# Debugging with GDB on FreeBSD

by John Baldwin

Debuggers are tools used to inspect the state of processes in a system. For running processes, debuggers can examine the values of variables as well as control process execution. For processes that have terminated abnormally due to a bug, debuggers can parse a process core dump generated by the kernel to inspect the state of the process at the time of the crash.

Most debuggers support a base set of features such as examining the value of global and local variables, generating stack traces, interrupting process execution via breakpoints, and controlling process or thread execution via stepping. This article focuses on some of the other features supported by modern versions of the GNU debugger (gdb) on FreeBSD. Some of these features are only supported in the latest version of gdb (8.3 at the time of writing), while other features are available in older versions.

To get started, install an up-to-date version of gdb. The simplest way is to install the pre-built package by running `pkg install gdb`. Alternatively, gdb can be built from source via the `devel/gdb` port (https://www.freshports.org/devel/gdb).

## The info proc Command

The `info proc` command can be used to examine process state beyond memory and threads. By default, `info proc` provides basic information such as the process ID and command line of a process. However, additional information is available via subcommands including the list of open file descriptors via `info proc files` and the list of active memory mappings via `info proc mappings`. In the first two examples, the command `wc /usr/src/bin/ls/ls.c` is being executed under the debugger. Execution is paused inside the `cnt` function with the target file open. The first example shows the information provided by the basic command. The second example shows the list of open files and includes the offset of the file descriptor for `ls.c` showing how much of the file the `wc` process has read.

While all the information available via info proc is also provided by other utilities such as `ps(1)` (https://www.freebsd.org/cgi/man.cgi?query=ps(1)).and `procstat(1)` (https://www.freebsd.org/cgi/man.cgi?query=procstat(1)), the info proc command permits a user to access them from within the debugger without having to open a separate window. These commands can also be used on a cross-debugger hosted on a non-FreeBSD OS while examining a core dump.

More detailed information on the info proc command and its subcommands can be found in the Process Information section (https://sourceware.org/gdb/current/onlinedocs/gdb/Process-Information.html) of the Debugging with GDB manual (https://sourceware.org/gdb/current/onlinedocs/gdb/).

**Example 1: info proc**

```
(gdb) info proc
process 85146
cmdline = '/usr/bin/wc /usr/src/bin/ls/ls.c'
cwd = '/usr/home/john'
exe = '/usr/bin/wc'
```

**Example 2: info proc files**

```
(gdb) info proc files
process 85153
Open files:

  FD      Type       Offset      Flags            Name
 text     file            -      r----------      /usr/bin/wc
 ctty      chr            -      rw--------       /dev/pts/20
  cwd      dir            -      r----------      /usr/home/john
 root      dir            -      r----------      /
    0      chr      0xac82      rw--------       /dev/pts/20
    1      chr      0xac82      rw--------       /dev/pts/20
    2      chr      0xac82      rw--------       /dev/pts/20
    3     file      0x63dd      r----------      /usr/src/bin/ls/ls.c
```

## Intercepting System Calls

GDB supports a special class of breakpoints called catchpoints. Catchpoints permit the user to pause execution when certain types of events occur during execution. One of the catchpoint types supported by GDB is a system call catchpoint. A system call catchpoint pauses execution on entry and exit from system calls.

System call catchpoints are created using the `catch syscall` command. If no arguments are specified, execution will pause on entry and exit from all system calls. More specific catchpoints can be defined by specifying a list of system calls as arguments to the command. System calls can be named either by name or number. For example, `catch syscall write` sets a catchpoint that will pause execution on entry and exit from the `write(2)` (https://www.freebsd.org/cgi/man.cgi?query=write(2)) system call.

Once a system call catchpoint has been created, it can be managed using other commands used with breakpoints. The `info breakpoints` command will list catchpoints along with other breakpoints. Catchpoints are removed via the `delete` command. Example 3 intercepts a `write(2)` (https://www.freebsd.org/cgi/man.cgi?query=write(2)) system call of a `ls(1)` (https://www.freebsd.org/cgi/man.cgi?query=ls(1)) process.

