, update=0x1)
    at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinumconfig.c:1641
#4  0xc06abdc5 in vinum_scandisk (devicename=0xc18d68a0 "da5 da4 da3 da2 da1 da0 ad0")
    at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinumio.c:942
#5  0xc06a4c65 in vinumattach (dummy=0x0) at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinum.c:176
#6  0xc06a4f6d in vinum_modevent (mod=0xc0b89f00, type=1792, unused=0x0)
    at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinum.c:277
#7  0xc0308541 in module_register_init (arg=0xc06b5054) at /src/FreeBSD/ZAPHOD/src/sys/kern/kern_module.c:107
#8  0xc02ed275 in mi_startup () at /src/FreeBSD/ZAPHOD/src/sys/kern/init_main.c:214

```

This shows that we got to `get_empty_drive` from `find_drive`. Why?

```

486 int
487 find_drive(const char *name, int create)
488 {
489     int driveno;
490     struct drive *drive;
491
492     if (name != NULL) {
493         for (driveno = 0; driveno < vinum_conf.drives_allocated; driveno++) {
494             drive = &DRIVE[driveno];
495             if ((drive->label.name[0] != ' ') /* it has a name */
496                 &&(strcmp(drive->label.name, name) == 0) /* and it's this one */
497                 &&(drive->state > drive_unallocated)) /* and it's a real one: found */
498                 return driveno;
499         }
500     }
501     /* the drive isn't in the list. Add it if he wants */
502     if (create == 0) /* don't want to create */
503         return -1; /* give up */
504
505     driveno = get_empty_drive();
506     drive = &DRIVE[driveno];
507     if (name != NULL)
508         strncpy(drive->label.name, name, sizeof(drive->label.name)); /* put in its name */
509     drive->state = drive_referenced; /* in use, nothing worthwhile */
510     return driveno; /* return the index */

```

So we're trying to find a drive, but it doesn't exist. Looking at `config_subdisk`, we find we're in a case statement:

```

1151     case kw_drive:
1152         sd->driveno = find_drive(token[++parameter], 1); /* insert info */
1153         break;

```


This is part of the config line parsing. The config line might look something like:

```
sd usr.p0.s0 drive virtual size 43243243222s
```

Unfortunately, Vinum doesn't know a drive called `virtual`: maybe it was a drive which has failed. In such a case, Vinum creates a drive entry with the state referenced.

Looking further down the stack, we see our `vinum_scandisk`, as expected:

```
(gdb) f 4
#4 0xc06abdc5 in vinum_scandisk (devicename=0xc18d68a0 "da5 da4 da3 da2 da1 da0 ad0")
    at /src/FreeBSD/ZAPHOD/src/sys/dev/vinum/vinumio.c:942
942      parse_status = parse_config(config_line, &keyword_set, 1);
```

Looking back to `vinum_scandisk`, we see:

```
    else {
        vinum_conf.drives_used++;                /* another drive in use */
        /* Parse the configuration, and add it to the global configuration */
        for (cptr = config_text; *cptr != '\0';) {
            volatile int parse_status;           /* return value from parse_config */

            for (eptr = config_line; (*cptr != '\n') && (*cptr != '\0');)
                *eptr++ = *cptr++;               /* until the end of the line */
            *eptr = '\0';                         /* and delimit */
            if (setjmp(command_fail) == 0) { /* come back here on error and continue */
(line 942)      parse_status = parse_config(config_line, &keyword_set, 1); /* parse config */
... error check code
            }
        }
        drive->flags |= VF_CONFIGURED;           /* this drive's configuration is complete */
    }
```

The problem here is that `parse_config` changes the location of the drive, but the drive pointer remains pointing to the old location. At the end of the example, it then sets the `VF_CONFIGURED` bit. It's not immediately apparent that the pointer is reset in a function called indirectly from `parse_config`, particularly in a case like this where `parse_config` does not normally allocate a drive. It's easy to look for the bug where the code is obviously creating new drive entries.

Once we know this, solving the problem is trivial: reinitialize the drive pointer after the call to `parse_config`:

```
@@ -940,6 +940,14 @@
    *eptr = '\0';                                /* and delimit */
    if (setjmp(command_fail) == 0) { /* come back here on error and continue */
        parse_status = parse_config(config_line, &keyword_set, 1); /* parse config */
+
+        /*
+         * parse_config recognizes referenced
+         * drives and builds a drive entry for
+         * them. This may expand the drive
+         * table, thus invalidating the pointer.
+         */
        drive = &DRIVE[drivelist[driveno]]; /* point to the drive */
+
        if (parse_status < 0) {                    /* error in config */
            /*
             * This config should have been parsed
```

Another panic

After fi xing the previous bug, we get the following panic:

```
Mounting root from ufs:/dev/ad0s2a
Memory modified at 0xc1958838 after free 0xc1958000(4092)
panic: Most recently used by devbuf
```

This looks almost identical, and the obvious fi rst conclusion is that the change didn't fi x the bug. That's jumping to conclusions, though: the panic message is a symptom, not a cause, and we should look at it more carefully. Again, the fi rst thing to do is to look at the back trace. We fi nd something very similar to the previous example: the process involved is almost certainly not the culprit. Instead, since we're working on Vinum, we suspect Vinum.

Looking at the memory allocation, we see:

```
(gdb) finfo                                show info about freed memory
Block      Time      Sequence      size      address      line      file
0          19.539380      8          512      0xc1975c00    318      vinumio.c
1          19.547689      10         512      0xc197a000    318      vinumio.c
2          19.554801      12         512      0xc197a800    318      vinumio.c
3          19.568804      14         512      0xc197ae00    318      vinumio.c
4          19.568876      0         1024      0xc1981c00    468      vinumconfig.c
5          19.583257      17         512      0xc1975e00    318      vinumio.c
6          19.597787      19         512      0xc1975e00    318      vinumio.c
7          19.598547      21         512      0xc197a800    318      vinumio.c
8          19.602026      20         256      0xc1991700    598      vinumconfig.c
9          19.602936      23         512      0xc1975c00    318      vinumio.c
10         19.606420      22         256      0xc1991400    598      vinumconfig.c
11         19.607325      25         512      0xc197ac00    318      vinumio.c
12         19.610766      24         256      0xc1991100    598      vinumconfig.c
13         19.611664      27         512      0xc197ac00    318      vinumio.c
14         19.615103      26         256      0xc198dd00    598      vinumconfig.c
15         19.616040      29         512      0xc197ac00    318      vinumio.c
16         19.619775      28         256      0xc198da00    598      vinumconfig.c
17         19.620171      5         1024      0xc197ec00    882      vinumio.c
18         19.655536      1          768      0xc1981400    845      vinumconfig.c
19         19.659108      15        2048      0xc18e5000    468      vinumconfig.c
20         19.696490      2         2144      0xc1958000    765      vinumconfig.c
21         19.828777      30       131072      0xc1994000    974      vinumio.c
22         19.828823      6         1024      0xc197f000    975      vinumio.c
23         19.829590      4          28      0xc18d68a0    982      vinumio.c
```

The address 0xc1958838 is in the block freed at sequence number 20, which fi nishes at address 0xc1958000 + 2144, or 0xc1958860. It would be interesting to know where it points:

```
(gdb) p/x *0xc1958838
$2 = 0xc1994068
```

After a lot of investigation, including another *meminfo* output like the one on page 54, we conclude that this pointer doesn't point into a Vinum structure. Maybe this isn't Vinum after all?

Look at the code round where the block was freed, *vinumconfi g.c* line 765:

```
if (plexno >= vinum_conf.plexes_allocated)
    EXPAND(PLEX, struct plex, vinum_conf.plexes_allocated, INITIAL_PLEXES);

/* Found a plex. Give it an sd structure */
```

```
plex = &PLEX[plexno];                                /* this one is ours */
```

We've already seen the `EXPAND` macro, which is effectively the same as `realloc`. As before, the pointer to the plex is not allocated until after the call to `EXPAND`, and it's probably from a function which calls it. There are two ways to look at this problem:

1. Look at all the calls and read code to see where something might have happened.
2. Look at what got changed and try to guess what it was.

Which is better? We won't know until we've done both. Normally we'll be happy with the first one unless we're not sure that we've done it right, in which case we can check the validity of our assumptions by doing it the other way too.

Finding what changed is relatively easy. First we need to know how long `struct plex` is. There are a couple of ways of doing this:

- Count it in the header files. Good for sleepless nights.
- Look at the length that was allocated, 2144 bytes. From *vinumvar.h* we find:

```
INITIAL_PLEXES = 8,
```

So the length of a plex must be $2144 / 8$ bytes, or 268 bytes. This method is easier, but it requires finding this definition.

- Look at the addresses:

```
(gdb) p &vinum_conf.plex[0]
$5 = (struct plex *) 0xc18a7000
(gdb) p &vinum_conf.plex[1]
$6 = (struct plex *) 0xc18a710c
```

What you can't do is:

```
(gdb) p &vinum_conf.plex[1] - &vinum_conf.plex[0]
$7 = 0x1
```

This gives you a result in units of `sizeof (struct plex)`, not bytes. You have to do:

```
(gdb) p (char*) &vinum_conf.plex[1] - (char *) &vinum_conf.plex[0]
$8 = 0x10c
```

Whichever method you use, we have the length of `struct plex`, so we can determine which plex entry was affected: it's the offset divided by the length, $0x838 / 0x10c$, or 7. The offset in the plex is the remainder, $0x838 - 0x10c * 7$:

```
(gdb) p 0x838 - 0x10c * 7
$9 = 0xe4
```

That's pretty close to the end of the plex. Looking at the struct, we see:

```
(gdb) p ((struct plex *) 0xc1958000) [7]
$10 = {
  organization = 3735929054,
```


the plex lockmtx, so it might seem that we need to look at them all. But the value that has changed is a list pointer, so it's a good choice that this is creating or destroying a mutex. That leaves only the first two mutexes, in *vinumconfi g.c*.

Looking at the code around line 831, we find it's in `free_plex`:

```
/*
 * Free an allocated plex entry
 * and its associated memory areas
 */
void
free_plex(int plexno)
{
    struct plex *plex;

    plex = &PLEX[plexno];
    if (plex->sdnos)
        Free(plex->sdnos);
    if (plex->lock)
        Free(plex->lock);
    if (isstripped(plex))
        mtx_destroy(&plex->lockmtx);
    destroy_dev(plex->dev);
    bzero(plex, sizeof(struct plex));
    plex->state = plex_unallocated;
    /* and clear it out */
}
```

Here, the parameter passed is the plex number, not the plex pointer, which is initialized in the function. Theoretically it could also be a race condition, which would imply a problem with the config lock. But more important is that the plex lock is being freed immediately before. If it were working on freed memory, the value of `plex->lock` would be `0xdead0de`, so it would try to free it and panic right there, since `0xdead0de` is not a valid address. So it can't be this one.

Line 1457 is in `config_plex`:

```
if (isstripped(plex)) {
    plex->lock = (struct rangelock *)
        Malloc(PLEX_LOCKS * sizeof(struct rangelock));
    CHECKALLOC(plex->lock, "vinum: Can't allocate lock table\n");
    bzero((char *) plex->lock, PLEX_LOCKS * sizeof(struct rangelock));
    mtx_init(&plex->lockmtx, plex->name, "plex", MTX_DEF);
}
```

Again, if we had been through this code, we would have allocated a lock table, but there's no evidence of that.

We could go on looking at the other instances, but it's unlikely that any of those functions would change the linkage. What *does* change the linkage is the creation or destruction of other mutexes. This is a basic problem with the approach: you can't move an element in a linked list without changing the linkage. That's the bug.

So how do we solve the problem? Again, there are two possibilities:

- When moving the plex table, adjust the mutex linkage.
- Don't move the mutexes.

Let's look at how this mutex gets used, in `lock_plex`:

```

/*
 * we can't use 0 as a valid address, so
 * increment all addresses by 1.
 */
stripe++;
mtx_lock(&plex->lockmtx);

/* Wait here if the table is full */
while (plex->usedlocks == PLEX_LOCKS) /* all in use */
    msleep(&plex->usedlocks, &plex->lockmtx, PRIBIO, "vlock", 0);

```

In older versions of FreeBSD, as well as NetBSD and OpenBSD, the corresponding code is:

```

/*
 * we can't use 0 as a valid address, so
 * increment all addresses by 1.
 */
stripe++;
/*
 * We give the locks back from an interrupt
 * context, so we need to raise the spl here.
 */
s = splbio();

/* Wait here if the table is full */
while (plex->usedlocks == PLEX_LOCKS) /* all in use */
    tsleep(&plex->usedlocks, PRIBIO, "vlock", 0);

```

In other words, the mutex simply replaces an `splbio` call, which is a no-op in FreeBSD release 5. So why one mutex per plex? It's simply an example of finer-grained locking. There are two ways to handle this issue:

- Use a single mutex for all plexes. That's the closest approximation to the original, but it can mean unnecessary waits: the only thing we want to avoid in this function is having two callers locking the same plex, not two callers locking different plexes.
- Use a pool of mutexes. Each plex is allocated one of a number of mutexes. If more than one plex uses the same mutex, there's a possibility of unnecessary delay, but it's not as much as if all plexes used the same mutex.

I chose the second way. In Vinum startup, I added this code:

```

#define MUTEXNAMELEN 16
char mutexname[MUTEXNAMELEN];
#if PLEXMUTEXES > 10000
#error Increase size of MUTEXNAMELEN
#endif

...

for (i = 0; i < PLEXMUTEXES; i++) {
    snprintf(mutexname, MUTEXNAMELEN, "vinumplex%d", i);
    mtx_init(&plexmutex[i], mutexname, "plex", MTX_DEF);
}

```

Then the code in `config_plex` became:

```

if (isstriped(plex)) {
    plex->lock = (struct rangelock *)
        Malloc(PLEX_LOCKS * sizeof(struct rangelock));
    CHECKALLOC(plex->lock, "vinum: Can't allocate lock table\n");
    bzero((char *) plex->lock, PLEX_LOCKS * sizeof(struct rangelock));
}

```

```
    plex->lockmtx = &plexmutex[plexno % PLEXMUTEXES]; /* use this mutex for locking */  
}
```

Since the mutexes no longer belong to a single plex, there's no need to destroy them when destroying the plex; instead, they're destroyed when unloading the Vinum module.

8

panic: cleaned vnode isn't

zaphod, a FreeBSD 5-CURRENT system, panics regularly with the message:

```
panic: cleaned vnode isn't
at line 755 in file /usr/src/sys/kern/vfs_subr.c
```

Look at the dump:

```
# cd /usr/obj/usr/src/sys/ZAPHOD/
# ls -l kernel* /boot/kernel/kernel
-r-xr-xr-x  1 root  wheel   5403188 May  6 08:41 /boot/kernel/kernel
-rwxr-xr-x  1 root  wheel   5403188 May  6 08:41 kernel
-rwxr-xr-x  1 root  wheel  30470585 May  6 08:41 kernel.debug
# gdb -k kernel.debug /var/crash/vmcore.8
...
This GDB was configured as "i386-undermydesk-freebsd"...
panic: cleaned vnode isn't
panic messages:
---
panic: cleaned vnode isn't
at line 755 in file /usr/src/sys/kern/vfs_subr.c
cpuid = 0;
Debugger("panic")
Dumping 384 MB
 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256 272 288 304 320 336 352 368
---
Reading symbols from /usr/obj/usr/src/sys/ZAPHOD/modules/usr/src/sys/modules/dcons/dco
ns.ko.debug...done.
Loaded symbols for /usr/obj/usr/src/sys/ZAPHOD/modules/usr/src/sys/modules/dcons/dcons
.ko.debug
Reading symbols from /usr/obj/usr/src/sys/ZAPHOD/modules/usr/src/sys/modules/dcons_cro
m/dcons_crom.ko.debug...done.
Loaded symbols for /usr/obj/usr/src/sys/ZAPHOD/modules/usr/src/sys/modules/dcons_crom/
dcons_crom.ko.debug
#0  doadump () at /usr/src/sys/kern/kern_shutdown.c:236
236      dumping++;
Ready to go.  Enter 'tr' to connect to the remote target
with /dev/cuaa0, 'tr /dev/cuaal' to connect to a different port
or 'trf portno' to connect to the remote target with the firewire
interface.  portno defaults to 5556.

Type 'getsyms' after connection to load kld symbols.

If you're debugging a local system, you can use 'kldsyms' instead
```


to load the kld symbols. That's a less obnoxious interface.

As always, the first thing to do is to look at a stack trace:

```
(kgdb) bt
#0 doadump () at /usr/src/sys/kern/kern_shutdown.c:236
#1 0xc045c882 in db_fncall (dummy1=0x0, dummy2=0x0, dummy3=0xc0886034,
dummy4=0xd7d427f4 "ÃÑ\211Ã((ÔxR<sÃ((Ôx¿<sÃ\220\a") at /usr/src/sys/ddb/db_command.
c:551
#2 0xc045c688 in db_command (last_cmdp=0xc08535c0, cmd_table=0x0, aux_cmd_tablep=0xc0
7d66a8,
aux_cmd_tablep_end=0xc07d66c0) at /usr/src/sys/ddb/db_command.c:348
#3 0xc045c768 in db_command_loop () at /usr/src/sys/ddb/db_command.c:475
#4 0xc045eefd in db_trap (type=0x3, code=0x0) at /usr/src/sys/ddb/db_trap.c:73
#5 0xc073a219 in kdb_trap (type=0x3, code=0x0, regs=0xd7d42920) at /usr/src/sys/i386/
i386/db_interface.c:159
#6 0xc074c67c in trap (frame=
{tf_fs = 0x18, tf_es = 0x10, tf_ds = 0x10, tf edi = 0xc07ba264, tf esi = 0x1, tf
_ebp = 0xd7d42964, tf_esp = 0xd7d4294c, tf_ebx = 0x0, tf_edx = 0x0, tf_ecx = 0xc101400
0, tf_eax = 0x12, tf_trapno = 0x3, tf_err = 0x0, tf_eip = 0xc073a4de, tf_cs = 0x8, tf_
eflags = 0x296, tf_esp = 0xd7d42998, tf_ss = 0xd7d42984}) at /usr/src/sys/i386/tr
ap.c:579
#7 0xc073a4de in Debugger (msg=0xc07b390c "panic") at machine/cpufunc.h:56
#8 0xc05ddc85 in __panic (file=0xc07ba1fb "/usr/src/sys/kern/vfs_subr.c", line=0x2f3,
fmt=0xc07ba264 "cleaned vnode isn't") at /usr/src/sys/kern/kern_shutdown.c:532
#9 0xc06259b0 in getnewvnode (tag=0xc07bdc45 "ufs", mp=0xc399e800, vops=0x0, vpp=0x0)
at /usr/src/sys/kern/vfs_subr.c:785
#10 0xc06f7cb0 in ffs_vget (mp=0xc399e800, ino=0x39471e, flags=0x2, vpp=0xd7d42a84)
at /usr/src/sys/ufs/ffs/ffs_vfsops.c:1252
#11 0xc06fe9da in ufs_lookup (ap=0xd7d42b40) at /usr/src/sys/ufs/ufs/ufs_lookup.c:599
#12 0xc0704ae7 in ufs_vnoperate (ap=0x0) at /usr/src/sys/ufs/ufs/ufs_vnops.c:2819
#13 0xc061deb1 in vfs_cache_lookup (ap=0x0) at vnode_if.h:82
#14 0xc0704ae7 in ufs_vnoperate (ap=0x0) at /usr/src/sys/ufs/ufs/ufs_vnops.c:2819
#15 0xc0622377 in lookup (ndp=0xd7d42c30) at vnode_if.h:52
#16 0xc0621df8 in namei (ndp=0xd7d42c30) at /usr/src/sys/kern/vfs_lookup.c:179
#17 0xc062ccde in lstat (td=0xc5333bd0, uap=0xd7d42d14) at /usr/src/sys/kern/vfs_sysca
lls.c:2063
#18 0xc074ce57 in syscall (frame=
{tf_fs = 0x805002f, tf_es = 0xffff002f, tf_ds = 0xbfbf002f, tf edi = 0x8066600,
tf esi = 0x8066648, tf_ebp = 0xbfbfec58, tf_esp = 0xd7d42d74, tf_ebx = 0x2812e78c, tf_
edx = 0x80533c0, tf_ecx = 0x0, tf_eax = 0xbe, tf_trapno = 0x0, tf_err = 0x2, tf_eip =
0x280bd2a7, tf_cs = 0x1f, tf_eflags = 0x292, tf_esp = 0xbfbfbbbc, tf_ss = 0x2f})
at /usr/src/sys/i386/i386/trap.c:1004
#19 0x280bd2a7 in ?? ()
---Can't read userspace from dump, or kernel process---
```

This last message comes from FreeBSD 5.0 round mid-2004, where *gdb* no longer accesses userland.

Looking at the back trace, frame 9 (*getnewvnode*) is the culprit.

```
(kgdb) f 9
#9 0xc06259b0 in getnewvnode (tag=0xc07bdc45 "ufs", mp=0xc399e800, vops=0x0, vpp=0x0)
at /usr/src/sys/kern/vfs_subr.c:785
785 KASSERT(vp->v_dirtyblkroot == NULL, ("dirtyblkroot not NULL"));
(kgdb) l
780 lockdestroy(vp->v_vnlock);
781 lockinit(vp->v_vnlock, PVFS, tag, VLKTIMEOUT, LK_NOPAUSE);
782 KASSERT(vp->v_cleanbufcnt == 0, ("cleanbufcnt not 0"));
783 KASSERT(vp->v_cleanblkroot == NULL, ("cleanblkroot not NULL"));
784 KASSERT(vp->v_dirtybufcnt == 0, ("dirtybufcnt not 0"));
785 KASSERT(vp->v_dirtyblkroot == NULL, ("dirtyblkroot not NULL"));
786 } else {
787 numvnodes++;
788 mtx_unlock(&vnode_free_list_mtx);
789
```

The code is funny. We have a `KASSERT`, which asserts that a certain condition exists. If it doesn't, it panics with the second string. But the string isn't correct: the panic message is "cleaned vnode isn't", but the message in the code is "dirtyblkroot not NULL". The problem here is the optimizer: there are many potential calls to `panic`, and the optimizer improves the code by creating only one call and getting the other calls to jump to that one call. Looking for the panic message "cleaned vnode isn't" in that file, we find it at line 755:

```
(kgdb) 1 755
750             mtx_unlock(&vnode_free_list_mtx);
751
752     #ifdef INVARIANTS
753     {
754             if (vp->v_data)
755                 panic("cleaned vnode isn't");
756             if (vp->v_numoutput)
757                 panic("Clean vnode has pending I/O's");
758             if (vp->v_writecount != 0)
759                 panic("Non-zero write count");
```

We should confirm that we're in the right place; this kind of discrepancy could also be due to the use of the incorrect source file. We can get confirmation by looking at the code at that line:

```
(kgdb) i 1i 754
Line 754 of "/usr/src/sys/kern/vfs_subr.c" starts at address 0xc0625870 <getnewvnode+516>
and ends at 0xc062587c <getnewvnode+528>.
(kgdb) x/10i 0xc0625870
0xc0625870 <getnewvnode+516>: add     $0x10,%esp
0xc0625873 <getnewvnode+519>: cmpl   $0x0,0xa8(%esi)
0xc062587a <getnewvnode+526>: je     0xc062588c <getnewvnode+544>
0xc062587c <getnewvnode+528>: push  $0xc07ba264
0xc0625881 <getnewvnode+533>: push  $0x2f3
0xc0625886 <getnewvnode+538>: jmp    0xc06259a6 <getnewvnode+826>
(kgdb) x/10i 0xc06259a6
0xc06259a6 <getnewvnode+826>: push  $0xc07ba1fb
0xc06259ab <getnewvnode+831>: call  0xc05ddb48 <__panic>
0xc06259b0 <getnewvnode+836>: incl  0xc0878014
```

The address after the call to `panic` is the return address in our stack trace, so it's reasonable to assume that this is, in fact, correct. So the test is at line 754: is `vp->v_data` set to `NULL`? Let's look at the vnode:

```
(kgdb) p *vp
$1 = {
  v_interlock = {
    mtx_object = {
      lo_class = 0xc080c83c,
      lo_name = 0xc07ba2fb "vnode interlock",
      lo_type = 0xc07ba2fb "vnode interlock",
      lo_flags = 0x30000,
      lo_list = {
        tqe_next = 0x0,
        tqe_prev = 0x0
      },
      lo_witness = 0x0
    },
    mtx_lock = 0xc5333bd0,
    mtx_recurse = 0x0
  },
  v_iflag = 0x80,
  v_usecount = 0x0,
  v_numoutput = 0x0,
  v_vxthread = 0x0,
```

```

v_holdcnt = 0x0,
v_cleanblkhd = {
    tqh_first = 0x0,
    tqh_last = 0xc4804858
},
v_cleanblkroot = 0x0,
v_cleanbufcnt = 0x0,
v_dirtyblkhd = {
    tqh_first = 0x0,
    tqh_last = 0xc4804868
},
v_dirtyblkroot = 0x0,
v_dirtybufcnt = 0x0,
v_vflag = 0x0,
v_writecount = 0x0,
v_object = 0x0,
v_lastw = 0x0,
v_cstart = 0x0,
v_lasta = 0x0,
v_clen = 0x0,
v_un = {
    vu_mountedhere = 0x0,
    vu_socket = 0x0,
    vu_spec = {
        vu_cdev = 0x0,
        vu_specnext = {
            sle_next = 0x0
        }
    },
    vu_fifoinfo = 0x0
},
v_freelist = {
    tqe_next = 0x0,
    tqe_prev = 0xc42e07a4
},
v_nmntvnodes = {
    tqe_next = 0xc506f71c,
    tqe_prev = 0xc3d207ac
},
v_synclist = {
    le_next = 0x0,
    le_prev = 0x0
},
v_type = VBAD,
v_tag = 0xc07bdc45 "ufs",
v_data = 0xc489b578,
v_lock = {
    lk_interlock = 0xc08705b4,
    lk_flags = 0x1000040,
    lk_sharecount = 0x0,
    lk_waitcount = 0x0,
    lk_exclusivecount = 0x0,
    lk_prio = 0x50,
    lk_wmesg = 0xc07bdc45 "ufs",
    lk_timo = 0x6,
    lk_lockholder = 0xffffffff,
    lk_newlock = 0x0
},
v_vnlock = 0xc48048cc,
v_op = 0xc38ed000,
v_mount = 0xc399e800,
v_cache_src = {
    lh_first = 0x0
},
v_cache_dst = {
    tqh_first = 0xc529d8c4,
    tqh_last = 0xc529d8d4
},
v_id = 0x1329183,
v_dd = 0xc4804820,
v_ddid = 0x0,

```

```

    v_pollinfo = 0x0,
    v_label = 0x0,
    v_cachedfs = 0x41b,
    v_cachedid = 0x391bf8
}

```

So `vp->v_data` isn't NULL. Why not? The first obvious thing to do would be to look at the rest of the structure. For example, it could conceivably be complete junk, which could happen if the pointer itself were corrupted, or if something overwrote the object. In this case, though, without looking at all the pointers there's not much that looks obviously wrong. The interlock mutex has a name `vnode interlock`, which looks plausible. The list links look reasonable (they're well above the kernel base address of `0xc0000000`). The `v_tag` is `ufs`, which seems reasonable. In general, at first glance there's no reason to believe that this isn't a valid vnode pointer, and the vnode hasn't been overwritten *en masse*. About the only thing that is unusual is the field `v_type`: it's `VBAD`. With *etags* or similar we find it's in `sys/sys/vnode.h`:

```

/*
 * Vnode types.  VNON means no type.
 */
enum vtype      { VNON, VREG, VDIR, VBLK, VCHR, VLNK, VSOCK, VFIFO, VBAD };

```

There's no further explanation, but the name of the *enum*, as well as the fact that *gdb* even uses it, shows that it's in the correct place. The name suggests that there's something wrong with this vnode.

But what's the file? Looking further down the stack we find we're called from `namei`, which resolves path names. Looking at it, we see:

```

(kgdb) l namei
92     *      }
93     */
94     int
95     namei(ndp)
96         register struct nameidata *ndp;
97     {
98         register struct filedesc *fdp; /* pointer to file descriptor state */
99         register char *cp;             /* pointer into pathname argument */
100        register struct vnode *dp;      /* the directory we are searching */
101        struct iovec aiov;               /* uio for reading symbolic links */

```

The parameter `ndp` passed to `namei` contains all the to `namei`. It is defined in `sys/namei.h`:

```

56     /*
57     * Encapsulation of namei parameters.
58     */
59     struct nameidata {
60         /*
61         * Arguments to namei/lookup.
62         */
63         const char *ni_dirp;             /* pathname pointer */
64         enum uio_seg ni_segflg;          /* location of pathname */
65         /*
66         * Arguments to lookup.
67         */
68         struct vnode *ni_startdir;        /* starting directory */
69         struct vnode *ni_rootdir;         /* logical root directory */
70         struct vnode *ni_topdir;          /* logical top directory */
71         /*
72         * Results: returned from/manipulated by lookup
73         */

```

```

74         struct vnode *ni_vp;           /* vnode of result */
75         struct vnode *ni_dvp;          /* vnode of intermediate directory */
76         /*
77          * Shared between namei and lookup/commit routines.
78          */
79         size_t ni_pathlen;              /* remaining chars in path */
80         char *ni_next;                  /* next location in pathname */
81         u_long ni_loopcnt;              /* count of symlinks encountered */
82         /*
83          * Lookup parameters: this structure describes the subset of
84          * information from the nameidata structure that is passed
85          * through the VOP interface.
86          */
87         struct componentname ni_cnd;
88     };

```

So there we have the pathname at `ndp->ni_dirp`. Looking at it, we find:

```

(kgdb) f 16
#16 0xc0621df8 in namei (ndp=0xd7d42c30) at /usr/src/sys/kern/vfs_lookup.c:179
179                                     error = lookup(ndp);
(kgdb) p ndp->ni_dirp
$4 = 0x80666a8---Can't read userspace from dump, or kernel process---

```

This is the same bug in *gdb* that we saw above, and now it's very annoying. Looking at the message buffer, we see:

```

(kgdb) dmesg
... much output omitted
<118>Aug 26 18:59:34 zaphod postfix/postqueue[1750]: fatal: Cannot flush mail queue -
mail system is down
panic: cleaned vnode isn't
at line 755 in file /usr/src/sys/kern/vfs_subr.c
cpu0id = 0;
Debugger("panic")
Dumping 384 MB
 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256 272 288 304 320 336 352 368

```

In other words, no messages about bad files. The next possibility is to look through the stack for where the name gets used. This requires a little more code reading. It doesn't make much difference to finding the name whether we start at the top or bottom of the stack, but starting at the bottom might make it easier to understand the calling sequence.

syscall

`syscall` is the clearing house function for all system calls. It takes the trap frame from the `int0x80` instruction and extracts the register contents from it. Here's a simplified version:

```

894     /*
895     *      syscall -      system call request C handler
896     *
897     *      A system call is essentially treated as a trap.
898     */
899     void
900     syscall(frame)
901         struct trapframe frame;
902     {
903         caddr_t params;
904         struct sysent *callp;
905         struct thread *td = curthread;
906         struct proc *p = td->td_proc;
907         register_t orig_tf_eflags;

```

```

908         u_int sticks;
909         int error;
910         int narg;
911         int args[8];
912         u_int code;
913
914         /*
915          * note: PCPU_LAZY_INC() can only be used if we can afford
916          * occasional inaccuracy in the count.
917          */
918         PCPU_LAZY_INC(cnt.v_syscall);
919
920 #ifdef DIAGNOSTIC
921         if (ISPL(frame.tf_cs) != SEL_UPL) {
922             mtx_lock(&Giant);          /* try to stabilize the system XXX */
923             panic("syscall");
924             /* NOT REACHED */
925             mtx_unlock(&Giant);
926         }
927 #endif
928
929         sticks = td->td_sticks;
930         td->td_frame = &frame;
931         if (td->td_ucred != p->p_ucred)
932             cred_update_thread(td);
933         if (p->p_flag & P_SA)
934             thread_user_enter(p, td);
935         params = (caddr_t)frame.tf_esp + sizeof(int);
936         code = frame.tf_eax;
937         orig_tf_eflags = frame.tf_eflags;
938
939         if (p->p_sysent->sv_prepsyscall) {
940             /*
941              * The prep code is MP aware.
942              */
943             (*p->p_sysent->sv_prepsyscall)(&frame, args, &code, &params);
944         } else {
945             /*
946              * Need to check if this is a 32 bit or 64 bit syscall.
947              * fuword is MP aware.
948              */
949             if (code == SYS_syscall) {
950                 /*
951                  * Code is first argument, followed by actual args.
952                  */
953                 code = fuword(params);
954                 params += sizeof(int);
955             } else if (code == SYS___syscall) {
956                 /*
957                  * Like syscall, but code is a quad, so as to maintain
958                  * quad alignment for the rest of the arguments.
959                  */
960                 code = fuword(params);
961                 params += sizeof(quad_t);
962             }
963         }
964
965         if (p->p_sysent->sv_mask)
966             code &= p->p_sysent->sv_mask;
967
968         if (code >= p->p_sysent->sv_size)
969             callp = &p->p_sysent->sv_table[0];
970         else
971             callp = &p->p_sysent->sv_table[code];
972
973         narg = callp->sy_narg & SYF_ARGMASK;
974
975         /*
976          * copyin and the ktrsyscall()/ktrsysret() code is MP-aware
977          */
978         if (params != NULL && narg != 0)

```

```

979             error = copyin(params, (caddr_t)args,
980                             (u_int)(narg * sizeof(int)));
981     else
982         error = 0;

```

The following code is used by *ktrace* to trace system calls

```

984     #ifdef KTRACE
985         if (KTRPOINT(td, KTR_SYSCALL))
986             ktrsyscall(code, narg, args);
987     #endif

```

Next, we call the function which handles the system call:

```

989     /*
990     * Try to run the syscall without Giant if the syscall
991     * is MP safe.
992     */
993     if ((callp->sy_narg & SYF_MPSAFE) == 0)
994         mtx_lock(&Giant);
995
996     if (error == 0) {
997         td->td_retval[0] = 0;
998         td->td_retval[1] = frame.tf_edx;
999
1000         STOPEVENT(p, S_SCE, narg);
1001
1002         PTRACESTOP_SC(p, td, S_PT_SCE);
1003
1004         error = (*callp->sy_call)(td, args);

```

This is the call to the system call itself. The code below handles the return values.

