Kernel-Level Debuggers : KGDB

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license**.
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

<< Source: "Linux Device Drivers" by J Corbet, A Rubini & GK Hartman, 3rd Ed., O'Reilly >>

Using gdb (for kernel-space)

gdb can be quite useful for looking at the system internals. Proficient use of the debugger at this level requires some confidence with gdb commands, some understanding of assembly code for the target platform, and the ability to match source code and optimized assembly.

The debugger must be invoked as though the kernel were an application. In addition to specifying the filename for the ELF kernel image, you need to provide the name of a core file on the command line. For a running kernel, that core file is the kernel core image, /proc/kcore. A typical invocation of gdb looks like the following:

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

[Note that you should be running the kernel that the vmlinux image corresponds to].

The first argument is the name of the uncompressed ELF kernel executable, not the zImage or bzImage or anything built specifically for the boot environment.

The second argument on the gdb command line is the name of the core file. Like any file in /proc, /proc/kcore is generated when it is read. When the read system call executes in the /proc filesystem, it maps to a data-generation function rather than a data retrieval one; we've already exploited this feature in the section "Using the /proc Filesystem" earlier in this chapter. kcore is used to represent the kernel "executable" in the format of a core file; it is a huge file, because it represents the whole kernel address space, which corresponds to all physical memory. From within gdb, you can look at kernel variables by issuing the standard gdb commands. For example, **`p jiffies`** prints the number of clock ticks from system boot to the current time.

When you print data from gdb, the kernel is still running, and the various data items have different values at different times; gdb, however, optimizes access to the core file by caching data that has already been read. If you try to look at the jiffies variable once again, you'll get the same answer as before. Caching values to avoid extra disk access is a correct behavior for conventional core files but is inconvenient when a "dynamic" core image is used. The solution is to issue the command **`core-file /proc/kcore`** whenever you want to flush the gdb cache; the debugger gets ready to use a new core file and discards any old information. You won't, however, always need to issue core-file when reading a new datum; gdb reads the core in chunks of a few kilobytes and caches only chunks it has already referenced.

Numerous capabilities normally provided by gdb are not available when you are working with the kernel. For example, gdb is not able to modify kernel data; it expects to be running a program to be debugged under its own control before playing with its memory image. It is also not possible to set breakpoints or watchpoints, or to single-step through kernel functions.

Note that, in order to have symbol information available for gdb, you must compile your kernel with the CONFIG_DEBUG_INFO option set. The result is a far larger kernel image on disk, but, without that information, digging through kernel variables is almost impossible.

<<

Sample Session:

```
# gdb <...>/linux-2.6.17/vmlinux /proc/kcore
GNU gdb Red Hat Linux (6.3.0.0-1.96rh)
[...]
Core was generated by `ro root=LABEL=/1 rhgb quiet 1'.
#0  0x00000000 in ?? ()
(gdb) p jiffies
$1 = 3700920
(gdb)

## <wait a bit...>

(gdb) p jiffies
$2 = 3700920                                    ## does'nt change
(gdb) core-file /proc/kcore                     ## refresh
Core was generated by `ro root=LABEL=/1 rhgb quiet 1'.
#0  0x00000000 in ?? ()
(gdb) p jiffies
$3 = 3717432
(gdb)
(gdb) p tasklist_lock
$4 = {raw_lock = {<No data fields>}, magic = 3736018669, owner_cpu =
4294967295, owner = 0xffffffff}
(gdb) set print pretty
(gdb) p tasklist_lock
$5 = {
  raw_lock = {<No data fields>},
  magic = 3736018669,
  owner_cpu = 4294967295,
  owner = 0xffffffff
}
(gdb) p task_state_array
$8 = {0xc02c6b9c "R (running)", 0xc02c6ba8 "S (sleeping)", 0xc02c6bb5 "D
(disk sleep)",
  0xc02c6bc4 "T (stopped)", 0xc02c6bd0 "T (tracing stop)", 0xc02c6be1 "Z
(zombie)", 0xc02c6bec "X (dead)"}
(gdb)
```

>>

Hardware Watchpoints

A hardware watchpoint has GDB use CPU hardware registers to “watch” a variable or expression. It’s much faster than a software watchpoint but obviously requires processor support.

How do we know?

```
(gdb) show can-use-hw-watchpoints
Debugger's willingness to use watchpoint hardware is 1.
(gdb)
```

There are three types of (hardware) watchpoint GDB commands:

- **watch** <var/expression> : sets up a hardware watchpoint that triggers on the variable/expression changing (being written to)
- **rwatch** <var/expression> : sets up a hardware watchpoint that triggers on the variable/expression being read
- **awatch** <var/expression> : sets up a hardware watchpoint that triggers on either a read or write on the variable/expression

On triggering, GDB will show the hardware watchpoint number, name and the old and new value of the variable/expression being watched.

An example snippet (when debugging the Linux kernel with KGDB):

```
(gdb) info watchpoints
No watchpoints.
(gdb) watch jiffies<tab><tab>
jiffies                 jiffies_64_to_clock_t   jiffies_till_next_fqs
jiffies_to_timespec
jiffies.c               jiffies_lock            jiffies_till_sched_qs
jiffies_to_timeval
jiffies.h               jiffies_read            jiffies_to_clock_t
jiffies_to_usecs
jiffies_64              jiffies_till_first_fqs  jiffies_to_msecs
(gdb) watch jiffies
Hardware watchpoint 5: jiffies
(gdb) c
Continuing.

[...]

Hardware watchpoint 5: jiffies    << we hit the watchpoint => 'jiffies' has
                                     been written to >>
Old value = 4294938708
New value = 4294938709
do_timer (ticks=1) at kernel/time/timekeeping.c:1674
1674       calc_global_load(ticks);
(gdb) bt
#0  do_timer (ticks=1) at kernel/time/timekeeping.c:1674
#1  0x8007fe7c in tick_periodic (cpu=<optimized out>) at kernel/time/tick-
common.c:86
```

```
#2  0x8008004c in tick_handle_periodic (dev=0x8fdf1e00) at
kernel/time/tick-common.c:103
#3  0x80015da4 in twd_handler (irq=<optimized out>, dev_id=<optimized out>)
at arch/arm/kernel/smp_twd.c:236
#4  0x80067904 in handle_percpu_devid_irq (irq=29, desc=0x8f805600) at
kernel/irq/chip.c:711
#5  0x800635dc in generic_handle_irq_desc (desc=<optimized out>,
irq=<optimized out>)
    at include/linux/irqdesc.h:128
#6  generic_handle_irq (irq=29) at kernel/irq/irqdesc.c:351
#7  0x80063920 in __handle_domain_irq (domain=0x8f803000, hwirq=<optimized
out>, lookup=true,
    regs=<optimized out>) at kernel/irq/irqdesc.c:388
#8  0x800086e4 in handle_domain_irq (regs=<optimized out>, hwirq=<optimized
out>, domain=<optimized out>)
    at include/linux/irqdesc.h:146
#9  gic_handle_irq (regs=0x1 <__vectors_start>) at drivers/irqchip/irq-
gic.c:273
#10 0x80013844 in __irq_svc () at arch/arm/kernel/entry-armv.S:205
Backtrace stopped: frame did not save the PC
(gdb) c
Continuing.
Hardware watchpoint 5: jiffies    << we hit the same watchpoint again >>

Old value = 4294938709
New value = 4294938710
do_timer (ticks=1) at kernel/time/timekeeping.c:1674
1674       calc_global_load(ticks);
(gdb) info watchpoints
Num     Type           Disp Enb Address    What
5       hw watchpoint  keep y              jiffies
      breakpoint already hit 2 times
(gdb) delete 5
(gdb) rwatch jiffies   << setup a 'read' watchpoint on 'jiffies' >>
Hardware read watchpoint 6: jiffies
(gdb) c
Continuing.
Hardware read watchpoint 6: jiffies   << we hit the read watchpoint =>
                                         'jiffies' has been read from >>
Value = 4294938710
0x8004f70c in calc_global_load (ticks=1) at kernel/sched/proc.c:347
347        if (time_before(jiffies, calc_load_update + 10))
(gdb) bt
#0  0x8004f70c in calc_global_load (ticks=1) at kernel/sched/proc.c:347
#1  0x8007aa2c in do_timer (ticks=<optimized out>) at
kernel/time/timekeeping.c:1674
#2  0x8007fe7c in tick_periodic (cpu=<optimized out>) at kernel/time/tick-
common.c:86
#3  0x8008004c in tick_handle_periodic (dev=0x8fdf1e00) at
kernel/time/tick-common.c:103
#4  0x80015da4 in twd_handler (irq=<optimized out>, dev_id=<optimized out>)
at arch/arm/kernel/smp_twd.c:236
```

```
#5  0x80067904 in handle_percpu_devid_irq (irq=29, desc=0x8f805600) at
kernel/irq/chip.c:711
#6  0x800635dc in generic_handle_irq_desc (desc=<optimized out>,
irq=<optimized out>)
    at include/linux/irqdesc.h:128
#7  generic_handle_irq (irq=29) at kernel/irq/irqdesc.c:351
#8  0x80063920 in __handle_domain_irq (domain=0x8f803000, hwirq=<optimized
out>, lookup=true,
    regs=<optimized out>) at kernel/irq/irqdesc.c:388
#9  0x800086e4 in handle_domain_irq (regs=<optimized out>, hwirq=<optimized
out>, domain=<optimized out>)
    at include/linux/irqdesc.h:146
#10 gic_handle_irq (regs=0x1 <__vectors_start>) at drivers/irqchip/irq-
gic.c:273
#11 0x80013844 in __irq_svc () at arch/arm/kernel/entry-armv.S:205
Backtrace stopped: frame did not save the PC
(gdb) info watchpoints
Num     Type           Disp Enb Address    What
6       read watchpoint keep y              jiffies
     breakpoint already hit 1 time
(gdb) delete 6
(gdb) c
Continuing.
```

Ref [SO]: Can I set a breakpoint on 'memory access' in GDB?

Kernel Modules and gdb

<<
Note- recent kernel doc:
Debugging kernel and modules via gdb
>>

With the debugging information available, you can learn a lot about what is going on inside the kernel. gdb happily prints out structures, follows pointers, etc. One thing that is harder, however, is examining modules. Since modules are not part of the vmlinux image passed to gdb , the debugger knows nothing about them. Fortunately, as of kernel 2.6.7, it is possible to teach gdb what it needs to know to examine loadable modules.

Linux loadable modules are ELF-format executable images; as such, they have been divided up into numerous sections. A typical module can contain a dozen or more sections, but there are typically three that are relevant in a debugging session:

.text
: This section contains the executable code for the module. The debugger must know where this section is to be able to give tracebacks or set breakpoints. (Neither of these operations is relevant when running the debugger on /proc/kcore , but they can useful when working with kgdb).

.bss
.data
: These two sections hold the module's variables. Any variable that is not initialized at compile time ends up in .bss , while those that are initialized go into .data .

Making gdb work with loadable modules requires informing the debugger about where a given module's sections have been loaded. That information is available in sysfs, under */sys/module/<module-name>/sections* . For example, after loading the scull module, the directory */sys/module/scull/sections* contains files with names such as .text ; the content of each file is the base address for that section.

<<
Note!
You require root access to retrieve data from */sys/module/<module-name>/sections/.<file>*
>>

We are now in a position to issue a gdb command telling it about our module. The command we need is *add-symbol-file* ; this command takes as parameters the name of the module object file, the .text base address, and a series of optional parameters describing where any other sections of interest have been put. After digging through the module section data in sysfs, we can construct a command such as:

```
(gdb) help add-symbol-file
```

```
Load symbols from FILE, assuming FILE has been dynamically loaded.
Usage: add-symbol-file FILE ADDR [-s <SECT> <SECT_ADDR> -s <SECT>
<SECT_ADDR> ...]
ADDR is the starting address of the file's text.
The optional arguments are section-name section-address pairs and
should be specified if the data and bss segments are not contiguous
with the text.  SECT is a section name to be loaded at SECT_ADDR.
(gdb) add-symbol-file <...>/scull.ko 0xd0832000 \
-s .bss 0xd0837100 \
-s .data 0xd0836be0
```

We have included a small script in the sample source (gdbline.sh) that can create this command for a given module.

We can now use gdb to examine variables in our loadable module. Here is a quick example taken from a scull debugging session:

```
(gdb) add-symbol-file scull.ko 0xd0832000 \
-s .bss 0xd0837100 \
-s .data 0xd0836be0
add symbol table from file "scull.ko" at
     .text_addr = 0xd0832000
     .bss_addr = 0xd0837100
     .data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
(gdb) p scull_devices[0]
$1 = {data = 0xcfd66c50,
quantum = 4000,
qset = 1000,
size = 20881,
access_key = 0,
...}
```

Here we see that the first scull device currently holds 20,881 bytes. If we wanted, we could follow the data chain, or look at anything else of interest in the module.

<< See (and try out) the hello_gdb kernel module >>
Sample Session:

Ensure that the Makefile includes the '-g' option:

```
~~EXTRA_CFLAGS += -DDEBUG -g~~
ccflags-y += -DDEBUG -g

# make
make -C /lib/modules/2.6.17/build SUBDIRS=/mnt/.../hello_gdb modules
make[1]: Entering directory `/mnt/data_5GB/kbuild/linux-2.6.17'
  CC [M]  /mnt/.../hello_gdb/hello_gdb.o
/mnt/.../hello_gdb/hello_gdb.c: In function `procread':
```

```
/mnt/.../hello_gdb/hello_gdb.c:66: warning: unused variable `p'
  Building modules, stage 2.
  MODPOST
  CC      /mnt/.../hello_gdb/hello_gdb.mod.o
  LD [M]  /mnt/.../hello_gdb/hello_gdb.ko
make[1]: Leaving directory `/mnt/data_5GB/kbuild/linux-2.6.17'
#
# insmod hello_gdb.ko
```

Here's a helper shell script that can construct the appropriate gdb "**add-symbol-file**" command that you'll use within gdb to help debug kernel modules:

```
# cat gdbline.sh
[ ... ]
cd /sys/module/$1/sections
echo "Copy-paste the following lines into GDB"
echo "---snip---"

# Don't issue newlines and the continuation \ ; results in GDB err:
# Unrecognized argument "
# "
[[ -f .text ]] && {
   sudo echo -n "add-symbol-file $2 `/bin/cat .text` "
   #sudo echo  " \\"
} || [[ -f .init.text ]] && {
   sudo echo -n "-s .init.text `/bin/cat .init.text` "
}

for section in .[a-z]* *; do
    if [[ ${section} != ".text" || ${section} != ".init.text" ]]; then
      sudo echo -n " -s" ${section} `/bin/cat ${section}`
    fi
done
echo "
---snip---"
$ ./gdbline.sh
Usage: ./gdbline.sh module-name image-filename
  module-name: name of the (already inserted) kernel module (without
the .ko)
  image-filename: pathname to the kernel module.

$ sudo ./gdbline.sh hello_gdb ./hello_gdb.ko
Copy-paste the following lines into GDB
---snip---
add-symbol-file ./hello_gdb.ko 0xffffffffc386d000 -s .init.text
0xffffffffc1027000  -s .bss 0xffffffffc386f380 -s .gnu.linkonce.this_module
0xffffffffc386f000 -s .init.text 0xffffffffc1027000 -s .note.gnu.build-id
0xffffffffc386e000 -s .note.Linux 0xffffffffc386e024 -s .return_sites
0xffffffffc386e0b6 -s .rodata.str1.1 0xffffffffc386e054 -s .strtab
0xffffffffc10284b0 -s .symtab 0xffffffffc1028000 -s .text
0xffffffffc386d000 -s __mcount_loc 0xffffffffc386e0a6
```

```
---snip---
```

Copy the above output from the shell script (the lines between --snip--).

```
hello_gdb # ./gdbline.sh hello_gdb ./hello_gdb.ko
Copy-paste the following lines into GDB
---snip---
add-symbol-file ./hello_gdb.ko 0xffffffffc09f5000 -s .init.text 0xffffffffc0a01000  -s .bss 0xffffffffc09f7400 -s .gnu.
linkonce.this_module 0xffffffffc09f7000 -s .init.text 0xffffffffc0a01000 -s .note.gnu.build-id 0xffffffffc09f6000 -s .n
ote.Linux 0xffffffffc09f6024 -s .return_sites 0xffffffffc09f609e -s .rodata.str1.1 0xffffffffc09f603c -s .strtab 0xffff
ffffc0a02498 -s .symtab 0xffffffffc0a02000 -s .text 0xffffffffc09f5000 -s __mcount_loc 0xffffffffc09f608e
---snip---

hello_gdb # gdb -q -c /proc/kcore /lib/modules/5.10.153-mykernel/build/vmlinux
Reading symbols from /lib/modules/5.10.153-mykernel/build/vmlinux...
[New process 1]
Core was generated by `BOOT_IMAGE=/vmlinuz-5.10.153-mykernel root=UUID=b67edd03-5aa9-4826-add9-3de240b'.
#0  0x0000000000000000 in fixed_percpu_data ()
(gdb) add-symbol-file ./hello_gdb.ko 0xffffffffc09f5000 -s .init.text 0xffffffffc0a01000  -s .bss 0xffffffffc09f7400 -s
 .gnu.linkonce.this_module 0xffffffffc09f7000 -s .init.text 0xffffffffc0a01000 -s .note.gnu.build-id 0xffffffffc09f6000
 -s .note.Linux 0xffffffffc09f6024 -s .return_sites 0xffffffffc09f609e -s .rodata.str1.1 0xffffffffc09f603c -s .strtab
0xffffffffc0a02498 -s .symtab 0xffffffffc0a02000 -s .text 0xffffffffc09f5000 -s __mcount_loc 0xffffffffc09f608e
add symbol table from file "./hello_gdb.ko" at
        .text_addr = 0xffffffffc09f5000
        .init.text_addr = 0xffffffffc0a01000
        .bss_addr = 0xffffffffc09f7400
        .gnu.linkonce.this_module_addr = 0xffffffffc09f7000
        .init.text_addr = 0xffffffffc0a01000
        .note.gnu.build-id_addr = 0xffffffffc09f6000
        .note.Linux_addr = 0xffffffffc09f6024
        .return_sites_addr = 0xffffffffc09f609e
        .rodata.str1.1_addr = 0xffffffffc09f603c
        .strtab_addr = 0xffffffffc0a02498
        .symtab_addr = 0xffffffffc0a02000
        .text_addr = 0xffffffffc09f5000
        __mcount_loc_addr = 0xffffffffc09f608e
(y or n) y
Reading symbols from ./hello_gdb.ko...
warning: section .init.text not found in /home/osboxes/kaiwanTECH/L5_debug_trg/kernel_debug/hello_gdb/hello_gdb.ko
warning: section .strtab not found in /home/osboxes/kaiwanTECH/L5_debug_trg/kernel_debug/hello_gdb/hello_gdb.ko
warning: section .symtab not found in /home/osboxes/kaiwanTECH/L5_debug_trg/kernel_debug/hello_gdb/hello_gdb.ko
warning: section .text not found in /home/osboxes/kaiwanTECH/L5_debug_trg/kernel_debug/hello_gdb/hello_gdb.ko
(gdb) p mine
$1 = (MYS *) 0xffff894fc5e8ce48
```

<< NOTE: it's IMP to pass along the path to the (debug) vmlinux ! >>

<< Note: Below, the values are different from those above >>

```
...
(gdb) set print pretty
(gdb) p *mine
$4 = {
  tx = 0,
 Corresponds to jiffies value  rx = 0,
 j = 6128429
}
(gdb) x/16 0xc37f6a60
0xc37f6a60:     0x00000000      0x005d832d      0x5a5a5a5a      0x5a5a5a5a
0xc37f6a70:     0x5a5a5a5a      0x5a5a5a5a      0x5a5a5a5a      0xa55a5a5a
0xc37f6a80:     0x170fc2a5      0xd081e04d      0x170fc2a5      0x00000002
0xc37f6a90:     0x00000001      0xdead4ead      0xffffffff      0xffffffff
(gdb) p/x *mine
$5 = {
  tx = 0x0,
```

```
  rx = 0x0,
  j = 0x5d832d
}
(gdb) q
#
```

One other useful trick worth knowing about is this:

```
(gdb) print *(address)
```

Here, fill in a hex address for address ; the output is a file and line number for the code corresponding to that address. This technique may be useful, for example, to find out where a function pointer really points.

