Linux Kernel Debugging – Miscellaneous Tips and Tricks

First see:

Ubuntu Wiki: KernelDebuggingTricks

http://elinux.org/Debugging_by_printing

Kernel doc : dynamic debug: https://docs.kernel.org/admin-guide/dynamic-debug-howto.html

Bug Hunting
https://www.kernel.org/doc/Documentation/admin-guide/bug-hunting.rst

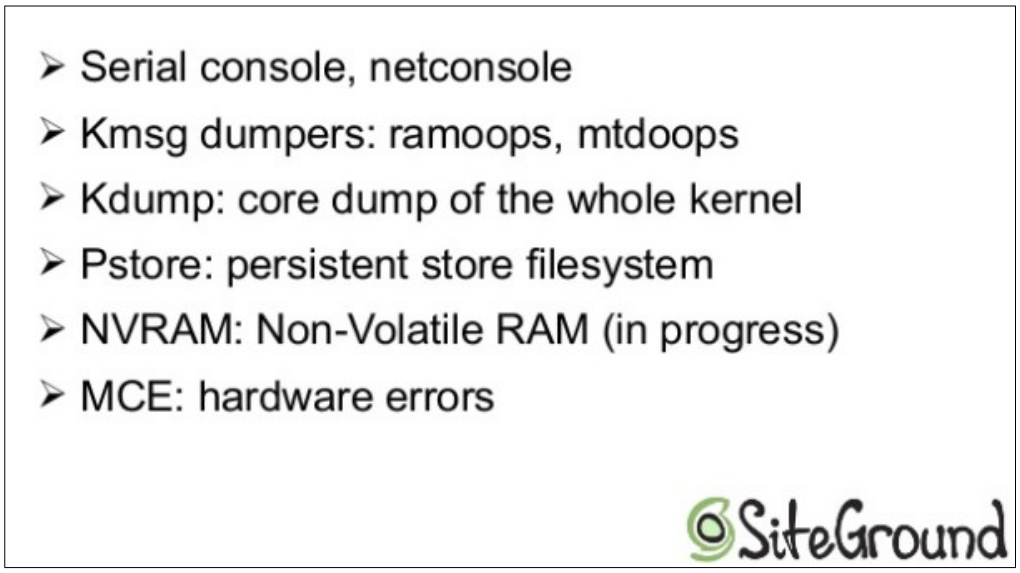
Storing crash data of the Linux kernel for post-crash debugging, Dec 2019

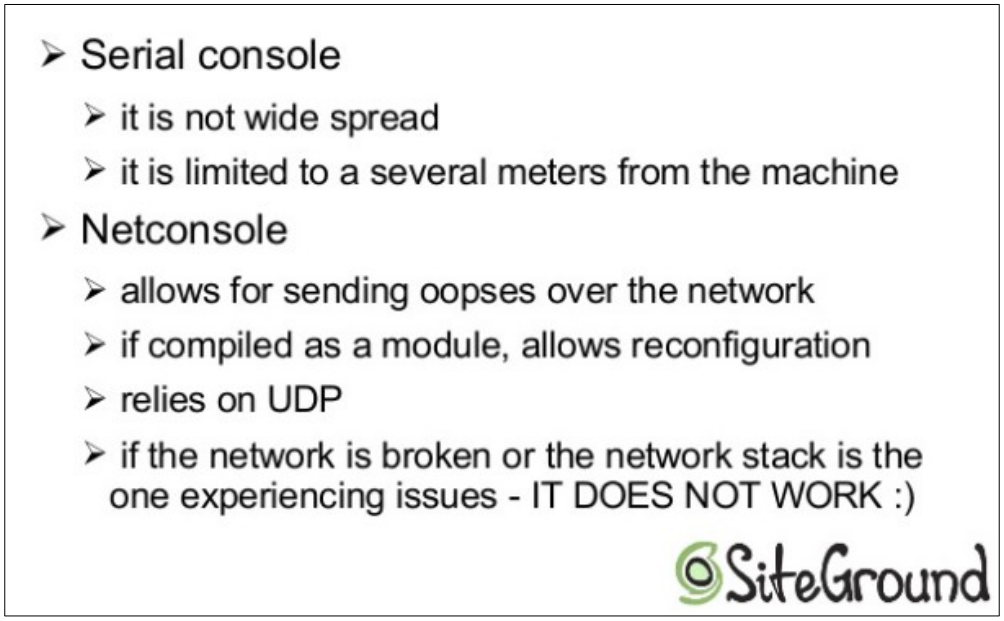
***Hunting and fixing bugs all over the Linux kernel, Gustavo AR Silva**, Kernel Recipes conference, Sept 2019 (LF CII): video slides*

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

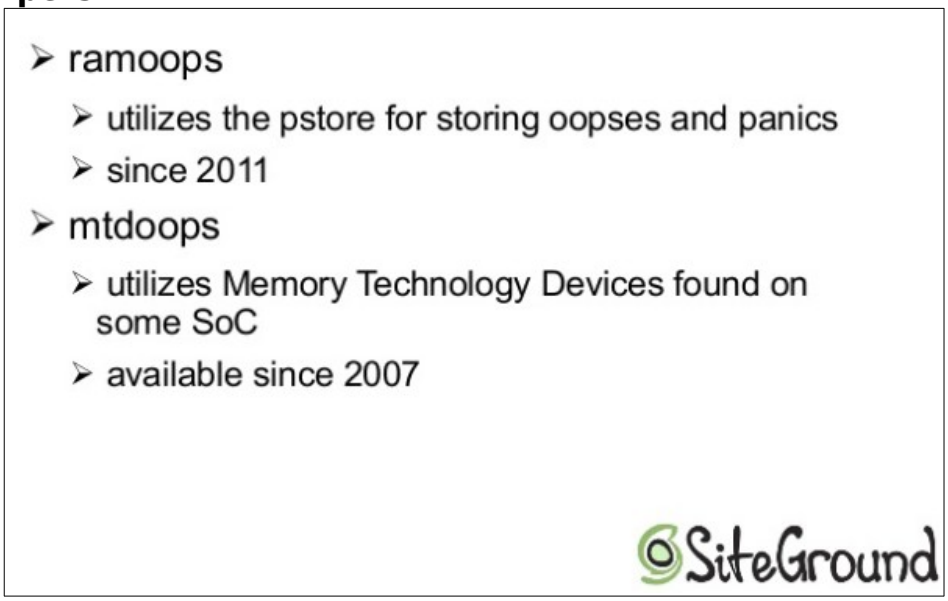
Source: https://www.slideshare.net/azilian/linux-kernel-crashdump

