Kdump -very Briefly!

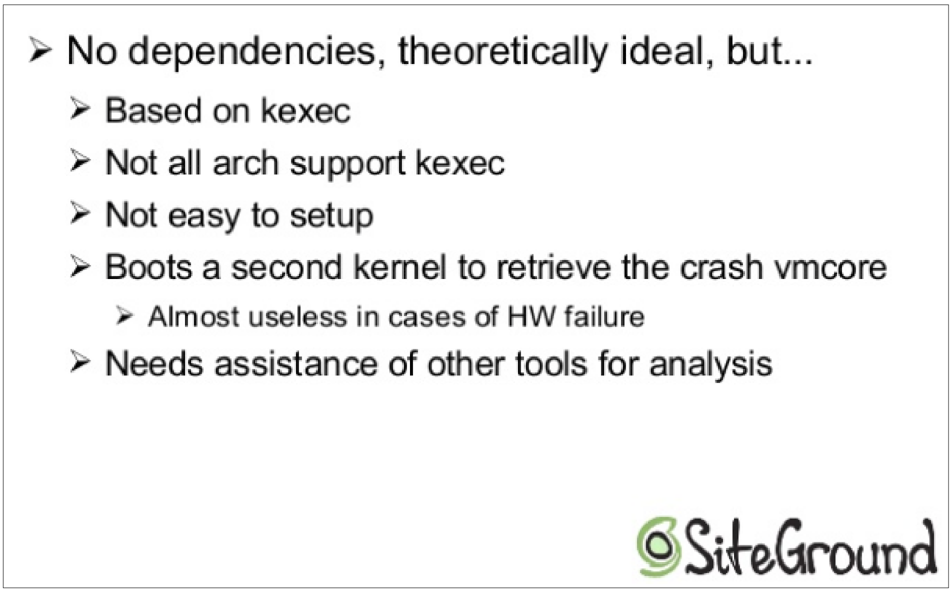
'Official' Kernel Documentation for Kdump - The kexec-based Crash Dumping Solution

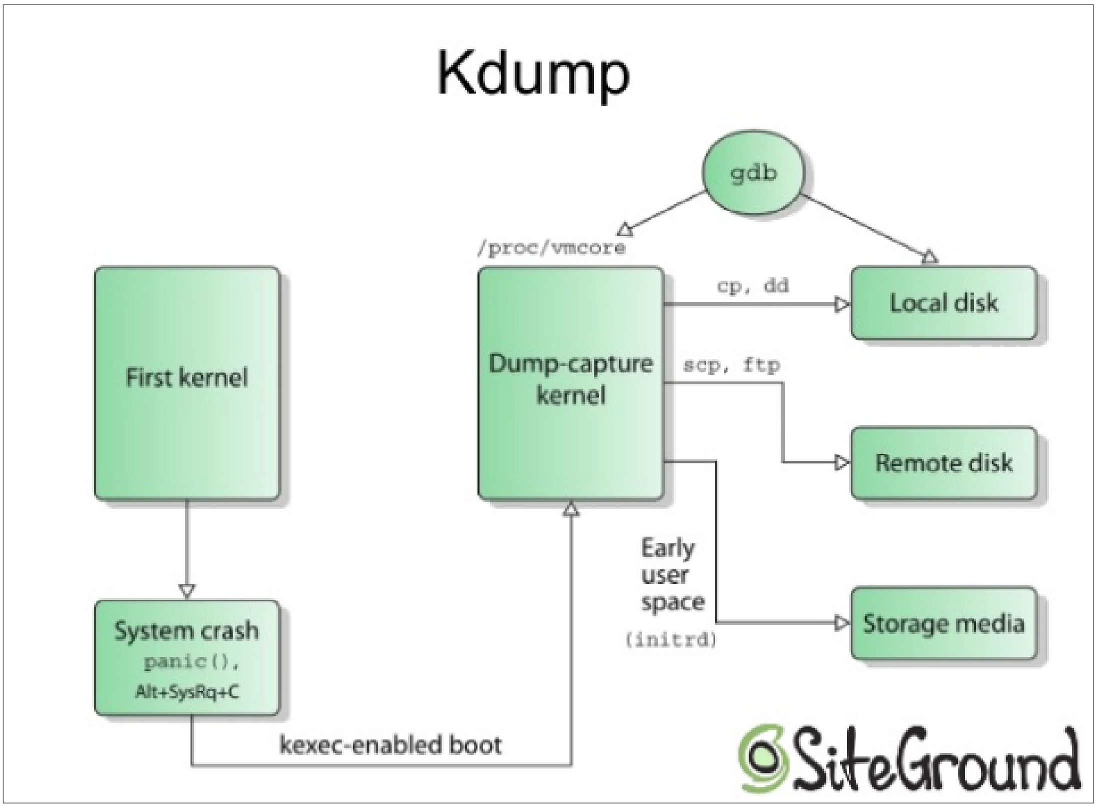
Oops! Debugging Kernel Panics, LJ, Aug 2019

Kdump setup on Ubuntu

kdump (Linux) on Wikipedia

Source: https://www.slideshare.net/azilian/linux-kernel-crashdump





Tip: Analyze the kdump image with crash (instead of GDB)

While writing the *Linux Kernel Programming, 2nd Ed* book, wrt this:
"What if your system just hangs upon insertion of this LKM? Well, that's a taste of the difficulty of kernel debugging! One thing you can try (which worked for me when trying this very example on a x86_64 Fedora 29 VM) is to reboot the hung VM and look up the kernel log by leveraging systemd's powerful `journalctl(1)` utility with the `journalctl --since="1 hour ago"` command; you should be able to see the printks from `lockdep` now. Again, unfortunately, it's not guaranteed that the key portion of the kernel log is saved to disk (at the time it hung) for `journalctl` to be able to retrieve. This is why using the kernel's **kdump** feature – and then performing postmortem analysis of the kernel dump image file with `crash(8)` – can be a lifesaver (see resources on using `kdump` and *crash* in the *Further reading* section for this chapter)."
...my Technical Reviewer, Chi Thanh Hoang, a very experienced embedded developer, commented:
"I enforce kdump on commercial systems I work, people are often very happy to use crash tool later to figure out issues, removing all the guessing work they would do. Crash tool can easily dump kernel log that was still in RAM, no need for systemd like you said, kdump works 100%."

