Linux Kernel Debugging Tools

Jason Wessel

- Product Architect for WR Linux Core Runtime
- Kernel.org KGDB Maintainer

June 23th, 2010

Agenda

- Talk about a number of common kernel debugging tools
- Describe high level tactics for using kernel debugging tools
- Demonstrate using several tools

The harsh reality is you could spend a whole day or more talking about each tool.

*** Later find slides/code at: http://kgdb.wiki.kernel.org ***

Exciting news

- For 2.6.35-rc1
  - kdb (the kernel debug shell) merged to mainline!
  - Ability to debug before console_init()
  - You can use the EHCI debug port with kgdb
- There are thoughts about next few years of kgdb/kdb
  - Implement complete atomic kernel mode setting
  - Continue to improve the non ehci debug usb console
  - Improve keyboard panic handler
  - Further integration with kprobes and hw assisted debugging
  - netconsole / kgdboe v2 – Use dedicated HW queues
- The only bad news is it takes a long time to get there.

Is there anything better than kgdb?

- Good
  - kgdb / kdb
- Better
  - QEMU/KVM backend debugger
  - Virtual box backend debugger
  - vmware backend debugger
  - kdump/kexec
- Best
  - ICE (usb or ethernet)
  - Simics (because it has backward stepping)
- In a class by itself
  - printk() / trace_printk()

The challenge is knowing what to use when... Working tools rock!

A bit about printk() and timing

- printk is probably the #1 most reliable debug
- Any seasoned kernel developer has surely experienced:
  - Add a printk and the bug goes away!
  - Timing in the kernel is complex and printk can be quite expensive particularly if you are writing to a serial console because it synchronously pauses the kernel
- This leads some folks to try some other tools or use printk() in conjunction with the tools
  - kprobes / perf / ftrace / kgdb where you toggle
  - Some folks still like to get more creative with printk and counters

What about faster printk vs timing?

- Use something faster than serial
  - USB EHCI debug port
  - netconsole (**not robust** to printk in IRQs)
- Use console log levels and run dmesg
  - echo 0 > /proc/sys/kernel/printk
- Consider trace_printk() (CONFIG_FTRACE)
  - This uses a per cpu ring buffer so as to be lighter weight than printk
  - ftrace has a lot you can do with it
    - Read more at: Documentation/trace/ftrace.txt
    - Read about trace-cmd: http://lwn.net/Articles/341902/
    - Good for irq_off, preempt_off, scheduler debugging and much more

trace_printk() demo

- I have a kernel module with extended logging using trace_printk()
  - This is nothing more than putting some code in such as:
    - trace_printk("arbitrary_keywords: Some Message\n");
- Get at the logs by mounting the debugfs

  mount -t debugfs nodev /sys/kernel/debug

  cat /sys/kernel/debug/tracing/trace
- Also consider dumping on oops:

  echo 1 > /proc/sys/kernel/ftrace_dump_on_oops

EHCI Debug Port

- Great for when you do not have rs232
- Higher speed than rs232
- Works with KGDB

  kgdbdbgp=0
- Use it as a Linux Console

  console=ttyUSB0 AND/OR earlyprintk=kdbgp0

- Read more in your kernel source tree:

  Documentation/x86/earlyprintk.txt
- You can buy one at

  http://www.semiconductorstore.com/cart/pc/viewPrd.asp?idproduct=12083

KDB – The in-kernel debug shell

- To use kdb you must meet one of following constraints
  - Use a non usb keyboard + vga text console
  - Serial port OR EHCI debug port and dongle
- kdb is not a source debugger
  - However you can use it in conjunction with gdb and an external symbol file
- Maybe you don't need a kernel debugger, but you at least want a chance to see ftrace logs, dmesg, poke a stack trace or do one final sysrq.
  - kdb might be the tool you are looking for

Loading KDB

Having kdb loaded allows you to trap the panic handler.

- For a serial port:

  echo ttyS0 > /sys/module/kgdboc/kernel/kgdboc

- For the keyboard + vga text console

  echo kbd > /sys/module/kgdboc/kernel/kgdboc

- Enter kdb with sysrq-g

  echo g > /proc/sysrq-trigger

- Remember kdb is a stop mode debugger
  - Entering kdb means all the other processors skid to a stop
  - You can run some things like: lsmod, ps, kill, dmesg, bt
  - ftdump to dump ftrace logs (not merged to mainline yet)
  - You can also use hw breakpoints or modify memory

