GDB and Linux Kernel Awareness

Peter Griffin

Background & Purpose of this talk

1. Many of us need to debug the Linux kernel
2. Proprietary tools like Trace32 and DS-5 are very $$$
3. Open source debuggers like GDB lack many ‘kernel debug’ features found in expensive proprietary tools.

So...

1. What exists today in open source?
2. How we can make it better?
3. Where we should make the changes?
4. What are the challenges?

High level Overview

- Ways to debug Linux using GDB
  - Pros / Cons
- What does “Linux kernel awareness” mean?
- Different approaches to implement “awareness”
  - Pros / Cons
- Proposed plan for moving forward

Ways to debug the Linux kernel with GDB (1)

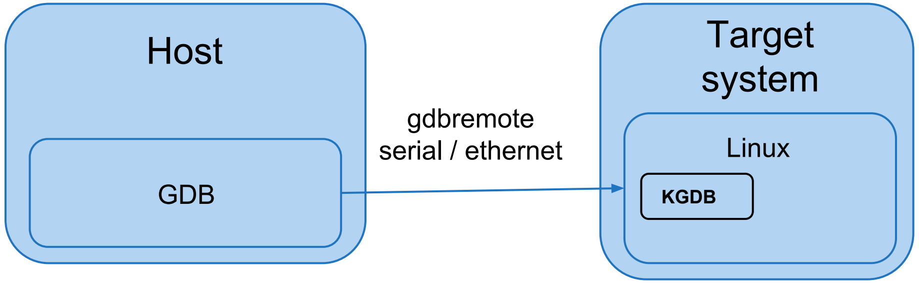
1. GDB client using gdbremote protocol
   a. to a KGDB stub in the kernel
   b. to Qemu running a kernel on an architecture of your choice
   c. to gdbremote compliant JTAG probe e.g. OpenOCD

a. KGDB and GDB

- Debug stub in the kernel compliant with gdbremote protocol.
  - Enable using CONFIG_KGDB

+ Already supported on many platforms
+ **All kernel threads enumerated in GDB** (via gdbremote)

- Requires cooperation between debugger and kernel stub
  - less suitable for serious crashes e.g. memory corruption.
- Isn't enabled in production systems.
- Requires enough support for serial or ethernet (no good for bringup).



b. Qemu and gdbserver

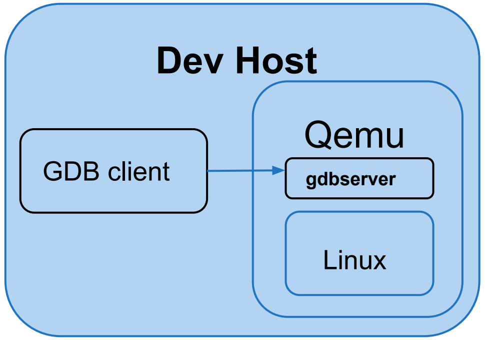
Advantages

+ Qemu is open source and has gdbremote stub
+ No “real” hardware required
+ Good for testing generic kernel code, on many architectures
+ **Good environment for developing GDB linux awareness extensions**

Disadvantages

- If your bug is SoC or board related it is unlikely to be useful

.