---

1. Build a kernel with Kdump support

A simple Kdump kernel config check script:

```
#!/bin/bash
name=$(basename $0)

usage()
{
  echo "Usage: ${name} [kernel-config-file]"
}
```

```
[[ $1 = "-h" ]] && {
  usage ; exit 0
}
if [[ $# -ge 1 ]] ; then
   KCONFIG=$1
else
   # NOTE! ASSUMING arch is x86-64
   KCONFIG=/boot/config-$(uname -r)
fi

[[ ! -f ${KCONFIG} ]] && {
   echo "${name}: kernel config file ${KCONFIG} not found, aborting" ; exit 1
}
echo "Kernel config file: ${KCONFIG}"

# From https://docs.kernel.org/admin-guide/kdump/kdump.html
KCONFIGS_ARR=(KEXEC KEXEC_CORE CRASH_CORE SYSFS DEBUG_INFO CRASH_DUMP PROC_VMCORE
RELOCATABLE)

for KCONF in ${KCONFIGS_ARR[@]} ; do
    printf "checking for CONFIG_%-15s" ${KCONF}
    grep "CONFIG_${KCONF}" ${KCONFIG} >/dev/null 2>&1
    [[ $? -ne 0 ]] && printf "   NOT found!\n" || printf "   [OK]\n"
done
exit 0
```

2. Boot into the regular kernel reserving space for the dump kernel:

Now boot with the kernel cmdline param '`crashkernel=Y@X`'
On x86-64, '`crashkernel=256M`' is sufficient.

3. After boot, load the dump-capture kernel into reserved RAM:

```
sudo kexec -p /boot/vmlinuz-5.10.153 --initrd /boot/initrd.img-5.10.153 \
 --append "irqpoll nr_cpus=1 reset_devices root=UUID=b67e<...> 3"
```

---

1. Bootloader kernel command-line to first kernel:

```
console=tty<...> rootfstype=ext4 root=/dev/<...> rw rootwait init=/sbin/init
crashkernel=128M@0x78000000
```

2. Just after first / primary kernel has booted:

```
$ dmesg |grep -i crash
[    0.000000] Reserving 128MB of memory at 1920MB for crashkernel (System RAM: 1784MB)
[    0.000000] Kernel command line: console=ttymxc0 rootfstype=ext4 root=/dev/mmcblk0
rw rootwait init=/sbin/init crashkernel=128M@0x78000000
$
```

Once *kexec* has successfully run on the original (first) kernel, can verify via sysfs (CONFIG_SYSFS required for exactly this):

3. Before *kexec*

```
ARM / $ ls /sys/kernel/kexec_*
/sys/kernel/kexec_crash_loaded   /sys/kernel/kexec_loaded
/sys/kernel/kexec_crash_size
ARM / $ cat /sys/kernel/kexec_*
0
134217728
0
ARM / $
```

4. After successful *kexec*

```
ARM / $ cat /sys/kernel/kexec_*
cat /sys/kernel/kexec_*
1
134217728
0
ARM / $
```

5. Cause a panic!

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

Demo on a Qemu-emulated Freescale i.MX6 platform ([more details here on the kaiwanTECH blog]()). The first kernel crashes, and then, the dump-capture kernel boots!

```
ARM / $ id
uid=0 gid=0
ARM / $ echo c > /proc/sysrq-trigger    << or an actual kernel Oops/panic occurs >>
[  460.417261] sysrq: SysRq : Trigger a crash
[  460.423293]
[  460.424273] ===============================  << from lockdep, ignore for now >>
[  460.424965] [ INFO: suspicious RCU usage. ]
[  460.426708] 4.1.46 #2 Not tainted
[  460.427276] -------------------------------
[  460.427864] include/linux/rcupdate.h:570 Illegal context switch in RCU read-side critical
section!
[  460.429056]
[  460.429056] other info that might help us debug this:
[  460.429056]
[  460.430726]
[  460.430726] rcu_scheduler_active = 1, debug_locks = 0
[  460.432040] 3 locks held by sh/130:
[  460.432717]  #0:  (sb_writers#4){.+.+.+}, at: [<800fb398>] vfs_write+0x140/0x15c
[  460.437331]  #1:  (rcu_read_lock){......}, at: [<8031c664>] __handle_sysrq+0x0/0x254
[  460.438777]  #2:  (&mm->mmap_sem){++++++}, at: [<8001ecb0>] do_page_fault+0x78/0x37c
[  460.440515]
[  460.440515] stack backtrace:    << back to the crash >>
[  460.441491] CPU: 0 PID: 130 Comm: sh Not tainted 4.1.46 #2
[  460.442026] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[  460.443113] Backtrace:
[  460.443772] [<80013330>] (dump_backtrace) from [<80013544>] (show_stack+0x18/0x1c)

[...]

[  460.476990] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[  460.477819] pgd = e71c8000
[  460.478133] [00000000] *pgd=771ad831, *pte=00000000, *ppte=00000000
[  460.479304] Internal error: Oops: 817 [#1] SMP ARM
[  460.480044] Modules linked in:
```

```
[  460.481368] CPU: 0 PID: 130 Comm: sh Not tainted 4.1.46 #2
[  460.481945] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[  460.482577] task: e7abd000 ti: e728e000 task.ti: e728e000
[  460.483090] PC is at sysrq_handle_crash+0x3c/0x4c
[  460.483490] LR is at sysrq_handle_crash+0x34/0x4c
[  460.483850] pc : [<8031bd80>]    lr : [<8031bd78>]    psr: a0000013
[  460.483850] sp : e728fe50  ip : e728fe50  fp : e728fe5c
[  460.484480] r10: 00000000  r9 : 00000000  r8 : 00000007
[  460.484849] r7 : 00000000  r6 : 00000063  r5 : 80a4d5e8  r4 : 80a61694
[  460.485287] r3 : 00000000  r2 : 00000001  r1 : f0004000  r0 : 00000063
[  460.485741] Flags: NzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
[  460.486446] Control: 10c5387d  Table: 771c8059  DAC: 00000015
[  460.486846] Process sh (pid: 130, stack limit = 0xe728e210)
[  460.487240] Stack: (0xe728fe50 to 0xe7290000)
[  460.487605] fe40:                                     e728fe94 e728fe60 8031c744 8031bd50

[...]

[  460.492794] Backtrace:
[  460.493127] [<8031bd44>] (sysrq_handle_crash) from [<8031c744>] (__handle_sysrq+0xe0/0x254)
[  460.493599] [<8031c664>] (__handle_sysrq) from [<8031cd14>] (write_sysrq_trigger+0x50/0x60)
[  460.493982]  r8:00000000 r7:e7bf5c00 r6:00000000 r5:00ab6400 r4:00000002
[  460.494647] [<8031ccc4>] (write_sysrq_trigger) from [<8015815c>] (proc_reg_write+0x68/0x90)
[  460.495126]  r5:00000001 r4:00000000
[  460.495537] [<801580f4>] (proc_reg_write) from [<800faa4c>] (__vfs_write+0x2c/0xe0)
[  460.495945]  r9:00ab6400 r8:00000002 r7:e728ff78 r6:e71aa8c0 r5:00ab6400 r4:80766b40
[  460.496617] [<800faa20>] (__vfs_write) from [<800fb2f4>] (vfs_write+0x9c/0x15c)
[  460.497292]  r8:00000002 r7:00000002 r6:e728ff78 r5:00ab6400 r4:e71aa8c0
[  460.498014] [<800fb258>] (vfs_write) from [<800fbb24>] (SyS_write+0x44/0x98)
[  460.498402]  r9:00ab6400 r8:00000002 r7:e71aa8c0 r6:e71aa8c0 r5:00000000 r4:00000000
[  460.499154] [<800fbae0>] (SyS_write) from [<8000f960>] (ret_fast_syscall+0x0/0x54)
[  460.499635]  r9:e728e000 r8:8000fb44 r7:00000004 r6:00ab6400 r5:00000001 r4:000f8e2c
[  460.500640] Code: 0a000000 e12fff33 e3a03000 e3a02001 (e5c32000)
[  460.503826] Loading crashdump kernel...
[  460.504424] Bye!
[    0.000000] Booting Linux on physical CPU 0x0  << the dump kernel starts! >>
[    0.000000] Linux version 4.1.46 (kai@klaptop) (gcc version 4.8.3 20140320 (prerelease)
(Sourcery CodeBench Lite 2014.05-29) ) #2 SMP Mon Nov 27 17:16:22 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 model: Freescale i.MX6 DualLite SABRE Smart Device Board
[    0.000000] Ignoring memory block 0x10000000 - 0x50000000
[    0.000000] cma: Reserved 16 MiB at 0x7ec00000
[    0.000000] Memory policy: Data cache writeback
[    0.000000] CPU: All CPU(s) started in SVC mode.
[    0.000000] PERCPU: Embedded 12 pages/cpu @87cb9000 s16640 r8192 d24320 u49152
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 32260
[    0.000000] Kernel command line: console=ttymxc0 root=/dev/mmcblk0 rootfstype=ext4 rootwait
init=/sbin/init maxcpus=1 reset_devices elfcorehdr=0x7ff00000 mem=130048K
[    0.000000] PID hash table entries: 512 (order: -1, 2048 bytes)
...
[    0.000000] Virtual kernel memory layout:
[    0.000000]     vector  : 0xffff0000 - 0xffff1000   (   4 kB)
[    0.000000]     fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
[    0.000000]     vmalloc : 0x88000000 - 0xff000000   (1904 MB)
[    0.000000]     lowmem  : 0x80000000 - 0x87f00000   ( 127 MB)
[    0.000000]     pkmap   : 0x7fe00000 - 0x80000000   (   2 MB)
[    0.000000]     modules : 0x7f000000 - 0x7fe00000   (  14 MB)
[    0.000000]       .text : 0x80008000 - 0x809d7fdc   (10048 kB)
[    0.000000]       .init : 0x809d8000 - 0x80a3a000   ( 392 kB)
[    0.000000]       .data : 0x80a3a000 - 0x80a9a6e0   ( 386 kB)
[    0.000000]        .bss : 0x80a9a6e0 - 0x812c3164   (8355 kB)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=2, Nodes=1

[...]

[    8.142812] fec 2188000.ethernet eth0: Link is Up - 100Mbps/Full - flow control rx/tx
```