Ways to Gather Crash Data





Kmsg dumpers



Resources
- Kernel Documentation: https://www.kernel.org/doc/html/latest/admin-guide/ramoops.html
- State of Kernel Debugging, LinuxCon 2014

Using netconsole

netconsole: sends all kernel printk's over the network...

From **netconsole-setup(8)**:
Netconsole is a Linux kernel module that sends all kernel log messages over the network to another computer. It was designed to be as instantaneous as possible, to enable the logging of even the most critical kernel bugs. It works from IRQ contexts as well, and does not enable interrupts while sending packets. Due to these unique needs, only IP networks, UDP packets and Ethernet devices are supported. ...

Doc:
Official kernel doc: https://www.kernel.org/doc/Documentation/networking/netconsole.txt
Ubuntu Wiki - Netconsole
Linux Configure Netconsole To Log Messages Over UDP Network

Our intention: setup the
- sender system : an x86_64 desktop/laptop/server
- receiver system: a Raspberry Pi (it will receive and store the kernel log from the sender for later analysis)

You'll can setup the receiver with a **static IP** (on just the LAN for now is fine). See here.

***Sender system** (x86_64):*

Easiest to use **netconsole-setup(8)** (on sender system):
" netconsole-setup eases the configuration of the netconsole by just taking the target hostname or IP address. It determines all the needed configuration values from the running network device (like network interface, target IP and MAC address). Multiple target hosts can be specified. Each host can be prefixed with a target port and a plus sign to enable extended console support."

```
netconsole-setup [+port#@]<hostname/IP-addr>
```

You require to setup the sender with a **static IP** (on just the LAN for now is fine), so that the receiver can specify it in the netconsole modprobe ...

Run at boot; can use a systemd service or a cron job (via the @reboot) to guarantee running it upon bootup.

[If netconsole-setup isn't available, then you can manually setup the netconsole kernel module on the sender system; here's an example on the sender:

<<
netconsole module parameter format:
netconsole=[+][src-port]@[src-ip]/[<dev>],[tgt-port]@<tgt-ip>/[tgt-macaddr])

F.e.:
```
sudo modprobe netconsole \
   netconsole=@192.168.1.100/wlan0,@192.168.1.201/
```
>>

***Receiver system** (a Raspberry Pi):*

https://www.kernel.org/doc/Documentation/networking/netconsole.txt

```
...
Sender and receiver configuration:
==================================

It takes a string configuration parameter "netconsole" in the
following format:

 netconsole=[+][src-port]@[src-ip]/[<dev>],[tgt-port]@<tgt-ip>/[tgt-macaddr]

   where
        +             if present, enable extended console support
```

```
        src-port      source for UDP packets (defaults to 6665)
        src-ip        source IP to use (interface address)
        dev           network interface (eth0)
        tgt-port      port for logging agent (6666)
        tgt-ip        IP address for logging agent
        tgt-macaddr   ethernet MAC address for logging agent (broadcast)

Examples:

 linux netconsole=4444@10.0.0.1/eth1,9353@10.0.0.2/12:34:56:78:9a:bc

...
```

Test it:

```
dmesg -C; rmmod netconsole 2>/dev/null ; \
sudo modprobe netconsole netconsole=@192.168.1.101/wlan0,@192.168.1.200/ ;
                                  << src-ip    / dev ,  dest-ip/ >>
 dmesg|grep -v "Journal effective settings"
[ 1337.793067] netpoll: netconsole: local port 6665
[ 1337.829245] netpoll: netconsole: local IPv4 address 192.168.1.101
[ 1337.829251] netpoll: netconsole: interface 'wlan0'
[ 1337.829258] netpoll: netconsole: remote port 6666
[ 1337.829268] netpoll: netconsole: remote IPv4 address 192.168.1.200
[ 1337.878908] netpoll: netconsole: remote ethernet address ff:ff:ff:ff:ff:ff
[ 1337.978565] printk: console [netcon0] enabled
[ 1337.991812] netconsole: network logging started
rpi pi #
```

On sender:

```
# echo "$(date): hey " > /dev/kmsg
```

On receiver:

```
rpi ~ $ netcat -d -u -l 6666
[14219.718949] Friday 01 January 2021 12:33:23 PM IST: hey
...
```

Now, set it up (via systemd/cron) such that both sender/receiver startup their netconsole services at boot; have the netcat append to a log file; you're all set!