```

1005     }
1006
1007     switch (error) {
1008     case 0:
1009         frame.tf_eax = td->td_retval[0];
1010         frame.tf_edx = td->td_retval[1];
1011         frame.tf_eflags &= ~PSL_C;
1012         break;
1013
1014     case ERESTART:
1015         /*
1016          * Reconstruct pc, assuming lcall $X,y is 7 bytes,
1017          * int 0x80 is 2 bytes. We saved this in tf_err.
1018          */
1019         frame.tf_eip -= frame.tf_err;
1020         break;
1021
1022     case EJUSTRETURN:
1023         break;
1024
1025     default:
1026         if (p->p_sysent->sv_errsize) {
1027             if (error >= p->p_sysent->sv_errsize)
1028                 error = -1; /* XXX */
1029             else
1030                 error = p->p_sysent->sv_errtbl[error];
1031         }
1032         frame.tf_eax = error;
1033         frame.tf_eflags |= PSL_C;
1034         break;
1035     }
1036
1037     /*

```

```

1038         * Release Giant if we previously set it.
1039         */
1040         if ((callp->sy_narg & SYF_MPSAFE) == 0)
1041             mtx_unlock(&Giant);
1042
1043         /*
1044         * Traced syscall.
1045         */
1046         if ((orig_tf_eflags & PSL_T) && !(orig_tf_eflags & PSL_VM)) {
1047             frame.tf_eflags &= ~PSL_T;
1048             trapsignal(td, SIGTRAP, 0);
1049         }
1050
1051         /*
1052         * Handle reschedule and other end-of-syscall issues
1053         */
1054         userret(td, &frame, sticks);
1055
1056 #ifdef KTRACE
1057         if (KTRPOINT(td, KTR_SYSRET))
1058             ktrsysret(code, error, td->td_retval[0]);
1059 #endif
1060
1061         /*
1062         * This works because errno is findable through the
1063         * register set.  If we ever support an emulation where this
1064         * is not the case, this code will need to be revisited.
1065         */
1066         STOPEVENT(p, S_SCX, code);
1067
1068         PTRACESTOP_SC(p, td, S_PT_SCX);
1069
1070 #ifdef DIAGNOSTIC
1071         cred_free_thread(td);
1072 #endif
1073         WITNESS_WARN(WARN_PANIC, NULL, "System call %s returning",
1074             (code >= 0 && code < SYS_MAXSYSCALL) ? syscallnames[code] : "???");
1075         mtx_assert(&sched_lock, MA_NOTOWNED);
1076         mtx_assert(&Giant, MA_NOTOWNED);
1077     }

```

This code doesn't actually look at the contents of the parameters, so we move on.

lstat

Clearly this system call is an `lstat` call, since that's where we arrive next. As we saw above, `syscall` calls the function with two arguments:

```

2039     /*
2040     * Get file status; this version does not follow links.
2041     */
2042 #ifndef _SYS_SYSPROTO_H_
2043     struct lstat_args {
2044         char    *path;
2045         struct stat *ub;
2046     };
2047 #endif
2048     int
2049     lstat(td, uap)
2050         struct thread *td;
2051         register struct lstat_args /* {
2052             char *path;
2053             struct stat *ub;
2054         } */ *uap;

```

`td` is a pointer to the thread of the current process, and `uap` ("user argument pointer") points to the arguments. Nearly all system calls have the same parameter names, so you should recognize

the name `uap`. The kind of structure depends on the function; in this case, it's defined at line 2043.

The first parameter is the path name, which is what we're looking for:

```
(kgdb) p *uap
$6 = {
  path_l_ = 0xd7d42d14 "f\006\bHf\006\b(Å2Å\0244\005\b",
  path = 0x80666a8---Can't read userspace from dump, or kernel process---
```

What's this? This has nothing to do with our definition of `uap`. It appears to be a bug in *gdb*, but it's not clear where. In particular, there doesn't seem to be any structure with a member called `path_l` in the kernel source tree. We could follow this, but it's probably better to leave that until we need it. In this case, the function is relatively short:

```
2055     {
2056         int error;
2057         struct vnode *vp;
2058         struct stat sb;
2059         struct nameidata nd;
2060
2061         NDINIT(&nd, LOOKUP, NOFOLLOW | LOCKLEAF | NOOBJ, UIO_USERSPACE,
2062             uap->path, td);
2063         if ((error = namei(&nd)) != 0)
2064             return (error);
2065         vp = nd.ni_vp;
2066         error = vn_stat(vp, &sb, td->td_ucred, NOCRED, td);
2067         NDFREE(&nd, NDF_ONLY_PNBUF);
2068         vput(vp);
2069         if (error)
2070             return (error);
2071         error = copyout(&sb, uap->ub, sizeof (sb));
2072         return (error);
2073     }
```

`NDINIT` uses the path name. Let's look at that. The name in all capitals suggests that it's a macro, but in fact it's an inline function in *sys/namei.h*:

```
142     /*
143      * Initialization of a nameidata structure.
144      */
145     static void NDINIT(struct nameidata *, u_long, u_long, enum uio_seg,
146         const char *, struct thread *);
147     static __inline void
148     NDINIT(struct nameidata *ndp,
149         u_long op, u_long flags,
150         enum uio_seg segflg,
151         const char *namep,
152         struct thread *td)
153     {
154         ndp->ni_cnd.cn_nameiop = op;
155         ndp->ni_cnd.cn_flags = flags;
156         ndp->ni_segflg = segflg;
157         ndp->ni_dirp = namep;
158         ndp->ni_cnd.cn_thread = td;
159     }
```

Yes, it uses the path name, but just to put it into the variable `nd`. We can check that:

```
(kgdb) p nd
$7 = {
  ni_dirp = 0x80666a8---Can't read userspace from dump, or kernel process---
```

Well, at least it's consistent, but this doesn't help us much more. The next line is a call to `namei`, so let's look there.

`namei`

`namei` is quite long, so we'll just look at parts of it. It starts with:

```

74      /*
75      * Convert a pathname into a pointer to a locked inode.
76      *
77      * The FOLLOW flag is set when symbolic links are to be followed
78      * when they occur at the end of the name translation process.
79      * Symbolic links are always followed for all other pathname
80      * components other than the last.
81      *
82      * The segflg defines whether the name is to be copied from user
83      * space or kernel space.
84      *
85      * Overall outline of namei:
86      *
87      *     copy in name
88      *     get starting directory
89      *     while (!done && !error) {
90      *         call lookup to search path.
91      *         if symbolic link, massage name in buffer and continue
92      *     }
93      */
94     int
95     namei(ndp)
96     {
97         register struct nameidata *ndp;
```

The obvious first pass is to search the function for references to `ndp` which come before the call to `lookup` at line 179. There are quite a few of them:

```

98         register struct filedesc *fdp; /* pointer to file descriptor state */
99         register char *cp; /* pointer into pathname argument */
100        register struct vnode *dp; /* the directory we are searching */
101        struct iovec aiov; /* uio for reading symbolic links */
102        struct uio auio;
103        int error, linklen;
104        struct componentname *cnp = &ndp->ni_cnd;
```

This isn't much use, since this data hasn't been completely initialized yet. From the definition of `NDINIT` `ndp->ni_cnd.cn_nameiop` and `ndp->ni_cnd.cn_flags` are initialized at this point.

Continuing,

```

105        struct thread *td = cnp->cn_thread;
106        struct proc *p = td->td_proc;
107
108        ndp->ni_cnd.cn_cred = ndp->ni_cnd.cn_thread->td_ucred;
```

This one is just credentials; not much help there.

```

109        KASSERT(cnp->cn_cred && p, ("namei: bad cred/proc"));
110        KASSERT((cnp->cn_nameiop & (~OPMASK)) == 0,
111            ("namei: nameiop contaminated with flags"));
112        KASSERT((cnp->cn_flags & OPMASK) == 0,
113            ("namei: flags contaminated with nameiops"));
114        fdp = p->p_fdp;
```

```

115
116      /*
117      * Get a buffer for the name to be translated, and copy the
118      * name into the buffer.
119      */
120      if ((cnp->cn_flags & HASBUF) == 0)
121          cnp->cn_pnbuf = uma_zalloc(namei_zone, M_WAITOK);
122      if (ndp->ni_segflg == UIO_SYSSPACE)
123          error = copystr(ndp->ni_dirp, cnp->cn_pnbuf,
124                          MAXPATHLEN, (size_t *)&ndp->ni_pathlen);
125      else
126          error = copyinstr(ndp->ni_dirp, cnp->cn_pnbuf,
127                           MAXPATHLEN, (size_t *)&ndp->ni_pathlen);

```

This one looks better. It copies the directory name to the component name variable `cnp->cn_pnbuf`. Should we look at it? That depends on whether it's been overwritten afterwards or not. Let's note this one and move on.

```

128
129      /*
130      * Don't allow empty pathnames.
131      */
132      if (!error && *cnp->cn_pnbuf == ' ')
133          error = ENOENT;
134
135      if (error) {
136          uma_zfree(namei_zone, cnp->cn_pnbuf);
137      #ifdef DIAGNOSTIC
138          cnp->cn_pnbuf = NULL;
139          cnp->cn_nameptr = NULL;
140      #endif
141          ndp->ni_vp = NULL;

```

This doesn't help much. We're just noting that we don't yet have a vnode pointer.

```

142          return (error);
143      }
144      ndp->ni_loopcnt = 0;

```

And here we're just initializing a variable.

```

145      #ifdef KTRACE
146          if (KTRPOINT(td, KTR_NAMEI)) {
147              KASSERT(cnp->cn_thread == curthread,
148                      ("namei not using curthread"));
149              ktrnamei(cnp->cn_pnbuf);
150          }
151      #endif
152
153      /*
154      * Get starting point for the translation.
155      */
156      FILEDESC_LOCK(fdp);
157      ndp->ni_rootdir = fdp->fd_rdir;
158      ndp->ni_topdir = fdp->fd_jdir;

```

This looks more interesting. What's in `fdp`?

```

(kgdb) p *fdp
$10 = {
    fd_ofiles = 0xc080c83c,
    fd_ofileflags = 0xc07ba2fb "vnode interlock",
    fd_cdir = 0xc07ba2fb,
    fd_rdir = 0x30000,

```

```

fd_jdir = 0x0,
fd_nfiles = 0x0,
fd_map = 0x0,
fd_lastfile = 0x4,
fd_freefile = 0x0,
fd_cmask = 0x0,
fd_refcnt = 0x0,
fd_knlistsize = 0x3,
fd_knlist = 0x0,
fd_knhashmask = 0x0,
fd_knhash = 0x2,
fd_mtx = {
    mtx_object = {
        lo_class = 0xcb237338,
        lo_name = 0xcb2373dc "",
        lo_type = 0xcb237338 " 01",
        lo_flags = 0x1,
        lo_list = {
            tqe_next = 0x0,
            tqe_prev = 0xc50e9a70
        },
        lo_witness = 0x0
    },
    mtx_lock = 0x0,
    mtx_recurse = 0x8
},
fd_holdleaderscount = 0x0,
fd_holdleaderswakeup = 0xc50ed000
}

```

Neither of these are interesting: if `fd_rdir` is a string, it would be in user space, so we couldn't do anything with it. `fd_jdir` is NULL, so it's not of interest. But there's another field there, `fd_cdir`, which looks like a valid pointer. Before seeing what it's used for, it's easier to check what it contains:

```

(kgdb) p *fdp->fd_cdir
$11 = {
    v_interlock = {
        mtx_object = {
            lo_class = 0x646f6e76,
            lo_name = 0x6e692065---Can't read userspace from dump, or kernel process---

```

We've seen this before, but this one is in a mutex; possibly there's other stuff behind which is of interest. So we go and look for the definition. It's a struct `filedesc`, which is defined in `sys/filedesc.h`:

```

42      /*
43      * This structure is used for the management of descriptors. It may be
44      * shared by multiple processes.
45      */
46      /* A process is initially started out with NDFILE descriptors stored within
47      * this structure, selected to be enough for typical applications based on
48      * the historical limit of 20 open files (and the usage of descriptors by
49      * shells). If these descriptors are exhausted, a larger descriptor table
50      * may be allocated, up to a process' resource limit; the internal arrays
51      * are then unused.
52      */
60
61      struct filedesc {
62          struct file **fd_ofiles;          /* file structures for open files */
63          char *fd_ofileflags;              /* per-process open file flags */
64          struct vnode *fd_cdir;            /* current directory */
65          struct vnode *fd_rdir;            /* root directory */
66          struct vnode *fd_jdir;            /* jail root directory */

```

So `fd_cdir` is the current directory, and it's of type `vnode`. That's defined in file `sys/vnode.h`. Omitting some comments and `#ifdefined` code, it looks like this:

```

93  * Vnodes may be found on many lists. The general way to deal with operating
94  * on a vnode that is on a list is:
95  *     1) Lock the list and find the vnode.
96  *     2) Lock interlock so that the vnode does not go away.
97  *     3) Unlock the list to avoid lock order reversals.
98  *     4) vget with LK_INTERLOCK and check for ENOENT, or
99  *     5) Check for XLOCK if the vnode lock is not required.
100  *     6) Perform your operation, then vput().
101  *
102  * XXX Not all fields are locked yet and some fields that are marked are not
103  * locked consistently. This is a work in progress. Requires Giant!
104  */
105
106 struct vnode {
107     struct    mtx v_interlock;           /* lock for "i" things */
108     u_long    v_iflag;                   /* i vnode flags (see below) */
109     int        v_usecount;               /* i ref count of users */
110     long       v_numoutput;              /* i writes in progress */
111     struct    thread *v_vxthread;        /* i thread owning VXLOCK */
112     int        v_holdcnt;                /* i page & buffer references */
113     struct    buflists v_cleanblkhd;     /* i SORTED clean blocklist */
114     struct    buf *v_cleanblkroot;       /* i clean buf splay tree */
115     int        v_cleanbufcnt;            /* i number of clean buffers */
116     struct    buflists v_dirtyblkhd;     /* i SORTED dirty blocklist */
117     struct    buf *v_dirtyblkroot;       /* i dirty buf splay tree */
118     int        v_dirtybufcnt;            /* i number of dirty buffers */
119     u_long     v_vflag;                  /* v vnode flags */
120     int        v_writecount;             /* v ref count of writers */
121     struct    vm_object *v_object;       /* v Place to store VM object */
122     daddr_t    v_lastw;                  /* v last write (write cluster) */
123     daddr_t    v_cstart;                 /* v start block of cluster */
124     daddr_t    v_lasta;                  /* v last allocation (cluster) */
125     int        v_clen;                  /* v length of current cluster */
126     union {
127         struct    mount *vu_mountedhere; /* v ptr to mounted vfs (VDIR) */
128         struct    socket *vu_socket;      /* v unix ipc (VSOCK) */
129         struct {
130             struct    cdev *vu_cdev; /* v device (VCHR, VBLK) */
131             SLIST_ENTRY(vnode) vu_specnext; /* s device aliases */
132         } vu_spec;
133         struct    fifoinfo *vu_fifoinfo; /* v fifo (VFIFO) */
134     } v_un;
135     TAILQ_ENTRY(vnode) v_freelist;       /* f vnode freelist */
136     TAILQ_ENTRY(vnode) v_nmmtvnodes;     /* m vnodes for mount point */
137     LIST_ENTRY(vnode) v_synclist;        /* S dirty vnode list */
138     enum        vtype v_type;            /* u vnode type */
139     const char *v_tag;                   /* u type of underlying data */
140     void        *v_data;                 /* u private data for fs */
141     struct    lock v_lock;               /* u used if fs don't have one */

```

```

142         struct    lock *v_vnlock;           /* u pointer to vnode lock */
143         vop_t      **v_op;                   /* u vnode operations vector */
144         struct      mount *v_mount;           /* u ptr to vfs we are in */
145         LIST_HEAD(, namecache) v_cache_src;  /* c Cache entries from us */
146         TAILQ_HEAD(, namecache) v_cache_dst; /* c Cache entries to us */
147         u_long      v_id;                     /* c capability identifier */
148         struct      vnode *v_dd;              /* c .. vnode */
149         u_long      v_ddid;                   /* c .. capability identifier */
150         struct      vpollinfo *v_pollinfo;    /* p Poll events */
151         struct      label *v_label;           /* MAC label for vnode */
152         udev_t       v_cachedfs;              /* cached fs id */
153         ino_t        v_cachedid;              /* cached file id */
154     };

```

The letters at the beginning of the comments refer to the locks required to access the individual fields of the vnode. What we're interested in here are any path names, but there aren't any: path names are a level above the vnode layer. We return to namei:

```

159
160         dp = fdp->fd_cdir;
161         VREF(dp);
162         FILEDESC_UNLOCK(fdp);
163         for (;;) {
164             /*
165              * Check if root directory should replace current directory.
166              * Done at start of translation and after symbolic link.
167              */
168             cnp->cn_nameptr = cnp->cn_pnbuf;
169             if (*(cnp->cn_nameptr) == '/') {
170                 vref(dp);
171                 while (*(cnp->cn_nameptr) == '/') {
172                     cnp->cn_nameptr++;
173                     ndp->ni_pathlen--;

```

This code strips leading / characters, which probably doesn't change very much. There's not much else in the loop:

```

174             }
175             dp = ndp->ni_rootdir;
176             VREF(dp);
177         }
178         ndp->ni_startdir = dp;
179         error = lookup(ndp);

```

So it would be interesting to find out what's in cnp:

```

(kgdb) p *cnp
$1 = {
  cn_nameiop = 0x0,
  cn_flags = 0xc084,
  cn_thread = 0xc5333bd0,
  cn_cred = 0xc5202580,
  cn_pnbuf = 0xc39d6400 "mime",
  cn_nameptr = 0xc39d6400 "mime",
  cn_namelen = 0x4,
  cn_consume = 0x0
}

```

At this point, and assuming that the called functions don't change our structures further, we seem to have only one lead: the pathname *mime*. There are a total of 725 directories called *mime* on this file system, so this doesn't help too much.

There's also another issue: since we're just allocating the vnode for this file, it can't be the file

that caused the problem. It's possible that it would happen in the same manner every time, but it's also possible that it might not: depending on what the system has been doing previously, vnodes could be recycled in different ways, and this one might be assigned to a different file on every occasion.

Since the machine keeps panicking, it's easy enough to check this. With another dump we see:

```
(kgdb) p *cnp
$4 = {
  cn_nameiop = 0x0,
  cn_flags = 0xc084,
  cn_thread = 0xc4891930,
  cn_cred = 0xc4922780,
  cn_pnbuf = 0xc4895000 "cpphash.h",
  cn_nameptr = 0xc4895000 "cpphash.h",
  cn_namelen = 0x9,
  cn_consume = 0x0
}
```

This tells us not one, but two things:

1. The path name *does* change.
2. This name is almost certainly the name of a file, not of a directory. Without checking, it's possible that it could be either.

It's possible that we could get more information with this approach, but it's looking less likely. Let's consider an alternative way to do it.

An alternative approach: find VBAD

One problem with the previous approach is that it's looking for the wrong file name. The vnode with VBAD set has already been freed, and we're trying to reuse it. A better way to look for the problem might be to look at where VBAD is used. Using the *etags* search function, we find:

1. A number of references in file systems we're not using, such as *fs/coda/*, *fs/ntfs/*, *fs/udf* and so on. We won't look at them.
2. In file *fs/devfs/devfs_vnops.c*, function *devfs_allocv*, we set VBAD if the directory type is incorrect:

```
151         if (de->de_dirent->d_type == DT_CHR) {
152             vp->v_type = VCHR;
153             vp = addaliasu(vp, dev->si_udev);
154             vp->v_op = devfs_specop_p;
155         } else if (de->de_dirent->d_type == DT_DIR) {
156             vp->v_type = VDIR;
157         } else if (de->de_dirent->d_type == DT_LNK) {
158             vp->v_type = VLNK;
159         } else {
160             vp->v_type = VBAD;
161         }
```

3. In function *acctwatch* in *kern/kern_acct.c* we abort if we find a vnode with VBAD set. This could be a possibility, but since this happens with *find*, it seems rather unlikely.

4. In file *kern/tty_cons.c* there's a macro definition that refers to it. We're dealing with a disk here, so we'll ignore this one too.
5. File *kern/vfs_subr.c* has a conversion table which uses it. It's possible that it's relevant, but we'll see that later.
6. In the same file, function *vlrureclaim* checks for it, but doesn't do anything useful if it finds it.
7. Still in *kern/vfs_subr.c*, function *vtryrecycle* checks for it:

```

588      /*
589      * Check to see if a free vnode can be recycled. If it can,
590      * recycle it and return it with the vnode interlock held.
591      */
592      static int
593      vtryrecycle(struct vnode *vp)
594      {
595          ...
596          /*
597          * If we got this far, we need to acquire the interlock and see if
598          * anyone picked up this vnode from another list. If not, we will
599          * mark it with XLOCK via vgonel() so that anyone who does find it
600          * will skip over it.
601          */
602          VI_LOCK(vp);
603          if (VSHOULDBUSY(vp) && (vp->v_iflag & VI_XLOCK) == 0) {
604              VI_UNLOCK(vp);
605              error = EBUSY;
606              goto done;
607          }
608          mtx_lock(&vnode_free_list_mtx);
609          TAILQ_REMOVE(&vnode_free_list, vp, v_freelist);
610          vp->v_iflag &= ~VI_FREE;
611          mtx_unlock(&vnode_free_list_mtx);
612          vp->v_iflag |= VI_DOOMED;
613          if (vp->v_type != VBAD) {
614              VOP_UNLOCK(vp, 0, td);
615              vgonel(vp, td);
616              VI_LOCK(vp);
617          } else
618              VOP_UNLOCK(vp, 0, td);
619          vn_finished_write(vnmp);
620          return (0);
621      }

```

This looks like a possibility for further investigation; we note it and continue searching for places.

8. Yet again in *kern/vfs_subr.c*, function *vgonel* (called from the previous function) sets it:

```

2594      /*
2595      * If it is on the freelist and not already at the head,
2596      * move it to the head of the list. The test of the
2597      * VDOOMED flag and the reference count of zero is because
2598      * it will be removed from the free list by getnewvnode,
2599      * but will not have its reference count incremented until
2600      * after calling vgone. If the reference count were
2601      * incremented first, vgone would (incorrectly) try to
2602      * close the previous instance of the underlying object.
2603      */
2604      if (vp->v_usecount == 0 && !(vp->v_iflag & VI_DOOMED)) {
2605          mtx_lock(&vnode_free_list_mtx);
2606          if (vp->v_iflag & VI_FREE) {
2607              TAILQ_REMOVE(&vnode_free_list, vp, v_freelist);
2608          } else {

```



```

2609                                vp->v_iflag |= VI_FREE;
2610                                freevnodes++;
2611                                }
2612                                TAILQ_INSERT_HEAD(&vnnode_free_list, vp, v_freelist);
2613                                mtx_unlock(&vnnode_free_list_mtx);
2614                                }
2615
2616                                vp->v_type = VBAD;
2617                                vx_unlock(vp);
2618                                VI_UNLOCK(vp);
2619                                }

```

This seems to be a general thing, so maybe VBAD isn't as seldom as it might appear. We need to look back at the vnode in question. What flags are set?

```

(kgdb) f 9
#9 0xc06259b0 in getnewvnode (tag=0xc07bdc45 "ufs", mp=0xc399e800, vops=0x0, vpp=0x0)
    at /usr/src/sys/kern/vfs_subr.c:785
785                                KASSERT(vp->v_dirtyblkroot == NULL, ("dirtyblkroot not NULL"));
(kgdb) p vp->v_iflag
$14 = 0x80
(kgdb)

```

From *sys/vnode.h* we read:

```

#define VI_DOOMED                0x0080 /* This vnode is being recycled */

```

So yes, indeed, it looks as if this vnode has been freed by this method.

But if that's the case, why is the *v_data* field not zeroed out?

9. Still in *kern/vfs_subr.c*, function *kern_mknod* checks for it:

```

switch (mode & S_IFMT) {
case S_IFMT: /* used by badsect to flag bad sectors */
    vattr.va_type = VBAD;
    break;

```

Clearly this isn't of interest to us, since we're not making a node when this panic occurs.

The reference in *vgone1* is important: we've been assuming that the value VBAD was a clue; now it looks as if any valid vnode we pull off the free list will have its type field set to VBAD. It looks as if this whole exercise was a waste of time. What now? We'll have to try yet another tack.

Zeroing vp->v_data

The immediate cause of the panic had nothing to do with the value of the *vp->va_type*: it was that *vp->v_data* was not set to NULL. So where does that get done? Again, we search the source tree, this time for the variable *v_data*. We find:

1. In file *coda/cnode.h* it's used to define a macro:

```

#define VTOC(vp)                ((struct cnode *) (vp)->v_data)

```

This is potentially a reason to note the name VTOC: it could be used to set the *v_data*

field. In this case, though, the name of the file shows us that it's only used in the *coda* file system, which we're not using. So we can forget this one. There's also another hit in *coda/coda_vnops.c*, which we won't discuss further.

2. We get a few false positives with names like `recv_data` and `ncv_data_read_bytes`. Clearly they're not what we're looking for, so we can ignore them too.
3. In function `devfs_delete`, *fs/devfs/devfs_devs.c* we find:

```

257         if (de->de_vnode)
258             de->de_vnode->v_data = NULL;
259         TAILQ_REMOVE(&dd->de_dlist, de, de_list);
260         FREE(de, M_DEVFS);

```

This time we're in *devfs*, so this reference isn't of immediate relevance. But it looks like the sort of thing that we might expect: before removing a vnode entry, we zero out the data pointer. We can expect to find a similar definition that *is* relevant to our code.

4. In function `devfs_populate` in the same file, we find:

```

308         if (dev == NULL && de != NULL) {
309             dd = de->de_dir;
310             *dep = NULL;
311             TAILQ_REMOVE(&dd->de_dlist, de, de_list);
312             if (de->de_vnode)
313                 de->de_vnode->v_data = NULL;
314             FREE(de, M_DEVFS);
315             devfs_dropref(i);
316             continue;
317         }

```

This is part of code which decides that the vnode in question is no longer required and recycles it. It's interesting to note that the `v_data` field requires explicit clearing. This could be a clue.

5. There are many further read-only references to the `v_data` field in this file; further down, though, we see:

```

649     static int
650     devfs_reclaim(ap)
651         struct vop_reclaim_args /* {
652             struct vnode *a_vp;
653         } */ *ap;
654     {
655         struct vnode *vp = ap->a_vp;
656         struct devfs_dirent *de;
657         int i;
658
659         de = vp->v_data;
660         if (de != NULL)
661             de->de_vnode = NULL;
662         vp->v_data = NULL;
663         if (vp->v_rdev != NODEV && vp->v_rdev != NULL) {
664             i = vcount(vp);
665             if ((vp->v_rdev->si_flags & SI_CHEAPCLONE) && i == 0 &&
666                 (vp->v_rdev->si_flags & SI_NAMED))
667                 destroy_dev(vp->v_rdev);
668         }
669         return (0);
670     }

```

This function has no comments whatsoever, but the name suggests that the vnode is no longer needed. Similar code also occurs in the function `devfs_remove`. There are also similar references in many other file systems.

6. The next reference of interest is one we know well, in function `getnewvnode` in *kern/vfs_subr.c*. This is where we panicked from.
7. A few lines down in the same function, we initialize the newly found vnode:

```

801         TAILQ_INIT(&vp->v_cleanblkhd);
802         TAILQ_INIT(&vp->v_dirtyblkhd);
803         vp->v_type = VNON;
804         vp->v_tag = tag;
805         vp->v_op = vops;
806         *vpp = vp;
807         vp->v_usecount = 1;
808         vp->v_data = 0;
809         vp->v_cachedid = -1;

```

Why do this? We've just checked for `v_data` being non-zero and panicked if it is.

The issue here is where we panicked. The test which failed is done in a section marked `#ifdef INVARIANTS`. It doesn't normally get executed, but since this machine was running a development kernel, `INVARIANTS` were turned on.

8. After that, `addaliasu` in *kern/vfs_subr.c* sets `v_data` to `NULL` while copying a vnode:

```

1816         /*
1817         * Discard unneeded vnode, but save its node specific data.
1818         * Note that if there is a lock, it is carried over in the
1819         * node specific data to the replacement vnode.
1820         */
1821         vref(ovp);
1822         ovp->v_data = nvp->v_data;
1823         ovp->v_tag = nvp->v_tag;
1824         nvp->v_data = NULL;

```

Again, this doesn't fit our scenario.

9. In *sys/vnode.h* we find the definition of `struct vnode`.
10. In file *ufs/ufs/inode.h* we find a macro that looks familiar:

```

/* Convert between inode pointers and vnode pointers. */
#define VTOI(vp)      ((struct inode *) (vp)->v_data)
#define ITOV(ip)      ((ip)->i_vnode)

```

We saw an almost identical macro `VTOC` in the coda code above. This time it's in the UFS code, so we need to take it seriously. We'll do that in a second pass.

11. Finally, in function `ufs_reclaim` in file *ufs/ufs/ufs_inode.c* we find the code:

```

135         /*
136         * Reclaim an inode so that it can be used for other purposes.
137         */
138         int
139         ufs_reclaim(ap)
140             struct vop_reclaim_args /* {
141                 struct vnode *a_vp;

```

