Debugging Kernel Problems

by Greg Lehey

Edition for AsiaBSDCon 2004
Taipei, 13 March 2004

Debugging Kernel Problems

by Greg Lehey (`grog@FreeBSD.org`)

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. The BSD kernels are no exception. Compared to most other operating systems, both free and commercial, the 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.

In this tutorial, 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.
- Demonstration: panicing and dumping a system.
- 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 tools

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
...
cd1 at ahc0 bus 0 target 1 lun 0
cd1: <TEAC CD-ROM CD-532S 1.0A> Removable CD-ROM SCSI-2 device
cd1: 20.000MB/s transfers (20.000MHz, offset 15)
cd1: 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:

```
sio1: 1 more silo overflow (total 1607)
```

```
sio1: 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 *)msgbufp->msg_ptr
```

For NetBSD or OpenBSD, enter:

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

Most people use *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:

```
$ 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.

```
 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).

```
    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.

```
    0     7     0   0 171  0     0   12 -      RL    ??   80:46.00  (pagezero)
```

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

```
    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 comment says it all.

```
    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.

```
    0    12     0   0 -44  0     0   12 -      WL    ??   39:11.32  (swi1: 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 FreeB-SD release 5.

```
    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 never quite match 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 reactiveness of a system with high system load and a system with high interrupt load: since load-balancing doesn't work for interrupts, 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: rl0
   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:

```
 $ 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 24  0  271 1471  94 36 45 20
  1 1 0  18488  22292 2654   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. The 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 page faults, 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  19  0  6  1 74
    0  222 56.00   1 0.05   0.00   0 0.00   0.00   0 0.00  69  0 29  2  0
    0   75  0.00   0 0.00   0.00   0 0.00   0.00   0 0.00  81  0 19  0  0
    0   76 32.00   1 0.03   0.00   0 0.00   0.00   0 0.00  84  0 16  0  0
    0   74  0.00   0 0.00   0.00   0 0.00   0.00   0 0.00  90  0  7  3  0
    0   74  0.00   0 0.00   0.00   0 0.00   0.00   0 0.00  95  0  5  0  0
    0   74  5.30  20 0.10   0.00   0 0.00   0.00   0 0.00  40  0 31  0 29
    0   73  6.40  51 0.32   0.00   0 0.00   0.00   0 0.00  12  0 10  3 75
    0   75  5.55  49 0.27   0.00   0 0.00   0.00   0 0.00  24  0 12  3 61
    0   73  4.91  54 0.26   0.00   0 0.00   0.00   0 0.00  21  0  9  1 69
    0   75  6.91  54 0.36   0.00   0 0.00   0.00   0 0.00  39  0  7  3 51
    0   72  9.80  49 0.46   0.00   0 0.00   0.00   0 0.00  31  0  6  4 59
    0   76 17.94  36 0.63   0.00   0 0.00   0.00   0 0.00  34  0 12  0 54
    0   75 19.20   5 0.09   0.00   0 0.00   0.00   0 0.00  93  0  5  1  1
    0   74 37.33   3 0.11   0.00   0 0.00   0.00   0 0.00  93  0  6  1  0
    0   75 56.00   1 0.06   0.00   0 0.00   0.00   0 0.00  82  0 17  1  0
    0   73  0.00   0 0.00   0.00   0 0.00   0.00   0 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

```
                    /0   /1   /2   /3   /4   /5   /6   /7   /8   /9   /10
     Load Average   ||

          /0   /10  /20  /30  /40  /50  /60  /70  /80  /90  /100
cpu  user|XXXXXXXXXXXXXXXXXXXXXX
     nice|
   system|XXXXX
interrupt|
     idle|XXXXXXXXXXXXXXXXXXXXX

          /0   /10  /20  /30  /40  /50  /60  /70  /80  /90  /100
ad0   MB/sXXXX
      tps|XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

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 count
All  252828    4808  3565340    15372         pages
                                                          zfod   Interrupts
Proc:r  p  d  s  w    Csw  Trp  Sys  Int  Sof  Flt        cow   62295 total
     2     1 24       147   14  63262294   26    6  56060 wire      1 ata0 irq14
                                                   162880 act         ata1 irq15
 1.5%Sys  98.5%Intr  0.0%User  0.0%Nice  0.0%Idl    24140 inact       ahc0 irq11
|    |    |    |    |    |    |    |    |    |       9748 cache    27 mux irq10
=++++++++++++++++++++++++++++++++++++++++++++++++     656 free      4 atkbd0 irq
                                                          daefr       psm0 irq12
Namei         Name-cache    Dir-cache                     prcfr    77 sio1 irq3
    Calls     hits    %     hits    %                     react       ppc0 irq7
                                                          pdwak    99 clk irq0
                                                          pdpgs   128 rtc irq8
Disks   ad0   ad2   cd0   cd1   sa0 pass0 pass1           intrn 61959 lpt0 irq7
KB/t   8.00  0.00  0.00  0.00  0.00  0.00  0.00     35712 buf
tps       1     0     0     0     0     0     0        27 dirtybuf
MB/s   0.01  0.00  0.00  0.00  0.00  0.00  0.00     17462 desiredvnodes
% busy    0     0     0     0     0     0     0     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
```

```
         real   virtual    free            in  out   in  out       132 total
Active   9868     14100    6364     ops     1                      100 irq0
All     21140     25372  658588     pages                           14 irq9
                                                                    18 irq10
Proc:r  d  s  w       Csw  Trp  Sys  Int  Sof  Flt        forks
     2  1  5           40   27  193  133   20    8        fkppw
                                                          fksvm
  95.9% Sy   1.4% Us   0.0% Ni   0.0% In   2.7% Id        pwait
|    |    |    |    |    |    |    |    |    |    |     6 relck
===============================================>       6 rlkok
                                                          noram
Namei         Sys-cache     Proc-cache                    ndcpy
    Calls     hits    %     hits     %                    fltcp
     1043      806   77       34     3                  1 zfod
                                                          cow
Discs  fd0  sd0  md0                                   64 fmin
seeks                                                  85 ftarg
xfers        14                                      1372 itarg
Kbyte       164                                       941 wired
%busy      21.2                                           pdfre
                                                          pdscn
```

