# NON-STOP MULTI-THREADED DEBUGGING IN GDB

NATHAN SIDWELL, VLADIMIR PRUS, PEDRO ALVES,
SANDRA LOOSEMORE, MENTOR GRAPHICS

JIM BLANDY, RED-BEAN.COM

**Mentor Graphics**®

W H I T E P A P E R

E M B E D D E D   S O F T W A R E

# Non-stop Multi-Threaded Debugging in GDB

Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore

*CodeSourcery Inc*

{nathan,vladimir,pedro,sandra}@codesourcery.com

Jim Blandy

jimb@red-bean.com

## Abstract

When debugging a multi-threaded program, should a debugger stop all threads when any thread stops, or should it stop only those threads that have something to report, like a breakpoint hit? When debugging a live system, the latter approach may be less intrusive, as threads other than those under inspection can continue to respond to external events. We have implemented this behavior, which we call *non-stop* debugging, in the GNU Debugger, GDB. We have adapted GDB's control commands, strengthened GDB's event loop, extended GDB's remote protocol, and implemented new techniques for inserting, removing, and stepping past breakpoints.

This project lifts long-standing restrictions in GDB's thread support, using a number of interesting techniques. It opens a way to supporting multi-process and multi-core debugging.

## 1 Background

GDB[1] is the debugger of choice for the GNU project[2]. It started life as a debugger for single-threaded programs. In its original model, when debugging a target program, either the target program is running and GDB waits on it, or GDB is running (or waiting for user input) and the target program is halted. This model simplifies control of the target program, because GDB does not have to contend with target state changes except at the well-defined point of having the target run.

As multi-threaded applications arrived, GDB was extended to debug them. The first implementation of threading support was for LynxOS in 1993. The changes made were straightforward: GDB's program state was augmented with a thread identifier, and commands added to switch user control between available threads. What was not changed was the underlying run control model in which either GDB or the target is running. When GDB examines the state of any particular thread, all other threads are also stopped.

Later, in 1999, additional run control was added so that, when continuing the thread of interest, one can select whether other threads are held stopped, or whether they continue too. The fundamental restriction that the target is stopped when GDB has control remained unchanged. To distinguish this existing behavior from the new functionality we have implemented, we have coined the term *all-stop* for it.

For some debugging uses, GDB's all-stop behavior is desired. The relative schedules of independent threads is disturbed as little as possible — by the simple expedient of stopping all of them. As discussed below the schedules are not completely undisturbed though. In other debugging uses, the all-stop behavior is an undesirable perturbation to the target program — completely separate threads have their scheduling interrupted. Users desire the ability to debug live systems with minimal intrusion. In particular, when parts of a multi-threaded program have real-time constraints it becomes impossible to debug even the non-real-time threads because the system is too badly disturbed.

To address this problem, we have implemented a *non-stop* debugging mode in GDB. In this mode, only the thread of interest is held stopped, whilst its state is being manipulated by the user. Other threads continue execution until they encounter an event that GDB needs to be aware of. Implementing this required changes to:

- Breakpoint management
- Single stepping
- Event loop
- Serial protocol
- MI interface

These changes, in addition to allowing non-stop debugging of a single multi-threaded program, enable other modes of operation, which we discuss in Section 10.

Our ultimate target of interest is a 32-bit x86 system, accessed remotely via the GDB serial protocol. We implemented non-stop debugging for native 32-bit x86 Linux systems in addition to extending the serial protocol for remote debugging. The `gdbserver` program has not yet been updated.

## 2  Run Control

In order to debug a target system, GDB requires very few fundamental operations:

- Read and write registers

- Read and write memory

- Single step target

- Run target

For the purposes of this discussion, the latter two are the important parts of *run control*. When the target completes a single step, or stops after running freely, it reports an event to GDB. These events let GDB know that the target has stopped because of some condition – single step complete, breakpoint hit, segmentation fault. The events are the target-side responses that GDB processes for run control.

This level of run control is sufficient to implement `stepi`, `continue` and `break` commands directly. More advanced commands are synthesized from these basic operations in a multi-action sequence. For instance the `nexti` command records the current frame identification, performs a single step and then determines the now-current frame identification.[1] If the frames are different, a procedure has been entered. GDB then determines the return location (by target-dependent means) and inserts a temporary breakpoint at that location. Then it runs the target, which stops when a breakpoint is hit. If the temporary breakpoint caused the stop,

---

[1]To avoid confusion between the GDB `step` command and the target-side single step action, we refer to the former with 'step' and the latter with 'single step'. Similarly, we distinguish GDB's `continue` command and the target-side run action.

the `nexti` command has completed. If another breakpoint is hit (because the called routine hit a user-inserted breakpoint), the `nexti` command is abandoned. In both cases the temporary breakpoint is removed.

