GDB, so where are we now?

Status of GDB's ongoing target and run control projects.

Pedro Alves

Red Hat

2014-02-02 Sun

Outline

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End

Topic

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End

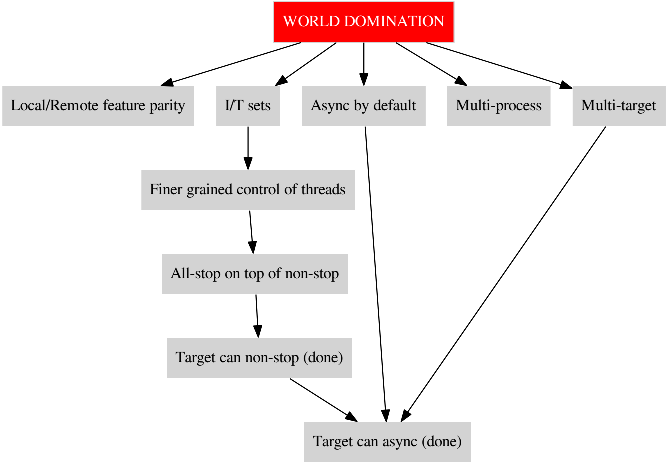
License

- License: Attribution-ShareAlike 4.0 Unported (CC BY-SA 4.0)
- http://creativecommons.org/licenses/by-sa/4.0/

Current mess

- set non-stop on/off
- set target-async on/off
- set scheduler-locking on/of/step
- set schedule-multiple on/off
- 'target remote' vs 'target extended-remote'

Where we're headed



Topic

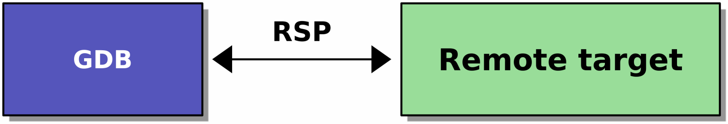
1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End

GDBserver, what's that?

- For native/local debugging on the host, GDB alone is sufficient.
  - spawn processes ("run")
  - attach to existing processes

GDBserver, how's that?

- For remote / cross debugging, GDB connects to something on the target end.



- bare metal embedded systems → remote stub, debug probe.
- emulators → builtin RSP implementation
- GNU/Linux (and others) → the GDBserver program.

GDBserver, basic usage

GDBserver

```
$ gdbserver :9999 a.out
Process /tmp/a.out created; pid = 22952
Listening on port 9999
```

GDB

```
$ gdb /tmp/a.out
Reading symbols from /tmp/a.out...done.
(gdb) target remote :9999
Remote debugging using :9999
0x000000323d001530 in _start () from \
    /lib64/ld-linux-x86-64.so.2
(gdb)
```

Topic

1 Introduction
2 GDBserver
3 Remote Serial Protocol
4 Local vs remote feature parity
5 I/t sets
6 All-stop vs non-stop modes
7 All-stop UI on top of non-stop target
8 Target async by default
9 Multi-process debugging
10 Multi-target
11 Reverse debugging
12 End

Remote Serial Protocol (RSP)

- Client/Server model
  - GDB == Client
    - runs on the host
  - Target == Server

Remote Serial Protocol (RSP)

- Client/Server model
  - GDB == Client
    - runs on the host
  - Target == Server
- Variety of transports
  - Serial
  - TCP/IP
  - UDP/IP
  - POSIX pipes

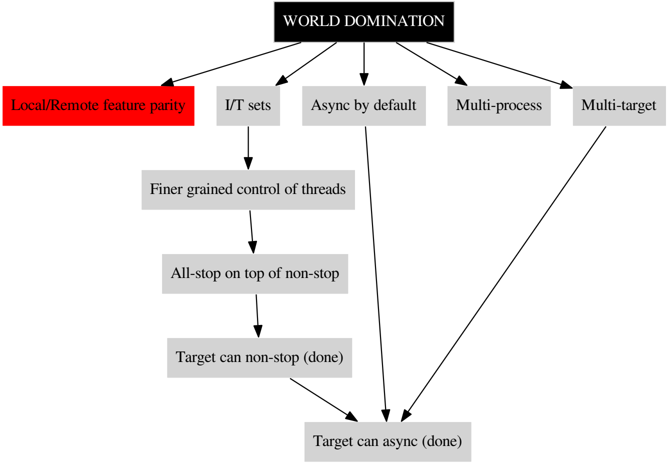
Remote Serial Protocol (RSP)