We still cannot perform typical debugging tasks like setting breakpoints or modifying data; to perform those operations, we need to use a tool like kdb or kgdb.

KGDB

The KGDB project allows one to perform *source-level* debugging of the kernel.
From kernel ver 2.6.26, the kgdb facility is built into the mainline kernel; hence, activation is a lot easier (than before when multiple patches had to be applied).

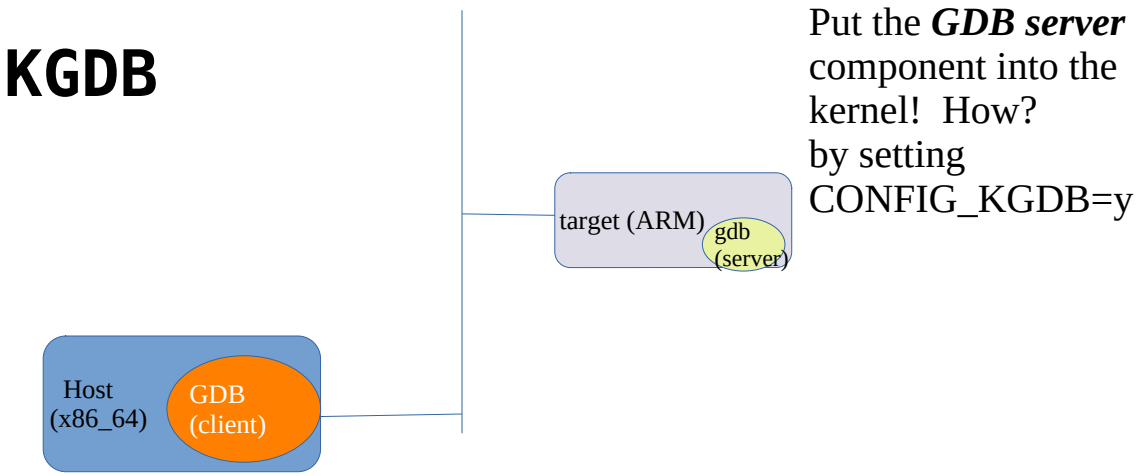
The section below on “Using QEMU and GDB for source-level kernel debugging” is almost identical to how KGDB is used on an actual machine.

Resources on KGDB:

Kernel docs: Using kgdb, kdb and the kernel debugger internals

http://elinux.org/Kgdb

A KDB / KGDB SESSION ON THE POPULAR RASPBERRY PI EMBEDDED LINUX BOARD




```
(gdb) target remote :1234
```

Using QEMU and GDB for source-level kernel debugging

1. Download and install the emulator package, Qemu. http://wiki.qemu.org/Download . (We're using the latest one at the time of this writing, i.e., ver 0.14.1)

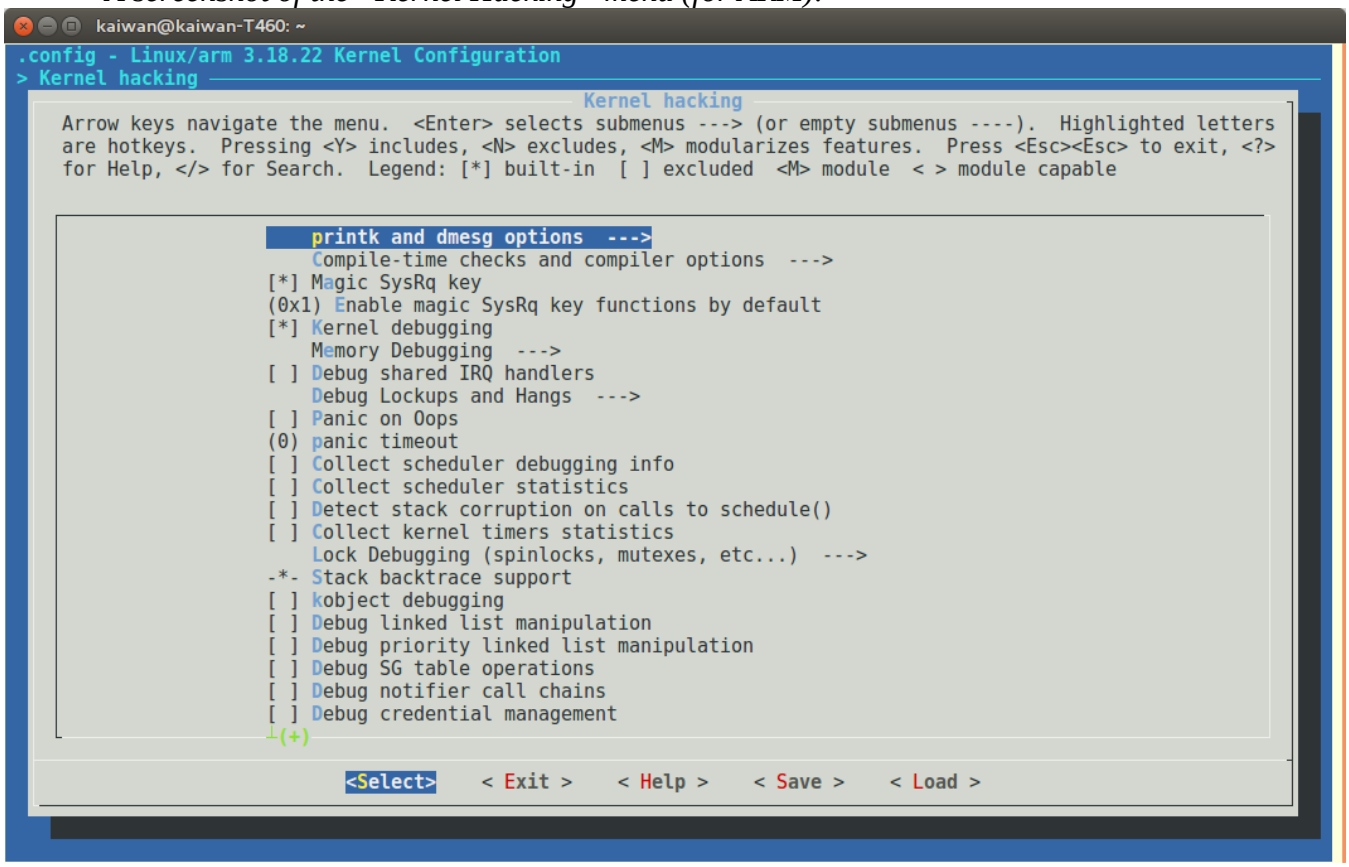
2. We assume native gdb is present (we're using gdb ver 7.3).

Kernel Only Debugging

3. If the intention is to just perform pure kernel source debugging, then we don't even require a root filesystem.

<<

A screenshot of the "Kernel Hacking" menu (for ARM):



Q. Where are the debug "menus" generated from?
A. The Kconfig files of course.
Specifically:
lib/Kconfig.debug

>>

Build your kernel with debug on and frame pointers on.

CONFIG_DEBUG_KERNEL=y << basic support for kernel debug; turns the *Kernel Hacking* menu on >>

CONFIG_DEBUG_INFO=y << build with -g ! includes symbols >>

CONFIG_KGDB=y << KGDB remote debugging support! >>

CONFIG_FRAME_POINTER=y << reccomended >>

<< On recent kernels, when configuring with the "menuconfig" UI, goto:

Kernel Hacking / Compile-time checks and compiler options → Compile the kernel with debug info
>>

and, *optionally (useful)*,

CONFIG_DEBUG_SPINLOCK=y
CONFIG_DEBUG_MUTEXES=y
CONFIG_DEBUG_LOCK_ALLOC=y
CONFIG_PROVE_LOCKING=y
CONFIG_DEBUG_BUGVERBOSE=y
CONFIG_EARLY_PRINTK=y

Kernel Command-line Parameters relevant to KGDB
(see *Documentation/kernel-parameters.txt)*

```
...
      ekgdboc=    [X86,KGDB] Allow early kernel console debugging
                  ekgdboc=kbd

...

      kgdbdbgp=   [KGDB,HW] kgdb over EHCI usb debug port.
                  Format: <Controller#>[,poll interval]
                  The controller # is the number of the ehci usb debug
                  port as it is probed via PCI.  The poll interval is
                  optional and is the number seconds in between
                  each poll cycle to the debug port in case you need
                  the functionality for interrupting the kernel with
                  gdb or control-c on the dbgp connection.  When
                  not using this parameter you use sysrq-g to break into
                  the kernel debugger.

      kgdboc=     [KGDB,HW] kgdb over consoles.
                  Requires a tty driver that supports console polling,
                  or a supported polling keyboard driver (non-usb).
```

```
                    Serial only format: <serial_device>[,baud]
                    keyboard only format: kbd
                    keyboard and serial format: kbd,<serial_device>[,baud]
                   Optional Kernel mode setting:
                    kms, kbd format: kms,kbd
                    kms, kbd and serial format: kms,kbd,<ser_dev>[,baud]

      kgdbwait    [KGDB] Stop kernel execution and enter the
                  kernel debugger at the earliest opportunity.
...
```

Using KGDB via a AMD64 Qemu/KVM guest and host

Ref: ***Linux Kernel Exploitation 0x0] Debugging the Kernel with QEMU***
posted by Keith Makan

A few tips:

- Install all required packages first:

```
sudo apt-get update
sudo apt-get upgrade

sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils
libssl-dev bc flex libelf-dev bison debootstrap qemu-system-x86
```

- Once the kernel's built, test the Qemu guest VM (leave out the -s -S which makes it wait for gdb to connect); once it works, reboot but this time with the '-S -s' Qemu options:

<<
-S Do not start CPU at startup (you must type 'c' in the monitor).
-s Shorthand for -gdb tcp::1234, i.e. open a gdbserver on TCP port 1234.
>>

```
$ cat run.sh
#!/bin/bash
# ref: http://blog.k3170makan.com/2020/11/linux-kernel-exploitation-
0x0-debugging.html
qemu-system-x86_64 \
 -kernel ../linux-5.10.3/arch/x86/boot/bzImage \
 -append "console=ttyS0 root=/dev/sda earlyprintk=serial nokaslr" \
 -hda ./stretch.img \
 -net user,hostfwd=tcp::10021-:22 -net nic   \
 -enable-kvm   \
 -nographic   \
```

```
-m 2G \
-smp 2  \
-S -s   \
-pidfile vm.pid \
2>&1 | tee vm.log
```

It should bootup; you're logged in as root (no passwd).
Shut down and continue...

NOTE

- *You CANNOT run another hypervisor (or Docker container) along with Qemu/KVM with the -enable-kvm (accelerator) option !!*
  If VirtualBox (or another) is running, and you want to use *-enable-kvm,* first shut it down and then retry...

- Before running GDB on the 'host', cd to the relevant kernel source directory (here it's 5.10.3)... else, you won;t see the kernel source (with list, ...)

- Setting hardware breakpoints (with 'hb <func>' is considered a safer bet

- to quit out of Qemu, type Ctrl-a x

Host: run the above script:

```
kdebug_kmakan $ ./run.sh
Note:
1. First shut down any other hypervisor instance
2. once run, the guest qemu system will *wait* for GDB to connect from the host:
On the host, do:

       $ gdb linux-5.10.3/vmlinux -q
       (gdb) target remote :1234

qemu-system-x86_64 -kernel linux-5.10.3/arch/x86/boot/bzImage -append
console=ttyS0 root=/dev/sda earlyprintk=serial nokaslr -s
WARNING: Image format was not specified for 'image/stretch.img' and probing
guessed raw.
         Automatically detecting the format is dangerous for raw images, write
operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
qemu-system-x86_64: warning: host doesn't support requested feature:
CPUID.80000001H:ECX.svm [bit 2]
qemu-system-x86_64: warning: host doesn't support requested feature:
CPUID.80000001H:ECX.svm [bit 2]

<< ... kernel GDB server is waiting for GDB client to connect ... >>
```

From another terminal window, connect via GDB client:

```
$ cd <...>/linux-5.10.3
$ gdb -q vmlinux
Reading symbols from vmlinux...
(gdb) target remote :1234
Remote debugging using :1234
0x000000000000fff0 in gdt_page ()
(gdb)
```

It's worked!

*Try it: set **breakpoints** (on kmem_cache_alloc(), schedule_timeout(), etc) and see...*

```
(gdb) b kmem_cache_alloc
Breakpoint 3 at 0xffffffff8184a740: file mm/slub.c, line 2903.
(gdb) c
Continuing.
[Switching to Thread 1.2]
Cannot remove breakpoints because program is no longer writable.
Further execution is probably impossible.

Thread 2 hit Breakpoint 3, kmem_cache_alloc (s=0xffff888006217280,
gfpflags=gfpflags@entry=3264) at mm/slub.c:2903
2903  {
(gdb) bt
#0  kmem_cache_alloc (s=0xffff888006217280, gfpflags=gfpflags@entry=3264) at
mm/slub.c:2903
#1  0xffffffff8191b3cd in getname_flags (empty=0x0 <fixed_percpu_data>, flags=1,
filename=0x4ccca3 ".") at fs/namei.c:138
#2  getname_flags (filename=0x4ccca3 ".", flags=1, empty=0x0 <fixed_percpu_data>)
at fs/namei.c:128
#3  0xffffffff8191ec94 in user_path_at_empty (dfd=dfd@entry=-100,
name=name@entry=0x4ccca3 ".", flags=flags@entry=1,
    path=path@entry=0xffff888009e07d60, empty=empty@entry=0x0 <fixed_percpu_data>)
at fs/namei.c:2647
#4  0xffffffff818f1ef3 in user_path_at (path=0xffff888009e07d60, flags=1,
name=0x4ccca3 ".", dfd=-100) at ./include/linux/namei.h:59
#5  vfs_statx (dfd=dfd@entry=-100, filename=filename@entry=0x4ccca3 ".",
flags=flags@entry=2048, stat=stat@entry=0xffff888009e07e08,
    request_mask=request_mask@entry=2047) at fs/stat.c:185
#6  0xffffffff818f330d in vfs_fstatat (flags=0, stat=0xffff888009e07e08,
filename=0x4ccca3 ".", dfd=-100) at fs/stat.c:207
#7  vfs_stat (stat=0xffff888009e07e08, filename=0x4ccca3 ".") at
./include/linux/fs.h:3121
#8  __do_sys_newstat (filename=0x4ccca3 ".", statbuf=0x7ffc6ca64730) at
fs/stat.c:349
#9  0xffffffff818f33d9 in __se_sys_newstat (statbuf=<optimized out>,
filename=<optimized out>) at fs/stat.c:345
#10 __x64_sys_newstat (regs=0xffff888009e07f58) at fs/stat.c:345
#11 0xffffffff833aefb8 in do_syscall_64 (nr=<optimized out>,
regs=0xffff888009e07f58) at arch/x86/entry/common.c:46
#12 0xffffffff8340008c in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:120
#13 0x00000000025ee508 in ?? ()
#14 0x00007ffc6ca64730 in ?? ()
```

```
#15 0x0000000000000024 in fixed_percpu_data ()
#16 0x00000000025e9278 in ?? ()
#17 0x00000000025ee508 in ?? ()
#18 0x0000000000000000 in ?? ()
(gdb)
```

Source code's visible and step-able !

```
(gdb) l
2898  {
2899        return slab_alloc_node(s, gfpflags, NUMA_NO_NODE, addr);
2900  }
2901
2902  void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags)
2903  {
2904        void *ret = slab_alloc(s, gfpflags, _RET_IP_);
2905
2906        trace_kmem_cache_alloc(_RET_IP_, ret, s->object_size,
2907                               s->size, gfpflags);
(gdb)
...
```

 # Added a breakpoint on schedule_timeout()

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0xffffffff8184a740 in kmem_cache_alloc at mm/slub.c:2903
2       breakpoint     keep y   0xffffffff833d1c70 in schedule_timeout at kernel/time/timer.c:1833
        breakpoint already hit 8 times
(gdb) c
Continuing.
Cannot remove breakpoints because program is no longer writable.
Further execution is probably impossible.

Thread 2 hit Breakpoint 2, schedule_timeout (timeout=timeout@entry=125) at kernel/time/timer.c:1833
1833    {
(gdb) bt
#0  schedule_timeout (timeout=timeout@entry=125) at kernel/time/timer.c:1833
#1  0xffffffff81750c17 in freezable_schedule_timeout (timeout=<optimized out>) at ./include/linux/freezer.h:192
#2  kcompactd (p=p@entry=0xffff88807ffb5000) at mm/compaction.c:2818
#3  0xffffffff812152e4 in kthread (_create=0xffff888006b62000) at kernel/kthread.c:292
#4  0xffffffff81006642 in ret_from_fork () at arch/x86/entry/entry_64.S:296
#5  0x0000000000000000 in ?? ()
(gdb)
```

...

Tip

Also, it's common practise to setup commonly used gdb commands as macros and save them in the gdb initialization file `~/.gdbinit`.

For example:

```
$ cat ~/.gdbinit
set history save on
set history filename ~/.gdb_history
```

```
set output-radix 16
# Careful! disable security protection, auto-load vmlinux-gdb.py
set auto-load safe-path /

#------------------------------------------------------------------------
# connect
define connect_serial
    set remotebaud 115200
    target remote /dev/ttyS0
    hb panic                    << hb = hardware breakpoint >>
    hb sys_sync
end
define connect_qemu
    target remote :1234
    hb start_kernel
    hb panic
    hb sys_sync
end

...