Scripts to help automate using netconsole

Find them on
https://github.com/PacktPublishing/Linux-Kernel-Debugging/blob/main/ch10/netcon
and
https://github.com/kaiwan/usefulsnips
as netcon*

Running it via cron, perhaps-

```
# crontab -e
```

```
@reboot netconsole-setup 192.168.1.200
#
```

Receiver (Linux on R Pi / x86_64):

```
$ crontab -l
@reboot /home/pi/netcon_recv.sh

$ cat /home/pi/netcon_recv.sh
#!/bin/bash
# netcon_recv.sh
# netconsole: receiver
#  Receives the console output from another system on port 6666
PORT=6666
LOG=~/netcon.log  # update as required
echo "------------- $(date) -------------" >> ${LOG}
netcat -d -u -l ${PORT} |tee -a ${LOG}
$
```

Receiver (Windows):

```
REM bat script for running netcat receiver - for netconsole
@echo off
title Netconsole Receiver
set logfile="netcon.txt"
ECHO "Running: nc64.exe -luv -p 6666 >> %logfile%"
nc64.exe -luv -p 6666 >> %logfile%
```

Quick sanity test

- On sender (where the netconsole module's now running):

```
# echo "hello from $USER" > /dev/kmsg
```

On receiver:

```
$ netcat -d -u -l 6666
[584388.729694] hello from root
...
```

Done.

(Virtual) Serial console: especially useful when working on a VM:
Like the Seawolf custom appliance on VirtualBox:

...

Serial Console in VirtualBox

In some debug scenerios it can be helpful to debug the kernel running inside a virtual machine. This is useful for some classes of non-hardware specific bugs, for example generic kernel core problems or debugging file system drivers.

One can capture Linux console messages running inside VirtualBox by setting it the VirtualBox serial log to /tmp/vbox and running a serial tty communications program such as minicom, and configure it to communicate with a named pipe tty called unix#/tmp/vbox
Boot with virtualised kernel boot line:

```
console=ttyS0,9600
```

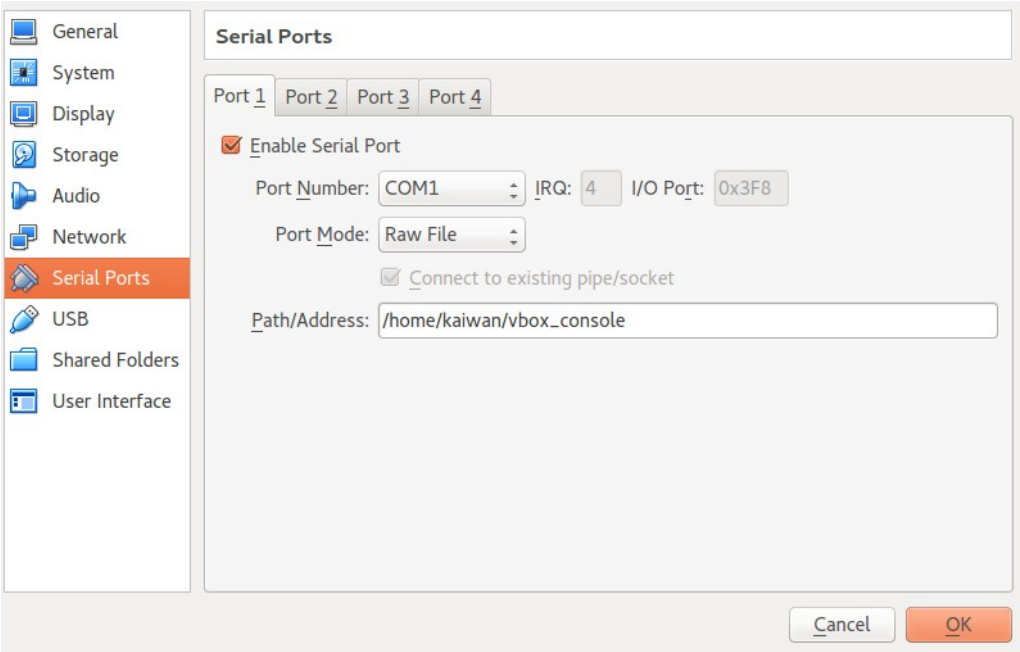
and minicom will capture the console messages

...

Source: https://www.virtualbox.org/wiki/Serial_redirect

How to redirect the Linux guest kernel console to a serial port / file

For debugging purposes it's often useful to redirect the output of a guest Linux kernel to the host. To



achieve this, set up a serial port on the VM setting of the virtual machine. Choose Raw File which will write the kernel output into a file at the host. More advanced users might select Host Pipe and connect a proper application on the host with the pipe.<<
For example, on the Seawolf appliance, go to Settings / Serial Ports:

>>

Then edit the kernel parameters of the guest Linux kernel. Either do this directly from the GRUB command line or edit the appropriate GRUB config file. For the original GRUB, this is /boot/grub/menu.lst *[outdated: it's /etc/default/grub]* . Add the following parameters:

```
console=ttyS0 console=tty0 ignore_loglevel
```

The first parameter will ensure that all kernel output is redirected to the serial console. The second parameter will ensure that the kernel output is still logged to the guest text console. The third parameter enforces the guest Linux kernel to print all kernel messages to the console.

<<
Now, once the guest is running, the console output gets sent over the (virtual) serial port to the configured "raw" file on the host system:

On the host:

```
$ tail -f ~/vbox_console

...

[    0.000000] BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
[    0.000000] debug: ignoring loglevel setiigg
[    ..000000  XX(EeeuueeDDsall))ppooecio::aattvv
    ..00000]SSBBOO  .5ppeeeett
[   00000000] MM::iinnttk mmHHVVrrualBxxVVrruullox  IISSVVrtuaBBxx11//11200
[    0.000000] Hypervisor detected: KVM
[    0.000000] e820: update [mem 0x00000000-0x00000fff] usable ==> resrrved
[    0.000000] e820: remove [mem 0x000a0000-0x000fffff] usable
[    0.000000] e820: last_pfn = 0x36000 max_arch_pfn = 0x400000000

...

[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000000001000-0x0000000000ffffff]
[    0.000000]   DMA32    [mem 0x0000000001000000-0x0000000035fffff]]
[    0.000000]   Normal   empty
[    0.000000] Movabee zone start for each ooee

    00000000] aall  eeor  ooeerrngss
[   37.156149] PCI host bridge to bus 0000:00
[   37.206873] pci_bus 0000:00: root bus resource [io  0x0000-0xffff]
[   37.319413] pci_bus 0000:00: root bus resource [mem 0x00000000-0x7fffffffff]

...
$
>>
```