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
                                                        2 objht     100 prof
   9.3% Sys  85.5% User   0.0% Nice   4.4% Idle        62 zfod
|    |    |    |    |    |    |    |    |    |    |   385 nzfod
=====>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>    16.14 %zfod
                                                          kern
Namei         Sys-cache     Proc-cache               5408 wire
    Calls     hits    %     hits     %              18312 act
      212      203   96        3     1              11220 inact
                                                    27016 free
Discs  sd0  rd0  rd1                                      daefr
seeks  411                                            372 prcfr
xfers  411                                             46 react
Kbyte   33                                                scan
  sec  0.1                                                hdrev
                                                          intrn
```

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.

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.

The resultant memory image looks like:

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.

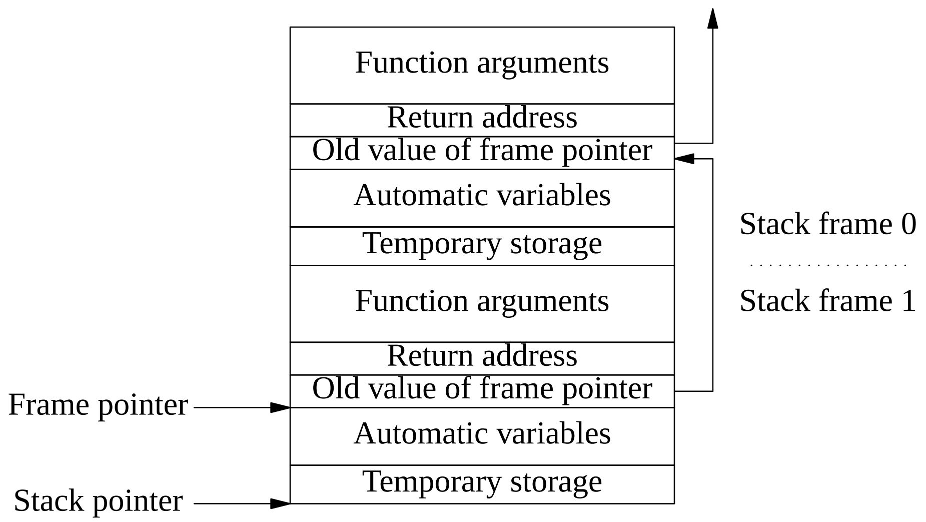


Figure 1: Function stack frame

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 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:

```
        pushl -4(%ebp)            value of x
        pushl -8(%ebp)            value of y
        call _foo                 call the function
        addl $8,%esp              and remove parameters
```

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:

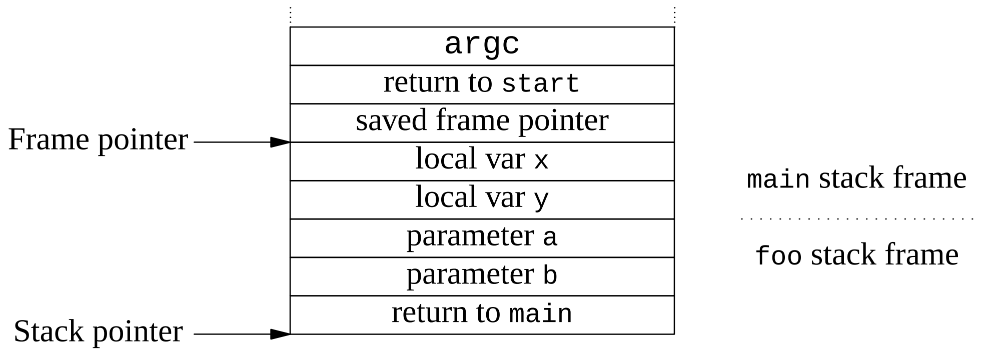


Figure 2: Stack frame after `call` instruction

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
```

Our stack is now complete.