$
```

SIDEBAR

At times, gdb cannot detect the value of variables and results in:

```
(gdb) p var
$1 = <value optimized out>
```

In these situations, compiling at a high level of optimization might have caused these issues. The kernel is built at '-O2'. Since we cannot realistically rebuild the kernel without optimization, what's to be done?

The only realistic way to debug then is to view the disassembly listing along with the relevant 'C' source and debug. This is easily achieved by entering gdb's TUI mode (notes were provided in the earlier module on gdb usage).

The trick is then to view the local variables and parameters as register values according to the calling convention (and compiler) for that processor architecture.

Also, the 'si' gdb command can be very useful here (when stepping through assembly): step forward a single instruction.

GDB TUI Mode Quick Ref Table

Keyboard key	Action
^X A	Toggle between “regular” GDB and TUI mode
^X 2	Toggle between different TUI layouts: <source-assembly> to <source-registers> to

	<registers-assembly> to <registers-source> to <source-assembly>.
^P	Recall previous command (history) (equivalent to up arrow) at gdb prompt
^N	Recall next command (history) (equivalent to down arrow) at gdb prompt

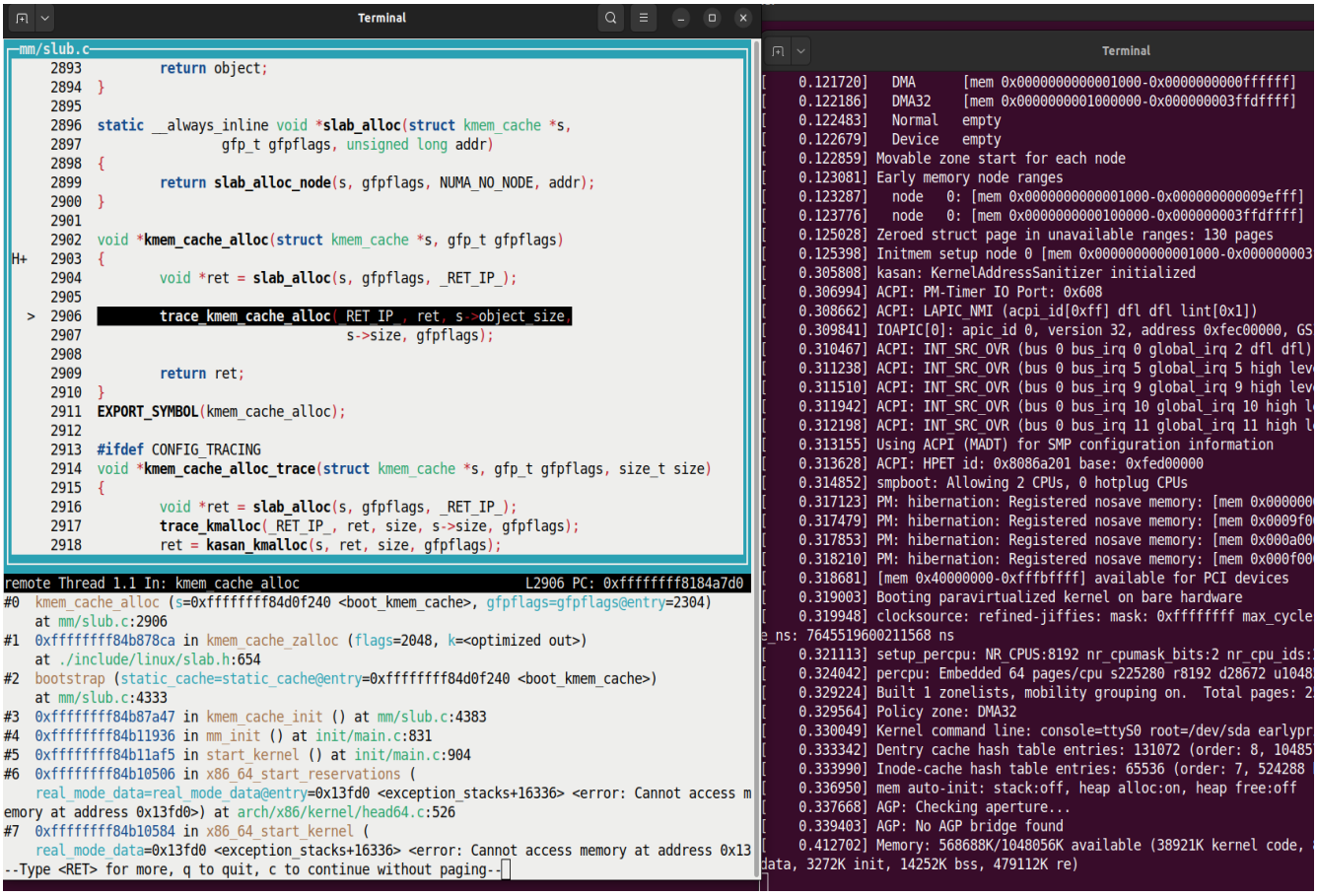
^ = Ctrl

Note: Linux kernel configurable:
lib/Kconfig.debug

```
...
config DEBUG_INFO_DWARF4
   bool "Generate dwarf4 debuginfo"
   depends on DEBUG_INFO
   help
     Generate dwarf4 debug info. This requires recent versions
     of gcc and gdb. It makes the debug information larger.
     But it significantly improves the success of resolving
     variables in gdb on optimized code.
...
```

- If we want to have (or debug) a functioning system with libraries and applications, then obviously we require to build and have QEMU recognize a root filesystem (rfs) image. See the section below which details how to do this.

  Sample screenshot of our KGDB session with a Qemu-emulated x86_64 Debian guest (left terminal, white bg) and a x86_64 Ubuntu host (right terminal, dark colour):



Sample Session 2 : KGDB on a QEMU ARM System

```
$ qemu-system-arm --help
QEMU emulator version 2.5.0 (Debian 1:2.5+dfsg-5ubuntu10.6), Copyright (c)
2003-2008 Fabrice Bellard
usage: qemu-system-arm [options] [disk_image]

--snip--

-S              freeze CPU at startup (use 'c' to start execution)
...
-gdb dev        wait for gdb connection on 'dev'
-s              shorthand for -gdb tcp::1234
$ cat ./run_kgdb_qemu.sh
...
#####
## UPDATE for your box
STG=~/scratchpad/SEALS_staging/
#####

ARMPLAT=vexpress-a9  ## make sure it's right! ##
PORT=1235
```

```
qemu-system-arm -m 256 -M ${ARMPLAT} -kernel $1 \
      -drive file=${STG}/images/rfs.img,if=sd,format=raw \
      -append "console=ttyAMA0 root=/dev/mmcblk0 init=/sbin/init" -
nographic \
      -gdb tcp::${PORT} -S
...
$ ./run_kgdb_qemu.sh </path/to/>zImage

REMEMBER this qemu instance is run w/ the -S QEMU switch: it *waits* for a
gdb client to connect to it...

You are expected to run (in another terminal window):
$ arm-none-linux-gnueabi-gdb <path-to-ARM-built-kernel-src-tree>/vmlinux  #
<--built w/ -g
...
# and then have gdb connect to the target kernel using
(gdb) target remote :1235
...

<< On the ARM target >>
pulseaudio: set_sink_input_volume() failed
pulseaudio: Reason: Invalid argument
pulseaudio: set_sink_input_mute() failed
pulseaudio: Reason: Invalid argument
<< it's waiting for GDB to connect >>
```

In another terminal window:

```
<< On the x86 host >>

$ cd <ARM-kernel-source-tree>
$ ls -lh vmlinux
-rwxr-xr-x 1 root root 78M Jan 19 12:46 vmlinux
$ file vmlinux
vmlinux: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically
linked, BuildID[sha1]=abc..., not stripped
$
$ arm-none-linux-gnueabi-gdb ./vmlinux
GNU gdb (Sourcery CodeBench Lite 2014.05-29) 7.7.50.20140217-cvs
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
...
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vmlinux...done.
(gdb) target remote :1234
:1234: Connection timed out.  << Oops, we gave the wrong port# (see above)
>>
(gdb) target remote :1235      << attempt to connect to the (QEMU) ARM
target >>
```

```
Remote debugging using :1235
0x60000000 in ?? ()             << connected! >>
(gdb) bt
#0  0x60000000 in ?? ()
(gdb) b vfs_write         << set a breakpoint >>
Breakpoint 1 at 0x80139a00: file fs/read_write.c, line 519.
(gdb) c
Continuing.


<< On the ARM target >>
...
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Linux version 3.18.22-kgdb (root@kaiwan-T460) (gcc version
4.8.3 20140320 (prerelease) (Sourcery CodeBench Lite 2014.05-29) ) #2 SMP
Thu Jan 19 12:46:36 IST 2017
[    0.000000] CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7),
cr=10c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing
instruction cache
[    0.000000] Machine: ARM-Versatile Express
[    0.000000] Memory policy: Data cache writeback
[    0.000000] CPU: All CPU(s) started in SVC mode.
[    0.000227] sched_clock: 32 bits at 24MHz, resolution 41ns, wraps every
178956969942ns
[    0.001330] PERCPU: Embedded 11 pages/cpu @8fde9000 s12352 r8192 d24512
u45056
[    0.003103] Built 1 zonelists in Zone order, mobility grouping on.
Total pages: 65024
[    0.003234] Kernel command line: console=ttyAMA0 root=/dev/mmcblk0
init=/sbin/init
[    0.003942] PID hash table entries: 1024 (order: 0, 4096 bytes)
[    0.004142] Dentry cache hash table entries: 32768 (order: 5, 131072
bytes)
[    0.004510] Inode-cache hash table entries: 16384 (order: 4, 65536
bytes)
[    0.008112] Memory: 244816K/262144K available (6163K kernel code, 339K
rwdata, 1872K rodata, 436K init, 6169K bss, 17328K reserved)
[    0.008229] Virtual kernel memory layout:
[    0.008229]     vector  : 0xffff0000 - 0xffff1000   (   4 kB)
[    0.008229]     fixmap  : 0xffc00000 - 0xffe00000   (2048 kB)
[    0.008229]     vmalloc : 0x90800000 - 0xff000000   (1768 MB)
[    0.008229]     lowmem  : 0x80000000 - 0x90000000   ( 256 MB)
[    0.008229]     modules : 0x7f000000 - 0x80000000   (  16 MB)
[    0.008229]       .text : 0x80008000 - 0x807e0edc   (8036 kB)
[    0.008229]       .init : 0x807e1000 - 0x8084e000   ( 436 kB)
[    0.008229]       .data : 0x8084e000 - 0x808a2f38   ( 340 kB)
[    0.008229]        .bss : 0x808a2f38 - 0x80ea93f4   (6170 kB)
[    0.011639] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1

--snip--
```

```
 /devices/mb:kmi1/serio1/input/input2
[    1.927263] EXT3-fs (mmcblk0): error: couldn't mount because of
unsupported optional features (240)
[    1.935603] EXT2-fs (mmcblk0): error: couldn't mount because of
unsupported optional features (240)
[    1.968311] EXT4-fs (mmcblk0): mounted filesystem with ordered data
mode. Opts: (null)
[    1.969544] VFS: Mounted root (ext4 filesystem) readonly on device
179:0.
[    1.980846] Freeing unused kernel memory: 436K (807e1000 - 8084e000)
[    2.143933] random: nonblocking pool is initialized
<< breakpoint hit, waiting to be continued >>


<< On the x86 host >>
Breakpoint 1, vfs_write (file=0x8fbc7780, buf=0xf8f88 "/etc/init.d/rcS
running now ...\n", count=32, pos=0x8f9eff78)
    at fs/read_write.c:519
519   {
(gdb) bt
#0  vfs_write (file=0x8fbc7780, buf=0xf8f88 "/etc/init.d/rcS running
now ...\n", count=32, pos=0x8f9eff78)
    at fs/read_write.c:519
#1  0x8013a00c in SYSC_write (count=<optimized out>, buf=<optimized out>,
fd=<optimized out>) at fs/read_write.c:585
#2  SyS_write (fd=<optimized out>, buf=1019784, count=32) at
fs/read_write.c:577
#3  0x8000f980 in ?? ()
Backtrace stopped: frame did not save the PC
(gdb) l
514   }
515
516   EXPORT_SYMBOL(__kernel_write);
517
518   ssize_t vfs_write(struct file *file, const char __user *buf, size_t
count, loff_t *pos)
519   {
520         ssize_t ret;
521
522         if (!(file->f_mode & FMODE_WRITE))
523               return -EBADF;
(gdb) p file
$1 = (struct file *) 0x8fbc7780
(gdb) set print pretty
(gdb) p *file
$2 = {
  f_u = {
    fu_llist = {
      next = 0x0 <__vectors_start>
    },
```

```
[...]

  f_path = {
    mnt = 0x8f80f0d0,
    dentry = 0x8f61f170
  },
  f_inode = 0x8f638aa0,
  f_op = 0x805ce634 <console_fops>,

[...]

---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
...
Breakpoint 1, vfs_write (file=0x8fbf9780, buf=0xf9060 "8 4 1 7\n", count=8,
pos=0x8f9eff78) at fs/read_write.c:519
519   {
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x80139a00 in vfs_write at
fs/read_write.c:519
      breakpoint already hit 2 times
(gdb) disable 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x80139a00 in vfs_write at
fs/read_write.c:519
      breakpoint already hit 2 times
(gdb) c
Continuing.

...

<< Want to setup another breakpoint; so we 'break' into GDB with ^C >>
^C
Program received signal SIGINT, Interrupt.
cpu_v7_do_idle () at arch/arm/mm/proc-v7.S:74
74          ret   lr
(gdb) b kmem_cache_free
Breakpoint 3 at 0x80131e80: file mm/slub.c, line 2679.
(gdb) c
Continuing.

...


Breakpoint 3, kmem_cache_free (s=0x8f801b80, x=0x8f81e000) at
mm/slub.c:2679 << hit ! >>
2679    {
(gdb) bt
```

```
#0  kmem_cache_free (s=0x8f801b80, x=0x8f81e000) at mm/slub.c:2679
#1  0x80148840 in final_putname (name=0x8f81e000) at fs/namei.c:127
#2  0x8013892c in do_sys_open (dfd=-1895797304, filename=<optimized out>,
flags=-1895797312, mode=<optimized out>)
    at fs/open.c:1007
#3  0x801389d0 in SYSC_openat (mode=<optimized out>, flags=<optimized out>,
filename=<optimized out>,
    dfd=<optimized out>) at fs/open.c:1025
#4  SyS_openat (dfd=<optimized out>, filename=<optimized out>,
flags=<optimized out>, mode=<optimized out>)
    at fs/open.c:1019
#5  0x8000f980 in ?? ()
Backtrace stopped: frame did not save the PC
(gdb) l
2674            __slab_free(s, page, x, addr);
2675
2676 }
2677
2678 void kmem_cache_free(struct kmem_cache *s, void *x)
2679 {
2680       s = cache_from_obj(s, x);
2681       if (!s)
2682             return;
2683       slab_free(s, virt_to_head_page(x), x, _RET_IP_);
```

We can see in the above funciton (kmem_cache_free) that the first parameter is a pointer to struct kmem_cache. From the current ARM Procedure Call Standard (APCS) document (defined by ARM Ltd in this PDF document*), we know that the first four parameters will be in registers r0 through r3.*

Thus the first parameter should be in the **r0** register:

```
(gdb) info registers
r0             0x8f801b80   -1887429760
r1             0x8f81e000   -1887313920
r2             0x0    0

[...]

sp             0x8f043f38   0x8f043f38
lr             0x80148840   -2146138048
pc             0x80131e80   0x80131e80 <kmem_cache_free>
cpsr           0x60000013   1610612755
(gdb) p s
$1 = (struct kmem_cache *) 0x8f801b80       << indeed it is! >>
(gdb) p *s
$2 = {cpu_slab = 0x8084f0d0 <init_thread_union+4304>, flags = 0,
min_partial = 6, size = 4096, object_size = 4096,
  offset = 0, cpu_partial = 2, oo = {x = 196616}, max = {x = 196616}, min =
{x = 1}, allocflags = 16384,
```

```
  refcount = 2, ctor = 0x0 <__vectors_start>, inuse = 4096, align = 64,
reserved = 0,
  name = 0x8f8021c0 "kmalloc-4096", list = {next = 0x8f801c44, prev =
0x8f801b44}, kobj = {
    name = 0x8fab5900 ":t-0004096", entry = {next = 0x8f801c50, prev =
0x8f801b50}, parent = 0x8f9d9e28,
...
(gdb) set print pretty
...


<< On the ARM target >>
/bin/sh: can't access tty; job control turned off
ARM / $
ARM / $ halt
ARM / $ [    7.653539] EXT4-fs (mmcblk0): re-mounted. Opts: (null)
The system is going down NOW!
Sent SIGTERM to all processes
Sent SIGKILL to all processes
Requesting system halt
[    9.706334] Flash device refused suspend due to active operation (state
20)
[    9.707181] Flash device refused suspend due to active operation (state
20)
[    9.708762] reboot: System halted
QEMU: Terminated
$

<< On the x86 host >>
Remote connection closed
(gdb) q
$
```

Debugging a Kernel Module with KGDB

In order to debug a kernel module within KGDB, the only difference from debugging the “regular” in-line kernel code, is that the kernel module's ELF sections information has to be provided to gdb, so that gdb can “see” it.

We can achieve this quite easily:

1. (Cross) Compile the kernel module with debug symbols:
   In the Makefile, keep the line

   `EXTRA_CFLAGS += -DDEBUG -g -ggdb`
   or (modern):
   `ccflags-y += -DDEBUG -g -ggdb`

   (Caution: above, do NOT add -O0 ; the kernel and modules MUST be compiled with optimization on, usually -O2; this is unlike userspace...)

2. Copy the kernel module and the *gdbline.sh* shell script (seen earlier in this module) onto the target

3. With the KGDB session active:
   1. Load the kernel module into RAM
      `insmod <mykm.ko>`
   2. Query it's ELF sections using
      `/sys/module/<module-name>/sections/.<section>` either directly (or, easier, via the *gdbline.sh* shell script seen earlier).

An Example Session:
Using a custom network driver kernel module and the gdbline.sh script

On the target:

```
ARM /myprj/veth_dyn_ops $ ../gdbline.sh  veth_dyn_ops ./veth_dyn_ops.ko
add-symbol-file ./veth_dyn_ops.ko 0xbf000000 -s .alt.smp.init 0xbf000f34 -s
.bss 0xbf001a10 -s .data 0xbf001758 -s .exit.text 0xbf000dac
[ ... ] -s __mcount_loc 0xbf00165c
ARM /myprj/veth_dyn_ops $
```

On the host gdb (in this case, it's the x86-to-ARM cross-compiler gdb):

<< cd to the source folder of the kernel module you're going to debug >>

```
(gdb) cd <...>/3_veth_dyn_ops
Working directory <...>/3_veth_dyn_ops
 (canonically <…/…>/3_veth_dyn_ops).
```

```
(gdb)
```

<< Copy-paste the output of the gdbline.sh shell script >>

```
(gdb) add-symbol-file ./veth_dyn_ops.ko 0xbf000000 -s .alt.smp.init
0xbf000f34 -s .bss 0xbf001a10 -s .data 0xbf001758 [ … ] -s __mcount_loc
0xbf00165c
add symbol table from file "./veth_dyn_ops.ko" at
     .text_addr = 0xbf000000
     .alt.smp.init_addr = 0xbf000f34
     .bss_addr = 0xbf001a10
     .data_addr = 0xbf001758
...
     __mcount_loc_addr = 0xbf00165c
(y or n) y
Reading symbols from <path/to/>/veth_dyn_ops.ko...warning: section .strtab
not found in <path/to/>/veth_dyn_ops.ko
warning: section .symtab not found in <path/to/>/veth_dyn_ops.ko
done.
(gdb) b vnet_ <<tab-tab>>
vnet_exit        vnet_init        vnet_open        vnet_remove
vnet_stop
vnet_get_stats   vnet_netdev_ops  vnet_probe       vnet_start_xmit
vnet_tx_timeout
(gdb) b vnet_start_xmit
Breakpoint 1 at 0xbf0003fc: file <path/to/>/veth_netdrv.c, line 73.
(gdb) c
Continuing.
```

<< Have some data sent to the network interface in question, so that the kernel invokes the network driver's 'start_xmit' method.. ... >>

```
Breakpoint 1, vnet_start_xmit (skb=0xc78819c0, dev=0xc61de800)
    at <path/to/>/veth_netdrv.c:73
73         if (!skb) { // paranoia!
(gdb) bt
#0  vnet_start_xmit (skb=0xc78819c0, dev=0xc61de800)
    at <path/to/>/veth_netdrv.c:73
#1  0xc036cd78 in dev_hard_start_xmit (skb=0xc78819c0, dev=0xc61de800,
txq=0xc11e1120) at net/core/dev.c:2222
#2  0xc0387930 in sch_direct_xmit (skb=0xc78819c0, q=0xc625e200,
dev=0xc61de800, txq=0xc11e1120, root_lock=0xc625e25c) at
net/sched/sch_generic.c:125
#3  0xc036d224 in __dev_xmit_skb (skb=0xc78819c0) at net/core/dev.c:2427
#4  dev_queue_xmit (skb=0xc78819c0) at net/core/dev.c:2502
#5  0xc0379880 in neigh_resolve_output (neigh=0xc7881900, skb=0xc78819c0)
at net/core/neighbour.c:1274
#6  0xc039c2d8 in neigh_output (skb=0xc78819c0) at
include/net/neighbour.h:352
#7  ip_finish_output2 (skb=0xc78819c0) at net/ipv4/ip_output.c:211
#8  ip_finish_output (skb=0xc78819c0) at net/ipv4/ip_output.c:244
```