GDB can even debug targets that cannot single step. It does this by inserting temporary breakpoints at the next instruction after the one being single stepped. Determining the next instruction requires target-specific code to decode flow control instructions. We do not discuss this kind of target further.

GDB uses internal breakpoints to detect certain actions such as `longjmp`, exception handling, dynamic loading and, as described below, threading operations. The `longjmp` and exception handling detection is necessary to catch when `next` skips a routine that calls `longjmp` or throws an exception — it would be unpleasant for the `next` command not to terminate.

## 3  Threading

To support threading, GDB has to interact with the threading library on the target. Additional operations required are:

- List available threads

- Select active thread

- Single step one thread

- Run one thread

- Run all threads

GDB's serial protocol can describe more flexible run control than that described here, but the extra functionality is not used by GDB at present.

The mechanism GDB uses to detect thread creation and destruction events is specific to the target system. For instance, with Linux the target must use a special threading library with debug capabilities, `thread_db`. GDB uses an internal breakpoint inserted at a special location in the threading library. When this breakpoint is hit, GDB calls a special function provided by the threading library to retrieve an event message.

Run control is made more complex because of the issue of which threads are single stepped or run. This is controlled GDB's *scheduler locking* parameter. Its values are:

off All threads are run whenever the thread of interest is single stepped or run.

on Only the thread of interest is single stepped or run. Other threads remain stopped.

step Single stepping only steps the thread of interest. All threads are run whenever the thread of interest is run.

Note that scheduler locking supports more than the minimum additional target run control actions described above. Some targets cannot offer all the variations that scheduler locking allows.

GDB commands that perform a series of single steps or runs at the target level may terminate early if another thread hits a breakpoint. This can only happen when scheduler locking is off or step. For instance, for a next operation when a procedure is entered, GDB inserts a temporary breakpoint at the return location and runs the target. During that time, other threads run, unless scheduler locking is on. Should one of them hit a breakpoint before the thread whose procedure is being skipped past finishes, that other breakpoint is reported. The temporary breakpoint remains active and the next operation completes once it is hit. If another breakpoint is hit in the thread that GDB is performing the next operation for, then the temporary breakpoint is removed and the next operation abandoned, just as it is in the single-threaded case described above. Thus GDB keeps state describing each thread's outstanding multiple-action operation, and processes them independently.

There is one action where exactly one thread must be single stepped, and that is when continuing from a breakpoint. Here the original instruction must be written back and the target single stepped. Once the breakpointed instruction has been stepped over, the breakpoint can be reinserted. Clearly, when the breakpoint is not inserted, other threads must not execute because they may be executing the same code and consequently not hit the breakpoint the user placed there.

## 4 GDB Implementation Interfaces

GDB abstracts the architecture and system-specific features of the target in an *architecture structure*. This contains numerous hooks to handle stack unwinding, specify register types and contents, determine breakpoint

formats and the like. Additional hooks must be added to support the new stepping mechanism described below.

The type of architecture and target operating system is determined when GDB is configured and built. For example, it is not possible to use a GDB built for ARM-Linux and use it to debug a program running on some other ARM operating system.[2]

GDB abstracts the interface to the target with a *target control* stack. This allows a single GDB to debug a program that is running on local hardware, remote hardware, internal or external simulator, or even a post-mortem analysis of a core file.

Two target control instances are relevant for this discussion. One allows native debugging of another process running on the same x86-Linux system. It uses the ptrace[4] interface. The other allows debugging of a process running on a remote system. It uses the GDB serial protocol, over which transactions are serialized. A process on the remote system translates the serial protocol into the appropriate run control actions for that system. In the case of debugging a remote x86-Linux system, this is done by gdbserver, which also uses the ptrace system call, but on the remote system. Other target systems may embed the GDB protocol engine directly into the kernel. A simulator may implement the GDB protocol as a mechanism for accessing the simulated program, for instance QEMU[5].

## 5 Breakpoint Persistence

In the traditional all-stop run control model, when GDB has stopped a target program, it removes all the breakpoints from the target. This allows simplified memory access as there is no need to consider the backing store for patched breakpoints. It also simplifies single stepping of the target as there is no need to determine if a breakpointed location is being single stepped. Clearly, for non-stop debugging, all breakpoints must be inserted all the time. Otherwise other threads would not stop when they hit a breakpoint location.

