GDB Stub Documentation

Code Structure

Gdb.h/Gdb.cc – static class serving as an interface to gdb stub mechanisms.
GdbCpu.h/GdbCpu.cc – stores register values for each CPU in gdb session
gdb_asm_functions.h/gdb_asm_functions.S – exception handlers for exception #1, #3 and faulty exceptions #12, #13, #14; fault-safe memory read/write functions (get_char/set_char)
NonBlockSemaphore.h – spinning version of semaphore
VContAction.h – vCont packet representation
x86_64-stub.cc – gdb stub implementation for x86-64 machines

Overview

Gdb stub provides a hook to connect to gdb frontend during kernel initialization. Gdb::start() first installs gdb stub's exception handlers for exceptions #1-#7, #12-#14, #16.
Exception #1, #3 are important for Gdb to work. Then, Gdb::start() calls breakpoint() which does "int3" assembly instruction to force #3 exception. The installed #3 exception handler directs code to gdb stub in x86_64-stub.cc.

Communication Protocol

Gdb stub communicates to gdb frontend over a serial line to send requests and receive replies. Replies include request status and data if requests asked for register values or memory address values, etc.
The protocol descriptions can be found at http://sourceware.org/gdb/onlinedocs/gdb/Packets.html

Currently, gdb stub does not support all of the packet types listed in above webpage, but only those that are needed for supported features. Future developers should be aware of this and add support as more features are added in the future.

I have extracted protocol parsing part into functions for each type of packet for easier understanding and cleanness of code. Below are descriptions for each packet type imlpemented:

Queries
qSupported – currently specifies maximum packet size that the Gdb stub supports
qC – returns current thread ID
qAttached – specifies that the gdb stub process is not attached to an existing process
qOffsets – specifies that the target offset for text/data/bss sections are 0
qSymbol – ask for _Unwind_DebugHook's address
qSymbolLookup – parse _Unwind_DebugHook's address; tells gdb frontend symbol lookup is done
qTStatus – tells gdb frontend that there's no trace experiment running
qfThreadInfo – tells gdb frontend list of all active thread IDs
qThreadExtraInfo – tells printable string descriptions of each thread to be used by 'info threads'

Others
T – tells if a thread is alive
H – tells for which thread the subsequent operations ('m','M','g','G',etc) are directed to
? - tells why the thread stopped in gdb session

g – read all of general registers for specified thread
G – write all of general registers for specified thread
P – write a register to specified thread
m – read specified bytes from a given memory address
M – write specified bytes to a given memory address
Z0 – set a software breakpoint
z0 – remove a software breakpoint
k – kill the target process

Multi-threaded Gdb stub

The gdb stub supports multiple CPU cores in the system KOS is running. Each core registers itself to gdb stub during booting (initAP).
Baton passing is used to restrict only one thread to communicate with gdb frontend at a time. It will be the core the user is interested in debugging at the time, and if there are other cores that hit breakpoints, they will be spinning, trying to get a baton.
A core can pick up a baton only if there are not cores holding it, or the current holder chose to pass the baton to it before returning to kernel.

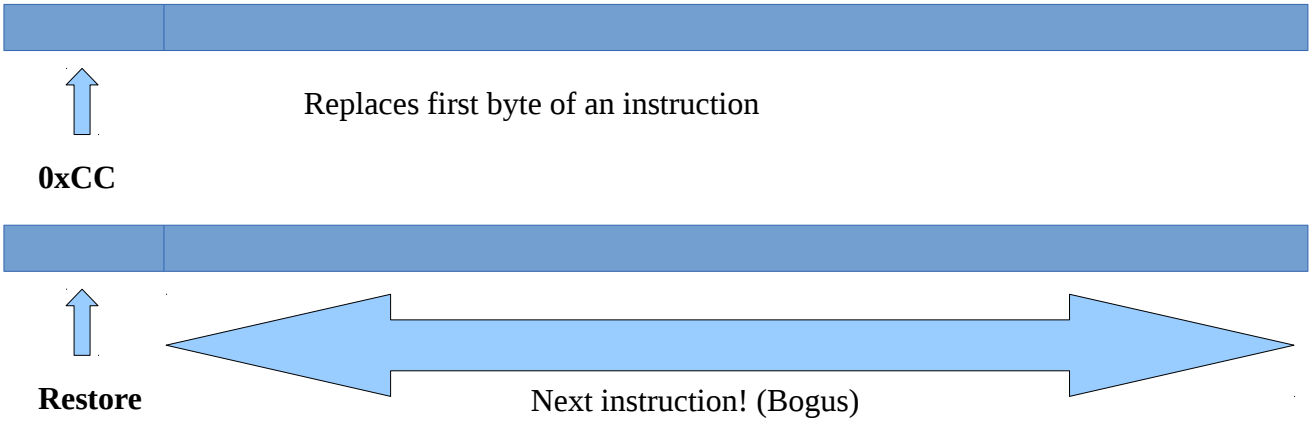
Baton passing is described in CS343 textbook.
https://www.student.cs.uwaterloo.ca/~cs343/documents/notes.pdf