```
#9  0xc039c5e8 in NF_HOOK_COND (skb=0xc78819c0) at
include/linux/netfilter.h:233
#10 ip_output (skb=0xc78819c0) at net/ipv4/ip_output.c:317
#11 0xc039b59c in dst_output (skb=0xc78819c0) at include/net/dst.h:430
#12 ip_local_out (skb=0xc78819c0) at net/ipv4/ip_output.c:111
#13 0xc039b5bc in ip_send_skb (skb=0xc78819c0) at net/ipv4/ip_output.c:1371
#14 0xc03be734 in udp_send_skb (skb=0xc78819c0, fl4=0xc11cfd5c) at
net/ipv4/udp.c:753
#15 0xc03c0630 in udp_sendmsg (iocb=<value optimized out>, sk=0xc11d4220,
msg=0xc11cff54, len=16) at net/ipv4/udp.c:966
#16 0xc03c92c4 in inet_sendmsg (iocb=0xc11cfde8, sock=<value optimized
out>, msg=0x4000, size=268503562) at net/ipv4/af_inet.c:744
#17 0xc0357724 in __sock_sendmsg_nosec (sock=0xc7404d80, msg=0xc11cff54,
size=16) at net/socket.c:557
#18 __sock_sendmsg (sock=0xc7404d80, msg=0xc11cff54, size=16) at
net/socket.c:565
#19 sock_sendmsg (sock=0xc7404d80, msg=0xc11cff54, size=16) at
net/socket.c:576
#20 0xc0357f18 in sys_sendto (fd=<value optimized out>, buff=0xc11cfed4,
len=16, flags=0, addr=0xbe845cdc, addr_len=16) at net/socket.c:1708
#21 0xc000ed80 in ?? ()
Cannot access memory at address 0x0
#22 0xc000ed80 in ?? ()
Cannot access memory at address 0x0
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
(gdb) l
68         struct udphdr *udph;
69         pstVnetIntfCtx pstCtx = netdev_priv (dev);
70         //struct timespec ts1, ts2, tdiff;
71         struct timeval ts1, ts2, tdiff;
72
73         if (!skb) { // paranoia!
74              printk (KERN_ALERT "%s:%s: skb NULL!\n", DRVNAME,
__func__);
75              return -EAGAIN;
76         }
77   //    SKB_PEEK(skb);
...
...
97         ip = ip_hdr(skb);
(gdb) p ip
$3 = (struct iphdr *) 0xc11c3e10
(gdb) set print pretty
(gdb) p/x *ip
$5 = {
  ihl = 0x5,
  version = 0x4,
  tos = 0x0,
  tot_len = 0x2c00,
  id = 0x0,
  frag_off = 0x40,
  ttl = 0x40,
```

```
  protocol = 0x11,
  check = 0x9924,
  saddr = 0x5010a0a,
  daddr = 0x10010a0a
}
(gdb)
```

<< Change an existing variable's value! >>

```
(gdb) help set variable
Evaluate expression EXP and assign result to variable VAR, using assignment
syntax appropriate for the current language (VAR = EXP or VAR := EXP for
example).  VAR may be a debugger "convenience" variable (names starting
with $), a register (a few standard names starting with $), or an actual
variable in the program being debugged.  EXP is any valid expression.
This may usually be abbreviated to simply "set".

(gdb) p dev
$13 = (struct net_device *) 0xcfa5f800
(gdb) p dev->promiscuity
$11 = 0x0
(gdb) set variable dev->promiscuity = 1          ← here!
(gdb) p dev->promiscuity
$12 = 0x1
(gdb)
```

Note:

From (around) the 3.20 Linux kernel, (and GDB ver >= 7.2 required) we have an interesting Kernel configurable:

lib/Kconfig.debug

```
...
config GDB_SCRIPTS
    bool "Provide GDB scripts for kernel debugging"
    depends on DEBUG_INFO
    help
      This creates the required links to GDB helper scripts in the
      build directory. If you load vmlinux into gdb, the helper
      scripts will be automatically imported by gdb as well, and
      additional functions are available to analyze a Linux kernel
      instance. See Documentation/gdb-kernel-debugging.txt for
further
```

```
      details.
```

See *Documentation/gdb-kernel-debugging.txt* .

```
...

List of commands and functions
------------------------------

The number of commands and convenience functions may evolve over the time,
this is just a snapshot of the initial version:

 (gdb) apropos lx
 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
 lx-dmesg -- Print Linux kernel log buffer
 lx-lsmod -- List currently loaded modules
 lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded
modules

Detailed help can be obtained via "help <command-name>" for commands and
"help
function <function-name>" for convenience functions.
```

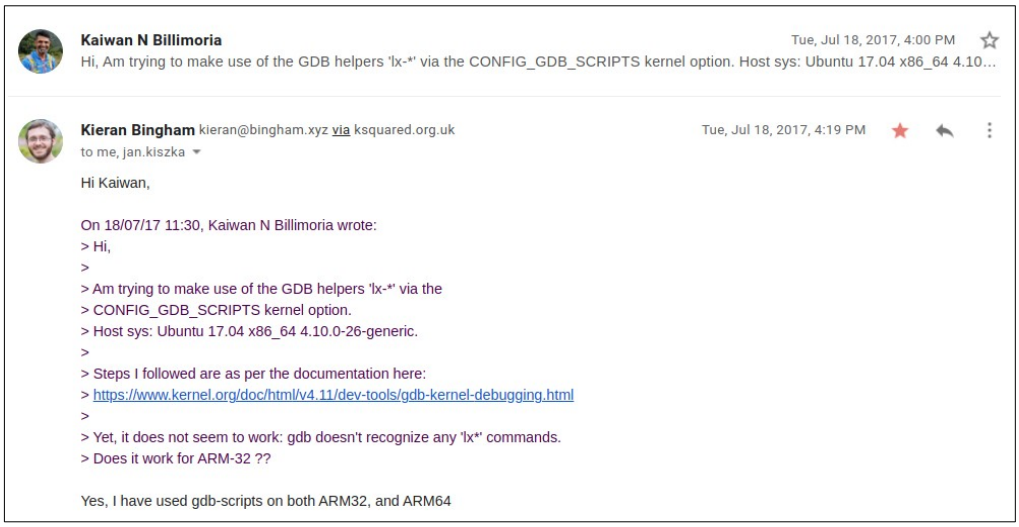
SIDEBAR ::

The MAINTAINERS file

Very useful!
I once had an issue with getting the Python-based GDB scripts running on an ARM-based system. Looked up the MAINTAINERS file, grepping for help; I soon found it!

```
linux-5.10.3 $ grep -i -A5 "gdb " MAINTAINERS
GDB KERNEL DEBUGGING HELPER SCRIPTS
M:      Jan Kiszka <jan.kiszka@siemens.com>
M:      Kieran Bingham <kbingham@kernel.org>
S:      Supported
F:      scripts/gdb/

--
KGDB / KDB /debug_core
M:      Jason Wessel <jason.wessel@windriver.com>
M:      Daniel Thompson <daniel.thompson@linaro.org>
R:      Douglas Anderson <dianders@chromium.org>
L:      kgdb-bugreport@lists.sourceforge.net
S:      Maintained
linux-5.10.3 $
```

Wrote to the maintainers, and had a good reply within an hour!!



Kaiwan N Billimoria Tue, Jul 18, 2017, 4:00 PM
Hi, Am trying to make use of the GDB helpers 'lx-*' via the CONFIG_GDB_SCRIPTS kernel option. Host sys: Ubuntu 17.04 x86_64 4.10...

Kieran Bingham kieran@bingham.xyz via ksquared.org.uk Tue, Jul 18, 2017, 4:19 PM
to me, jan.kiszka

Hi Kaiwan,

On 18/07/17 11:30, Kaiwan N Billimoria wrote:
> Hi,
>
> Am trying to make use of the GDB helpers 'lx-*' via the
> CONFIG_GDB_SCRIPTS kernel option.
> Host sys: Ubuntu 17.04 x86_64 4.10.0-26-generic.
>
> Steps I followed are as per the documentation here:
> https://www.kernel.org/doc/html/v4.11/dev-tools/gdb-kernel-debugging.html
>
> Yet, it does not seem to work: gdb doesn't recognize any 'lx*' commands.
> Does it work for ARM-32 ??

Yes, I have used gdb-scripts on both ARM32, and ARM64

...

...

Jan Kiszka Jul 18, 2017, 5:44 PM
Ask CodeSourcery folks, can't tell. But the Linaro toolchain supports this mode.

Test run on an x86_64 VM:

```
tty0 $ ls -lh vmlinux
-rwxrwxr-x 1 seawolf seawolf 439M Jul 16 11:18 vmlinux*  << vmlinux with debugsym >>
tty0 $ ls -l vmlinux*
```

```
-rwxrwxr-x 1 seawolf seawolf 459318832 Jul 16 11:18 vmlinux*
lrwxrwxrwx 1 seawolf seawolf        58 Jul 16 11:16 vmlinux-gdb.py ->
<...>/linux-4.4.21/scripts/gdb/vmlinux-gdb.py  << notice the slink ! >>
-rw-rw-r-- 1 seawolf seawolf 542215936 Jul 16 11:16 vmlinux.o

tty0 $ cat ~/.gdbinit   << setup your ~/.gdbinit to auto-load the python
scripts >>
# ~/.gdbinit

# For GDB helper scripts pkg w/ modern kernels (GDB_SCRIPTS)
add-auto-load-safe-path <...>linux-4.4.21/scripts/gdb/vmlinux-gdb.py
tty0 $

tty0 $ gdb ./vmlinux
GNU gdb (Ubuntu 7.11.90.20161005-0ubuntu2) 7.11.90.20161005-git
[...]
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vmlinux...done.
(gdb) info auto-load
gdb-scripts:  No auto-load scripts.
libthread-db:  No auto-loaded libthread-db.
local-gdbinit:  Local .gdbinit file was not found.
python-scripts:
Loaded  Script
Yes     /home/seawolf/0tmp/linux-4.4.21/vmlinux-gdb.py
(gdb) apropos lx
function lx_current -- Return current task  << the lx-* gdb goodies are now
available ! >>
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
lx-dmesg -- Print Linux kernel log buffer
lx-list-check -- Verify a list consistency
lx-lsmod -- List currently loaded modules
lx-ps -- Dump Linux tasks
lx-symbols -- (Re-)load symbols of Linux kernel and currently loaded
modules
(gdb)
```

It may not always be the case that GDB supports python scripts. For eg., on a cross-compile GDB that I use at times for embedded Linux work, the support was not present:

```
$ arm-none-linux-gnueabi-gdb ./vmlinux
GNU gdb (Sourcery CodeBench Lite 2014.05-29) 7.7.50.20140217-cvs << GDB ver
>= 7.2 yet ... >>
[...]
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vmlinux...done.
(gdb) python import sys            << check for python scripting support >>
Python scripting is not supported in this copy of GDB.
```

(gdb)

(Tip: Looks to be an issue with the Code Sourcery (Mentor Graphics) toolchain (this ver only??); Linaro toolchains support python scripting).

[OPTIONAL/FYI]

KDB

<<
Refer "Linux Debugging and Performance Tuning" by Steve Best, Ch 13 "Kernel-Level Debuggers (kgdb and kdb)" section "kdb" page 348 onwards.
>>

Additional Notes for kernel-level debuggers topic: kdb
Author: kaiwan.

Earlier, KDB had to be patched into a Linux kernel via an arch-dependent and an arch-independent patch.

Recent Linux kernels:

The procedure below to apply the two kdb patches is now outdated. That's good news: **kdb is now in mainline (from ver 2.6.26) and is integrated as a "frontend" into the KGDB system.**

What is different about the old KDB vs the merged KDB?

In April 2009 KDB 4.4 had significant chunks of the code base removed and hooked it up to the same debug core and polled I/O drivers used by kgdb.

At a really high level, the only difference is how you configure and connect. The kdb disassembler was the main casualty of the merger. Architecturally the merged kdb is 99% platform independent and actually gained new features via the debug core API.

…

The kdb command shell differences

- There are fewer commands in the merged kdb & kgdb.
    - The kdump / kexec analysis modules tools were removed in the process of merging the code bases.

- The status command emits a bit less information in that you no longer get the exact output you would from cat /proc/meminfo

- The bt command does uses the kernel's backtracer and not a disassembly engine which shows you function arguments
- In the merged kdb & kgdb, presently there is no disassembler (known as the "id" command)
- In the merged kdb & kgdb, when using a non-vga style connection you can transition into kgdb mode and attach gdb, by using the kgdb command.
- The ability to reference elements of a structures was removed. This may get implemented a different way in the future.

Architecture Support:

- KDB 4.4 supported only ia64, x86_64 and i386
- The merged kdb & kgdb supports arm, blackfin, mips, x86_64, i386, sh, powerpc, and sparc.

Kernel Configuration

```
$ make ARCH=<arch> menuconfig
```

…

Under 'Kernel Hacking / KGDB: kernel debugger' :

i.e. select the option labelled:

KGDB_KDB : include kdb frontend for kgdb

, as shown above.

The relevant configuration options are:

```
# grep -i kdb .config
CONFIG_KGDB_KDB=y
CONFIG_KDB_KEYBOARD=y
CONFIG_KDB_CONTINUE_CATASTROPHIC=0
#
```

Save & build. When booted, the kernel now supports debugging via both the KGDB and KDB interfaces.

How do we enter kdb?
There are a few ways (several described below); the “new” technique is conveniently captured by the shell script below:

```
$ cat kdb.sh
#!/bin/sh
# kdb.sh
# Invoke KDB - kernel debugger
# Of course, we have to have KGDB/KDB compiled into the kernel
# Relevant k config options:
#  CONFIG_KGDB_KDB=y
#  CONFIG_KDB_KEYBOARD=y
```

```
# Enable console for KDB use
# (use ttyO2 for the pandaboard; ttyAMA0 for QEMU/ARM-vexpress)
CONSOLEDEV=ttyAMA0

echo ${CONSOLEDEV} > /sys/module/kgdboc/parameters/kgdboc || {
 echo "Failed."
 exit 1
}

# Invoke KDB (via Magic-SysRq key 'g' (debug))
echo "+++ Invoking KDB now...(type 'go' @ kdb prompt to exit kdb)"
echo g > /proc/sysrq-trigger
$
```

Run this script on the target and you'll end up at the kdb prompt!

A Quick KDB Session using ARM/Linux on QEMU

```
ARM / # echo g > /proc/sysrq-trigger
SysRq : HELP : loglevel(0-9) reBoot Crash terminate-all-tasks(E) memory-
full-oom-kill(F) kill-all-tasks(I) thaw-filesystems(J) saK show-backtrace-
all-active-cpus(L) show-memory-usage(M) nice-all-RT-tasks(N) powerOff show-
registers(P) show-all-timers(Q) unRaw Sync show-task-states(T) Unmount
show-blocked-tasks(W) dump-ftrace-buffer(Z)
ARM / #
```

Doesn't work ?? Try this:

```
ARM / # echo ttyAMA0 > /sys/module/kgdboc/parameters/kgdboc
kgdb: Registered I/O driver kgdboc.
ARM / #
ARM / # echo g > /proc/sysrq-trigger        << now it works! >>
SysRq : DEBUG

Entering kdb (current=0xcf8a54a0, pid 489) on processor 0 due to Keyboard
Entry
[0]kdb> ?
Command         Usage                Description
-----------------------------------------------------------
md              <vaddr>             Display Memory Contents, also mdWcN,
e.g. md8c1
mdr             <vaddr> <bytes>     Display Raw Memory
mdp             <paddr> <bytes>     Display Physical Memory
mds             <vaddr>             Display Memory Symbolically
mm              <vaddr> <contents>  Modify Memory Contents
go              [<vaddr>]           Continue Execution
rd                                  Display Registers
rm              <reg> <contents>    Modify Registers
ef              <vaddr>             Display exception frame
```

```
bt              [<vaddr>]           Stack traceback
btp             <pid>               Display stack for process <pid>
bta             [D|R|S|T|C|Z|E|U|I|M|A]
                                    Backtrace all processes matching state
flag
btc                                 Backtrace current process on each cpu
btt             <vaddr>             Backtrace process given its struct task
address
env                                 Show environment variables
set                                 Set environment variables
help                                Display Help Message
?                                   Display Help Message
cpu             <cpunum>            Switch to new cpu
kgdb                                Enter kgdb mode
ps              [<flags>|A]         Display active task list
pid             <pidnum>            Switch to another task
reboot                              Reboot the machine immediately
lsmod                               List loaded kernel modules
sr              <key>               Magic SysRq key
dmesg           [lines]             Display syslog buffer
defcmd          name "usage" "help" Define a set of commands, down to
endefcmd
kill            <-signal> <pid>     Send a signal to a process
summary                             Summarize the system
per_cpu         <sym> [<bytes>] [<cpu>]
                                    Display per_cpu variables
grephelp                            Display help on | grep
bp              [<vaddr>]           Set/Display breakpoints
bl              [<vaddr>]           Display breakpoints
bc              <bpnum>             Clear Breakpoint
be              <bpnum>             Enable Breakpoint
bd              <bpnum>             Disable Breakpoint
ss                                  Single Step
dumpcommon                          Common kdb debugging
dumpall                             First line debugging
dumpcpu                             Same as dumpall but only tasks on cpus
ftdump          [skip_#lines] [cpu] Dump ftrace log
[0]kdb> summary           << kdb: summ: Summarize the system >>
sysname    Linux
release    3.18.22-kgdb
version    #2 SMP Thu Jan 19 12:46:36 IST 2017
machine    armv7l
nodename   (none)
domainname (none)
ccversion  CCVERSION
date       2017-01-19 10:25:03 tz_minuteswest 0
uptime     00:04
load avg   0.00 0.00 0.00

MemTotal:         245252 kB
MemFree:          233244 kB
Buffers:             156 kB
```

```
[0]kdb> ps           << kdb: ps: Display active task list >>
30 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Task Addr       Pid   Parent [*] cpu State Thread     Command
0x8f911e00      609        1  1    0   R  0x8f9120e0 *sh

0x8f880000        1        0  0    0   S  0x8f8802e0  init
0x8f911e00      609        1  1    0   R  0x8f9120e0 *sh

[0]kdb> md 0x8f880000    << kdb: md: Display Memory Contents, also mdWcN,
e.g. md8c1 >>
0x8f880000 00000001 8f844000 00000002 00400100   .....@........@.
0x8f880010 00000000 00000000 00000000 8fbe8a00   ................
0x8f880020 0000021c 00000000 00000000 00000000   ................
0x8f880030 00000078 00000078 00000078 00000000   x...x...x.......
0x8f880040 8059adec 00000000 00000400 00400000   ..Y...........@.
0x8f880050 00000001 00000000 00000000 8f88005c   ............\...
0x8f880060 8f88005c 00000000 ff197121 00000000   \.......!q......
0x8f880070 32e56250 00000000 7e74b0ec 00000000   Pb.2......t~....