- (Mostly) text-based
  1. ⇒ `m aa55aa55,4` (read 4 bytes at 0xaa55aa55)
  2. ⇐ `ff00ff00` (here's your bytes)
  3. ⇒ `Z0 0x1234` (insert breakpoint at 0x1234)
  4. ⇐ `OK`
  5. Frame format:
     '$' packet-data '#' checksum
- Try '(gdb) set debug remote 1' to see all the RSP traffic.

https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html

Topic

1 Introduction
2 GDBserver
3 Remote Serial Protocol
4 Local vs remote feature parity
5 I/t sets
6 All-stop vs non-stop modes
7 All-stop UI on top of non-stop target
8 Target async by default
9 Multi-process debugging
10 Multi-target
11 Reverse debugging
12 End

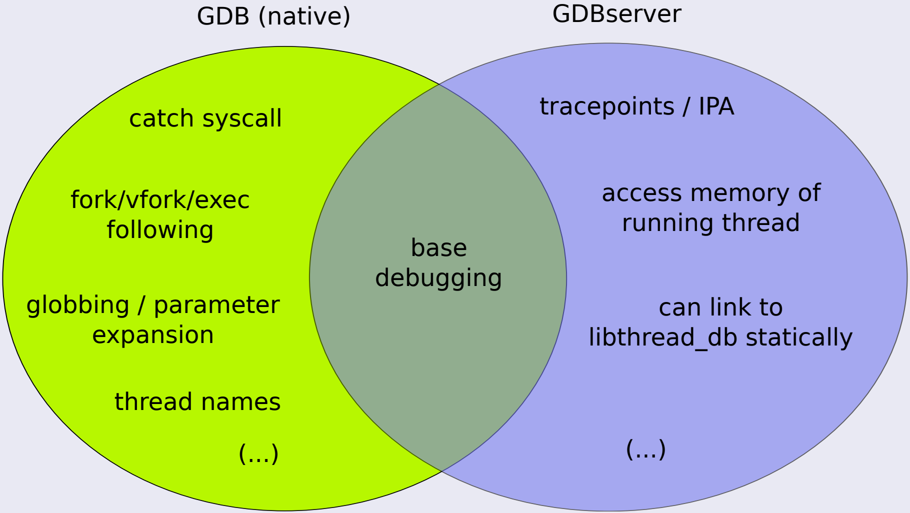

WORLD DOMINATION
Local/Remote feature parity
I/T sets
Async by default
Multi-process
Multi-target
Finer grained control of threads
All-stop on top of non-stop
Target can non-stop (done)
Target can async (done)

Local vs remote debugging

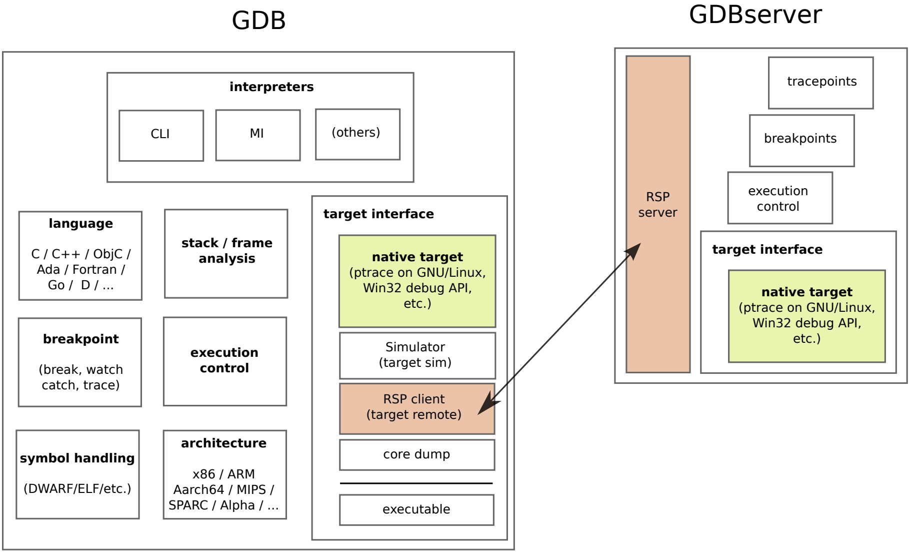
- Should be transparent, right?