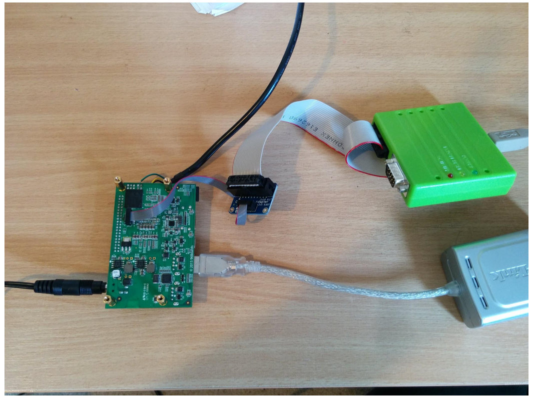
b. Qemu and gdbserver (example)

```
> qemu-system-arm -M vexpress-a9 -cpu cortex-a9 -m 256M -nographic -kernel ./zImage -append
'console=ttyAMA0,115200 rw nfsroot=10.0.2.2:/opt/debian/wheezy-armel-rootfs,v3 ip=dhcp' -dtb .
/vexpress-v2p-ca9.dtb -s
[       0.000000] Booting Linux on physical CPU 0x0
[       0.000000] Linux version 4.3.0-rc3-00008-g6e751b6-dirty (griffinp@X1-Carbon) (gcc version
4.9.3 20141031 (prerelease) (Linaro GCC 2014.11) ) #317 SMP Tue Sep 29 13:20:36 BST 2015
[       0.000000] CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c5387d
[       0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[       0.000000] Machine model: V2P-CA9
[..]
[   98.288845] VFS: Unable to mount root fs via NFS, trying floppy.
[   98.290767] VFS: Cannot open root device "(null)" or unknown-block(2,0): error -6
[   98.291132] Please append a correct "root=" boot option; here are the available partitions:
[   98.291681] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(2,0)
[   98.292215] CPU: 0 PID: 1 Comm: swapper/0 Not tainted 4.3.0-rc3-00008-g6e751b6-dirty #317
```

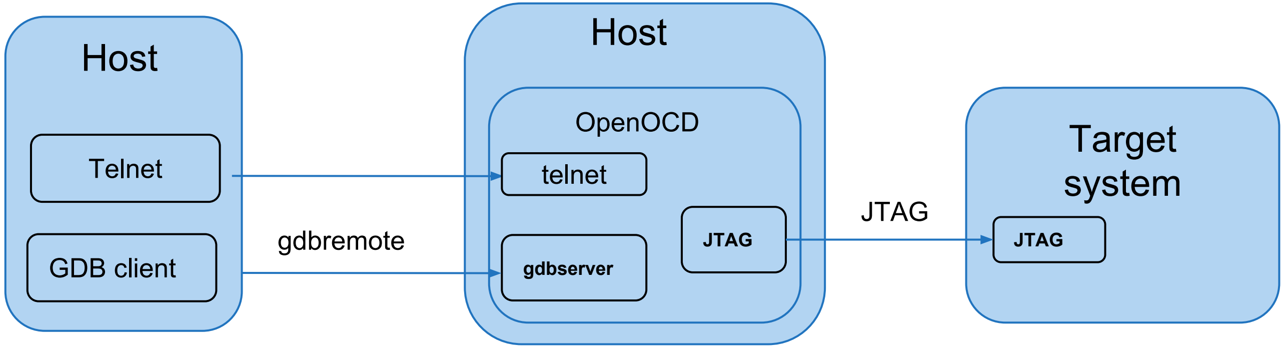
```
arm-linux-gnueabi-gdb vmlinux
(gdb) target remote :1234
Remote debugging using :1234
__loop_delay () at ../arch/arm/lib/delay-loop.S:47
47              subs    r0, r0, #1
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 1 (CPU#0 [running]) __loop_delay () at ../arch/arm/lib/delay-loop.S:47
(gdb) bt
#0  __loop_delay () at ../arch/arm/lib/delay-loop.S:47
#1  0xc02cbbe0 in panic (fmt=0xc0c340a8 "VFS: Unable to mount root fs on %s") at ../kernel/panic.c:
201
#2  0xc0dad19c in mount_block_root (name=0xc0c34148 "/dev/root", flags=32768) at ../init/do_mounts.
c:421
#3  0xc0dad354 in mount_root () at ../init/do_mounts.c:541
#4  0xc0dad4b4 in prepare_namespace () at ../init/do_mounts.c:600
#5  0xc0dace6c in kernel_init_freeable () at ../init/main.c:1026
#6  0xc09ea2f4 in kernel_init (unused <optimised out>) at ../init/main.c:936
```

c. gdbremote compliant JTAG - e.g. OpenOCD (1)

+ OpenOCD is a open source project
+ Supports gdbremote protocol
+ Supports many ARM and MIPS CPUs
+ Supports many cheap FTDI based JTAG probes



http://openocd.org



c. gdbremote compliant JTAG - e.g. OpenOCD

Compile it

```
> git clone  ssh://git@git.linaro.org:/people/peter.griffin/openocd-code
> cd openocd-code
> git submodule update --init --recursive
> autoreconf -iv
> export PKG_CONFIG_PATH=/usr/lib/x86_64-linux-gnu/pkgconfig/
> ./configure --enable-ftdi
> make
> make install
```

