FreeBSD and GDB

John Baldwin

June 11, 2016

Overview

- Structure of GDB
- Recent Userland Debugging Changes
- Kernel Debugging

GDB Concepts

- Inferior
  - Something you can debug (e.g. a running process, or a former process described by a core dump)
- GDB Architecture
  - Describes a process ABI (e.g. FreeBSD/amd64 ELF)
- Targets
  - Interface for interacting with an inferior

GDB Architectures (ABIs)

- struct gdbarch describes an ABI “class”
- Includes ABI-specific methods for certain targets
  - Core file target uses ABI methods to parse core file register notes
- Pointer to a shared library operations structure
- Signal frame handling

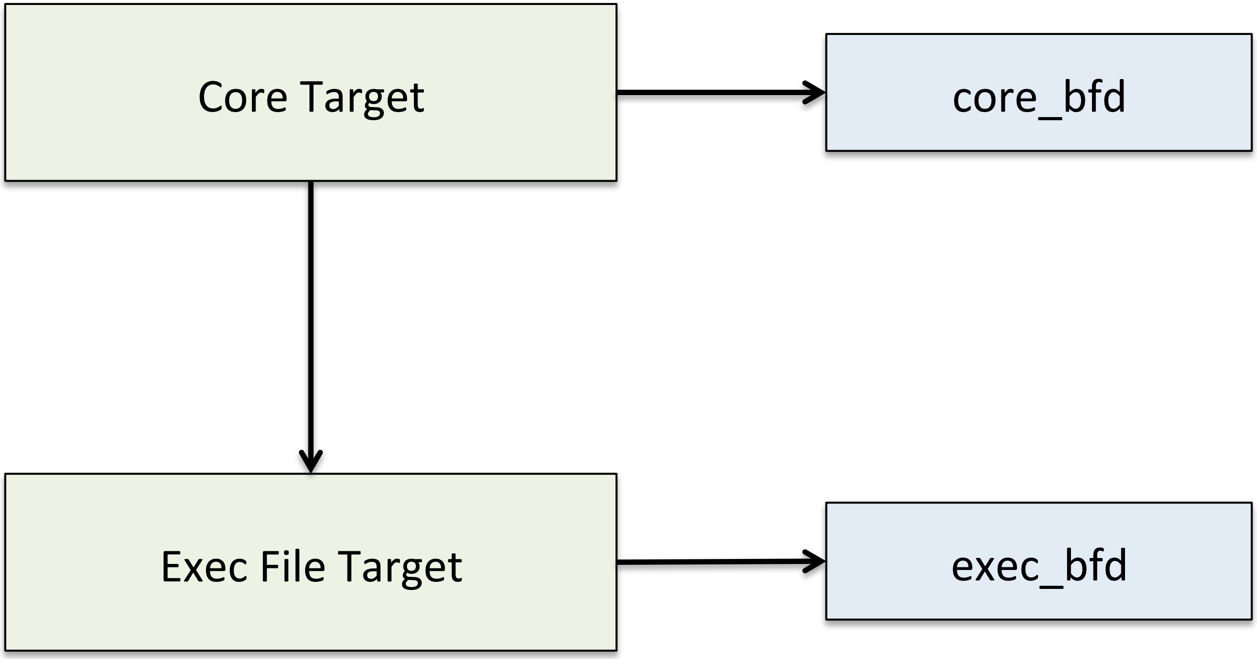
GDB Architectures (ABIs)

- ABIs are defined in ‘*tdep.c’ files
  - fbsd-tdep.c holds FreeBSD routines common to all FreeBSD ABIs
  - amd64fbsd-tdep.c defines the FreeBSD/amd64 ABI
- ABI “sniffers” match against binaries
  - For example, ELF header fields
- Associated initialization routine sets gdbarch members when sniffer “matches”