<<
Misc: print journal with metadata, showing the (printk) priority level as well:

```
journalctl -o export |egrep  "^PRIORITY|^MESSAGE"
>>
```

The CPU Application Binary Interface (ABI) – a minimal understanding

Refer this blog article

and
Src: the 'Linux Kernel Debugging', Kaiwan NB, Packt, book

Arch (CPU) family	How parameters to a function are passed	Other useful info (typical usage)
IA-32 (x86-32)	All parameters passed via the stack. Access is via offsets to the Stack Pointer (SP) and Base Pointer (BP) registers.	• Stack frame layout: o Parameters <-- higher address o `RET` address o `[SFP]`: Optional, if Frame Pointer (FP) enabled o Local variables <-- lower address (top of stack) • Return value: Typically in accumulator (EAX)
ARM-32 (Aarch32)	Follows the ARM Procedure Call Standard (APCS). The first four parameters to the function are passed in these 32-bit CPU registers: `r0`, `r1`, `r2`, `r3`. Any remaining parameters are passed via the stack (stack frame layout is as with x86-32).	• r4 to r9: Scratch registers (often used for local variables) • r7: If system call being issued, holds system call (syscall) # • r11: If enabled, the FP • r13: Stack Pointer (SP) • r14: Link Register (LR); holds the return address, used to return at function exit • r15: Program Counter (PC). • Return value: typically in `r0`
x86_64	The first six parameters to the function are passed in these 64-bit CPU registers: `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`. Any remaining parameters are passed via the stack (stack frame layout is as with the x86-32, except that 64-bit alignments are used and `RBP` acts as the base pointer register),	• RAX: Accumulator; holds syscall # when syscall being issued • RBP: Base pointer, start of stack • RSP: Stack pointer, current location on stack • Return value: Typically in accumulator (`RAX`) • PSR: Processor Status Register
Aarch64 (ARMv8)	APCS: the first eight parameters to the function are passed in these 64-bit CPU registers: `X0` to `X7`. Any remaining parameters are passed via the stack.	• X8: (indirect) Return value address • X9 to X15: Local variables, caller saved • X29 (FP): Frame pointer • X30 (LR): Link register, used to return at function exit • X31 (SP): Stack pointer or a zero register, depending on context • Return value: Typically in `X0`

Use the “Magic SysRq” feature

An indispensable tool for many lockups is the “magic SysRq key,” which is available on most architectures. Magic SysRq is invoked with the combination of the Alt and SysRq keys on the PC keyboard, or with other special keys on other platforms (see Documentation/admin-guide/sysrq.rst* for details), and is available on the serial console as well. A third key, pressed along with these two, performs one of a number of useful actions:

<<
* .rst?
.**rst files** are ReStructuredText format. They look like text **files**, but can be rendered into HTML with the Python docutils package!
>>

```
How do I enable the magic SysRq key?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

You need to say "yes" to 'Magic SysRq key (CONFIG_MAGIC_SYSRQ)' when
configuring the kernel. When running a kernel with SysRq compiled in,
/proc/sys/kernel/sysrq controls the functions allowed to be invoked via
the SysRq key. The default value in this file is set by the
CONFIG_MAGIC_SYSRQ_DEFAULT_ENABLE config symbol, which itself defaults
to 1. Here is the list of possible values in /proc/sys/kernel/sysrq:

   -  0 - disable sysrq completely
   -  1 - enable all functions of sysrq
   - >1 - bitmask of allowed sysrq functions (see below for detailed function
     description)::

          2 =   0x2 - enable control of console logging level
          4 =   0x4 - enable control of keyboard (SAK, unraw)
          8 =   0x8 - enable debugging dumps of processes etc.
         16 =  0x10 - enable sync command
         32 =  0x20 - enable remount read-only
         64 =  0x40 - enable signalling of processes (term, kill, oom-kill)
        128 =  0x80 - allow reboot/poweroff
        256 = 0x100 - allow nicing of all RT tasks

You can set the value in the file by the following command::

    echo "number" >/proc/sys/kernel/sysrq
```

<<
To enable all SysRq functions, as root:

echo 1 > /proc/sys/kernel/sysrq
>>

```
[...]


What are the 'command' keys?
~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Command	Function
``b``	Will immediately reboot the system without syncing or unmounting your disks.
``c``	Will perform a system crash by a NULL pointer dereference. A crashdump will be taken if configured.
``d``	Shows all locks that are held.
``e``	Send a SIGTERM to all processes, except for init.
``f``	Will call the oom killer to kill a memory hog process, but do not panic if nothing can be killed.
``g``	Used by kgdb (kernel debugger)
``h``	Will display help (actually any other key than those listed here will display help. but ``h`` is easy to remember :-)
``i``	Send a SIGKILL to all processes, except for init.
``j``	Forcibly "Just thaw it" - filesystems frozen by the FIFREEZE ioctl.
``k``	Secure Access Key (SAK) Kills all programs on the current virtual console. NOTE: See important comments below in SAK section.
``l``	Shows a stack backtrace for all active CPUs.
``m``	Will dump current memory info to your console.
``n``	Used to make RT tasks nice-able
``o``	Will shut your system off (if configured and supported).
``p``	Will dump the current registers and flags to your console.
``q``	Will dump per CPU lists of all armed hrtimers (but NOT regular timer_list timers) and detailed information about all clockevent devices.
``r``	Turns off keyboard raw mode and sets it to XLATE.
``s``	Will attempt to sync all mounted filesystems.
``t``	Will dump a list of current tasks and their information to your console.
``u``	Will attempt to remount all mounted filesystems read-only.
``v``	Forcefully restores framebuffer console
``v``	Causes ETM buffer dump [ARM-specific]
``w``	Dumps tasks that are in uninterruptable (blocked) state.
``x``	Used by xmon interface on ppc/powerpc platforms. Show global PMU Registers on sparc64.