[0]kdb> rd         << kdb: rd: Display Registers >>
r0: 00000067  r1: 00000000  r2: 90800000  r3: 90800730  r4: 80e36b40
r5: 80e36b44  r6: 80856448  r7: 8f20a010  r8: 00000000  r9: 00000008
r10: 00000000  fp: 8f20bebc  ip: 8f20bea0  sp: 8f20bea0  lr: 800a62b4
pc: 800a62b4  f0: ??  f1: ??  f2: ??  f3: ??  f4: ??  f5: ??  f6: ??
f7: ??
fps: 00000000  cpsr: a0000013

[0]kdb> btc         << kdb: btc: Backtrace current process on each cpu >>
btc: cpu status: Currently on cpu 0
Available cpus: 0
Stack traceback for pid 609
0x8f911e00      609        1  1    0   R  0x8f9120e0 *sh
[<80017cdc>] (unwind_backtrace) from [<80013c5c>] (show_stack+0x20/0x24)
[<80013c5c>] (show_stack) from [<800b00bc>] (kdb_show_stack+0x54/0x68)
[<800b00bc>] (kdb_show_stack) from [<800b016c>] (kdb_bt1.isra.0+0x9c/0xe8)
[<800b016c>] (kdb_bt1.isra.0) from [<800b0398>] (kdb_bt+0x1e0/0x454)
[<800b0398>] (kdb_bt) from [<800ad99c>] (kdb_parse+0x2cc/0x6c4)
[<800ad99c>] (kdb_parse) from [<800b0450>] (kdb_bt+0x298/0x454)
[<800b0450>] (kdb_bt) from [<800ad99c>] (kdb_parse+0x2cc/0x6c4)
[<800ad99c>] (kdb_parse) from [<800ae57c>] (kdb_main_loop+0x530/0x790)
[<800ae57c>] (kdb_main_loop) from [<800b1174>] (kdb_stub+0x230/0x488)
[<800b1174>] (kdb_stub) from [<800a6e8c>] (kgdb_cpu_enter+0x43c/0x730)
[<800a6e8c>] (kgdb_cpu_enter) from [<800a7464>]
(kgdb_handle_exception+0x194/0x1e0)
[<800a7464>] (kgdb_handle_exception) from [<80017038>]
(kgdb_compiled_brk_fn+0x38/0x40)
[<80017038>] (kgdb_compiled_brk_fn) from [<800083fc>]
(do_undefinstr+0x17c/0x214)
[<800083fc>] (do_undefinstr) from [<80594c10>] (__und_svc_finish+0x0/0x30)
Exception stack(0x8f20be18 to 0x8f20be60)
```

```
be00:                                                     00000067
00000000
be20: 90800000 90800730 80e36b40 80e36b44 80856448 8f20a010 00000000
00000008
be40: 00000000 8f20bebc 8f20bea0 8f20bea0 800a62b4 800a62b4 a0000013
ffffffff
[<80594c10>] (__und_svc_finish) from [<800a62b4>]
(kgdb_breakpoint+0x58/0x94)
[<800a62b4>] (kgdb_breakpoint) from [<800a6368>]
(sysrq_handle_dbg+0x4c/0x70)
[<800a6368>] (sysrq_handle_dbg) from [<80349734>]
(__handle_sysrq+0x100/0x1f0)
[<80349734>] (__handle_sysrq) from [<80349c98>]
(write_sysrq_trigger+0x58/0x68)
[<80349c98>] (write_sysrq_trigger) from [<8019083c>]
(proc_reg_write+0x6c/0x94)
[<8019083c>] (proc_reg_write) from [<80139ab4>] (vfs_write+0xb4/0x1c0)
[<80139ab4>] (vfs_write) from [<8013a00c>] (SyS_write+0x4c/0xa0)
[<8013a00c>] (SyS_write) from [<8000f980>] (ret_fast_syscall+0x0/0x50)

[0]kdb> cpu     << kdb: cpu: <cpunum>            Switch to new cpu >>
Currently on cpu 0
Available cpus: 0

[0]kdb> lsmod
Module                  Size  modstruct     Used by
```

```
[0]kdb> dumpall    << kdb: dumpall: First line debugging (a kind of wrapper cmd) >>
[dumpall]kdb>   pid R

KDB current process is sh(pid=609)
[dumpall]kdb>   -dumpcommon         << commands shown like this => internally executed by
                                                the wrapper cmd 'dumpall' >>

[dumpcommon]kdb>   set BTAPROMPT 0

[dumpcommon]kdb>   set LINES 10000

[dumpcommon]kdb>   -summary

sysname    Linux
release    3.18.22-kgdb
version    #2 SMP Thu Jan 19 12:46:36 IST 2017
machine    armv7l
nodename   (none)
domainname (none)
ccversion  CCVERSION
date       2017-01-19 10:25:03 tz_minuteswest 0
uptime     00:12
load avg   0.00 0.00 0.00

MemTotal:         245252 kB
MemFree:          233244 kB
Buffers:             156 kB
[dumpcommon]kdb>   -cpu

Currently on cpu 0
Available cpus: 0
[dumpcommon]kdb>   -ps

30 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Task Addr       Pid   Parent [*] cpu State Thread     Command
0x8f911e00      609        1  1    0   R  0x8f9120e0 *sh

0x8f880000        1        0  0    0   S  0x8f8802e0  init
0x8f911e00      609        1  1    0   R  0x8f9120e0 *sh
[dumpcommon]kdb>   -dmesg 600

buffer only contains 164 lines, first 164 lines printed
<6>[    0.000000] Booting Linux on physical CPU 0x0
<6>[    0.000000] Initializing cgroup subsys cpuset
<5>[    0.000000] Linux version 3.18.22-kgdb (root@kaiwan-T460) (gcc
version 4.8.3 20140320 (prerelease) (Sourcery CodeBench Lite 2014.05-29) )
#2 SMP Thu Jan 19 12:46:36 IST 2017
<6>[    0.000000] CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7),
cr=10c5387d
```

```
<6>[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing
instruction cache
<4>[    0.000000] Machine: ARM-Versatile Express

--snip--

<6>[    2.148985] VFS: Mounted root (ext4 filesystem) readonly on device
179:0.
<6>[    2.161095] Freeing unused kernel memory: 436K (807e1000 - 8084e000)
<5>[    2.434529] random: nonblocking pool is initialized
<6>[    3.174955] EXT4-fs (mmcblk0): re-mounted. Opts: data=ordered
<6>[    3.241284] smsc911x smsc911x eth0: SMSC911x/921x identified at
0x90a40000, IRQ: 47

<6>[   71.141098] kgdb: Registered I/O driver kgdboc.
<6>[   83.832502] SysRq : DEBUG
[dumpcommon]kdb>   -bt

Stack traceback for pid 609
0x8f911e00      609        1  1    0   R  0x8f9120e0 *sh
[<80017cdc>] (unwind_backtrace) from [<80013c5c>] (show_stack+0x20/0x24)
[<80013c5c>] (show_stack) from [<800b00bc>] (kdb_show_stack+0x54/0x68)
[<800b00bc>] (kdb_show_stack) from [<800b016c>] (kdb_bt1.isra.0+0x9c/0xe8)
[<800b016c>] (kdb_bt1.isra.0) from [<800b0508>] (kdb_bt+0x350/0x454)
[<800b0508>] (kdb_bt) from [<800ad99c>] (kdb_parse+0x2cc/0x6c4)
[<800ad99c>] (kdb_parse) from [<800ade3c>] (kdb_exec_defcmd+0xa8/0xf0)
[<800ade3c>] (kdb_exec_defcmd) from [<800ad99c>] (kdb_parse+0x2cc/0x6c4)
[<800ad99c>] (kdb_parse) from [<800ade3c>] (kdb_exec_defcmd+0xa8/0xf0)
[<800ade3c>] (kdb_exec_defcmd) from [<800ad99c>] (kdb_parse+0x2cc/0x6c4)
[<800ad99c>] (kdb_parse) from [<800ae57c>] (kdb_main_loop+0x530/0x790)
[<800ae57c>] (kdb_main_loop) from [<800b1174>] (kdb_stub+0x230/0x488)
[<800b1174>] (kdb_stub) from [<800a6e8c>] (kgdb_cpu_enter+0x43c/0x730)
[<800a6e8c>] (kgdb_cpu_enter) from [<800a7464>]
(kgdb_handle_exception+0x194/0x1e0)
[<800a7464>] (kgdb_handle_exception) from [<80017038>]
(kgdb_compiled_brk_fn+0x38/0x40)
[<80017038>] (kgdb_compiled_brk_fn) from [<800083fc>]
(do_undefinstr+0x17c/0x214)
[<800083fc>] (do_undefinstr) from [<80594c10>] (__und_svc_finish+0x0/0x30)
Exception stack(0x8f20be18 to 0x8f20be60)
be00:                                                     00000067
00000000
be20: 90800000 90800730 80e36b40 80e36b44 80856448 8f20a010 00000000
00000008
be40: 00000000 8f20bebc 8f20bea0 8f20bea0 800a62b4 800a62b4 a0000013
ffffffff
[<80594c10>] (__und_svc_finish) from [<800a62b4>]
(kgdb_breakpoint+0x58/0x94)
[<800a62b4>] (kgdb_breakpoint) from [<800a6368>]
(sysrq_handle_dbg+0x4c/0x70)
[<800a6368>] (sysrq_handle_dbg) from [<80349734>]
(__handle_sysrq+0x100/0x1f0)
```

```
[<80349734>] (__handle_sysrq) from [<80349c98>]
(write_sysrq_trigger+0x58/0x68)
[<80349c98>] (write_sysrq_trigger) from [<8019083c>]
(proc_reg_write+0x6c/0x94)
[<8019083c>] (proc_reg_write) from [<80139ab4>] (vfs_write+0xb4/0x1c0)
[<80139ab4>] (vfs_write) from [<8013a00c>] (SyS_write+0x4c/0xa0)
[<8013a00c>] (SyS_write) from [<8000f980>] (ret_fast_syscall+0x0/0x50)
[dumpall]kdb>   -bta

30 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Stack traceback for pid 609
0x8f911e00      609        1  1    0   R  0x8f9120e0 *sh
[<80017cdc>] (unwind_backtrace) from [<80013c5c>] (show_stack+0x20/0x24)
[<80013c5c>] (show_stack) from [<800b00bc>] (kdb_show_stack+0x54/0x68)
[<800b00bc>] (kdb_show_stack) from [<800b016c>] (kdb_bt1.isra.0+0x9c/0xe8)
[<800b016c>] (kdb_bt1.isra.0) from [<800b0260>] (kdb_bt+0xa8/0x454)
[<800b0260>] (kdb_bt) from [<800ad99c>] (kdb_parse+0x2cc/0x6c4)
[<800ad99c>] (kdb_parse) from [<800ade3c>] (kdb_exec_defcmd+0xa8/0xf0)
[<800ade3c>] (kdb_exec_defcmd) from [<800ad99c>] (kdb_parse+0x2cc/0x6c4)
[<800ad99c>] (kdb_parse) from [<800ae57c>] (kdb_main_loop+0x530/0x790)
[<800ae57c>] (kdb_main_loop) from [<800b1174>] (kdb_stub+0x230/0x488)
[<800b1174>] (kdb_stub) from [<800a6e8c>] (kgdb_cpu_enter+0x43c/0x730)
[<800a6e8c>] (kgdb_cpu_enter) from [<800a7464>]
(kgdb_handle_exception+0x194/0x1e0)
[<800a7464>] (kgdb_handle_exception) from [<80017038>]
(kgdb_compiled_brk_fn+0x38/0x40)
[<80017038>] (kgdb_compiled_brk_fn) from [<800083fc>]
(do_undefinstr+0x17c/0x214)
[<800083fc>] (do_undefinstr) from [<80594c10>] (__und_svc_finish+0x0/0x30)
Exception stack(0x8f20be18 to 0x8f20be60)
be00:                                                     00000067
00000000
be20: 90800000 90800730 80e36b40 80e36b44 80856448 8f20a010 00000000
00000008
be40: 00000000 8f20bebc 8f20bea0 8f20bea0 800a62b4 800a62b4 a0000013
ffffffff
[<80594c10>] (__und_svc_finish) from [<800a62b4>]
(kgdb_breakpoint+0x58/0x94)
[<800a62b4>] (kgdb_breakpoint) from [<800a6368>]
(sysrq_handle_dbg+0x4c/0x70)
[<800a6368>] (sysrq_handle_dbg) from [<80349734>]
(__handle_sysrq+0x100/0x1f0)
[<80349734>] (__handle_sysrq) from [<80349c98>]
(write_sysrq_trigger+0x58/0x68)
[<80349c98>] (write_sysrq_trigger) from [<8019083c>]
(proc_reg_write+0x6c/0x94)
[<8019083c>] (proc_reg_write) from [<80139ab4>] (vfs_write+0xb4/0x1c0)
[<80139ab4>] (vfs_write) from [<8013a00c>] (SyS_write+0x4c/0xa0)
[<8013a00c>] (SyS_write) from [<8000f980>] (ret_fast_syscall+0x0/0x50)
Stack traceback for pid 1
0x8f880000        1        0  0    0   S  0x8f8802e0  init
```

```
[<8058e5b0>] (__schedule) from [<8058e9c4>] (schedule+0x40/0x8c)
[<8058e9c4>] (schedule) from [<8002b340>] (do_wait+0x220/0x28c)
[<8002b340>] (do_wait) from [<8002b7b4>] (SyS_wait4+0x74/0xd8)
[<8002b7b4>] (SyS_wait4) from [<8000f980>] (ret_fast_syscall+0x0/0x50)
[0]kdb>
[0]kdb> go   << kdb: go: [<vaddr>]          Continue Execution >>
ARM / #
```

When kdb is enabled, a kernel Oops causes the debugger to be invoked, and the keyboard LEDs blink. Once the kdb prompt is displayed, you can enter kdb commands.

If the system will be run in graphical mode with kdb enabled, it is recommend that kdb be set up to use a serial console so that the kdb prompt can be seen.

kdb Activation

kdb can be activated by configuring it at kernel build time. If kdb off by default (CONFIG_KDB_OFF) was not selected during kernel configuration, kdb is active by default.

[DEPRECATED : below]
The runtime options are as follows:

- Activate it by passing the `kdb=on` flag to the kernel during boot.

  Turn it on or off through the /proc file system entry

- To enable kdb through the /proc, use this command:

  ```
  # echo "1" > /proc/sys/kernel/kdb
  ```

  To disable kdb through the /proc, use this command:

  ```
  # echo "0" > /proc/sys/kernel/kdb
  ```

If kdb is needed during boot, specify the `kdb=early` flag to the kernel during boot. This allows kdb to be activated during the boot process.

kdb is invoked in the following ways:
- Pressing the Pause key on the keyboard manually invokes kdb.
- Whenever there is an Oops or kernel panic.
- If kdb is set up through a serial port between two machines, pressing Ctrl-A invokes kdb from the serial console. Several programs can be used to communicate with kdb through a serial port, such as minicom and kermit.
  Note-The key sequence Ctrl-A has been changed in version 4.4 of kdb. It is now Esc-KDB.

- Also see: Entering kdb from qemu

<<
FYI, an interesting blog post (from your author):
"A KDB / KGDB session on the popular Raspberry Pi embedded Linux board"

>>

Another Sample Session

We'll use the same Oops-generating kernel module – *hello_oops.ko* – that we used earlier.

Important:
When trying this out, first switch to console mode (on a PC, Ctrl-Alt-F1 should do it); as kdb will show up only here of course (not in the GUI X Windows mode).

<< On console >>

```
# uname -r
2.6.22-kdb
# insmod hello_oops.ko
# dmesg
--snip--
[4295047.638000] hello_oops 1.0 initialize
# cat /proc/oops-test
...
[ 3739.122922] *pte = 00000000
[ 3739.122975] Oops: 0000 [#9]
[ 3739.123023] PREEMPT
[ 3739.123109] Modules linked in: ipv6 autofs4 hello_oops nls_ascii vfat
fat
sd_mod usb_storage scsi_mod dm_mod button battery ac uhci_hcd shpchp
i2c_i801
i2c_core floppy
[ 3739.123889] CPU:    0
[ 3739.123891] EIP:    0060:[<d081e031>]    Not tainted VLI
[ 3739.123893] EFLAGS: 00010286   (2.6.22-kdb #1)
[ 3739.124046] EIP is at procread+0x31/0x5f [hello_oops]
[ 3739.124100] eax: 00000039   ebx: 00000039   ecx: 00000000   edx:
d081e0de
[ 3739.124156] esi: 00001000   edi: c9c52000   ebp: c9d23f30   esp:
c9d23f1c
[ 3739.124212] ds: 007b   es: 007b   fs: 0000  gs: 0033  ss: 0068
[ 3739.124267] Process cat (pid: 5807, ti=c9d23000 task=ca8e0b00
task.ti=c9d23000)
[ 3739.124322] Stack:c9c52000 d081e0a7 000016af 00000000 cd261bc0 c9d23f70
c0183b88 00000c00
```

```
[ 3739.124717]       c9d23f5c 00000000 00000c00 cd261bc0 00000000 ce29ca84
00001000 0804d858
[ 3739.125112]       00000000 00000000 ca4539c0 0804d858 c030b1a0 c9d23f90
c01561f2 c9d23f9c
[ 3739.125506] Call Trace:
[ 3739.125598]  [<c01040f5>] show_trace_log_lvl+0x19/0x2e
[ 3739.125698]  [<c01041b7>] show_stack_log_lvl+0x99/0xa1
[ 3739.125789]  [<c01043b9>] show_registers+0x1b8/0x2cb
[ 3739.124212] ds: 007b   es: 007b   fs: 0000  gs: 0033  ss: 0068
[ 3739.124267] Process cat (pid: 5807, ti=c9d23000 task=ca8e0b00
task.ti=c9d23000)
[ 3739.124322] Stack:c9c52000 d081e0a7 000016af 00000000 cd261bc0 c9d23f70
c0183b88 00000c00
[ 3739.124717]       c9d23f5c 00000000 00000c00 cd261bc0 00000000 ce29ca84
00001000 0804d858
[ 3739.125112]       00000000 00000000 ca4539c0 0804d858 c030b1a0 c9d23f90
c01561f2 c9d23f9c
[ 3739.125506] Call Trace:
[ 3739.125598]  [<c01040f5>] show_trace_log_lvl+0x19/0x2e
[ 3739.125698]  [<c01041b7>] show_stack_log_lvl+0x99/0xa1
[ 3739.125789]  [<c01043b9>] show_registers+0x1b8/0x2cb
[ 3739.125879]  [<c0104628>] die+0x100/0x1dc
[ 3739.125968]  [<c01142be>] do_page_fault+0x44d/0x528
[ 3739.126059]  [<c02fdc7a>] error_code+0x6a/0x70
[ 3739.126152]  [<c0183b88>] proc_file_read+0x116/0x229
[ 3739.126247]  [<c01561f2>] vfs_read+0x8a/0x109
[ 3739.126341]  [<c01564b3>] sys_read+0x3d/0x61
[ 3739.126431]  [<c0103c86>] sysenter_past_esp+0x5f/0x85
[ 3739.126522]  =======================
[ 3739.126571] Code: 10 c7 44 24 0c 00 00 00 00 8b 15 00 40 3d c0 8b 92 a4
00
00 00 c7 44 24 04 a7 e0 81 d0 89 04 24 89 54 24 08 e8 49 71 9b ef 89 c3
<a1>
00 00 00 00 c7 44 24 04 e0 e0 81 d0 c7 04 24 eb e0 81 d0 89
[ 3739.129038] EIP: [<d081e031>] procread+0x31/0x5f [hello_oops] SS:ESP
0068:c9d

<< The system traps into kdb now >>

Entering kdb (current=0xca8e0b00, pid 5807) Oops: Oops
due to oops @ 0xd081e031
eax = 0x00000039 ebx = 0x00000039 ecx = 0x00000000 edx = 0xd081e0de
esi = 0x00001000 edi = 0xc9c52000 esp = 0xc9d23f1c eip = 0xd081e031
ebp = 0xc9d23f30 xss = 0xc0290068 xcs = 0x00000060 eflags = 0x00010286
xds = 0x0000007b xes = 0xc9c5007b origeax = 0xffffffff &regs = 0xc9d23ee4
kdb> ps