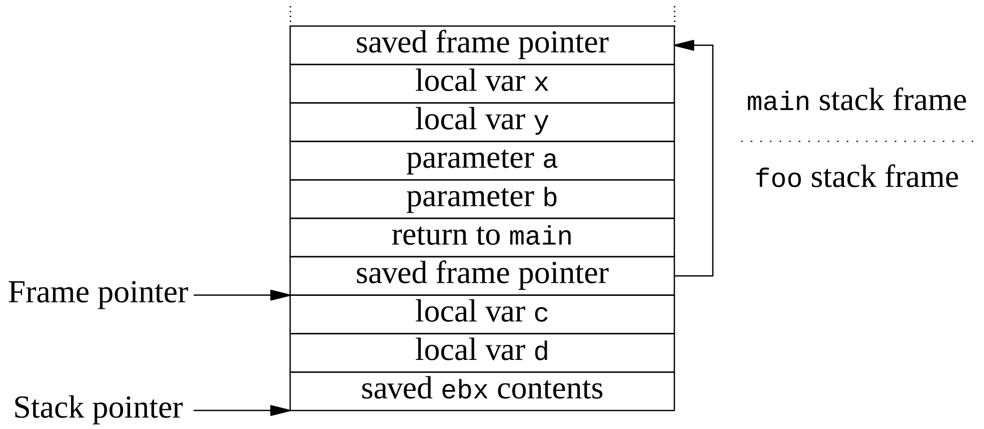


Figure 3: Complete stack frame after entering called function

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.

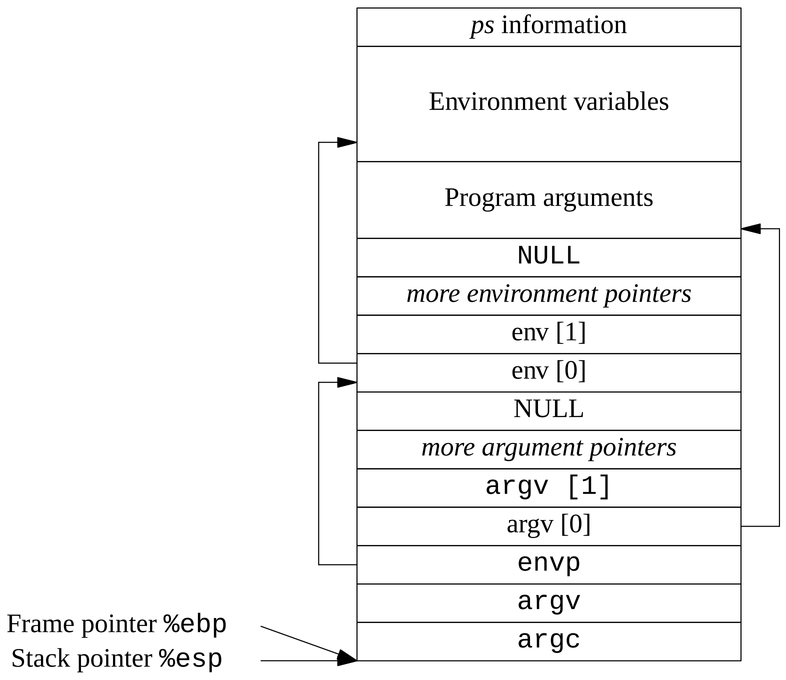
Next, the *ret* instruction pops the return address into the instruction pointer, causing the next instruction to be fetched from the address following the *call* instruction in the calling function.

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 []);
```

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 almost 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

One 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_dss1 (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_dss1 (isdnfd=4, s=0xefbfd504 "i") at msgutil.c:324
324           handle_charge (isdnfd, cp, i, &time_now, appl_no, channel);
(gdb) l                 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", controller = 0,
  isdntype = 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",
  controller = 0,
  isdntype = 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_typ == 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

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/5-CURRENT-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/5-CURRENT-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, */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 almost no code, and the kernel is almost identical in size. Providing symbols does eliminate the chance for some optimizations, so the code may not be identical, but the differences are very minor.

  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 builds a kernel */usr/obj/sys/ZAPHOD/kernel.debug* and a stripped copy at */usr/obj/sys/ZAPHOD/kernel*. It then installs */usr/obj/sys/ZAPHOD/kernel*.

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/5-CURRENT-ZAPHOD/src*, the kernel binaries will be placed in */usr/obj/src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/ZAPHOD*, and the debug kernel will be called */usr/obj/src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/ZAPHOD/kernel.debug*.

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.

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*. 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 ***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.

Alternative debugging interfaces are on the horizon. NetBSD supports debugging over Ethernet, but only on NE2000 cards. There's some code for FreeBSD for the Intel `fxp` driver, but it hasn't been committed yet. In addition, the FreeBSD firewire (IEEE 1349) driver supports remote debugging. We'll look at this below.

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**.

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 0xfffff, 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.

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 post-mortem dumps. In the latter 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 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_edi = 0xbfbfef34, tf_esi = 0xbfbfef64, tf_ebp = 0xbfbfe9cc,
    tf_isp = 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.

Debugging running systems

For some things, you don't need to stop the kernel. If you're only looking, for example, 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 /isr/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

Firewire offers new possibilities for remote debugging. On the one hand, it provides a much faster method of remote debugging, though the speed is still limited by the inefficiencies of *gdb* processing. On the other hand, it provides a completely new method to debug systems which have crashed or hung: firewire can access the memory of the remote machine without its intervention, which provides an interface similar to local memory debugging. This promises to be a way of debugging 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 separate ports for the console and debug connection.

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

- Ensure that both systems have firewire support, and that the kernel of the remote system includes the `dcons` and `dcons_crom` 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* 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* and *dcons_crom* on the local system, but if you only want to load the *firewire* module, include the following in */boot/loader.conf*:

```
firewire_enable="YES"
```

- Next, use *fwcontrol* 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*:

```
# dconschat -br -G 5556 -t 0x000199000003622b
```

`0x000199000003622b` 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.

The *dconschat* 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/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) 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* 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 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.

6

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 might 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/5-CURRENT-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. The 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 0xfffff, 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 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_edi = -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_edi = 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 0x280a1eee 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, abbreviated 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

Our Vinum test machine panics at boot time:

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

The first thing to do is to look at the back trace. In this case, however, we find 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.

Vinum includes a number of kernel debug tools, including some macros which keep track of memory allocation. One is `finfo`, which keeps track of recently freed memory areas. It's only enabled on request.

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 not in any block freed by Vinum, but it's just after the block at sequence number 2. That makes it a lot more suspect than if it were just before an allocated block: most addressing errors go off the end of the data block.

```
(gdb) **meminfo**
Block   Sequence          size    address         line          file
     0       3            3136  0xc1957000         140          vinum.c
     1       7             256  0xc1991d00         117          vinumio.c
     2       9             256  0xc1991c00         117          vinumio.c
     3      11             256  0xc1991b00         117          vinumio.c
     4      13             256  0xc1991a00         117          vinumio.c
     5      16             256  0xc1991900         117          vinumio.c
     6      18             256  0xc1991800         117          vinumio.c
     7      31            2048  0xc19b7000         902          vinumio.c
     8      32            1536  0xc19b6800         120          vinummemory.c
     9      33              16  0xc19885a0         770          vinumconfig.c
    10      34              16  0xc19885c0         770          vinumconfig.c
    11      35              16  0xc19885e0         770          vinumconfig.c
    12      36              16  0xc1988610         770          vinumconfig.c
    13      37              16  0xc1988620         770          vinumconfig.c
    14      38            3072  0xc1953000        1454          vinumconfig.c
    15      39              16  0xc18d93a0         770          vinumconfig.c
    16      40              16  0xc1988600         770          vinumconfig.c
    17      41            3072  0xc1952000        1454          vinumconfig.c
    18      42              16  0xc1988690         770          vinumconfig.c
    19      43            3072  0xc1951000        1454          vinumconfig.c
    20      44            3072  0xc1950000         120          vinummemory.c
    21      45            4288  0xc18a7000         120          vinummemory.c
    22      46              16  0xc1988640         770          vinumconfig.c
    23      47              16  0xc1988670         770          vinumconfig.c
    24      48              16  0xc19886a0         770          vinumconfig.c
    25      49              16  0xc19886f0         770          vinumconfig.c
    26      50              16  0xc19886b0         770          vinumconfig.c
    27      51              16  0xc1988710         770          vinumconfig.c
    28      52              16  0xc1988730         770          vinumconfig.c
    29      53              16  0xc1988750         770          vinumconfig.c
    30      54              16  0xc1988780         770          vinumconfig.c
    31      55              16  0xc19882d0         770          vinumconfig.c
    32      56              16  0xc19887d0         770          vinumconfig.c
    33      57              16  0xc19887a0         770          vinumconfig.c
    34      58              16  0xc1988800         770          vinumconfig.c
    35      59              16  0xc1988810         770          vinumconfig.c
    36      60              16  0xc19887e0         770          vinumconfig.c
    37      61              16  0xc1988840         770          vinumconfig.c
    38      62              16  0xc1988860         770          vinumconfig.c
    39      63              16  0xc18d9ab0         770          vinumconfig.c
    40      64              16  0xc18d9340         770          vinumconfig.c
    41      65              16  0xc18d9e40         770          vinumconfig.c
    42      66              16  0xc0b877d0         770          vinumconfig.c
    43      67              16  0xc18d99c0         770          vinumconfig.c
    44      68              16  0xc18d9b40         770          vinumconfig.c
    45      69              16  0xc19888c0         770          vinumconfig.c
    46      70              16  0xc19888e0         770          vinumconfig.c
    47      71              16  0xc18d9d00         770          vinumconfig.c
    48      72              16  0xc1817cf0         770          vinumconfig.c
    49      73              16  0xc18d9eb0         770          vinumconfig.c
    50      74              16  0xc19881c0         770          vinumconfig.c
    51      75              16  0xc18d9a30         770          vinumconfig.c
    52      76              16  0xc1988580         770          vinumconfig.c
    53      77              16  0xc1988560         770          vinumconfig.c
    54      78              16  0xc1988570         770          vinumconfig.c
    55      79              16  0xc18d9360         770          vinumconfig.c
    56      80               16  0xc1988500         770          vinumconfig.c
    57      81               16  0xc19884c0         770          vinumconfig.c
    58      82               16  0xc1988520         770          vinumconfig.c
    59      83               16  0xc19884e0         770          vinumconfig.c
    60      84               16  0xc19884b0         770          vinumconfig.c
    61      85               16  0xc19884d0         770          vinumconfig.c
    62      86               16  0xc19884a0         224          vinumdaemon.c
```

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

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, *vinumconfig.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 */
```

The `EXPAND` macro is effectively the same as `realloc`. It allocates `INITIAL_PLEXES * sizeof (struct plex)` more memory and copies the old data to it, then frees the old data; that's the `free` call we saw. If a pointer remains pointing into the old area, it's reasonable for it to go over the end. In this case, the issue is muddied because the memory area has apparently been reallocated in a length of 4096 bytes and then freed again; but this is our luck, because it means that the allocation routines will catch it.