```
            Dump all TLB entries on MIPS.

``y``       Show global CPU Registers [SPARC-64 specific]

``z``       Dump the ftrace buffer

``0``-``9`` Sets the console log level, controlling which kernel messages
            will be printed to your console. (``0``, for example would make
            it so that only emergency messages like PANICs or OOPSes would
            make it to your console.)
=========== ========================================================================
```

Okay, so what can I use them for?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Well, unraw(r) is very handy when your X server or a svgalib program crashes.

sak(k) (Secure Access Key) is useful when you want to be sure there is no
trojan program running at console which could grab your password
when you would try to login. It will kill all programs on given console,
thus letting you make sure that the login prompt you see is actually
the one from init, not some trojan program.

.. important::

   In its true form it is not a true SAK like the one in a
   c2 compliant system, and it should not be mistaken as
   such.

It seems others find it useful as (System Attention Key) which is
useful when you want to exit a program that will not let you switch consoles.
(For example, X or a svgalib program.)

``reboot(b)`` is good when you're unable to shut down. But you should also
``sync(s)`` and ``umount(u)`` first.

``crash(c)`` can be used to manually trigger a crashdump when the system is
hung.
Note that this just triggers a crash if there is no dump mechanism available.

``sync(s)`` is great when your system is locked up, it allows you to sync your
disks and will certainly lessen the chance of data loss and fscking. Note
that the sync hasn't taken place until you see the "OK" and "Done" appear
on the screen. (If the kernel is really in strife, you may not ever get the
OK or Done message...)

``umount(u)`` is basically useful in the same ways as ``sync(s)``. I generally
``sync(s)``, ``umount(u)``, then ``reboot(b)`` when my system locks. It's saved
me many a fsck. Again, the unmount (remount read-only) hasn't taken place until
you see the "OK" and "Done" message appear on the screen.

The loglevels ``0``-``9`` are useful when your console is being flooded with
kernel messages you do not want to see. Selecting ``0`` will prevent all but
the most urgent kernel messages from reaching your console. (They will
still be logged if syslogd/klogd are alive, though.)

``term(e)`` and ``kill(i)`` are useful if you have some sort of runaway process
you are unable to kill any other way, especially if it's spawning other
processes.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
"just thaw ``it(j)``" is useful if your system becomes unresponsive due to a
frozen (probably root) filesystem via the FIFREEZE ioctl.
```

...

Note that magic SysRq must be explicitly enabled in the kernel configuration and that most distributions do not enable it, for obvious security reasons. For a system used to develop drivers, however, enabling magic SysRq is worth the trouble of building a new kernel in itself. Magic SysRq may be disabled at runtime with a command such as the following:
echo 0 > /proc/sys/kernel/sysrq

Brendan Gregg's perf-tools

Interesting:
Brendan Gregg has developed useful and powerful bash script wrappers over ftrace, kprobes, etc. Install the '`perf-tools-unstable`' package to try them out.

They come with man pages too! :-)

Git repo.

Once installed:

```
$ (cd /usr/sbin; \ls *-perf)
bitesize-perf     funccount-perf     functrace-perf  killsnoop-perf  perf-stat-
hist-perf  tcpretrans-perf
cachestat-perf    funcgraph-perf     iolatency-perf  kprobe-perf   reset-ftrace-
perf    tpoint-perf
execsnoop-perf    funcslower-perf  iosnoop-perf  opensnoop-perf  syscount-perf
           uprobe-perf
```

OR

```
$ dpkg -L perf-tools-unstable
/.
/usr
/usr/bin
/usr/bin/bitesize
/usr/bin/cachestat
/usr/bin/execsnoop
/usr/bin/funccount
/usr/bin/funcgraph
/usr/bin/funcslower
/usr/bin/functrace
/usr/bin/iolatency
/usr/bin/iosnoop
```