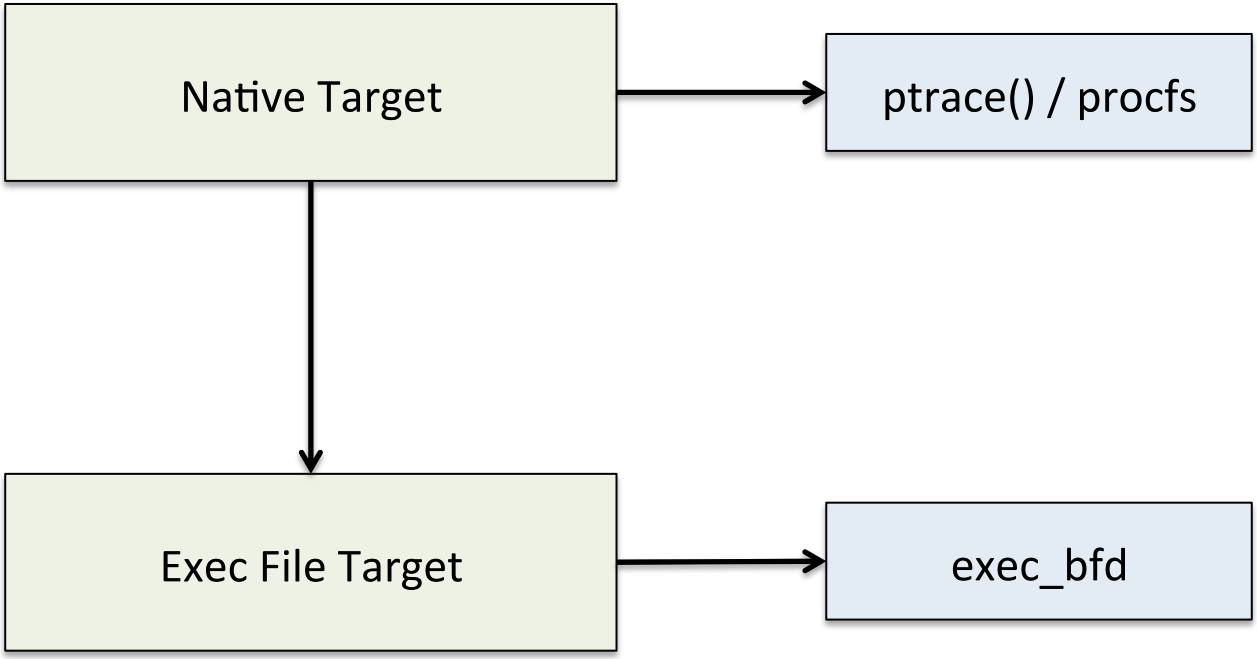
GDB Targets

- Targets provide an interface to interact with an inferior
  - Read and write memory
  - Get and set register values
  - Enumerate threads
  - Wait for an event
- Multiple targets can be attached to a single inferior in a stack
  - Upper targets may pass operations down to lower targets

GDB Targets – Core Dump



GDB Targets – Running



Native Target

ptrace() / procfs

Exec File Target

exec_bfd

Native Targets

- Native targets are used with executing processes
  - “run”
  - Attach to an existing process
- Native targets are defined in ‘inf-*.c’ and ‘*nat.c’ files

Native Targets

- inf-child.c
  - Base class of all native targets
- inf-ptrace.c
  - OS-independent base ptrace() target
    - PT_IO, PT_CONTINUE, PT_STEP, wait()
- fbsd-nat.c
  - Platform-independent FreeBSD-specific ptrace() methods

Native Targets (BSD)

- *BSD targets often share pan-BSD code
- amd64bsd-nat.c
  - ptrace() operations to get and set registers
- amd64fbsd-nat.c
  - FreeBSD/amd64 specific target
  - Glues together bits from amd64bsd-nat.c and fbsd-nat.c

Recent Userland Changes

- Fork following (gdb 7.10)
- LWP-based thread support (gdb 7.11)

Fork Following

- Native target requirements
  - Automatically stop new child processes
  - Report fork() event (including pid of new child process) to debugger
- Could handle second by tracing all system call exits and pulling return value out of registers for SYS_fork and SYS_vfork
  - That's ugly and requires an MD callback
  - Still doesn't solve first requirement

PT_LWPINFO

- FreeBSD’s ptrace() includes a PT_LWPINFO operation to request extended state on a process or thread
- Requesting state for a process reports the thread that triggered the current stop
- PT_LWPINFO populates a ‘struct ptrace_lwpinfo’ structure

struct ptrace_lwpinfo

- More details in ptrace(2)
- pl_lwpid
- pl_flags
  - PL_FLAG_SCE: stopped at system call entry
  - PL_FLAG_SCX: stopped at system call exit
- pl_tdname

Fork Following in FreeBSD

- Fully functional ptrace() interface shipped in 9.1
- PT_FOLLOW_FORK
  - Requests auto-attach to new child process
  - Set ‘data’ to zero to disable or non-zero to enable