Looking at the code, though, you'll see that the pointer to the plex is not allocated until after the call to `EXPAND`. So maybe it's 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.

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,
  state = 3735929054,
  length = 0xdeadc0dedeadc0de,
  flags = 0xdeadc0de,
  stripesize = 0xdeadc0de,
  sectorsize = 0xdeadc0de,
  subdisks = 0xdeadc0de,
  subdisks_allocated = 0xdeadc0de,
  sdnos = 0xdeadc0de,
  plexno = 0xdeadc0de,
  volno = 0xdeadc0de,
  volplexno = 0xdeadc0de,
  reads = 0xdeadc0dedeadc0de,
  writes = 0xdeadc0dedeadc0de,
  bytes_read = 0xdeadc0dedeadc0de,
  bytes_written = 0xdeadc0dedeadc0de,
  recovered_reads = 0xdeadc0dedeadc0de,
  degraded_writes = 0xdeadc0dedeadc0de,
  parityless_writes = 0xdeadc0dedeadc0de,
  multiblock = 0xdeadc0dedeadc0de,
  multistripe = 0xdeadc0dedeadc0de,
  sddowncount = 0xdeadc0de,
  usedlocks = 0xdeadc0de,
  lockwaits = 0xdeadc0de,
  checkblock = 0xdeadc0dedeadc0de,
  name = "ÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞ",
  lock = 0xdeadc0de,
  lockmtx = {
    mtx_object = {
      lo_class = 0xdeadc0de,
      lo_name = 0xdeadc0de <Address 0xdeadc0de out of bounds>,
      lo_type = 0xdeadc0de <Address 0xdeadc0de out of bounds>,
      lo_flags = 0xdeadc0de,
      lo_list = {
        tqe_next = 0xc1994068,
        tqe_prev = 0xdeadc0de
      },
      lo_witness = 0xdeadc0de
    },
    mtx_lock = 0xdeadc0de,
    mtx_recurse = 0xdeadc0de,
    mtx_blocked = {
      tqh_first = 0xdeadc0de,
      tqh_last = 0xdeadc0de
    },
    mtx_contested = {
      le_next = 0xdeadc0de,
      le_prev = 0xdeadc0de
    }
  },
  dev = 0xdeadc0de
}
```

That's inside the plex's lock mutex. Nothing touches mutexes except the mutex primitives, so this looks like somewhere a mutex constructor has been handed a stale pointer. That helps us narrow our search:

```
$ grep -n mtx *.c
vinumconfig.c:831:      mtx_destroy(&plex->lockmtx);
vinumconfig.c:1457:     mtx_init(&plex->lockmtx, plex->name, "plex", MTX_DEF);
vinumdaemon.c:74:    mtx_lock_spin(&sched_lock);
vinumdaemon.c:76:    mtx_unlock_spin(&sched_lock);
vinumlock.c:139:    mtx_lock(&plex->lockmtx);
vinumlock.c:143:        msleep(&plex->usedlocks, &plex->lockmtx, PRIBIO, "vlock", 0);
vinumlock.c:171:                    msleep(lock, &plex->lockmtx, PRIBIO, "vrlock", 0);
vinumlock.c:195:    mtx_unlock(&plex->lockmtx);
```

The calls in *vinumdaemon.c* are for `sched_lock`, so we can forget them. The others refer to 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 *vinumconfig.c*.

Looking at the code round 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 (isstriped(plex))
        mtx_destroy(&plex->lockmtx);
    destroy_dev(plex->dev);
    bzero(plex, sizeof(struct plex));                        /* and clear it out */
    plex->state = plex_unallocated;
}
```

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 `0xdeadc0de`, so it would try to free it and panic right there, since `0xdeadc0de` is not a valid address. So it can't be this one.

Line 1457 is in `config_plex`:

```
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));
    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.

Another panic

After fixing that, our Vinum test system panics again, this time 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
```

Hey, that's exactly the same panic as before. Maybe the bug didn't get fixed after all?

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/5-CURRENT-ZAPHOD/src/sys/GENERIC
# gdb
…
Debugger (msg=0x12 <Address 0x12 out of bounds>) at /src/FreeBSD/5-CURRENT-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.

```
Id Refs Address    Size     Name
 1    2 0xc0100000   59f5dc kernel
 2    1 0xc06a0000    c84cc vinum.ko
Select the list above with the mouse, paste into the screen
and then press ^D.  Yes, this is annoying.
 2    1 0xc06a0000    c84cc vinum.ko
add symbol table from file "/src/FreeBSD/obj/src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/GEN
ERIC/modules/src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/modules/vinum/vinum.ko.debug" at
        .text_addr = 0xc06a4920
        .data_addr = 0xc06b5000
        .bss_addr = 0xc06b5400
```

The output above comes from the FreeBSD debugging macros in */usr/src/tools/debugscripts*. Currently the only way to load the symbols is to use the mouse to copy and paste (or type in manually if you’re using a non-graphics terminal). The *gdb* startup calls a macro *asf* which calls the program *asf* to interpret the information about the *kld* s and produce a command file to load the correct symbol information, then loads it. This is what causes the subsequent output. The cut and paste is necessary because there’s no way to pass parameters from *gdb* to the shell script.