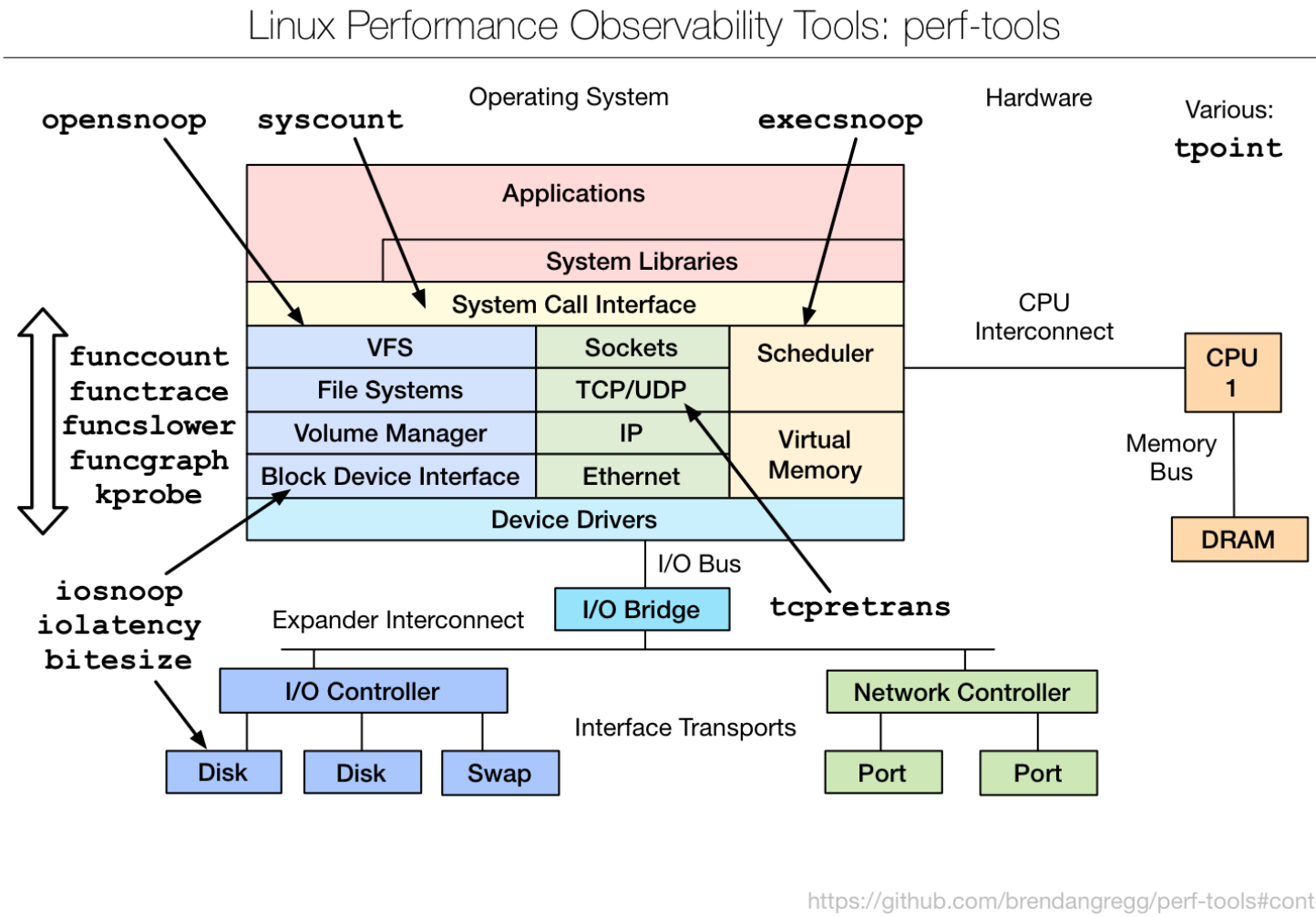
```
/usr/bin/killsnoop
/usr/bin/kprobe
/usr/bin/opensnoop
/usr/bin/perf-stat-hist
/usr/bin/reset-ftrace
/usr/bin/syscount
/usr/bin/tcpretrans
/usr/bin/tpoint
/usr/bin/uprobe
...
$
```

(On some installations, the script name is suffixed with **-perf**).

!WARNING! Many of these tools are considered to be Experimental and Unstable!
Should only be used on kernel ver >= 4.0.

Source

Linux Performance Observability Tools: Perf Tools, Brendan Gregg



Refer to my LKD book, *Ch 9 – Tracing the kernel flow* section *An introduction to using perf-tools* for details and examples.

Eg. screenshot:

Tracing functions that are latency outliers via perf-tool's funcslower

Here's one more quick `perf-tools` example – finding functions that are *latency outliers* with the `funcslower[-perf]` tool! To try this, I check for the `mutex_lock()` kernel function, taking longer than 50 microseconds to complete (I ran this on my native x86_64 laptop, running Ubuntu 20.04 LTS):

```
Linux-Kernel-Debugging $ sudo funcslower-perf -a mutex_lock 50
Tracing "mutex_lock" slower than 50 us... Ctrl-C to end.
# tracer: function_graph
#
#     TIME        CPU  TASK/PID         DURATION                  FUNCTION CALLS
#      |          |     |    |           |   |                     |   |   |   |
284741.775198 |   10) Qt bear-11719  | + 54.044 us   |  } /* mutex_lock */
284741.775400 |   10) Qt bear-11719  | + 61.039 us   |  } /* mutex_lock */
284741.775507 |    0) Qt bear-2678454 | ! 106.166 us  |  } /* mutex_lock */
 10) Qt bear-11719  => chrome-3780976
284761.091939 |   10) chrome-3780976 | + 52.208 us   |  } /* mutex_lock */
284794.433903 |   11) VizComp-13360  | ! 302.772 us  |  } /* mutex_lock */
284811.775269 |    0) Qt bear-2678454 | + 84.911 us   |  } /* mutex_lock */
284811.775321 |    6) Qt bear-11719  | + 51.145 us   |  } /* mutex_lock */
284811.775503 |    6) Qt bear-11719  | + 60.297 us   |  } /* mutex_lock */
284811.775570 |    0) Qt bear-2678454 | + 65.780 us   |  } /* mutex_lock */
284821.775447 |    0) Qt bear-2678454 | ! 101.478 us  |  } /* mutex_lock */
284821.775560 |    6) Qt bear-11719  | ! 112.713 us  |  } /* mutex_lock */
284825.251178 |   10) kworker-3759943 | * 32702.53 us |  } /* mutex_lock */
284831.775498 |    6) Qt bear-11719  | + 53.848 us   |  } /* mutex_lock */
284837.937573 |    1) gnome-s-11328  | ! 144.973 us  |  } /* mutex_lock */
284851.775317 |    6) Qt bear-11719  | + 50.153 us   |  } /* mutex_lock */
284851.775515 |    6) Qt bear-11719  | + 60.809 us   |  } /* mutex_lock */

^C
Ending tracing...
```

Figure 9.26 – A partial screenshot showing how the funcslower[-perf] tool catches function outliers

README: Leveraging Kprobes, kernel (dynamic) event tracing and the kprobe-perf script

1. One more way to see, actually to *trap* into, when (almost) any kernel or module code runs, is via the powerful Kprobes kernel mechanism.