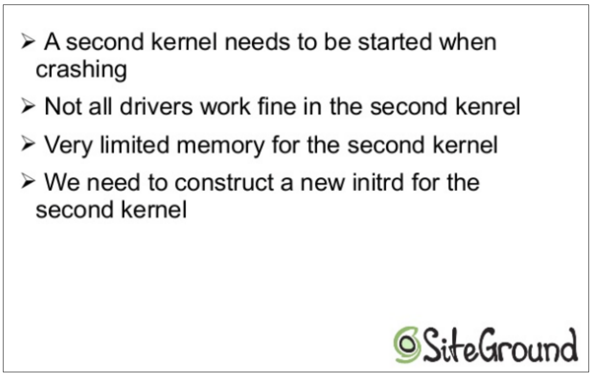
```
[    8.145506] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[   10.488149] cfg80211: Calling CRDA to update world regulatory domain

ARM / $ ls -lh /proc/vmcore
-r--------    1 0        0           1.9G Jan  1 00:07 /proc/vmcore
ARM / $ cp /proc/vmcore /kdump.img   << or transfer via scp >>
ARM / $
```

(Large dumpfile because the RAM on this system is 2 GB).

---

kdump Downsides



+ a significant amount of RAM HAS to be set aside, though, most of the time it remains unused.

---

makedumpfile

Makes a small dumpfile from the panicked kernel's /proc/vmcore image (which can be very large).

From it's man page:

Description
With kdump, the memory image of the first kernel (called "panicked kernel") can be taken as /proc/vmcore while the second kernel (called "kdump kernel" or "capture kernel") is running. This document represents /proc/vmcore as VMCORE. makedumpfile makes a small DUMPFILE by compressing dump data or by excluding unnecessary pages for analysis, or both. makedumpfile needs the first kernel's debug information, so that it can distinguish unnecessary pages by analyzing how the first kernel uses the memory. The information can be taken from VMLINUX or VMCOREINFO.

makedumpfile can exclude the following types of pages while copying VMCORE to DUMPFILE, and a user can choose which type of pages will be excluded.
- Pages filled with zero
- Cache pages
- User process data pages
- Free pages

makedumpfile provides two DUMPFILE formats (the ELF format and the kdump-compressed format). By default, makedumpfile makes a DUMPFILE in the kdump-compressed format. The kdump-compressed format is readable only with the crash utility, and it can be smaller than the ELF format because of the compression support. The ELF format is readable with GDB and the crash utility. If a user wants to use

GDB, DUMPFILE format has to be explicitly specified to be the ELF format.
...
...

Related:
https://github.com/makedumpfile/makedumpfile

Linux crash utility

Resources

Crash Whitepaper : superb!
https://crash-utility.github.io/crash_whitepaper.html

crash page with overrall links

Analyzing Linux kernel crash dumps with crash – Dedoimedo

‘crash’ source repo on GitHub – do read the README here

RedHat
How to troubleshoot kernel crashes, hangs, or reboots with kdump on Red Hat Enterprise Linux
RHEL 7 : Chapter 7. Kernel crash dump guide

http://docs.oracle.com/cd/E37670_01/E41138/html/ch10s02.html

From the README
“...
At this point, x86, ia64, x86_64, ppc64, ppc, arm, arm64, alpha, mips,
s390 and s390x-based kernels are supported.
...
o One size fits all -- the utility can be run on any Linux kernel version
version dating back to 2.2.5-15. A primary design goal is to always
maintain backwards-compatibility.

o In order to contain debugging data, the top-level kernel Makefile's CFLAGS
definition must contain the -g flag. Typically distributions will contain
a package containing a vmlinux file with full debuginfo data. If not, the
kernel must be rebuilt

...