22 sleeping system daemon (state M) processes suppressed
Task Addr       Pid   Parent [*] cpu State Thread     Command
0xca8e0b00     5807     5806  1    0   R  0xca8e0cb0 *cat

0xc1241450        1        0  0    0   S  0xc1241600  init
```

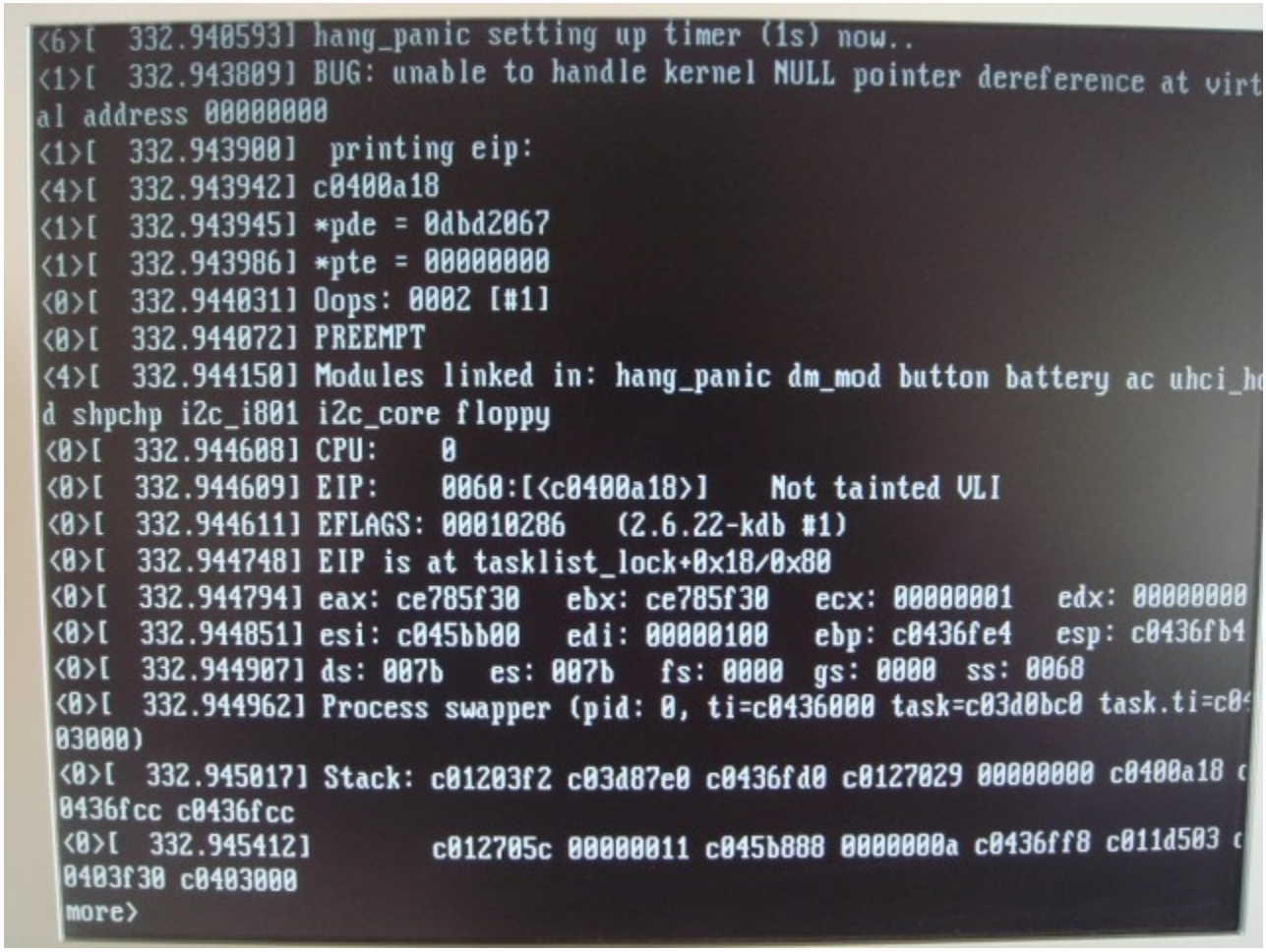
```
0xcdfd0070     1210        1  0    0   S  0xcdfd0220  udevd
0xcff70030     3472        1  0    0   R  0xcff701e0  klogd
0xcff0b490     3501        1  0    0   S  0xcff0b640  portmap
0xcff48070     3521        1  0    0   S  0xcff48220  rpc.statd
0xcff6e170     4370        1  0    0   S  0xcff6e320  apmd
0xc83be0b0     4460        1  0    0   S  0xc83be260  smartd
0xc7c7cb80     4470        1  0    0   S  0xc7c7cd30  acpid
...
...
more> q
kdb> bt

Stack traceback for pid 5807
0xca8e0b00     5807     5806  1    0   R  0xca8e0cb0 *cat
esp        eip        Function (args)
kdb_bb: address 0xffffffff not recognised
Using old style backtrace, unreliable with no arguments
esp        eip        Function (args)
0xc9d23f10 0xd081e031 [hello_oops]procread+0x31     <--- function+offset
causing the Oops
0xc9d23f34 0xc0183b88 proc_file_read+0x116
0xc9d23f74 0xc01561f2 vfs_read+0x8a
kdb> id procread               <--- [DEPRECATED] disassemble the procread
function
0xd081e000 procread:         push   %ebp
0xd081e001 procread+0x1:     mov    %esp,%ebp
0xd081e003 procread+0x3:     push   %ebx
...
0xd081e02a procread+0x2a:    call   0xc01d5178 sprintf
0xd081e02f procread+0x2f:    mov    %eax,%ebx
0xd081e031 procread+0x31:    mov    0x0,%eax
0xd081e036 procread+0x36:    movl   $0xd081e0e0,0x4(%esp)
0xd081e03e procread+0x3e:    movl   $0xd081e0eb,(%esp)
0xd081e045 procread+0x45:    mov    %eax,0x8(%esp)
kdb> id procread+0x45

0xd081e045 procread+0x45:    mov    %eax,0x8(%esp)
0xd081e049 procread+0x49:    call   0xc0119a43 printk
0xd081e04e procread+0x4e:    mov    0xc(%ebp),%eax
0xd081e051 procread+0x51:    movl   $0x1,(%eax)
0xd081e057 procread+0x57:    add    $0x10,%esp
0xd081e05a procread+0x5a:    mov    %ebx,%eax
0xd081e05c procread+0x5c:    pop    %ebx
0xd081e05d procread+0x5d:    pop    %ebp
0xd081e05e procread+0x5e:    ret
0xd081e05f hello_oops_cleanup_module:         push   %ebp
0xd081e060 hello_oops_cleanup_module+0x1:     xor    %edx,%edx
0xd081e062 hello_oops_cleanup_module+0x3:     mov    %esp,%ebp
0xd081e064 hello_oops_cleanup_module+0x5:     mov    $0xd081e0f6,%eax
0xd081e069 hello_oops_cleanup_module+0xa:     sub    $0xc,%esp
0xd081e06c hello_oops_cleanup_module+0xd:     call   0xc018453e
remove_proc_entry
```

```
0xd081e071 hello_oops_cleanup_module+0x12:    movl   $0xd081e136,0x8(%esp)
kdb> go

Catastrophic error detected
kdb_continue_catastrophic=0, type go a second time if you really want to
continue
kdb> go

