Advanced ptrace() Usage on FreeBSD

John Baldwin
BSDCan
May 19, 2023
Overview

• Basics of `ptrace()`
• `ptrace()` extensions to support threads (LWPs)
• Improving support for multiple processes in GDB
• Some remaining issues to address in GDB’s native target
• Future Work
pttrace() Basics

• Debugger attaches to another process via PT_ATTACH
• Attached processes report status to the debugger via wait() for signals and process exit
  • Debugger can choose to discard an intercepted signal or pass it on when resuming a process via PT_CONTINUE
  • SIGTRAP for breakpoint instructions are typically discarded for example
• The kernel can inform the debugger of other interesting events by raising a special signal (usually SIGTRAP)
  • System call entry/exit
Fork Following

• Debuggers want the opportunity to debug new children of a debuggee

• PT_FOLLOW_FORK enables following
  • Kernel auto-attaches existing debugger to new child processes

• Events reported in both parent (”I forked”) and new child (”I’m a new fork child”)

• More details in an earlier BSDCan talk: https://papers.freebsd.org/2016/bsdcan/baldwin-freebsd_and_gdb/
ptrace() Extensions for Threads

- FreeBSD 5.x added initial support for multiple kernel threads (LWPs) per process.
- Each time a thread reports an event (signal), all threads stop:
  - Race to set p_xthread member to reporting thread.
- `PT_LWPINFO` provides more details about thread stop (and which LWP).
- `PT_SUSPEND` and `PT_RESUME` permit resuming only a subset of threads via `PT_CONTINUE`.
- `PT_LWP_EVENTS` added in 11.0 to report thread birth and exit.
PT_CONTINUE and threads

• Each PT_CONTINUE or PT_STEP “acknowledges” one thread event

• If multiple threads have events pending, then after PT_CONTINUE the remaining threads will race to set p_xthread and stop all the other threads that were just resumed

• Can only pass along a signal to a thread for the PT_CONTINUE or PT_STEP for that thread’s signal event
  • If you PT_SUSPEND the thread instead planning to resume it later, you can’t pass along the signal later when you PT_RESUME it
FreeBSD Native Target in GDB 13

• Supports threads (LWPs) in native processes
• Supports fork following
• Supports various other extensions like system call events and `info proc`
• Recently supports async mode and hardware watchpoints on aarch64
• Claims to support multi-process debugging
  • But is rather broken due to misunderstandings on my part
GDB bug 21497

- https://sourceware.org/bugzilla/show_bug.cgi?id=21497

A new thread “arrives” when GDB thinks it shouldn’t be executing:

- A new thread is created in a process but has not yet started executing
- Some event occurs that causes the process to stop and report an event to the debugger (e.g. an existing thread triggers a breakpoint)
- GDB wants to single-step one thread in the process leaving all other threads stopped (common after a breakpoint hit)
  - PT_SUSPEND all the other threads that GDB knows about (doesn’t know about new thread)
  - PT_STEP desired thread
- GDB expects next event to be result of PT_STEP, instead the new thread executes and reports its thread creation event
Fix for New Thread Race

• This is a race between the callback that resumes execution and the thread reporting its birth

• One fix: use PT_GETLWPLIST each time the resume callback is invoked to search for new, but not-yet-announced, threads so they can be suspended via PT_SUSPEND
  • Would add at least two additional ptrace() calls for each resume

• Second fix: “remember” that GDB is executing a single thread from a process (rather than all threads from a process) and defer thread birth events
  • Only adds overhead in the uncommon case
Fix Details

• GDB’s resume callback takes a few arguments: a ptid_t identifying the process/thread to resume, whether to step vs continue, and signal to deliver (if any)
• ptid_t can either be an entire process, a single LWP, or a wildcard meaning all processes
• The fix saves the value of this ptid_t in a global used in wrapper around wait() system call
• If a new event has a LWP ID not matching the ptid_t from resume, PT_SUSPEND the LWP and PT_CONTINU process to get next event
Down the Rabbit Hole...

- As part of the fix, added various assertions to document my assumptions
- Ran GDB’s test suite and new assertion failures confirmed false assumptions on my part
- First false assumption: The resume callback is only called once before each call to wait
  - I had asserted that the new global variable wasn’t set multiple times
- Actual truth: The resume callback can be called for multiple processes before calling wait
Multiple Processes for Real

- This assumption exposed broader false assumptions by myself about how multiple process support worked in GDB
  - I’m not sure exactly what my old model really was, but it was wrong
- The real model is that GDB will resume one or more processes/threads before doing a wait
  - If the ptid_t passed to resume is the wildcard, all of the currently debugged processes should be resumed, not just the “current” one
- When a process stops to report an event, GDB expects all the other currently running processes to also stop
  - GDB calls this “all-stop” mode
Fixes for Multiple Processes

• Instead of a global copy of the resume `ptid_t`, store a copy for each active process
  • ”inferior” in GDB parlance
• If resume is invoked with the wildcard `ptid_t`, iterate over all active processes resuming each one
• Added a new helper function to stop a process
  • Tries `wait()` with `W_NOHANG` first in case it is already stopped
  • Otherwise, send `SIGSTOP` and `wait()` for an event from the process
  • If the event isn’t the `SIGSTOP`, remember to ignore the next `SIGSTOP` for this process
Fixes for Multiple Processes

- When waiting for an event, first check for any previously deferred events that are now eligible to be reported.
- If there is no pending event, call `wait()` to get an event.
- If the event returned from `wait()` is for a thread that shouldn’t be running, defer it and call `wait()` again.
- Once there is an event to return to the caller, iterate over all the active processes and stop them if they are running via the new helper.
- Added lots more assertions to document assumptions.
Farther Down the Hole...