Traditionally, the first thing you do with a panic is to see where it happens. That’s less important with this bug, because it refers to a problem which has happened long before, but we’ll do it anyway:

```
(gdb) bt
#0  Debugger (msg=0x12 <Address 0x12 out of bounds>)
    at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/i386/i386/db_interface.c:330
#1  0xc031294b in panic (fmt=0x1 <Address 0x1 out of bounds>)
    at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/kern/kern_shutdown.c:527
#2  0xc0462137 in mtrash_ctor (mem=0xc1996000, size=0x20, arg=0x0)
    at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/vm/uma_dbg.c:138
#3  0xc04609ff in uma_zalloc_arg (zone=0xc0b65240, udata=0x0, flags=0x2)
    at /src/FreeBSD/5-CURRENT-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/5-CURRENT-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/5-CURRENT-ZAPHOD/src/sys/kern/vfs_bio.c:2606
#7  0xc0356732 in breadn (vp=0xc1a1936c, blkno=0x2000000012, size=0x12, rablkno=0x0,
rabsize=0x0, cnt=0x0, cred=0x0,
    bpp=0x12) at /src/FreeBSD/5-CURRENT-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/5-CURRENT-ZAPHOD/src/sys/kern/vfs_bio.c:683
#9  0xc043586f in ffs_blkatoff (vp=0xc1a1936c, offset=0x0, res=0x0, bpp=0xcccb3988)
    at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/ufs/ffs/ffs_subr.c:91
#10 0xc043f5a7 in ufs_lookup (ap=0xcccb3ab8) at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys
/ufs/ufs/ufs_lookup.c:266
#11 0xc0446dd8 in ufs_vnoperate (ap=0x0) at /src/FreeBSD/5-CURRENT-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/5-CURRENT-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/5-CURRENT-ZAPHOD/src/sys/ker
n/vfs_lookup.c:181
#16 0xc036ee32 in lstat (td=0xc199b980, uap=0xcccb3d10)
    at /src/FreeBSD/5-CURRENT-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 = 0xfffff
fff, tf_ecx = 0x2, tf_eax = 0xbe, tf_trapno = 0xc, tf_err = 0x2, tf_eip = 0x804ac0b,
tf_cs = 0x1f, tf_eflags = 0x282, tf_esp = 0xbfbffcbc, tf_ss = 0x2f})
    at /src/FreeBSD/5-CURRENT-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/5-CURRENT-ZAPHOD/src/sys/vm/uma_dbg.c:138
138                             panic("Most recently used by %s\n", (*ksp == NULL)?
(gdb) l                                        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
(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. In debug mode, Vinum maintains statistics about memory allocations and frees. Let's take a look at them with some of the the Vinum debug macros:

```
(gdb) meminfo                           look at currently allocated memory
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                     look at already freed memory
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, this block of memory was allocated to Vinum until time 19.601196. 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 *vinumconfig.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 μs earlier) an allocation of 3072 bytes, which is the replacement area.

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: `0xdeadc0de` became `0xdeafc0de`, so we're probably looking at a logical bit set operation which *or*s 0x20000 with the previous value.

But what's the value? It's part of the drive, but which part? The memory area is of type `struct drive []`, and it contains information for a number of drives. The first thing to do is to find which drive this error belongs to. We need to do a bit of arithmetic. First, find out how long a drive entry is. We can do that by comparing the address of the start of the area with the address of the second drive entry (`drive [1]`):

```
 (gdb) p &((struct drive *) 0xc1996000)[1]
 $2 = (struct drive *) 0xc1996100
```

So `struct drive` is exactly 256 bytes long. That means that our fault address `0xc199657c` is in plex 5 at offset `0x7c`. We can look at the entry like this:

```
(gdb) p ((struct drive *) 0xc1996000)[5]
$3 = {
  devicename = "ÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞ",
  label = {
    sysname = "ÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞ",
    name = "ÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞÞÀÞ",
    date_of_birth = {
      tv_sec = 0xdeadc0de,
      tv_usec = 0xdeadc0de
    },
    last_update = {
      tv_sec = 0xdeadc0de,
      tv_usec = 0xdeadc0de
    },
    drive_size = 0xdeadc0dedeadc0de
  },
  state = 3735929054,
  flags = 0xdeafc0de,
  subdisks_allocated = 0xdeadc0de,
  subdisks_used = 0xdeadc0de,
  blocksize = 0xdeadc0de,
  pid = 0xdeadc0de,
  sectors_available = 0xdeadc0dedeadc0de,
  secsperblock = 0xdeadc0de,
  lasterror = 0xdeadc0de,
  driveno = 0xdeadc0de,
  opencount = 0xdeadc0de,
  reads = 0xdeadc0dedeadc0de,
  writes = 0xdeadc0dedeadc0de,
  bytes_read = 0xdeadc0dedeadc0de,
  bytes_written = 0xdeadc0dedeadc0de,
  active = 0xdeadc0de,
  maxactive = 0xdeadc0de,
  freelist_size = 0xdeadc0de,
  freelist_entries = 0xdeadc0de,
  freelist = 0xdeadc0de,
  sectorsize = 0xdeadc0de,
  mediasize = 0xdeadc0dedeadc0de,
  dev = 0xdeadc0de,
  lockfilename = "ÞÀÞÞÀÞÞÀÞÞÀÞ",
  lockline = 0xdeadc0de
}
```