### 5.1 Breakpoint Insertion and Removal

When breakpoints are permanently inserted, it is necessary for the breakpoint management commands, such

---

[2]There is some work to allow GDB to debug multiple target types, but this is not generalized for all possible target types.

as `break`, `delete`, `enable` and `disable` to be active immediately. Special care must be taken with multiple breakpoints at the same location. If one of them is disabled or deleted, GDB cannot even momentarily remove the breakpoint. GDB's breakpoint code was formerly distributed in several places. We redesigned breakpoint management to use a single new function, `update_global_location_list`, that is called for all breakpoint creation, destruction or altering. The function computes the change set and then applies it in a single operation. This redesign even fixed some breakpoint bugs in all-stop mode.

If breakpoints are inserted when GDB has control, care must be taken when reading memory locations. It is necessary to have all memory accesses go via routines that maintain patched breakpoint backing storage, and allow it to do the necessary caching. GDB has such a routine, `target_xfer_partial`, and it is used for nearly all memory accesses. Self-modifying code presents a problem with breakpoint removal, as the breakpoint location may have been overwritten by the program itself. GDB does not usually check for this situation, because it is rare, and the performance impact of rereading each breakpoint location before restoring the original contents is too high. Unfortunately both PowerPC and MIPS have writable PLTs, where the PLT slot for a function is rewritten on first use. When setting a breakpoint on an unresolved function reached via a PLT, GDB places a breakpoint in the PLT itself. Such a breakpoint is lost if the location is rewritten by the dynamic loader. A `make_show_memory_breakpoints_cleanup` function was added, which makes memory reads show real underlying memory until the cleanup is run. This allows GDB to check for overwritten breakpoints in places it expects they may occur. Of course, if the breakpoint is rewritten this way, GDB will fail to stop at that location. This is a preexisting problem with self-modifying code, not specific to non-stop mode.

As thread events are queued and can be reported to GDB some time later (see Section 8), there is a complication with removing breakpoints. A thread may have hit a breakpoint and queued that event, but GDB might not have responded to it. If the breakpoint is deleted, GDB will become confused when the breakpoint stop event is finally processed. Without any record of the breakpoint, the event is indistinguishable from a spurious trap, a situation likely to confuse users. To resolve this, GDB must retain a record of breakpoints that have been deleted. By consulting that table GDB can determine if a thread event can be ignored (or reported as a moribund breakpoint). GDB has to keep such a record until it can be certain that no thread can have queued a trap for that breakpoint. GDB does not need to remember such locations indefinitely. Because threads can only have one outstanding event queued (see Section 8), once a thread has reported an event, it cannot subsequently report an event related to the removed breakpoint. Thus, to determine when to delete a record of the breakpoint it is sufficient to note that all threads that were active at the time the breakpoint was removed have reported events.

Unfortunately, it is quite possible in practice that one or more threads never stop, because they are not involved in the problem being debugged. Such threads would cause the moribund breakpoint list to grow indefinitely. Instead a retirement heuristic is implemented for retiring breakpoints once a certain number of events have been reported, regardless of which threads reported them. The event limit is set from the number of active events at the time the breakpoint is deleted.

## 5.2 Stepping Over Breakpoints

A much harder problem is single stepping past a breakpoint. In all-stop mode, when a breakpoint is hit, GDB performs the following sequence of operations to continue execution:

- Remove the breakpoint being stepped. This is done implicitly because all breakpoints are removed when GDB regains control.

- Single step the target. This executes the instruction at the breakpoint location and advances the PC past it.

- Insert the breakpoint that was just stepped past. This is done implicitly, as GDB reinserts all breakpoints just before running the target.

- Run the target.

With non-stop mode there is a conflict. The breakpoint being stepped past cannot be removed, otherwise another thread executing the same code will fail to hit the breakpoint. Clearly the breakpointed location cannot be single stepped if the breakpoint has not been removed for that would execute the wrong instruction.

Solutions to this conflict are either to simulate the stepped instruction's behavior inside the debugger, or to single step a displaced copy of the breakpointed instruction.

The simulation approach requires an instruction set simulator be available. GDB has many such simulators, so this would be practical for a wide range of GDB targets. However not all CPUs that GDB supports have simulators, and hooking in a simulator to operate alongside an already executing target may be tricky. The simulator would need to fetch the target's registers and access target memory. Most tricky would generating exceptions such as page faults — many simulators simply do not generate these conditions.