```

142         struct thread *a_td;
143     } /* *ap;
144     {
145         struct vnode *vp = ap->a_vp;
146         struct inode *ip = VTOI(vp);
147         struct ufsmount *ump = ip->i_ump;
148     #ifdef QUOTA
149         int i;
150     #endif
151
152         VI_LOCK(vp);
153         if (prtactive && vp->v_usecount != 0)
154             vprint("ufs_reclaim: pushing active", vp);
155         VI_UNLOCK(vp);
156         if (ip->i_flag & IN_LAZYMOD) {
157             ip->i_flag |= IN_MODIFIED;
158             UFS_UPDATE(vp, 0);
159         }
160         /*
161          * Remove the inode from its hash chain.
162          */
163         ufs_ihashrem(ip);
164         /*
165          * Purge old data structures associated with the inode.
166          */
167         vrel(ip->i_devvp);
168     #ifdef QUOTA
169         for (i = 0; i < MAXQUOTAS; i++) {
170             if (ip->i_dquot[i] != NODQUOT) {
171                 dqrele(vp, ip->i_dquot[i]);
172                 ip->i_dquot[i] = NODQUOT;
173             }
174         }
175     #endif
176     #ifdef UFS_DIRHASH
177         if (ip->i_dirhash != NULL)
178             ufsdirhash_free(ip);
179     #endif
180         UFS_IFREE(ump, ip);
181         vp->v_data = 0;
182         return (0);
183     }

```

This looks like the most likely place.

Although everything points to line 181 of `ufs_reclaim`, we should consider if there aren't other ways to clear it. An obvious possibility might be a macro. We've already seen that `VTOI` refers to the field. Before we go on to look for references to `VTOI`, we should take stock:

- `v_data` is one of the most important fields in `struct vnode`: it's a pointer to the underlying inode. This in itself is interesting enough, but it also saves us some work: the macro `VTOI` above extracts the value of the `v_data` field; it doesn't point to the field itself. So we can't use this macro to zero out the field, and we don't need to look where it's referenced.
- The panic wouldn't have occurred if we hadn't set `INVARIANTS` when building the kernel. Maybe this happens all the time and nobody notices.
- On the other hand, we can't just drop the test: the reason for `INVARIANTS` is precisely to check for problems of this nature. In this case, we know that the code will work if we remove the check, since we always set `v_data` to `NULL` later in the function. But there's the possibility of a memory leak (what if the underlying inode hasn't been freed?). So we should continue looking.

- The error occurs when taking a vnode off the free list. This implies that vnodes on the free list should have `v_data` set to NULL. An obvious next place to look is when freeing the vnode to see if we get a vnode with `v_data` not set to NULL. We might use a breakpoint in the kernel debugger to do so.

Freeing vnodes

To check what we're freeing, we first need to know where we free the vnode. Typically the functions to allocate and free objects are close to each other in the same file, or in some cases in two different files in the same file. The function to allocate a vnode is called `getnewvnode`. We might expect the corresponding function to release it to be called `putoldvnode` or `freevnode` or some such. Looking through the code, we don't find anything like this. Instead, we find the function `vfree`:

```

3096  /*
3097  * Mark a vnode as free, putting it up for recycling.
3098  */
3099  void
3100  vfree(vp)
3101      struct vnode *vp;
3102  {
3103
3104      ASSERT_VI_LOCKED(vp, "vfree");
3105      mtx_lock(&vnode_free_list_mtx);
3106      KASSERT((vp->v_iflag & VI_FREE) == 0, ("vnode already free"));
3107      if (vp->v_iflag & VI_AGE) {
3108          TAILQ_INSERT_HEAD(&vnode_free_list, vp, v_freelist);
3109      } else {
3110          TAILQ_INSERT_TAIL(&vnode_free_list, vp, v_freelist);
3111      }
3112      freevnodes++;
3113      mtx_unlock(&vnode_free_list_mtx);
3114      vp->v_iflag &= ~VI_AGE;
3115      vp->v_iflag |= VI_FREE;
3116  }

```

This certainly frees a vnode. But is it really the function corresponding to `getnewvnode`? It inserts the vnode on the list `vnode_free_list`. Where does `getnewvnode` get its vnode from? Looking at the code again, we see:

```

738          TAILQ_REMOVE(&vnode_free_list, vp, v_freelist);
739          TAILQ_INSERT_TAIL(&vnode_free_list, vp, v_freelist);

```

So yes, this looks like the correct function.

One thing that's obviously missing in this function is a corresponding check for valid fields. Why should this be? It would be a lot easier to catch a culprit when freeing rather than when allocating, possibly much later.

One reason might be that the debugging code in `getnewvnode` was added to address a specific case of corruption on the free list. Another might be that it was just done in a hurry. We'll have to look further to find out which it is. At any rate, we now have some code which we can investigate with the kernel debugger. Let's look at the code again:

```

3102  {
3103

```

```

3104         ASSERT_VI_LOCKED(vp, "vfree");
3105         mtx_lock(&vnnode_free_list_mtx);
3106         KASSERT((vp->v_iflag & VI_FREE) == 0, ("vnnode already free"));

```

It's a good idea to avoid putting breakpoints within locked areas, because they might interact with the debugger. In the case of the lock `vnnode_free_list`, this is as good as impossible, but in the interests of consistency, we set the breakpoint before, on the call to `ASSERT_VI_LOCKED`:

```

(kgdb) b 3104
Breakpoint 1 at 0xc06293af: file /usr/src/sys/kern/vfs_subr.c, line 3104.
(kgdb) c
Continuing.

```

We know that the panics occur when running *fi nd*, so it seems a good idea to run it to get the system running:

```
$ find / > /dev/null
```

Shortly afterwards we hit the breakpoint:

```

Breakpoint 1, vfree (vp=0xc48ce924) at /usr/src/sys/kern/vfs_subr.c:3105
3105         mtx_lock(&vnnode_free_list_mtx);
(kgdb) bt
#0  vfree (vp=0xc48ce924) at /usr/src/sys/kern/vfs_subr.c:3105
#1  0xc0627b99 in vrele (vp=0xc48ce924) at /usr/src/sys/kern/vfs_subr.c:2000
#2  0xc0631025 in vn_close (vp=0xc48ce924, flags=0x1, file_cred=0x1, td=0x1) at /usr/src/sys/kern/vfs_vnops.c:328
#3  0xc0631cee in vn_closefile (fp=0x1, td=0xc3aa7930) at /usr/src/sys/kern/vfs_vnops.c:914
#4  0xc05c6b8b in fdrop_locked (fp=0xc39e07f8, td=0xc3aa7930) at /usr/src/sys/sys/file.h:288
#5  0xc05c5f84 in fdrop (fp=0xc39e07f8, td=0xc3aa7930) at /usr/src/sys/kern/kern_descrip.c:1879
#6  0xc05c5f57 in closef (fp=0xc39e07f8, td=0xc3aa7930) at /usr/src/sys/kern/kern_descrip.c:1865
#7  0xc05c57ff in fdfree (td=0xc3aa7930) at /usr/src/sys/kern/kern_descrip.c:1582
#8  0xc05cae03 in exit1 (td=0xc3aa7930, rv=0x0) at /usr/src/sys/kern/kern_exit.c:249
#9  0xc05ca9a4 in exit1 (td=0xc3aa7930, rv=0x116) at /usr/src/sys/kern/kern_exit.c:94
#10 0xc074ce57 in syscall (frame=
    {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf edi = 0xbfbfe440, tf esi = 0xbfbfe448, tf_ebp = 0xbfbfe408, tf_isp = 0xd7c08d74, tf_ebx = 0x281a778c, tf_edx = 0x281b9b60, tf_ecx = 0x281b9b60, tf_eax = 0x1, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0x28134e43, tf_cs = 0x1f, tf_eflags = 0x292, tf_esp = 0xbfbfe3ec, tf_ss = 0x2f})
    at /usr/src/sys/i386/i386/trap.c:1004
#11 0x28134e43 in ?? ()
#12 0x08049327 in ?? ()
(kgdb) p vp->v_data
$1 = (void *) 0xc52f1280

```

Hmm, hit it the first time round? That looks suspicious. How often does this the case? We can save a lot of work by giving commands to the breakpoint to display the field and continue automatically:

```

(gdb) comm 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>p vp->v_data          print the field
>c                    and continue
>(gdb) c              (hit ^D)
Continuing.

```

```

Breakpoint 1, vfree (vp=0xc519f410) at /usr/src/sys/kern/vfs_subr.c:3105
3105      mtx_lock(&vnnode_free_list_mtx);
$2 = (void *) 0xc620f5c0

Breakpoint 1, vfree (vp=0xc4d1eb2c) at /usr/src/sys/kern/vfs_subr.c:3105
3105      mtx_lock(&vnnode_free_list_mtx);
$3 = (void *) 0xc5b889d8

(etc)
Breakpoint 1, vfree (vp=0xc4eae410) at /usr/src/sys/kern/vfs_subr.c:3105
3105      mtx_lock(&vnnode_free_list_mtx);
$18 = (void *) 0xc46ddf00
---Type <return> to continue, or q <return> to quit---Quit

```

In other words, it seems that `v_data` is *always* set to something on entering this function. It's obviously normally reset before we get to `getnewvnode`. So where does that happen? Looking at the code above, the obvious place is in `ufs_reclaim`. When does that happen?

```

(gdb) l ufs_reclaim
139      ufs_reclaim(ap)
(etc)
180          UFS_IFREE(ump, ip);
181          vp->v_data = 0;
182          return (0);
183      }
(gdb) b 181
Breakpoint 2 at 0xc06fdcdc: file /usr/src/sys/ufs/ufs/ufs_inode.c, line 181.
(gdb) c
Continuing.

Breakpoint 2, ufs_reclaim (ap=0x1) at /usr/src/sys/ufs/ufs/ufs_inode.c:181
181          vp->v_data = 0;
(gdb) bt
#0  ufs_reclaim (ap=0x1) at /usr/src/sys/ufs/ufs/ufs_inode.c:181
#1  0xc0704ae7 in ufs_vnoperate (ap=0x4) at /usr/src/sys/ufs/ufs/ufs_vnops.c:2819
#2  0xc062855f in vclean (vp=0xc4d4b000, flags=0x8, td=0xc688a540) at vnode_if.h:981
#3  0xc062898d in vgonel (vp=0xc4d4b000, td=0xc688a540) at /usr/src/sys/kern/vfs_subr.c:2577
#4  0xc06255d9 in vtryrecycle (vp=0xc4d4b000) at /usr/src/sys/kern/vfs_subr.c:678
#5  0xc0625819 in getnewvnode (tag=0xc07bdc45 "ufs", mp=0xc46a3c00, vops=0x1, vpp=0x1)
    at /usr/src/sys/kern/vfs_subr.c:741
#6  0xc06f7cb0 in ffs_vget (mp=0xc46a3c00, ino=0x2a7a8, flags=0x2, vpp=0xe122da84)
    at /usr/src/sys/ufs/ufs/ufs_vfsops.c:1252
(etc)
(gdb) p vp->v_data
$19 = (void *) 0xc5115ec4

```

Frame 5 is `getnewvnode`. Let's look at that code more carefully:

```

693      int
694      getnewvnode(tag, mp, vops, vpp)
695          const char *tag;
696          struct mount *mp;
697          vop_t **vops;
698          struct vnode **vpp;
699      {
700          struct vnode *vp = NULL;
701          struct vpollinfo *pollinfo = NULL;
702
703          mtx_lock(&vnnode_free_list_mtx);
704
705          /*
706           * Try to reuse vnodes if we hit the max. This situation only
707           * occurs in certain large-memory (2G+) situations. We cannot
708           * attempt to directly reclaim vnodes due to nasty recursion
709           * problems.
710           */

```

```

711         while (numvnodes - freevnodes > desiredvnodes) {
712             if (vnlruproc_sig == 0) {
713                 vnlruproc_sig = 1; /* avoid unnecessary wakeups */
714                 wakeup(vnlruproc);
715             }
716             mtx_unlock(&vnnode_free_list_mtx);
717             tsleep(&vnlruproc_sig, PVFS, "vnlruwk", hz);
718             mtx_lock(&vnnode_free_list_mtx);
719         }
720
721         /*
722         * Attempt to reuse a vnode already on the free list, allocating
723         * a new vnode if we can't find one or if we have not reached a
724         * good minimum for good LRU performance.
725         */
726
727         if (freevnodes >= wantfreevnodes && numvnodes >= minvnodes) {
728             int error;
729             int count;
730
731             for (count = 0; count < freevnodes; count++) {
732                 vp = TAILQ_FIRST(&vnnode_free_list);
733
734                 KASSERT(vp->v_usecount == 0 &&
735                     (vp->v_iflag & VI_DOINGINACT) == 0,
736                     ("getnewvnode: free vnode isn't"));
737
738                 TAILQ_REMOVE(&vnnode_free_list, vp, v_freelist);
739                 TAILQ_INSERT_TAIL(&vnnode_free_list, vp, v_freelist);
740                 mtx_unlock(&vnnode_free_list_mtx);
741                 error = vtryrecycle(vp);      call ufs_reclaim via here
742                 mtx_lock(&vnnode_free_list_mtx);
743                 if (error == 0)
744                     break;
745                 vp = NULL;
746             }
747         }
748         if (vp) {
749             freevnodes--;
750             mtx_unlock(&vnnode_free_list_mtx);
751
752             #ifdef INVARIANTS
753             {
754                 if (vp->v_data)
755                     panic("cleaned vnode isn't");   panic here
756                 if (vp->v_numoutput)
757                     panic("Clean vnode has pending I/O's");
758                 if (vp->v_writecount != 0)
759                     panic("Non-zero write count");
760             }
761             #endif

```

In other words, the vnode doesn't get cleaned until just before it's reused—a "just in time" approach. But why didn't it work this time? At line 727 we check whether we should be reusing an existing vnode; if we don't, `vp` is still set to `NULL`, so obviously the condition applies. Next, in the loop starting on line 731 we look for a free vnode. If we find one, we try to clean it (line 741), and if that succeeds, we exit the loop and continue.

If that's the case, the variable `error` should be set to 0, but since it's local to the block starting at line 727, it no longer exists. It's possible that the registers still hold a clue, but for now we can assume that `vtryrecycle` returned 0. Something else must have gone wrong. The stack trace above shows the *correct* sequence; in the case of our panic, it doesn't seem to have worked quite like that. So where did things go wrong? Let's look at the called functions, starting with `vtryrecycle`.

vtryrecycle

vtryrecycle looks like this:

```

588      /*
589       * Check to see if a free vnode can be recycled. If it can,
590       * recycle it and return it with the vnode interlock held.
591       */
592      static int
593      vtryrecycle(struct vnode *vp)
594      {
595          struct thread *td = curthread;
596          vm_object_t object;
597          struct mount *vnmp;
598          int error;
599
600          /* Don't recycle if we can't get the interlock */
601          if (!VI_TRYLOCK(vp))
602              return (EWOULDBLOCK);
603          /*
604           * This vnode may found and locked via some other list, if so we
605           * can't recycle it yet.
606           */
607          if (vn_lock(vp, LK_INTERLOCK | LK_EXCLUSIVE | LK_NOWAIT, td) != 0)
608              return (EWOULDBLOCK);
609          /*
610           * Don't recycle if its filesystem is being suspended.
611           */
612          if (vn_start_write(vp, &vnmp, V_NOWAIT) != 0) {
613              VOP_UNLOCK(vp, 0, td);
614              return (EBUSY);
615          }

```

So far, we've done a few checks, but in any case where they fail, we return an obvious error number. Continuing,

```

616
617      /*
618       * Don't recycle if we still have cached pages.
619       */
620      if (VOP_GETVOBJECT(vp, &object) == 0) {
621          VM_OBJECT_LOCK(object);
622          if (object->resident_page_count ||
623              object->ref_count) {
624              VM_OBJECT_UNLOCK(object);
625              error = EBUSY;
626              goto done;

```

This looks alright as well, but we have to assume that the code at done does the right thing. We'll check that below.

```

627          }
628          VM_OBJECT_UNLOCK(object);
629      }
630      if (LIST_FIRST(&vp->v_cache_src)) {
631          /*
632           * note: nameileafonly sysctl is temporary,
633           * for debugging only, and will eventually be
634           * removed.
635           */
636          if (nameileafonly > 0) {
637              /*
638               * Do not reuse namei-cached directory
639               * vnodes that have cached
640               * subdirectories.
641              */

```

```

642             if (cache_leaf_test(vp) < 0) {
643                 error = EISDIR;
644                 goto done;
645             }
646         } else if (nameileafonly < 0 ||
647                 vmiodirenable == 0) {
648             /*
649              * Do not reuse namei-cached directory
650              * vnodes if nameileafonly is -1 or
651              * if VMIO backing for directories is
652              * turned off (otherwise we reuse them
653              * too quickly).
654              */
655             error = EBUSY;
656             goto done;
657         }
658     }
659     /*
660     * If we got this far, we need to acquire the interlock and see if
661     * anyone picked up this vnode from another list. If not, we will
662     * mark it with XLOCK via vgonel() so that anyone who does find it
663     * will skip over it.
664     */
665     VI_LOCK(vp);
666     if (VSHOULDBUSY(vp) && (vp->v_iflag & VI_XLOCK) == 0) {
667         VI_UNLOCK(vp);
668         error = EBUSY;
669         goto done;
670     }

```

In the code above, we see more cases of setting `error` and going to `done`. We still need to check, but there's nothing obviously wrong with the code.

```

671         mtx_lock(&vnode_free_list_mtx);
672         TAILQ_REMOVE(&vnode_free_list, vp, v_freelist);
673         vp->v_iflag &= ~VI_FREE;
674         mtx_unlock(&vnode_free_list_mtx);

```

Here we lock the vnode free list so that it doesn't change while we manipulate it, then we remove a vnode from it, then we unlock it.

```

675         vp->v_iflag |= VI_DOOMED;
676         if (vp->v_type != VBAD) {
677             VOP_UNLOCK(vp, 0, td);
678             vgonel(vp, td);
679             VI_LOCK(vp);
680         } else
681             VOP_UNLOCK(vp, 0, td);

```

Next we check the vnode and possibly clean it (if the type isn't set to VBAD). Based on what we've seen in `ufs_reclaim`, we'd expect to have to clean it.

But one thing looks strange here: if the vnode type is not VBAD, the code cleans it and locks the vnode pointer. If it is VBAD, however, it does not. Could this be the problem? We'll defer that question until we have finished reading the code.

```

682         vn_finished_write(vnmp);
683         return (0);

```

We obviously get this far. We wouldn't expect `vn_finished_write` to change much, but it's worth bearing it in mind in case we draw a blank elsewhere.

```

684     done:
685         VOP_UNLOCK(vp, 0, td);
686         vn_finished_write(vnmp);
687         return (error);
688     }

```

Finally, we come to `done`. As expected, it will always return an error indication. The occurrence of `vn_finished_write` here as well suggests that it's not going to do very much to the vnode, because in this case we can't use it.

So, at this point we have the question of the call to `VI_LOCK`. What does it do? It's a maze of twisty little macros: in file `vnode.h` we read:

```

397     #define VI_LOCK(vp)         mtx_lock(&(vp)->v_interlock)

```

So we're taking a mutex. This code is specific to FreeBSD versions 5 and 6, so it's new, and there's a better than average chance that the problem could be here, rather than with code which has been with BSD for decades.

`mtx_lock` is described in *mutex(9)*:

```

void
mtx_lock(struct mtx *mutex);
...
The mtx_lock() function acquires a MTX_DEF mutual exclusion lock on
behalf of the currently running kernel thread. If another kernel thread
is holding the mutex, the caller will be disconnected from the CPU until
the mutex is available (i.e. it will sleep).

```

This doesn't tell us how to decide whether a mutex has been acquired or not when looking at it in a dump. Before jumping into the mutex implementation, let's take a look at what we have. From above (page 70) we see:

```

v_interlock = {
  mtx_object = {
    lo_class = 0xc080c83c,
    lo_name = 0xc07ba2fb "vnode interlock",
    lo_type = 0xc07ba2fb "vnode interlock",
    lo_flags = 0x30000,
    lo_list = {
      tqe_next = 0x0,
      tqe_prev = 0x0
    },
    lo_witness = 0x0
  },
  mtx_lock = 0xc5333bd0,
  mtx_recurse = 0x0
},

```

This is the field that's referenced to in `VI_LOCK`. The obvious field to look at is `mtx_lock`. But what is it? If it's a pointer to the locker, then it's obviously locked. But it might be a pointer to a lock structure, in which case we'd need to look at the structure. So we don't get away without looking in *_mutex.h*, where we find:

```

34     /*
35      * Sleep/spin mutex.
36      */
37     struct mtx {
38         struct lock_object      mtx_object; /* Common lock properties. */

```

```

39          volatile uintptr_t      mtx_lock; /* Owner and flags. */
40          volatile u_int          mtx_recurse; /* Number of recursive holds. */
60      };

```

This doesn't help very much: the field `mtx_lock` has deliberately been made non-transparent (type `uintptr_t`). About the only thing that it does seem to imply is that the mutex has an owner if the field is non-zero. So let's assume that this is correct behaviour. We might find more information about the information if we knew what `mtx_lock` is pointing to.

We could continue in this manner for some time, but it's gradually moving into the "too hard" department. Are we even on the right track? If so, we can expect to pass via line 681 of `vtryrecycle` with `vp->v_data` not set to 0. We should be able to do that with a conditional breakpoint:

```

(gdb) l vtryrecycle
589      * Check to see if a free vnode can be recycled. If it can,
590      * recycle it and return it with the vnode interlock held.
... (do this to ensure we're pointing into the right function)
(gdb) l 681
680          } else
681              VOP_UNLOCK(vp, 0, td);
682          vn_finished_write(vnmp);
(gdb) b 681 if vp->v_data != 0
Breakpoint 3 at 0xc062561c: file /usr/src/sys/kern/vfs_subr.c, line 681.
(gdb) c
Continuing.

```

There's a theoretical danger with this, though: what if the optimizer has coalesced the code? In this case it doesn't matter much, because we would want to stop in any case if `vp->v_data` is not 0. The only issue is that breakpoints slow down execution, especially if they're not taken and happen frequently; they can slow down by a couple of orders of magnitude, even if you're debugging via firewire.

With this breakpoint in place, we try to provoke the problem:

```
$ find /src -type f > /dev/null
```

The `type f` isn't important because it's looking for files; it's important because it makes `find` look at the inode. If we didn't do that, it would just read the directories, and the problem would probably not occur. During this, we get the following messages:

```

$ find /src -type f >/dev/null
find: /src/Ports/LEMIS/logwatch/copyright: Bad file descriptor
find: /src/Ports/LEMIS/logwatch/pkginfo: Bad file descriptor
find: /src/Ports/LEMIS/logwatch/preinstall: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/doc: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/coffgrok.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/coffgrok.h: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/config.in: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/configure.in: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/configure.tgt: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/debug.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/debug.h: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/deflex.l: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/defparse.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/defparse.h: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/defparse.y: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/dep-in.sed: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/dlltool.h: Bad file descriptor

```

```

find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/dllwrap.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/emul_vanilla.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/filemode.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/is-ranlib.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/is-strip.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/maybe-ranlib.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/maybe-strip.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/nm.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/not-ranlib.c: Bad file descriptor
find: /src/FreeBSD/BFS/src/contrib/binutils/binutils/not-strip.c: Bad file descriptor
panic: cleaned vnode isn't
at line 755 in file /usr/src/sys/kern/vfs_subr.c
cpuid = 0;
Debugger("panic")

```

Looking at the directory `/src/FreeBSD/BFS/src/contrib/binutils/binutils/`, after rebooting, we see:

```

$ ls -il /src/FreeBSD/BFS/src/contrib/binutils/binutils
ls: coffgkok.c: Bad file descriptor
(the same messages as above)
total 1517313
3742650 -rw-r--r--  1 grog  lemis           89000 Dec  2  2002 ChangeLog
3742651 -rw-r--r--  1 grog  lemis        192257 Jan 27  2002 ChangeLog-9197
3742652 -rw-r--r--  1 grog  lemis         65072 May 28  2001 ChangeLog-9899
...
3742673 -rw-r--r--  1 grog  24             46498 Oct 11  2002 coffdump.c
3742697 lr--r-x--T  2 root  340        1093514658 Dec  2  2002 configure -> /ucb/*) ;;?
*)?          # OSF1 and SCO ODT 3.0 have their own names for install.? # Don't use
installbsd from OSF since it installs stuff as root? # by default.? for ac_
prog in ginstall scoinst install; do? if test -f $ac_dir/$ac_prog; then?? if t
est $ac_prog = install &&? grep dspmsg $ac_dir/$ac_prog >/dev/null 2>&1; th
en?? # AIX install. It has an incompatible calling convention.?? :?? else??
ac_cv_path_install="$ac_dir/$ac_prog -c"?? break 2?? fi??fi? done? ;;?
esac? done? IFS="$ac_save_IFS"??fi? if test "${ac_cv_path_install+set}" = set; t
hen? INSTALL="$ac_cv_path_install"? else? # As a last resort, use the slow shel
l script. We don't cache a? # path for INSTALL within a source directory, because
that will? # break other packages using the cache if that directory is? # remove
d, or if the path is relative.? INSTALL="$ac_install_sh"? fi?fi?echo "$ac_t"$INST
ALL" 1>&6??# Use test -z because SunOS4 sh mishandles braces in ${
3742707 b-wsrws--- 1 root 184           75, 0x01c80018 Oct 11  2002 dlltool.c
3742720 -rw-r--r--  1 grog  lemis         75533 Oct 11  2002 objcopy.c
...

```

There are a number of things to note here:

- Our assumption that the problem might be related to the locking calls at the end of `vtryre-cycle` have proved incorrect. We didn't hit the breakpoint, so it can't be that.
- Obviously the directory is badly broken.
- The inode numbers (the first column of the listing) are all closely related. This is relatively common in directories that are created at once and that subsequently don't get changed very much.
- The files that are reported as "Bad file descriptor" (which indicates that the system call returned an `EBADF` error code) don't appear in the list. This is a decision of `ls` rather than of the kernel.
- This has been happening for some time. Why? At boot time we see the messages:

```

WARNING: / was not properly dismounted
WARNING: /src was not properly dismounted
WARNING: /blackwater/home was not properly dismounted

```

After that, the system continues running without running *fsck* on the disk. That in itself is not surprising: FreeBSD version 5 offers background *fsck* for all file systems except root. On investigation, though, we discover that it doesn't run *fsck* at all. In */etc/fstab* we find:

| # Device | Mountpoint | FStype | Options | Dump | Pass# |
|----------------|------------|--------|---------|------|-------|
| # echunga:/src | /src | nfs | rw | 0 | 0 |
| /dev/adlh | /src | ufs | rw | 0 | 0 |

The problem here was that the file system had been moved to this system "temporarily" during a rebuild, so the file system type changed from *nfs* to *ufs*. You don't run *fsck* on NFS file system, so the *Pass#* field is 0; we forgot to change it.

The missing *fsck* makes it easier to understand the problem; it doesn't mean we're done, though. As stated at the beginning of this text,

Good kernels should not fail. They must protect themselves against a number of external influences, including hardware failure, both deliberately and accidentally badly written user programs, and kernel programming errors. In some cases, of course, there is no way a kernel can recover, for example if the only processor fails. On the other hand, a good kernel should be able to protect itself from badly written user programs.

So what do we do next? We now have more information. We can:

1. Continue with our examination of the call sequence to *ufs_reclaim* and find out what went wrong.
2. Now that we know that some system call returned *EBADF*, we can check why that happened.
3. The *vp->v_data* field should point to an inode. Now that we have an inode number, it's easy to check whether the numbers are in the general area of what we saw in this directory.

We should do all of these things, but the easiest thing to do is to look at *vp->v_data*, so we'll do that first:

```
(kgdb) f 9
#9  0xc06259b0 in getnewvnode (tag=0xc07bdc45 "ufs", mp=0xc45bf400, vops=0x0, vpp=0x0)
    at /usr/src/sys/kern/vfs_subr.c:785
785      KASSERT(vp->v_dirtyblkroot == NULL, ("dirtyblkroot not NULL"));
(kgdb) p vp->v_data
$1 = (void *) 0xc5989c08
(kgdb) p *(struct inode *)vp->v_data
$2 = {
  i_hash = {
    le_next = 0x0,
    le_prev = 0xc454f050
  },
  i_nextsnap = {
    tqe_next = 0x0,
    tqe_prev = 0x0
  },
  i_vnode = 0xc5c03410,
  i_ump = 0xc463d000,
  i_flag = 0x20,
  i_dev = 0xc45c2800,
  i_number = 0x391bf8,
  i_effnlink = 0x2,
  i_fs = 0xc4603000,
  i_dquot = {0x0, 0x0},
```

```

i_modrev = 0xefd6515a6286,
i_lockf = 0x0,
i_count = 0x0,
i_endoff = 0x0,
i_diroff = 0x0,
i_offset = 0x0,
i_ino = 0x0,
i_reclen = 0x0,
i_un = {
    dirhash = 0x0,
    snapblklist = 0x0
},
i_ea_area = 0x0,
i_ea_len = 0x0,
i_ea_error = 0x0,
i_mode = 0xf502,
i_nlink = 0x2,
i_size = 0x412db5a3,
i_flags = 0x0,
i_gen = 0x68f46009,
i_uid = 0x0,
i_gid = 0x17c,
dinode_u = {
    din1 = 0xc4d8bd00,
    din2 = 0xc4d8bd00
}
}

```

Some of these fields (highlighted above) give us clues:

- `i_vnode` is a pointer to the parent vnode. If we're correct that this is really supposed to be an inode (and given the content there are good reasons that speak against it), this field will point to `vp`. It does:

```

(kgdb) p vp
$5 = (struct vnode *) 0xc5c03410

```

- The inode number should be close to the directory we're looking at. It's in hex, which doesn't make things any easier. Looking at it in decimal (d modifier), we see:

```

(kgdb) p/d (struct inode *)vp->v_data->i_number
Attempt to dereference a generic pointer.
(kgdb) p/d ((struct inode *)vp->v_data)->i_number
$4 = 3742712

```

So the number is in the same range, but a quick *grep* shows that it's not present in the directory listing. There's a very good chance that it's one of the EBADF directory entries.

The first attempt to list the value failed because *gdb* tried to take the entire expression `vp->v_data->i_number` as an inode pointer; since it's a scalar, that can't work. To get the correct results, we need to tell *gdb* which part of the expression is the inode pointer by putting brackets around it.

- The file mode looks funny. Looking at it in octal (which is the way it's done for this particular field), we find:

```

(kgdb) p/o ((struct inode *)vp->v_data)->i_mode
$7 = 0172402

```

The first four bits of the file mode (0170000) specify the file type. It's described in

sys/stat.h:

```
#define S_IFMT    0170000    /* type of file mask */
#define S_IFIFO   0010000    /* named pipe (fifo) */
#define S_IFCHR   0020000    /* character special */
#define S_IFDIR   0040000    /* directory */
#define S_IFBLK   0060000    /* block special */
#define S_IFREG   0100000    /* regular */
#define S_IFLNK   0120000    /* symbolic link */
#define S_IFSOCK  0140000    /* socket */
```

In other words, (0170000) is not a valid file type (though it is defined above as a mask, since all bits are set). This is probably why the system returned EBADF. The permissions part (the last 16 bits) look very unlikely too.

- `i_size` is the file size, a little bit over 1 GB. That's not impossible, but unlikely. In combination with the other fields, we can assume that this inode is not really an inode at all.

So where do we look next? We can continue looking at our call chain to `ufs_reclaim`, or we can go looking for where the EBADF comes from. Let's do both, in that sequence. Based on not hitting our breakpoint in `vtryrecycle`, we know that we called `vgonel`.

vgonel

Looking at `vgonel`, we see:

```
2554    /*
2555     * vgone, with the vp interlock held.
2556     */
2557    void
2558    vgone(vp, td)
2559        struct vnode *vp;
2560        struct thread *td;
2561    {
2562        /*
2563         * If a vgone (or vclean) is already in progress,
2564         * wait until it is done and return.
2565         */
2566        ASSERT_VI_LOCKED(vp, "vgone");
2567        if (vp->v_iflag & VI_XLOCK) {
2568            vp->v_iflag |= VI_XWANT;
2569            msleep(vp, VI_MTX(vp), PINOD | PDROP, "vgone", 0);
2570            return;
2571        }
2572        vx_lock(vp);
2573
2574        /*
2575         * Clean out the filesystem specific data.
2576         */
2577        vclean(vp, DOCLOSE, td);
```

There's not very much that can go wrong there, unless again we have problems with locking. That seems unlikely, though.

There's a lot more code after this call, but we're unlikely to hit it. It has some interesting comments, however:

```
2578        VI_UNLOCK(vp);
2579
2580        /*
2581         * If special device, remove it from special device alias list
```



```

2582         * if it is on one.
2583         */
2584     VI_LOCK(vp);
2585     if (vp->v_type == VCHR && vp->v_rdev != NODEV) {
2586         mtx_lock(&spechash_mtx);
2587         SLIST_REMOVE(&vp->v_rdev->si_hlist, vp, vnode, v_specnext);
2588         vp->v_rdev->si_usecount -= vp->v_usecount;
2589         mtx_unlock(&spechash_mtx);
2590         dev_rel(vp->v_rdev);
2591         vp->v_rdev = NULL;
2592     }
2593
2594     /*
2595     * If it is on the freelist and not already at the head,
2596     * move it to the head of the list. The test of the
2597     * VDOOMED flag and the reference count of zero is because
2598     * it will be removed from the free list by getnewvnode,
2599     * but will not have its reference count incremented until
2600     * after calling vgone. If the reference count were
2601     * incremented first, vgone would (incorrectly) try to
2602     * close the previous instance of the underlying object.
2603     */
2604     if (vp->v_usecount == 0 && !(vp->v_iflag & VI_DOOMED)) {
2605         mtx_lock(&vnode_free_list_mtx);
2606         if (vp->v_iflag & VI_FREE) {
2607             TAILQ_REMOVE(&vnode_free_list, vp, v_freelist);
2608         } else {
2609             vp->v_iflag |= VI_FREE;
2610             freevnodes++;
2611         }
2612         TAILQ_INSERT_HEAD(&vnode_free_list, vp, v_freelist);
2613         mtx_unlock(&vnode_free_list_mtx);
2614     }
2615
2616     vp->v_type = VBAD;
2617     vx_unlock(vp);
2618     VI_UNLOCK(vp);
2619 }

```

vclean

The name `vclean` suggests that we should find something relating to our problem in this function. It's quite long, from line 2314 to line 2430. Where's the call to `ufs_reclaim`? It doesn't show up in the code. Looking at the backtrace, we find:

```

#1  0xc0704ae7 in ufs_vnoperate (ap=0x4) at /usr/src/sys/ufs/ufs/ufs_vnops.c:2819
#2  0xc062855f in vclean (vp=0xc4d4b000, flags=0x8, td=0xc688a540) at vnode_if.h:981

```

Huh? The second frame should be in `vclean`, and that's what *gdb* claims, but the file name and the line number are all wrong. Looking at line 981 of `vnode_if.h` (in the kernel build tree), we find:

```

970     static __inline int VOP_RECLAIM(
971         struct vnode *vp,
972         struct thread *td)
973     {
974         struct vop_reclaim_args a;
975         int rc;
976         a.a_desc = VDESC(vop_reclaim);
977         a.a_vp = vp;
978         a.a_td = td;
979         ASSERT_VI_UNLOCKED(vp, "VOP_RECLAIM");
980         ASSERT_VOP_UNLOCKED(vp, "VOP_RECLAIM");
981         rc = VCALL(vp, VOFFSET(vop_reclaim), &a);

```

```

982         CTR2(KTR_VOP, "VOP_RECLAIM(vp 0x%lX, td 0x%lX)", vp, td);
983     if (rc == 0) {
984         ASSERT_VI_UNLOCKED(vp, "VOP_RECLAIM");
985         ASSERT_VOP_UNLOCKED(vp, "VOP_RECLAIM");
986     } else {
987         ASSERT_VI_UNLOCKED(vp, "VOP_RECLAIM");
988         ASSERT_VOP_UNLOCKED(vp, "VOP_RECLAIM");
989     }
990     return (rc);
991 }

```

Yes, the indentation is like that. These are automatically generated inline functions. So what we should be looking for is an invocation of the macro `VOP_RECLAIM`. The `VCALL` macro calls the correct clean function for the file system in question.