There's a problem here: some of the fields are not represented in hex. The device name is in text, so it looks completely different. We can't rely on finding our `0xdeafc0de` here, and looking at the output makes your eyes go funny. About the only alternative we have is something approximating to a binary search:

```
(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, that's the value, 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 then 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) l 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/5-CURRENT-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                                       find how we got here
Breakpoint 1, 0xc06a6010 in get_empty_drive () at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sy
s/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/5-CURRENT-ZAPHOD/src/sys/dev/vinu
m/vinumconfig.c:468
#1  0xc06a60f9 in find_drive (name=0xc199581a "virtual", create=0x1)
    at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/dev/vinum/vinumconfig.c:505
#2  0xc06a7217 in config_subdisk (update=0x1) at /src/FreeBSD/5-CURRENT-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/5-CURRENT-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/5-CURRENT-ZAPHOD/src/sys/dev/vinum/vinumio.c:942
#5  0xc06a4c65 in vinumattach (dummy=0x0) at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/dev
/vinum/vinum.c:176
#6  0xc06a4f6d in vinum_modevent (mod=0xc0b89f00, type=1792, unused=0x0)
    at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/dev/vinum/vinum.c:277
#7  0xc0308541 in module_register_init (arg=0xc06b5054) at /src/FreeBSD/5-CURRENT-ZAPH
OD/src/sys/kern/kern_module.c:107
#8  0xc02ed275 in mi_startup () at /src/FreeBSD/5-CURRENT-ZAPHOD/src/sys/kern/init_mai
n.c:214
```

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

```
int
find_drive(const char *name, int create)
{
    int driveno;
    struct drive *drive;

    if (name != NULL) {
        for (driveno = 0; driveno < vinum_conf.drives_allocated; driveno++) {
            drive = &DRIVE[driveno];               /* point to drive */
            if ((drive->label.name[0] != ' ')     /* it has a name */
            &&(strcmp(drive->label.name, name) == 0)  /* and it's this one */
            &&(drive->state > drive_unallocated))  /* and it's a real one: found */
                return driveno;
        }
    }
    /* the drive isn't in the list.  Add it if he wants */
    if (create == 0)                                /* don't want to create */
        return -1;                                  /* give up */

    driveno = get_empty_drive();
    drive = &DRIVE[driveno];
    if (name != NULL)
        strlcpy(drive->label.name,                  /* put in its name */
            name,
            sizeof(drive->label.name));
    drive->state = drive_referenced;                /* in use, nothing worthwhile */
    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:

```
         case kw_drive:
            sd->driveno = find_drive(token[++parameter], 1); /* insert info */
            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/5-CURRENT-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
```

7 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 0xfffff, 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, rablkno=0x0,
    rabsize=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_isp = 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
#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_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
#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_edi = 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 associated 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 narg;
       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 &narg
$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 `narg`, `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 0xdd363cc0
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 `int`s (easy), two `struct` pointers (not much more difficult) and two `struct`s, 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,0xfffffff4(%ebp)
0xc02ef069 <setitimer+9>:       mov    %esi,0xfffffff8(%ebp)
```

```
0xc02ef06c <setitimer+12>:      mov    %edi,0xfffffffc(%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 li 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    0xffffffd0(%ebp),%eax
0xc045fc24 <syscall+644>:       mov    %eax,0x4(%esp,1)
0xc045fc28 <syscall+648>:       mov    0xffffffc4(%ebp),%edx
0xc045fc2b <syscall+651>:       call   *0x4(%edx)
```

This code is confusing because some instructions us `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
                                         esp   0xc55b58f0      0xdd363d10
0xdd363cf0:     0xc04de816      0x00000409      0x00000003      0x00009a8d
0xdd363d00:     0x00000202      0xc05134f8      0xbfbfe624      0x00000053
0xdd363d10:     0x00000000      0x00000000      0x00000000      0x00009a8d
0xdd363d20:     0x00000000      0xc55ba9a0      0xc1619500      0x00000001
0xdd363d30:     0x0fffffff      0x00000000      0x0001869d      0x0827aec0
0xdd363d40:
         ebp   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    0xffffffd0(%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    0xffffffc4(%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 aitv;
        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, &aitv, sizeof(struct itimerval))))
                return (error);

        mtx_lock(&Giant);

        if ((uap->itv = uap->oitv) &&
            (error = getitimer(td, (struct getitimer_args *)uap))) {
                goto done2;
        }
        if (itvp == 0) {
                error = 0;
                goto done2;
        }
        if (itimerfix(&aitv.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,0xfffffff4(%ebp)      save ebx
0xc02ef069 <setitimer+9>:       mov    %esi,0xfffffff8(%ebp)      save esi
0xc02ef06c <setitimer+12>:      mov    %edi,0xfffffffc(%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_isp = 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 = 0xdd363d10, tf_ebp =
    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.

8

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. In this section we'll look at some practical examples.