The displaced stepping technique requires the location of a safe area of target memory in which to place the copied instruction. Also GDB must be careful with instructions that manipulate or use the program counter, because it does not have the expected value during stepping.

For the target of immediate interest (x86) we chose the second technique, which is essentially the same as the Linux kernel's kprobes interface.[6]

As the implementation details are architecture-specific, we added several new hooks to the architecture structure. These are:

- `gdbarch_max_insn_length` and `gdbarch_displaced_step_location` determine where an instruction to be stepped may be copied for stepping.

- `gdbarch_displaced_step_copy_insn` is used to copy the instruction. It may need to adjust the instruction, register contents, or memory as detailed below.

- `gdbarch_displaced_step_fixup` is used after the instruction has been single stepped. It adjusts registers and memory to yield the effects the instruction would have had if it had been stepped in situ.

- `gdbarch_displaced_step_free_closure` is a cleanup to free any resources allocated during stepping by the other hooks.

These hooks show a bias towards a kprobes-like implementation, but would be suitable for a simulation-type implementation. However the existing patches to GDB do not fully support that scheme yet. For instance it presumes that `gdbarch_displaced_step_location` is not `NULL`, which would not be the case in simulation.

A suitable place to copy the instruction is often the program's entry point (usually `_start`). This location is already used by GDB to place a breakpoint when it executes a target routine as part of expression evaluation. There is a `displaced_step_at_entry_point` hook implementation to provide this common need. The x86-Linux implementation does this.

As the x86 architecture has no PC-relative data access instructions, the action of copying the instruction and applying any fixup before stepping is simply a matter of copying the instruction. A `simple_displaced_step_copy_insn` hook implementation is provided to do this, and the x86-Linux implementation uses it. For an instruction that uses PC-relative data access this hook could use one of the following techniques:

- Rewrite the instruction to adjust PC-relative displacement by the distance between the displaced location and the original location.

- Calculate the effective address and rewrite the instruction to use an absolute address.

- Calculate the effective address and save it in a register (which itself would have to be preserved). Then rewrite the instruction to use an indirect addressing mode.

- Decode the instruction and emulate it within GDB.

For x86 the complexity of displaced stepping is in the fixup hook:

- First the instruction pointer, `%eip`, must be corrected. For non-absolute jump, call and return instructions, the instruction pointer is relative to the displaced instruction, so `%eip` must be adjusted by the distance between the displaced location and the original location.

There is one special case with system calls. Normally system calls behave as any other non-branch

5

instruction and leave %eip at the next instruction. However a signal trampoline system call leaves %eip somewhere else. Rather than embed knowledge of the system call number into GDB, this condition is detected dynamically. After displaced stepping a system call instruction, GDB examines %eip and sees if it is at the next instruction. If it is, GDB adjusts it as above. If it is not, it is presumed such a signal trampoline occurred and %eip is not adjusted.

- If the stepped instruction is a call, the return address that has been pushed onto the stack is incorrect. It too needs adjusting by the distance between the displaced location and the original location.

- If the displaced stepped instruction raises an exception, the %eip reported is correct after the above adjustments. If the exceptional %eip is written into the signal address, GDB needs to correct that too. Illegal instructions which raise SIGILL have this issue.

For an architecture that rewrites displaced instructions, the fixup hook may need to perform additional PC adjustment, particularly on targets with a variable-length instruction encoding.

The current implementation is limited to x86-32 instructions that may be generated by GCC. Some sequences of x86 instructions are not correctly recognized, such as addr16 call *(%di), which forces 16-bit address calculation, or repz call *%eax, which is an undefined code sequence. Neither of these cases are recognized by GDB's displaced stepping and so no adjustment to the pushed return address occurs and the final %eip value is incorrectly adjusted. Also x86-64 targets are not supported.

### 5.3 Hardware Breakpoints and Watchpoints

Depending on the target architecture, hardware breakpoints may be reported in different ways, presenting a number of different scenarios with different solutions:

- The hardware breakpoint is reported after the instruction has executed. There is no need to single step the breakpointed instruction for it has already been performed. We are not aware of any hardware that behaves in this manner, and it is not supported.

- The hardware breakpoint is reported before the instruction executes, but it is permitted to single step an instruction at a hardware breakpoint. There is no need to perform displaced single stepping in this case, as the instruction can be stepped in situ. We are not aware of hardware that behaves in this manner, and GDB does not support it.

- The hardware breakpoint is reported before the instruction executes and it is possible to remove the hardware watchpoint from exactly one thread. In this case GDB can simply remove the watchpoint while stepping the thread past it. This does not occur with x86 and is not implemented.