**Example 3: Catching a System Call**

```
% gdb -q --args /bin/ls -l /bin/sh
Reading symbols from /bin/ls...
Reading symbols from /usr/lib/debug//bin/ls.debug...
(gdb) catch syscall write
Catchpoint 1 (syscall 'write' [4])
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       catchpoint     keep y              syscall "write"
(gdb) run
Starting program: /bin/ls -l /bin/sh

Catchpoint 1 (call to syscall write), _write () at _write.S:3
3     PSEUDO(write)
(gdb) c
Continuing.
-r-xr-xr-x  1 root  wheel  168880 Nov 17 17:38 /bin/sh

Catchpoint 1 (returned from syscall write), _write () at _write.S:3
3     PSEUDO(write)
(gdb) c
Continuing.
[Inferior 1 (process 24875) exited normally]
```

For FreeBSD, GDB recognizes compatibility system calls. Catching a system call by name that has compatibility system calls for older versions will catch all versions of that system call. For example, several system calls moved to new numbers in FreeBSD 12 due to changes in `struct stat`. The existing system calls continue to use the old layout of `struct stat` but were renamed to add a `freebsd11_` prefix. When one of these system calls is caught by name, GDB catches both since applications might use either version. In Example 4, catching the `fstat(2)` (https://www.freebsd.org/cgi/man.cgi?query=fstat(2)) system call registers catchpoints for both versions.

**Example 4: Catch fstat(2)**

```
(gdb) catch syscall fstat
Catchpoint 1 (syscalls 'freebsd11_fstat' [189] 'fstat' [551])
(gdb) info breakpoints
Num     Type          Disp Enb Address     What
1       catchpoint    keep y               syscalls "freebsd11_fstat, fstat"
```

## Debugging Forks

Many programs create new processes via the `fork(2)` (https://www.freebsd.org/cgi/man.cgi?query=fork(2)) and `vfork(2)` (https://www.freebsd.org/cgi/man.cgi?query=vfork(2)) system calls. GDB provides several facilities for working with child processes created as a result of a fork. The following examples will demonstrate these facilities using the test program from Listing 1.

**Listing 1: forktest.c**

```
#include <sys/types.h>
#include <sys/wait.h>
#include <err.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(void)
{
	pid_t pid, wpid;

	pid = fork();
	if (pid == -1)
		err(1, "fork");
	if (pid == 0) {
		printf("I'm in the child\n");
		exit(1);
	}
	printf("I'm in the parent\n");
	wpid = waitpid(pid, NULL, 0);
	if (wpid < 0)
		err(1, "waitpid");

	return (0);
}
```

## Fork Follow Mode

When a debugged process forks, GDB has to choose which process to continue debugging: the original (parent) process, or the new (child) process. By default, GDB follows the parent process letting the child process run freely after a fork as in Example 5. Note that the child process writes its output to the console and exits even while the parent process is paused in the debugger.

**Example 5: `Following the Parent`**

```
(gdb) start
Temporary breakpoint 1 at 0x201354: file forktest.c, line 13.
Starting program: /usr/home/john/work/johnsvn/test/forktest/forktest

Temporary breakpoint 1, main () at forktest.c:13
13          pid = fork();
(gdb) n
[Detaching after fork from child process 25302]
I'm in the child
14          if (pid == -1)
(gdb) p pid
$1 = 25302
(gdb) n
20          printf("I'm in the parent\n");
(gdb) c
Continuing.
I'm in the parent
[Inferior 1 (process 25297) exited normally]
```

GDB uses the `follow-fork-mode` setting to determine which process to follow after a fork. To follow the child process instead of the parent, use the "child" setting. To restore the default behavior, use the "parent" setting. The setting is changed via the `set follow-fork-mode` command. The `show follow-fork-mode` command displays the current setting. Example 6 executes the test program again but follows the child process instead of the parent process.