Run it

```
./openocd -f ./tcl/interface/ftdi/flyswatter2.cfg -f ./tcl/target/hi6220.cfg
Open On-Chip Debugger 0.9.0-dev-00241-gd356c86-dirty (2015-07-10-08:20)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
adapter speed: 10 kHz
Info : session transport was not selected, defaulting to JTAG
jtag_ntrst_delay: 100
trst_and_srst combined srst_gates_jtag trst_push_pull srst_open_drain
connect_deassert_srst
hi6220.cpu
Info : clock speed 10 kHz
Info : JTAG tap: hi6220.dap tap/device found: 0x5ba00477 (mfg: 0x23b,
part: 0xba00, ver: 0x5)
Info : hi6220.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Example telnet interface

```
telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> halt
number of cache level 2
cache l2 present :not supported
hi6220.cpu cluster 0 core 0 multi core
target state: halted
target halted in ARM64 state due to debug-request,
current mode: EL2H
cpsr: 0x800003c9 pc: 0x3ef7e908
MMU: disabled, D-Cache: disabled, I-Cache: disabled
```

Example gdb interface trace

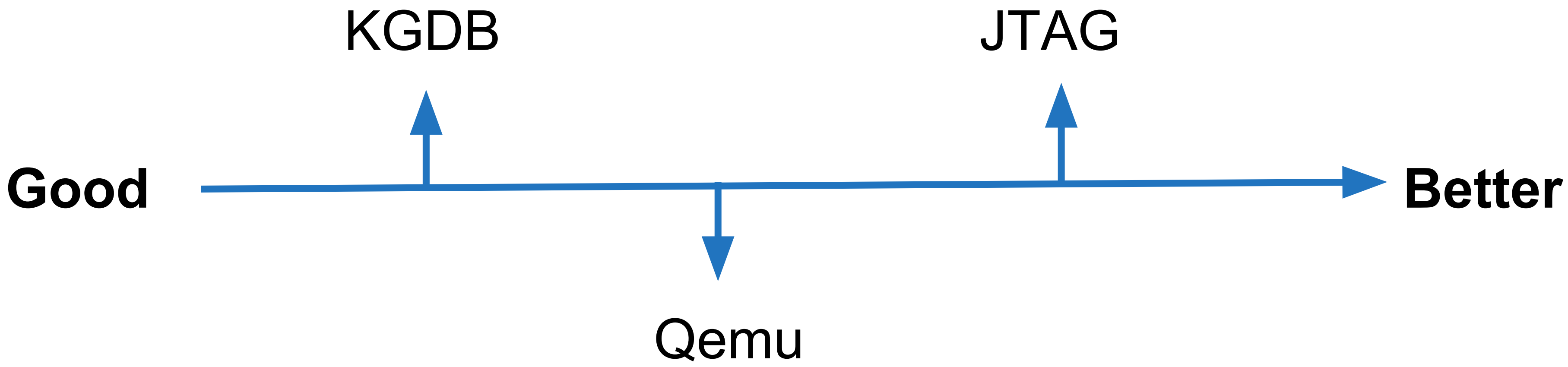
```
> aarch64-linux-gnu-gdb
(gdb) target remote :3333
(gdb) add-symbol-file build-hikey/u-boot 0x3ef47000
(gdb) hbreak do_version
Hardware assisted breakpoint 1 at 0x3ef4cc80: file ..
/common/cmd_version.c, line 19.
(gdb) c
Continuing.