KDB Demonstration 1

- Simply loading kdb gives you the opportunity to stop an look at faults perhaps using external tools

  echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc

  insmod test_panic.ko

  echo 1 > /proc/test_panic/panic

- After the panic collect dmesg, ftdump, bt, and lsmod
- Use gdb to load the symbol file and kernel module

  gdb ./vmlinux

  add-symbol-file test_panic.ko ADDR_FROM_LSMOD

  info line *0xADDR_FROM_BT

KDB Demonstration 2 - breakpoints

- Load kdb and use a data write breakpoint

  insmod test_panic.ko

  echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc

  echo g > /proc/sysrq-trigger

  bph tp_address_ref dataw

  go

- Cause the problem and collect the data

  echo 1 > /proc/test_panic/bad_access

  bt

  rd

  lsmod

- Statically look at the source with gdb + module address

KDB (totally experimental demo)

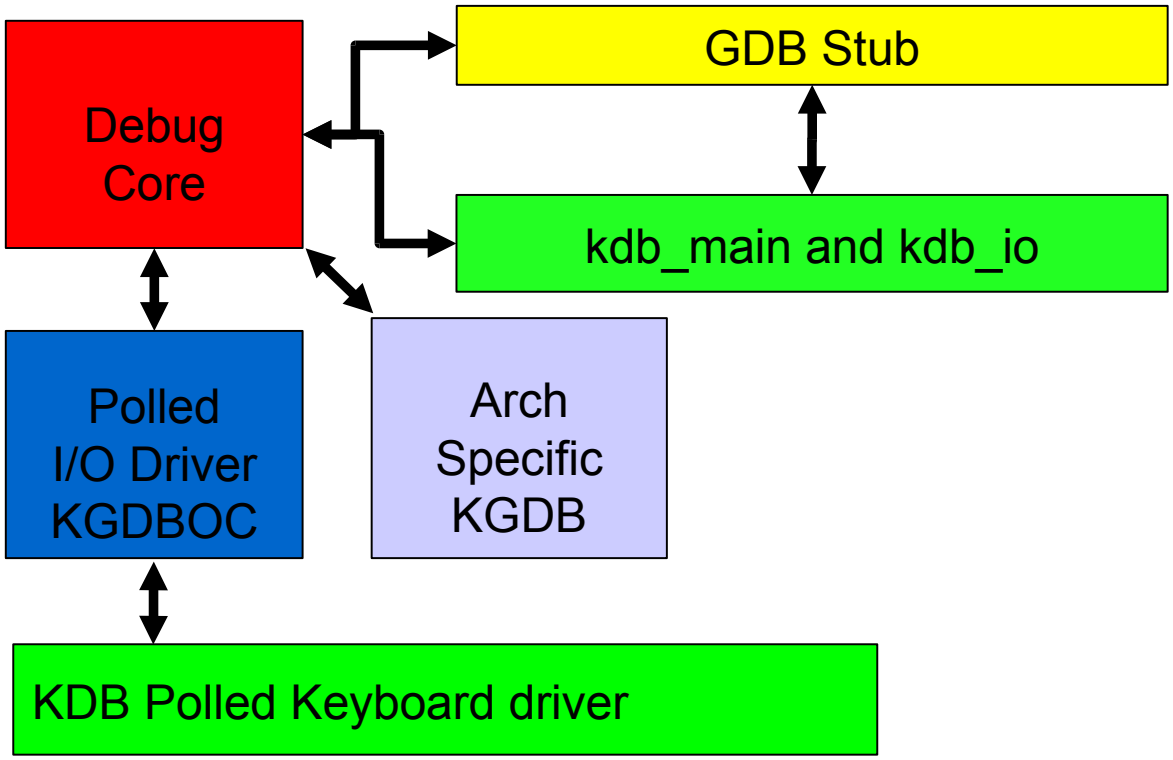
- Eventually for KDB to expand to the desktop / netbook / small embedded device we need KMS (Kernel Mode Setting) integration
- A proof of concept version of the code exists and we can watch a short video of what it looks like and you can determine if it would be useful to you in the long run
  - Think of a panic you just don't get to see on a small device
  - Think of a crash bad enough there are no logs synced to disk
  - No kdump available because you do not have any persistent storage
  - You have no serial, ethernet, just the small production device