Catastrophic error detected
kdb_continue_catastrophic=0, attempting to continue
Segmentation fault
#
```

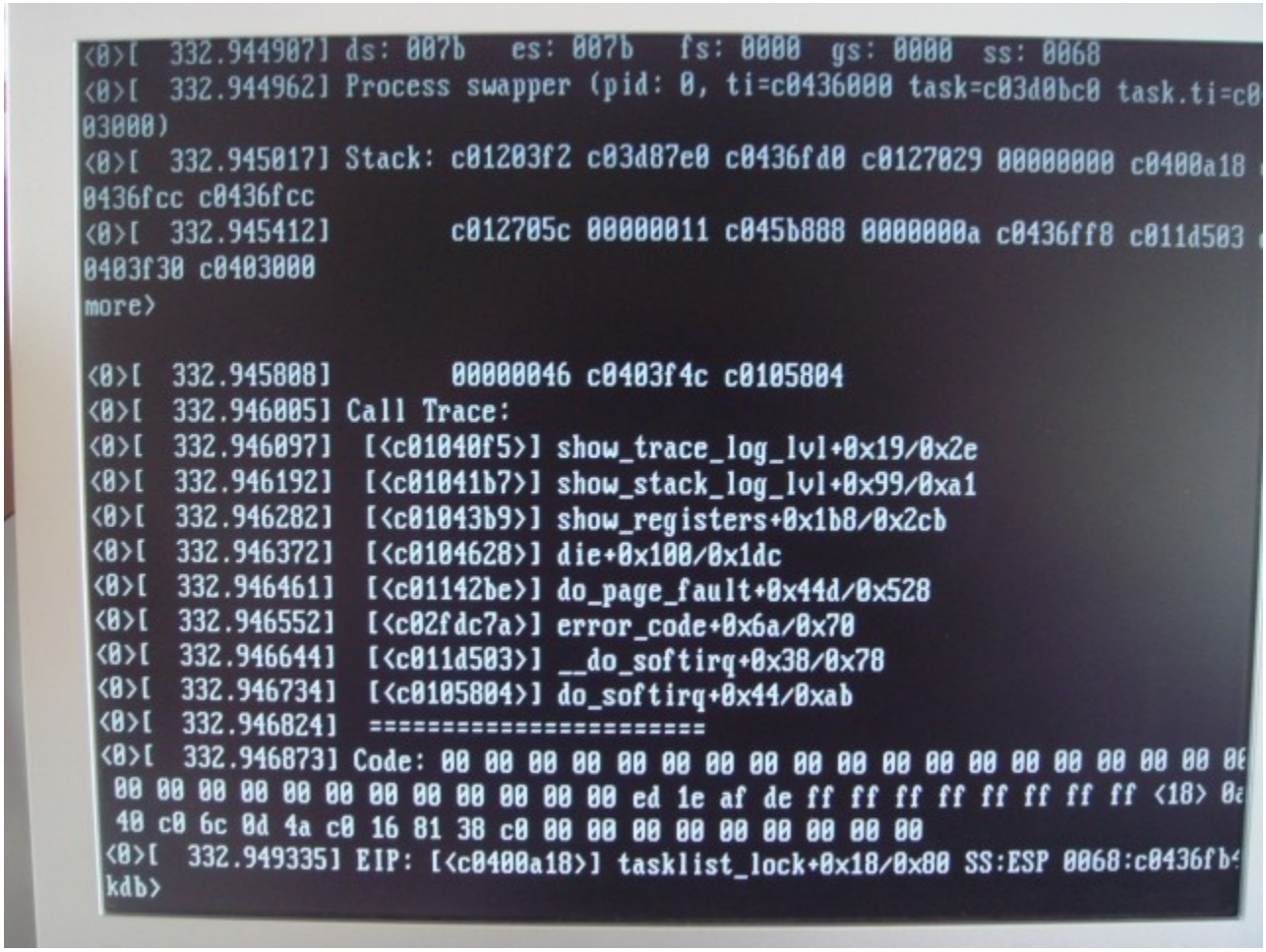
<<
Instructor Note: photos taken of the screen showing the hang_panic.ko kernel module debug steps with kdb >>

1. With (buggy) kernel module hang_panic.ko : About to trigger the Oops by reading the proc entry. Of course, we're running on the console. (Photo above).

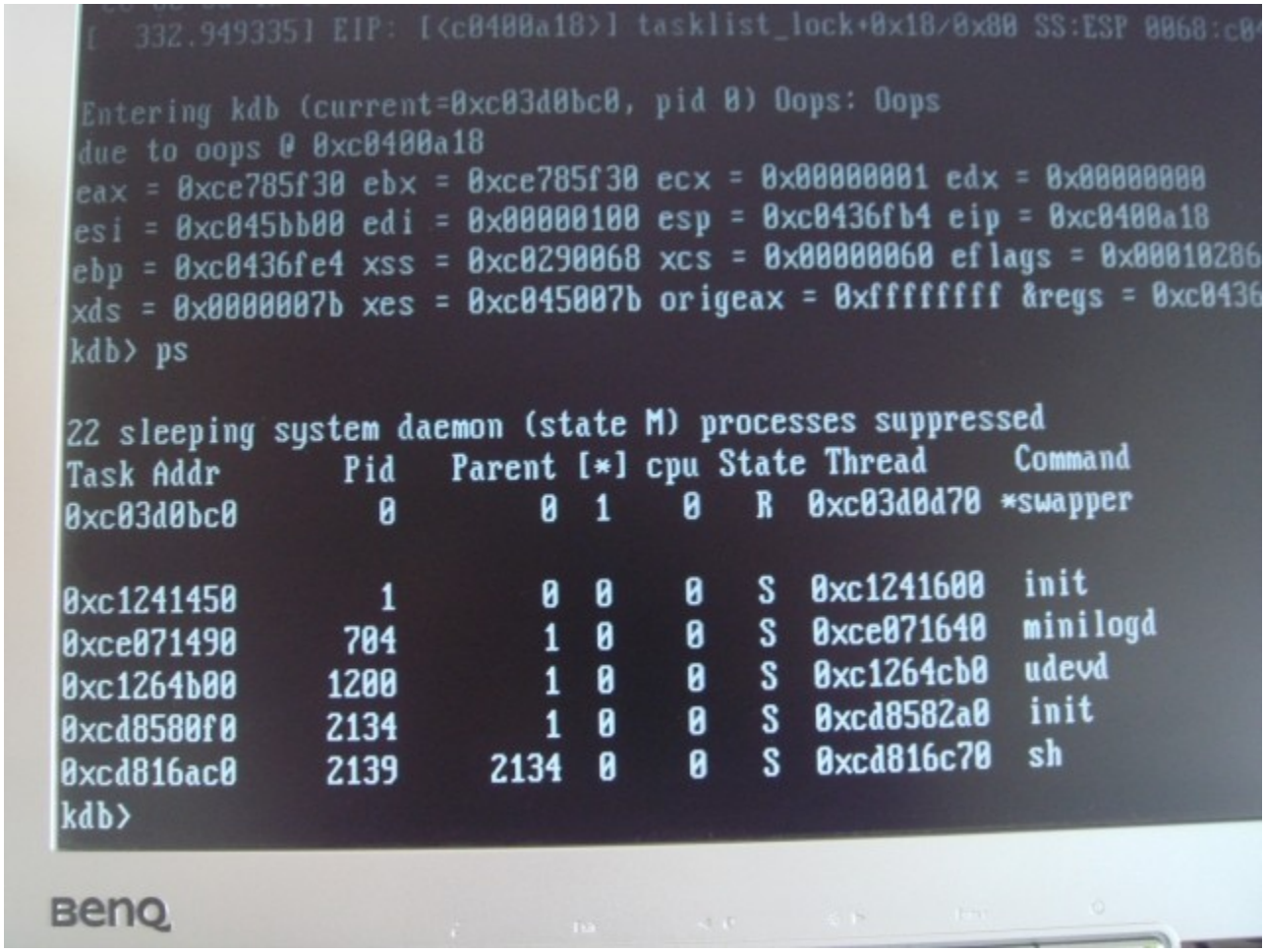
2. [Enter] pressed, the Oops occours (sure enough); and, there, kdb kicks in!



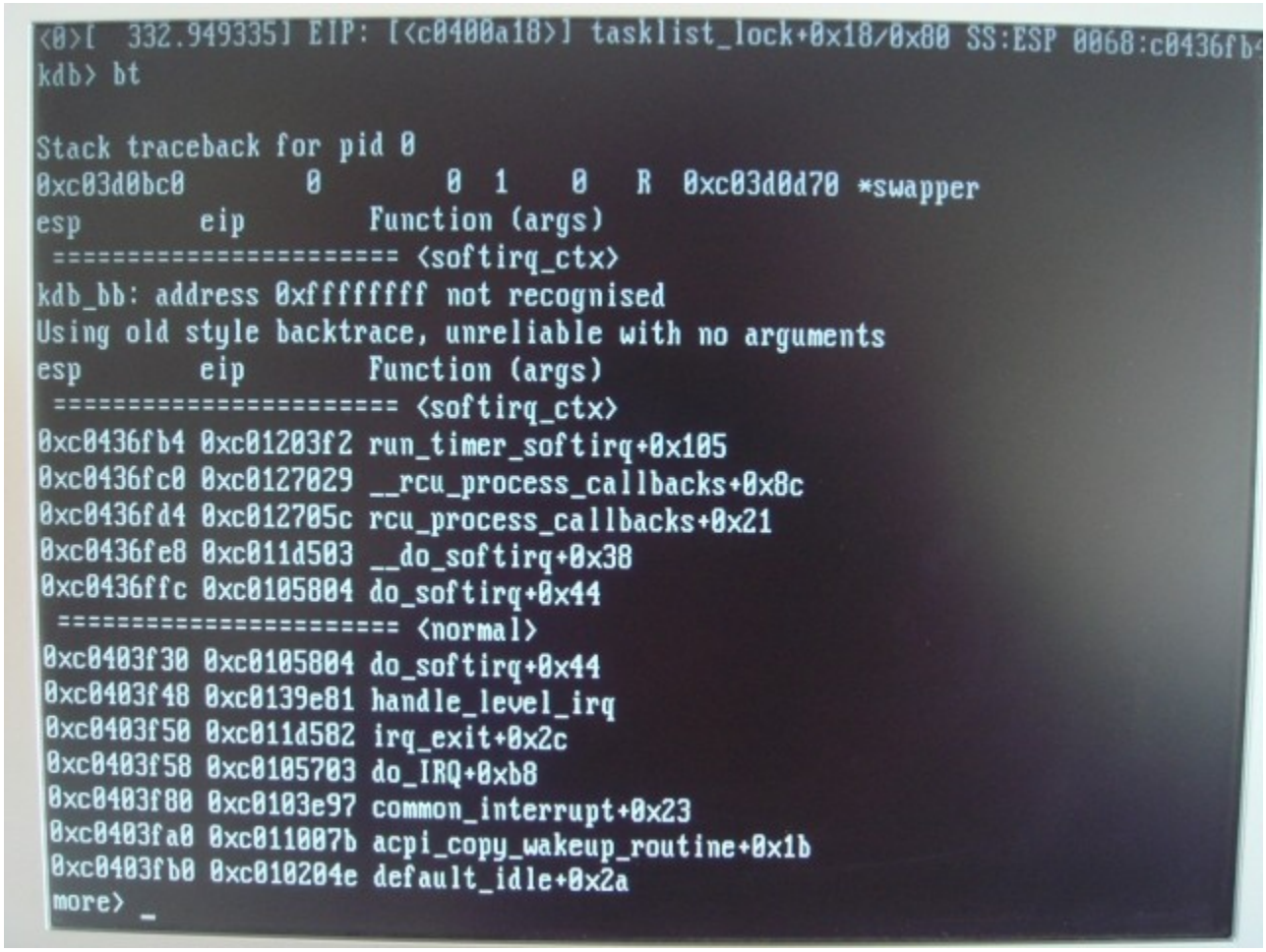
3. Output of kdb's 'dmesg' command (1 of 2)



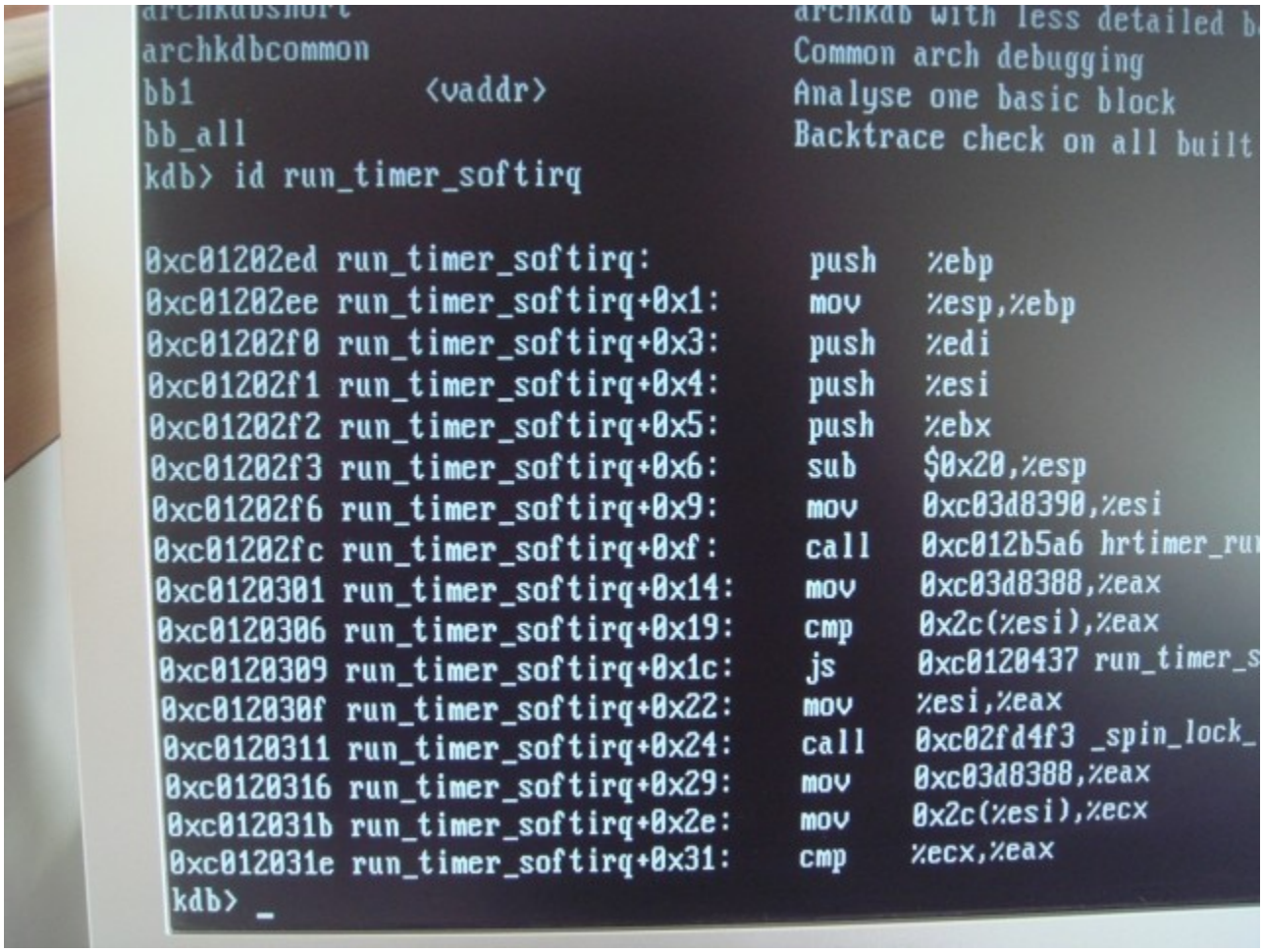
4. (Continued) Output of kdb's 'dmesg' command (2 of 2)



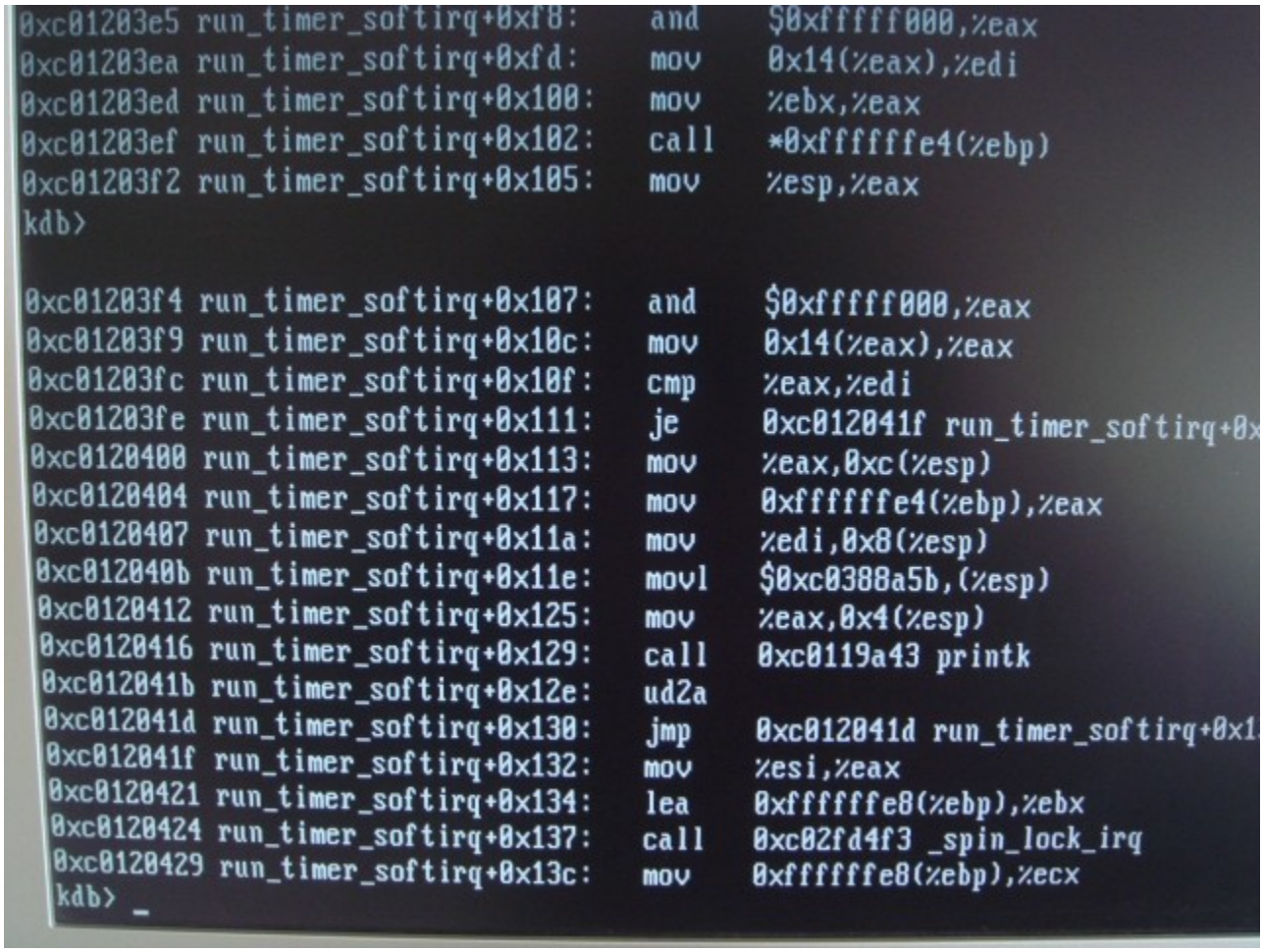
5. Output of kdb's 'ps' command; the process marked with the asterisk '*' was the one running when the Oops occurred.



6. Output of kdb's 'bt' command; note that the last recognizable function that ran is 'run_timer_softirq' in (soft)irq context. This clearly tells us that some timer code was running when the crash happened. Also, note that the code offset is 0x105 bytes.

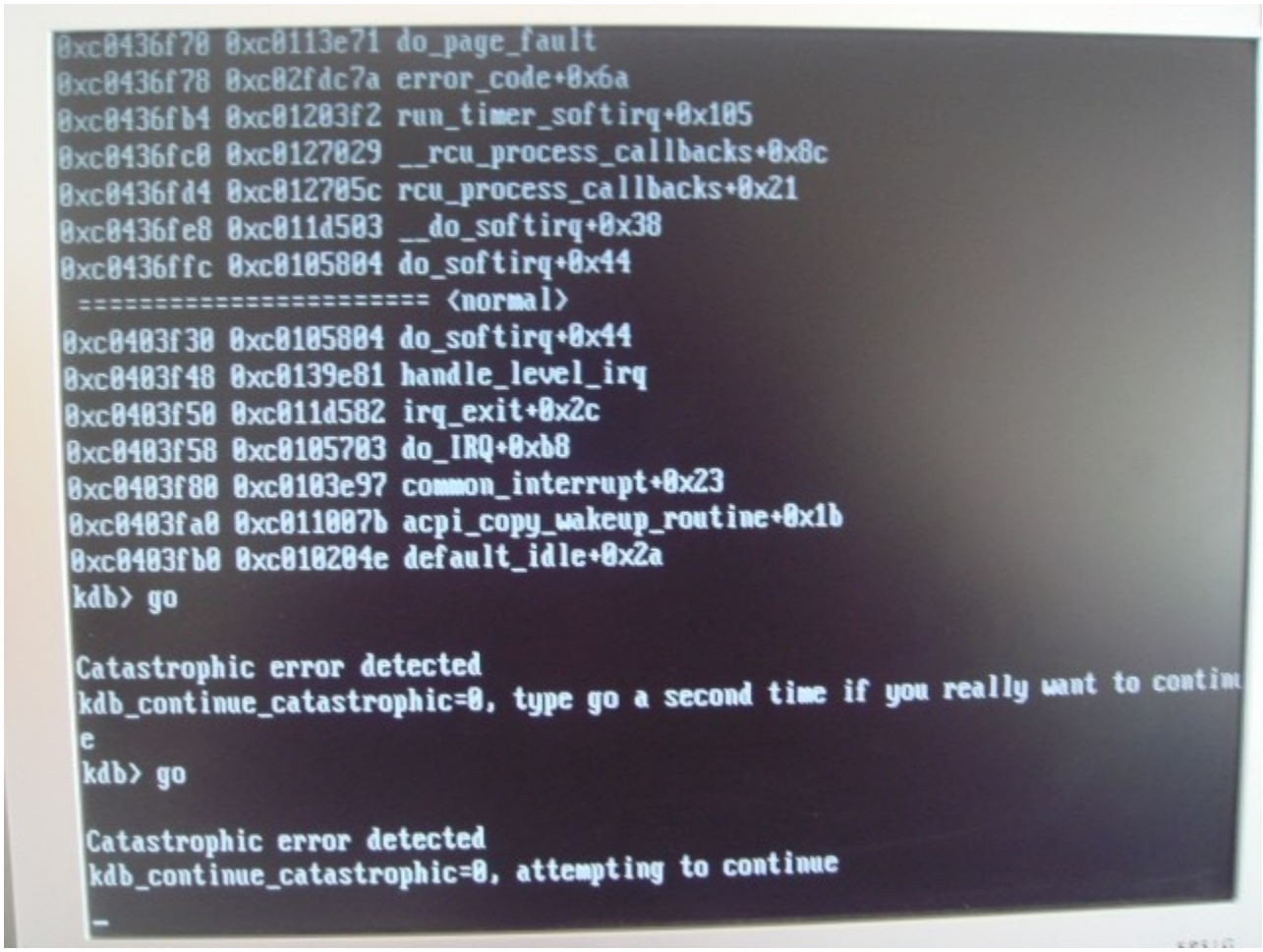


7. Investigate further by viewing the output of kdb's 'id' (disassemble) command on the relevant function (1 of 2).



8. Investigate further by viewing the output of kdb's 'id' (disassemble) command on the relevant function; this time we can see offset 0x105 (2 of 2).
Of course, this does not necessarily mean that *this* code is buggy: rather it should point us to carefully check how *we invoked* the timer. Careful study will point us to the bug, viz., the timer data structure was declared *locally* and is therefore out of scope (yielding some arbitrary junk values) when the timer fires and the kernel code attempts to access the timer_struct contents.

9. Output of kdb's 'lsmod' and 'go' commands (1 of 2).



10. Output of kdb's second 'go' command (2 of 2).

[FYI / OPTIONAL]

Useful Tips

Resource
KernelDebuggingTricks

…

gdb on vmlinux

One can disassemble a built kernel using gdb on the vmlinux image. This is useful when one gets a kernel Oops message and a stack dump - one can then disassemble the object code and see where the Oops is occuring. For example:

```
gdb  debian/build/build-generic/vmlinux
(gdb) disassemble printk
Dump of assembler code for function printk:
0xffffffff8023dce0 <printk+0>:  sub    $0xd8,%rsp
0xffffffff8023dce7 <printk+7>:  lea    0xe0(%rsp),%rax
0xffffffff8023dcef <printk+15>: mov    %rsi,0x28(%rsp)
0xffffffff8023dcf4 <printk+20>: mov    %rsp,%rsi
0xffffffff8023dcf7 <printk+23>: mov    %rdx,0x30(%rsp)
0xffffffff8023dcfc <printk+28>: mov    %rcx,0x38(%rsp)
0xffffffff8023dd01 <printk+33>: mov    %rax,0x8(%rsp)
0xffffffff8023dd06 <printk+38>: lea    0x20(%rsp),%rax
0xffffffff8023dd0b <printk+43>: mov    %r8,0x40(%rsp)
0xffffffff8023dd10 <printk+48>: mov    %r9,0x48(%rsp)
0xffffffff8023dd15 <printk+53>: movl   $0x8,(%rsp)
0xffffffff8023dd1c <printk+60>: movl   $0x30,0x4(%rsp)
0xffffffff8023dd24 <printk+68>: mov    %rax,0x10(%rsp)
0xffffffff8023dd29 <printk+73>: callq  0xffffffff8023d980 <vprintk>
0xffffffff8023dd2e <printk+78>: add    $0xd8,%rsp
0xffffffff8023dd35 <printk+85>: retq
End of assembler dump.
```

Objdump

If one has the built object code at hand, one can disassemble the object using objdump as follows:
`objdump -SdCg debian/build/build-generic/fs/dcache.o`

Using GDB to find the location where your kernel panicked or oopsed.

A quick and easy way to find the line of code where your kernel panicked or oopsed is to use GDB list command. You can do this as follows.

Lets assume your panic/oops message says something like:

```
[  174.507084] Stack:
```

```
[  174.507163]  ce0bd8ac 00000008 00000000 ce4a7e90 c039ce30 ce0bd8ac c0718b04
c07185a0
[  174.507380]  ce4a7ea0 c0398f22 ce0bd8ac c0718b04 ce4a7eb0 c037deee ce0bd8e0
ce0bd8ac
[  174.507597]  ce4a7ec0 c037dfe0 c07185a0 ce0bd8ac ce4a7ed4 c037d353 ce0bd8ac
ce0bd8ac
[  174.507888] Call Trace:
[  174.508125]  [<c039ce30>] ? sd_remove+0x20/0x70
[  174.508235]  [<c0398f22>] ? scsi_bus_remove+0x32/0x40
[  174.508326]  [<c037deee>] ? __device_release_driver+0x3e/0x70
[  174.508421]  [<c037dfe0>] ? device_release_driver+0x20/0x40
[  174.508514]  [<c037d353>] ? bus_remove_device+0x73/0x90
[  174.508606]  [<c037bccf>] ? device_del+0xef/0x150
[  174.508693]  [<c0399207>] ? __scsi_remove_device+0x47/0x80
[  174.508786]  [<c0399262>] ? scsi_remove_device+0x22/0x40
[  174.508877]  [<c0399324>] ? __scsi_remove_target+0x94/0xd0
[  174.508969]  [<c03993c0>] ? __remove_child+0x0/0x20
[  174.509060]  [<c03993d7>] ? __remove_child+0x17/0x20
[  174.509148]  [<c037b868>] ? device_for_each_child+0x38/0x60
[  174.509241]  [<c039938f>] ? scsi_remove_target+0x2f/0x60
[  174.509393]  [<d0c38907>] ? __iscsi_unbind_session+0x77/0xa0
[scsi_transport_iscsi]
[  174.509699]  [<c015272e>] ? run_workqueue+0x6e/0x140
[  174.509801]  [<d0c38890>] ? __iscsi_unbind_session+0x0/0xa0
[scsi_transport_iscsi]
[  174.509977]  [<c0152888>] ? worker_thread+0x88/0xe0
[  174.510047]  [<c01566a0>] ? autoremove_wake_function+0x0/0x40
```

Lets say you want to know what line of code represents sd_remove+0x20/0x70. cd to the ubuntu debian/build/build-generic directory in your kernel tree and run gdb on the ".o" file which has the function sd_remove() in this case in sd.o, and use the gdb "list" command, (gdb) list *(function+0xoffset), in this case function is sd_remove() and offset is 0x20, and gdb should tell you the line number where you hit the panic or oops. This has worked for me very reliably for most cases.

```
manjo@hungry:~/devel/ubuntu/kernel/ubuntu-karmic-397906/debian/build/build-
generic/drivers/scsi$ gdb sd.o
GNU gdb (GDB) 6.8.50.20090628-cvs-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
(gdb) list *(sd_remove+0x20)
0x1650 is in sd_remove (/home/manjo/devel/ubuntu/kernel/ubuntu-karmic-
397906/drivers/scsi/sd.c:2125).
2120    static int sd_remove(struct device *dev)
2121    {
```

```
2122            struct scsi_disk *sdkp;
2123
2124            async_synchronize_full();
2125            sdkp = dev_get_drvdata(dev);
2126            blk_queue_prep_rq(sdkp->device->request_queue,
scsi_prep_fn);
2127            device_del(&sdkp->dev);
2128            del_gendisk(sdkp->disk);
2129            sd_shutdown(dev);
(gdb)
```

---

objdump

Use the objdump utility to glean useful imformation from the object file.

```
$ man objdump
--snip--
       -g
       --debugging
           Display  debugging  information.  This attempts to parse
debugging
information stored in the file and print it out using a C like syntax.
Only certain types of debugging information have been implemented. Some
other types are            supported by readelf -w.

       -e
       --debugging-tags
           Like -g, but the information is generated in a format compatible
with ctags tool.

       -d
       --disassemble
           Display  the  assembler  mnemonics for the machine instructions
from objfile.
This option only disassembles those sections which are expected to contain
instructions.

       -D
       --disassemble-all
           Like -d, but disassemble the contents of all sections, not just
those expected to contain instructions.

--snip--

      -S
       --source
           Display source code intermixed with disassembly, if possible.
Implies -d.

--snip--

     --start-address=address
           Start displaying data at the specified address.  This affects
the output of the -d, -r and -s options.

       --stop-address=address
           Stop displaying data at the specified address.  This affects the
output of the -d, -r and -s options.

