NAME

gdb - external kernel debugger

SYNOPSIS

makeoptions DEBUG=-g
options DDB

DESCRIPTION

The **gdb** kernel debugger is a variation of gdb(1) (*ports/devel/gdb*) 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) (*ports/devel/gdb*) that the "dump file" */dev/mem* is a kernel data file. You can look at live data, and if you include the **-wcore** option, you can change it at your peril. The system does not stop (obviously), so a number of things will not work. You can set breakpoints, but you cannot "continue" execution, so they will not work.

Debugging a crash dump

By default, crash dumps are stored in the directory */var/crash*. Investigate them from the kernel build directory with:

```
gdb -k kernel.debug /var/crash/vmcore.29
```

In this mode, the system is obviously stopped, so you can only look at it.

Debugging a live system with a remote link

In the following discussion, the term "local system" refers to the system running the debugger, and "remote system" refers to the live system being debugged.

To debug a live system with a remote link, the kernel must be compiled with the option **options DDB**. The option **options BREAK_TO_DEBUGGER** enables the debugging machine stop the debugged machine once a connection has been established by pressing ‘^C’.

Debugging a live system with a remote serial link

When using a serial port for the remote link on the i386 platform, the serial port must be identified by setting the flag bit 0x80 for the specified interface. Generally, this port will also be used as a serial console (flag bit 0x10), so the entry in */boot/device.hints* should be:

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

Debugging a live system with a remote firewire link

As with serial debugging, to debug a live system with a firewire link, the kernel must be compiled with the option **options DDB**.

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

- Ensure that both systems have firewire(4) support, and that the kernel of the remote system includes the dcons(4) and dcons_crom(4) drivers. If they are not compiled into the kernel, load the KLDs:

```
kldload firewire
```

On the remote system only:

```
kldload dcons
kldload dcons_crom
```

You should see something like this in the dmesg(8) output of the remote system:

```
fwohci0: BUS reset
fwohci0: node_id=0x8800ffc0, gen=2, non CYCLEMASTER mode
firewire0: 2 nodes, maxhop <= 1, cable IRM = 1
firewire0: bus manager 1
firewire0: New S400 device ID:00c04f3226e88061
dcons_crom0: <dcons configuration ROM> on firewire0
dcons_crom0: bus_addr 0x22a000
```

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

```
dcons_crom_enable="YES"
```

This ensures that all three modules are loaded. There is no harm in loading dcons(4) and dcons_crom(4) on the local system, but if you only want to load the firewire(4) module, include the following in */boot/loader.conf*:

```
firewire_enable="YES"
```

- Next, use fwcontrol(8) to find the firewire node corresponding to the remote machine. On the local machine you might see:

```
# fwcontrol
2 devices (info_len=2)
node         EUI64        status
   1  0x00c04f3226e88061      0
   0  0x000199000003622b      1
```

The first node is always the local system, so in this case, node 0 is the remote system. If there are more than two systems, check from the other end to find which node corresponds to the remote system. On the remote machine, it looks like this:

```
# fwcontrol
2 devices (info_len=2)
```

```
node       EUI64        status
  0  0x000199000003622b      0
  1  0x00c04f3226e88061      1
```

- Next, establish a firewire connection with dconschat(8):

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

  0x000199000003622b is the EUI64 address of the remote node, as determined from the output of fwcontrol(8) above. When started in this manner, dconschat(8) establishes a local tunnel connection from port localhost:5556 to the remote debugger. You can also establish a console port connection with the **-C** option to the same invocation dconschat(8). See the dconschat(8) manpage for further details.

  The dconschat(8) utility does not return control to the user. It displays error messages and console output for the remote system, so it is a good idea to start it in its own window.

- Finally, establish connection:

  ```
  # gdb kernel.debug
  GNU gdb 5.2.1 (FreeBSD)
  (political statements omitted)
  Ready to go.  Enter 'tr' to connect to the remote target
  with /dev/cuau0, 'tr /dev/cuau1' to connect to a different port
  or 'trf portno' to connect to the remote target with the firewire
  interface.  portno defaults to 5556.

  Type 'getsyms' after connection to load kld symbols.

  If you are debugging a local system, you can use 'kldsyms' instead
  to load the kld symbols.  That is a less obnoxious interface.
  (gdb) trf
  0xc21bd378 in ?? ()
  ```

  The **trf** macro assumes a connection on port 5556. If you want to use a different port (by changing the invocation of dconschat(8) above), use the **tr** macro instead. For example, if you want to use port 4711, run dconschat(8) like this:

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

Then establish connection with:

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

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

In addition to the conventional debugging via firewire described in the previous section, it is possible to debug a remote system without its cooperation, once an initial connection has been established. This corresponds to debugging a local machine using */dev/mem*. It can be very useful if a system crashes and the debugger no longer responds. To use this method, set the sysctl(8) variables *hw.firewire.fwmem.eui64_hi* and *hw.firewire.fwmem.eui64_lo* to the upper and lower halves of the EUI64 ID of the remote system, 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) (*ports/devel/gdb*), so gdb(1) (*ports/devel/gdb*) 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/cuau0*.

tr1 Debug a remote system via serial interface */dev/cuau1*.

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) (*ports/devel/gdb*) 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) (*ports/devel/gdb*) 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.

SEE ALSO

gdb(1) (*ports/devel/gdb*), ps(1), ddb(4), firewire(4), dconschat(8), dmesg(8), fwcontrol(8), kldload(8)

AUTHORS

This man page was written by Greg Lehey <*grog@FreeBSD.org*>.

BUGS

The gdb(1) (*ports/devel/gdb*) 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 grew." 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.