Kernel Panic. Now what?

Hannes Frederic Sowa
<hannes@redhat.com>

Outline

- When do kernel panics happen?
- Where do fails happen?
- Consequences
- Capturing information from kernel panics
- Interpreting kernel panics
- Postmortem analysis

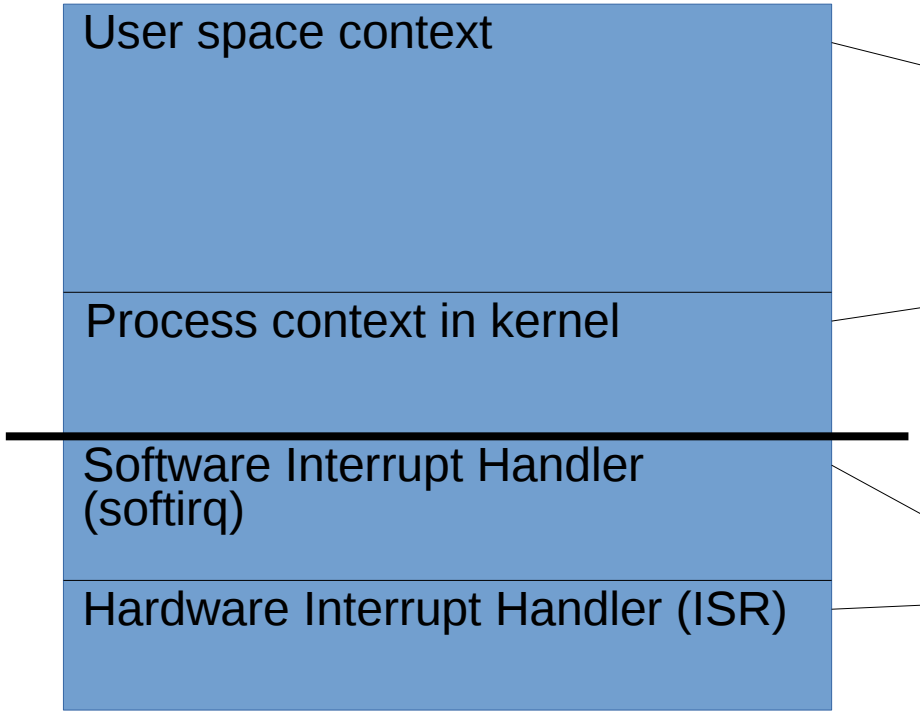
When do kernel panics happen?

- Normally they don’t
  - Some CPUs (mostly with complex MMUs) allow the separation of operating system code and ordinary user code – those are CPUs that Linux targets
  - Well defined entry points to the kernel, so called syscalls, validate input data and act accordingly under the premise to not crash the kernel but fail softly – return an error to user space instead of a transition to an invalid state
  - External input data (from IO devices) either is treated transparently for the kernel or subject to the same tight input validation restrictions

When do kernel panics happen?

- Invalid memory dereference or invalid opcode
  - Panic via exception handler
- panic() or BUG_ON() macro in kernel code
  - Expands to undefined instruction and also causes exception handler to run
- Recursive exception handling (double and triple faults)
- Division by zero
- Stack overflows
- Unkown NMIs
- Out of memory
- watchdogs or lockups or stalls
  - Based on timeout
  - Sometimes imprecise
- MCE and MCA
  - Unrecoverable CPU, memory or bus faults
    - Might be asynchronous
  - Be aware of data corruption
- Intentionally triggered kernel panics
  - e.g. via sysrq
  - Failures to record audit information in high sensitive environments
  - Used for hot stand by systems to e.g. avoid split horizons
    - be aware – less safe than STONITH hardware devices

Where do fails happen?



Programs get notified of invalid actions via signal handlers. Can log output via coredump_filter or to dmesg.

Oops in process context, user space program transitions to an invalid program state and freezes.
Depending on panic_on_oops setting (default enabled on RHEL, disabled upstream). Introspectable via procfs.

Real non-recoverable panic happens.
No context available – kernel does not know how to proceed.

The context matters!

Kernel panics stop the whole system, an Oops allows the kernel to continue.
The same code can trigger either oops or panic depending from where it gets called

Consequences

- System comes to a halt and dumps panic information
  - Obviously panic handlers are platform dependent
    - Register file layout
    - Stack layout
- Panic does not sync filesystems – be aware
  - *Normally* not a problem due to journals or CoW and boot up fsck
  - But user space data consistency (think about databases)
- Panic capture process can start automatically
- Panic timeouts can cause an automatic reboot
  - Due to availability reasons system should automatically reboot on panic and try to restore services ASAP
  - Asynchronous problem investigation