KGDB facts

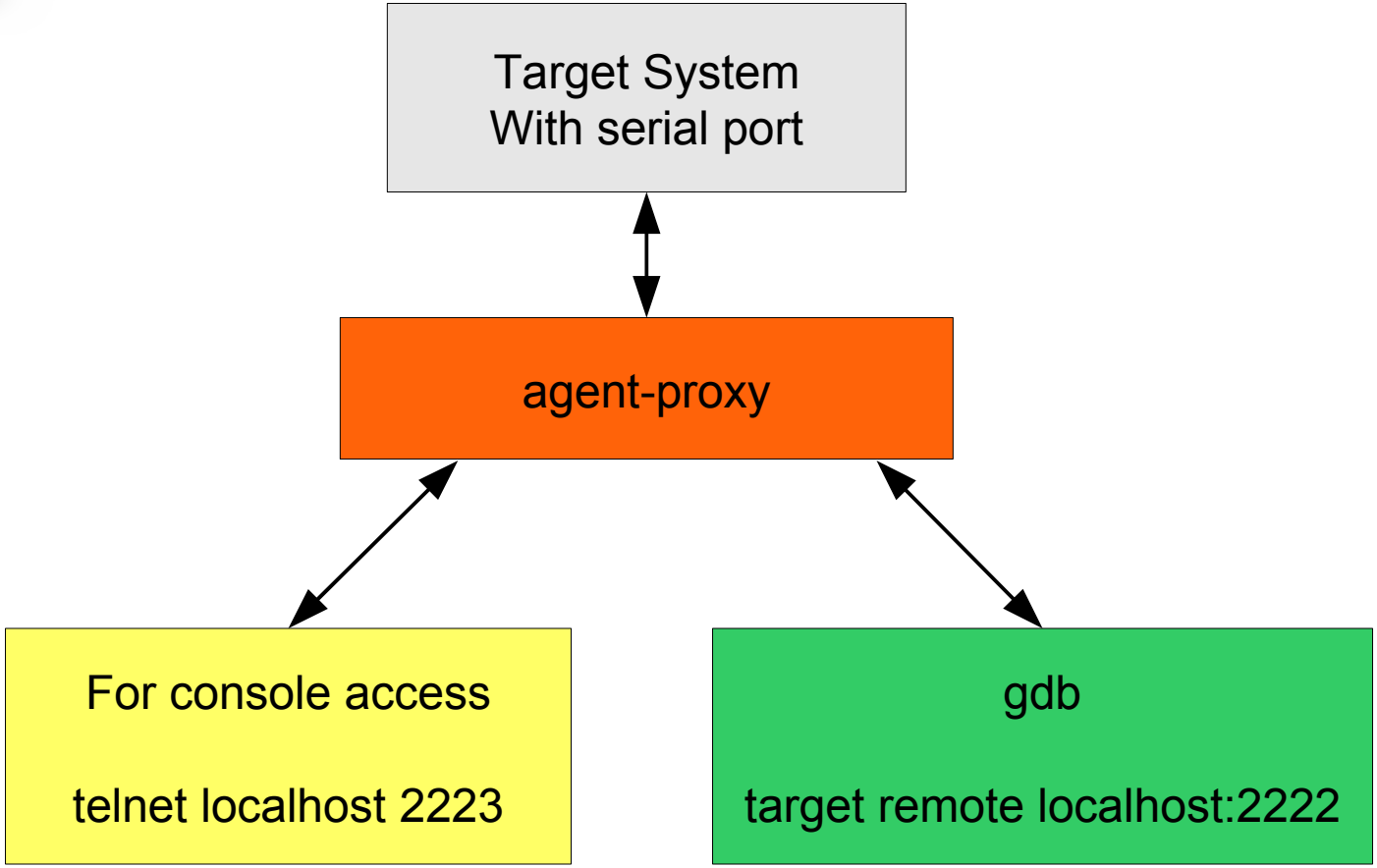
- kgdb and kdb use the same debug backend
- kgdboe (kgdb over ethernet) is not always reliable
  - kgdboe in the current form WILL NOT BE MAINLINED
  - Linux IRQs can get preempted and hold locks making it unsafe or impossible for the polled ethernet driver to run
  - Some ethernet drivers are so complex with separate kernel thread that the polled mode ethernet can hang due to locking or unsafe HW resource access
  - If you really want to attempt use kgdboe successfully, use a dedicated interface if you have one and do not use kernel soft or hard IRQ preemption.
- kgdboc is slow but the most reliable
- The EHCI debug port is currently the fastest kgdb connection

KGDB

What does a block digram of what kgdb looked like?



Sharing the console - kgdboc



KGDB demonstration setup

- Use a connection multiplexer
  - By default you can only connect one application at a time to the console
  - In the case of kgdboc you want an interactive console & a debug port

  agent-proxy *CONSOLE_PORT*^DEBUG_PORT IP_ADDR PORT
- More or less turns your local serial port into a terminal server

  agent-proxy 2223^2222 0 /dev/ttyS0,115200
- Use it to multiplex a remote terminal server or simulator connection

  agent-proxy 2223^2222 128.224.50.38 8181

- The agent-proxy will be available on kgdb.wiki.kernel.org later before LinuxCon 2010.