I wish it were so

Local/Remote feature set comparison

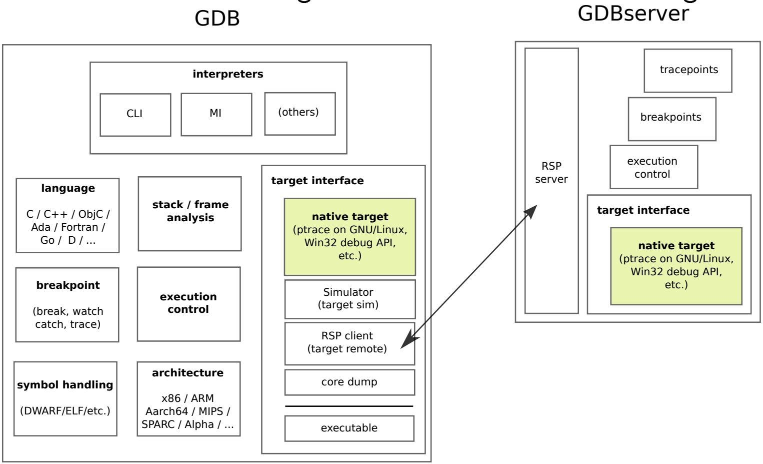


GDBserver, in blocks



Surprise, we love code duplication

- GDBserver's native target code != GDB's native target code



Bright idea

- Gosh, we could share all that code, couldn't we?

GDBserver-only features

- tracepoints
- fast tracepoints / in-process agent (IPA)
- can access memory of running thread
- other libcs (uCLinux/uClibc, Android, etc.)
  - static `libthread_db.a`, no `libthread_db` at all.
- misc others

Native-only features, part 1

- fork/vfork/exec
  - set follow-fork-mode (child/parent)
  - catch fork/vfork/exec
- catch syscall
- '(gdb) set environment FOO=bar'
- set inferior cwd
  - (gdb) cd somewhere
  - (gdb) pwd

Native-only features, part 2

- use shell to start program (globbing, wildcard expansion and I/O redirection)

Native

```
$ gdb /usr/bin/ls
(gdb) run *
Starting program: /usr/bin/ls *
1  2
[Inferior 1 (process 4750) exited normally]
```

GDBserver

```
Process /usr/bin/ls created; pid = 5260
/usr/bin/ls: cannot access *: No such file or directory
Child exited with status 2
```

Native-only features, part 3

- GDB can set/show (user defined) thread names:

Example (Thread names)

```
(gdb) info threads
  Id   Target Id                              Frame
* 1    Thread 0x77fc9740 (LWP 932) "foo" main () at foo.c:29
                                    ^^^

(gdb) thread name bar
                  ^^^

(gdb) info threads
  Id   Target Id                              Frame
* 1    Thread 0x77fc9740 (LWP 932) "bar" main () at foo.c:29
                                    ^^^

(gdb)
```

Yet more missing features when remote debugging

- Others:
  - Attach auto-load exec
  - Graceful handling of leader thread exiting
  - Inferior IO
- More. . .

Other differences

- Synching inferior thread list needs explicit "info threads".
- "info threads" output different between native/remote:

GDB

```
(gdb) info threads
   Id   Target Id         Frame
 * 1    Thread 0x7ffff7fcc740 (LWP 19056) "test" main ()
          at test.c:35
```

GDBserver

```
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 19056      main () at test.c:35
```

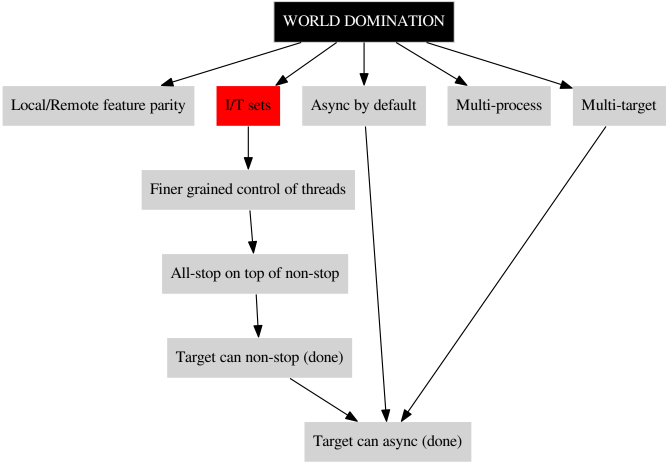
Current direction

1. GDBserver > GDB (targets backends)
2. Drop GDB's backends

- Project is tracked here: https://sourceware.org/gdb/wiki/LocalRemoteFeatureParity
- Related: https://sourceware.org/gdb/wiki/Common