Capturing information from kernel panics

- Traditional Unix™ way is to write memory dump to swap and save on boot up to a file
  - Solaris, BSDs
- Linux is different:
  - No low level independent block disk handlers
    - How reliable would that be anyway but worked fine for above mentioned operating systems though.
  - Several alternatives on Linux
    - Taking picture with mobile phone (Google summer of code project: QR code)
    - Saving kmsg to pstore on panic
      - Stores kmsg buffer in efi variables on x86
      - Depending on EFI implemenation, visible machine setup
      - Dangerous due to the often broken efi implementations (garbage collection)
      - ramoops can store kmsg in RAM and next boot allows to preserve this area
    - netconsole (dump kmsg buffer over network)
    - kexec with kdump (Red Hat's preferred way)
    - QEMU memory dump over monitor / libvirt if virtualized
    - kgdb and kdb
    - Serial console or early_printk via USB EHCI debug port
- Note: sometimes an oops or panic comes not alone
  - One particular panic or oops can be followed by multiple related or unrelated failure cases
  - Most of the time only the first one is relevant to diagnose the problem (but provide support all of them)

Interpreting kernel panics 1/3

```
[  381.176301] BUG: unable to handle kernel NULL pointer dereference at           (null)
[  381.185049] IP: sysrq_handle_crash+0x16/0x20
[  381.189812] PGD 0
[  381.189813] P4D 0
[  381.192053]
[  381.195953] Oops: 0002 [#1] SMP
[  381.199456] Modules linked in: intel_rapl x86_pkg_temp_thermal intel_powerclamp coretemp kvm_intel kvm ixgbe igb
irqbypass crct10dif_pclmul crc32_pclmul be2net ipmi_ssif ghash_clmulni_intel ptp mdio ipmi_si pps_core mei_me
ipmi_devintf iTCO_wdt iTCO_vendor_support dca intel_cstate snd_pcsp snd_pcm snd_timer snd intel_uncore dcdbas
soundcore ipmi_msghandler mei lpc_ich intel_rapl_perf shpchp wmi acpi_power_meter tpm_tis tpm_tis_core tpm xfs
libcrc32c mgag200 i2c_algo_bit drm_kms_helper ttm drm crc32c_intel
[  381.249448] CPU: 16 PID: 1266 Comm: bash Not tainted 4.12.9-300.fc26.x86_64 #1
[  381.257508] Hardware name: Dell Inc. PowerEdge R730/0599V5, BIOS 1.2.10 03/09/2015
[  381.265955] task: ffffa0995460a5c0 task.stack: ffffc3130ea84000
[  381.272564] RIP: 0010:sysrq_handle_crash+0x16/0x20
[  381.277908] RSP: 0018:ffffc3130ea87dd8 EFLAGS: 00010282
[  381.283737] RAX: 000000000000000f RBX: ffffffff91f93280 RCX: 0000000000000000
[  381.291699] RDX: 0000000000000000 RSI: ffffa0997f40e128 RDI: 0000000000000063
[  381.299661] RBP: ffffc3130ea87dd8 R08: 0000000000000534 R09: 0000000000000007
[  381.307622] R10: 0000000000000001 R11: ffffffff92311b6d R12: 0000000000000007
[  381.315584] R13: 0000000000000063 R14: 0000000000000002 R15: ffffa0996e32d400
[  381.323547] FS:  00007f3f2bcb5b40(0000) GS:ffffa0997f400000(0000) knlGS:0000000000000000
[  381.332577] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  381.338989] CR2: 0000000000000000 CR3: 0000001fed462000 CR4: 00000000001406e0
```

Can we relate the information to some specific user space action?

- Timestamp when the panic happened
- Process scheduled to run at that time
- CPU the bug is happening on
- Function in the kernel that caused the panic
- Are we tainted?

Interpreting kernel panics 2/3

```
[  381.346950] Call Trace:
[  381.349679]  __handle_sysrq+0x99/0x160
[  381.353862]  write_sysrq_trigger+0x34/0x40
[  381.358435]  proc_reg_write+0x42/0x70
[  381.362522]  __vfs_write+0x37/0x160
[  381.366415]  ? selinux_file_permission+0xfb/0x120
[  381.371664]  ? security_file_permission+0x3b/0xc0
[  381.376914]  vfs_write+0xb1/0x1a0
[  381.380612]  SyS_write+0x55/0xc0
[  381.384215]  entry_SYSCALL_64_fastpath+0x1a/0xa5
[  381.389367] RIP: 0033:0x7f3f2b397210
[  381.393355] RSP: 002b:00007ffed6fd4d28 EFLAGS: 00000246 ORIG_RAX: 0000000000000001
[  381.401803] RAX: ffffffffffffffda RBX: 0000000000000003 RCX: 00007f3f2b397210
[  381.409764] RDX: 0000000000000002 RSI: 00005564e4c30410 RDI: 0000000000000001
[  381.417726] RBP: 00005564e4c5466d R08: 000000000000000a R09: 00007f3f2bcb5b40
[  381.425688] R10: 00005564e4c388a0 R11: 0000000000000246 R12: 00005564e4c54685
[  381.433650] R13: 00005564e4c583e8 R14: 00000000ffffffffd R15: 0000000000000000
[  381.441612] Code: 4c 89 e7 e8 9d fb ff ff e9 be fe ff ff 0f 1f 84 00 00 00 00 00 0f 1f 44 00 00 55 c7
05 18 f6 dd 00 01 00 00 00 48 89 e5 0f ae f8 <c6> 04 25 00 00 00 00 01 5d c3 0f 1f 44 00 00 55 c7 05 50
dd 92
[  381.462711] RIP: sysrq_handle_crash+0x16/0x20 RSP: ffffc3130ea87dd8
[  381.469703] CR2: 0000000000000000
```