KGDB demonstration

- On the target system
  - echo ttyS0 > /sys/module/kgdboc/parameters/kgdboc
  - insmod test_panic.ko
- In gdb
  - tar remote localhost:2222
  - break sys_sync
  - c
- On the target
  - sync
- In gdb
  - awatch tp_address_ref
  - inf br
  - c
- On the target
  - echo 1 > /proc/test_panic/bad_access
- Back to gdb where we can pass along the exception
- signal 9

GDB and unoptimizing

- Unoptimizing files for understanding (Source stepping)
  - If you have a single file you are interested in, un-optimize it by editing the Makefile

Exmaple: un-optimize kernel/fork.c – Edit kernel/Makefile

- Insert the line at the top:

  CFLAGS_fork.o += -O0

- For an entire directory of files, insert at the top of the Makefile

  EXTRA_CFLAGS += -O0

- Don't UNOPTIMZE the whole kernel – Some specific IP routines require the kernel to be optimized or the kernel stack can overflow, or in the case of ARM v5, NFS will not work reliably. This is due to compiler/linux specific hacks.

- You can make use of this to get stack traces which include static functions that are typically squashed by the compiler or make the debugger stop jumping around

GDB and module debugging

- Use a gdb that supports kernel modules
  - Link all your modules to the same directory as the vmlinux file such that gdb will automatically load the symbolic version correctly

```
cd linux-2.6-dir
for e in `find . -name \*.ko `; do ln -s $e . ; done
```

- Without a kernel module aware gdb, use kgdb+kdb:

```
monitor lsmod
add-symbol-file kernel_module.ko 0xaddress
```

Still have a timing problem?

Remember the kernel debugger can easily modify memory

- Try Read/Write additional counters

  … got_here1++ … got_here2++ ...

- Use a conditional variable to control a printk()

  If (global_doit) { printk("state info ..."); }

- Use a conditional to execute a variable++

  If (global_doit) { got_here1++; }

NOTE: Any new code you insert is also a place you can set a breakpoint or conditional breakpoint

Internal KGDB variables

- Q: What processor am I on? A: value of counter in kgdb_active

```
(gdb) p kgdb_active
$1 = {counter = 0}
```

- kgdb_info contains all the individual cpu state of the master and slave cpus

```
(gdb) ptype kgdb_info
type = struct debuggerinfo_struct {
    void *debuggerinfo;
    struct task_struct *task;
} [8]
(gdb) p kgdb_info[0]
$2 = {debuggerinfo = 0xc7829f04, task = 0xc7824000}
(gdb) p kgdb_info[1]
$3 = {debuggerinfo = 0xc7843f6c, task = 0xc7825c00}
(gdb) print kgdb_info[0].task.comm
$4 = "sh\000t\000er\000\000\000\000\000\000\000\000"
```

kgdbwait

- kgdbwait as a kernel argument will stop as early as the I/O driver supports
- kgdbwait is useful for attaching early and setting early breakpoints for kernel initialization, even for kernel module loading
- You must have a built in kgdb I/O module which you also configure via the kernel boot argument

  Example: kgdboc=ttyS0,115200 kgdbwait

The sys_sync breakpoint

- A “nice” place to put a break point is sys_sync
- This allows you to easily enter the debugger by typing “sync” on the command line
- You can use this if you have trouble with sysrq
- Of course you could always execute from a root shell:

```
echo g > /proc/sysrq-trigger
```

X86 has hardware / data breakpoints

- A quick example of gdb and a data write breakpoint
  - Boot with kgdbwait
  - watch system_state!=SYSTEM_BOOTING
  - disas /m $pc-32 $pc+32
- You can also use rwatch (read) and awatch (access read/write)
- “info watchpoints” tells you what you set
- hbreak is the gdb HW equivelent of break
- Remember you have only 4 HW breakpoints on x86

Simulators

- Today 90% of kgdb / kdb development uses simulators
- Simulators are better than kgdb / kdb
  - you can debug things like the exception vectors
  - If the simulator cannot simulate your proprietary hardware it is probably of little value.
- QEMU/KVM, VMware, Virtual Box, and Simics all have a connector for gdb
- Quick demo of stepping through kgdb

  target remote localhost:1234

  break kgdb_handle_exception

Simics

- Sometimes the most tricky part of a problem is reproducing the problem (1 of 100 runs for example)
- Duplicate it 1 time in simics and run forward and backward until you find the problem!
- Demonstration of rewinding time after a crash

```
target remote localhost:9123
monitor enable-reverse-execution
break at_crash_address
reverse-continue
reverse-finish
step
```

References

- KGDB/KDB Website

  http://kgdb.wiki.kernel.org

- KGDB/KDB Mailing list
  - kgdb-bugreport@lists.sourceforge.net
  - https://lists.sourceforge.net/lists/listinfo/kgdb-bugreport
- If you contact Wind River in regard to one of the tools in this presentation please be sure to mention you watched the kernel debugging webinar.

WIND RIVER