```

2314     /*
2315      * Disassociate the underlying filesystem from a vnode.
2316      */
2317     static void
2318     vclean(vp, flags, td)
2319         struct vnode *vp;
2320         int flags;
2321         struct thread *td;
2322     {
2323         int active;
2324
2325         ASSERT_VI_LOCKED(vp, "vclean");
2326         /*
2327          * Check to see if the vnode is in use. If so we have to reference it
2328          * before we clean it out so that its count cannot fall to zero and
2329          * generate a race against ourselves to recycle it.
2330          */
2331         if ((active = vp->v_usecount))
2332             v_incr_usecount(vp, 1);

```

Here's a situation that we hadn't expected: it's obviously valid to call this function with an active vnode. This may be of relevance. It's tempting to look at the value of `vp->v_usecount` here, but that doesn't help much; first we need to see if it gets changed.

```

2334     /*
2335      * Even if the count is zero, the VOP_INACTIVE routine may still
2336      * have the object locked while it cleans it out. The VOP_LOCK
2337      * ensures that the VOP_INACTIVE routine is done with its work.
2338      * For active vnodes, it ensures that no other activity can
2339      * occur while the underlying object is being cleaned out.
2340      */
2341     VOP_LOCK(vp, LK_DRAIN | LK_INTERLOCK, td);
2342
2343     /*
2344      * Clean out any buffers associated with the vnode.
2345      * If the flush fails, just toss the buffers.
2346      */
2347     if (flags & DCLOSE) {

```

`vgone1` calls `vclean` with `DCLOSE` set, so we execute the following code, which invalidates the buffers associated with the vnode. We're not too interested in this at the moment; `vinvalbuf` can either return an errors or panic if it fails. We didn't have a panic here, and the return value isn't checked the second time round, so we obviously got past this point. It would, however, make an interesting area to look at in more detail.

```

2348         struct buf *bp;
2349         bp = TAILQ_FIRST(&vp->v_dirtyblkhd);

```

```

2350             if (bp != NULL)
2351                 (void) vn_write_suspend_wait(vp, NULL, V_WAIT);
2352             if (vinvalbuf(vp, V_SAVE, NOCRED, td, 0, 0) != 0)
2353                 vinvalbuf(vp, 0, NOCRED, td, 0, 0);
2354         }
2355
2356     VOP_DESTROYVOBJECT(vp);
2357
2358     /*
2359     * Any other processes trying to obtain this lock must first
2360     * wait for VXLOCK to clear, then call the new lock operation.
2361     */
2362     VOP_UNLOCK(vp, 0, td);
2363
2364     /*
2365     * If purging an active vnode, it must be closed and
2366     * deactivated before being reclaimed. Note that the
2367     * VOP_INACTIVE will unlock the vnode.
2368     */
2369     if (active) {
2370         if (flags & DOCLOSE)
2371             VOP_CLOSE(vp, FNONBLOCK, NOCRED, td);
2372         VI_LOCK(vp);
2373         if ((vp->v_iflag & VI_DOINGINACT) == 0) {
2374             vp->v_iflag |= VI_DOINGINACT;
2375             VI_UNLOCK(vp);
2376             if (vn_lock(vp, LK_EXCLUSIVE | LK_NOWAIT, td) != 0)
2377                 panic("vclean: cannot relock.");
2378             VOP_INACTIVE(vp, td);
2379             VI_LOCK(vp);
2380             KASSERT(vp->v_iflag & VI_DOINGINACT,
2381                 ("vclean: lost VI_DOINGINACT"));
2382             vp->v_iflag &= ~VI_DOINGINACT;
2383         }
2384         VI_UNLOCK(vp);
2385     }

```

The code above may or may not have been called; there's no way to know. At the time we panic, we have:

```

(kgdb) p vp->v_iflag
$11 = 0x80

```

In *vnode.h*, we see:

```

220     #define VI_DOOMED          0x0080 /* This vnode is being recycled */

```

So again, it looks as if this condition either didn't apply, or that the purge was successful.

```

2386     /*
2387     * Reclaim the vnode.
2388     */
2389     if (VOP_RECLAIM(vp, td))
2390         panic("vclean: cannot reclaim");

```

This is the macro that calls `ufs_vnoperate`. To be sure, we check the addresses:

```

(kgdb) info line 2389
Line 2389 of "/usr/src/sys/kern/vfs_subr.c" is at address 0xc0628566 <vclean+510> but
contains no code.

```

That doesn't help. There should be code there, but the address given suggests that the code we're looking for is earlier in the file. Going back to the next line with code in it, 2384, we find:

(kgdb) info line 2384

Line 2384 of "/usr/src/sys/kern/vfs_subr.c" starts at address 0xc062852b <vclean+451> and ends at 0xc0628540 <vclean+472>.

That doesn't make sense; it's probably part of the general problem of the inline functions. In any case, we're obviously in the right place, more or less.

Again, there's nothing in this function that would explain why we wouldn't get this far. We'll keep the rest of the function for future reference and move on to ufs_vnoperate:

```

2391
2392         if (active) {
2393             /*
2394              * Inline copy of vrele() since VOP_INACTIVE
2395              * has already been called.
2396              */
2397             VI_LOCK(vp);
2398             v_incr_usecount(vp, -1);
2399             if (vp->v_usecount <= 0) {
2400 #ifdef INVARIANTS
2401                 if (vp->v_usecount < 0 || vp->v_writecount != 0) {
2402                     vprint("vclean: bad ref count", vp);
2403                     panic("vclean: ref cnt");
2404                 }
2405 #endif
2406                 if (VSHOULDFREE(vp))
2407                     vfree(vp);
2408             }
2409             VI_UNLOCK(vp);
2410         }
2411         /*
2412          * Delete from old mount point vnode list.
2413          */
2414         if (vp->v_mount != NULL)
2415             insmntque(vp, (struct mount *)0);
2416         cache_purge(vp);
2417         VI_LOCK(vp);
2418         if (VSHOULDFREE(vp))
2419             vfree(vp);
2420
2421         /*
2422          * Done with purge, reset to the standard lock and
2423          * notify sleepers of the grim news.
2424          */
2425         vp->v_vnlock = &vp->v_lock;
2426         vp->v_op = dead_vnodeop_p;
2427         if (vp->v_pollinfo != NULL)
2428             vn_pollgone(vp);
2429         vp->v_tag = "none";
2430     }

```

ufs_vnoperate

ufs_vnoperate is mercifully short:

```

2813     int
2814     ufs_vnoperate(ap)
2815     struct vop_generic_args /* {
2816         struct vnodeop_desc *a_desc;
2817     } */ *ap;
2818     {
2819         return (VOCALL(ufs_vnodeop_p, ap->a_desc->vdesc_offset, ap));
2820     }

```

From the stack trace, we know where we're going next:

ufs_reclaim

```

2813     int
139     ufs_reclaim(ap)
140         struct vop_reclaim_args /* {
141             struct vnode *a_vp;
142             struct thread *a_td;
143         } */ *ap;
144     {
145         struct vnode *vp = ap->a_vp;
146         struct inode *ip = VTOI(vp);
147         struct ufsmount *ump = ip->i_ump;
151
152         VI_LOCK(vp);
153         if (prtactive && vp->v_usecount != 0)
154             vprint("ufs_reclaim: pushing active", vp);
155         VI_UNLOCK(vp);
156         if (ip->i_flag & IN_LAZYMOD) {
157             ip->i_flag |= IN_MODIFIED;
158             UFS_UPDATE(vp, 0);
159         }
160         /*
161          * Remove the inode from its hash chain.
162          */
163         ufs_ihashrem(ip);
164         /*
165          * Purge old data structures associated with the inode.
166          */
167         vrel(ip->i_devvp);
180         UFS_IFREE(ump, ip);
181         vp->v_data = 0;
182         return (0);
183     }

```

This listing omits some *#ifdef*ed code, thus the jumps in the line numbers.

This is fairly simple code. How can we avoid setting `vp->v_data` here? Maybe it's a race condition after all. On the other hand, it's possible that we've missed something. Let's go back and look at what happens when we return `EBADF`. *Something* in the `vnode` must be different, or we'd handle it the same when picking it off the free list.

The obvious place to look at would be in `vfree`, which we've already seen. Now that we know which files are triggering the problem, we can be much more selective. First we set a breakpoint on `vfree`, then we trigger it by looking at a couple of files. First, we find a good file in the list on page 95. We'll choose this one:

```
3742650 -rw-r--r--  1 grog  lemis           89000 Dec  2  2002 ChangeLog
```

We set a breakpoint to catch exactly this inode:

```

(gdb) b vfree if ((struct inode *) vp->v_data)->i_number == 3742650
Breakpoint 1 at 0xc06293af: file /usr/src/sys/kern/vfs_subr.c, line 3105.
(gdb) c
# ls -l ls -li /src/FreeBSD/BFS/src/contrib/binutils/binutils/size.c
Breakpoint 1, vfree (vp=0xc49ebc30) at /usr/src/sys/kern/vfs_subr.c:3105
3105         mtx_lock(&vnode_free_list_mtx);
(gdb) p *vp
$1 = {
  v_interlock = {
    mtx_object = {
      lo_class = 0xc080c83c,
      ...
    }
  }
}
(gdb) p *((struct inode *) vp->v_data)

```

```

$2 = {
  i_hash = {
...
(gdb) c
Continuing.

Breakpoint 1, vfree (vp=0xc49ebc30) at /usr/src/sys/kern/vfs_subr.c:3105
3105      mtx_lock(&vnode_free_list_mtx);
(gdb) p *vp
$3 = {
  v_interlock = {
    mtx_object = {
      lo_class = 0xc080c83c,
...
(gdb) p *((struct inode *) vp->v_data)
$4 = {
  i_hash = {
...
(gdb) c
Continuing.
(gdb) c
Continuing.
3742736 -rw-r--r--  1 grog  lemis  14176 Oct 11  2002 /src/FreeBSD/BFS/src/contrib/bin
utils/binutils/size.c

```

Rather to our surprise, we hit the breakpoint twice. The commands we input give a lot of output, more than our tired eyes can handle. We save it in a file, *goodvnode*, for later comparison.

Next, we do the same with the vnode that we found in the previous section, number 3742712, and do almost the same thing. We don't know the name of this file, since *ls* didn't tell us, but we can confirm that it's in this directory by listing it:

```

(gdb) b vfree if ((struct inode *) vp->v_data)->i_number == 3742712
Note: breakpoint 1 also set at pc 0xc06293af.
Breakpoint 2 at 0xc06293af: file /usr/src/sys/kern/vfs_subr.c, line 3105.

```

This was really a mistake. It would have been easier to change the condition of breakpoint 1 rather than creating a new one. Breakpoints take time even if the condition doesn't apply, and two take twice as long as one. As a result, it takes several seconds before we hit our breakpoint:

```

(gdb) c
Continuing.

Breakpoint 2, vfree (vp=0xc49ee104) at /usr/src/sys/kern/vfs_subr.c:3105
3105      mtx_lock(&vnode_free_list_mtx);
(gdb) p *vp
$5 = {
  v_interlock = {
    mtx_object = {
      lo_class = 0xc080c83c,
...
(gdb) p *((struct inode *) vp->v_data)
$6 = {
  i_hash = {
    le_next = 0x0,
...

```

Again we save the output (this time only one set) in a file, this time called *badvnode*. When we have both, we run *diff* against them:

```

--- goodvnode   Fri Oct  1 17:26:28 2004
+++ badvnode    Fri Oct  1 17:17:24 2004
@@ -1,158 +1,13 @@

```

```

-Breakpoint 1, vfree (vp=0xc49ebc30) at /usr/src/sys/kern/vfs_subr.c:3105
-3105          mtx_lock(&vnode_free_list_mtx);
-(gdb) p *vp
-$1 = {
-  v_interlock = {
...

```

For some reason, *diff* has decided that the second set of outputs for the “good” vnode compares better with the output for the “bad” vnode; this may mean that they’re different, or it may just be the way *diff* handles this occurrence.

```

-Breakpoint 1, vfree (vp=0xc49ebc30) at /usr/src/sys/kern/vfs_subr.c:3105
+Breakpoint 2, vfree (vp=0xc49eel04) at /usr/src/sys/kern/vfs_subr.c:3105
3105          mtx_lock(&vnode_free_list_mtx);
(gdb) p *vp
-$3 = {
+$5 = {
    v_interlock = {
        mtx_object = {
            lo_class = 0xc080c83c,
@@ -165,7 +20,7 @@
        },
        lo_witness = 0x0
    },
    mtx_lock = 0xc47b7150,
+   mtx_lock = 0xc46fbd20,
    mtx_recurse = 0x0
},
    v_iflag = 0x0,
@@ -175,13 +30,13 @@
    v_holdcnt = 0x0,
    v_cleanblkhd = {
        tqh_first = 0x0,
-   tqh_last = 0xc49ebc68
+   tqh_last = 0xc49eel3c
    },
    v_cleanblkroot = 0x0,
    v_cleanbufcnt = 0x0,
    v_dirtyblkhd = {
        tqh_first = 0x0,
-   tqh_last = 0xc49ebc78
+   tqh_last = 0xc49eel4c
    },
    v_dirtyblkroot = 0x0,
    v_dirtybufcnt = 0x0,
@@ -205,21 +60,21 @@
    },
    v_freelist = {
        tqe_next = 0x0,
-   tqe_prev = 0xc49c2290
+   tqe_prev = 0x0
    },
    v_nmmtvnodes = {
        tqe_next = 0x0,
-   tqe_prev = 0xc49e78b0
+   tqe_prev = 0xc49ee6a8
    },
    v_synclist = {
        le_next = 0x0,
        le_prev = 0x0
    },
},

```

Everything we’ve seen so far is more coincidental. They’re linkage and lock addresses. This would happen with two different good vnodes as well, so we can ignore them. Next, however, is something more important:

```
- v_type = VREG,
+ v_type = VBAD,
```

The bad vnode already has its type field set to VBAD. We know these values: we've code which is conditional on the type field being VBAD. This is definitely interesting.

The differences in the remainder of the vnode are also different pointers. Nothing stands out. We've already looked at the inode structure for the bad inode; the diffs show nothing further apart from the expected differences in pointers and other fields.

So it looks as if the problem is related to the end of `vtryrecycle` after all. But we set a breakpoint there on the *else* clause of the condition. If our current assumptions are correct, we should have hit it. Why didn't we? Let's take another look:

```
(gdb) i 11 681
Line 681 of "/usr/src/sys/kern/vfs_subr.c" is at address 0xc062561c <vtryrecycle+672>
but contains no code.
```

OK, that's a giveaway: the optimizer has tricked us again. Where are we?

```
(gdb) i 11 682
Line 682 of "/usr/src/sys/kern/vfs_subr.c" starts at address 0xc062561c <vtryrecycle+672>
and ends at 0xc0625624 <vtryrecycle+680>.
(gdb) i 11 676
Line 676 of "/usr/src/sys/kern/vfs_subr.c" starts at address 0xc062559d <vtryrecycle+545>
and ends at 0xc06255a9 <vtryrecycle+557>.
(gdb) x/50i 0xc062559d
0xc062559d <vtryrecycle+545>: add    $0x10,%esp
0xc06255a0 <vtryrecycle+548>: cmpl   $0x8,0xa0(%ebx)
0xc06255a7 <vtryrecycle+555>: je     0xc06255f0 <vtryrecycle+628>
0xc06255a9 <vtryrecycle+557>: movl   $0xc0852220,0xffffffffe4(%ebp)
0xc06255b0 <vtryrecycle+564>: mov     %ebx,0xffffffffe8(%ebp)
0xc06255b3 <vtryrecycle+567>: movl   $0x0,0xffffffffec(%ebp)
0xc06255ba <vtryrecycle+574>: mov     %esi,0xfffffffff0(%ebp)
0xc06255bd <vtryrecycle+577>: mov     0xd4(%ebx),%eax
0xc06255c3 <vtryrecycle+583>: lea     0xffffffffe4(%ebp),%edx
0xc06255c6 <vtryrecycle+586>: push    %edx
0xc06255c7 <vtryrecycle+587>: mov     0xc0852220,%edx
0xc06255cd <vtryrecycle+593>: call    *(%eax,%edx,4)
0xc06255d0 <vtryrecycle+596>: mov     %esi,(%esp,1)
0xc06255d3 <vtryrecycle+599>: push    %ebx
0xc06255d4 <vtryrecycle+600>: call    0xc0628950 <vgonel>
0xc06255d9 <vtryrecycle+605>: push    $0x2a7
0xc06255de <vtryrecycle+610>: push    $0xc07ba1fb
0xc06255e3 <vtryrecycle+615>: push    $0x0
0xc06255e5 <vtryrecycle+617>: push    %ebx
0xc06255e6 <vtryrecycle+618>: call    0xc05d5ef0 <_mtx_lock_flags>
0xc06255eb <vtryrecycle+623>: add     $0x18,%esp
0xc06255ee <vtryrecycle+626>: jmp     0xc062561c <vtryrecycle+672>
0xc06255f0 <vtryrecycle+628>: movl   $0xc0852220,0xffffffffe4(%ebp)
0xc06255f7 <vtryrecycle+635>: mov     %ebx,0xffffffffe8(%ebp)
0xc06255fa <vtryrecycle+638>: movl   $0x0,0xffffffffec(%ebp)
0xc0625601 <vtryrecycle+645>: mov     %esi,0xfffffffff0(%ebp)
0xc0625604 <vtryrecycle+648>: mov     0xd4(%ebx),%eax
0xc062560a <vtryrecycle+654>: lea     0xffffffffe4(%ebp),%edx
0xc062560d <vtryrecycle+657>: push    %edx
0xc062560e <vtryrecycle+658>: mov     0xc0852220,%edx
0xc0625614 <vtryrecycle+664>: call    *(%eax,%edx,4)
0xc0625617 <vtryrecycle+667>: add     $0x4,%esp
0xc062561a <vtryrecycle+670>: mov     %esi,%esi
0xc062561c <vtryrecycle+672>: pushl   0xffffffffe0(%ebp)
0xc062561f <vtryrecycle+675>: call    0xc0631e48 <vn_finished_write>
0xc0625624 <vtryrecycle+680>: mov     $0x0,%edx
```


We should recognize the last three lines: they're clearly a call to the function `vn_finished_write` with a single parameter copied from something on the stack. That closely matches line 682, so indeed it's correct. But this code is executed every call, so it's fairly clear that we didn't get a breakpoint on it. Why not?

It might be worth going back and finding out, but I'm not going to do so here. It's an indication of the general fakeness of kernel debugging. We can be pretty sure that something went wrong, and while it's annoying, it gave us a chance to investigate more of the code. We can be pretty sure now that the immediate cause of the panic is that the vnode in question had its type field set to `VBAD`, but `vp->v_data` was not zero. This code assumes that it is and doesn't try to clean it.

Are we done? Not by a long way. Not only have we not fixed the bug, we still don't even understand how this is happening. Let's go back and look again at the code which frees the vnode:

Freeing the vnode

We saw on page 88 that vnodes get freed by the function `vfree`. If we're correct, we should see something returning a vnode of type `VBAD`. Let's go looking for it:

```
(gdb) l vfree
3097  * Mark a vnode as free, putting it up for recycling.
3098  */
3099  void
3100  vfree(vp)
3101      struct vnode *vp;
3102  {
3103
3104      ASSERT_VI_LOCKED(vp, "vfree");
3105      mtx_lock(&vnode_free_list_mtx);
3106      KASSERT((vp->v_iflag & VI_FREE) == 0, ("vnode already free"));
(gdb) b 3104
Breakpoint 1 at 0xc06293af: file /usr/src/sys/kern/vfs_subr.c, line 3104.
(gdb) c
Continuing.

Breakpoint 1, vfree (vp=0xc4ea3514) at /usr/src/sys/kern/vfs_subr.c:3105
3105      mtx_lock(&vnode_free_list_mtx);
(gdb) p vp->v_type
$1 = VCHR
```

That's what we'd normally expect; the vast majority of vnodes will have a different type field. Let's refine our search by setting a condition on the breakpoint:

```
(gdb) cond 1 vp->v_type == VBAD
(gdb) c
Continuing.
(on a different terminal)
# ls -l /src/FreeBSD/BFS/src/contrib/binutils/binutils
(back to the debug terminal)
Breakpoint 1, vfree (vp=0xc4e9f30c) at /usr/src/sys/kern/vfs_subr.c:3105
3105      mtx_lock(&vnode_free_list_mtx);
(gdb) p vp->v_type
$2 = VBAD
(gdb) bt
#0  vfree (vp=0xc4e9f30c) at /usr/src/sys/kern/vfs_subr.c:3105
#1  0xc0627d66 in vput (vp=0xc4e9f30c) at /usr/src/sys/kern/vfs_subr.c:2055
#2  0xc062cc74 in stat (td=0xc47bca80, uap=0xe1186d14) at /usr/src/sys/kern/vfs_syscall.c:2032
#3  0xc074ce57 in syscall (frame=
    {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf edi = 0x8054e00, tf esi = 0x8054e4
```

```
8, tf_ebp = 0xbfbfdcb8, tf_isp = 0xe1186d74, tf_ebx = 0x2817f78c, tf_edx = 0x7, tf_ecx
= 0x0, tf_eax = 0xbc, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0x2810e2e7, tf_cs = 0x1
f, tf_eflags = 0x296, tf_esp = 0xbfbfdclc, tf_ss = 0x2f}) at /usr/src/sys/i386/i386/tr
ap.c:1004
```

So in this case it's a `stat` system call. Going back down the stack, we see:

```
2014 /*
2015  * Release an already locked vnode. This give the same effects as
2016  * unlock+vrel(), but takes less time and avoids releasing and
2017  * re-acquiring the lock (as vrel() acquires the lock internally.)
2018  */
2019 void
2020 vput(vp)
2021     struct vnode *vp;
2022 {
2023     struct thread *td = curthread; /* XXX */
2024
2025     GIANT_REQUIRED;
2026
2027     KASSERT(vp != NULL, ("vput: null vp"));
2028     VI_LOCK(vp);
2029     /* Skip this v_writecount check if we're going to panic below. */
2030     KASSERT(vp->v_writecount < vp->v_usecount || vp->v_usecount < 1,
2031         ("vput: missed vn_close"));
2032
2033     if (vp->v_usecount > 1 || ((vp->v_iflag & VI_DOINGINACT) &&
2034         vp->v_usecount == 1)) {
2035         v_incr_usecount(vp, -1);
2036         VOP_UNLOCK(vp, LK_INTERLOCK, td);
2037         return;
2038     }
2039
2040     if (vp->v_usecount == 1) {
2041         v_incr_usecount(vp, -1);
2042         /*
2043          * We must call VOP_INACTIVE with the node locked, so
2044          * we just need to release the vnode mutex. Mark as
2045          * as VI_DOINGINACT to avoid recursion.
2046          */
2047         vp->v_iflag |= VI_DOINGINACT;
2048         VI_UNLOCK(vp);
2049         VOP_INACTIVE(vp, td);
2050         VI_LOCK(vp);
2051         KASSERT(vp->v_iflag & VI_DOINGINACT,
2052             ("vput: lost VI_DOINGINACT"));
2053         vp->v_iflag &= ~VI_DOINGINACT;
2054         if (VSHOULDFREE(vp))
2055             vfree(vp);
2056         else
2057             vlruvp(vp);
2058         VI_UNLOCK(vp);
2059     }
2060     (etc)
```

There's nothing there that looks very much like setting the vnode type. It's reasonable to assume that it was already set when the function was called. Let's look further back, to `stat`:

```
2009 int
2010 stat(td, uap)
2011     struct thread *td;
2012     register struct stat_args /* {
2013         char *path;
2014         struct stat *ub;
2015     } */ *uap;
2016 {
2017     struct stat sb;
2018     int error;
```

```

2019         struct nameidata nd;
2020
2025         NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | NOOBJ, UIO_USERSPACE,
2026             uap->path, td);
2028         if ((error = namei(&nd)) != 0)
2029             return (error);
2030         error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);
2031         NDFREE(&nd, NDF_ONLY_PNBUF);
2032         vput(nd.ni_vp);
2033         if (error)
2034             return (error);
2035         error = copyout(&sb, uap->ub, sizeof (sb));
2036         return (error);
2037     }

```

There are a couple of possibilities here. It could be `namei` that sets the type to `VBAD`, or it could be `vn_stat`. In all probability it's `vn_stat`. Looking there, we see:

```

628     /*
629     * Stat a vnode; implementation for the stat syscall
630     */
631     int
632     vn_stat(vp, sb, active_cred, file_cred, td)
633         struct vnode *vp;
634         register struct stat *sb;
635         struct ucred *active_cred;
636         struct ucred *file_cred;
637         struct thread *td;
638     {
639         struct vattr vattr;
640         register struct vattr *vap;
641         int error;
642         u_short mode;
643
644         #ifdef MAC
645         error = mac_check_vnode_stat(active_cred, file_cred, vp);
646         if (error)
647             return (error);
648         #endif
649
650         vap = &vattr;
651         error = VOP_GETATTR(vp, vap, active_cred, td);
652         if (error)
653             return (error);
654
655         vp->v_cachedfs = vap->va_fsid;
656         vp->v_cachedid = vap->va_fileid;
657
658         /*
659         * Zero the spare stat fields
660         */
661         bzero(sb, sizeof *sb);
662
663         /*
664         * Copy from vattr table
665         */
666         if (vap->va_fsid != VNOVAL)
667             sb->st_dev = vap->va_fsid;
668         else
669             sb->st_dev = vp->v_mount->mnt_stat.f_fsid.val[0];
670         sb->st_ino = vap->va_fileid;
671         mode = vap->va_mode;
672         switch (vap->va_type) {
673         case VREG:
674             mode |= S_IFREG;
675             break;
676         case VDIR:
677             mode |= S_IFDIR;
678             break;

```

```

679         case VBLK:
680             mode |= S_IFBLK;
681             break;
682         case VCHR:
683             mode |= S_IFCHR;
684             break;
685         case VLNK:
686             mode |= S_IFLNK;
687             /* This is a cosmetic change, symlinks do not have a mode. */
688             if (vp->v_mount->mnt_flag & MNT_NOSYMFOLLOW)
689                 sb->st_mode &= ~ACCESSPERMS; /* 0000 */
690             else
691                 sb->st_mode |= ACCESSPERMS; /* 0777 */
692             break;
693         case VSOCK:
694             mode |= S_IFSOCK;
695             break;
696         case VFIFO:
697             mode |= S_IFIFO;
698             break;
699         default:
700             return (EBADF);
701     };

```

In other words, `vn_stat` returns `EBADF` if it doesn't recognize the type of the inode. This is the same field that we looked at on page 98, and as we saw there, it's invalid. So that explains the `EBADF`.

Are we done? Not yet. We now understand how the `EBADF` is occurring, but what about the `VBAD`? Let's take a look:

```

(gdb) b 700
Breakpoint 2 at 0xc0631810: file /usr/src/sys/kern/vfs_vnops.c, line 700.
(gdb) c
Continuing.