Breakpoint 1, do_version (cmdtp=0x3ef8c0c0, flag=0,
argc=1, argv=0x3e746850) at ../common/cmd_version.c:19
19   {
(gdb) bt
#0  do_version (cmdtp=0x3ef8c0c0, flag=0, argc=1,
argv=0x3e746850) at ../common/cmd_version.c:19
#1  0x000000003ef604a8 in cmd_call (argv=0x3e746850,
argc=1,
```

Why JTAG?

+ Invaluable tool for SoC and board bring-up
+ Allows you to debug / inspect very broken systems
  - e.g. when KGDB stub might no longer work due to corruption
+ Allows you to debug before a KGDB stub is functional
+ Uses dedicated debug HW on the SoC (hw breakpoints / watchpoints etc).

- Configuration scripts (clocks, ddr timings) can be fiddly to setup
  - Most likely need support from the SoC vendor or validation, or hot attach


KGDB
JTAG
Good
Better
Qemu

Ways to debug the Linux kernel - dumps (2)

2. GDB to debug a kernel dump file
   a. obtained from /proc/kcore
      i. Enable CONFIG_PROC_KCORE and CONFIG_DEBUG_INFO
      ii. Provides a virtual ELF core file of the live kernel. No modifications can be made

```
> sudo gdb vmlinux /proc/kcore
(gdb) p jiffies_64          (print the value of jiffies_64)
$1 = 4326692196         (and there it is)

(gdb) core-file /proc/kcore #to refresh core file contents
```

   b. obtained from /proc/vmcore
      i. Enable CONFIG_PROC_VMCORE
      ii. Dumps image of the crashed kernel in ELF format
      iii. Used in conjunction with kexec and kdump, and crash utility from RedHat
      iv. See http://www.dedoimedo.com/computers/kdump.html

Upstream GDB to debug the Linux kernel

So you have GDB debugging the Linux Kernel via JTAG, or Qemu…

but compared to $$$ commercial tools it can be somewhat disappointing...

Upstream GDB to debug the Linux kernel

- **(gdb) info threads** - Shows thread running **on each physical CPU**
  - No visibility of other sleeping threads
  - No visibility of user processes

Eeek...only
1 thread!

```
arm-linux-gnueabi-gdb vmlinux
(gdb) target remote :1234
Remote debugging using :1234
__loop_delay () at ../arch/arm/lib/delay-loop.S:47
47              subs    r0, r0, #1
(gdb) info threads
  Id   Target Id        Frame
* 1  Thread 1 (CPU#0 [running]) __loop_delay () at ..
/arch/arm/lib/delay-loop.S:47
```

Upstream GDB to debug the Linux kernel

- **(gdb) bt -** Works for CPU executing in kernel space. No unwinding for user tasks.

Example ‘bt’ executing in kernel space

```
arm-linux-gnueabi-gdb vmlinux
(gdb) target remote :1234
(gdb) bt
#0  __loop_delay () at ..
/arch/arm/lib/delay-loop.S:47
#1  0xc02cbbe0 in panic (fmt=0xc0c340a8
"VFS: Unable to mount root fs on %s") at ..
/kernel/panic.c:201
#2  0xc0dad19c in mount_block_root
(name=0xc0c34148 "/dev/root", flags=32768)
at ../init/do_mounts.c:421
#3  0xc0dad354 in mount_root () at ..
/init/do_mounts.c:541
#4  0xc0dad4b4 in prepare_namespace () at
../init/do_mounts.c:600
```

Example ‘bt’ executing in userspace