The crash binary can only be used on systems of the same architecture as
the host build system. There are a few optional manners of building the
crash binary:

- o On an x86_64 host, a 32-bit x86 binary that can be used to analyze 32-bit x86 dumpfiles may be built by typing "make target=X86".
- o On an x86 or x86_64 host, a 32-bit x86 binary that can be used to analyze 32-bit arm dumpfiles may be built by typing "make target=ARM".
- o On an x86 or x86_64 host, a 32-bit x86 binary that can be used to analyze 32-bit mips dumpfiles may be built by typing "make target=MIPS".
- o On an ppc64 host, a 32-bit ppc binary that can be used to analyze 32-bit ppc dumpfiles may be built by typing "make target=PPC".
- o On an x86_64 host, an x86_64 binary that can be used to analyze arm64 dumpfiles may be built by typing "make target=ARM64".

...”

Investigate a Live Linux system with crash

Required: the vmlinux kernel image built with debug symbolic information.

Getting the kernel vmlinux with debug symbolic info

For Ubuntu
Where can one get the Ubuntu linux debug kernel vmlinux from? See: https://wiki.ubuntu.com/Kernel/Systemtap#Where_to_get_debug_symbols_for_kernel_X.3F Short answer: here: http://ddebs.ubuntu.com/pool/main/l/linux/ Download the ***linux-image-$(uname -r)-dbgsym_$(uname -r)_amd64.ddeb*** file (assuming you're running on an x86_64 system). Eg. Required file for kernel ver 3.16.0-37-generic is linux-image-3.16.0-37-generic-dbgsym_3.16.0-37.49_amd64.ddeb Extract the *vmlinux-$(uname -r)* image from the downloaded ddeb file. ***Unresolved:*** recent Ubuntu x86_64 (tried with 16.10, 17.04 and 17.10), there seems to be an issue running crash with the Ubuntu debugsym vmlinux (it *does* work well with older Ubuntu distros; tried with 14.04 LTS and it’s fine). ?? *<< Update: does work on Ubuntu 18.04.2 and 20.04 LTS >>*
For Fedora
Kernel-debug repo: https://www.rpmfind.net/linux/rpm2html/search.php?query=kernel-debug How to use kdump to debug kernel crashes https://fedoraproject.org/wiki/How_to_use_kdump_to_debug_kernel_crashes

Essentially, need to do this to get the latest debug kernel:
`sudo dnf install kernel-debug-<tab><tab>`

(Observation: it's not always *the* latest kernel installed, so debugging the "live" way can become an issue).

Two broad ways to run crash:

- Running with a **kdump image** (obtained via Kdump/kexec facility upon kernel crash/Oops/panic):

  ```
  sudo crash <vmlinux-with-symbolic-info> <kdump-image> [corr System.map]
  ```

- Running with the '**live**' kernel image:

  ```
  sudo crash <vmlinux-with-symbolic-info> /proc/kcore [corr System.map]
  ```

Note:
The */proc/kcore* pseudo-file – the kernel memory snapshot – becomes visible when CONFIG_PROC_KCORE=y.

For a live debug session, the `<vmlinux-with-symbolic-info>` must precisely match the currently running kernel version.

```
$ sudo crash
...
```

If that doesn't work, try passing parameters explicitly; for example, below with a 5.10.153 'debug' kernel on an Ubuntu 22.04 LTS x86_64 guest:

```
$ sudo crash ~/linux-5.10.153/vmlinux /proc/kcore

crash 8.0.0
Copyright (C) 2002-2021  Red Hat, Inc.
Copyright (C) 2004, 2005, 2006, 2010  IBM Corporation
Copyright (C) 1999-2006  Hewlett-Packard Co
Copyright (C) 2005, 2006, 2011, 2012  Fujitsu Limited
Copyright (C) 2006, 2007  VA Linux Systems Japan K.K.
Copyright (C) 2005, 2011, 2020-2021  NEC Corporation
Copyright (C) 1999, 2002, 2007  Silicon Graphics, Inc.
Copyright (C) 1999, 2000, 2001, 2002  Mission Critical Linux, Inc.
Copyright (C) 2015, 2021  VMware, Inc.
This program is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it under
certain conditions.  Enter "help copying" to see the conditions.
This program has absolutely no warranty.  Enter "help warranty" for details.

GNU gdb (GDB) 10.2
Copyright (C) 2021 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-pc-linux-gnu".
Type "show configuration" for configuration details.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...

      KERNEL: /home/osboxes/linux-5.10.153/vmlinux  [TAINTED]
    DUMPFILE: /proc/kcore
        CPUS: 6
        DATE: Tue Jun 13 16:08:03 IST 2023
      UPTIME: 00:07:50
LOAD AVERAGE: 0.25, 0.13, 0.08
       TASKS: 551
    NODENAME: osboxes
     RELEASE: 5.10.153-kdbg1
     VERSION: #1 SMP Tue Jun 13 15:51:43 IST 2023
     MACHINE: x86_64  (2592 Mhz)
      MEMORY: 2 GB
         PID: 2539
     COMMAND: "crash"
        TASK: ffff9f11c40217c0  [THREAD_INFO: ffff9f11c40217c0]
         CPU: 4
       STATE: TASK_RUNNING (ACTIVE)

crash>
```

It works!
(Sometimes leaving out the last System.map parameter actually helps (?)).

If you get errors like this (this was on an Ubuntu 22.04 LTS running a custom 6.1.25 debug kernel (configured with Kexec/Kdump support)):

```
...
Type "apropos word" to search for commands related to "word"...

WARNING: kernel version inconsistency between vmlinux and live memory


crash: invalid structure member offset: kmem_cache_s_num
       FILE: memory.c  LINE: 9619  FUNCTION: kmem_cache_init()

[/usr/bin/crash] error trace: 557782d9969e => 557782d6d2f4 => 557782e3b11b =>
557782e3b09c
```

I find that it’s hard to resolve... Instead, carefully and correctly reconfiguring and rebuilding the target kernel is the approach that has it then work (also ensure that besides kdump, kernel debug info is on).

<<
Tip
Particularly for the above failure, *see this link*; there is indeed an issue within crash, which has been fixed in later releases of the app. I built crash ver 8.0.3 (above was 8.0.0), ran it and it worked!.
(*crash source available here*).
>>

Crash Commands - Quick Notes

The tool's environment is context-specific. On a live system, the default context is the command itself; on a dump the default context will be the task that panicked [can be changed with the ‘set’ command].

Structures

Do what	How / Command	Example
See structure definition	Give name of structure or use 'whatis' <struct_name>	crash> file crash> whatis file
See structure runtime contents	[*] <struct_name> <ptr>	crash> * file ffff88018c320d00
See particular member(s) of a struct	<struct_name> | grep <member-name>	crash> task | grep uid uid = 3369, euid = 3369, suid = 3369, fsuid = 3369,

Set the (process) context
set <PID> ...
Set a new task context by PID, task address, or cpu. Since several `crash` commands are context-sensitive, it's helpful to be able to change the context to avoid having to pass the PID or task address to those context-sensitive commands in order to access the data of a task that is *not* the current context.