Here is the sequence of actions a thread does when it enters the exception handler:

1. Enable preemption for the core if previously disabled
2. Set core status as in BREAKPOINT
3. If exception #3 caused it and TF is set to 0 (gdb frontend inserted breakpoint from gdbinit file), decrement instruction pointer (rip) to point to correct instruction. (See RIP manipulation)
4. Grab a entry_q semaphore, which provides a mutual exclusion for baton grabbing procedure
5. If a baton is grabbed by someone else, set it's waiting flag to true, release entry_q and sleep on its private semaphore. Each core has a private semaphore and with entry_q, they form "split binary semaphore". (Refer to CS343 text)
6. If a core is woken up from its private semaphore, check if it should return to kernel. This will happen if gdb session is in all-stop mode, and all threads are required to leave gdb session. Otherwise, you are passed a baton and now free to talk to gdb frontend!
7. Now you own a baton. Next step is to check if any of vCont packets are executed by you or previous baton holder. (See vCont packet) In that case, you need to generate proper stop reply to let gdb frontend know that its vCont packet is executed. A core executing vCont packet and a core replying stop reply may be different because vCont packets usually ask for stepping/next/continue, which makes the executor return to the kernel.
8. Cause all other cores to break if in all-stop mode. (Get it? ALL STOP :))
9. Now, entry protocol is done. Release entry_q semaphore.
10. Start communicating to gdb frontend.

RIP manipulation

As far as I know, gdb frontend secretly replaces first byte of an instruction with a breakpoint with 0xCC (int3). As 0xCC is an instruction by itself, rest of the instruction is temporarily regarded as a separate instruction. When a breakpoint is hit, gdb frontend replaces back the original byte of the instruction; however, since #3 is a TRAP, rip is set to the next instruction, which is a bogus instruction.



Therefore, gdb stub decrements the RIP by 1 to point to correct start of the original instruction. Otherwise, you may experience running non-existent bogus instruction on gdb return.

vCont Packet

vCont packets are newer way of serving step/next/continue commands that supports multi-threadedness. vCont packets are handled as following in gdb stub.

vContAction.h contains a struct representing vCont packets. It specifies, thread id, action and executed status of each vCont packet. Three types of vCont packets that I have come across are:

vCont;s
vCont;s:threadId;c
vCont;c

's' tells gdb stub to execute a single-step. Gdb stub currently sets TF flag of the appropriate eflags register for the target core before returning to kernel. It triggers exception #1 on next instruction. 'c' tells gdb stub to return to the kernel without single-stepping. For next, gdb sets a breakpoint to an instruction that gdb should break before executing 'c'. For continue, gdb will not set a breakpoint; therefore, the core will run freely until it hits a next breakpoint.

Since the nature of these packets involve returning to kernel and reporting back the result later, gdb stub queues each request so that next thread that holds a baton can reply the status of vCont packet for it. It is in fact necessary. For example, for continue, if other thread does not send stop reply, gdb won't know if continue is execute successfully or not.

Preemption

Gdb stub disables a preemption for a core in gdb session for stepping or next. It is disabled because for those commands, the core should return to the gdb session in short while. With preemption enabled, a thread may migrate to a different core and cause exception #1 (for breakpoints), which will show 'Trap detected on a thread ....'. By only enabling preemption for continue, we do not let the awkward warning (even though it is harmless) to appear.

To correctly distinguish between step,next and continue, gdb stub relies on a fact that _Unwind_DebugHook is secretly set on a break by gdb frontend for every step or next command.

Currently, there seems to be no definite way of distinguishing them from parsing messages between gdb frontend and gdb stub. Therfore, if gdb stub sees a break has been set to _Unwind_DebugHook, it knows that vCont packet is not for ‘continue’ but ‘step or next’.

Note: Even ‘step’ may see ‘c’ vCont packets because vCont packets has assembly instruction granularity. Even a single step may consist of several assembly instructions which may require ‘c’ vCont packets.

Note: This mechanism fails if you do ‘next’ on the very last instruction of KOS because kernel does not return to gdb session anymore. (Preemption is turned off, but you don’t return to gdb session to re-enable it) You need to do ‘continue’ if you are at the very last instruction of kernel if you want preemption to be not turned off.

Software Breakpoints

Gdb stub uses software breakpoints to support limitless number of breakpoints. You can tell gdb frontend to set a breakpoint or you can tell gdb frontend that you will set a breakpoint by yourself. You simply gain more control.

As an extension, you can implement hardware breakpoints, and watchpoints in similar manner. In fact they are all in group of ‘Z’ packets.

All Stop Mode

All-stop mode is regular gdb session we know. All threads are stopped on a breakpoint, and all are resumed when a thread returns to its source. In gdb stub, we force this by sending IPI (inter-processor interrupt) to other cores, if any core grabs a baton in gdb stub. It forces other cores to stop and gdb session will have full information about all cores. It enables you to switch between cores during gdb session in more controlled way.