• New assertions found more incorrect assumptions
• Specifically, two other callbacks in the native target can be called on a process that is still running: detach and kill
• Extra wrinkles to fix for these cases
  • If the process to detach/kill has a thread with a pending fork event, the kernel has already attached to the child and GDB needs to detach from the child
  • If the process has active breakpoints during detach, need to clear them before detach
  • If the process to detach has a thread with a pending breakpoint event, need to fixup PC for the thread before detaching
Dealing With the Wrinkles

• Having to drain certain types of events (pending signal such as SIGSTOP, SIGTRAP for some debugger event like a breakpoint hit, fork events) during detach/kill

• ptrace() can only discard a signal for the current reporting thread for PT_CONTINUE
  • Can’t just use a single loop to clear any pending events in the process
  • Instead, have to scan for anything pending in other threads and PT_CONTINUE + wait() to clear the next event
  • Keep looping until no threads have any pending events
Found a Bug

• While testing the detach fixes, found a bug (my fault) with PT_LWP_EVENTS

• PT_DETACH doesn’t clear the flag (TDB_BORN) used to instruct a new thread to report SIGTRAP before its first instruction
  • If you detach in this state, the thread delivers the SIGTRAP after PT_DETACH and promptly dies

• Fine-tuning and verifying the fix: https://reviews.freebsd.org/D39856
Deeper Still...

• At this point, the GDB test suite is now only raising a single new assertion failure
• But it’s a doozy: in one test GDB is resuming two specific threads (but not others) from a single process
• Can even do this from the command line in GDB with scheduler locking and use of continue&
Multiple Resumes for a Process

• Can no longer trigger PT_CONTINUE/PT_STEP from resume callback since there might be multiple callbacks for a single process
• Instead, track set of resumed LWPs for each process along with other “pending resume” state like a stepping LWP and pending signal
• Each call to the resume callback updates the pending resume state for the process
• At start of wait callback, iterate over processes to resume them via PT_CONTINUE/PT_STEP
Issues Observed with ptrace()

- There are still many bugs to work through in GDB’s test suite, but from this recent work I’ve encountered some limitations in FreeBSD’s ptrace()
  - GDB wants to at least read (and possibly even write?) to process memory while threads are running
    - Reading is racy, sure, but for reading this shouldn’t be too hard to fix
  - Need a way to keep a deferred signal deferred until the thread is really resumed
    - Allow signal to remain pending while in PT_SUSPEND state and only clear it/deliver it when actually resumed
Issues Observed with ptrace()

• Would really like a way to drain multiple events from a process while it is stopped
  • This would simplify the detach/kill handling while also avoiding a loop that can in theory never make progress

• LWP create events are not like fork
  • Fork reports events for both parent and child, and Linux threads do the same
  • Current approach means you can have an “empty” process if thread A creates thread B and then exits and you get thread B’s exit event before thread A reports its birth
Current Status

• Fix for TDB_BORN bug will land soon

• Patches for GDB have gone through some review and I’m still refining them (in particular the patch to batch up resume requests is still a WIP)
  • [https://github.com/bsdjhb/gdb/compare/master...defer_resume](https://github.com/bsdjhb/gdb/compare/master...defer_resume)
Future Work

• Have some old work (need to rebase and retest) to replace p_xthread race in the kernel with an explicit linked-list of threads with events to report
  • Interacts poorly with some tests added a few years ago that I still need to work out
• On top of the thread queue is a less-polished patch that tries to avoid spurious EINTR for deferred signals
  • Goal is to leave threads asleep in the kernel while a signal is deferred to the debugger
Future Work

• Fixes for some of the issues raised earlier
  • Keeping a deferred signal deferred while PT_SUSPEND
  • A way to ack the current event and fetch the next one without PT_CONTINUE
  • LWP create event reported by the creating thread

• Single stepping and signal handlers
  • Linux steps into signal handlers, FreeBSD steps over
  • Could add new mode that raises SIGTRAP at start of signal handler
  • Need to PT_CLEARSTEP after return from signal handler
Future Work

• More GDB test suite failure chasing

• Non-stop mode?
  • Would not stop the entire process when an event is reported, just the reporting thread
  • Probably depends on the thread queue patches
  • Use thr_kill2() to send SIGSTOP to individual LWPs
Questions?