--snip--
$
```

A very useful piece of information one obtains from a backtrace ('bt'), either from gdb directly, or indirectly, via KGDB's or KDB's backtrace command, is the exact location of where the processor was in a function when the Oops occured. We can easily search for and display the disassembled code for this offset (and the code around it) using objdump.

For example,we earlier saw a demo of generating an Oops using the hello_oops.ko kernel module, which of course did this by dereferencing a NULL pointer.

Here's a snippet of that Oops dump:

```
...
CPU:    0
EIP:    0060:[<d085d041>]    Tainted: G     U VLI
EFLAGS: 00010286   (2.6.16.21-0.8-default #1)
EIP is at procread+0x1d/0x48 [hello_oops]
eax: 00000039   ebx: d085d024   ecx: c9ec9f40   edx: c9ec9f40
esi: 00000400   edi: 00000400   ebp: c3e55000   esp: c9ec9f38
ds: 007b   es: 007b   ss: 0068
Process cat (pid: 6154, threadinfo=c9ec8000 task=c197aab0)
Stack: <0>c3e55000 d085d132 0000180a 00000000 d085d024 c0174370 00000400 c9ec9f70
       00000000 00000400 08051000 00000000 cfc11e40 00000000 00000000 c87c3740
       c0174270 08051000 00000400 c014b351 c9ec9fa4 c87c3740 fffffff7 00000400
Call Trace:
 [<d085d024>] procread+0x0/0x48 [hello_oops]
 [<c0174370>] proc_file_read+0x100/0x234
 [<c0174270>] proc_file_read+0x0/0x234
 [<c014b351>] vfs_read+0xa8/0x14d
 [<c014b6bd>] sys_read+0x3c/0x63
 [<c010299b>] sysenter_past_esp+0x54/0x79
Code: 68 9c d0 85 d0 e8 9d ac 8b ef 83 c4 0c c3 53 ba 00 e0 ff ff 6a 00 21 e2 8b
12 ff b2 b0 00 00 00 68 32 d1 85 d0 50 e8 8f 81 94 ef <ff> 35 00 00 00 00 89 c3 68
91 d0 85 d0 68 6b d1 85 d0 e8 65 ac
```

You can see above (highlighted in bold) that the processor was at the procread function at an offset of 0x1d bytes from the start of the function (whose length is approximately 0x48 bytes) when the Oops occured.
So what's at offset 0x1d? Use objdump to easily discover the same:

```
$ objdump -g -d hello_oops.o

hello_oops.o:     file format elf32-i386

Disassembly of section .text:

00000000 <procread>:
   0:   53                      push   %ebx
   1:   ba 00 f0 ff ff          mov    $0xffff000,%edx
   6:   21 e2                   and    %esp,%edx
   8:   8b 0a                   mov    (%edx),%ecx
   a:   ff 72 10                pushl  0x10(%edx)
   d:   ff b1 94 00 00 00       pushl  0x94(%ecx)
  13:   68 00 00 00 00          push   $0x0
  18:   50                      push   %eax
  19:   e8 fc ff ff ff          call   1a <procread+0x1a>
  1e:   ff 35 00 00 00 00       pushl  0x0
```

```
  24:   89 c3                   mov    %eax,%ebx
  26:   68 39 00 00 00          push   $0x39
  2b:   68 44 00 00 00          push   $0x44
...
...
$
```

Another example, Mixing source and assembler:

```
$ objdump -g -S hang_panic.o

hang_panic.o:     file format elf32-i386

Disassembly of section .text:

00000000 <ding>:
#define procname        "lab2_test"

static int ding(unsigned long dope)
{
        printk(KERN_INFO
   0:   50                      push   %eax
   1:   68 00 00 00 00          push   $0x0
   6:   68 0b 00 00 00          push   $0xb
   b:   e8 fc ff ff ff          call   c <ding+0xc>
                "%s: In timeout function \"ding\" now...dope=%ld\n",
                MODULE_NAME, dope);
/*
info: PID %d, interrupt-context: %d, processor # %d\n",
        MODULE_NAME, dope, current->pid,
        in_interrupt() ? 1:0, smp_processor_id());
*/
        return 0;
  10:   83 c4 0c                add    $0xc,%esp
}
  13:   31 c0                   xor    %eax,%eax


/* read callback */
static int my_procread(char *buf, char **start, off_t offset,
                   int count, int *eof, void *data)
{
        int n;
        struct timer_list timer;

        n = sprintf(buf, "In proc read callback.\n\
                current->pid, smp_processor_id());

        /* Setup a timeout */
```

```
:objdump: hang_panic.o: no recognized debugging information
       printk(KERN_INFO "%s setting up timer (1s) now..\n",
                MODULE_NAME);
        init_timer (&timer);
        timer.expires = HZ;     /* 1 sec */
        timer.function = ding;
        timer.data = jiffies;
        add_timer (&timer);

        *eof=1;
        return n;
}

/*
 * function hang_panic_init
 */
static int __init hang_panic_init_module(void)
{

        if (!create_proc_read_entry (procname, 0444, /* mode */
                NULL,           /* parent dir is /proc */
                my_procread,    /* read callback */
                NULL )) {       /* client data or tag */
                        printk(KERN_ALERT "Proc entry creation failure,
aborting..\n");
  15:   c3                      ret

00000016 <my_procread>:
  16:   53                      push   %ebx
  17:   ba 00 f0 ff ff          mov    $0xfffff000,%edx
  1c:   83 ec 30                sub    $0x30,%esp
  1f:   21 e2                   and    %esp,%edx
  21:   8b 0a                   mov    (%edx),%ecx
  23:   ff 72 10                pushl  0x10(%edx)
  26:   ff b1 94 00 00 00       pushl  0x94(%ecx)
  2c:   68 3d 00 00 00          push   $0x3d
  31:   50                      push   %eax
  32:   e8 fc ff ff ff          call   33 <my_procread+0x1d>
  37:   68 00 00 00 00          push   $0x0
  3c:   89 c3                   mov    %eax,%ebx
  3e:   68 76 00 00 00          push   $0x76
  43:   e8 fc ff ff ff          call   44 <my_procread+0x2e>
  48:   83 c4 18                add    $0x18,%esp
  4b:   ba 01 00 00 00          mov    $0x1,%edx
  50:   89 e0                   mov    %esp,%eax
  52:   89 54 24 0c             mov    %edx,0xc(%esp)
  56:   8b 15 00 00 00 00       mov    0x0,%edx
  5c:   b9 ad 4e ad de          mov    $0xdead4ead,%ecx
  61:   c7 44 24 20 00 00 00    movl   $0x0,0x20(%esp)
  68:   00
  69:   89 54 24 1c             mov    %edx,0x1c(%esp)
  6d:   ba e8 03 00 00          mov    $0x3e8,%edx
```

```
  72:   c7 44 24 14 6e ad 87    movl   $0x4b87ad6e,0x14(%esp)
  79:   4b
  7a:   89 4c 24 10             mov    %ecx,0x10(%esp)
  7e:   c7 44 24 08 e8 03 00    movl   $0x3e8,0x8(%esp)
  85:   00
  86:   c7 44 24 18 00 00 00    movl   $0x0,0x18(%esp)
  8d:   00
  8e:   e8 fc ff ff ff          call   8f <my_procread+0x79>
  93:   8b 44 24 3c             mov    0x3c(%esp),%eax
                      return -ENOMEM;
        }

        /* Delay for 1 second; just for the heck of it */
        printk(KERN_INFO "%s delaying for 1s now..\n", MODULE_NAME);
        set_current_state (TASK_INTERRUPTIBLE);
...
...
  79:   c3                      ret
  7a:   68 2d 01 00 00          push   $0x12d
  7f:   68 00 00 00 00          push   $0x0
  84:   68 31 01 00 00          push   $0x131
  89:   e8 fc ff ff ff          call   8a <init_module+0x8a>
  8e:   31 c0                   xor    %eax,%eax
  90:   83 c4 0c                add    $0xc,%esp
  93:   c3                      ret
$
```

IMP: How exactly can one disassemble the kernel image file for a given function (by name)?

The following shell script lets you achieve this, using the extremely versatile tool objdump; it relies on your providing the path to the *uncompressed* kernel image (*vmlinux)* and the function name you'd like to disassemble. Using the '-S' switch with objdump even attempts to intermix C source with assembly!

```
$ cat disasfunc.sh
#!/bin/sh
# Source: http://kgdb.linsyssoft.com/downloads/miscmacros
#
# Eg.
# ./disasfunc.sh <path/to/>vmlinux do_fork
#
if [ $# != 2 ]
then
      echo disasfun objectfile functionname
      exit 1
fi
OBJFILE=$1
FUNNAME=$2
```

```
ADDRSZ=`objdump -t $OBJFILE | gawk -- "{
     if (\\\$3 == \\"F\\" && \\\$6 == \\"$FUNNAME\\") {
          printf(\\"%s %s\\", \\\$1, \\\$5)
     }
}"`
ADDR=`echo $ADDRSZ | gawk "{ printf(\\"%s\\",\\\$1)}"`
SIZE=`echo $ADDRSZ | gawk "{ printf(\\"%s\\",\\\$2)}"`
if [ -z "$ADDR" -o -z "$SIZE" ]
then
     echo Cannot find address or size of function $FUNNAME
     exit 2
fi
objdump -S $OBJFILE --start-address=0x$ADDR --stop-address=$((0x$ADDR +
0x$SIZE))
$
```

Eg.

```
$ ./disasfunc.sh <path>/<to>/vmlinux strnlen

<path>/<to>/vmlinux:     file format elf32-i386


Disassembly of section .text:

08078ac0 <strnlen>:
EXPORT_SYMBOL(memscan);
#endif

#ifdef __HAVE_ARCH_STRNLEN
size_t strnlen(const char *s, size_t count)
{
 8078ac0:  55                    push   %ebp
 8078ac1:  89 e5                 mov    %esp,%ebp
     int d0;
     int res;
     asm volatile("movl %2,%0\n\t"
 8078ac3:  8b 55 0c              mov    0xc(%ebp),%edx
 8078ac6:  8b 4d 08              mov    0x8(%ebp),%ecx
 8078ac9:  89 c8                 mov    %ecx,%eax
 8078acb:  eb 06                 jmp    8078ad3 <strnlen+0x13>
 8078acd:  80 38 00              cmpb   $0x0,(%eax)
 8078ad0:  74 07                 je     8078ad9 <strnlen+0x19>
 8078ad2:  40                    inc    %eax
 8078ad3:  4a                    dec    %edx
 8078ad4:  83 fa ff              cmp    $0xffffffff,%edx
 8078ad7:  75 f4                 jne    8078acd <strnlen+0xd>
 8078ad9:  29 c8                 sub    %ecx,%eax
          "3:\tsubl %2,%0"
          : "=a" (res), "=&d" (d0)
          : "c" (s), "1" (count)
          : "memory");
```

```
      return res;
}
 8078adb:  5d                    pop    %ebp
 8078adc:  c3                    ret
$
```

Setting gdb debugging flags:

```
EXTRA_CFLAGS += -O0 -DDEBUG -g -ggdb -g-dwarf4
```

```
$ man gcc
...
...
```

-g

Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

On most systems that use stabs format, -g enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but will probably make other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use -gstabs+, -gstabs, -gxcoff+, -gxcoff, or -gvms (see below).

Unlike most other C compilers, GCC allows you to use -g with -O. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

The following options are useful when GCC is generated with the capability for more than one debugging format.

-ggdb

Produce debugging information for use by GDB. This means to use the most expressive format available (DWARF, stabs, or the native format if neither of those are supported), including GDB extensions if at all possible.

…

-gdwarf

-gdwarf-version

Produce debugging information in DWARF format (if that is supported). The value of version may be either 2, 3, 4 or 5; the default version for most targets is 4. DWARF Version 5 is only experimental.

...

$

Enabling these flags in the Makefile is pretty common practise to enable symbolic debugging information.

More gcc / as flags:

```
EXTRA_CFLAGS += -DDEBUG -g -ggdb -gdwarf-4  -Wa,-a,-ad,-as
```

These gcc flags cause the **assembler** ('as') to emit a wealth of information: essentially, one is able to view a mix of the source code (this includes the source of all '#include 'd files), the resulting assembly and machine language generated by the compiler/assembler, all in a printer-friendly (paginated) ASCII text format.

Specifically, the meanings of the flags are:

```
$ man gcc
...
...
       Passing Options to the Assembler

       You can pass options to the assembler.

       -Wa,option
           Pass option as an option to the assembler.  If option contains
commas, it is split into multiple options at the commas.
...
$
$ man as
...
...
       -a[cdhlmns]
           Turn on listings, in any of a variety of ways:

           -ac omit false conditionals
           -ad omit debugging directives
           -ah include high-level source
           -al include assembly
           -am include macro expansion
           -an omit forms processing
           -as include symbols

           =file
               set the name of the listing file

           You  may  combine  these  options; for example, use -aln for
assembly listing without forms processing.  The =file option, if used, must
be the last one.  By itself, -a defaults to -ahls.
...
$
```

When make is run, we get output similar to the following:

```
$ make
make -C /lib/modules/2.6.9-34.ELsmp/build SUBDIRS=/mnt/<...>/lab2 modules
make[1]: Entering directory `/usr/src/kernels/2.6.9-34.EL-smp-i686'
  CC [M]  /mnt/<...>/lab2/hang_panic.o
GAS LISTING                page 1


   1                  .file "hang_panic.c"
   9                  .Ltext0:
  10                  .section .rodata.str1.1,"aMS",@progbits,1
  11                  .LC0:
  12 0000 68616E67    .string "hang_panic"
  12      5F70616E
  12      696300
  13                  .LC1:
  14 000b 3C363E25    .string "<6>%s: In timeout function \"ding\"
now...dope=%ld\n"
  14      733A2049
  14      6E207469
  14      6D656F75
  14      74206675
  15                  .text
  17                  ding:
  18                  .LFB463:
  19                  .file 1 "/mnt/<...>/lab2/hang_panic.c"
   1:/mnt/<...>/lab2/hang_panic.c **** /*
   2:/mnt/<...>/lab2/hang_panic.c ****  * hang_panic.c v1.0 <date>
   3:/mnt/<...>/lab2/hang_panic.c ****  *
   4:/mnt/<...>/lab2/hang_panic.c ****  *
   5:/mnt/<...>/lab2/hang_panic.c ****  * Simple kernel module that delays
for a second during init,
   6:/mnt/<...>/lab2/hang_panic.c ****  * creates a proc file; when the
proc file is read, a timeout
   7:/mnt/<...>/lab2/hang_panic.c ****  * is setup (for 1 second). Once
timeout occours, the function
   8:/mnt/<...>/lab2/hang_panic.c ****  * 'ding' is called.
   9:/mnt/<...>/lab2/hang_panic.c ****  *
  10:/mnt/<...>/lab2/hang_panic.c ****  * But, of course, it has a bug (or
bugs). Read, run, test,
  11:/mnt/<...>/lab2/hang_panic.c ****  * debug, correct, verify, repeat.
  12:/mnt/<...>/lab2/hang_panic.c ****  *
  13:/mnt/<...>/lab2/hang_panic.c ****  * (c) kaiwan.
  14:/mnt/<...>/lab2/hang_panic.c ****  *
  15:/mnt/<...>/lab2/hang_panic.c ****  * This program is free software;
you can redistribute it
  16:/mnt/<...>/lab2/hang_panic.c ****  * and/or modify it under the terms
of the GNU Library
[…]
  31:/mnt/<...>/lab2/hang_panic.c ****  */
  32:/mnt/<...>/lab2/hang_panic.c ****
```

```
  33:/mnt/<...>/lab2/hang_panic.c **** #include <linux/module.h>
  34:/mnt/<...>/lab2/hang_panic.c **** #include <linux/kernel.h>
  35:/mnt/<...>/lab2/hang_panic.c **** #include <linux/init.h>
  36:/mnt/<...>/lab2/hang_panic.c **** #include <linux/sched.h>
  37:/mnt/<...>/lab2/hang_panic.c **** #include <linux/proc_fs.h>
  38:/mnt/<...>/lab2/hang_panic.c **** #include <linux/timer.h>
  39:/mnt/<...>/lab2/hang_panic.c **** #include <linux/interrupt.h>
  40:/mnt/<...>/lab2/hang_panic.c ****

GAS LISTING                page 2


  41:/mnt/<...>/lab2/hang_panic.c **** #define MODULE_VER   "1.0"
  42:/mnt/<...>/lab2/hang_panic.c **** #define MODULE_NAME    "hang_panic"
  43:/mnt/<...>/lab2/hang_panic.c **** #define procname     "lab2_test"
  44:/mnt/<...>/lab2/hang_panic.c ****
  45:/mnt/<...>/lab2/hang_panic.c **** static int ding(unsigned long dope)
  46:/mnt/<...>/lab2/hang_panic.c **** {
  20                   .loc 1 46 0
  21                   .LVL0:
  47:/mnt/<...>/lab2/hang_panic.c ****     printk(KERN_INFO
  22                   .loc 1 47 0
  23 0000 50           pushl %eax
  24                   .LCFI0:
  25 0001 68000000     pushl $.LC0
  25      00
  26                   .LCFI1:
  27 0006 680B0000     pushl $.LC1
  27      00
  28                   .LCFI2:
  29 000b E8FCFFFF     call printk
  29      FF
  30                   .LVL1:
  48:/mnt/<...>/lab2/hang_panic.c ****           "%s: In timeout
function \"ding\" now...dope=%ld\n",
  49:/mnt/<...>/lab2/hang_panic.c ****           MODULE_NAME, dope);
  50:/mnt/<...>/lab2/hang_panic.c **** /*
  51:/mnt/<...>/lab2/hang_panic.c **** info: PID %d, interrupt-context: %d,
processor # %d\n",
  52:/mnt/<...>/lab2/hang_panic.c ****     MODULE_NAME, dope, current-
>pid,
  53:/mnt/<...>/lab2/hang_panic.c ****     in_interrupt() ? 1:0,
smp_processor_id());
  54:/mnt/<...>/lab2/hang_panic.c **** */
  55:/mnt/<...>/lab2/hang_panic.c ****     return 0;
  31                   .loc 1 55 0
  32 0010 83C40C       addl $12,%esp
  33                   .LCFI3:
  56:/mnt/<...>/lab2/hang_panic.c **** }
  34                   .loc 1 56 0
  35 0013 31C0         xorl %eax,%eax
  36 0015 C3           ret
```

```
  37                   .LFE463:
  39                   .section .rodata.str1.1
  40                   .LC2:

...
--snip--
...

  63:/mnt/<...>/lab2/hang_panic.c ****     int n;
  64:/mnt/<...>/lab2/hang_panic.c ****     struct timer_list timer;
  65:/mnt/<...>/lab2/hang_panic.c ****
  66:/mnt/<...>/lab2/hang_panic.c ****     n = sprintf(buf, "In proc read
callback.\n\
  67:/mnt/<...>/lab2/hang_panic.c **** PID %d running on processor # %d\n",
  68:/mnt/<...>/lab2/hang_panic.c ****           current->pid,
smp_processor_id());
  69:/mnt/<...>/lab2/hang_panic.c ****
  70:/mnt/<...>/lab2/hang_panic.c ****     /* Setup a timeout */
  71:/mnt/<...>/lab2/hang_panic.c ****     printk(KERN_INFO "%s setting up
timer (1s) now..\n",
  90                   .loc 1 71 0
  91 0037 68000000     pushl $.LC0
  91      00
  92                   .LCFI10:
  93                   .loc 3 9 0
  94 003c 89C3         movl %eax,%ebx
  95                   .LVL6:

...
...

 323 00bb C3           ret
…
 333                   __mod_author124:
 334 0000 61757468     .string "author=My Name"
 334      6F723D4D
 334      79204E61
 334      6D6500
 335 000f 00000000     .align 32
 …
...
jiffies
__mod_timer
create_proc_entry
schedule_timeout
  Building modules, stage 2.
  MODPOST
  CC      /mnt/<...>/lab2/hang_panic.mod.o
  LD [M]  /mnt/<...>/lab2/hang_panic.ko
make[1]: Leaving directory `/usr/src/kernels/2.6.9-34.EL-smp-i686'
$
```

Misc:

See the book "Linux Debugging and Performance Tuning", by Steve Best for excellent "real-world" kdb Debug sessions; page 354 onwards.

Kinda interesting:

Determining cause of Linux kernel panic – on stackexchange

- Spotted this one:
http://lkml.org/lkml/2010/6/1/138
which the kernel bugzilla reported here.