**Example 6: `Following the Child`**

```
(gdb) set follow-fork-mode child
(gdb) start
Temporary breakpoint 1 at 0x201354: file forktest.c, line 13.
Starting program: /usr/home/john/work/johnsvn/test/forktest/forktest

Temporary breakpoint 1, main () at forktest.c:13
13        pid = fork();
```

Continues next page

Example 6 continued from page 8

```
(gdb) n
[Attaching after LWP 100857 of process 26342 fork to child LWP 101958 of process 26347]
[New inferior 2 (process 26347)]
[Detaching after fork from parent process 26342]
[Inferior 1 (process 26342) detached]
I'm in the parent
[Switching to LWP 101958 of process 26347]
main () at forktest.c:14
14          if (pid == -1)
(gdb) p pid
$1 = 0
(gdb) n
17                printf("I'm in the child\n");
(gdb) c
Continuing.
I'm in the child
[Inferior 2 (process 26347) exited with code 01]
```

## Detach on Fork

In addition to deciding which process to follow after a fork, GDB provides a choice on how to treat the non-followed process. By default, GDB detaches from the non-followed process allowing it to run freely after the fork. The **`detach-on-fork`** setting can be changed to "no" to change this behavior. When it is set to "no", GDB stays attached to both processes and leaves both processes paused after a fork.

To manage the two processes, GDB's multiprocess support (https://sourceware.org/gdb/current/onlinedocs/gdb/Inferiors-and-Programs.html#Inferiors-and-Programs) is used. In GDB terminology, each process is associated with an "inferior". The **`info inferiors`** command is used to list the active inferiors. The **`inferior`** command is used to switch between inferiors. Threads from separate processes are also displayed in the **`info threads`** command. Switching to a thread in different inferior is another way to switch inferiors. After a fork, the inferior of the followed process is set as the current inferior.

Example 7 provides one more example of the test program. This time, **`detach-on-fork`** is disabled leaving both processes paused after the fork. The default fork follow mode is used, so GDB remains focused on the parent process after the fork. Note that the child process is stopped at the first instruction after the fork.

**Example 7: Staying Attached after Fork**

```
(gdb) set detach-on-fork off
(gdb) start
Temporary breakpoint 1 at 0x201354: file forktest.c, line 13.
```

Continues next page

## Example 7 continued from page 9

```
Starting program: /usr/home/john/work/johnsvn/test/forktest/forktest

Temporary breakpoint 1, main () at forktest.c:13
13          pid = fork();
(gdb) n
[New inferior 2 (process 26828)]
14          if (pid == -1)
(gdb) p pid
$1 = 26828
(gdb) info inferiors
  Num      Description     Executable
   *1      process 26823   /usr/home/john/work/johnsvn/test/forktest/forktest
    2      process 26828   /usr/home/john/work/johnsvn/test/forktest/forktest
(gdb) inferior 2
[Switching to inferior 2 [process 26828]
(/usr/home/john/work/johnsvn/test/forktest/forktest)]
[Switching to thread 2.1 (LWP 101425 of process 26828)]
Reading symbols from
/usr/home/john/work/johnsvn/test/forktest/forktest.debug...done.
Reading symbols from /usr/lib/debug/lib/libc.so.7.debug...done.
Reading symbols from /usr/lib/debug/libexec/ld-elf.so.1.debug...done.
#0    _fork () at _fork.S:3
3      PSEUDO(fork)
Warning: the current language does not match this frame.
(gdb) n
main () at forktest.c:14
14          if (pid == -1)
(gdb) p pid
$2 = 0
(gdb) info threads
  Id  Target Id                         Frame
  1.1   LWP 100970 of process 26823  main () at forktest.c:14
 *2.1   LWP 101425 of process 26828  main () at forktest.c:14
(gdb) c
Continuing.
I'm in the child
[Inferior 2 (process 26828) exited with code 01]
(gdb) thread 1.1
[Switching to thread 1.1 (LWP 100970 of process 26823)]
#0 main () at forktest.c:14
14          if (pid == -1)
(gdb) c
Continuing.
I'm in the parent
[Inferior 1 (process 26823) exited normally]
```

## Catching Forks

GDB provides one final set of tools to aid with debugging forking processes: a set of catchpoints for events related to forks. The `catch fork` command installs a catchpoint for fork invocations that are not from `vfork(2)`(https://www.freebsd.org/cgi/man.cgi?query=vfork(2)). The catchpoint triggers when the followed process returns from forking. The `catch vfork` command installs a catchpoint for fork invocations from `vfork(2)`. Finally, the `catch exec` command installs a catchpoint for returns from the exec family of system calls. Example 8 follows a shell process that forks a child process to execute a command.