```
Program received signal SIGINT,
Interrupt.
0xb6e4978c in ?? ()
(gdb) bt
#0  0xb6e4978c in ?? ()
#1  0xb6e4a408 in ?? ()
Backtrace stopped: previous frame
identical to this frame (corrupt
stack?)
```

Upstream GDB to debug the Linux kernel

`up / down / frame <num> -` same as bt, will show physical CPU, and only work if executing in kernel space.

`thread <num> -` switch physical CPU’s

This could be better, much more like KGDB or $$ JTAG tools

What is “Linux awareness”?

- Provides the debugger with additional knowledge of the underlying operating system to enable a better debugging experience. e.g. Where is the task list in memory? kernel log buffer?

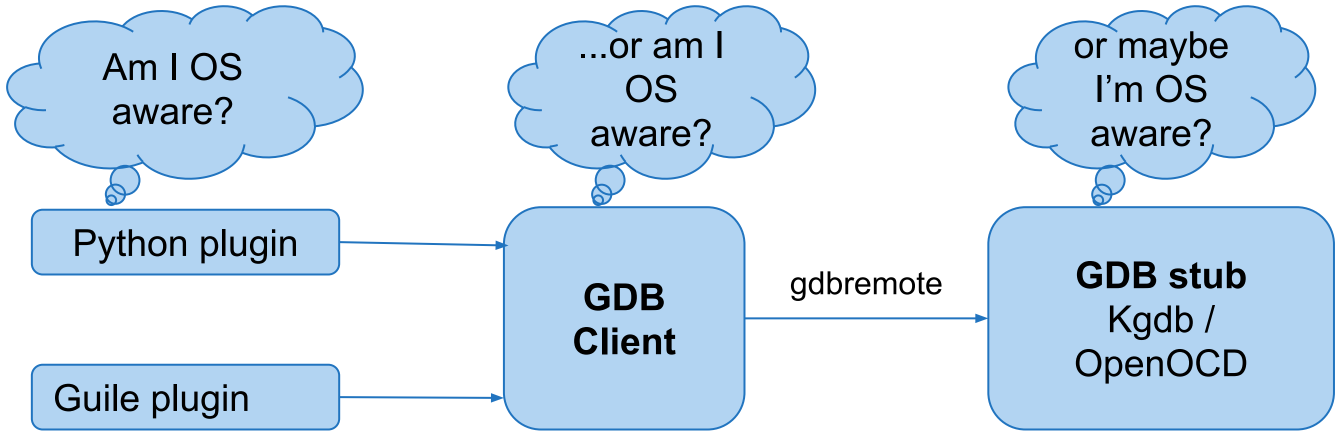
- ‘Linux awareness’ in this talk split into 3 areas: -

1. **Task awareness**
   - Ability to report all `task_structs` as threads into GDB
   - Threads selectable via usual GDB execution context commands (`info threads, thread <num>, frame <num>, up, down`)
2. **Loadable module support**
   - Ability to debug loadable modules (behave much like shared library support for userspace)
   - Hooks to allow automatic symbol resolution when modules inserted / removed.
   - Hooks to allow the debug of module probe() and remove()
3. **OS helper commands**
   - Extract useful pieces of state from the OS (e.g. dmesg buffer)

Where to put the “awareness”?

- We have 3 options
  1. Scripting extension to GDB (Python / Guile)
  2. Awareness in GDB stub
  3. C extension to GDB

- Each of these approaches has advantages and disadvantages



1. Scripting extension to GDB

Implementing awareness - GDB python interface

- You can extend GDB using python. See https://sourceware.org/gdb/onlinedocs/gdb/Python.html#Python

- What is exposed via python interface has been steadily improving in GDB
  - Add custom commands
  - Implement pretty printers
  - frame filters, frame decorators and much more.

- The **major** advantage of implementing “Linux awareness” in Python, is the code can live in the kernel source tree.
  - + **Extensions should evolve with the kernel source**
  - + No cross dependencies between kernel & debugger

Python GDB linux awareness “extensions”

Jan Kiszka from Siemens has already implemented some python Linux awareness extensions.

- See Docs @ Documentation/gdb-kernel-debugging.txt
- See Python @ scripts/gdb/*
- Enable CONFIG_GDB_SCRIPTS
- Provides a series of custom Linux commands and functions
- Can be easily tested using Qemu and OpenOCD

Python GDB linux awareness “extensions”

Python GDB commands already merged in the kernel source

command	**lx-dmesg**	Print Linux kernel log buffer
command	**lx-lsmod**	List currently loaded modules
command	**lx-symbols**	(Re-)load symbols of Linux kernel and currently loaded modules
command	**lx-ps**	List current kernel tasks

function	**lx_current**	Return current task
function	**lx_module**	Find module by name and return the module variable
function	**lx_per_cpu**	Return per-cpu variable
function	**lx_task_by_pid**	Find Linux task by PID and return the task_struct variable
function	**lx_thread_info**	Calculate Linux thread_info from task variable

Python GDB linux awareness “extensions”

Possible improvements

- lx-ps: Command is the beginning of thread awareness in python.
  - However currently just prints a list of tasks.
  - Thread objects aren’t created inside gdb debug session
    - Therefore the usual GDB execution context commands `“thread <num>”, “frame up/down”, “bt”` still unusable.
    - GDB Python API would need to be extended to support this
      - pass a thread list, or create thread objects via python
- Extra ‘OS helper’ commands
- python frame unwinders & pretty printers could be implemented
  - stop back tracing on various kernel entry points

Thread awareness in GDB stub

OpenOCD -rtos linux task awareness

- An example of putting “thread awareness” in the GDB stub
- OpenOCD already supports various rtos task awareness (eCos, ThreadX, FreeRTOS, ChibiOS, embKernel, mqx and Linux)!
  - See openocd-code/src/rtos/linux.c
  - Disabled by default

- Enabled by adding “-rtos linux” to the target command in OpenOCD config file