Breakpoint 2, vn_stat (vp=0xc5692a28, sb=0xdff60c80, active_cred=0x0, file_cred=0x0, t
d=0xc456da80)
    at /usr/src/sys/kern/vfs_vnops.c:700
700             return (EBADF);
(gdb) p vp->v_type
$3 = VNON
(gdb) bt
#0  vn_stat (vp=0xc5692a28, sb=0xdff60c80, active_cred=0x0, file_cred=0x0, td=0xc456da
80)
    at /usr/src/sys/kern/vfs_vnops.c:700
#1  0xc062cc59 in stat (td=0xc456da80, uap=0xdff60d14) at /usr/src/sys/kern/vfs_syscal
ls.c:2030
#2  0xc074ce57 in syscall (frame=
    {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf edi = 0x8053b00, tf esi = 0x8053b4
8, tf_ebp = 0xbfbfdcb8, tf_esp = 0xdff60d74, tf_ebx = 0x2817f78c, tf_edx = 0x4, tf_ecx
= 0x0, tf_eax = 0xbc, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0x2810e2e7, tf_cs = 0x1
f, tf_eflags = 0x296, tf_esp = 0xbfbfdclc, tf_ss = 0x2f}) at /usr/src/sys/i386/i386/tr
ap.c:1004
#3  0xc073b81f in Xint0x80_syscall () at {standard input}:136
OK, this is the right one
(gdb) fini
Run till exit from #0  vn_stat (vp=0xc5692a28, sb=0xdff60c80, active_cred=0x0, file_cr
ed=0x0, td=0xc456da80)
    at /usr/src/sys/kern/vfs_vnops.c:700
0xc062cc59 in stat (td=0xc456da80, uap=0xdff60d14) at /usr/src/sys/kern/vfs_syscalls.c
:2030
2030             error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);
Value returned is $4 = 0x9
(gdb) p error
$5 = 0x9
(gdb) p nd.ni_vp.v_type
$6 = VNON

```

So, although the vnode has been established to be in error, it's still of indeterminate type (VNON). How does it get VBAD?

```
(gdb) s
2032          vput(nd.ni_vp);
(gdb) p nd.ni_vp.v_type
$7 = VNON
(gdb)
vput (vp=0xc5692a28) at machine/pcpu.h:156
156      {
(gdb) p vp->v_type
$8 = VNON
(gdb) n
(several more steps, showing nothing remarkable)
961      }
(gdb) p vp->v_type
$9 = VNON
(gdb) disp vp->v_type                                display it on each stop
1: vp->v_type = VNON
(gdb) n
2047          vp->v_iflag |= VI_DOINGINACT;
1: vp->v_type = VNON
(gdb)
2048          VI_UNLOCK(vp);
1: vp->v_type = VNON
(gdb)
945      {
1: vp->v_type = VNON
(gdb)
948          a.a_desc = VDESC(vop_inactive);
1: vp->v_type = VNON
(gdb)
949          a.a_vp = vp;
1: vp->v_type = VNON
(gdb)
950          a.a_td = td;
1: vp->v_type = VNON
(gdb) n
953          rc = VCALL(vp, VOFFSET(vop_inactive), &a);
1: vp->v_type = VNON
(gdb)
2050          VI_LOCK(vp);
1: vp->v_type = VBAD
(gdb) bt
#0  vput (vp=0xc5692a28) at /usr/src/sys/kern/vfs_subr.c:2050
#1  0xc062cc74 in stat (td=0xc456da80, uap=0xdff60d14) at /usr/src/sys/kern/vfs_syscall.c:2032
#2  0xc074ce57 in syscall (frame=
    {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf edi = 0x8053b00, tf esi = 0x8053b4
    8, tf_ebp = 0xbfbfdcb8, tf_esp = 0xdff60d74, tf_ebx = 0x2817f78c, tf_edx = 0x4, tf_ecx
    = 0x0, tf_eax = 0xbc, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0x2810e2e7, tf_cs = 0x1
    f, tf_eflags = 0x296, tf_esp = 0xbfbfdclc, tf_ss = 0x2f}) at /usr/src/sys/i386/i386/tr
    ap.c:1004
```

Where are we now? When we stopped, we were at line 2050 of *vfs_subr.c*, but the code before doesn't match:

```
2048          VI_UNLOCK(vp);
2049          VOP_INACTIVE(vp, td);
2050          VI_LOCK(vp);
```

Once again it's in a generated header file; clearly it's the function called by *VOP_INACTIVE* that's setting VBAD. Let's take a look at that in more detail. The breakpoint in *vn_stat* is handy because it won't trigger unless we have a bad vnode. We trigger it again and proceed as fast as we can to the correct place:

```

Breakpoint 2, vn_stat (vp=0xc4e25514, sb=0xe1174c80, active_cred=0x0, file_cred=0x0, td=0xc47bc2a0)
    at /usr/src/sys/kern/vfs_vnops.c:700
700             return (EBADF);
(gdb) fini
Run till exit from #0 vn_stat (vp=0xc4e25514, sb=0xe1174c80, active_cred=0x0, file_cred=0x0, td=0xc47bc2a0)
    at /usr/src/sys/kern/vfs_vnops.c:700
0xc062cc59 in stat (td=0xc47bc2a0, uap=0xe1174d14) at /usr/src/sys/kern/vfs_syscalls.c:2030
2030             error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);
Value returned is $10 = 0x9
(gdb) b vput
Breakpoint 3 at 0xc0627bf7: file machine/pcpu.h, line 156.
(gdb) c
Continuing.

Breakpoint 3, vput (vp=0xc4e25514) at machine/pcpu.h:156
156     {
1: vp->v_type = VNON
(gdb) b 2049
No line 2049 in file "machine/pcpu.h".
(gdb) l
151     #define PCPU_PTR(member)      __PCPU_PTR(pc_ ## member)
152     #define PCPU_SET(member, val) __PCPU_SET(pc_ ## member, val)
153
154     static __inline struct thread *
155     __curthread(void)
156     {
157         struct thread *td;
158
159         __asm __volatile("movl %%fs:0,%0" : "=r" (td));
160         return (td);

```

The problem here is that we're in yet another inline function, so the line numbers are wrong. Moving to the next instruction should solve the problem:

```

(gdb) n
2025             GIANT_REQUIRED;
1: vp->v_type = VNON
(gdb) b 2049
Breakpoint 4 at 0xc0627d02: file /usr/src/sys/kern/vfs_subr.c, line 2049.
(gdb) c
Continuing.

Breakpoint 4, vput (vp=0xc4e25514) at /usr/src/sys/kern/vfs_subr.c:2050
2050             VI_LOCK(vp);
1: vp->v_type = VBAD

```

Here, although we set a breakpoint on the correct line, we didn't hit it, because we were on yet *another* inline function. We'll have to try again:

```

(gdb) disa 2 3                                     in case something else goes through here
(gdb) c
Continuing.

Breakpoint 3, vput (vp=0xc4e25514) at machine/pcpu.h:156
156     {
1: vp->v_type = VNON
(gdb) b 2048                                         set breakpoint on previous line
Breakpoint 5 at 0xc0627ccd: file /usr/src/sys/kern/vfs_subr.c, line 2048.
(gdb) c
Continuing.

Breakpoint 5, vput (vp=0xc4e25514) at /usr/src/sys/kern/vfs_subr.c:2048
2048             VI_UNLOCK(vp);
1: vp->v_type = VNON

```

```

(gdb) n
945 {
1: vp->v_type = VNON
(gdb) s step into functions
948     a.a_desc = VDESC(vop_inactive);
1: vp->v_type = VNON
(gdb)
949     a.a_vp = vp;
1: vp->v_type = VNON
(gdb)
950     a.a_td = td;
1: vp->v_type = VNON
(gdb)
953     rc = VCALL(vp, VOFFSET(vop_inactive), &a);
1: vp->v_type = VNON
(gdb)
ufs_vnoperate (ap=0xe1174bf0) at /usr/src/sys/ufs/ufs/ufs_vnops.c:2819
2819     return (VCALL(ufs_vnodeop_p, ap->a_desc->vdesc_offset, ap));

```

We no longer have a pointer `vp` in the current frame, so the display stops. Continuing,

```

ufs_inactive (ap=0xe1174bf0) at /usr/src/sys/ufs/ufs/ufs_inode.c:71
71     struct vnode *vp = ap->a_vp;
(gdb)
72     struct inode *ip = VTOI(vp);
(gdb) i dis
Auto-display expressions now in effect:
Num Enb Expression
1: y vp->v_type (cannot be evaluated in the current context)

```

This doesn't make any sense: we've just defined (and correctly initialized) a new `vp` pointer. We can display it:

```

(gdb) p vp
$11 = (struct vnode *) 0xc4e25514
(gdb) p vp->v_type
$12 = VNON
(gdb) disp vp->v_type
2: vp->v_type = VNON
(gdb) i dis
Auto-display expressions now in effect:
Num Enb Expression
2: y vp->v_type
1: y vp->v_type (cannot be evaluated in the current context)

```

Not for the first time, this is a bug in *gdb*. We move on:

```

(gdb) n
80     VI_UNLOCK(vp);
(a number of further steps)
(gdb)
131     vrecycle(vp, NULL, td);
2: vp->v_type = VNON
(gdb)
133 }
2: vp->v_type = VBAD

```

So there's a good chance that `vrecycle` is responsible for setting `VBAD`. We could go back and step through it again, but it's likely that we can also just look at it directly:

```

2474 /*
2475  * Recycle an unused vnode to the front of the free list.
2476  * Release the passed interlock if the vnode will be recycled.
2477  */

```

```

2478     int
2479     vrecycle(vp, inter_lkp, td)
2480         struct vnode *vp;
2481         struct mtx *inter_lkp;
2482         struct thread *td;
2483     {
2484
2485         VI_LOCK(vp);
2486         if (vp->v_usecount == 0) {
2487             if (inter_lkp) {
2488                 mtx_unlock(inter_lkp);
2489             }
2490             vgonel(vp, td);
2491             return (1);
2492         }
2493         VI_UNLOCK(vp);
2494         return (0);
2495     }

```

There's a function we recognize! It's on page 98. From there, we know that it ultimately calls `ufs_reclaim`, which sets `VBAD`. But that doesn't help us much: we also know that `ufs_reclaim` resets the `vp->v_data`. We didn't check that; what do we have here? We disable all breakpoints except the one in `vn_stat`, then try again and this time set a breakpoint at `vgonel`, then single step from there:

```

Breakpoint 2, vn_stat (vp=0xc4e25514, sb=0xellb5c80, active_cred=0x0, file_cred=0x0,
td=0xc4978000)
    at /usr/src/sys/kern/vfs_vnops.c:700
700             return (EBADF);
(gdb) b vgonel
Breakpoint 6 at 0xc0628957: file /usr/src/sys/kern/vfs_subr.c, line 2567.
(gdb) c
Continuing.

Breakpoint 6, vgonel (vp=0xc4e25514, td=0xc4978000) at /usr/src/sys/kern/vfs_subr.c:2
567
2567         if (vp->v_iflag & VI_XLOCK) {
(gdb) p vp->v_type
$13 = VNON
(gdb) p vp->v_data
$14 = (void *) 0xc586dec4
(gdb) disp vp->v_data
3: vp->v_data = (void *) 0xc586dec4
(gdb) disp vp->v_type
4: vp->v_type = VNON
(gdb) n
2572             vx_lock(vp);
4: vp->v_type = VNON
3: vp->v_data = (void *) 0xc586dec4
(gdb)
2577             vclean(vp, DOCLOSE, td);
4: vp->v_type = VNON
3: vp->v_data = (void *) 0xc586dec4
(gdb)
2578             VI_UNLOCK(vp);
4: vp->v_type = VNON
3: vp->v_data = (void *) 0x0
...
2617             vx_unlock(vp);
4: vp->v_type = VBAD
3: vp->v_data = (void *) 0x0
(gdb)
2618             VI_UNLOCK(vp);
4: vp->v_type = VBAD
3: vp->v_data = (void *) 0x0
(gdb)

```


That's an interesting thing: it looks like a function hiding behind `DOCLOSE` is resetting `vp->v_data`, and later an unlock function is setting the type to `VBAD`, both a very different scenario from that which we saw at the beginning of this dump. It's worth investigating the reasons for that, but there's not enough time. At any rate, there's a good chance that in at least one case the function behind `DOCLOSE` doesn't like what it sees, and doesn't reset `vp->v_data`. We've know the inode number of the inode that has caused all the panics so far, so let's wait for that to go by:

```
(gdb) cond 6 ((struct inode *)vp->v_data)->i_number == 3742712
(gdb) c
Continuing.
```

And that's it. We don't hit the breakpoint. We remove the condition and instead display the inode number:

```
(gdb) cond 6
Breakpoint 6 now unconditional.
(gdb) disp/d ((struct inode *)vp->v_data)->i_number
No symbol "vp" in current context.
```

That's another problem with *gdb*: you can only display objects which are currently accessible. We can handle that:

```
(gdb) c
Continuing.

Breakpoint 6, vgonel (vp=0xc5071a28, td=0xc497b000) at /usr/src/sys/kern/vfs_subr.c:2567
2567         if (vp->v_iflag & VI_XLOCK) {
(gdb) disp/d ((struct inode *)vp->v_data)->i_number
5: /d ((struct inode *) vp->v_data)->i_number = 3742674
(gdb) c
Continuing.
...
Breakpoint 6, vgonel (vp=0xc5071a28, td=0xc497b000) at /usr/src/sys/kern/vfs_subr.c:2567
2567         if (vp->v_iflag & VI_XLOCK) {
3: vp->v_data = (void *) 0xc56d59d8
(gdb)
Continuing.

Breakpoint 6, vgonel (vp=0xc5071a28, td=0xc497b000) at /usr/src/sys/kern/vfs_subr.c:2567
2567         if (vp->v_iflag & VI_XLOCK) {
5: /d ((struct inode *) vp->v_data)->i_number = 3742713
(gdb)
```

In other words, we don't come here for this specific inode. There are two possible reasons:

- The code doesn't do the same thing in that case.
- Something has happened on disk to make that particular inode go away.

So we set the conditional breakpoint in `vn_stat` instead:

```
(gdb) cond 2 ((struct inode *)vp->v_data)->i_number == 3742712
(gdb) c
Continuing.

Breakpoint 2, vn_stat (vp=0xc4e9f30c, sb=0xe11d0c80, active_cred=0x391bf8, file_cred=0
x0, td=0xc4978bd0)
at /usr/src/sys/kern/vfs_vnops.c:700
700         return (EBADF);
```

```

6: /d ((struct inode *) vp->v_data)->i_number = 3742712
(gdb) fini
Run till exit from #0  vn_stat (vp=0xc4e9f30c, sb=0xe11d0c80, active_cred=0x391bf8, fi
le_cred=0x0, td=0xc4978bd0)
    at /usr/src/sys/kern/vfs_vnops.c:700
0xc062cc59 in stat (td=0xc4978bd0, uap=0xe11d0d14) at /usr/src/sys/kern/vfs_syscalls.c
:2030
2030         error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);
Value returned is $20 = 0x9
(gdb) disp nd.ni_vp.v_type
7: nd.ni_vp.v_type = VBAD

```

OK, now things are getting clearer: in this one case, the vnode is already set to VBAD on return from `vn_stat`. Our previous assumptions were based on a different vnode. It, too, was invalid, but in a different way. We'll have to go back and investigate again, looking for this specific inode number.

```

Breakpoint 2, vn_stat (vp=0xc4e9f30c, sb=0xe118fc80, active_cred=0x391bf8, file_cred=0
x0, td=0xc47bce70)
    at /usr/src/sys/kern/vfs_vnops.c:700
700         return (EBADF);
6: /d ((struct inode *) vp->v_data)->i_number = 3742712
(gdb) p vp->v_type
$21 = VBAD

```

So our vnode was bad even before we returned the EBADF. Let's look at `vn_stat` again (page 108). About the first place it could get set would be at line 651:

```

651         error = VOP_GETATTR(vp, vap, active_cred, td);

(gdb) b 651 if ((struct inode *)vp->v_data)->i_number == 3742712
Breakpoint 7 at 0xc0631771: file /usr/src/sys/kern/vfs_vnops.c, line 651.
(gdb) c
Continuing.

Breakpoint 7, vn_stat (vp=0xc4e9f30c, sb=0xe11c1c80, active_cred=0x0, file_cred=0x0, t
d=0xc4978540)
    at /usr/src/sys/kern/vfs_vnops.c:652
652         if (error)
6: /d ((struct inode *) vp->v_data)->i_number = 3742712
(gdb) p vp->v_type
$23 = VBAD

```

Incorrect assumption. It must have happened much earlier, possibly before entering the function. We can check that:

```

(gdb) b vn_stat if ((struct inode *)vp->v_data)->i_number == 3742712
Breakpoint 8 at 0xc063172e: file /usr/src/sys/kern/vfs_vnops.c, line 650.
(gdb) c
Continuing.

Breakpoint 8, vn_stat (vp=0xc4e9f30c, sb=0xe11d0c80, active_cred=0xe11d0c80, file_cred
=0x0, td=0xc4978bd0)
    at /usr/src/sys/kern/vfs_vnops.c:650
650         vap = &vattrib;
6: /d ((struct inode *) vp->v_data)->i_number = 3742712
(gdb) p vp->v_type
$24 = VBAD

```

Yes, we entered like that. Let's look back at the calling function: (`stat`, page 108). There's not very much to see there:

```

2025         NDINIT(&nd, LOOKUP, FOLLOW | LOCKLEAF | NOOBJ, UIO_USERSPACE,
2026             uap->path, td);
2028         if ((error = namei(&nd)) != 0)
2029             return (error);
2030         error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);

```

It must have happened in `namei` after all. We can check that, but we won't know the inode number until it's too late. Fortunately, `stat` and `namei` keep track of file names, so we can use them:

```

Breakpoint 8, vn_stat (vp=0xc4e9f30c, sb=0xe11b8c80, active_cred=0xe11b8c80, file_cred=0x0, td=0xc4978150)
  at /usr/src/sys/kern/vfs_vnops.c:650
650         vap = &vattr;
6: /d ((struct inode *) vp->v_data)->i_number = 3742712
(gdb) bt
#0  vn_stat (vp=0xc4e9f30c, sb=0xe11b8c80, active_cred=0xe11b8c80, file_cred=0x0, td=0xc4978150)
  at /usr/src/sys/kern/vfs_vnops.c:650
#1  0xc062cc59 in stat (td=0xc4978150, uap=0xe11b8d14) at /usr/src/sys/kern/vfs_syscalls.c:2030
(gdb) f 1
#1  0xc062cc59 in stat (td=0xc4978150, uap=0xe11b8d14) at /usr/src/sys/kern/vfs_syscalls.c:2030
2030         error = vn_stat(nd.ni_vp, &sb, td->td_ucred, NOCRED, td);
(gdb) p nd
$26 = {
  ni_dirp = 0x8050000 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
  ni_segflg = UIO_USERSPACE,
  ni_startdir = 0x0,
  ni_rootdir = 0xc46d1e38,
  ni_topdir = 0x0,
  ni_vp = 0xc4e9f30c,
  ni_dvp = 0xc54ea618,
  ni_pathlen = 0x1,
  ni_next = 0xc46d8c35 "",
  ni_loopcnt = 0x0,
  ni_cnd = {
    cn_nameiop = 0x0,
    cn_flags = 0x20c0c4,
    cn_thread = 0xc4978150,
    cn_cred = 0xc61dc700,
    cn_pnbuf = 0xc46d8c00 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
    cn_nameptr = 0xc46d8c2f "ieee.c",
    cn_namelen = 0x6,
    cn_consume = 0x0
  }
}

```

So now, for the first time, we know the name of the file which is causing us so much grief. We could try setting a conditional breakpoint based on the name, but *gdb* is not very good at handling strings. Instead, since we know the name, we can list it explicitly:

```
# ls -l /src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c
```

```

(gdb) b namei
Breakpoint 10 at 0xc0621c05: file /usr/src/sys/kern/vfs_lookup.c, line 104.
(gdb) c
Continuing.

Breakpoint 10, namei (ndp=0xdff5dbe4) at /usr/src/sys/kern/vfs_lookup.c:104
104         struct componentname *cnp = &ndp->ni_cnd;

```

```
(gdb) p *ndp
$27 = {
  ni_dirp = 0x8145a60 "/bin/ls",
  ni_segflg = UIO_USERSPACE,
  ni_startdir = 0x0,
  ni_rootdir = 0xc5466938,
  ni_topdir = 0xc4cd6168,
  ni_vp = 0x0,
  ni_dvp = 0x0,
  ni_pathlen = 0xc05de759,
  ni_next = 0xe1186ca0 "D\b",
  ni_loopcnt = 0xbfbfeb6c,
  ni_cnd = {
    cn_nameiop = 0x0,
    cn_flags = 0x844,
    cn_thread = 0xc47bca80,
    cn_cred = 0xe1186ccc,
    cn_pnbuf = 0xc47ba224 "<È\200À\227/{À\227/{À",
    cn_nameptr = 0x2cc <Address 0x2cc out of bounds>,
    cn_namelen = 0xc07d0c2f,
    cn_consume = 0xe1186cdc
  }
}
```

This is one of the problems with a function like `namei`: it gets called many times every time you start a program. We can't make it conditional on a string, but we can check individual characters. In this case, the second character of the pathname is `s`, so we can check for that:

```
(gdb) cond 10 ndp->ni_dirp[1] == 's'
```

`namei` allocates a `vnode`, so on entry the value is indeterminate. Before we start looking at the contents of the `vnode`, we need to be sure that it's valid. In the example above, it's set to `NULL`, but it doesn't have to be. But where is the `vnode` allocated? The code suggests that some function might have done it. An easy way to find out might be to single step through the main loop until the value changes:

```
Breakpoint 10, namei (ndp=0xe11d3c30) at /usr/src/sys/kern/vfs_lookup.c:104
104      struct componentname *cnp = &ndp->ni_cnd;
(gdb) disp ndp->ni_vp
11: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb) n
105      struct thread *td = cnp->cn_thread;
11: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb)
106      struct proc *p = td->td_proc;
11: ndp->ni_vp = (struct vnode *) 0x1d2
...
178      ndp->ni_startdir = dp;
11: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb)
179      error = lookup(ndp);
11: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb)
180      if (error) {
11: ndp->ni_vp = (struct vnode *) 0xc4e9f30c
(gdb) p ndp->ni_vp->v_data
$31 = (void *) 0xc4c7e230
(gdb) p ndp->ni_vp->v_type
$32 = VBAD
```

So not only the allocation, but also the setting of the type is done by `lookup`. In this case, we already have the scenario we've been looking at: the type is `VBAD`, but the `v_data` field is still set. That's the next thing to look at. `lookup` is in the file `sys/kern/vfs_lookup.c`. We quickly

establish that there's no reference to VBAD there, so it must be yet another called function. Again we single-step:

```
Breakpoint 11, lookup (ndp=0xe1186c30) at /usr/src/sys/kern/vfs_lookup.c:328
328      int dpunlocked = 0; /* dp has already been unlocked */
(gdb) disp ndp->ni_vp
12: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb) n
329      struct componentname *cnp = &ndp->ni_cnd;
12: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb)
330      struct thread *td = cnp->cn_thread;
12: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb)
...
482      ndp->ni_vp = NULL;
12: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb)
483      cnp->cn_flags &= ~PDIRUNLOCK;
12: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
42      {
12: ndp->ni_vp = (struct vnode *) 0x0
...
52      rc = VCALL(dvp, VOFFSET(vop_lookup), &a);
12: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
42      {
12: ndp->ni_vp = (struct vnode *) 0xc4760a28
(gdb) p ndp->ni_vp->v_data
$33 = (void *) 0xc470e834
(gdb) p ndp->ni_vp->v_type
$34 = VDIR
(gdb) p ndp
$35 = (struct nameidata *) 0xe1186c30
(gdb) p *ndp
$36 = {
    ni_dirp = 0x80511a8 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
    ni_segflg = UIO_USERSPACE,
    ni_startdir = 0x0,
    ni_rootdir = 0xc46d1e38,
    ni_topdir = 0x0,
    ni_vp = 0xc4760a28,
    ni_dvp = 0xc46d1e38,
    ni_pathlen = 0x32,
    ni_next = 0xc497e004 "/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
    ni_loopcnt = 0x0,
    ni_cnd = {
        cn_nameiop = 0x0,
        cn_flags = 0x2040c4,
        cn_thread = 0xc47bca80,
        cn_cred = 0xc61dc700,
        cn_pnbuf = 0xc497e000 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
        cn_nameptr = 0xc497e001 "src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
        cn_namelen = 0x3,
        cn_consume = 0x0
    }
}
(gdb) disp ndp->ni_vp->v_type
13: ndp->ni_vp->v_type = VDIR
(gdb) p *((struct inode *)ndp->ni_vp.v_data)
$38 = {
    i_hash = {
        le_next = 0x0,
        le_prev = 0xc4623078
    },
    i_nextsnap = {
        tqe_next = 0x0,
        tqe_prev = 0x0
    },
}
```

```

i_vnode = 0xc5937b2c,
i_ump = 0xc46cf000,
i_flag = 0x20,
i_dev = 0xc46aba00,
i_number = 0x2,
i_effnlink = 0x32,
i_fs = 0xc46ef800,
i_dquot = {0x0, 0x0},
i_modrev = 0xb161df9fa8c,
i_lockf = 0x0,
i_count = 0x0,
i_endoff = 0x0,
i_diroff = 0x200,
i_offset = 0x338,
i_ino = 0x86ee0f,
i_reclen = 0xc8,
i_un = {
    dirhash = 0x0,
    snapblklist = 0x0
},
i_ea_area = 0x0,
i_ea_len = 0x0,
i_ea_error = 0x0,
i_mode = 0x41ed,
i_nlink = 0x32,
i_size = 0x400,
i_flags = 0x0,
i_gen = 0xe6a9665,
i_uid = 0x0,
i_gid = 0x0,
dinode_u = {
    din1 = 0xc55d4d00,
    din2 = 0xc55d4d00
}
}
(gdb) p/o ((struct inode *)ndp->ni_vp.v_data)->i_mode
$39 = 040755

```

So far, this looks like a valid directory. That's not surprising: `lookup` iterates its way through the path name, a directory at a time. The function is 367 lines long, so it's not reproduced here. It's in `sys/kern/vfs_lookup.c` for reference. Investigation of the code suggests that this is the most likely place for the vnode to be modified. This listing omits some `#ifdef` code:

```

470          /*
471          * We now have a segment name to search for, and a directory to search.
472          */
473      unionlookup:
481          ndp->ni_dvp = dp;
482          ndp->ni_vp = NULL;
483          cnp->cn_flags &= ~PDIRUNLOCK;
484          ASSERT_VOP_LOCKED(dp, "lookup");
485          if ((error = VOP_LOOKUP(dp, &ndp->ni_vp, cnp)) != 0) {
486              KASSERT(ndp->ni_vp == NULL, ("leaf should be empty"));

```

Clearly, here the important structure is not the vnode pointer (`ndp->ni_vp`), but the “intermediate” vnode pointer `dp`. Also, the pointer `cnp` points to a “component name”. We stop on the conditional breakpoint in `namei`, then set a breakpoint before the call to `VOP_LOOKUP` and take a look at what we see. We’ve already been burnt by trying to set breakpoints on these macros, so we set it a little bit in advance, on line 481:

```

(gdb) en 10
(gdb) c

```

```

Breakpoint 10, namei (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:104

```

```

104          struct componentname *cnp = &ndp->ni_cnd;
11: ndp->ni_vp = (struct vnode *) 0x1d2
10: ndp->ni_dirp = 0x80511a8 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c"

(gdb) en 11
(gdb) c
Continuing.

Breakpoint 11, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:328
328          int dpunlocked = 0; /* dp has already been unlocked */
15: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb) b 481
Breakpoint 20 at 0xc062233c: file /usr/src/sys/kern/vfs_lookup.c, line 481.
(gdb) c
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481          ndp->ni_dvp = dp;
15: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb) p cnp
$63 = (struct componentname *) 0xe1144c58
(gdb) p *cnp
$64 = {
  cn_nameiop = 0x0,
  cn_flags = 0x4084,
  cn_thread = 0xc46fbd20,
  cn_cred = 0xc56a9400,
  cn_pnbuf = 0xc46d7c00 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
  cn_nameptr = 0xc46d7c01 "src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
  cn_namelen = 0x3,
  cn_consume = 0x0
}
(gdb) c
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481          ndp->ni_dvp = dp;
15: ndp->ni_vp = (struct vnode *) 0xc58b4208
(gdb) p *cnp
$65 = {
  cn_nameiop = 0x0,
  cn_flags = 0x204084,
  cn_thread = 0xc46fbd20,
  cn_cred = 0xc56a9400,
  cn_pnbuf = 0xc46d7c00 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
  cn_nameptr = 0xc46d7c01 "src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
  cn_namelen = 0x3,
  cn_consume = 0x0
}
(gdb) c
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481          ndp->ni_dvp = dp;
15: ndp->ni_vp = (struct vnode *) 0x1d2
(gdb) p cnp
$63 = (struct componentname *) 0xe1144c58
(gdb) p *cnp
$64 = {
  cn_nameiop = 0x0,
  cn_flags = 0x4084,
  cn_thread = 0xc46fbd20,
  cn_cred = 0xc56a9400,
  cn_pnbuf = 0xc46d7c00 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
  cn_nameptr = 0xc46d7c01 "src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
  cn_namelen = 0x3,
  cn_consume = 0x0
}
(gdb) c
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481          ndp->ni_dvp = dp;
15: ndp->ni_vp = (struct vnode *) 0xc58b4208
(gdb) p *cnp
$65 = {
  cn_nameiop = 0x0,
  cn_flags = 0x204084,

```

```

cn_thread = 0xc46fbd20,
cn_cred = 0xc56a9400,
cn_pnbuf = 0xc46d7c00 "/src/FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
cn_nameptr = 0xc46d7c05 "FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c",
cn_namelen = 0x7,
cn_consume = 0x0
}
(gdb) p cnp->cn_nameptr
$66 = 0xc46d7c05 "FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c"
(gdb) disp cnp->cn_nameptr
18: cnp->cn_nameptr = 0xc46d7c05 "FreeBSD/BFS/src/contrib/binutils/binutils/ieee.c"

```

We're now in the name parsing loop. Every iteration brings us one step closer to the end:

```

(gdb) c
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481      ndp->ni_dvp = dp;
18: cnp->cn_nameptr = 0xc46d7c0d "BFS/src/contrib/binutils/binutils/ieee.c"
15: ndp->ni_vp = (struct vnode *) 0xc4ad2b2c
(gdb)
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481      ndp->ni_dvp = dp;
18: cnp->cn_nameptr = 0xc46d7c11 "src/contrib/binutils/binutils/ieee.c"
15: ndp->ni_vp = (struct vnode *) 0xc6565a28
(gdb)
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481      ndp->ni_dvp = dp;
18: cnp->cn_nameptr = 0xc46d7c15 "contrib/binutils/binutils/ieee.c"
15: ndp->ni_vp = (struct vnode *) 0xc5b51d34
(gdb)
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481      ndp->ni_dvp = dp;
18: cnp->cn_nameptr = 0xc46d7c1d "binutils/binutils/ieee.c"
15: ndp->ni_vp = (struct vnode *) 0xc4d5d924
(gdb)
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481      ndp->ni_dvp = dp;
18: cnp->cn_nameptr = 0xc46d7c26 "binutils/ieee.c"
15: ndp->ni_vp = (struct vnode *) 0xc5a05514
(gdb)
Continuing.