**Example 8: `Catching Fork and Exec`**

```
% gdb -q /bin/sh
(gdb) catch fork
Catchpoint 1 (fork)
(gdb) catch exec
Catchpoint 2 (exec)
(gdb) set follow-fork-mode child
(gdb) run
Starting program: /bin/sh
$ ls -l /dev/null; exit

Catchpoint 1 (forked process 27644), _fork () at _fork.S:3
3     PSEUDO(fork)
(gdb) c
Continuing.
[Attaching after LWP 100469 of process 27639 fork to child LWP 101734 of
process 27644]
[New inferior 2 (process 27644)]
[Detaching after fork from parent process 27639]
[Inferior 1 (process 27639) detached]
process 27644 is executing new program: /bin/ls

Thread 2.1 hit Catchpoint 2 (exec'd /bin/ls), .rtld_start ()
      at /usr/src/libexec/rtld-elf/amd64/rtld_start.S:33
33          xorq%rbp,%rbp#        Clear frame pointer for good form
(gdb) c
Continuing.
crw-rw-rw- 1 root wheel 0xf Feb 2 18:00 /dev/null
[Inferior 2 (process 27644) exited normally]
```

For more information on using GDB to debug forks, see the Debugging Forks (https://sourceware.org/gdb/current/onlinedocs/gdb/Forks.html) chapter of the GDB manual.

## Debugging C++ STL Classes

For some data types, the raw layout of a data structure might not correspond to the use and representation of that structure in the source code. This can be especially true of C++ Standard Template Library (STL) classes. To aid with inspecting these structures, GDB permits python scripts to provide two types of helper classes: pretty printers (https://sourceware.org/gdb/current/onlinedocs/gdb/Pretty-Printing-API.html#Pretty-Printing-API) and xmethods (https://sourceware.org/gdb/current/onlinedocs/gdb/Xmethods-In-Python.html#Xmethods-In-Python).

Pretty printers override the default display of objects by the print command. Each pretty printer is associated with one or more C++ classes. They can also be associated with templated classes. For example, a pretty printer for `std::vector` can display the contents of the vector as an array.

Xmethods permit python scripts to simulate the effects of inlined C++ class methods. When evaluating expressions, GDB will call functions defined in the debugged program if needed to evaluate an expression. This includes calling C++ operator overloading functions. However, if methods are inlined, as is common with templated classes, there is no discrete function symbol for GDB to call. As a result, attempting to use these functions or operators in an expression fails. Xmethods can be used to bridge this gap. For example, an xmethod can be used to provide `operator[]` for `std::vector` objects permitting a user to directly index a vector with the same syntax used in the original C++ source.

The LLVM C++ library used by FreeBSD does not include a set of python scripts providing pretty printers and xmethods for commonly-used C++ STL classes. However, an initial set of scripts are available at (https://github.com/bsdjhb/libcxx-gdbpy). At the time of writing, these scripts have limited support for `std::string`, `std::unique_ptr`, and `std::vector`. Examples 9 and 10 compare examining a `std::vector` of integers without and with these scripts installed. These python scripts are included by the devel/gdb port by default in versions 8.2.1_1 and newer.

**Example 9: `std::vector Without Python Scripts`**

```
(gdb) p vector
$1 = {<std::__1::__vector_base<int, std::__1::allocator<int> >> =
{<std::__1::__vector_base_common<true>> = {<No data fields>}, __begin_ =
0x800244000,
     __end_ = 0x80024400c,
     __end_cap_ = {<std::__1::__compressed_pair_elem<int*, 0, false>> = {
          __value_ = 0x800244010},
<std::__1::__compressed_pair_elem<std::__1::allocator<int>, 1, true>> =
{std::__1::allocator<int>> = {<No data fields>}, <No data fields>}, <No data fields>}}, <No data fields>}
```

Continues next page

Example 9 continued from page 12

```
(gdb) p vector[1]
Could not find operator[].
```

**Example 10:** **`std::vector With Python Scripts`**

```
(gdb) p vector
$1 = std::vector of length 3 = {4, 5, 6}
(gdb) p vector[1]
$2 = 5
```

## Cross Debugging with a System Root

The GDB package from ports is built by default as a cross-debugger. This means that it is able to examine binaries and core dumps from other architectures and other operating systems. For example, one can examine a process core dump from an embedded FreeBSD ARM system using a GDB process on a faster x86 host. Another use case is debugging the kernel of a remote machine over a serial connection.