- The hardware breakpoint is reported before the instruction executes and it is not possible to single step an instruction at a hardware breakpoint. This situation is exactly the same as the patched breakpoint case, and is solved by performing a displaced single step. This occurs with x86 systems and is implemented.

Running or single stepping the target after hitting a hardware watchpoint is performed in different ways, depending on the target hardware:

- Hardware watchpoints are reported after the instruction executes. There is no issue with running or single stepping from that point. Watchpoints on x86 hardware behave in this manner.

- Hardware watchpoints are reported before the instruction executes, but a single step of that instruction does not report the watchpoint. It is possible to single step the watched instruction in situ. Watchpoints on IA64 behave in this manner. Although not explicitly tested, we believe this works.

- Hardware watchpoints are reported before the instruction executes and single stepping the instruction retriggers the watchpoint (without performing the single step). Such a target cannot be continued without removing the watchpoint. PowerPC watchpoints behave in this manner.

  If hardware watchpoints are thread-specific, they can temporarily be removed for the thread that needs to be single stepped while performing the step operation. Other threads can continue running without danger of missing a watchpoint.
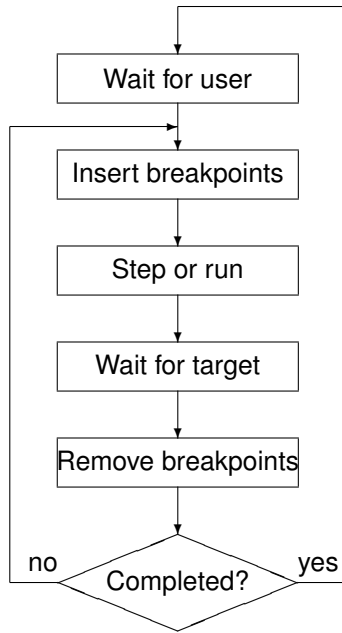
Figure 1: Synchronous Main Loop



Figure 2: Asynchronous Main Loop

If, however, watchpoints are global to the process, then either non-stop mode must be abandoned, one must accept that other threads may fail to report the same watchpoint during the single step operation. Furthermore, if several hardware watchpoints are inserted, it might be difficult or impossible to determine which one actually triggered. So all watchpoints might need to be removed during the single step operation.

GDB's current implementation removes all breakpoints and watchpoints in this case, and work is needed to improve this.

Finally, a hardware watchpoint may trigger while executing a displaced single step for an inserted breakpoint. This presents the same issues as for running or single stepping after hitting a hardware watchpoint with the additional complexity of adjusting the program counter reported to the user. The adjustment is the same as for when an exception occurs during such a displaced single step.

## 6  Event Loop

GDB's all-stop model event loop is conceptually very simple. As has been mentioned, in this model either GDB is active, or the target is active. GDB's main loop
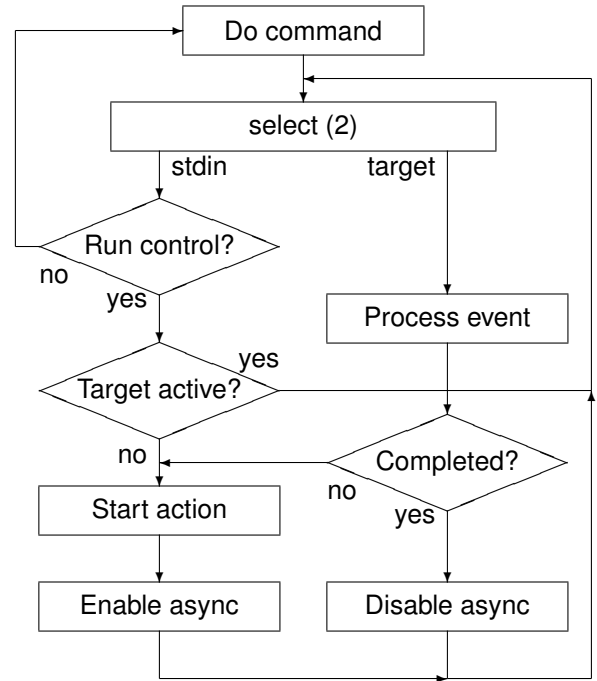
is shown in Figure 1. The point at which the target can report an event is well defined — it is only after GDB has started a target thread. Clearly, with non-stop debugging, this picture is no longer true. When threads are always active, a thread may want to report an event at any time. Allowing that to happen would be severely disruptive to GDB's internal design. Furthermore, as discussed in Section 8, this would cause difficulty with GDB's serial protocol.