Can we relate the information to some specific user space action?

- Call trace leading to the panic
  - Question mark indicates unreliable stack frame information
- Register file dump of user space process causing fault (ORIG_RAX → write)
- Assembly code dump around the faulting location

Disassembling the code

```
All code
========
   0:   4c 89 e7              mov    %r12,%rdi
   3:   e8 9d fb ff ff        callq  0xfffffffffffffba5
   8:   e9 be fe ff ff        jmpq   0xfffffffffffffecb
   d:   0f 1f 84 00 00 00 00  nopl   0x0(%rax,%rax,1)
  14:   00
  15:   0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)
  1a:   55                    push   %rbp
  1b:   c7 05 18 f6 dd 00 01  movl   $0x1,0xddf618(%rip)        # 0xddf63d
  22:   00 00 00
  25:   48 89 e5              mov    %rsp,%rbp
  28:   0f ae f8              sfence
  2b:*  c6 04 25 00 00 00 00  movb   $0x1,0x0              <-- trapping instruction
  32:   01
  33:   5d                    pop    %rbp
  34:   c3                    retq
  35:   0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)
  3a:   55                    push   %rbp
  3b:   c7                    .byte 0xc7
  3c:   05                    .byte 0x5
  3d:   50                    push   %rax
  3e:   dd                    .byte 0xdd
  3f:   92                    xchg   %eax,%edx

Code starting with the faulting instruction
===========================================
   0:   c6 04 25 00 00 00 00  movb   $0x1,0x0
   7:   01
   8:   5d                    pop    %rbp
   9:   c3                    retq
   a:   0f 1f 44 00 00        nopl   0x0(%rax,%rax,1)
   f:   55                    push   %rbp
  10:   c7                    .byte 0xc7
  11:   05                    .byte 0x5
  12:   50                    push   %rax
  13:   dd                    .byte 0xdd
  14:   92                    xchg   %eax,%edx
```

Align with C code

```
static void sysrq_handle_crash(int key)
{
     char *killer = NULL;

     /* we need to release the RCU read lock here,
      * otherwise we get an annoying
      * 'BUG: sleeping function called from invalid context'
      * complaint from the kernel before the panic.
      */
     rcu_read_unlock();
     panic_on_oops = 1;     /* force panic */
     wmb();
     *killer = 1;
}
```

In this case it is easy to align assembly output to C code. The panic happens synchronously with the action by the user space program and the instruction pointer directly points to the correct function. No compiler optimizations in place.

Otherwise:
Playing around with crash, gdb or objdump can hint you to the specific instruction stream that caused the particular panic.

Postmortem analysis

- GDB scripts and macros (scripts/gdb / CONFIG_GDB_SCRIPTS)
  - Allow for extraction of basic information
  - Along with dwarf debuginfo allow to navigate in the kernel memory dump (debuginfo available in separate package)
- crash
  - Tool for introspection of kernel memory dumps
  - Using dwarf debug information and hard coded structure introspection
    - Needs to be regularly updated to work with newer kernels
      - Normally the case for Red Hat released kernels
  - Higher level introspection possibilities than gdb
    - Allows associating kernel vma addresses with slabs
    - Resolves memory addresses to symbols via kallsyms
    - Helps with pfn, physical and virtual addresses

Reproducibility

- Interpreting from the stack trace the likely code path to hit the panic
- Kernel-debug package
  - enables more instrumentation:
    - Can change binary code dramatically
      - Stack traces and addresses don’t match up with the original kernel
    - uninlines several functions and thus allows tracing (e.g. locks)
    - Enables memory checking (red zones), thus e.g. out of bound writes cause earlier panics
    - Detection of possible dead locks
- Custom kernel could even enable more expensive debugging checks
  - Unmaps pages on free thus causing early faults on dereference
  - Stores allocating frame inside page structure to introspect the origin of the allocation
  - KASAN with more recent kernels