For the 'Linux Kernel Debugging' book, I wrote an interesting script - **kp_load.sh**; it allows one to trap into almost any kernel / module function and optionally display the call stack (that led up to this func being called!).

It's here:
https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch4/kprobes/4_kprobe_helper

Example: trap into the Intel e1000 network driver's hardirq handler:

```
sudo ./kp_load.sh --probe=e1000_intr --verbose --showstack
```

(look up dmesg, then rmmod it).

2. Even simpler, get deep insight into the kernel via the kernel's event tracing infrastructure:
ref: https://www.kernel.org/doc/html/latest/trace/events.html#event-tracing

Do this
```
# echo 'irq:*' > /sys/kernel/tracing/set_event
# cat /sys/kernel/tracing/trace_pipe
```

to see all irq related events !

More generally (as root):
```
cd /sys/kernel/tracing

ls events/irq
enable  irq_handler_entry/  softirq_entry/  softirq_raise/  tasklet_exit/
filter  irq_handler_exit/   softirq_exit/   tasklet_entry/

# capture hardirq entry for 1s
echo 1 > events/irq/irq_handler_entry/enable ; sleep 1; echo 0 >
events/irq/irq_handler_entry/enable

# show results
cat trace_pipe
...
...
 <idle>-0       [005] d.h1. 521639.978305: irq_handler_entry: irq=178
name=iwlwifi:default_queue
 <idle>-0       [004] d.h1. 521639.979391: irq_handler_entry: irq=183
name=iwlwifi:queue_5
 StreamT~s #5139-1549368 [009] d.h.. 521639.991592: irq_handler_entry: irq=194
name=eno2
 <idle>-0       [008] d.h1. 521639.995020: irq_handler_entry: irq=193 name=i915
```

...

Doc

3. Another eg.: trap into whenever a tasklet is scheduled to run!

```
sudo kprobe-perf -s 'p:mytasklets __tasklet_schedule'
```

Interestingly, this triggers on various events, where interrupts with tasklets as bottom halves exist:
- keyboard / input driver (just press/release any keyboard key to see it fire!)
- some types of network drivers
- some block drivers
- ...

4. Using dynamic kprobe event tracing on a kernel module
i. load the target module 'foo'.ko
ii. check for it's (text) symbols in the globak kernel symbole table

```
grep " [tT] .*\[foo\]$" /proc/kallsyms
```

iii. Add funcs to probe (say foo1())

```
cd /sys/kernel/tracing
echo "p:myfoo1 foo1" >> kprobe_events
echo 1 > events/kprobes/myfoo1/enable
cat trace_pipe
```

iv. In another window run the app that will cause the 'foo1()' func to be called.
v. `echo 0 > events/kprobes/myfoo1/enable`

5. Setup a kretprobe – to see the retutn value fro ma given kernel/modue function; eg.

```
kprobe-perf 'r:do_filp_open ret=$retval'
kprobe-perf 'r:__kmalloc ret=$retval'
```

(The ret value shows up as "ret=<ret-val>" !)

(Hmm, i currently find that these work on my native Linux system but not on a VM ??)

---

*"**Don't forget eBPF and its frontends***

Note that many of these perf-tools wrapper scripts are now superseded by the more recent and powerful eBPF technology. Brendan Gregg's answer to this is his newer frontend to eBPF – the *-bpfcc toolset! (You can read more on it here: https://www.brendangregg.com/ebpf.html.) Recall how, in *Chapter 4, Debug via Instrumentation – Kprobes*, in the *Observability with eBPF tools – an introduction* section, when we tried to figure out who was issuing the execve() system call to execute a process, the perf-tools execsnoop-perf wrapper script didn't quite cut it. The

execsnoop-bpfcc BCC frontend wrapper script worked well instead."

Scripts / Utils

1. To resolve a hexadecimal (virtual) address to a source file and line number, use the **addr2line** utility!
`addr2line -e <kernel binary> <addr>`

2. BUT it can't work with KASLR (kernel address space layout randomization) addresses, which most distro kernels use (you could work around this by passing `nokaslr` as a kernel parameter at boot).

From kernel ver 4.8, there's a script within the kernel source tree: **scripts/faddr2line**. It can translates KASLR addresses, unlike addr2line. Requires function name + offset/size string as input (it's usually part of the Oops).

Usage example:
`$ <...>/scripts/faddr2line ~/mykernel/vmlinux meminfo_proc_show+0x5`

See the script & try it out.

Ref:
https://stackoverflow.com/questions/62899999/how-does-addr2line-work-with-virtual-addresses-for-kernel-space-debugging

<<
TIP:
Use gdb's list command `to` figure out the precise place in the code – requires compilation with **-g**

```
$ gdb -q ./oops2.ko
Reading symbols from ./oops2.ko...

[ Let's say the Oops output shows the instruction ptr / program counter like
this:
   RIP: 0010:oops2_init+0x49/0x1000 oops2

  Use the gdb list command to figure out the precise place in the code –
requires
  compilation with -g !
]

(gdb) l *oops2_init+0x49
0x85 is in oops2_init (<...>/k_oops_warn_panic/oops2/oops2.c:43).
38                return -ENOMEM;
39          MSG("sizeof(long) = %d, sizeof(struct faker) = %lu, actual space
alloced = %lu\n",
40              sizeof(long), sizeof(struct faker), ksize(f1));
41    #endif
42          MSG("Hello, about to Oops!\n");
43          f1->bad_cache_align = 0xabcd;  << exactly right! >>
```

```
44
45          return 0;
46      }
47
(gdb)

>>
```

3. Use these

- **scripts/decodecode** : to see the assembly where the Oops occured; input: redirect the oops text as stdin (via the <)

- **scripts/decode_stacktrace.sh** : to get a better stack trace; input: the kernel vmlinux with symbols + kernel source tree + the Oops redirected as stdin.