To implement a non-stop event loop, we took a patch that had been developed by Nick Roberts[7]. This patch implements an asynchronous target mode, called *target async*. This is different from the original proposal posted to the GDB mailing list.[8].

Async mode changes the event loop to that shown in Figure 2 so that it waits in exactly one place. This behavior is achieved by adding a `target_async` hook to the target stack. This hook should register a file descriptor and callback function with GDB's `select(2)` logic. If successful, GDB then waits on the input file descriptors for an action. If unsuccessful (because the target does not support asynchronous behavior), GDB reverts to the original behavior of waiting for the target explicitly. If the target file descriptor is active, the target event is processed, which may lead to a continuation of
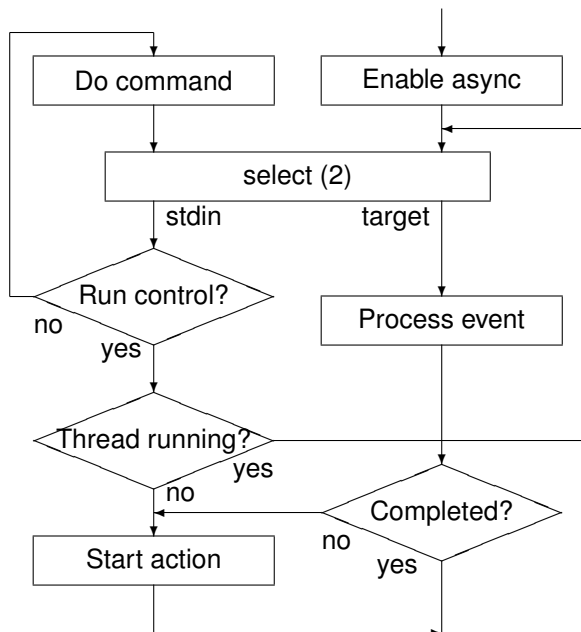
Figure 3: Non-stop Main Loop

run control, or may terminate the pending operation and remove the target file descriptor from the select table.

This modified loop allows exactly one run control operation to be outstanding and permits manipulation of some of GDB's state while waiting for the operation to complete. It is not possible to examine target state while a thread is running. The target file descriptor is transitory and removed once the operation has completed.

The target file descriptor may be the file descriptor for the serial protocol, if remote debugging is in use. For native debugging a pipe has to be created, as described in Section 7.

For non-stop debugging the asynchronous main loop was modified to that shown in Figure 3. With this new loop, the pending operations are recorded in per-thread state. In particular, GDB's continuation processing, which determines whether a compound operation such as `next` is completed, needed to be made per-thread.

As with the all-stop behavior with scheduler locking `off`, if a multi-action sequence is interrupted by an orthogonal event in a separate thread, the multi-action sequence is remembered as part of that thread's state and the orthogonal event is reported.

## 7 Native Debugging

For native Linux debugging, GDB uses the `ptrace` interface. Whenever a signal is to be delivered to the program being debugged, the program is stopped and GDB is informed when it `waits`. GDB can process the signal and determine whether or not to continue delivery to the child. Normally breakpoints, single step completion and hardware watchpoints cause a `SIGTRAP`, which GDB processes and does not pass on to the debugged program.

This behavior is implemented in the native Linux target stack. Whenever GDB runs or single steps the program, a `PTRACE_CONT` or `PTRACE_SINGLESTEP` is performed, followed by a `wait`.

For non-stop debugging a number of changes must be made:

- GDB cannot `wait` for the target to stop.

- GDB's event loop must be informed, via a file descriptor, that an event of interest has occurred.

- Other threads in the debugged program must be continued as soon as possible.

Whenever the debugged program is to be sent a signal, GDB receives a `SIGCHILD` signal. GDB installs a handler for that signal to support Linux native async mode. When the handler runs, there are one or more child events of interest. It writes to a pipe that has been registered with the main event loop. The main event loop will see this activity and invoke the callback associated with that file descriptor.