Topic

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End


WORLD DOMINATION
Local/Remote feature parity
I/T sets
Async by default
Multi-process
Multi-target
Finer grained control of threads
All-stop on top of non-stop
Target can non-stop (done)
Target can async (done)

inferior/thread sets, history 1

Currently GDB can debug:

- multi-threaded programs
- programs composed of multiple processes

By default:

- any event triggers in the debugged program $\Rightarrow$ all threads stop

inferior/thread sets, history 2

Too intrusive when debugging live running systems

- Enter non-stop mode (GDB 7.0)
  - Keep all threads running, except the thread that hit the event

[The old (and default) mode was named the `all-stop` mode]

inferior/thread sets, history 3

All or nothing...

- Not flexible enough.

Desirable to group related threads, and apply group actions, e.g.:

- step, continue, etc.
- set breakpoints specific to said groups or sets
- specify what should be implicitly paused when a breakpoint triggers

inferior/thread sets, specs

- collection/combination of execution/scoping objects:
  - inferiors/processes, threads, cores, Ada tasks, etc.
- ranges and wildards
- assignable names
- union (,) and intersection (.) operators
- set negation (˜)
- refer to current and/or future entities
- predefined sets:
  - all threads, all running, all stopped, etc.

Example (a spec)

'stopped.i2.c3-5,t3'

- every thread of inferior 2, running on cores 3 to 5, but actually stopped
- plus thread 3

inferior/thread sets specs, examples

[scope TRIGGER-SET] break [-stop STOP-SET] LINESPEC

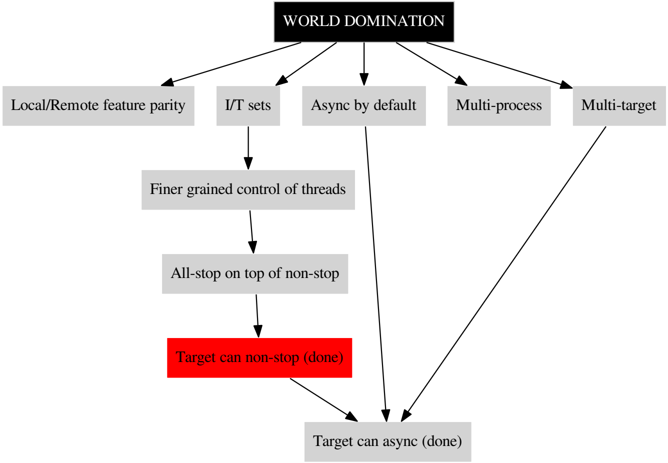
```
(gdb) scope t3 break -stop i1 main
```

```
(gdb) all> scope i1
Current scope is inferior 1.
(gdb) i1>
```

```
(gdb) all> step
(gdb) i1> step
(gdb) t1> step
(gdb) i1> step -p t2,t3
(gdb) i1> step -p c1
(gdb) i1> scope i1,i2 step
```

Topic

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End


WORLD DOMINATION
Local/Remote feature parity
I/T sets
Async by default
Multi-process
Multi-target
Finer grained control of threads
All-stop on top of non-stop
Target can non-stop (done)
Target can async (done)

all-stop vs non-stop modes

- user-visible differences
- target-side / RSP differences

all-stop vs non-stop modes, user visible differences

Different user-visible behavior:

- All-stop always stops all threads
- Non-stop leaves threads running

- All-stop always switches current thread to thread that last stopped
- Non-stop never switches the current thread

- In non-stop, resumption commands only apply to the current thread, unless explicitly overriden
- In all-stop, what's resumed depends on the `scheduler-locking` setting (and more).

all-stop vs non-stop modes, target backend / RSP differences

In all-stop RSP, resumes are synchronous/blocking

1. → `vCont;c` (continue)

all-stop vs non-stop modes, target backend / RSP differences

In all-stop RSP, resumes are synchronous/blocking

1. → `vCont;c` (continue)
2. **(program continues)**

all-stop vs non-stop modes, target backend / RSP differences

In all-stop RSP, resumes are synchronous/blocking

1. → `vCont;c` (continue)
2. **(program continues)**
3. ← `T05 ... ;thread:999` (stopped with SIGTRAP)

all-stop vs non-stop modes, target backend / RSP differences

In all-stop RSP, resumes are synchronous/blocking

1. → `vCont;c` (continue)
2. **(program continues)**
3. ← `T05 ... ;thread:999` (stopped with SIGTRAP)

- Can't send another packet while the program is running.
  - Can't insert/remove breakpoints
  - Can't list threads
  - Can't inspect globals
  - Can only explicitly stop target
    - interrupt request byte `0x03` (no packet structure)
- Or ... wait for the target to stop itself

Non-stop RSP, asynchronous notifications

Asynchronous notifications!