Open Files

Do what	How / Command	Example
See all open files for process context	files	crash> files

Useful Commands

log : view kernel printks

bt : get a kernel stack backtrace of the current context
bt -a : kernel stack trace of the active task(s) when the kernel panicked
(*help*: https://crash-utility.github.io/help_pages/bt.html)

task : task structure of the current context

files : see open file information of the current context

vm : see virtual memory information of the current context

vtop : translate virtual to physical address

kmem : kernel memory subsystems

To see number to different bases, use the '**eval**' command.
Eg.

```
crash> eval 140737488351232
hexadecimal: 7ffffffff000  (137438953468KB)
    decimal: 140737488351232
      octal: 3777777777770000
     binary: 0000000000000000011111111111111111111111111111111000000000000
crash>
```

net : networking information

runq : runqueue information

Quick summary of common 'crash' commands [*src*]

- bt
    - bt -c 17: only task at cpu-17
    - bt -c 16-31: cpu 16 ~ 31
- foreach bt: backtrace of all processes
- log: kernel log
- ps
- disassemble
    - disassemble /r : print opcode and instruction
- rd : read memory from address
- gdb list *(memcpy+16): find c-code line
- mod: print modules
    - no information for module loaded: ffffffffa0316fa0 kvm 342174 (not loaded) [CONFIG_KALLSYMS]
    - mod -s: load modules
      -s module [objfile] Loads symbolic and debugging data from the object file for the module specified. ...
    - There not only vmlinux but also modules should be copied

Common Commands

bt	Display the backtrace of the current context, or as specified with arguments. This command is typically the first command entered after starting a dumpfile session. Since the initial context is the panic context, it will show the function trace leading up to the kernel panic. `bt -a` will

	show the trace of the *active* task on each CPU, since there may be an interrelationship between the panicking task on one CPU and the running task(s) on the other CPU(s). When `bt` is given as the argument to `foreach`. displays the backtraces of *all* tasks. *<< bt **-l** : show file and line number of each stack trace text location. >>*

`struct`	Print the contents of a data structure at a specified address. This command is so common that it is typically unnecessary to enter the `struct` command name on the command line; if the first command line argument is not a `crash` or `gdb` command, but it *is* the name of a known data structure, then all the command line arguments are passed to the `struct` command. So for example, the following two commands yield the same result:
	
	`crash> struct vm_area_struct d3cb2600`
	
	`crash> vm_area_struct d3cb2600`

`set`	Set a new task context by PID, task address, or cpu. Since several `crash` commands are context-sensitive, it's helpful to be able to change the context to avoid having to pass the PID or task address to those context-sensitive commands in order to access the data of a task that is *not* the current context.
`p`	Prints the contents of a kernel variable; since it's a gateway to the `print` command of the mbedded `gdb` module, it can also be used to print complex C language expressions.
`rd`	Read memory, which may be either kernel virtual, user virtual, or physical, and display it several different formats and sizes.
`ps`	Lists basic task information for each process; it can also display parent and child hierarchies.
`log`	Dump the kernel `log_buf`, which often contains clues leading up to a subsequent kernel crash.
`foreach`	Execute a `crash` command on all tasks, or those specified, in the system; can be used with `bt`, `vm`, `task`, `files`, `net`, `set`, `sig` and `vtop`.
`files`	Dump the open file descriptor data of a task; most usefully, the `file`, `dentry` and `inode` structure addresses for each open file descriptor.

vm	Dump the virtual memory map of a task, including the vital information concerning each `vm_area_struct` making up a task's address space. It can also dump the physical address of each page in the address space, or if not mapped, its location in a file or on the swap device.

whatis <struct-or-symbol-name> : displays the structure members

Eg.
<< on Ubuntu 15.04 kernel ver 3.19.0-22-generic >>

```
crash> whatis thread_info
struct thread_info {
    struct task_struct *task;
    struct exec_domain *exec_domain;
    __u32 flags;
    __u32 status;
    __u32 cpu;
    int saved_preempt_count;
    mm_segment_t addr_limit;
    struct restart_block restart_block;
    void *sysenter_return;
    unsigned int sig_on_uaccess_error : 1;
    unsigned int uaccess_err : 1;
}
SIZE: 104
crash> struct task_struct {
    volatile long state;
    void *stack;
    atomic_t usage;
...
  unsigned int sequential_io_avg;
}
SIZE: 2504
crash>
```

Disassembly:

```
crash> dis printk
0xffffffff817c1bed <printk>:    data32 data32 data32 xchg %ax,%ax [FTRACE NOP]
0xffffffff817c1bf2 <printk+5>:  push   %rbp
0xffffffff817c1bf3 <printk+6>:  mov    %rsp,%rbp
0xffffffff817c1bf6 <printk+9>:  sub    $0x50,%rsp
...
```

Disassemble 20 instructions of tcp_sendmsg, showing source line numbers (-l option) and change radix to hex (-x):

```
crash> dis -l tcp_sendmsg 20 -x
/build/buildd/linux-3.19.0/net/ipv4/tcp.c: 1069
0xffffffff8170bf80 <tcp_sendmsg>:       data32 data32 data32 xchg %ax,%ax [FTRACE NOP]
0xffffffff8170bf85 <tcp_sendmsg+0x5>:   push   %rbp
0xffffffff8170bf86 <tcp_sendmsg+0x6>:   mov    %rsp,%rbp
0xffffffff8170bf89 <tcp_sendmsg+0x9>:   push   %r15
0xffffffff8170bf8b <tcp_sendmsg+0xb>:   push   %r14
```

```
0xffffffff8170bf8d <tcp_sendmsg+0xd>:   push   %r13
0xffffffff8170bf8f <tcp_sendmsg+0xf>:   push   %r12
0xffffffff8170bf91 <tcp_sendmsg+0x11>:  mov    %rsi,%r15
0xffffffff8170bf94 <tcp_sendmsg+0x14>:  push   %rbx
/build/buildd/linux-3.19.0/include/net/sock.h: 1536
0xffffffff8170bf95 <tcp_sendmsg+0x15>:  xor    %esi,%esi
/build/buildd/linux-3.19.0/net/ipv4/tcp.c: 1069
0xffffffff8170bf97 <tcp_sendmsg+0x17>:  mov    %rdx,%rbx
/build/buildd/linux-3.19.0/include/net/sock.h: 1536
0xffffffff8170bf9a <tcp_sendmsg+0x1a>:  mov    %r15,%rdi
/build/buildd/linux-3.19.0/net/ipv4/tcp.c: 1069
0xffffffff8170bf9d <tcp_sendmsg+0x1d>:  mov    %rcx,%r12
0xffffffff8170bfa0 <tcp_sendmsg+0x20>:  sub    $0x98,%rsp
...

<< Disassemble help:
...
With a /s modifier, source lines are included (if available).
In this mode, the output is displayed in PC address order, and
file names and contents for all relevant source files are displayed.

With a /m modifier, source lines are included (if available).
This view is "source centric": the output is in source line order, ...
>>

crash> disassemble /s tty_read
Dump of assembler code for function tty_read:
drivers/tty/tty_io.c:
916     {
   0xffffffffbbfa9d20 <+0>:     nopl   0x0(%rax,%rax,1)
   0xffffffffbbfa9d25 <+5>:     push   %rbp

919             struct inode *inode = file_inode(file);
920             struct tty_struct *tty = file_tty(file);
921             struct tty_ldisc *ld;
922
923             if (tty_paranoia_check(tty, inode, "tty_read"))
   0xffffffffbbfa9d26 <+6>:     mov    $0xffffffffbce0315f,%rdx

916     {
   0xffffffffbbfa9d2d <+13>:    mov    %rsp,%rbp
   0xffffffffbbfa9d30 <+16>:    push   %r15
...
```

Example- Figuring out how stdin/stdout/stderr actually work!