Spicing up the panic

- ftrace_dump_on_oops
  - If panic is reproducible, the panic itself can be spiced up with additional information
  - Trace points can be activated that dump their content into a ring buffer (relayfs)
  - If above mentioned sysctl is enabled, ftrace buffer will dumped along with the kernel panic
  - Can provide much more context which functions got called leading to the panic

Spicing up the panic

```
[  186.649946]     bash-1068    9.... 14093368us : sysrq_handle_crash <-__handle_sysrq
[  186.658504]     bash-1068    9d... 14093369us : __do_page_fault <-do_page_fault
[  186.666673]     bash-1068    9.... 14093369us : down_read_trylock <-__do_page_fault
[  186.675230]     bash-1068    9.... 14093369us : _cond_resched <-__do_page_fault
[  186.683399]     bash-1068    9.... 14093370us : find_vma <-__do_page_fault
[  186.691080]     bash-1068    9.... 14093370us : vmacache_find <-find_vma
[  186.698568]     bash-1068    9.... 14093370us : vmacache_update <-find_vma
[  186.706251]     bash-1068    9.... 14093371us : bad_area <-__do_page_fault
[  186.713934]     bash-1068    9.... 14093371us : up_read <-bad_area
[  186.720842]     bash-1068    9.... 14093371us : __bad_area_nosemaphore <-bad_area
[  186.729203]     bash-1068    9.... 14093371us : no_context <-__bad_area_nosemaphore
[  186.737760]     bash-1068    9.... 14093372us : fixup_exception <-no_context
[  186.745639]     bash-1068    9.... 14093372us : search_exception_tables <-fixup_exception
[  186.754778]     bash-1068    9.... 14093373us : search_module_extables <-search_exception_tables
[  186.764594]     bash-1068    9.... 14093373us : __module_address <-search_module_extables
[  186.773732]     bash-1068    9.... 14093374us : is_prefetch.isra.23 <-no_context
[  186.781996]     bash-1068    9.... 14093374us : convert_ip_to_linear <-is_prefetch.isra.23
[  186.791232]     bash-1068    9.... 14093374us : __probe_kernel_read <-is_prefetch.isra.23
[  186.800360]     bash-1068    9.... 14093374us : __check_object_size <-__probe_kernel_read
[  186.809498]     bash-1068    9.... 14093375us : __virt_addr_valid <-__check_object_size
[  186.818441]     bash-1068    9.... 14093375us : check_stack_object <-__check_object_size
[  186.827483]     bash-1068    9.... 14093375us : oops_begin <-no_context
[  186.834869]     bash-1068    9.... 14093375us : oops_enter <-oops_begin
[  186.842252] ---------------------------------
[  186.847113] Modules linked in: intel_rapl x86_pkg_temp_thermal intel_powerclamp coretemp kvm_intel ixgbe igb kvm irqbypass crct10dif_pclmul
crc32_pclmul ptp ipmi_ssif pps_core ghash_clmulni_intel snd_pcsp snd_pcm snd_timer mdio intel_cstate snd dca be2net ipmi_si intel_uncore dcdbas
soundcore iTCO_wdt intel_rapl_perf iTCO_vendor_support lpc_ich ipmi_devintf ipmi_msghandler tpm_tis tpm_tis_core tpm mei_me mei shpchp
acpi_power_meter wmi xfs libcrc32c mgag200 i2c_algo_bit drm_kms_helper ttm drm crc32c_intel
[  186.897156] CPU: 9 PID: 1068 Comm: bash Tainted: G            L  4.12.9-300.fc26.x86_64 #1
[  186.906479] Hardware name: Dell Inc. PowerEdge R730/0599V5, BIOS 1.2.10 03/09/2015
[  186.914927] task: ffff889b934da5c0 task.stack: ffffaf340dffc000
[  186.921538] RIP: 0010:sysrq_handle_crash+0x16/0x20
[  186.926887] RSP: 0018:ffffaf340dfffdd8 EFLAGS: 00010286
[  186.932721] RAX: 000000000000000f RBX: ffffffff83f93280 RCX: 0000000000000000
[  186.940687] RDX: 0000000000000000 RSI: ffff889bbf24e128 RDI: 0000000000000063
[  186.948652] RBP: ffffaf340dfffdd8 R08: 000000000000055b R09: 0000000000000007
[  186.956617] R10: ffff889bbec02540 R11: ffff889bb1ff7534 R12: 0000000000000007
[  186.964581] R13: 0000000000000063 R14: 0000000000000002 R15: ffff889bae0f9100
[  186.972547] FS:  00007f81fd7cab40(0000) GS:ffff889bbf240000(0000) knlGS:0000000000000000
[  186.981578] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  186.987990] CR2: 0000000000000000 CR3: 0000001fda1e4000 CR4: 00000000001406e0
```