- Initiated by the server
- Can be sent at any time, even when target is running
- Just like other packets but start with '%' instead of '$' (at the frame level)
- Currently defined:
  - %Stop: <regular stop reply here>

Non-stop resumptions

- In the non-stop RSP variant, resumes are asynchronous

Non-stop resumptions

- In the non-stop RSP variant, resumes are asynchronous
- Other RSP traffic possible while the target is running!

Non-stop resumptions

- In the non-stop RSP variant, resumes are asynchronous
- Other RSP traffic possible while the target is running!

Example (insert breakpoint while program is running)

1. → `vCont;c` (continue all threads)

Non-stop resumptions

- In the non-stop RSP variant, resumes are asynchronous
- Other RSP traffic possible while the target is running!

Example (insert breakpoint while program is running)

1. → `vCont;c` (continue all threads)
2. ← `OK` (immediate reply) **(program continues)**

Non-stop resumptions

- In the non-stop RSP variant, resumes are asynchronous
- Other RSP traffic possible while the target is running!

Example (insert breakpoint while program is running)

1. → `vCont;c` (continue all threads)
2. ← `OK` (immediate reply) **(program continues)**
3. → `Z0 <addr1>` (Insert breakpoint)

Non-stop resumptions

- In the non-stop RSP variant, resumes are asynchronous
- Other RSP traffic possible while the target is running!

Example (insert breakpoint while program is running)

1. → `vCont;c` (continue all threads)
2. ← `OK` (immediate reply) **(program continues)**
3. → `Z0 <addr1>` (Insert breakpoint)
4. ← `OK`

Non-stop resumptions

- In the non-stop RSP variant, resumes are asynchronous
- Other RSP traffic possible while the target is running!

Example (insert breakpoint while program is running)

1. → `vCont;c` (continue all threads)
2. ← `OK` (immediate reply) **(program continues)**
3. → `Z0 <addr1>` (Insert breakpoint)
4. ← `OK`
5. **(program eventually hits breakpoint)**

Non-stop resumptions

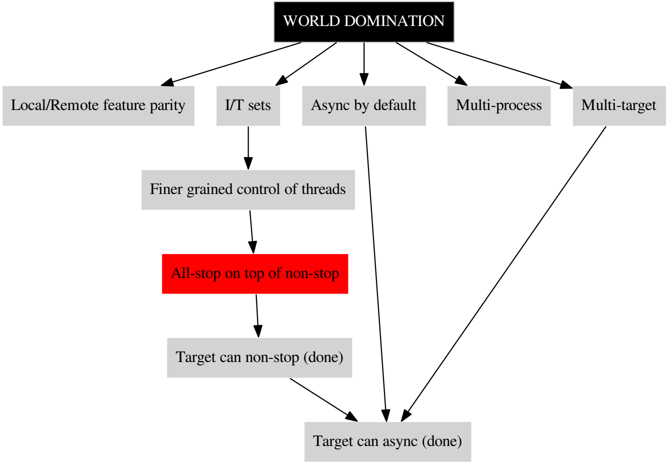
- In the non-stop RSP variant, resumes are asynchronous
- Other RSP traffic possible while the target is running!