Breakpoint 20, lookup (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:481
481      ndp->ni_dvp = dp;
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0xc5a3f71c

```

Now we're at the final part of the path name, where we expect the sparks to fly. It's interesting to note that the vnode pointer changes every time; if we had watched the single vnode, we wouldn't have found anything in particular.

From here on we single step to find what function performs the lookup:

```

(gdb) s
482      ndp->ni_vp = NULL;
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0xc5a3f71c

```



```

(gdb)
483         cnp->cn_flags &= ~PDIRUNLOCK;
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
42     {
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
45         a.a_desc = VDESC(vop_lookup);
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0x0
(gdb) bt
#0  lookup (ndp=0xe1144c30) at vnode_if.h:45
#1  0xc0621df8 in namei (ndp=0xe1144c30) at /usr/src/sys/kern/vfs_lookup.c:179
#2  0xc062ccde in lstat (td=0xc46fbd20, uap=0xe1144d14) at /usr/src/sys/kern/vfs_syscalls.c:2063
#3  0xc074ce57 in syscall (frame=
    {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf_ei = 0x8051100, tf_ei = 0x805114
    8, tf_ebp = 0xbfbfdd08, tf_esp = 0xe1144d74, tf_ebx = 0x2817f78c, tf_edx = 0x804f000,
    tf_ecx = 0x0, tf_eax = 0xbe, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0x2810e2a7, tf_cs
    = 0x1f, tf_eflags = 0x296, tf_esp = 0xbfbfdc6c, tf_ss = 0x2f}) at /usr/src/sys/i386/i
    386/trap.c:1004
#4  0xc073b81f in Xint0x80_syscall () at {standard input}:136
#5  0x28108413 in ?? ()
#6  0x08049ad9 in ?? ()
#7  0x08049a9d in ?? ()
#8  0x0804921e in ?? ()
(gdb) s
46         a.a_dvp = dvp;
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
47         a.a_vpp = vpp;
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
48         a.a_cnp = cnp;
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
52         rc = VCALL(dvp, VOFFSET(vop_lookup), &a);
18: cnp->cn_nameptr = 0xc46d7c2f "ieee.c"
15: ndp->ni_vp = (struct vnode *) 0x0
(gdb)
ufs_vnoperate (ap=0xe1144bb4) at /usr/src/sys/ufs/ufs/ufs_vnops.c:2819
2819         return (VCALL(ufs_vnodeop_p, ap->a_desc->vdesc_offset, ap));
(gdb)
vfs_cache_lookup (ap=0xe1144bb4) at /usr/src/sys/kern/vfs_cache.c:636
636         struct vnode **vpp = ap->a_vpp;
(gdb)
637         struct componentname *cnp = ap->a_cnp;

```

So we end up in `vfs_cache_lookup`. It's worth looking at that function:

```

620     /*
621     * Perform canonical checks and cache lookup and pass on to filesystem
622     * through the vop_cachedlookup only if needed.
623     */
624
625     int
626     vfs_cache_lookup(ap)
627         struct vop_lookup_args /* {
628             struct vnode *a_dvp;
629             struct vnode **a_vpp;
630             struct componentname *a_cnp;
631         } */ *ap;
632     {
633         struct vnode *dvp, *vp;
634         int lockparent;

```

```

635         int error;
636         struct vnode **vpp = ap->a_vpp;
637         struct componentname *cnp = ap->a_cnp;
638         struct ucred *cred = cnp->cn_cred;
639         int flags = cnp->cn_flags;
640         struct thread *td = cnp->cn_thread;
641         u_long vpid; /* capability number of vnode */
642
643         *vpp = NULL;
644         dvp = ap->a_dvp;
645         lockparent = flags & LOCKPARENT;
646
647         if (dvp->v_type != VDIR)
648             return (ENOTDIR);
649
650         if ((flags & ISLASTCN) && (dvp->v_mount->mnt_flag & MNT_RDONLY) &&
651             (cnp->cn_nameiop == DELETE || cnp->cn_nameiop == RENAME))
652             return (EROFS);
653
654         error = VOP_ACCESS(dvp, VEXEC, cred, td);
655
656         if (error)
657             return (error);

```

The checks above are the normal tests that would have given us a different error number (permissions, ENOTDIR, EROFS), so we can probably discount them. The rest looks less obvious:

```

659         error = cache_lookup(dvp, vpp, cnp);
687         if (!error)
688             return (VOP_CACHEDLOOKUP(dvp, vpp, cnp));
690
691         if (error == ENOENT)
692             return (error);
693
694         vp = *vpp;
695         vpid = vp->v_id;
696         cnp->cn_flags &= ~PDIRUNLOCK;
697         if (dvp == vp) { /* lookup on "." */
698             VREF(vp);
699             error = 0;
700         } else if (flags & ISDOTDOT) {
701             VOP_UNLOCK(dvp, 0, td);
702             cnp->cn_flags |= PDIRUNLOCK;
703             error = vget(vp, LK_EXCLUSIVE, td);
704
705             if (!error && lockparent && (flags & ISLASTCN)) {
706                 if ((error = vn_lock(dvp, LK_EXCLUSIVE, td)) == 0)
707                     cnp->cn_flags &= ~PDIRUNLOCK;
708             }
709         }

```

We set a breakpoint on line 659 and single step from there:

```

Breakpoint 25, vfs_cache_lookup (ap=0x0) at /usr/src/sys/kern/vfs_cache.c:659
659         error = cache_lookup(dvp, vpp, cnp);
(gdb) p vpp
$86 = (struct vnode **) 0xe1144c44
(gdb) p *vpp
$87 = (struct vnode *) 0x0

```

`vpp` is a pointer to a vnode pointer; this shows us that the pointer itself is currently unallocated.

```

(gdb) n
687         if (!error)
20: dvp->v_type = VDIR
(gdb) p *vpp
$88 = (struct vnode *) 0xc56fb924

```

```
(gdb) p *vpp->v_type
Attempt to take contents of a non-pointer value.
(gdb) p (*vpp)->v_type
$89 = VBAD
```

So it's `cache_lookup` that somehow returns the VBAD. We single step through it and get to this section:

```
406          /* We found a "positive" match, return the vnode */
407          if (ncp->nc_vp) {
408              numposhits++;
409              nchstats.ncs_goodhits++;
410              *vpp = ncp->nc_vp;
411              CACHE_UNLOCK();
412              return (-1);
413          }
```

Single stepping through, we find:

```
(gdb)
410          *vpp = ncp->nc_vp;
21: *vpp = (struct vnode *) 0x0
(gdb)
411          CACHE_UNLOCK();
21: *vpp = (struct vnode *) 0xc56fb924
(gdb) p (*vpp)->v_type
$91 = VBAD
```

So whatever caused the problem, it's now in cache, and so we can't find the original cause. We have to reboot.

After rebooting, and not surprisingly, `cache_lookup` returns a cache miss. `vfs_cache_lookup` moves on to:

```
687          if (!error)
688              return (VOP_CACHEDLOOKUP(dvp, vpp, cnp));
```

Behind this we find a call to `ufs_lookup`.

Giving up

Round here, it's becoming clear that finding the exact place where the problem occurs is not going to be very productive. It will almost certainly not be something that we can change easily. We're left with a number of possibilities:

- Send in a problem report. Maybe somebody will look at it. Without being too cynical, though, it's unlikely that a problem report will achieve very much. You'd need to send in the data disk as well to make it easy to reproduce the problem.
- Remove INVARIANTS. As we've seen, that would "solve" (in other words, ignore) the problem. The problem here is that we may have a memory leak as a result. One option here might be to print a warning instead: certainly we've seen that a panic doesn't help very much.
- Consider what would happen if we changed the test in `vtryrecycle` to try cleaning the vnode if the `v_data` field is not reset.

We can implement code for the last two:

Reporting errors instead of panicking

It's relatively trivial to replace the panic with an informative printout:

```
--- vfs_subr.c  11 Apr 2004 21:09:22 -0000      1.490
+++ vfs_subr.c  5 Oct 2004 06:31:49 -0000
@@ -752,7 +752,9 @@
 #ifdef INVARIANTS
 {
     if (vp->v_data)
-        panic("cleaned vnode isn't");
+        printf("cleaned vnode isn't, "
+            "address %p, inode %p\n",
+            vp, vp->vp_data);
     if (vp->v_numoutput)
         panic("Clean vnode has pending I/O's");
     if (vp->v_writcount != 0)
```

After this, when we run our *fi nd* command, instead of a panic we get:

```
cleaned vnode isn't, address 0xc49a8514, inode 0xc4996c08
```

It would be tempting to add the inode number, but that's a bad idea. This code is in the virtual file system. There's a reason why the field `vp->v_data` is of indeterminate type. Though unlikely, it would be a layering violation to try to interpret it as a UFS inode. In all probability, though, it will still be there when you see the message (we're counting on this being a memory leak), so we can look at it later:

```
# gdb -k kernel.debug /dev/mem
...
(kgdb) p *(struct inode *)0xc4996c08
$1 = {
  i_hash = {
    le_next = 0x0,
    le_prev = 0xc462a050
  },
  ...
  i_dev = 0xc4695b00,
  i_number = 0x391bf8,
  i_effnlink = 0x2,
}
(kgdb) p/d ((struct inode *)0xc4996c08)->i_number
$3 = 3742712
```

So this message will enable us to find out the information we want *without* panicking the machine. We commit the change:

```
grog      2004-10-06 02:06:11 UTC

FreeBSD src repository

Modified files:
  sys/kern          vfs_subr.c
Log:
getnewvnode: Weaken the panic "cleaned vnode isn't" to a warning.
```

```
Discussion: this panic (or warning) only occurs when the kernel is
compiled with INVARIANTS. Otherwise the problem (which means that
the vp->v_data field isn't NULL, and represents a coding error and
possibly a memory leak) is silently ignored by setting it to NULL
later on.
```

Panicking here isn't very helpful: by this time, we can only find the symptoms. The panic occurs long after the reason for "not cleaning" has been forgotten; in the case in point, it was the result of severe file system corruption which left the `v_type` field set to `VBAD`. That issue will be addressed by a separate commit.

| Revision | Changes | Path |
|----------|---------|-------------------------|
| 1.529 | +3 -1 | src/sys/kern/vfs_subr.c |

Cleaning if `v_data` is set

The other possibility is in `vtryrecycle`: currently it assumes that a vnode is clean if its type is `VBAD`. That's clearly incorrect in the situation we're looking at. It would be simple enough to fix:

```
--- vfs_subr.c  11 Apr 2004 21:09:22 -0000      1.490
+++ vfs_subr.c  5 Oct 2004 07:02:57 -0000
@@ -673,7 +673,7 @@
     vp->v_iflag &= ~VI_FREE;
     mtx_unlock(&vnode_free_list_mtx);
     vp->v_iflag |= VI_DOOMED;
-    if (vp->v_type != VBAD) {
+    if ((vp->v_type != VBAD) || (vp->v_data != NULL)) {
         VOP_UNLOCK(vp, 0, td);
         vgonel(vp, td);
         VI_LOCK(vp);
     }
```

This fix works for our particular case; however, it's not as sure a thing as the previous fix. Should we commit it anyway? If it works, it's probably OK.

We commit the fix, and it works. We no longer have any problems with this system:

```
grog          2004-10-06 02:09:59 UTC

FreeBSD src repository

Modified files:
  sys/kern          vfs_subr.c
Log:
vtryrecycle: Don't rely on type VBAD alone to mean that we don't need
              to clean the vnode.  If v_data is set, we still need to
              clean it.  This code change should catch all incidents of
              the previous commit (INVARIANTS only).

Revision  Changes  Path
1.530     +1 -1      src/sys/kern/vfs_subr.c
```

9

`gdb` macros

The *gdb* debugger includes a macro language. Its syntax is reminiscent of C, but different enough to be confusing. Unfortunately, there's no good reference to it. You can read the *texinfo* files which come with *gdb*, but it doesn't help much. This section is based on my experience, and it includes some practical examples.

`gdb` macro gotchas

As mentioned, *gdb* macros have a syntax which superficially resembles C, but there are many differences:

- Comments are written with a shell-like syntax: they start with `#` and continue to the end of the line.
- Commands are terminated by the end of the line, not `;`. If you want to carry a command over more than one line, use the shell-like syntax of putting a `\` character at the end of the line.
- Macro declarations don't specify parameters; the parameters which are supplied are allocated to the variables `$arg0` to `$arg9`.
- On the other hand, there's no way to find out how many parameters have been passed, and referring to parameters which haven't been passed will cause the macro execution to fail. This means that you can't have a macro which takes a variable number of parameters.
- Macro parameters are allocated lexically, with a space as a delimiter. As we'll see below, this significantly restricts what you can pass.
- You don't need to declare variables used in macros—in fact there's no provision to do so.
- Macro variables (both parameters and others) do have type, however. This means that a macro may or may not work depending on whether the name has been seen before, and if so, in which context. To assign a type to them, use a cast when assigning a value. We'll see an example below.

- Assignments require the `set` keyword, as we'll see below.
- *gdb* has to deal with three kinds of variables: variables in the program being debugged, variables local to the macros, and internal *gdb* variables. It differentiates between them in two ways:
 - `bp` might be a pointer to a buffer header in the kernel being debugged. To change the value of such a pointer, you might write:

```
(gdb) set bp = 0xc8154711
```

This changes the value in the kernel being debugged.

- In a macro, you might use the variable name `$bp` to point to a local variable. The `$` sign is *not* used in the same way as in shell scripts: it's part of the name. To change the value of such a pointer, you might write:

```
(gdb) set $bp = 0xc8154711
```

This changes only the value used in the macros.

- Finally, there are a number of internal variables. For example, to set the number of lines on a page (*gdb* doesn't understand window size changes), you might write:

```
(gdb) set height 80
```

This sets the number of lines on the window to 80. Note that there is no `=` symbol in this variant.

- Some commands don't exist (`case`, for example).
- Other commands are so lax about the syntax that, combined with the documentation, I'm not sure what the canonical version is. For example, `if` and `while` don't require parentheses around the condition argument.
- *gdb* does not seem to make a proper distinction between the operators `.` (structure member) and `->` (pointer to structure member). Again, I haven't found a rigorous distinction.

Displaying memory

In almost all debuggers, it's possible to display a block of memory in hexadecimal and character format; this is so ubiquitous that it's often called "canonical" format (in *hexdump(8)*, for example, which supplies this format with the `-C` option). *gdb* does not supply this format, which of particular concern because it's often not clear that it is displaying data correctly. In this section, we'll look at a macro to perform this simple task. The macro is called *dm* (for *display memory*). For example, we might have a data variable called `Cache`, with the following contents:

```
(gdb) p Cache
$2 = {
  blockcount = 1024,
  blocksize = 65536,
  allocount = 1024,
```

```

first = 535,
Block = 0x8173000,
stats = {
    reads = 0,
    writes = 0,
    updates = 0,
    flushes = 3,
    creates = 0,
    hits = 16738756,
    misses = 439,
    blockin = 0,
    blockout = 0
}
}

```

If we want to look at this data in raw form, we first need the address and length of the item. The address is simple, but we need to calculate the length:

```

(gdb) p sizeof Cache
$5 = 92

```

Then we can display the data:

```

(gdb) dm &Cache 92
08067160: 00 04 00 00 00 00 01 00 00 04 00 00 17 02 00 00 ~~~~~~
08067170: 00 30 17 08 00 00 00 00 00 00 00 00 00 00 00 00 ~0~~~~~
08067180: 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 00 ~~~~~~
08067190: 00 00 00 00 00 00 00 00 00 00 00 00 00 c4 69 ff 00 ~~~~~~Äiy~
080671a0: 00 00 00 00 b7 01 00 00 00 00 00 00 00 00 00 00 ~~~~~~.~~~~~
080671b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ~~~~~~

```

It would be tempting to write the following, but it doesn't work:

```

(gdb) dm &Cache sizeof Cache
A syntax error in expression, near ``.

```

The problem here is that *gdb* parses the parameters as text, so the first parameter (address) is correct, but the second parameter (length) is set to `sizeof`.

The following code implements this macro:

```

# Dump memory in "canonical" form.
# dm offset length
# This version starts lines at addr & ~0xf
define dm
set $offset = (int) $arg0
set $len = (int) $arg1
while $len > 0
# Print a line
printf "%08x: ", $offset
# byte address of start of line
set $byte = (unsigned char *) ($offset & ~0xf)
# first byte number to display
set $sbyte = $offset & 0xf
set $ebyte = $sbyte + $len
if $ebyte > 16
    set $ebyte = 16
end
# And number of bytes to print on this line
set $pos = 0
while $pos < 16
    if $pos < $sbyte || $pos >= $ebyte
        # just leave space
    else
        printf "%02x" ($byte[$pos])
    end
    set $pos = $pos + 1
end
set $len = $len - 16
end

```

first parameter, address
second parameter, length
loop over lines

address at start of line

See (3) below


```

        printf "    "
    else
        printf " %02x", *((unsigned char *) $byte) & 0xff
    end
    if $pos == 7
        printf "    "
    end
    set $pos = $pos + 1
    set $byte = $byte + 1
end
printf "    "
# Now start again with the character representation
# Start byte number on line
set $pos = 0
# byte address of start of line
set $byte = (unsigned char *) ($offset & ~0xf)
while $pos < 16
    if $pos < $sbyte || $pos >= $ebyte
# just leave space
        printf "    "
    else
        if ((*$byte & 0x7f) < 0x20)
            printf "~"
        else
            printf "%c", *$byte
        end
    end
    set $byte = $byte + 1
    set $pos = $pos + 1
end
printf "\n"
set $len = $len - 16 + ($offset & 0xf)
set $offset = ($offset + 16) & ~0xf
end
end
document dm
Dump memory in hex and chars  dm offset length
end

```

document after the event

There are a number of things to note about the way this macro has been written:

1. *gdb* automatically names the parameters `$arg0` and `$arg1`. There can be up to ten parameters.
2. We've renamed the parameter for this macro to `$offset` and `$len` to make the mess marginally more legible.
3. The pointer `$byte` is of type `unsigned char *`. Since we don't declare variables, we use casts to force a particular type.

kldstat

As we've seen, *gdb* understands nothing of kernel data structures. Many other kernel debuggers, including *ddb*, can simulate userland commands such as *ps* and the FreeBSD command *kldstat*, which shows the currently loaded kernel loadable modules (*klds*, called *LKM*s in NetBSD and OpenBSD). To get *gdb* to do the same thing, you need to write a macro which understands the kernel internal data structures. We'll call it *kldstat* after the userland macro which does the same thing.

FreeBSD keeps track of *klds* with the variable `linker_files`, described in `sys/kern/kern_linker.c`

```
static linker_file_list_t linker_files;
```

In *sys/sys/linker.h*, we read:

```
typedef struct linker_file* linker_file_t;
...
struct linker_file {
    KOBJ_FIELDS;
    int refs; /* reference count */
    int userrefs; /* kldload(2) count */
    int flags;
#define LINKER_FILE_LINKED 0x1 /* file has been fully linked */
    TAILQ_ENTRY(linker_file) link; /* list of all loaded files */
    char* filename; /* file which was loaded */
    int id; /* unique id */
    caddr_t address; /* load address */
    size_t size; /* size of file */
    int ndeps; /* number of dependencies */
    linker_file_t* deps; /* list of dependencies */
    STAILQ_HEAD(, common_symbol) common; /* list of common symbols */
    TAILQ_HEAD(, module) modules; /* modules in this file */
    TAILQ_ENTRY(linker_file) loaded; /* preload dependency support */
};
```

This is a linked list, and we access the linkage by the standard macros. *gdb* doesn't understand these macros, of course, so we have to do things manually. The best way is to start with the pre-processor output of the compilation of *sys/kern/kern_linker.o*

```
# cd /usr/src/sys/i386/compile/GENERIC
# make kern_linker.o
cc -c -O -pipe -mcpu=pentiumpro -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes -Wmissing-prototypes -Wpointer-arith -Winline -Wcast-qual -fformat-extensions -std=c99 -g -nostdinc -I- -I. -I../... -I../.../dev -I../.../contrib/dev/acpica -I../.../contrib/ipfilter -I../.../contrib/dev/ath -I../.../contrib/dev/ath/fr eebbsd -D_KERNEL -include opt_global.h -fno-common -finline-limit=15000 -fno-strict-aliasing -mno-align-long-strings -mpreferred-stack-boundary=2 -ffreestanding -Werror ../.../kern/kern_linker.c
copy and paste into the window, then add the text in italic
# cc -c -O -pipe -mcpu=pentiumpro -Wall -Wredundant-decls -Wnested-externs -Wstrict-prototypes -Wmissing-prototypes -Wpointer-arith -Winline -Wcast-qual -fformat-extensions -std=c99 -g -nostdinc -I- -I. -I../... -I../.../dev -I../.../contrib/dev/acpica -I../.../contrib/ipfilter -I../.../contrib/dev/ath -I../.../contrib/dev/ath/fre eebbsd -D_KERNEL -include opt_global.h -fno-common -finline-limit=15000 -fno-strict-aliasing -mno-align-long-strings -mpreferred-stack-boundary=2 -ffreestanding -Werror ../.../kern/kern_linker.c -C -Dd -E | less
```

Then search through the output for *linker_file* (truncating lines where necessary to fit on the page):

```
struct linker_file {
    kobj_ops_t ops;
    int refs; /* reference count */
    int userrefs; /* kldload(2) count */
    int flags;
#define LINKER_FILE_LINKED 0x1
    struct { struct linker_file *tqe_next; struct linker_file **tqe_prev; } link;
    char* filename; /* file which was loaded */
    int id; /* unique id */
    caddr_t address; /* load address */
    size_t size; /* size of file */
    int ndeps; /* number of dependencies */
    linker_file_t* deps; /* list of dependencies */
    struct { struct common_symbol *stqh_first; struct common_symbol **stqh_last; }
    struct { struct module *tqh_first; struct module **tqh_last; } modules;
    struct { struct linker_file *tqe_next; struct linker_file **tqe_prev; } loaded;
```

```
};
```

With this information, we can walk through the list manually. In *gdb* macro form, it looks like this:

```
# kldstat(8) lookalike
define kldstat
  set $file = linker_files.tqh_first
  printf "Id Refs Address      Size      Name\n"
  while ($file != 0)
    printf "%2d %4d 0x%8x %8x %s\n", \
      $file->id, \
      $file->refs, \
      $file->address, \
      $file->size, \
      $file->filename
    set $file = $file->link.tqe_next
  end
end
document kldstat
Equivalent of the kldstat(8) command, without options.
end
```

note \$ for local variables
no parentheses for functions
effectively C syntax
note set keyword for assignments

Document the macro after its definition, not before. If you try to do it before, *gdb* complains that the function doesn't exist.

Your first attempt will almost certainly fail. To re-read the macros, use *gdb*'s *source* command:

```
(gdb) source .gdbinit
```

ps

One of the most important things you want to know is what is going on in the processor. Traditional BSD commands such as *ps* have options to work on a core dump for exactly this reason, but they have been neglected in modern BSDs. Instead, here's a *gdb* macro which does nearly the same thing.

```
define ps
  set $nproc = nprocs
  set $aproc = allproc.lh_first
  set $proc = allproc.lh_first
  printf " pid   proc   addr   uid   ppid   pgrp   flag stat comm           wchan\n"
  while (--$nproc >= 0)
    set $pptr = $proc.p_pptr
    if ($pptr == 0)
      set $pptr = $proc
    end
    if ($proc.p_stat)
      printf "%5d %08x %08x %4d %5d %5d %06x %d %-10s ", \
        $proc.p_pid, $aproc, \
        $proc.p_addr, $proc.p_cred->p_ruid, $pptr->p_pid, \
        $proc.p_pgrp->pg_id, $proc.p_flag, $proc.p_stat, \
        &$proc.p_comm[0]
      if ($proc.p_wchan)
        if ($proc.p_wmesg)
          printf "%s ", $proc.p_wmesg
        end
        printf "%x", $proc.p_wchan
      end
      printf "\n"
    end
  end
```

```

        end
        set $aproc = $proc.p_list.le_next
        set $proc = $aproc
    end
end

```

This macro runs relatively slowly over a serial line, since it needs to transfer a lot of data. The output looks like this:

```

(kgdb) ps
  pid  proc      addr      uid  ppid  pgrp   flag stat comm      wchan
2638  c9a53ac0  c99f7000      0  2624  2402  004004  2  find
2626  c9980f20  c99b0000      0  2614  2402  004084  3  sort      piperd c95d2cc0
2625  c9a53440  c9a94000      0  2614  2402  004084  3  xargs     piperd c95d3080
2624  c9a53780  c9a7d000      0  2614  2402  000084  3  sh        wait c9a53780
2616  c9a535e0  c9a72000      0  2615  2402  004184  3  postdrop  piperd c95d2e00
2615  c997e1a0  c9a4d000      0  2612  2402  004084  3  sendmail  piperd c95d3b20
2614  c9a53e00  c9a41000      0  2612  2402  004084  3  sh        wait c9a53e00
2612  c997f860  c99e8000      0  2413  2402  004084  3  sh        wait c997f860
2437  c9a53c60  c9a54000      0  2432  2432  004184  3  postdrop  piperd c95d34e0
2432  c997e340  c9a1d000      0  2400  2432  004084  3  sendmail  piperd c95d31c0
2415  c997eb60  c9a21000      0  2414  2402  004084  3  cat       piperd c95d3760
2414  c997f1e0  c99f2000      0  2404  2402  000084  3  sh        wait c997f1e0
2413  c997e9c0  c9a30000      0  2404  2402  000084  3  sh        wait c997e9c0
2404  c997e4e0  c9a38000      0  2402  2402  004084  3  sh        wait c997e4e0

```

Both FreeBSD and NetBSD include some macros in the source tree. In FreeBSD you'll find them in `/usr/src/tools/debugscripts/`, and in NetBSD they're in `/usr/src/sys/gdbscripts/`. Both are good choices for examples for writing macros.

10

Spontaneous traps

Sometimes you'll see a backtrace like this:

```
Fatal trap 12: page fault while in kernel mode
fault virtual address  = 0xb
fault code             = supervisor write, page not present
instruction pointer    = 0x8:0xdd363ccc
stack pointer          = 0x10:0xdd363ca8
frame pointer          = 0x10:0xdd363ce0
code segment           = base 0x0, limit 0xffffffff, type 0x1b
                       = DPL 0, pres 1, def32 1, gran 1
processor eflags        = interrupt enabled, resume, IOPL = 0
current process         = 64462 (emacs)
trap number             = 12
panic: page fault

syncing disks... panic: bremfree: bp 0xce5f915c not locked
Uptime: 42d17h14m15s
pfs_vncache_unload(): 2 entries remaining
/dev/vmmon: Module vmmon: unloaded
Dumping 512 MB
ata0: resetting devices ..
done
```

This register dump looks confusing, but it doesn't give very much information. It's processor specific, so non-Intel traps can look quite different. What we see is:

- The trap was type 12, described as `page fault while in kernel mode`. In kernel mode you can't take a page fault, so this is fatal.
- The fault virtual address is the address of the memory reference which generated the page fault. In this case, `0xb`, it's almost certainly due to a NULL pointer dereference: a pointer was set to 0 instead of a valid address.
- The fault code gives more information about the trap. In this case, we see that it was a write access.
- The instruction pointer (eip) address has two parts: the segment (`0x8`) and the address (`0xdd363ccc`). In the case of a page fault, this is the address of the instruction which caused the fault.

- The stack pointer (esp) and frame pointer (ebp) are of limited use. Without a processor dump, it's not likely to be of much use, though in this case we note that the instruction pointer address is between the stack pointer and frame pointer address, which suggests that something has gone very wrong. The fact that the registers point to different segments is currently not of importance in this FreeBSD dump, since the two segments overlap completely.
- The remaining information is of marginal use. We've already seen the trap number, and under these circumstances you'd expect the panic message you see. The name of the process may help, though in general no user process (not even *Emacs*) should cause a panic.
- The message `syncing disks...` does not belong to the register dump. But then we get a second panic, almost certainly a result of the panic.

To find out what really went on, we need to look at the dump. Looking at the stack trace, we see:

```
(kgdb) bt
#0  doadump () at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/kern_shutdown.c:223
#1  0xc02e238a in boot (howto=0x104)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/kern_shutdown.c:355
#2  0xc02e25d3 in panic ()
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/kern_shutdown.c:508
#3  0xc0322407 in bremfree (bp=0xce5f915c)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/vfs_bio.c:632
#4  0xc0324e10 in getblk (vp=0xc42e5000, blkno=0x1bde60, size=0x4000, slpflag=0x0,
    slptimeo=0x0) at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/vfs_bio.c:2344
#5  0xc032253a in breadn (vp=0xc42e5000, blkno=0x0, size=0x0, rablkn=0x0,
    rabsz=0x0, cnt=0x0, cred=0x0, bpp=0x0)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/vfs_bio.c:690
#6  0xc03224ec in bread (vp=0x0, blkno=0x0, size=0x0, cred=0x0, bpp=0x0)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/vfs_bio.c:672
#7  0xc03efc46 in ffs_update (vp=0xc43fb250, waitfor=0x0)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/ufs/ffs/ffs_inode.c:102
#8  0xc040364f in ffs_fsync (ap=0xdd363ae0)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/ufs/ffs/ffs_vnops.c:315
#9  0xc04028be in ffs_sync (mp=0xc42d1200, waitfor=0x2, cred=0xc1616f00,
    td=0xc0513040) at vnode_if.h:612
#10 0xc0336268 in sync (td=0xc0513040, uap=0x0)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/vfs_syscalls.c:130
#11 0xc02e1fdc in boot (howto=0x100)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/kern_shutdown.c:264
#12 0xc02e25d3 in panic ()
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/kern/kern_shutdown.c:508
#13 0xc045f922 in trap_fatal (frame=0xdd363c68, eva=0x0)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c:846
#14 0xc045f602 in trap_pfault (frame=0xdd363c68, usermode=0x0, eva=0xb)
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c:760
#15 0xc045f10d in trap (frame=
    {tf_fs = 0x18, tf_es = 0x10, tf_ds = 0x10, tf edi = 0xc5844a80, tf esi = 0xdd36
    3d10, tf_ebp = 0xdd363ce0, tf_esp = 0xdd363c94, tf_ebx = 0xbfbfe644, tf_edx = 0x270c,
    tf_ecx = 0x0, tf_eax = 0xb, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0xdd363ccc, tf_cs
    = 0x8, tf_eflags = 0x10202, tf_esp = 0xdd363ccc, tf_ss = 0x0})
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c:446
#16 0xc044f3b8 in calltrap () at {standard input}:98
#17 0xc045fc2e in syscall (frame=
    {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf edi = 0x827aec0, tf esi = 0x1869d
    , tf_ebp = 0xbfbfe65c, tf_esp = 0xdd363d74, tf_ebx = 0x0, tf_edx = 0x847f380, tf_ecx
    = 0x0, tf_eax = 0x53, tf_trapno = 0x16, tf_err = 0x2, tf_eip = 0x284c4ff3, tf_cs = 0x
    1f, tf_eflags = 0x202, tf_esp = 0xbfbfe620, tf_ss = 0x2f})
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c:1035
#18 0xc044f40d in Xint0x80_syscall () at {standard input}:140
```

Here we have two panics, one at frame 2, the other at frame 12. If you have more than one panic, the one lower down the stack is the important one; any others are almost certainly a consequence of the first panic. This is also the panic that is reported in the message at the beginning: Fatal

trap 12: page fault while in kernel mode

Page faults aren't always errors, of course. In userland they happen all the time, as we've seen in the output from *vmstat*. They indicate that the program has tried to access data from an address which doesn't correspond to any page mapped in memory. It's up to the VM system to decide whether the page exists, in which case it gets it, maps it, and restarts the instruction.

In the kernel it's simpler: the kernel isn't pageable, so any page fault is a fatal error, and the system panics.

Looking at the stack trace in more detail, we see that the kernel is executing a system call (frame 17). Looking at the trap summary at the beginning, we find one of the few useful pieces of information about the environment:

```
current process          = 64462 (emacs)
```

Looking at the frame, we see:

```
(kgdb) f 17
#17 0xc045fc2e in syscall (frame=
    {tf_fs = 0x2f, tf_es = 0x2f, tf_ds = 0x2f, tf_ei = 0x827aec0, tf_esi = 0x1869d
    , tf_ebp = 0xbfbfe65c, tf_isp = 0xdd363d74, tf_ebx = 0x0, tf_edx = 0x847f380, tf_ecx
    = 0x0, tf_eax = 0x53, tf_trapno = 0x16, tf_err = 0x2, tf_eip = 0x284c4ff3, tf_cs = 0x
    1f, tf_eflags = 0x202, tf_esp = 0xbfbfe620, tf_ss = 0x2f})
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c:1035
1035          error = (*callp->sy_call)(td, args);
```

Which system call is this? *syscall* is no normal function: it's a trap function,