  Can introduce this within the kernel module Makefile as a target:

```
decode_stack:
 # expect you have stored the 'Oops' in a file named ./oops.txt
      $(KDIR)/scripts/decode_stacktrace.sh $(KDIR)/vmlinux $(KDIR)/ <
oops.txt > decoded_oops.txt
```

ref: https://lwn.net/Articles/592724/

Tip: if cross-compiled, first set the variables ARCH and CROSS_COMPILE appropriately

---

Tips and tricks in debugging kernel drivers in Linux, Feb 2020

- tip on how to configure and use the kernel's **dynamic debug** facility is here (u/s Dynamic debug)

Miscellaneous tips on OSDev's wiki site:
https://wiki.osdev.org/Kernel_Debugging

(The pseudo breakpoint is interesting).

Can see a few actual bugs and their fixes here, at the ***Online Linux Driver Verification Service***
http://linuxtesting.org/ldv/online?action=rules

F.e.:

- memory allocation while holding a spinlock: must be done with GFP_ATOMIC (and not with GFP_KERNEL): bug “orinoco: fix GFP_KERNEL in orinoco_set_key with interrupts disabled” *link*
- mutex: Locking a mutex twice or unlocking without prior locking; *link* ; *sample bug and fix*
- USB memory leak; usb_alloc_urb() requires a corresponding usb_free_urb(); *sample bugfix*

Detected by Coverity !

Recent:

https://cos.googlesource.com/third_party/kernel/+/490ca207fb0f23bca9d21c04e309502e1ed8b58d%5E%21/#F0

```
    commit 490ca207fb0f23bca9d21c04e309502e1ed8b58d                [log] [tgz]
    author Eric Sandeen <sandeen@redhat.com>                       Tue Jul 13 17:49:23 2021 +0200
committer COS Cherry Picker <cloud-image-release@prod.google.com>Wed Jul 21 09:56:21 2021 -0700
      tree aa7087ee642486f3962129d4df712bfd9c565ae1
    parent b419957a802ee4d0cd7a646a2fd6cb2967dc11ef [diff]
```

```
seq_file: disallow extremely large seq buffer allocations

There is no reasonable need for a buffer larger than this, and it avoids
int overflow pitfalls.

BUG=b/194226724
TEST=presubmit
SOURCE=UPSTREAM(8cae8cd89f05f6de223d63e6d15e31c8ba9cf53b)
RELEASE_NOTE=Fixed CVE-2021-33909

cos-patch: security-high
Fixes: 058504edd026 ("fs/seq_file: fallback to vmalloc allocation")
Suggested-by: Al Viro <viro@zeniv.linux.org.uk>
Reported-by: Qualys Security Advisory <qsa@qualys.com>
Signed-off-by: Eric Sandeen <sandeen@redhat.com>
Cc: stable@kernel.org
Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>
Change-Id: Ief7d9d22ef5e05a29acc3817d4554556f05ecf8a
Reviewed-on: https://cos-review.googlesource.com/c/third_party/kernel/+/19731
Tested-by: Cusky Presubmit Bot <presubmit@cos-infra-prod.iam.gserviceaccount.com>
Main-Branch-Verified: Cusky Presubmit Bot <presubmit@cos-infra-prod.iam.gserviceaccount.com>
Reviewed-by: Vaibhav Rustagi <vaibhavrustagi@google.com>
```

```
diff --git a/fs/seq_file.c b/fs/seq_file.c
index 1600034..c19ecc1 100644
--- a/fs/seq_file.c
+++ b/fs/seq_file.c
```

```
@@ -29,6 +29,9 @@

 static void *seq_buf_alloc(unsigned long size)
 {
+       if (unlikely(size > MAX_RW_COUNT))
+               return NULL;
+
        return kvmalloc(size, GFP_KERNEL_ACCOUNT);
 }
```

Also see : http://linuxtesting.org/ldv

...

Results

Of course, the numerous hours of computing power spent testing our tools were not wasted. **More than 400 problems** were found in Linux device drivers, and they have been already fixed in upstream. The list of problems found is constantly growing as the tools used are improved.

We have uploaded error traces for some of the problems found. You can investigate how LDV Tools results look like on these examples:

- error traces reported by CPAchecker engine;
- error traces reported by BLAST engine.

This outcome proves to be interesting for verification community as well. Verification tasks generated by LDV Tools are included into SV-COMP verification benchmarks and it has been adopted as regression tests for CPAchecker verification framework.

...

Problems in Linux Kernel

This section contains information about problems in Linux kernel found within Linux Driver Verification program as well as within KEDR and Linux File System Verification projects.

No.	Type	Brief	Added on	Accepted	Status
L0351	Crash	regulator: tps65217: NULL pointer dereference on probe	2019-09-26	https://lkml.org/lkml/2018/7/27/661 commit	Fixed in kernel v4.19-rc1
L0350	Crash	scsi: 3ware: fix return 0 on the error path of probe	2019-09-25	https://lkml.org/lkml/2018/7/27/655 commit	Fixed in kernel v4.19-rc1
L0349	Crash	gpio: ml-ioh: buffer underwrite on probe error path	2019-09-24	https://lkml.org/lkml/2018/7/23/949 commit	Fixed in kernel v4.19-rc1

[...]