Ok, which process is in context now?

```
crash> set
    PID: 3835
COMMAND: "crash"
   TASK: ffff93a44402b000  [THREAD_INFO: ffff93a44402b000]
    CPU: 2
  STATE: TASK_RUNNING (ACTIVE)
```

Let’s see what files the *crash* process has open:

```
crash> files
PID: 3835   TASK: ffff93a44402b000  CPU: 2   COMMAND: "crash"
ROOT: /    CWD: /home/osboxes
 FD       FILE            DENTRY           INODE       TYPE PATH
  0 ffff93a46d985500 ffff93a4628d0900 ffff93a494150e40 CHR  /dev/pts/4
  1 ffff93a46d985500 ffff93a4628d0900 ffff93a494150e40 CHR  /dev/pts/4
  2 ffff93a46d985500 ffff93a4628d0900 ffff93a494150e40 CHR  /dev/pts/4
  3 ffff93a4832a7900 ffff93a4414433c0 ffff93a441c2a500 CHR  /dev/null
  4 ffff93a444b72800 ffff93a4645d5780 ffff93a47c56bd60 REG  /proc/kcore
  5 ffff93a444b72300 ffff93a47c6d6d80 ffff93a4a6067040 REG  /home/osboxes/linux-
5.10.153/vmlinux
  6 ffff93a4a1aec600 ffff93a4a602bc00 ffff93a47c765300 FIFO
  7 ffff93a4a1aece00 ffff93a4a602bc00 ffff93a47c765300 FIFO
  8 ffff93a4a1aec400 ffff93a4a602bcc0 ffff93a47c764260 FIFO
  9 ffff93a4a1aecb00 ffff93a4a602bcc0 ffff93a47c764260 FIFO
 10 ffff93a4a1aecf00 ffff93a4a602bb40 ffff93a47c766600 FIFO
 11 ffff93a4a1aec900 ffff93a4a602bb40 ffff93a47c766600 FIFO
 12 ffff93a4a1aec200 ffff93a4a602b000 ffff93a47c764be0 FIFO
 13 ffff93a4a1aecc00 ffff93a4a602b000 ffff93a47c764be0 FIFO
 14 ffff93a4a1aec500 ffff93a47c7b8e40 ffff93a4bd306700 REG  /tmp/#4849686
 16 ffff93a46ebf6700 ffff93a450c6d6c0 ffff93a450f19ee0 FIFO
crash>
```

We understand that file descriptors 0,1,2 represent stdin,stdout and stderr of the process context (here, it's the *crash* process itself) respectively... So, look up their open file structure (notice it's the same for all three):

```
crash> file ffff93a46d985500
struct file {
  f_u = {
    fu_llist = {
      next = 0x0
    },
    fu_rcuhead = {
      next = 0x0,
      func = 0x0
    }
  },
  f_path = {
    mnt = 0xffff93a442d7c520,
    dentry = 0xffff93a4628d0900
  },
  f_inode = 0xffff93a494150e40,
  f_op = 0xffffffffbcac1e00 <tty_fops>,
  f_lock = {
    {
      rlock = {
...
```

A big structure... grep for the *fops*, the *file_operations* structure:

```
crash> file ffff93a46d985500 | grep f_op
 f_op = 0xffffffffbcac1e00 <tty_fops>,
```

We can see the *fops*, the *file_operations* structure! It holds the key to how the 'device' or 'file' works...So let's peek into it; but we need to translate the kernel virtual address (kva) to a symbolic name:

```
crash> sym 0xffffffffbcac1e00
ffffffffbcac1e00 (d) tty_fops
```

Cool, it's the *tty_fops* structure; lets look it up:

```
crash> tty_fops
tty_fops = $3 = {
  owner = 0x0,
  llseek = 0xffffffffbbb61dc0 <no_llseek>,
  read = 0x0,
  write = 0x0,
  read_iter = 0xffffffffbbfa9d20 <tty_read>,
  write_iter = 0xffffffffbbfaa6f0 <tty_write>,
  iopoll = 0x0,
  iterate = 0x0,
  iterate_shared = 0x0,
  poll = 0xffffffffbbfac290 <tty_poll>,
  unlocked_ioctl = 0xffffffffbbfaabc0 <tty_ioctl>,
  compat_ioctl = 0xffffffffbbfab4f0 <tty_compat_ioctl>,
  mmap = 0x0,
  mmap_supported_flags = 0,
  open = 0xffffffffbbfad5f0 <tty_open>,
  flush = 0x0,
  release = 0xffffffffbbfabca0 <tty_release>,
  fsync = 0x0,
  ...
  may_pollfree = false
}
```

Wow; we got the read/write '_iter' routines that perform I/O on the TTY device! which show as the tty_read(), tty_write() !
Let's disassemble, for exampe, the code that runs when userspace writes (via the write() syscall) to the tty device, i.e., the *tty_write()* function within the kernel (/s => source lines are shown, if possible (if `-g` used when compiling)):

```
crash> disassemble /s  tty_write
Dump of assembler code for function tty_write:
drivers/tty/tty_io.c:
1124    {
   0xffffffffbbfaa6f0 <+0>:     nopl   0x0(%rax,%rax,1)

1125            return file_tty_write(iocb->ki_filp, iocb, from);
   0xffffffffbbfaa6f5 <+5>:     push   %rbp
...
```

Ah, so it's a wrapper over file_tty_write():