## 8 Serial Protocol

GDB's serial protocol provides a mechanism to attach GDB to a remote system. The serial protocol is synchronous, and it is well defined as to whether GDB or the target system is to send the next message. Normally the target is waiting for GDB to send it a request, to which it replies. When GDB sends it a run (or single step) request, the target starts running and GDB waits for a reply. When the target hits a breakpoint, completes the single step or encounters some other trap event, the target sends a stop reply back to GDB. At this point all
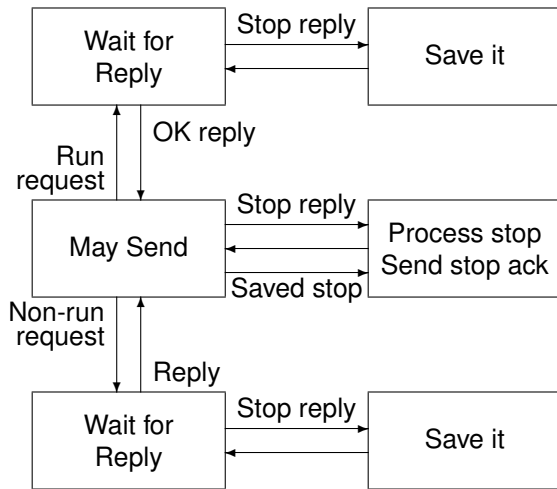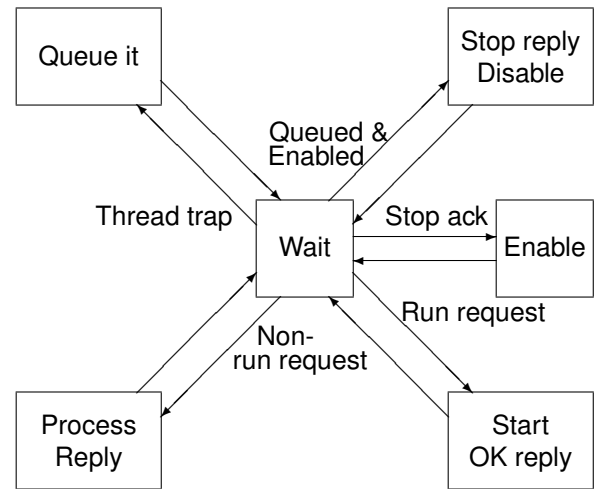
Figure 4: Serial Protocol: GDB Engine



Figure 5: Serial Protocol: Target Engine

target threads are stopped and the target returns to waiting for a request from GDB.[3]

With the all-stop debugging paradigm, this synchronous protocol is satisfactory. With non-stop debugging there are two issues to resolve:

- When the target runs or single steps, GDB waits for a stop reply packet. GDB is unable to send another request until that stop reply is received.[4] This prevents the user from examining program state, and prevents GDB from inserting breakpoints, starting other threads and performing other run control operations.

- In non-stop mode, threads continue to run when one of them hits a breakpoint or other trap. Another thread could also trap, thus generating a set of stop replies that need to be sent to GDB. There could be zero or more stop replies to send, and GDB needs to be able to accept these.

The first issue is resolved by changing the serial protocol to respond immediately to a single step or run request. In non-stop mode, an s, c or vCont request produces an OK response. Thus GDB is free to make further requests while the target is active.

The serial protocol is further augmented to allow a single stop reply at any point. This introduces a race condition, because concurrent with the target sending a stop reply, GDB could be sending an unrelated request. The stop reply must be distinguishable from any other reply packet. If GDB receives a stop reply when it is waiting for another reply, it can save or process the stop reply and then resume waiting for the reply for its outstanding request.

Once the target has sent a stop reply, it does not send any further stop replies until it has seen a stop acknowledgement packet from GDB. The target must be prepared to handle other requests while waiting for the stop acknowledgment. In this way flow control of stop replies is achieved.

The two modified protocol engines are shown in Figure 4 and Figure 5. The GDB engine spends most of the time waiting in the may send state, ready to send a request to the target, and available to respond to stop replies from the target. As can be seen by the horizontal symmetry, run requests are treated the same as non-run requests. The target engine spends most of the time in the wait state, ready for requests from GDB, and for events from the program being debugged.

## 9 User Interface

GDB has traditionally used a command-line user interface. As graphical front ends were developed it was discovered that GDB's CLI was awkward. Commands

---

[3]The serial protocol has error checking, recovery and timeout framing. We shall not discuss that here, and simply treat it as a packet-based protocol. The error recovery adds some complexity, but does not otherwise affect this design.

[4]A special out-of-band ^C can be sent to regain control, but that is exceptional.

were insufficiently precise and responses were ambiguous. An alternative was developed called MI.[9] This remains a textual interface operating over the same input and output streams. However it is much more precise and unambiguous.

Every MI command is of the form `-command` and elicits a response of the form `^response`. Events such as breakpoints being hit are reported as `*event`. Additionally, command/response pairs may be labeled allowing a consumer to associate responses with commands. GDB prompts whenever it is ready to accept another MI command.