When cross-debugging a self-contained binary such as a static binary or monolithic kernel, GDB is able to find all of the information it needs from the binary. However, when debugging a binary that depends on other binaries such as shared libraries or kernel modules, GDB needs to be able to find these other binaries. Normally GDB looks for these binaries on the host where it is being run, which works fine when debugging a native process or core dump. However, when cross-debugging, GDB needs to be able to access these additional binaries. This can be solved by using a system root.

A system root is a copy of the shared binaries from a system stored at an alternate directory. Often a system root may contain a full system installation image. If the system root contains headers and libraries used by compilers, a cross-compiler can use the system root to compile binaries for the alternate system. Both GCC and clang use the `--sysroot` flag to instruct the compiler to look for headers and libraries inside of a system root. In GDB, a system root is indicated by setting the `sysroot` variable to the path of the system root. GDB will then look for shared libraries and kernel modules under that system root rather than in the host's root file system.

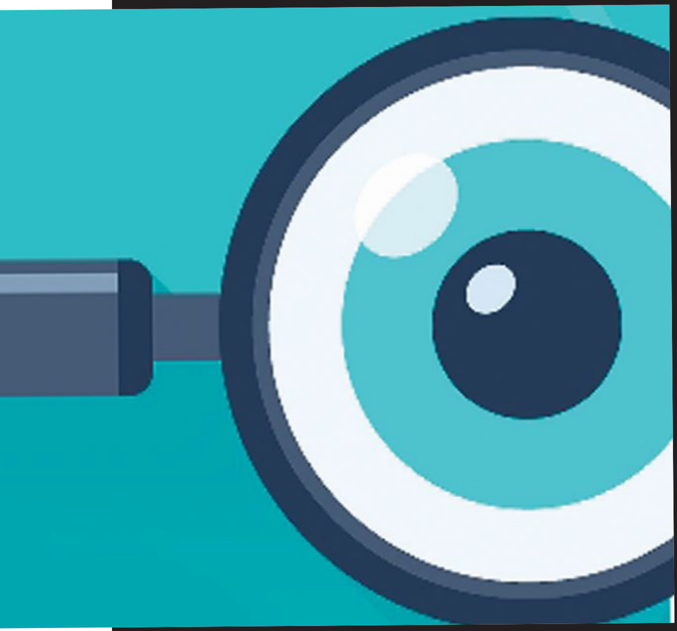
As an example, suppose a process run on a Raspberry Pi crashed generating a core dump. One can take the SD card from the Raspberry Pi and insert it into an x86 machine. The SD card can then be mounted, and the core dump can be examined on the x86 machine. GDB just needs to be told to use the mount point of the SD card as the system root. Example 11 uses this to look at an ARM core dump running on an x86 host. In this case, the SD card from the Raspberry Pi was mounted at `/mnt`.

**Example 11: Examining an ARM Core Dump on an x86 Host**

```
> gdb -q sigframe
Reading symbols from sigframe...Reading symbols from
/mnt/home/john/work/johnsvn/test/sigframe/sigframe.debug...done.
done.
(gdb) set sysroot /mnt
(gdb) core-file sigframe.core
[New LWP 100086]
Core was generated by `./sigframe'.
Program terminated with signal SIGABRT, Aborted.
#0 thr_kill () at thr_kill.S:3
3    RSYSCALL(thr_kill)
(gdb) info sharedlibrary
From           To             Syms Read      Shared Object Library
0x20092000     0x201e516c     Yes            /mnt/lib/libc.so.7
0x20016000     0x20030ef4     Yes            /mnt/libexec/ldelf.so.1
```

If you forget to set the sysroot variable before loading the core file, you can still set it after the core file is loaded. GDB will look for shared libraries under the new system root automatically after it is changed.

The sysroot variable also works when debugging a remote FreeBSD kernel. GDB will look for kernel modules and their associated debug information under the system root. This can be useful even when the target machine is the same architecture as the host but is running an operating system or operating system version different from the host. ■



**John Baldwin** is a systems software developer. He has directly committed changes to the FreeBSD operating system for nineteen years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger and LLVM. John lives in Concord, California with his wife, Kimberly, and three children: Janelle, Evan, and Bella.