Example (insert breakpoint while program is running)

1. → `vCont;c` (continue all threads)
2. ← `OK` (immediate reply) **(program continues)**
3. → `Z0 <addr1>` (Insert breakpoint)
4. ← `OK`
5. **(program eventually hits breakpoint)**
6. ← `%Stop:T05 ... ;thread:999` (stopped with SIGTRAP)

Topic

1 Introduction
2 GDBserver
3 Remote Serial Protocol
4 Local vs remote feature parity
5 I/t sets
6 All-stop vs non-stop modes
7 All-stop UI on top of non-stop target
8 Target async by default
9 Multi-process debugging
10 Multi-target
11 Reverse debugging
12 End


WORLD DOMINATION
Local/Remote feature parity
I/T sets
Async by default
Multi-process
Multi-target
Finer grained control of threads
All-stop on top of non-stop
Target can non-stop (done)
Target can async (done)

All-stop UI on top of non-stop target

What:

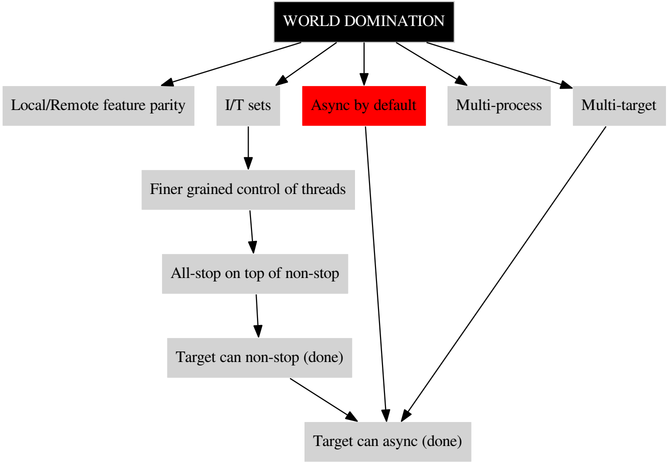
- always connect using the non-stop RSP variant
- present the all-stop behavior to the user

Why:

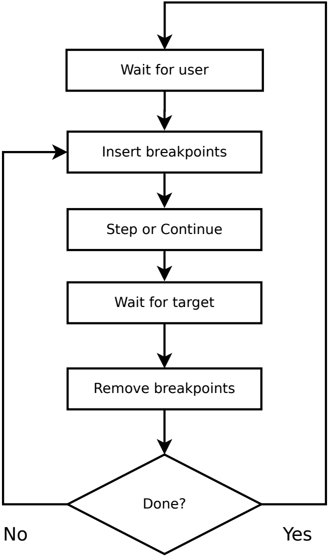
- Just one specific case in an i/t sets world – useful as incremental milestone.
- Allows true remote async

Topic

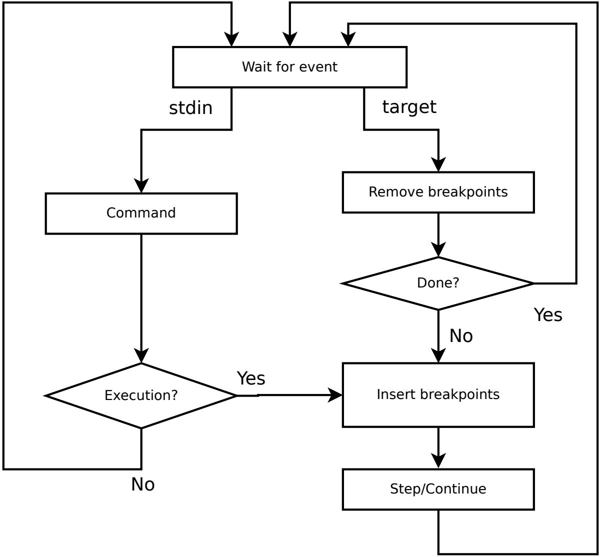
1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. **Target async by default**
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End


WORLD DOMINATION
Local/Remote feature parity
I/T sets
Async by default
Multi-process
Multi-target
Finer grained control of threads
All-stop on top of non-stop
Target can non-stop (done)
Target can async (done)

sync mode (what we always had by default)



async mode (not the default yet)