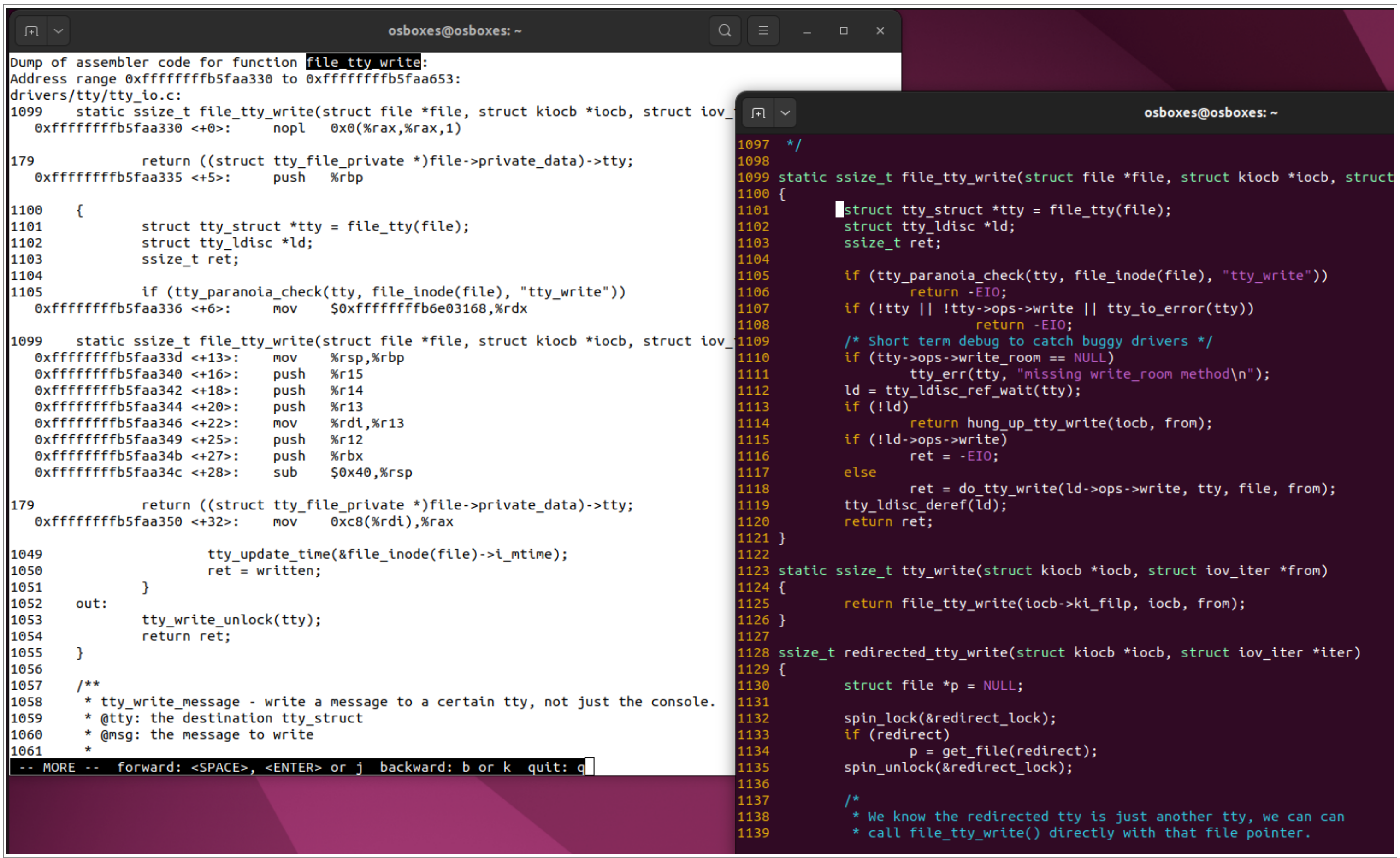
```
crash> disassemble /s file_tty_write
Dump of assembler code for function file_tty_write:
Address range 0xffffffffb5faa330 to 0xffffffffb5faa653:
drivers/tty/tty_io.c:
1099    static ssize_t file_tty_write(struct file *file, struct kiocb *iocb, struct
iov_iter *from)
   0xffffffffb5faa330 <+0>:     nopl   0x0(%rax,%rax,1)
179             return ((struct tty_file_private *)file->private_data)->tty;
   0xffffffffb5faa335 <+5>:     push   %rbp

1100    {
```

```
1101            struct tty_struct *tty = file_tty(file);
1102            struct tty_ldisc *ld;
1103            ssize_t ret;
1104
1105            if (tty_paranoia_check(tty, file_inode(file), "tty_write"))
   0xffffffffb5faa336 <+6>:     mov    $0xffffffffb6e03168,%rdx

1099    static ssize_t file_tty_write(struct file *file, struct kiocb *iocb, struct 
iov_iter *from)
   0xffffffffb5faa33d <+13>:    mov    %rsp,%rbp
   0xffffffffb5faa340 <+16>:    push   %r15
   0xffffffffb5faa342 <+18>:    push   %r14
   0xffffffffb5faa344 <+20>:    push   %r13
   0xffffffffb5faa346 <+22>:    mov    %rdi,%r13
   0xffffffffb5faa349 <+25>:    push   %r12
...
```

A screenshot- the crash app with the disassembled code on the left, the very same source code on the right:



Running crash in "batch mode"

Very useful technique to grab "just enough" information from a dumpfile and save it.

1. Create a crash "commands script" file; for example:

```
$ cat crash_getinfo
echo "=== System Info ==="
echo "--- sys ---"
sys

echo "--- log ---"
log

echo "--- ps ---"
ps

echo "--- dev ---"
dev

echo "=== Current Context Info ==="
echo "--- bt -a ---"
bt -a

echo "--- files ---"
files

echo "--- vm ---"
vm

exit
$
```

2. Invoke it via crash:

```
sudo crash vmlinux_dbgsym.img {dumpfile.img -or- /proc/kcore} < crash_getinfo >
report.txt
```

We now have a report!

---

sym (symbol)