Fork Following in FreeBSD

- New fields and flags in struct ptrace_lwpinfo
- PL_FLAG_FORKED
  - Set in pl_flags of parent process
- PL_FLAG_CHILD
  - Set in pl_flags of new child process on first stop
- pl_child_pid
  - Set to pid of new child process when PL_FLAG_FORKED is set

Fork Following in GDB

- fbsd-nat.c defines a new target "wait" method
- Uses PT_LWPINFO to recognize fork events and report them as fork events rather than plain "stops"
  - TARGET_WAITKIND_FORKED or TARGET_WAITKIND_VFORKED
  - Have to wait for both processes to stop before reporting event to GDB
- Enable PT_FOLLOW_FORK unconditionally

FreeBSD Thread Support in GDB

- Originally written by multiple developers under a BSD license
  - Not feasible to upstream
- Used libthread_db
  - Pros: supported libc_r, libkse, libthr
  - Cons: did not support other ABIs like compat32, Linux; would need API changes for XSAVE/AVX; each platform had to export custom register conversion routines

FreeBSD Thread Support in GDB

- Wanted an upstreamed thread target
- No one uses libc_r or libkse anymore
- Using libthread_db requires a lot of code
- Assuming LWPs (libthr) and using ptrace() directly is less code
- Platform native targets merely need to handle LWP IDs with ptrace() register requests
    - Some already did since other OS's do the same

ptrace() and LWPs in FreeBSD

- PT_GETNUMLWPS
  - Returns number of valid LWPs for a process
- PT_GETLWPLIST
  - Populates an array of LWP IDs
- PT_GETLWPINFO
  - Current state of each LWP
- PT_SUSPEND / PT_RESUME
  - Suspend/resume individual LWPs

Handling LWP Events

- Need to know when threads start and exit
- Older target using libthread_db sets breakpoints in pthread_create() and pthread_exit()
- Newer target can rescan the LWP list on each stop
  - Means multiple ptrace() calls on every stop

LWP Events via ptrace()

- FreeBSD 11 adds LWP event reporting via ptrace()
- PT_LWP_EVENTS
  - Enables / disables LWP event reporting
- PL_FLAG_BORN
  - Set in pl_flags on new LWP on first stop
- PL_FLAG_EXITED
  - Set in pl_flags on exiting LWP on last stop

LWP Events via ptrace()

- Initial return from thread create system call by new threads now reports a system call exit stop event
  - No event was reported previously
  - System call exit event is always reported if system call exits are traced regardless of PT_LWP_EVENTS
  - No event reported for initial thread
- Exiting threads report a new stop event for PL_FLAG_EXITED
  - Final thread exit is reported via exit() instead

LWP Thread Target

- Enumerates LWPs and adds them as threads
- Only change to platform-specific targets is supporting LWP IDs in register operations
  - get_ptrace_pid() helper function handles this
- Uses PT_RESUME / PT_SUSPEND if a resume operation targets a specific thread

Tangent: truss

- truss –f now uses PT_FOLLOW_FORK
  - Used to fork a new truss process to follow each new child process
- truss now uses PT_LWP_EVENTS to report thread events
  - Since it can now tell which thread called exit() it also logs an event for exit()

Kernel Debugging

- Cross-debugging support in libkvm
- Components of kgdb
- Cross-debugging support in kgdb

Cross-Debugging in libkvm

- libkvm is a library that includes support for examining kernel crash dumps
- Specifically, it is able to translate kernel virtual addresses into file offsets and return the data referenced by a given kernel virtual address
- FreeBSD 11 adds support for examining crash dumps from non-native kernels
  - Earlier versions could only read a crash dump from the same architecture as the host

libkvm API Changes

- kvaddr_t
  - Type (currently uint64_t) used for kernel virtual addresses
  - Previously was unsigned long
  - Allows 32-bit binaries to specify a 64-bit KVA
- struct kvm_nlist
  - Like struct nlist, but uses kvaddr_t for n_value

libkvm API Changes

- kvm_open2()
  - Like kvm_open() but accepts an additional parameter
  - Parameter is a function pointer to a symbol resolver function
  - Resolver is required for non-native vmcores
- kvm_read2()
  - Like kvm_read(), just uses kvaddr_t for KVA

KVM_ARCH

- libkvm now supports multiple backends
  - Each backend supports a different vmcore format
  - Separate backends for “full” vs “mini” dumps
- Backends added to linker set via KVM_ARCH()
- Backends cannot use native constants / types directly (e.g. PAGE_SIZE, PTE constants)
- kvm_<platform>.h define MI VM constants
  - Statically asserts constants match