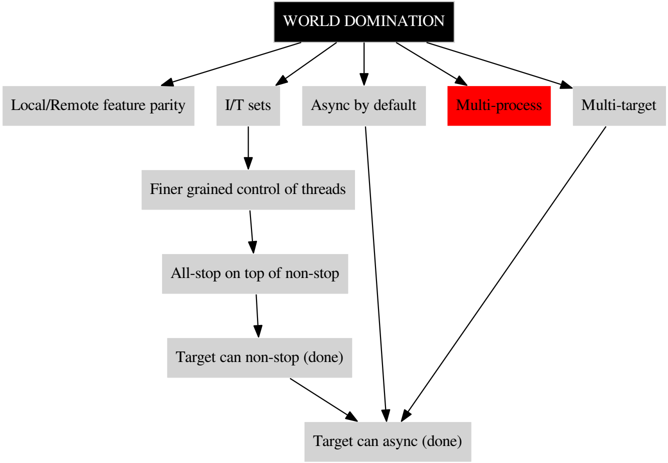


async mode (not the default yet)

```
(gdb) c&
Asynchronous execution not supported on this target.
(gdb) set target-async on
info threads
  Id   Target Id     Frame
  3    Thread 11457  0x004ba6ed in foo () at foo.c:82
  2    Thread 11456  0x004ba6ed in foo () at foo.c:82
* 1    Thread 11452  0x00408e60 in bar () at bar.c:93
(gdb) c&
Continuing.
(gdb) info threads
 Id   Target Id      Frame
  3    Thread 11457  (running)
  2    Thread 11456  (running)
* 1    Thread 11452  (running)
(gdb) interrupt ...
```

Topic

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End

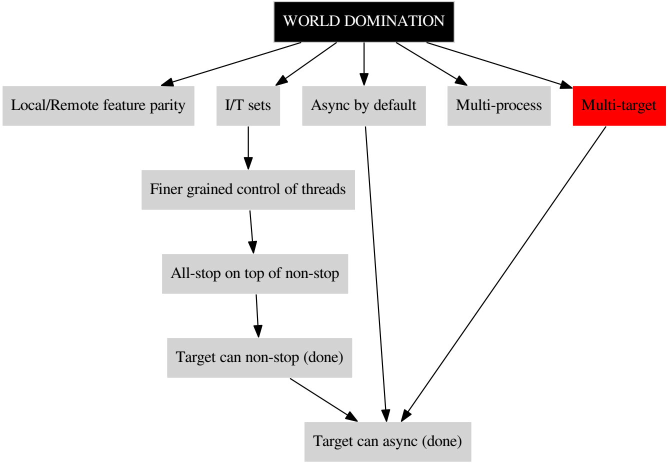

WORLD DOMINATION
Local/Remote feature parity
I/T sets
Async by default
Multi-process
Multi-target
Finer grained control of threads
All-stop on top of non-stop
Target can non-stop (done)
Target can async (done)

multi-process debugging

- Can debug several GNU/Linux programs under the same GDB session since ~7.2.
- Working on scalability now

Topic

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End


WORLD DOMINATION
Local/Remote feature parity
I/T sets
Async by default
Multi-process
Multi-target
Finer grained control of threads
All-stop on top of non-stop
Target can non-stop (done)
Target can async (done)

multi-target

Make it possible for users to connect to multiple targets at once:

- connect to multiple GDBservers at the same time
- freely mix native, remote, and core-file debugging

https://sourceware.org/gdb/wiki/MultiTarget

multi-target

- The branch is already functional
- Lots of global state needed to cleaned up. Some more to go.

Native GNU/Linux	✓
Core support	✓
Remote	almost
all others. . . .	X

- Target stack design
- User-interface not fully baked yet
  - add-inferior -new-target
- Change GDB to handle the same PID coming from multiple targets.
- Needs target-async
  - can't block waiting for a single remote file descriptor
- The usual: tests and documentation

Topic

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End

Running programs backwards

Commands

`reverse-step{,stepi,next,nexti,finish}, rc, rs, rsi, rni`

Running programs backwards

- w/ ‘target remote’ $\Rightarrow$ target does the hard work
  - Often simulators/emulators
  - Only two packets necessary:
    - ‘bc’ - backward continue
    - ‘bs’ - backward step

Running programs backwards

- Built-in process record and replay
  - “full” version:
    - allows replaying and reverse execution
    - force single-stepping, parses instructions, records effects
    - slow
    - single-threaded only
    - slow
    - x86/x86-64 GNU/Linux
    - slow
    - ARM GNU/Linux improved in 7.7 (syscall instruction recording, thumb32)
  - Intel’s branch trace (btrace) recording (GDB mainline)
    - h/w assisted (Branch Trace Store / BTS)
    - per-thread branch trace
    - does not record data
    - allows limited replay and reverse execution

Topic

1. Introduction
2. GDBserver
3. Remote Serial Protocol
4. Local vs remote feature parity
5. I/t sets
6. All-stop vs non-stop modes
7. All-stop UI on top of non-stop target
8. Target async by default
9. Multi-process debugging
10. Multi-target
11. Reverse debugging
12. End

End

- Questions
- <palves@redhat.com>