```
target create $_TARGETNAME_2 cortex_a -chain-position $_CHIPNAME.dap
-dbgbase $_DAP_DBG2 -coreid 1 -rtos linux
```

- This has been tested with Tincantools’ Flyswatter JTAG and a U8500 (armv7) snowball board.
- Required some hacking to get it working, but in the end I could successfully enumerate a full thread list in GDB!

OpenOCD -rtos task awareness

Advantages

- + No GDB changes required (threads returned via gdbremote packets).
- + All kernel threads can be enumerated in GDB (once working)
- + Can be used in conjunction with python extensions (e.g. lx-dmesg etc)
- + OpenOCD already has mmu virt to phys translation code for many CPUs

Disadvantages

- **Awareness in GDB means thread awareness needs re-implementing for each stub (Qemu, OpenOCD, etc).**
- **No good for debugging kernel dumps**
- **Tight coupling of code parsing kernel data structures in OpenOCD**
- **No debug information available in OpenOCD.**
- No way (I know of) to find field offsets via gdbremote protocol.
- Currently this means task_struct, thread_info and mm_struct field offsets require OpenOCD to be **recompiled** for the kernel you wish to debug :(

Improvements

- gdbremote protocol could be extended
- At a minimum task structure field offsets need to be configurable at runtime, rather than requiring a recompile!

OpenOCD linux-header.h

```
/*  gdb script to update the header file  according to kernel version and build option
    before executing function awareness  kernel symbol must be loaded : symbol
    vmlinux

define awareness

 set logging off

 set logging file linux_header.h

 set logging on

 printf "#define QAT %p\n",&((struct task_struct *)(0))->stack

 set $a=&((struct list_head *)(0))->next

 set $a=(int)$a+(int)&((struct task_struct *)(0))->tasks

 printf "#define NEXT  %p\n",$a

 printf "#define COMM  %p\n",&((struct task_struct *)(0))->comm

 printf "#define MEM  %p\n",&((struct task_struct *)(0))->mm

 printf "#define ONCPU %p\n",&((struct task_struct *)(0))->on_cpu

 printf "#define PID %p\n",&((struct task_struct *)(0))->pid

*/
```

```
#define QAT 0x4

#define NEXT  0x1b0

#define COMM  0x2d4

#define MEM  0x1cc

#define ONCPU 0x18

#define PID 0x1f4

#define CPU_CONT 0x1c

#define PREEMPT 0x4

#define MM_CTX 0x160
```

C extension to GDB

GDB C extension - Linux Kernel Debugger (LKD)

- STMicroelectronics has a patchset on vanilla GDB that adds better "Linux awareness".
- Implemented as a C code extension inside GDB 7.6 using the GDB target model.
- Originally developed for STMC2 JTAG debugger

  BUT...compliant with gdbremote so e.g. Qemu / OpenOCD

- Mainly tested with armv7 cortex a9 and SH4 ST40
- **ST would like Linaro to help contribute LKD upstream**

life.augmented

Linaro

GDB C extension - LKD

- LKD implements a `‘struct target_ops linux_aware_ops’` to overload GDB functions, and then places itself at the top of the target-stack.