KVM_ARCH

- Backends include a probe function that examines a vmcore to see if it matches
  - Uses libelf to parse ELF headers
- Backends also include a callback to translate a KVA to a file offset
  - Used by kvm_read() and kvm_read2()

kgdb Components

- What is added to gdb to create kgdb?
- vmcore target
  - fbsd-kvm.c
  - Uses libkvm to read kernel memory from /dev/mem or a crash dump
  - “proc” and “tid” commands
- Kernel thread enumeration
  - fbsd-kthr.c
  - Used by vmcore target
  - Remote debugging relies on in-kernel GDB stub to enumerate threads

kgdb Components

- Shared library target for kernel modules
  - fbsd-kld.c
  - Uses kernel linker data structures to enumerate KLDs
  - Presents KLDs to users as shared libraries
  - “add-kld” command
- New ABI – FreeBSD ELF Kernel
  - Allows gdb to treat kernels differently than regular userland binaries
  - Detects FreeBSD kernel by checking for “/red/herring” dynamic interpreter

kgdb Components – MD

- Platform-specific code
- Special frame handlers (“unwinders”)
  - Interrupt, fault, and exception frames
  - Most just use a trapframe
  - i386 double fault frames require dealing with TSS

kgdb Components – MD

- Process (really Thread) Control Block hooks
  - Extract register state from PCB
  - Locate PCB of currently executing thread
    - stoppcbs[cpuid] on most platforms
- Kernel ABIs defined in ‘*fbsd-kern.c’
  - ABIs use KLD solibs hook rather than svr4
  - ABIs add custom unwinders
  - ABIs register PCB hooks for vmcore target

Cross-Debugging in kgdb

- Old kgdb used native structures directly
  - E.g. read ‘struct proc’ and use ‘p_list.le_next’ to locate next process
- As with libkvm, cannot do that in a cross-debugger
- Have to query ABI for pointer size and endianness
- GDB provides methods to decode an integer

Cross-Debugging in kgdb

- Have to explicitly handle structure layouts
- Can use debug symbols and manual offsetof()

```
proc_off_p_pid = parse_and_eval_address(
    "&((struct proc *)0)->p_pid");
proc_off_p_comm = parse_and_eval_address(
    "&((struct proc *)0)->p_comm");
proc_off_p_list = parse_and_eval_address(
    "&((struct proc *)0)->p_list");
```

Cross-Debugging in kgdb

- Recent kernels include helper variables
- Permits enumerating threads without debug symbols

```
const int proc_off_p_pid = offsetof(struct proc, p_pid);
const int proc_off_p_comm = offsetof(struct proc, p_comm);
const int proc_off_p_list = offsetof(struct proc, p_list);
```

- kgdb uses these symbols if they exist instead of manual offsetof()

Reading struct proc Fields

```
struct type *ptr_type =
    builtin_type (gdbarch)->builtin_data_ptr;
enum bfd_endian byte_order =
    gdbarch_byte_order (gdbarch);

...
tdaddr = read_memory_typed_address (paddr +
    proc_off_p_threads, ptr_type);
pid = read_memory_integer (paddr + proc_off_p_pid, 4,
    byte_order);
pnext = read_memory_typed_address (paddr +
    proc_off_p_list, ptr_type);
```

Cross-Debugging in kgdb

- PCB hooks and custom unwinders have to define constants for structure layouts
  - Similar to existing tables in userland ABIs for core dump register notes
- Parsing cpuset_t for stopped_cpus
  - Have to query ABI for size of long
  - Effectively inline CPU_ISSET() by hand
  - cpu_stopped() in fbsd-kthr.c

Future Work

- Adding support for more architectures (both userland and kernel)
  - X86 works and cross-debug of x86 works
  - ppc64 userland works fine, kgdb can’t parse PCBs correctly
- Various gdb features not yet supported
  - info auxv, info os
  - powerpc vector registers

Future Work

- Portable libkvm
  - Would only include vmcore support, not kvm_getprocs, etc.
  - Would permit kgdb/lldb hosted on non-FreeBSD
- bhyve gdb stub ala qemu
  - Export each vCPU as a “thread”
  - Use VT-x to single-step, etc.
  - Needs a new vmcore-like target

Conclusion

- Available in devel/gdb port
- pkg install gdb
- Phase out old gdb in base system?
- https://github.com/bsdjhb/gdb.git
  - freebsd-*-kgdb branches hold kgdb (currently freebsd-7.11-kgdb)
  - Non-kgdb bits are upstreamed to gdb master