```
(kgdb) p *callp
$1 = {
    sy_narg = 0x10003,
    sy_call = 0xc02ef060 <setitimer>
}
```

It would be tempting to think that the error occurred here: that's where the trap frame appears to be pointing. In fact, though, that's not the case. Like *syscall*, the trap frame isn't a real C stack frame, and it confuses *gdb*, which thinks it's part of the called function, which is hidden in the middle. On this i386 architecture machine, the registers *eip* and *esp* of the trap frame (frame 15) tell us where the error really occurred: *eip* is 0xdd363ccc, and *esp* is 0xdd363ccc. That's strange. They're both the same. That's obviously wrong.

Looking at the code at this location, we see:

```
(kgdb) x/10i 0xdd363ccc
0xdd363ccc:    add    %al, (%eax)
0xdd363cce:    add    %al, (%eax)
0xdd363cd0:    popf
0xdd363cd1:    xchg   %al, (%ecx)
0xdd363cd3:    add    %ch, %al
0xdd363cd5:    dec    %edx
0xdd363cd6:    test   %al, %ch
0xdd363cd8:    lock  pop  %eax
0xdd363cda:    pop    %ebx
0xdd363cdb:    lds    0x40c5844a(%eax), %eax
```

There are two strange things about this code: first, it doesn't appear to have a symbolic name as-

sociated with it. Normally you'd expect to see something like:

```
kgdb) x/10i 0xc02ef078
0xc02ef078 <setitimer+24>:    inc    %ebp
0xc02ef079 <setitimer+25>:    fadds   (%eax)
0xc02ef07b <setitimer+27>:    add    %al, (%eax)
0xc02ef07d <setitimer+29>:    add    %al, 0xd76023e(%ebx)
0xc02ef083 <setitimer+35>:    mov    $0x16, %eax
0xc02ef088 <setitimer+40>:    jmp    0xc02ef257 <setitimer+503>
0xc02ef08d <setitimer+45>:    lea    0x0(%esi), %esi
0xc02ef090 <setitimer+48>:    mov    0x4(%esi), %ebx
0xc02ef093 <setitimer+51>:    test   %ebx, %ebx
0xc02ef095 <setitimer+53>:    je     0xc02ef0b9 <setitimer+89>
```

This code is also a long way from *setitimer*. In addition, the code doesn't seem to make any sense.

In fact, the address is well outside the bounds of kernel code:

```
(kgdb) kldstat
Id Refs Address      Size      Name
 1   15 0xc0100000    53ac68  kernel
 2    1 0xc4184000     5000  linprocfs.ko
 3    3 0xc43c1000    17000  linux.ko
 4    2 0xc422c000     a000  ibcs2.ko
 5    1 0xc43d8000     3000  ibcs2_coff.ko
 6    1 0xc4193000     2000  rtc.ko
 7    1 0xc1ed7000     9000  vmmon_up.ko
 8    1 0xc4264000     4000  if_tap.ko
 9    1 0xc7a40000     4000  snd_via8233.ko
10    1 0xc7aaa000    18000  snd_pcm.ko
```

Clearly, any address above 0xd0000000 is not a valid code address. So somehow we've ended up in the woods. How?

Things aren't made much easier by the fact that we don't have a stack frame for *setitimer*. It does tell us one thing, though: things must have gone off track in *setitimer* itself, and not in a function it called. Otherwise we would see the stack frame created by *setitimer* in the backtrace.

We obviously can't find the stack frame from the register values saved in the trap frame, because they're incorrect. Instead, we need to go from the stack frame of the calling function, *syscall*. Unfortunately, *gdb* is too stupid to be of much help here. Instead we dump the memory area in hexadecimal:

```
(kgdb) i reg
eax          0x0          0x0
ecx          0x0          0x0
edx          0x0          0x0
ebx          0xbfbfe644    0xbfbfe644
esp          0xdd363884    0xdd363884
ebp          0xdd363d40    0xdd363d40
esi          0xdd363d10    0xdd363d10
edi          0xc5844a80    0xc5844a80
eip          0xc045fc2e    0xc045fc2e
...
```

Hmm. This is interesting: even on entry, the *esp* values are above 0xdd000000. Normally they should be below the kernel text. Still, there's memory there, so it's not the immediate problem. The part of the stack we're interested in is between the values of the *%ebp* and *%esp* registers. There's quite a bit of data here:


```
(kgdb) p $ebp - $esp
$5 = 0x4bc
(kgdb) p/d $ebp - $esp           in decimal, overriding .gdbinit
$6 = 1212
```

In this case, it's probably better to look at the code first. It starts like this:

```
void
syscall(frame)
{
    struct trapframe frame;
    {
        caddr_t params;
        struct sysent *callp;
        struct thread *td = curthread;
        struct proc *p = td->td_proc;
        register_t orig_tf_eflags;
        u_int sticks;
        int error;
        int nargs;
        int args[8];
        u_int code;
```

We can normally look at the stack frame with `info local`, but in this case it doesn't work:

```
(kgdb) i loc
params = 0xbfbfe624---Can't read userspace from dump, or kernel process---
```

There are other ways. Normally the compiler allocates automatic variables in the order in which they appear in the source, but there are exceptions: it can allocate them to registers, in which case they don't appear on the stack at all, or it can optimize the layout to reduce stack usage. In this case, we have to check them all:

```
(kgdb) p &params
$7 = (char **) 0xdd363d08
(kgdb) p &callp
$8 = (struct sysent **) 0xdd363d04
(kgdb) p &td
Can't take address of "td" which isn't an lvalue.
(kgdb) p &p
Can't take address of "p" which isn't an lvalue.
(kgdb) p &orig_tf_eflag
$9 = (register_t *) 0xdd363d00
(kgdb) p &sticks
$10 = (u_int *) 0xdd363cfc
(kgdb) p &error
Can't take address of "error" which isn't an lvalue.
(kgdb) p &nargs
$11 = (int *) 0xdd363cf8
(kgdb) p &args
$12 = (int (*)[8]) 0xdd363d10
(kgdb) p &code
$13 = (u_int *) 0xdd363d0c
```

The error message `Can't take address` indicates that the compiler has allocated a register for this value. Interestingly, the last automatic variables are `args` and `code`, but they have been assigned the highest addresses. The lowest stack address is of `nargs`, `0xdd363cf8`. That's where we need to look. Below that on the stack we may find temporary storage, but below that we should find the two parameters for the `syscall` function, followed (in descending order) by the return address (`0xc045fc2e`). The return address is particularly useful because we can use it to locate the stack frame in the first place.

It would be nice to be able to dump memory backwards, but that's not possible. How far down the stack should we go? One way is to look at the stack frame of the next function. We have that in frame 15: the `esp` is `0xdd363ccc`. That's not so far down, so let's see what we find:

```
(kgdb) x/20x 0xdd363ccc
0xdd363cc0: 0xc5844ae8 0x00000000 0x00000000 0x00000000
0xdd363cd0: 0x0001869d 0xc5844ae8 0xc55b58f0 0xc5844a80
0xdd363ce0: 0xdd363d40 0xc045fc2e 0xc55b58f0 0xdd363d10
0xdd363cf0: 0xc04de816 0x00000409 0x00000003 0x00009a8d
```

When dumping data in this format, it's a good idea to start with an address with the last (hex) digit 0; otherwise it's easy to get confused about the address of each word.

We find our return address at `0xdd363ce4`. That means that the words at `0xdd363ce8` and `0xdd363cec` are the parameters, so there are apparently two words of temporary storage on the stack.

It's worth looking at the parameters. Again, the call is:

```
1035          error = (*callp->sy_call)(td, args);
```

So we'd expect to see the value of `td` in location `0xdd363ce8`, and the value of `args` in location `0xdd363cec`. Well, `&args` is really in `0xdd363cec`, but the value of `td` is

```
(kgdb) p td
$1 = (struct thread *) 0xdd363d10
```

Look familiar? That's the value of `args`. This is supposed to be a kernel thread descriptor, so the address on the local stack has to be wrong. There are a number of ways this could have happened:

- The variable may no longer be needed, so it could have been optimized away. This is unlikely here, since we've only just used it to call a function. We don't seem to have returned from the function, so there was no time for the calling function to reuse the storage space.
- Maybe the value was correct, but the called function could have changed the value of the copy of the value passed as an argument. This is possible, but it's pretty rare that a function changes the value of the arguments passed to it.
- Maybe a random pointer bug resulted in the value of `td` being overwritten by the called function or one of the functions that called it.

Which is it? Let's look at what might have happened in `setitimer`. Where is it? *gdb* lists it for you, but it doesn't tell you where it is:

```
(kgdb) l setitimer
455      /* ARGSUSED */
456      int
457      setitimer(struct thread *td, struct setitimer_args *uap)
458      {
459          struct proc *p = td->td_proc;
460          struct itimerval aitv;
461          struct timeval ctv;
462          struct itimerval *itvp;
463          int s, error = 0;
464
```

```

465         if (uap->which > ITIMER_PROF)
466             return (EINVAL);
467         itvp = uap->itv;
468         if (itvp && (error = copyin(itvp, &aitv, sizeof(struct itimerval))))
469             return (error);
470
471         mtx_lock(&Giant);
472
473         if ((uap->itv = uap->oitv) &&
474             (error = getitimer(td, (struct getitimer_args *)uap))) {
475             goto done2;
476         }
477         if (itvp == 0) {
478             error = 0;
479             goto done2;
480         }
481         if (itimerfix(&aitv.it_value)) {
482             error = EINVAL;

```

It doesn't tell you where it is, though; you can fake that by setting a breakpoint on the function. Never mind that you can't use the breakpoint; at least it tells you where it is:

```

(kgdb) b setitimer
Breakpoint 1 at 0xc02ef072: file /usr/src/sys/kern/kern_time.c, line 459.

```

The most interesting things to look at here are the automatic variables: we can try to find them on the stack. Unfortunately, since *gdb* doesn't recognize the stack frame for the function, we can't get much help from it. Doing it manually can be cumbersome: we have two ints (easy), two struct pointers (not much more difficult) and two structs, for which we need to find the sizes. Using *etags*, we find:

```

struct itimerval {
    struct timeval it_interval;    /* timer interval */
    struct timeval it_value;      /* current value */
};
(another file)
struct timeval { int i; };

```

So our struct `timeval` is 4 bytes long, and struct `itimerval` is 8 bytes long. That makes a total of 28 bytes on the stack. Looking at the assembler code, however, we see:

```

(kgdb) x/10i setitimer
0xc02ef060 <setitimer>: push    %ebp
0xc02ef061 <setitimer+1>:  mov     %esp,%ebp
0xc02ef063 <setitimer+3>:  sub     $0x38,%esp

```

That's our standard prologue, alright, but it's reserving 0x38 or 56 bytes of local storage, twice what we need for the automatic variables. Probably the compiler's using them for other purposes, but it could also mean that the variables aren't where we think they are. In fact, as the code continues, we see this to be true:

```

0xc02ef066 <setitimer+6>:  mov     %ebx,0xffffffff4(%ebp)
0xc02ef069 <setitimer+9>:  mov     %esi,0xffffffff8(%ebp)
0xc02ef06c <setitimer+12>: mov     %edi,0xffffffffc(%ebp)

```

In other words, it's saving the registers `ebx`, `esi` and `edi` on the stack immediately below the stack frame. That accounts for 12 further words. It also gives us a chance to check whether we know what the contents were. This will give us some confirmation that we're on the right track.

We call `setitimer` from this line:

```

1035                      error = (*callp->sy_call)(td, args);
(kgdb) i 1i 1035          get info about the instruction addresses
Line 1035 of "/src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c"
      starts at address 0xc045fc1e <syscall+638> and ends at 0xc045fc30 <syscall+656>.
(kgdb) x/10i 0xc045fc1e  look at the code
0xc045fc1e <syscall+638>:      mov     %esi, (%esp, 1)
0xc045fc21 <syscall+641>:      lea     0xfffffd0(%ebp), %eax
0xc045fc24 <syscall+644>:      mov     %eax, 0x4(%esp, 1)
0xc045fc28 <syscall+648>:      mov     0xfffffc4(%ebp), %edx
0xc045fc2b <syscall+651>:      call   *0x4(%edx)

```

This code is confusing because some instructions use `ebp` relative addressing, and others use `esp` relative addressing. We know what the contents of the `ebp` and `esp` registers were when these instructions were executed: `ebp` is saved on the stack at location `0xdd363ce0`: it's `0xdd363d40`. At the start of the instruction sequence, `esp` is pointing to the location above the return address, `0xdd363ce8`:

| | | | | |
|-------------|------------|------------|------------|------------|
| 0xdd363cc0: | 0xc5844ae8 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0xdd363cd0: | 0x0001869d | 0xc5844ae8 | 0xc55b58f0 | 0xc5844a80 |
| 0xdd363ce0: | 0xdd363d40 | 0xc045fc2e | | |
| | | <i>esp</i> | 0xc55b58f0 | 0xdd363d10 |
| 0xdd363cf0: | 0xc04de816 | 0x00000409 | 0x00000003 | 0x00009a8d |
| 0xdd363d00: | 0x00000202 | 0xc05134f8 | 0xbfbfe624 | 0x00000053 |
| 0xdd363d10: | 0x00000000 | 0x00000000 | 0x00000000 | 0x00009a8d |
| 0xdd363d20: | 0x00000000 | 0xc55ba9a0 | 0xc1619500 | 0x00000001 |
| 0xdd363d30: | 0xffffffff | 0x00000000 | 0x0001869d | 0x0827aec0 |
| 0xdd363d40: | | | | |
| <i>ebp</i> | 0xbfbfe65c | 0xc044f40d | 0x0000002f | 0x0000002f |
| 0xdd363d50: | 0x0000002f | 0x0827aec0 | 0x0001869d | 0xbfbfe65c |

Looking at these instructions one by one, we see:

```

0xc045fc1e <syscall+638>:      mov     %esi, (%esp, 1)

```

This moves the value in the `esi` register to location `0xdd363ce8`. This is the first parameter, `td`.

```

0xc045fc21 <syscall+641>:      lea     0xfffffd0(%ebp), %eax

```

This loads the effective address (`lea`) of offset `-0x30` from the `ebp` register contents, address `0xdd363d10`, into register `eax`. This data is in the calling function's local stack frame. Currently it's 0, though it may not have been at the time.

```

0xc045fc24 <syscall+644>:      mov     %eax, 0x4(%esp, 1)

```

This stores register `eax` at 4 from the `esp` register contents, address `0xdd363cec`. This is the second parameter to the function call, `args`. We can confirm that by looking at the local variables we printed out before:

```

(kgdb) p &args
$12 = (int *) [8] 0xdd363d10

```

As a result, we'd expect the contents of location `0xdd363cec` to contain `0xdd363d10`, which it does.

```

0xc045fc28 <syscall+648>:      mov     0xffffffffc4(%ebp),%edx
0xc045fc2b <syscall+651>:      call    *0x4(%edx)

```

This loads the contents of the storage location at offset `-0x3c` from the contents of the `ebp` into the `edx` register. Register `ebp` contains `0xdd363d40`, so we load `edx` from location `0xdd363d04`. Again, we confirm with the locations we printed out before:

```

(kgdb) p &callp
$8 = (struct sysent **) 0xdd363d04

```

Finally, this instruction:

```

1035                                error = (*callp->sy_call)(td, args);

```

calls the function whose address is at offset 4 from where `edx`. It's pretty clear that this worked, since we ended up in the correct function.

Where we are now

We've now found our way to the function call. We know that we the call was effectively:

```

setitimer (0xc55b58f0, 0xdd363d10)

```

We still haven't found out what happened, so the next thing to look at is the called function, `setitimer`.

Entering setitimer

On entering `setitimer`, we see:

```

int
setitimer(struct thread *td, struct setitimer_args *uap)
{
    struct proc *p = td->td_proc;
    struct itimerval aiov;
    struct timeval ctv;
    struct itimerval *itvp;
    int s, error = 0;

    if (uap->which > ITIMER_PROF)
        return (EINVAL);
    itvp = uap->itv;
    if (itvp && (error = copyin(itvp, &aiov, sizeof(struct itimerval))))
        return (error);

    mtx_lock(&Giant);

    if ((uap->itv = uap->oiov) &&
        (error = getitimer(td, (struct getitimer_args *)uap))) {
        goto done2;
    }
    if (itvp == 0) {
        error = 0;
        goto done2;
    }
    if (itimerfix(&aiov.it_value)) {
        error = EINVAL;
        goto done2;
    }
}

```

```

    }
    if (!timevalisset(&aitv.it_value)) {
        timevalclear(&aitv.it_interval);
    } else if (itimerfix(&aitv.it_interval)) {
        error = EINVAL;
        goto done2;
    }
    s = splclock(); /* XXX: still needed ? */
    if (uap->which == ITIMER_REAL) {
        if (timevalisset(&p->p_realtimer.it_value))
            callout_stop(&p->p_itcallout);
        if (timevalisset(&aitv.it_value))
            callout_reset(&p->p_itcallout, tvtohz(&aitv.it_value),
                          realitexpire, p);
        getmicrouptime(&ctv);
        timevaladd(&aitv.it_value, &ctv);
        p->p_realtimer = aitv;
    } else {
        p->p_stats->p_timer[uap->which] = aitv;
    }
    splx(s);
done2:
    mtx_unlock(&Giant);
    return (error);
}

```

The first code to be executed is the function prologue:

```

(kgdb) x/200i setitimer
prologue
0xc02ef060 <setitimer>: push    %ebp                save ebp
0xc02ef061 <setitimer+1>: mov     %esp,%ebp          and create a new stack frame
0xc02ef063 <setitimer+3>: sub     $0x38,%esp          make space on stack
0xc02ef066 <setitimer+6>: mov     %ebx,0xffffffff4(%ebp) save ebx
0xc02ef069 <setitimer+9>: mov     %esi,0xffffffff8(%ebp) save esi
0xc02ef06c <setitimer+12>: mov     %edi,0xffffffffc(%ebp) save edi

```

After executing the prologue, then, we'd expect to see the `esp` value to be 0x38 lower than the `ebp` value. It doesn't have to stay that way, but it shouldn't be any higher. The trap message shows the values:

```

stack pointer      = 0x10:0xdd363ca8
frame pointer      = 0x10:0xdd363ce0

```

That looks fine: the difference is the expected value of 0x38. But looking at the trap frame in the backtrace, we see:

```

#15 0xc045f10d in trap (frame=
    {tf_fs = 0x18, tf_es = 0x10, tf_ds = 0x10, tf_edi = 0xc5844a80,
      tf_esi = 0xdd363d10, tf_ebp = 0xdd363ce0, tf_esp = 0xdd363c94,
      tf_ebx = 0xbfbfe644, tf_edx = 0x270c, tf_ecx = 0x0, tf_eax = 0xb,
      tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0xdd363ccc, tf_cs = 0x8,
      tf_eflags = 0x10202, tf_esp = 0xdd363ccc, tf_ss = 0x0})
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c:446

```

What's wrong there? If you look at the function `trap_fatal`, conveniently in the same file as `syscall`, `/sys/i386/i386/trap.c`, we see that it's `trap_fatal` which prints out the values:

```

static void
trap_fatal(frame, eva)
    struct trapframe *frame;
    vm_offset_t eva;

```

```

{
    int code, type, ss, esp;
    struct soft_segment_descriptor softseg;

...
    printf("instruction pointer      = 0x%x:0x%x\n",
           frame->tf_cs & 0xffff, frame->tf_eip);
    if ((ISPL(frame->tf_cs) == SEL_UPL) || (frame->tf_eflags & PSL_VM)) {
        ss = frame->tf_ss & 0xffff;
        esp = frame->tf_esp;
    } else {
        ss = GSEL(GDATA_SEL, SEL_KPL);
        esp = (int)&frame->tf_esp;
    }
    printf("stack pointer          = 0x%x:0x%x\n", ss, esp);
    printf("frame pointer          = 0x%x:0x%x\n", ss, frame->tf_ebp);

```

The parameter `frame` is the same frame that we've been looking at:

```

(kgdb) f 15
#15 0xc045f10d in trap (frame=
    {tf_fs = 0x18, tf_es = 0x10, tf_ds = 0x10, tf edi = 0xc5844a80, tf esi = 0xdd36
3d10, tf_ebp = 0xdd363ce0, tf_esp = 0xdd363c94, tf_ebx = 0xbfbfe644, tf_edx = 0x270c,
    tf_ecx = 0x0, tf_eax = 0xb, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0xdd363ccc, tf_c
s = 0x8, tf_eflags = 0x10202, tf_esp = 0xdd363ccc, tf_ss = 0x0})
    at /src/FreeBSD/5-CURRENT-WANTADILLA/src/sys/i386/i386/trap.c:446
446                                     (void) trap_pfault(&frame, FALSE, eva);
Current language: auto; currently c
(kgdb) p &frame
$10 = (struct trapframe *) 0xdd363c68

```

Looking at the code, it's not surprising that the values of `eip` and `ebp` agree with what's in the trap frame. But what about `esp`? `trap_fatal` calculates that itself. Why does it do so, and why does it come to a different value? The test is:

```

if ((ISPL(frame->tf_cs) == SEL_UPL) || (frame->tf_eflags & PSL_VM)) {

```

The first test checks whether the saved code segment (`cs`) is a user code segment (the lowest two bits are 3). We have:

```

(kgdb) p frame->tf_cs
$12 = 0x8

```

So it's not that. The second one checks whether we're running in virtual 8086 mode, as signaled by the `PSL_VM` bit in the saved `eflags` value (see `sys/i386/include/psl.h`). That's not the case either:

```

(kgdb) p frame->tf_eflags
$13 = 0x10202

```

This is probably the normal case: instead of saved contents of `esp` value, it uses the address of the saved contents.

Summary

Working through a dump like this is an open-ended matter. It's never certain whether continuing will find something or not. This example shows a relatively painful trace through a processor dump. Will we find any more? It's uncertain. The dump came from a system with known hardware problems, so it's quite possible that all that can be found is just what kind of problem occurred.

NAME

ddb — interactive kernel debugger

SYNOPSIS

options DDB

To prevent activation of the debugger on kernel panic(9):

options KDB_UNATTENDED

DESCRIPTION

The **ddb** kernel debugger has most of the features of the old **kdb**, but with a more rational syntax inspired by **gdb**(1). If linked into the running kernel, it can be invoked locally with the **debug** keymap(5) action. The debugger is also invoked on kernel panic(9) if the *debug.debugger_on_panic* sysctl(8) MIB variable is set non-zero, which is the default unless the KDB_UNATTENDED option is specified.

The current location is called ‘dot’. The ‘dot’ is displayed with a hexadecimal format at a prompt. Examine and write commands update ‘dot’ to the address of the last line examined or the last location modified, and set ‘next’ to the address of the next location to be examined or changed. Other commands do not change ‘dot’, and set ‘next’ to be the same as ‘dot’.

The general command syntax is: **command**[/*modifier*] *address*[,*count*]

A blank line repeats the previous command from the address ‘next’ with count 1 and no modifiers. Specifying *address* sets ‘dot’ to the address. Omitting *address* uses ‘dot’. A missing *count* is taken to be 1 for printing commands or infinity for stack traces.

The **ddb** debugger has a feature like the **more**(1) command for the output. If an output line exceeds the number set in the *\$lines* variable, it displays “--db_more--” and waits for a response. The valid responses for it are:

SPC one more page

RET one more line

q abort the current command, and return to the command input mode

Finally, **ddb** provides a small (currently 10 items) command history, and offers simple emacs-style command line editing capabilities. In addition to the emacs control keys, the usual ANSI arrow keys might be used to browse through the history buffer, and move the cursor within the current line.

COMMANDS**examine****x**

Display the addressed locations according to the formats in the modifier. Multiple modifier formats display multiple locations. If no format is specified, the last formats specified for this command is used.

The format characters are:

| | |
|---|---|
| b | look at by bytes (8 bits) |
| h | look at by half words (16 bits) |
| l | look at by long words (32 bits) |
| a | print the location being displayed |
| A | print the location with a line number if possible |
| x | display in unsigned hex |
| z | display in signed hex |
| o | display in unsigned octal |

| | |
|---------|---|
| d | display in signed decimal |
| u | display in unsigned decimal |
| r | display in current radix, signed |
| c | display low 8 bits as a character. Non-printing characters are displayed as an octal escape code (e.g., '\000'). |
| s | display the null-terminated string at the location. Non-printing characters are displayed as octal escapes. |
| m | display in unsigned hex with character dump at the end of each line. The location is also displayed in hex at the beginning of each line. |
| i | display as an instruction |
| I | display as an instruction with possible alternate formats depending on the machine: |
| alpha | Show the registers of the instruction. |
| amd64 | No alternate format. |
| i386 | No alternate format. |
| ia64 | No alternate format. |
| powerpc | No alternate format. |
| sparc64 | No alternate format. |

xf

Examine forward: Execute an examine command with the last specified parameters to it except that the next address displayed by it is used as the start address.

xb

Examine backward: Execute an examine command with the last specified parameters to it except that the last start address subtracted by the size displayed by it is used as the start address.

print[/acdoruxz]

Print *addrs* according to the modifier character (as described above for *examine*). Valid formats are: a, x, z, o, d, u, r, and c. If no modifier is specified, the last one specified to it is used. *addr* can be a string, in which case it is printed as it is. For example:

```
print/x "eax = " $eax "\necx = " $ecx "\n"
```

will print like:

```
eax = xxxxxx
ecx = yyyyyy
```

write[/bhl] *addr* *expr1* [*expr2* ...]

Write the expressions specified after *addr* on the command line at succeeding locations starting with *addr*. The write unit size can be specified in the modifier with a letter b (byte), h (half word) or l (long word) respectively. If omitted, long word is assumed.

Warning: since there is no delimiter between expressions, strange things may happen. It is best to enclose each expression in parentheses.

set \$variable [=] *expr*

Set the named variable or register with the value of *expr*. Valid variable names are described below.

break[/u]

Set a break point at *addr*. If *count* is supplied, continues *count* - 1 times before stopping at the break point. If the break point is set, a break point number is printed with '#'. This number can be used in deleting the break point or adding conditions to it.

If the u modifier is specified, this command sets a break point in user space address. Without the u option, the address is considered in the kernel space, and wrong space address is rejected with an error message. This modifier can be used only if it is supported by machine dependent routines.

Warning: If a user text is shadowed by a normal user space debugger, user space break points may not work correctly. Setting a break point at the low-level code paths may also cause strange behavior.

delete *addr*

delete *#number*

Delete the break point. The target break point can be specified by a break point number with #, or by using the same *addr* specified in the original **break** command.

step [/p]

Single step *count* times (the comma is a mandatory part of the syntax). If the p modifier is specified, print each instruction at each step. Otherwise, only print the last instruction.

Warning: depending on machine type, it may not be possible to single-step through some low-level code paths or user space code. On machines with software-emulated single-stepping (e.g., pmax), stepping through code executed by interrupt handlers will probably do the wrong thing.

continue [/c]

Continue execution until a breakpoint or watchpoint. If the c modifier is specified, count instructions while executing. Some machines (e.g., pmax) also count loads and stores.

Warning: when counting, the debugger is really silently single-stepping. This means that single-stepping on low-level code may cause strange behavior.

until [/p]

Stop at the next call or return instruction. If the p modifier is specified, print the call nesting depth and the cumulative instruction count at each call or return. Otherwise, only print when the matching return is hit.

next [/p]

match [/p]

Stop at the matching return instruction. If the p modifier is specified, print the call nesting depth and the cumulative instruction count at each call or return. Otherwise, only print when the matching return is hit.

trace [/u] [*frame*] [*count*]

Stack trace. The u option traces user space; if omitted, **trace** only traces kernel space. *count* is the number of frames to be traced. If *count* is omitted, all frames are printed.

Warning: User space stack trace is valid only if the machine dependent code supports it.

search [/bhl] *addr value* [*mask*] [*count*]

Search memory for *value*. This command might fail in interesting ways if it does not find the searched-for value. This is because ddb does not always recover from touching bad memory. The optional *count* argument limits the search.

show all procs [/m]

ps [/m]

Display all process information. The process information may not be shown if it is not supported in the machine, or the bottom of the stack of the target process is not in the main memory at that time. The m modifier will alter the display to show VM map addresses for the process and not show other info.

show registers [/u]

Display the register set. If the u option is specified, it displays user registers instead of kernel or currently saved one.

Warning: The support of the u modifier depends on the machine. If not supported, incorrect information will be displayed.

show map[/f] *addr*

Prints the VM map at *addr*. If the *f* modifier is specified the complete map is printed.

show object[/f] *addr*

Prints the VM object at *addr*. If the *f* option is specified the complete object is printed.

show watches

Displays all watchpoints.

reset

Hard reset the system.

watch *addr, size*

Set a watchpoint for a region. Execution stops when an attempt to modify the region occurs. The *size* argument defaults to 4. If you specify a wrong space address, the request is rejected with an error message.

Warning: Attempts to watch wired kernel memory may cause unrecoverable error in some systems such as *i386*. Watchpoints on user addresses work best.

hwatch *addr, size*

Set a hardware watchpoint for a region if supported by the architecture. Execution stops when an attempt to modify the region occurs. The *size* argument defaults to 4.

Warning: The hardware debug facilities do not have a concept of separate address spaces like the watch command does. Use **hwatch** for setting watchpoints on kernel address locations only, and avoid its use on user mode address spaces.

dhwatc *addr, size*

Delete specified hardware watchpoint.

gdb

Toggles between remote GDB and DDB mode. In remote GDB mode, another machine is required that runs `gdb(1)` using the remote debug feature, with a connection to the serial console port on the target machine. Currently only available on the *i386* and *Alpha* architectures.

help

Print a short summary of the available commands and command abbreviations.

VARIABLES

The debugger accesses registers and variables as *\$name*. Register names are as in the “**show registers**” command. Some variables are suffixed with numbers, and may have some modifier following a colon immediately after the variable name. For example, register variables can have a *u* modifier to indicate user register (e.g., *\$eax:u*).

Built-in variables currently supported are:

| | |
|-----------------|--|
| radix | Input and output radix |
| maxoff | Addresses are printed as ‘symbol’+offset unless offset is greater than maxoff. |
| maxwidth | The width of the displayed line. |
| lines | The number of lines. It is used by “more” feature. |
| tabstops | Tab stop width. |
| workxx | Work variable. <i>xx</i> can be 0 to 31. |

EXPRESSIONS

Almost all expression operators in C are supported except ‘~’, ‘^’, and unary ‘&’. Special rules in **ddb** are:

Identifiers The name of a symbol is translated to the value of the symbol, which is the address of the corresponding object. ‘.’ and ‘:’ can be used in the identifier. If supported by an object format dependent routine, [*fi lename:*]*func:lineno*, [*fi lename:*]*variable*, and [*fi lename:*]*lineno* can be

accepted as a symbol.

Numbers Radix is determined by the first two letters: 0x: hex, 0o: octal, 0t: decimal; otherwise, follow current radix.

. 'dot'

+ 'next'

. . address of the start of the last line examined. Unlike 'dot' or 'next', this is only changed by "examine" or "write" command.

' last address explicitly specified.

\$variable Translated to the value of the specified variable. It may be followed by a : and modifiers as described above.

a#b a binary operator which rounds up the left hand side to the next multiple of right hand side.

**expr* indirection. It may be followed by a ': and modifiers as described above.

HINTS

On machines with an ISA expansion bus, a simple NMI generation card can be constructed by connecting a push button between the A01 and B01 (CHCHK# and GND) card fingers. Momentarily shorting these two fingers together may cause the bridge chipset to generate an NMI, which causes the kernel to pass control to **ddb**. Some bridge chipsets do not generate a NMI on CHCHK#, so your mileage may vary. The NMI allows one to break into the debugger on a wedged machine to diagnose problems. Other bus' bridge chipsets may be able to generate NMI using bus specific methods.

SEE ALSO

`gdb(1)`

HISTORY

The **ddb** debugger was developed for Mach, and ported to 386BSD 0.1. This manual page translated from **-man** macros by Garrett Wollman.

NAME

ddb – in-kernel debugger

SYNOPSIS

options DDB

To enable history editing:

options DDB_HISTORY_SIZE=integer

To disable entering **ddb** upon kernel panic:

options DDB_ONPANIC=0

DESCRIPTION

ddb is the in-kernel debugger. It may be entered at any time via a special key sequence, and optionally may be invoked when the kernel panics.

ENTERING THE DEBUGGER

Unless **DDB_ONPANIC** is set to 0, **ddb** will be activated whenever the kernel would otherwise panic.

ddb may also be activated from the console. In general, sending a break on a serial console will activate . There are also key sequences for each port that will activate **ddb** from the keyboard:

| | |
|---------|--|
| alpha | <Ctrl>-<Alt>-<Esc> on PC style keyboards. |
| amiga | <LAlt>-<LAmiga>-<F10> |
| atari | <Alt>-<LeftShift>-<F9> |
| hp300 | <Shift>-<Reset> |
| hpcmips | <Ctrl>-<Alt>-<Esc> |
| hpcsh | <Ctrl>-<Alt>-<Esc> |
| i386 | <Ctrl>-<Alt>-<Esc> <Break> on serial console. |
| mac68k | <Command>-<Power>, or the Interrupt switch. |
| macppc | Some models: <Command>-<Option>-<Power> |
| mvme68k | Abort switch on CPU card. |
| pmax | <Do> on LK-201 rcons console. <Break> on serial console. |
| sparc | <L1>-A, or <Stop>-A on a Sun keyboard. <Break> on serial console. |
| sun3 | <L1>-A, or <Stop>-A on a Sun keyboard. <Break> on serial console. |
| sun3x | <L1>-A, or <Stop>-A on a Sun keyboard. <Break> on serial console. |
| x68k | Interrupt switch on the body. |

In addition, **ddb** may be explicitly activated by the debugging code in the kernel if **DDB** is configured.

COMMAND SYNTAX

The general command syntax is:

command [/modifier] address [,count]

The current memory location being edited is referred to as *dot*, and the next location is *next*. They are displayed as hexadecimal numbers.

Commands that examine and/or modify memory update *dot* to the address of the last line examined or the last location modified, and set *next* to the next location to be examined or modified. Other commands don't

change *dot*, and set *next* to be the same as *dot*.

A blank line repeats the previous command from the address *next* with the previous **count** and no modifiers. Specifying **address** sets *dot* to the address. If **address** is omitted, *dot* is used. A missing **count** is taken to be 1 for printing commands, and infinity for stack traces.

The syntax:

,count

repeats the previous command, just as a blank line does, but with the specified **count**.

ddb has a `more(1)`-like functionality; if a number of lines in a command's output exceeds the number defined in the *lines* variable, then **ddb** displays "--db more--" and waits for a response, which may be one of:

<return> one more line.

<space> one more page.

q abort the current command, and return to the command input mode.

If **ddb** history editing is enabled (by defining the

options DDB_HISTORY_SIZE=num

kernel option), then a history of the last **num** commands is kept. The history can be manipulated with the following key sequences:

<Ctrl>-P retrieve previous command in history (if any).

<Ctrl>-N retrieve next command in history (if any).

COMMANDS

ddb supports the following commands:

!address[(*expression*[, ...])]

A synonym for **call**.

break[/u] *address*[,*count*]

Set a breakpoint at *address*. If *count* is supplied, continues (*count*-1) times before stopping at the breakpoint. If the breakpoint is set, a breakpoint number is printed with '#'. This number can be used to **delete** the breakpoint, or to add conditions to it.

If /u is specified, set a breakpoint at a user-space address. Without /u, *address* is considered to be in the kernel-space, and an address in the wrong space will be rejected, and an error message will be emitted. This modifier may only be used if it is supported by machine dependent routines.

Warning: if a user text is shadowed by a normal user-space debugger, user-space breakpoints may not work correctly. Setting a breakpoint at the low-level code paths may also cause strange behavior.

bt[/u] [*frame-address*][,*count*]

A synonym for **trace**.

bt/t [*pid*][,*count*]

A synonym for **trace**.

call *address*[(*expression*[, ...])]

Call the function specified by *address* with the argument(s) listed in parentheses. Parentheses may be omitted if the function takes no arguments. The number of arguments is currently limited to 10.

continue [/c]

Continue execution until a breakpoint or watchpoint. If /c is specified, count instructions while executing. Some machines (e.g., pmax) also count loads and stores.

Warning: when counting, the debugger is really silently single-stepping. This means that single-stepping on low-level may cause strange behavior.

delete *address* [#*number*]

Delete a breakpoint. The target breakpoint may be specified by *address*, as per **break**, or by the breakpoint number returned by **break** if it's prefixed with '#'.

dmesg [*count*]

Prints the contents of the kernel message buffer. The optional *count* argument will limit printing to at most the last *count* bytes of the message buffer.

dwatch *address*

Delete the watchpoint at *address* that was previously set with **watch** command.

examine [/*modifier*] *address* [,*count*]

Display the address locations according to the format in *modifier*. Multiple modifier formats display multiple locations. If *modifier* isn't specified, the modifier from the last use of **examine** is used.

The valid format characters for *modifier* are:

- b** examine bytes (8 bits).
- h** examine half-words (16 bits).
- l** examine words (legacy "long", 32 bits).
- L** examine long words (implementation dependent)
- a** print the location being examined.
- A** print the location with a line number if possible.
- x** display in unsigned hex.
- z** display in signed hex.
- o** display in unsigned octal.
- d** display in signed decimal.
- u** display in unsigned decimal.
- r** display in current radix, signed.
- c** display low 8 bits as a character. Non-printing characters as displayed as an octal escape code (e.g., '\000').
- s** display the NUL terminated string at the location. Non-printing characters are displayed as octal escapes.
- m** display in unsigned hex with a character dump at the end of each line. The location is displayed as hex at the beginning of each line.
- i** display as a machine instruction.
- I** display as a machine instruction, with possible alternative formats depending upon the machine:
 - alpha print register operands
 - m68k use Motorola syntax
 - pc532 print instruction bytes in hex
 - vax don't assume that each external label is a procedure entry mask

kill *pid* [,*signal_number*]

Send a signal to the process specified by the *pid*. Note that *pid* is interpreted using the current radix (see **trace/t** command for details). If *signal_number* isn't specified, the SIGTERM signal is sent.

match[/p]

A synonym for **next**.

next[/p]

Stop at the matching return instruction. If **/p** is specified, print the call nesting depth and the cumulative instruction count at each call or return. Otherwise, only print when the matching return is hit.

print[/axzodurc] address [address ...]

Print addresses *address* according to the modifier character, as per **examine**. Valid modifiers are: **/a**, **/x**, **/z**, **/o**, **/d**, **/u**, **/r**, and **/c** (as per **examine**). If no modifier is specified, the most recent one specified is used. *address* may be a string, and is printed “as-is”. For example:

```
print/x "eax = " $eax "\necx = " $ecx "\n"
```

will produce:

```
eax = xxxxxx
ecx = yyyyyy
```

ps[/a][/n][/w]

A synonym for **show all procs**.

reboot [flags]

Reboot, using the optionally supplied boot *flags*.

Note: Limitations of the command line interface preclude specification of a boot string.

search[/bhl] address value [mask] [,count]

Search memory from *address* for *value*. The unit size is specified with a modifier character, as per **examine**. Valid modifiers are: **/b**, **/h**, and **/l**. If no modifier is specified, **/l** is used.

This command might fail in interesting ways if it doesn't find *value*. This is because **ddb** doesn't always recover from touching bad memory. The optional *count* limits the search.

set \$variable [=] expression

Set the named variable or register to the value of *expression*. Valid variable names are described in **VARIABLES**.

show all procs[/a][/n][/w]

Display all process information. Valid modifiers:

/n show process information in a **ps(1)** style format (this is the default). Information printed includes: process ID, parent process ID, process group, UID, process status, process flags, process command name, and process wait channel message.

/a show the kernel virtual addresses of each process' proc structure, u-area, and vm space structure. The vm space address is also the address of the process' **vm_map** structure, and can be used in the **show map** command.

/w show each process' PID, command, system call emulation, wait channel address, and wait channel message.

show breaks

Display all breakpoints.

show buf[/f] address

Print the struct **buf** at *address*. The **/f** does nothing at this time.

show event [/f]
 Print all the non-zero event(9) event counters. If /f is specified, all event counters with a count of zero are printed as well.

show map [/f] *address*
 Print the vm_map at *address*. If /f is specified, the complete map is printed.

show ncache *address*
 Dump the namecache list associated with vnode at *address*.

show object [/f] *address*
 Print the vm_object at *address*. If /f is specified, the complete object is printed.

show page [/f] *address*
 Print the vm_page at *address*. If /f is specified, the complete page is printed.

show pool [/clp] *address*
 Print the pool at *address*. Valid modifiers:
 /c Print the cachelist and its statistics for this pool.
 /l Print the log entries for this pool.
 /p Print the pagelist for this pool.

show registers [/u]
 Display the register set. If /u is specified, display user registers instead of kernel registers or the currently save one.

 Warning: support for /u is machine dependent. If not supported, incorrect information will be displayed.

show uvmexp
 Print a selection of UVM counters and statistics.

show vnode [/f] *address*
 Print the vnode at *address*. If /f is specified, the complete vnode is printed.

show watches
 Display all watchpoints.

sifting [/F] *string*
 Search the symbol tables for all symbols of which *string* is a substring, and display them. If /F is specified, a character is displayed immediately after each symbol name indicating the type of symbol.

 For a.out(5)-format symbol tables, absolute symbols display @, text segment symbols display *, data segment symbols display +, BSS segment symbols display -, and filename symbols display /. For ELF-format symbol tables, object symbols display +, function symbols display *, section symbols display , and file symbols display /.

 To sift for a string beginning with a number, escape the first character with a backslash as:

 sifting \386

step [/p] [,count]
 Single-step *count* times. If /p is specified, print each instruction at each step. Otherwise, only print the last instruction.

 Warning: depending on the machine type, it may not be possible to single-step through some low-level code paths or user-space code. On machines with software-emulated single-stepping (e.g., pmax), stepping through code executed by interrupt handlers will probably do the wrong thing.

sync Force a crash dump, and then reboot.

trace [/u[1]] [*frame-address*][,*count*]

Stack trace from *frame-address*. If /u is specified, trace user-space, otherwise trace kernel-space. *count* is the number of frames to be traced. If *count* is omitted, all frames are printed. If /1 is specified, the trace is printed and also stored in the kernel message buffer.

Warning: user-space stack trace is valid only if the machine dependent code supports it.

trace/t[1] [*pid*][,*count*]

Stack trace by “thread” (process, on NetBSD) rather than by stack frame address. Note that *pid* is interpreted using the current radix, whilst **ps** displays pids in decimal; prefix *pid* with '0t' to force it to be interpreted as decimal (see **VARIABLES** section for radix). If /1 is specified, the trace is printed and also stored in the kernel message buffer.

Warning: trace by pid is valid only if the machine dependent code supports it.

until [/p]

Stop at the next call or return instruction. If /p is specified, print the call nesting depth and the cumulative instruction count at each call or return. Otherwise, only print when the matching return is hit.

watch *address*[,*size*]

Set a watchpoint for a region. Execution stops when an attempt to modify the region occurs. *size* defaults to 4.

If you specify a wrong space address, the request is rejected with an error message.

Warning: attempts to watch wired kernel memory may cause an unrecoverable error in some systems such as i386. Watchpoints on user addresses work the best.

write [/bhl] *address expression* [*expression* ...]

Write the *expressions* at succeeding locations. The unit size is specified with a modifier character, as per **examine**. Valid modifiers are: /b, /h, and /l. If no modifier is specified, /l is used.

Warning: since there is no delimiter between *expressions*, strange things may occur. It's best to enclose each *expression* in parentheses.

x [/modifier] *address*[,*count*]

A synonym for **examine**.

MACHINE-SPECIFIC COMMANDS

The "glue" code that hooks **ddb** into the NetBSD kernel for any given port can also add machine specific commands to the **ddb** command parser. All of these commands are preceded by the command word *machine* to indicate that they are part of the machine-specific command set (e.g. **machine reboot**). Some of these commands are:

ALPHA

halt Call the PROM monitor to halt the CPU.
reboot Call the PROM monitor to reboot the CPU.

ARM32

vmstat Equivalent to vmstat(1) output with "-s" option (statistics).
vnoded Print out a description of a vnode.

intrchain Print the list of IRQ handlers.
panic Print the current "panic" string.
frame Given a trap frame address, print out the trap frame.

MIPS

kvtop Print the physical address for a given kernel virtual address.
tlb Print out the Translation Lookaside Buffer (TLB). Only works in NetBSD kernels compiled with DEBUG option.

SH3

tlb Print TLB entries
cache Print cache entries
frame Print switch frame and trap frames.
stack Print kernel stack usage. Only works in NetBSD kernels compiled with the KSTACK_DEBUG option.

SPARC

prom Exit to the Sun PROM monitor.

SPARC64

buf Print buffer information.
ctx Print process context information.
dtlb Print data translation look-aside buffer context information.
dtlb Display data translation storage buffer information.
kmap Display information about the listed mapping in the kernel pmap. Use the "f" modifier to get a full listing.
pcb Display information about the "struct pcb" listed.
pctx Attempt to change process context.
page Display the pointer to the "struct vm_page" for this physical address.
phys Display physical memory.
pmap Display the pmap. Use the "f" modifier to get a fuller listing.
proc Display some information about the process pointed to, or curproc.
prom Enter the OFW PROM.
pv Display the "struct pv_entry" pointed to.
stack Dump the window stack. Use the "u" modifier to get userland information.
tf Display full trap frame state. This is most useful for inclusion with bug reports.
ts Display trap state.
traptrace Display or set trap trace information. Use the "r" and "f" modifiers to get reversed and full information, respectively.
uvmdump Dumps the UVM histories.
watch Set or clear a physical or virtual hardware watchpoint. Pass the address to be watched, or "0" to clear the watchpoint. Append "p" to the watch point to use the physical watchpoint registers.
window Print register window information about given address.

SUN3 and SUN3X

abort Drop into monitor via abort (allows continue).
halt Exit to Sun PROM monitor as in halt(8).
reboot Reboot the machine as in reboot(8).

pgmap Given an address, print the address, segment map, page map, and Page Table Entry (PTE).

VARIABLES

ddb accesses registers and variables as *\$name*. Register names are as per the **show registers** command. Some variables are suffixed with numbers, and may have a modifier following a colon immediately after the variable name. For example, register variables may have a 'u' modifier to indicate user register (e.g., *\$eax:u*).

Built-in variables currently supported are:

| | |
|--------------------|--|
| <i>lines</i> | The number of lines. This is used by the more feature. |
| <i>maxoff</i> | Addresses are printed as 'symbol'+offset unless offset is greater than <i>maxoff</i> . |
| <i>maxwidth</i> | The width of the displayed line. |
| <i>onpanic</i> | If non-zero (the default), ddb will be invoked when the kernel panics. If the kernel configuration option options DDB_ONPANIC=0 is used, <i>onpanic</i> will be initialized to off. |
| <i>fromconsole</i> | If non-zero (the default), the kernel allows to enter ddb from the console (by break signal or special key sequence). If the kernel configuration option options DDB_FROMCONSOLE=0 is used, <i>fromconsole</i> will be initialized to off. |
| <i>radix</i> | Input and output radix. |
| <i>tabstops</i> | Tab stop width. |

All built-in variables are accessible via `sysctl(3)`.

EXPRESSIONS

Almost all expression operators in C are supported, except '^', '~', and unary '-'. Special rules in **ddb** are:

| | |
|-------------------|---|
| <i>identifier</i> | name of a symbol. It is translated to the address (or value) of it. '.' and ':' can be used in the identifier. If supported by an object format dependent routine, [<i>filename</i>]: <i>function</i> [: <i>linenumber</i>], [<i>filename</i>]: <i>variable</i> , and [<i>filename</i>]: <i>linenumber</i> , can be accepted as a symbol. The symbol may be prefixed with <i>symbol_table_name::</i> (e.g., <i>emulator::mach_msg_trap</i>) to specify other than kernel symbols. |
| <i>number</i> | number. Radix is determined by the first two characters: '0x' - hex, '0o' - octal, '0t' - decimal, otherwise follow current radix. |
| . | <i>dot</i> |
| + | <i>next</i> |
| .. | address of the start of the last line examined. Unlike <i>dot</i> or <i>next</i> , this is only changed by the examine or write commands. |
| " | last address explicitly specified. |
| <i>\$name</i> | register name or variable. It is translated to the value of it. It may be followed by a ':' and modifiers as described above. |
| a | multiple of right-hand side. |
| <i>*expr</i> | expression indirection. It may be followed by a ':' and modifiers as described above. |

SEE ALSO

options(4), sysctl(8)

HISTORY

The **ddb** kernel debugger was written as part of the MACH project at Carnegie-Mellon University.

NAME

gdb — external kernel debugger

SYNOPSIS

```
makeoptions DEBUG=-g
options DDB
```

DESCRIPTION

The **gdb** kernel debugger is a variation of **gdb(1)** which understands some aspects of the FreeBSD kernel environment. It can be used in a number of ways:

- It can be used to examine the memory of the processor on which it runs.
- It can be used to analyse a processor dump after a panic.
- It can be used to debug another system interactively via a serial or fi rewire link. In this mode, the processor can be stopped and single stepped.
- With a fi rewire link, it can be used to examine the memory of a remote system without the participation of that system. In this mode, the processor cannot be stopped and single stepped, but it can be of use when the remote system has crashed and is no longer responding.

When used for remote debugging, **gdb** requires the presence of the **ddb(4)** kernel debugger. Commands exist to switch between **gdb** and **ddb(4)**.

PREPARING FOR DEBUGGING

When debugging kernels, it is practically essential to have built a kernel with debugging symbols (**makeoptions** **DEBUG=-g**). It is easiest to perform operations from the kernel build directory, by default `/usr/obj/usr/src/sys/GENERIC`.

First, ensure you have a copy of the debug macros in the directory:

```
make gdbinit
```

This command performs some transformations on the macros installed in `/usr/src/tools/debugscripts` to adapt them to the local environment.

Inspecting the environment of the local machine

To look at and change the contents of the memory of the system you are running on,

```
gdb -k -wcore kernel.debug /dev/mem
```

In this mode, you need the **-k** flag to indicate to **gdb(1)** that the “dump file” `/dev/mem` is a kernel data file. You can look at live data, and if you include the **-wcore** option, you can change it at your peril. The system does not stop (obviously), so a number of things will not work. You can set breakpoints, but you cannot “continue” execution, so they will not work.

Debugging a crash dump

By default, crash dumps are stored in the directory `/var/crash`. Investigate them from the kernel build directory with:

```
gdb -k kernel.debug /var/crash/vmcore.29
```

In this mode, the system is obviously stopped, so you can only look at it.

Debugging a live system with a remote link

In the following discussion, the term ‘local system’ refers to the system running the debugger, and ‘remote system’ refers to the live system being debugged.

To debug a live system with a remote link, the kernel must be compiled with the option **options DDB**. The option **options BREAK_TO_DEBUGGER** enables the debugging machine stop the debugged machine once a connection has been established by pressing ‘^C’.

Debugging a live system with a remote serial link

When using a serial port for the remote link on the i386 platform, the serial port must be identified by setting the flag bit 0x80 for the specified interface. Generally, this port will also be used as a serial console (flag bit 0x10), so the entry in `/boot/device.hints` should be:

```
hint.sio.0.flags="0x90"
```

Debugging a live system with a remote firewire link

As with serial debugging, to debug a live system with a firewire link, the kernel must be compiled with the option **options DDB**.

A number of steps must be performed to set up a firewire link:

- Ensure that both systems have firewire(4) support, and that the kernel of the remote system includes the dcons(4) and dcons_crom(4) drivers. If they are not compiled into the kernel, load the KLDs:

```
kldload firewire
```

On the remote system only:

```
kldload dcons
kldload dcons_crom
```

You should see something like this in the `dmesg(8)` output of the remote system:

```
fwohci0: BUS reset
fwohci0: node_id=0x8800ffc0, gen=2, non CYCLEMASTER mode
firewire0: 2 nodes, maxhop <= 1, cable IRM = 1
firewire0: bus manager 1
firewire0: New S400 device ID:00c04f3226e88061
dcons_crom0: <dcons configuration ROM> on firewire0
dcons_crom0: bus_addr 0x22a000
```

It is a good idea to load these modules at boot time with the following entry in `/boot/loader.conf`:

```
dcons_crom_enable="YES"
```

This ensures that all three modules are loaded. There is no harm in loading `dcons(4)` and `dcons_crom(4)` on the local system, but if you only want to load the `firewire(4)` module, include the following in `/boot/loader.conf`:

```
firewire_enable="YES"
```

- Next, use `fwcontrol(8)` to find the firewire node corresponding to the remote machine. On the local machine you might see:

```
# fwcontrol
2 devices (info_len=2)
node      EUI64      status
  1  0x00c04f3226e88061    0
  0  0x000199000003622b    1
```


The first node is always the local system, so in this case, node 0 is the remote system. If there are more than two systems, check from the other end to find which node corresponds to the remote system. On the remote machine, it looks like this:

```
# fwcontrol
2 devices (info_len=2)
node      EUI64      status
  0  0x000199000003622b  0
  1  0x00c04f3226e88061  1
```

- Next, establish a firewire connection with `dconschat(8)`:

```
dconschat -br -G 5556 -t 0x000199000003622b
```

`0x000199000003622b` is the EUI64 address of the remote node, as determined from the output of `fwcontrol(8)` above. When started in this manner, `dconschat(8)` establishes a local tunnel connection from port `localhost:5556` to the remote debugger. You can also establish a console port connection with the `-C` option to the same invocation `dconschat(8)`. See the `dconschat(8)` manpage for further details.

The `dconschat(8)` utility does not return control to the user. It displays error messages and console output for the remote system, so it is a good idea to start it in its own window.

- Finally, establish connection:

```
# gdb kernel.debug
GNU gdb 5.2.1 (FreeBSD)
(political statements omitted)
Ready to go. Enter 'tr' to connect to the remote target
with /dev/cuad0, 'tr /dev/cuad1' to connect to a different port
or 'trf portno' to connect to the remote target with the firewire
interface. portno defaults to 5556.
```

Type `'getsyms'` after connection to load kld symbols.

If you are debugging a local system, you can use `'kldsyms'` instead to load the kld symbols. That is a less obnoxious interface.

```
(gdb) trf
0xc21bd378 in ?? ()
```

The `trf` macro assumes a connection on port 5556. If you want to use a different port (by changing the invocation of `dconschat(8)` above), use the `tr` macro instead. For example, if you want to use port 4711, run `dconschat(8)` like this:

```
dconschat -br -G 4711 -t 0x000199000003622b
```

Then establish connection with:

```
(gdb) tr localhost:4711
0xc21bd378 in ?? ()
```

Non-cooperative debugging a live system with a remote firewire link

In addition to the conventional debugging via firewire described in the previous section, it is possible to debug a remote system without its cooperation, once an initial connection has been established. This corresponds to debugging a local machine using `/dev/mem`. It can be very useful if a system crashes and the debugger no longer responds. To use this method, set the `sysctl(8)` variables `hw.firewire.fwmem.eui64_hi` and `hw.firewire.fwmem.eui64_lo` to the upper and lower halves of the EUI64 ID of the remote system, respec-

tively. From the previous example, the remote machine shows:

```
# fwcontrol
2 devices (info_len=2)
node      EUI64      status
  0  0x000199000003622b  0
  1  0x00c04f3226e88061  1
```

Enter:

```
# sysctl -w hw.firewire.fwmem.eui64_hi=0x00019900
hw.firewire.fwmem.eui64_hi: 0 -> 104704
# sysctl -w hw.firewire.fwmem.eui64_lo=0x0003622b
hw.firewire.fwmem.eui64_lo: 0 -> 221739
```

Note that the variables must be explicitly stated in hexadecimal. After this, you can examine the remote machine's state with the following input:

```
# gdb -k kernel.debug /dev/fwmem0.0
GNU gdb 5.2.1 (FreeBSD)
(messages omitted)
Reading symbols from /boot/kernel/dcons.ko...done.
Loaded symbols for /boot/kernel/dcons.ko
Reading symbols from /boot/kernel/dcons_crom.ko...done.
Loaded symbols for /boot/kernel/dcons_crom.ko
#0  sched_switch (td=0xc0922fe0) at /usr/src/sys/kern/sched_4bsd.c:621
0xc21bd378 in ?? ()
```

In this case, it is not necessary to load the symbols explicitly. The remote system continues to run.

COMMANDS

The user interface to **gdb** is via **gdb(1)**, so **gdb(1)** commands also work. This section discusses only the extensions for kernel debugging that get installed in the kernel build directory.

Debugging environment

The following macros manipulate the debugging environment:

ddb Switch back to **ddb(4)**. This command is only meaningful when performing remote debugging.

getsyms

Display **kldstat** information for the target machine and invite user to paste it back in. This is required because **gdb** does not allow data to be passed to shell scripts. It is necessary for remote debugging and crash dumps; for local memory debugging use **kldsyms** instead.

kldsyms

Read in the symbol tables for the debugging machine. This does not work for remote debugging and crash dumps; use **getsyms** instead.

tr interface

Debug a remote system via the specified serial or firewire interface.

tr0 Debug a remote system via serial interface `/dev/cuad0`.

tr1 Debug a remote system via serial interface `/dev/cuad1`.

trf Debug a remote system via firewire interface at default port 5556.

The commands **tr0**, **tr1** and **trf** are convenience commands which invoke **tr**.

The current process environment

The following macros are convenience functions intended to make things easier than the standard `gdb(1)` commands.

| | |
|------------|--|
| f0 | Select stack frame 0 and show assembler-level details. |
| f1 | Select stack frame 1 and show assembler-level details. |
| f2 | Select stack frame 2 and show assembler-level details. |
| f3 | Select stack frame 3 and show assembler-level details. |
| f4 | Select stack frame 4 and show assembler-level details. |
| f5 | Select stack frame 5 and show assembler-level details. |
| xb | Show 12 words in hex, starting at current <i>ebp</i> value. |
| xi | List the next 10 instructions from the current <i>eip</i> value. |
| xp | Show the register contents and the first four parameters of the current stack frame. |
| xp0 | Show the first parameter of current stack frame in various formats. |
| xp1 | Show the second parameter of current stack frame in various formats. |
| xp2 | Show the third parameter of current stack frame in various formats. |
| xp3 | Show the fourth parameter of current stack frame in various formats. |
| xp4 | Show the fifth parameter of current stack frame in various formats. |
| xs | Show the last 12 words on stack in hexadecimal. |
| xxp | Show the register contents and the first ten parameters. |
| z | Single step 1 instruction (over calls) and show next instruction. |
| zs | Single step 1 instruction (through calls) and show next instruction. |

Examining other processes

The following macros access other processes. The **gdb** debugger does not understand the concept of multiple processes, so they effectively bypass the entire **gdb** environment.

| | |
|---------------------------|---|
| btp <i>pid</i> | Show a backtrace for the process <i>pid</i> . |
| btpa | Show backtraces for all processes in the system. |
| btp | Show a backtrace for the process previously selected with defproc . |
| btr <i>ebp</i> | Show a backtrace from the <i>ebp</i> address specified. |
| defproc <i>pid</i> | Specify the PID of the process for some other commands in this section. |
| fr <i>frame</i> | Show frame <i>frame</i> of the stack of the process previously selected with defproc . |

pcb *proc*

Show some PCB contents of the process *proc*.

Examining data structures

You can use standard `gdb(1)` commands to look at most data structures. The macros in this section are convenience functions which typically display the data in a more readable format, or which omit less interesting parts of the structure.

- bp** Show information about the buffer header pointed to by the variable *bp* in the current frame.
- bpd** Show the contents (*char **) of *bp->data* in the current frame.
- bp1** Show detailed information about the buffer header (*struct bp*) pointed at by the local variable *bp*.
- bpp *bp*** Show summary information about the buffer header (*struct bp*) pointed at by the parameter *bp*.
- bx** Print a number of fields from the buffer header pointed at in by the pointer *bp* in the current environment.
- vdev** Show some information of the *vnode* pointed to by the local variable *vp*.

Miscellaneous macros**checkmem**

Check unallocated memory for modifications. This assumes that the kernel has been compiled with **options DIAGNOSTIC**. This causes the contents of free memory to be set to 0xdead0de.

- dmesg** Print the system message buffer. This corresponds to the `dmesg(8)` utility. This macro used to be called **msgbuf**. It can take a very long time over a serial line, and it is even slower via firewire or local memory due to inefficiencies in **gdb**. When debugging a crash dump or over firewire, it is not necessary to start **gdb** to access the message buffer: instead, use an appropriate variation of

```
dmesg -M /var/crash/vmcore.0 -N kernel.debug
dmesg -M /dev/fwmem0.0 -N kernel.debug
```

kldstat

Equivalent of the `kldstat(8)` utility without options.

- pname** Print the command name of the current process.

- ps** Show process status. This corresponds in concept, but not in appearance, to the `ps(1)` utility. When debugging a crash dump or over firewire, it is not necessary to start **gdb** to display the `ps(1)` output: instead, use an appropriate variation of

```
ps -M /var/crash/vmcore.0 -N kernel.debug
ps -M /dev/fwmem0.0 -N kernel.debug
```

- y** Kludge for writing macros. When writing macros, it is convenient to paste them back into the **gdb** window. Unfortunately, if the macro is already defined, **gdb** insists on asking

```
Redefine foo?
```

It will not give up until you answer 'y'. This command is that answer. It does nothing else except to print a warning message to remind you to remove it again.

SEE ALSO

`gdb(1)`, `ps(1)`, `ddb(4)`, `firewire(4)`, `dconschat(8)`, `dmesg(8)`, `fwcontrol(8)`, `kldload(8)`

AUTHORS

This man page was written by Greg Lehey <grog@FreeBSD.org>.

BUGS

The `gdb(1)` debugger was never designed to debug kernels, and it is not a very good match. Many problems exist.

The **gdb** implementation is very inefficient, and many operations are slow.

Serial debugging is even slower, and race conditions can make it difficult to run the link at more than 9600 bps. Firewire connections do not have this problem.

The debugging macros “just grewed”. In general, the person who wrote them did so while looking for a specific problem, so they may not be general enough, and they may behave badly when used in ways for which they were not intended, even if those ways make sense.

Many of these commands only work on the ia32 architecture.

NAME

vinumdebug — debug macros for vinum(4)

DESCRIPTION

This man page describes `gdb(4)` macros for debugging the `vinum(4)` kernel module. See `gdb(4)` for the description of the kernel debugging environment. No further action is required to access the `vinum(4)` debug macros. They are loaded automatically along with the other macros.

COMMANDS

finfo Show recently freed `vinum(4)` memory blocks.

meminfo Equivalent of the **vinum info -v** command.

rq Show information about the request pointed to by the variable `rq` in the current frame.

rqe Show information about the request element pointed to by the variable `rqe` in the current frame.

rqi Print out a simplified version of the same information as the **vinum info -V** command.

rqinfo Show the `vinum(4)` request log buffer like the **vinum info -V** command.

rqq rq Show information about the request (`struct rq`) pointed at by `rq`.

rqq0 Print information on some `vinum(4)` request structures.

rqq1 Print information on some `vinum(4)` request structures.

rrqe rqe Show information about the request element (`struct rqe`) pointed at by the parameter `rqe`.

AUTHORS

This man page was written by Greg Lehey (grog@FreeBSD.org).

SEE ALSO

`gdb(4)`, `vinum(4)`, `vinum(8)`