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 (*kld*s, 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.

kldstat

We'll look at the *kldstat* macro first, because it's simpler. 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 preprocessor 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-prot
otypes  -Wmissing-prototypes -Wpointer-arith -Winline -Wcast-qual  -fformat-extensions
 -std=c99 -g -nostdinc -I-  -I. -I../../.. -I../../../dev -I../../../contrib/dev/acpic
a -I../../../contrib/ipfilter -I../../../contrib/dev/ath -I../../../contrib/dev/ath/fr
eebsd -D_KERNEL -include opt_global.h -fno-common -finline-limit=15000 -fno-strict-ali
asing  -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-pr
ototypes -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
ebsd -D_KERNEL -include opt_global.h -fno-common -finline-limit=15000 -fno-strict-alia
sing -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
y                                                  see text
   set $file = linker_files.tqh_first              note $ for local variables
   printf "Id Refs Address    Size     Name\n"     no parentheses for functions
   while ($file != 0)
     printf "%2d %4d 0x%8x %8x %s\n",   \          effectively C syntax
        $file->id,                      \
        $file->refs,                    \
        $file->address,                 \
        $file->size,                    \
        $file->filename
```

```
      set $file = $file->link.tqe_next           note set keyword for assignments
    end
end
document kldstat
Equivalent of the kldstat(8) command, without options.
end
```

Document the macro after its definition. If you try to do it before, *gdb* complains that the function doesn't exist.

Note the line `y` by itself on the second line. This is a macro call. `y` is defined as:

```
define y
echo Check your .gdbinit, it contains a y command\n
end
```

That doesn't seem to make much sense. In fact, this is a workaround for a limitation of *gdb* sooner or later you'll need to modify a macro (presumably in an editor) and copy and paste it back to the *gdb* session. There's no way to tell an interactive *gdb* "just do it": it responds with something like:

```
(gdb) define kldstat
Redefine command "kldstat"? (y or n)    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
Please answer y or n.
Redefine command "kldstat"? (y or n) Please answer y or n.
Redefine command "kldstat"? (y or n) Please answer y or n.
(etc)
```

In other words, it completely ignores all input until you enter `y` or `n`. You can solve this problem by putting a `y` in the second line, but then it wouldn't work the first time you tried to execute it. That's the purpose of the `y` macro.

Another example: 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
          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/*.

NAME

ddb — interactive kernel debugger

SYNOPSIS

options DDB

To prevent activation of the debugger on kernel `panic`(9):
options DDB_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 `DDB_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 don't 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:

VAX	don't assume that each external label is a procedure entry mask
i386	don't round to the next long word boundary
MIPS	print register contents

`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 *`addr`*s 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's 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 doesn't find the searched-for value. This is because ddb doesn't 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.

dhwatch *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.
`work`*xx*	Work variable. *xx* 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, [*filename*:]*func*:*lineno*, [*filename*:]*variable*, and [*filename*:]*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 **ddb**. 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 vmspace structure. The vmspace 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 evcnt(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**[**l**]] [*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

/l 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[**l**] [*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 **/l** 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 *expression*s 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 *expression*s, 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).
vnode	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.
dtsb	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:

lines	The number of lines. This is used by the **more** feature.
maxoff	Addresses are printed as `'symbol'+offset` unless `offset` is greater than *maxoff*.
maxwidth	The width of the displayed line.
onpanic	If non-zero (the default), **ddb** will be invoked when the kernel panics. If the kernel configuration option **options DDB_ONPANIC=0** is used, *onpanic* will be initialized to off.
fromconsole	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, *fromconsole* will be initialized to off.
radix	Input and output radix.
tabstops	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:

identifier	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, [*filename*:]*function*[:*line number*],[*filename*:]*variable*,and *filename*[:*line number*], can be accepted as a symbol. The symbol may be prefixed with *symbol_table_name*:: (e.g., `emulator::mach_msg_trap`) to specify other than kernel symbols.
number	number. Radix is determined by the first two characters: '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 the **examine** or **write** commands.
"	last address explicitly specified.
$*name*	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.
**expr*	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
options GDB_REMOTE_CHAT

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 firewire link. In this mode, the processor can be stopped and single stepped.
- With a firewire 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"
```

To share a console and debug connection on a serial line, use the **`options GDB_REMOTE_CHAT`** option.

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`**. The **`options GDB_REMOTE_CHAT`** is not necessary, since the firewire implementation uses separate ports for the console and debug connection.

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/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) 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, respectively. 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/cuaa0`.

tr1 Debug a remote system via serial interface `/dev/cuaa1`.

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 *ebp* value.

xi List the next 10 instructions from the current *eip* 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 *pid*
Show a backtrace for the process *pid*.

btpa Show backtraces for all processes in the system.

btpp Show a backtrace for the process previously selected with **defproc**.

btr *ebp*
Show a backtrace from the *ebp* address specified.

defproc *pid*
Specify the PID of the process for some other commands in this section.

fr *frame*
Show frame *frame* 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.

bpl 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 `0xdeadc0de`.

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.

AUTHORS

This man page was written by Greg Lehey ⟨grog@FreeBSD.org⟩.

SEE ALSO

`gdb`(1), `ps`(1), `ddb`(4), `firewire`(4), `vinumdebug`(4), `dconschat`(8), `dmesg`(8), `fwcontrol`(8), `kldload`(8)

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 growed". 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.