```
(gdb) set linux-awareness loaded
[..]
(gdb) maint print target-stack
The current target stack is:
  - linux-aware (Linux-aware target interface)
  - remote (Remote serial target in gdb-specific
protocol)
  - exec (Local exec file)
  - None (None)
```

GDB C extension - LKD - 1. Task Awareness

+ Maps anything with a `task_struct` to a thread in GDB
+ Allows usual GDB execution context commands (e.g. `thread <num>, frame up / down`) to be used.
+ Manages task list to correctly discover new threads.
+ Manages task list to correctly remove dead threads
+ Some JTAG optimizations to read whole “task struct”, rather than individual fields for efficiency.

- Still no unwinding of user processes (yet)

GDB C extension - LKD - 2. Loadable Modules

- Reuses GDB solib infrastructure (`struct target_so_ops`)
- Allows debugging of module probe/remove via bp hooks in `module_init` and `module_arch_cleanup`
- Enables automatic loading of .ko symbol information in GDB
    - Correctly removes symbol info from GDB for init sections which are freed after module load.

- Modules memory is allocated using `vmalloc()`
    - Which creates a dynamic mapping
    - This requires some extra virt to phys translation code in LKD, and the ability to reconfigure MMU via gdbremote protocol.

gdbremote extensions - MMU switching

To get LKD working with Qemu or OpenOCD the following additional gdbremote packets need to be implemented.

`st cp15 c1 0 c0 0`	Read System Control Register
`st cp15 c2 0 c0 0`	Read Translation Table Base Register 0
`st cp15 c2 0 c0 0 0x%x`	Write TTRB0
`st cp15 c13 0 c0 1`	Read Context ID register (ASID)
`st cp15 c13 0 c0 1 0x%x`	Write ASID

GDB C extension - LKD: 3. OS helper commands

LKD implements various Linux helper commands inside GDB such as: -

dmesg	dump dmesg log buffer from kernel
process_info	prints various info about current process
pmap	prints memory map of current process
vm_translate	Translates virtual to physical address
proc_interrupts	prints interrupt statistics
proc_iomem	prints I/O mem map
proc_cmdline	prints the contents of /proc/cmdline
proc_version	prints the contents of /proc/version
proc_mounts	Print the contents of /proc/mounts
proc_meminfo	Print the contents of /proc/meminfo.

LKD - example trace

```
(gdb) bt
#0  __loop_delay () at ../arch/arm/lib/delay-loop.S:47
#1  0xc097b9a4 in panic (fmt=0xc0bd4688 "VFS: Unable to mount root fs on %s") at ..
  /kernel/panic.c:201
#2  0xc0d401a0 in mount_block_root (name=0xc0bd4728 "/dev/root", flags=32768) at ..
  /init/do_mounts.c:421
#3  0xc0d40358 in mount_root () at ../init/do_mounts.c:541
#4  0xc0d404b8 in prepare_namespace () at ../init/do_mounts.c:600
#5  0xc0d3fe70 in kernel_init_freeable () at ../init/main.c:1027
#6  0xc0979b90 in kernel_init (unused=<optimized out>) at ../init/main.c:937
#7  0xc0210928 in ret_from_fork () at ../arch/arm/kernel/entry-common.S:95
#8  0xc0210928 in ret_from_fork () at ../arch/arm/kernel/entry-common.S:95
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
```

Linaro

LKD- task awareness example

```
(gdb) set linux-awareness loaded on
Loaded ARMv7 LKD support.
[New [swapper/0]]
[New [kthreadd]]
[New [ksoftirqd/0]]
[New [kworker/0:0]]
[New [kworker/0:0H]]
[New [kworker/u8:0]]
[New [rcu_sched]]
[New [rcu_bh]]
[New [migration/0]]
[New [watchdog/0]]
[New [watchdog/1]]
[New [migration/1]]
[..]
```

Starts enumerating the thread list

LKD- task awareness example