GDB's user interface and internals maintain the concept of *current thread*. The current thread can be changed by user command, or automatically when an executing thread stops at a breakpoint or other event. With all-stop mode, this automatic mode change is not problematic because it is not possible for the user to manipulate a thread concurrent with an executing thread hitting a breakpoint.

However, with non-stop mode, the user could be performing a sequence of thread control UI operations, and be interrupted by some other thread hitting a breakpoint. This asynchronous event changes GDB's current thread state and subsequent commands control the new thread rather than the old one. For a human using the CLI this would merely be an annoyance as the interruption would be noticed. For an IDE using MI, it would be a disaster as user actions in the GUI result in changing state unexpectedly.

We audited the MI interface for issues that non-stop debugging would introduce. The review was circulated on the GDB mailing list.[10] The main issues found were:

- MI was inconsistent about when a prompt is printed. The prompt tells the consumer that GDB is ready to accept input. In all-stop mode, after continuing the target and outputting a `^running` response, a prompt was printed even though GDB is not ready to accept input until after a subsequent `*stopped` event.

  This inconsistency becomes problematic when non-stop mode is available. The consumer needs to know unambiguously when GDB is ready to accept input.

- A `*stopped` event provided the identification number of the previous command. This is sufficient in all-stop mode, as it must have been the previous command that resumed the program leading to the `stopped` event. In non-stop mode that is no longer true. Indeed it may be impossible to determine which command was responsible for initiating execution causing the `*stopped` event. Therefore events no longer echo the previous command's identification.

- MI commands that operate on the current thread were susceptible to the race condition described earlier in this section. To correct this all MI commands that are thread-specific now have an additional thread identifier option.

  The removal of state may simplify the IDE that is using GDB. For instance there are places in Eclipse[11] where the current thread is temporarily changed to perform a sequence of commands and then changed back. Making the MI interface stateless helps reduce confusion between the IDE and GDB.

- The `^running` response can only be output in response to a continue. However, it is possible for a breakpoint action to resume different threads. In all-stop mode the IDE can examine threads when GDB stops. In non-stop mode, the IDE needs to know which threads have been started and stopped as time progresses. An additional `*running` event has been added to support this dynamic behavior.

Before executing any command that affects thread run control, GDB needs to determine if that thread is currently executing. If it is, the command is rejected.

It should be emphasized that in non-stop mode, clarity and correctness of MI is very important; hence the need for additional notifications. We implemented changes to make variable object commands work in threaded programs, and also some internal changes to improve the structure of MI processing code.

## 10 Future Directions

With non-stop debugging in place it becomes feasible to use GDB for multi-process or multi-core debugging.

Existing solutions to this often involve a single GDB instance per process or core. They use an IDE to deal with the synchronization between the GDB instances. This is tractable for a few processes or a few cores, but quickly runs into scalability issues. Running tens of GDB instances on the host and tens of GDB servers on the targets quickly uses up resources. From GDB's point of view, separate processes are just like separate threads except that they may run in their own address spaces (depending on the operating system). As separate processes can run different programs, GDB's program loader must manage multiple programs. However, it can already do this for single address-space systems due to the work we have already contributed. It will not be overly complicated to add process identification to program state and use that for memory accesses. Processes can show up as an additional level in the now familiar view of a program consisting of multiple threads.

This arrangement allows GDB to take care of program synchronization, and leaves the IDE free to focus on data presentation and user interaction.

## 11  Acknowledgements

## References

[1] GDB Home Page, `http://sourceware.org/gdb/`

[2] The GNU Project, `http://www.gnu.org/`

[3] The Free Software Foundation, `http:www.fsf.org`

[4] Ptrace system call, `man ptrace`

[5] QEMU Open Source Processor Emulator, `http://fabrice.bellard.free.fr/qemu/index.html`

[6] Probing the Guts of Kprobes, A. Mavinakayana-halli, P. Panchamukhi, J. Keniston, A. Keshava-murthy, & M. Hiramatsu, Linux Symposium 2006, `http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf`

[7] Async patch (no. 4), Nick Roberts, `http://sourceware.org/ml/gdb-patches/2006-10/msg00252.html`

[8] Non-stop multi-threaded debugging, Jim Blandy, `http://sourceware.org/ml/gdb/2007-11/msg00198.html`

[9] Debugger Machine Interface, `http://www.linux-foundation.org/en/DMI`

[10] MI non-stop mode specification, Vladimir Prus, `http://sourceware.org/ml/gdb/2008-03/msg00147.html`

[11] The Eclipse Project, `http://www.eclipse.org`