[Running crash on a dump image from an ARM-32; see how to here]

```
...
crash_32bit_for_arm> help sym
NAME
  sym - translate a symbol to its virtual address, or vice-versa

SYNOPSIS
  sym [-l] | [-M] | [-m module] | [-p|-n] | [-q string] | [symbol | vaddr]

DESCRIPTION
  This command translates a symbol to its virtual address, or a static
  kernel virtual address to its symbol -- or to a symbol-plus-offset value,
  if appropriate.  Additionally, the symbol type is shown in parentheses,
```

```
  and if the symbol is a known text value, the file and line number are shown.

...
<lots of examples!>

crash_32bit_for_arm> bt
PID: 735    TASK: 9f6af900  CPU: 0   COMMAND: "echo"
 #0 [<804060d8>] (sysrq_handle_crash) from [<804065bc>]
 #1 [<804065bc>] (__handle_sysrq) from [<80406ab8>]
 #2 [<80406ab8>] (write_sysrq_trigger) from [<80278588>]
 #3 [<80278588>] (proc_reg_write) from [<802235c4>]
 #4 [<802235c4>] (__vfs_write) from [<80224098>]
 #5 [<80224098>] (vfs_write) from [<80224d30>]
 #6 [<80224d30>] (sys_write) from [<801074a0>]
    pc : [<76e8d7ec>]    lr : [<0000f9dc>]    psr: 60000010
    sp : 7ebdcc7c  ip : 00000000  fp : 00000000
    r10: 0010286c  r9 : 7ebdce68  r8 : 00000020
    r7 : 00000004  r6 : 00103008  r5 : 00000001  r4 : 00102e2c
    r3 : 00000000  r2 : 00000002  r1 : 00103008  r0 : 00000001
    Flags: nZCv  IRQs on  FIQs on  Mode USER_32  ISA ARM

crash_32bit_for_arm> sym 801074a0
801074a0 (t) ret_fast_syscall ../arch/arm/kernel/entry-common.S
crash_32bit_for_arm> bt -l             << -l: show file and line number info >>
PID: 735    TASK: 9f6af900  CPU: 0   COMMAND: "echo"
 #0 [<804060d8>] (sysrq_handle_crash) from [<804065bc>]
    <...>/linux-4.9.1/drivers/tty/sysrq.c: 144
 #1 [<804065bc>] (__handle_sysrq) from [<80406ab8>]
    <...>/linux-4.9.1/drivers/tty/sysrq.c: 552
 #2 [<80406ab8>] (write_sysrq_trigger) from [<80278588>]
    <...>/linux-4.9.1/drivers/tty/sysrq.c: 1101
 #3 [<80278588>] (proc_reg_write) from [<802235c4>]
    <...>/linux-4.9.1/fs/proc/inode.c: 216
 #4 [<802235c4>] (__vfs_write) from [<80224098>]
    <...>/linux-4.9.1/fs/read_write.c: 510
 #5 [<80224098>] (vfs_write) from [<80224d30>]
    <...>/linux-4.9.1/fs/read_write.c: 561
 #6 [<80224d30>] (sys_write) from [<801074a0>]
    <...>/linux-4.9.1/fs/read_write.c: 608
    pc : [<76e8d7ec>]    lr : [<0000f9dc>]    psr: 60000010
    sp : 7ebdcc7c  ip : 00000000  fp : 00000000
    r10: 0010286c  r9 : 7ebdce68  r8 : 00000020
    r7 : 00000004  r6 : 00103008  r5 : 00000001  r4 : 00102e2c
    r3 : 00000000  r2 : 00000002  r1 : 00103008  r0 : 00000001
    Flags: nZCv  IRQs on  FIQs on  Mode USER_32  ISA ARM
crash_32bit_for_arm>
```

Module Debugging with crash

Compile the kernel module with -g (edit it’s Makefile, specify

~~EXTRA_CFLAGS   += -DDEBUG -g~~ #older/deprecated style

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

crash : mod -S :
   Load the symbolic and debugging data of all modules
```

Appending the directory pathname (where your kernel module resides) will restrict the search to that folder...
Eg. on a simple misc character driver kernel module:

```
crash> mod -S <...>/miscmj_tst/
     MODULE       NAME                   SIZE  OBJECT FILE
ffffffffc0393480  pata_acpi             16384
/lib/modules/4.17.0/kernel/drivers/ata/pata_acpi.ko
ffffffffc039d500  i2c_piix4             24576
/lib/modules/4.17.0/kernel/drivers/i2c/busses/i2c-piix4.ko
ffffffffc03a9580  libahci               32768
/lib/modules/4.17.0/kernel/drivers/ata/libahci.ko

[...]

ffffffffc072a940  vboxsf                45056  /lib/modules/4.17.0/misc/vboxsf.ko
ffffffffc075c080  miscmj_tst            16384
/home/seawolf/kaiwanTECH/L3_dd_trg/miscmj_tst/miscmj_tst.o
crash>

Set ctx to the process accessing the driver (it should be alive):
crash> set 4675
    PID: 4675
COMMAND: "echo"
   TASK: ffff9e7dbb865a00  [THREAD_INFO: ffff9e7dbb865a00]
    CPU: 1
  STATE: TASK_INTERRUPTIBLE

Now dump the stack frames - all show up!!

crash> bt
PID: 4675   TASK: ffff9e7dbb865a00  CPU: 1   COMMAND: "echo"
 #0 [ffffc24c01607cd0] __schedule at ffffffffa133731b
 #1 [ffffc24c01607d68] schedule at ffffffffa13378dc
 #2 [ffffc24c01607d78] schedule_timeout at ffffffffa133b69b
 #3 [ffffc24c01607df8] my_dev_write at ffffffffc075a0ff   [miscmj_tst]
 #4 [ffffc24c01607e18] __vfs_write at ffffffffa0c8bd3a
 #5 [ffffc24c01607ea0] vfs_write at ffffffffa0c8c011
 #6 [ffffc24c01607ed8] ksys_write at ffffffffa0c8c2a5
 #7 [ffffc24c01607f20] __x64_sys_write at ffffffffa0c8c32a
 #8 [ffffc24c01607f30] do_syscall_64 at ffffffffa0a041fa
 #9 [ffffc24c01607f50] entry_SYSCALL_64_after_hwframe at ffffffffa1400088
...

crash>

bt -f
...

 #3 [ffffc24c01607df8] my_dev_write at ffffffffc075a0ff [miscmj_tst]
    ffffc24c01607e00: 0000000000000000 ffff9e7db6d27900
    ffffc24c01607e10: ffffc24c01607e98 ffffffffa0c8bd3a
 #4 [ffffc24c01607e18] __vfs_write at ffffffffa0c8bd3a
    ffffc24c01607e20: ffff9e7db88ad330 ffffff0cdc0e22b70
    ffffc24c01607e30: 0000000000000000 0dabdbe391c9e100
    ffffc24c01607e40: 0000000000000055 ffff9e7db46c2708
    ffffc24c01607e50: 00005643ea666408 ffff9e7db6e49080
    ffffc24c01607e60: 0000000000000040 ffffc24c01607ea0
    ffffc24c01607e70: ffffffffa0c21fc3 0000000000000006
```

```
    ffffc24c01607e80: 0dabdbe391c9e100 0000000000000004
    ffffc24c01607e90: 0000000000000000 ffffc24c01607ed0
    ffffc24c01607ea0: ffffffffa0c8c011
 #5 [ffffc24c01607ea0] vfs_write at ffffffffa0c8c011
    ffffc24c01607ea8: ffff9e7db6d27900 ffff9e7db6d27900
    ffffc24c01607eb8: 00005643ea665400 0000000000000004
    ffffc24c01607ec8: 0000000000000000 ffffc24c01607f18
    ffffc24c01607ed8: ffffffffa0c8c2a5
...
```

Note-

- SETTING UP KDUMP AND CRASH FOR ARM-32 – AN ONGOING SAGA, kaiwanTECH, July 2017
- Running crash on ARM- see this link from the Linaro Wiki
- ***crash*** used quite a bit here: *A Short Guide to Kernel Debugging: A story about finding a kernel bug on a production system, Square*.

<< End document >>