```
(gdb) info threads
  Id   Target Id         Frame
  48   [kworker/0:1H] (TGID:52) context_switch  at ../kernel/sched/core.c:2596
  47   [deferwq] (TGID:51) context_switch  at ../kernel/sched/core.c:2596
  46   [kworker/0:2] (TGID:50) context_switch  at ../kernel/sched/core.c:2596
  45   [kpsmoused] (TGID:49) context_switch  at ../kernel/sched/core.c:2596
  44   [kworker/u8:1] (TGID:46) context_switch  at ../kernel/sched/core.c:2596
  43   [nfsiod] (TGID:41) context_switch  at ../kernel/sched/core.c:2596
  42   [fsnotify_mark] (TGID:40) context_switch  at ../kernel/sched/core.c:2596
  41   [kswapd0] (TGID:39) context_switch  at ../kernel/sched/core.c:2596
  40   [kworker/0:1] (TGID:38) context_switch  at ../kernel/sched/core.c:2596
  39   [rpciod] (TGID:37) context_switch  at ../kernel/sched/core.c:2596
  38   [devfreq_wq] (TGID:36) context_switch  at ../kernel/sched/core.c:2596
  37   [edac-poller] (TGID:35) context_switch  at ../kernel/sched/core.c:2596
[..]
  * 2    [swapper/0] (TGID:0 <C0>) cpu_v7_do_idle () at ../arch/arm/mm/proc-v7.S:
   74
```

All kernel threads now appear as threads in GDB

GDB C extension - LKD

Advantages

+ **Allows "Linux awareness" support to be re-used (JTAG / Qemu / kdumps), as opposed to an implementation in the GDB stub.**
+ All kernel tasks enumerated, allowing GDB execution commands on threads
+ Loadable module support uses solib infra in GDB
+ Debug info easily available inside GDB
+ LKD also provides a GDB testsuite
+ An implementation already exists

Disadvantages

- Creates a dependency between GDB and the kernel to enable parsing of kernel data structures.
- 'Program specific' information inside GDB, although kernel isn't "your average program".
- gdbremote protocol needs extending for MMU control

Proposed route forward

- RFC “GDB Linux awareness analysis” sent to GDB mailing list, to solicit community feedback https://sourceware.org/ml/gdb-patches/2015-06/msg00040.html

Next steps...

- Port LKD to GDB 7.10 - push to public Linaro GIT
- Remove C implementations which overlap with python (e.g. dmesg)
- Test LKD with OpenOCD (add additional
- Migrate parts of LKD into python and merge into kernel
- Post RFC patches to gdb-patches with core functionality
  - Lather, rinse, repeat :)

This topic (gdb linux awareness) was also discussed at GNU Cauldron in Prague. So far GDB community response has been positive, to having additional “Linux awareness” in GDB.

LKD: Overview of LKD patchset

```
 1129 lkd-arm.c          ARM arch specific (virt to phys)
  674 lkd.h
  491 lkd-irqs.c
 4953 lkd-main.c         linux-aware target_ops
 2456 lkd-modules.c      modules target_so_ops
   79 lkd-modules.h
 1666 lkd-process.c      task-wareness
   77 lkd-process.h
  220 lkd-proxy.c
 2076 lkd-sh4.c          SH4 specific
13874 total
```

Another C extension to GDB...

GDB C extension - kdump-gdb

- Discovered in response to “RFC linux awareness” analysis ML post
- Similar to LKD in that it adds “task awareness” for debugging kernel dumps into GDB.
- Uses a new library "libkdumpfile", to open the dumps in various formats, and handles virtual to physical memory mapping.
- Plans to re-implement other “crash” utility commands in python.
  - Support should be consolidated with LKD patchset.

https://www.sourceware.org/ml/gdb/2015-09/msg00014.html

https://github.com/alesax/gdb-kdump

https://github.com/ptesarik/libkdumpfile

Questions? Demo? Ideas?

- What other OS helper commands would be useful (lx-dt-dump)?
- Thoughts, opinions and ideas?
- Android “awareness”?

Other Useful links

https://people.redhat.com/anderson/crash_whitepaper/

https://gcc.gnu.org/wiki/cauldron2015?action=AttachFile&do=view&target=Andreas+Arnez_+Debugging+Linux+kernel+dumps+with+GDB.pdf

https://sourceware.org/gdb/onlinedocs/gdb/Python.html#Python

See Documentation/kdump/gdbmacros.txt - some more Linux awareness GDB macros

http://www.elinux.org/Debugging_The_Linux_Kernel_Using_Gdb

OpenOCD armv8 on HiKey https://github.com/96boards/documentation/wiki/JTAG-on-HiKey

Linaro

More about Linaro: www.linaro.org/about/
Linaro members: www.linaro.org/members