What's in a Kernel Oops?

Vlastimil Babka, LinuxDays 2015

CHARLES UNIVERSITY IN PRAGUE
faculty of mathematics and physics

Department of
Distributed and
Dependable
Systems
Kernel oops/panic output

- Printed in console typically on fatal CPU exceptions
  - Lots of architecture-specific information
  - May be enough to figure out the bug without a crash dump
- Oops leaves the system running
  - Kills just the current process (including kernel threads!)
  - System can still be left inconsistent (locks remain locked ... )
- Panic kills the system completely
  - Oops in interrupt, with panic_on_oops enabled, manual panic() calls
    - HW failure, critical memory allocation fail, init/idle task killed, int. handler killed
  - May trigger crash dump if configured, or reboot after delay
Example kernel oops (x86_64)

[ 266.491864] ------------[ cut here ]------------
[ 266.491904] kernel BUG at mm/rmap.c:399!
[ 266.491934] invalid opcode: 0000 [#1] SMP
[ 266.491962] Modules linked in: amdkfd amd_iommu_v2 radeon cfbfillrect cfbimgblt
cfbcopyarea drm_kms_helper ttm fuse
[ 266.492043] CPU: 3 PID: 5155 Comm: java Not tainted 3.19.0-rc3-kfd+ #24
[ 266.492087] Hardware name: AMD BALLINA/Ballina, BIOS WBL3B20N_Weekly_13_11_2
11/20/2013
[ 266.492141] task: ffff8800a3b3c840 ti: ffff8800916f8000 task.ti: ffff8800916f8000
[ 266.492191] RIP: 0010:<ffffffff81126630>  [<ffffffff81126630>]
unlink_anon_vmas+0x102/0x159
[ 266.492249] RSP: 0018:ffff8800916fbb68 EFLAGS: 00010286
[ 266.492285] RAX: ffff88008f6b3b90 RBX: ffff88008f6b3b90 RCX: ffff8800a3b3cf30
[ 266.492331] RDX: ffff8800914b3c98 RSI: 0000000000000001 RDI: ffff8800914b3c98
[ 266.492376] RBP: ffff8800916fbb8 R08: 0000000000000000 R09: 0000000000000000
[ 266.492421] R10: 0000000000000008 R11: 0000000000000001 R12: ffff88008f686068
[ 266.492465] R13: ffff8800914b3c98 R14: ffff88008f6b3b90 R15: ffff88008f686000
[ 266.492513] FS: 00007f8966f6700(0000) GS:ffff88011ed8000(0000)
knlGS:0000000000000000
[ 266.492566] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 266.492601] CR2: 00007f50fa190770 CR3: 0000000001b31000 CR4: 0000000000407e0
[ 266.492652] Stack:
[ 266.492665] 0000000000000000 ffff88008f686078 ffff8800916fbb8 ffff88008f686000
[ 266.492714] ffff8800916fbc08 0000000000000000 0000000000000000 ffff88008f686000
[ 266.492764] ffff8800916fbbf8 ffffffff8111ba5d 00007fb885918000 ffff88008edf3000
...
Example kernel oops (x86_64)

... 266.492815] Call Trace:
  266.492815]  [ffffffff8111ba5d] free_pgtables+0x8e/0xcc
  266.492834]  [ffffffff8112253e] exit_mmap+0x84/0x116
  266.492907]  [ffffffff8103f789] mmput+0x52/0xe9
  266.492940]  [ffffffff81043918] do_exit+0x3cd/0x9c9
  266.492975]  [ffffffff8170c1ec] ? _raw_spin_unlock_irq+0x2d/0x32
  266.493016]  [ffffffff81044d7f] do_group_exit+0x4c/0xc9
  266.493051]  [ffffffff8104eb87] get_signal+0x58f/0x5bc
  266.493090]  [ffffffff810022c4] do_signal+0x28/0x5b1
  266.493123]  [ffffffff8170ca0c] ? sysret_signal+0x5/0x43
  266.493162]  [ffffffff81002882] do_notify_resume+0x35/0x68
  266.493200]  [ffffffff8170cc7f] int_signal+0x12/0x17
  266.493235] Code: e8 03 b7 f4 ff 49 8b 47 78 4c 8b 20 48 8d 58 f0 48 8b 7b 08 83 bf 8c 00 00 00 00 74 02
  266.493404] RIP [ffffffff81126630] unlink_anon_vmas+0x102/0x159
  266.493447] RSP [ffffff8800916fbb68]
  266.508877] ---[ end trace 02d28fe9b3de2e1a ]---
  266.508880] Fixing recursive fault but reboot is needed!

(source: https://lkml.org/lkml/2015/1/11/14)
Example kernel oops

File + line translation enabled by `CONFIG_DEBUG_BUGVERBOSE` (implemented by `__bug_table` section on x86 - ~70-100kB)

The line in question contains:

```
BUG_ON(anon_vma->degree);
```

This is essentially a hard assertion:

```
if (<condition>) BUG()
```

On x86, BUG() emits a standardized invalid opcode UD2 (0F 0B) triggering an exception. The exception handler checks for UD2 opcode and searches __bug_table for details.
Example kernel oops

```
[ 266.491864] ------------[ cut here ]------------
[ 266.491904] kernel BUG at mm/rmap.c:399!
[ 266.491934] invalid opcode: 0000 [#1] SMP
[ 266.491962] Modules linked in: amdkfd amd_iommu_v2 radeon cfbfillrect cfbimgblt
cfbcopyarea drm_kms_helper ttm file
[ 266.492043] CPU: 3 PID: 5155 Comm: java Not tainted 3.19.0-rc3-kfd+ #24
[ 266.492087] Hardware name: AMD BALLINA/Ballina, BIOS WBL3B20N_Weekly_13_11_2
11/20/2013
[ 266.492141] task: ffff8800a3b3c840 ti: ffff8800916f8000 task.ti: ffff8800916f8000
[ 266.492191] RIP: 0010:[<ffffffff81126630>] [<ffffffff81126630>]
unlink_anon_vmas+0x102/0x159
[ 266.492249] RSP: 0018:ffff8800916fbb68 EFLAGS: 00010286
[ 266.492285] RAX: ffff88008f6b3ba0 RBX: ffff88008f6b3b90 RCX: ffff8800a3b3cf30
[ 266.492331] RDX: ffff8800914b3c98 RSI: 0000000000000001 RDI: ffff8800914b3c98
[ 266.492376] RBP: ffff8800916fbba8 R08: 0000000000000002 R09: 0000000000000000
[ 266.492421] R10: 0000000000000008 R11: 0000000000000001 R12: ffff88008f686068
[ 266.492465] R13: ffff8800914b3c98 R14: ffff88008f6b3b90 R15: ffff88008f686000
[ 266.492513] FS: 00007fb8966f6700(0000) GS:ffff88011ed8000(0000)
knlGS:0000000000000000
[ 266.492566] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 266.492601] CR2: 00007f50fa190770 CR3: 0000000001b31000 CR4: 00000000000407e0
[ 266.492652] Stack:
[ 266.492665] 0000000000000000 ffff88008f686078 ffff8800916fbba8 ffff88008f686000
[ 266.492714] ffff8800916fbc08 0000000000000000 0000000000000000 ffff88008f686000
[ 266.492764] ffff8800916fbbf8 ffffffff8111ba5d 00007fb885918000 ffff88008edf3000
...```

x86- and exception-specific error code (32-bit hex number). Typically useful for page fault exceptions where it's a mask:

- Bit 0 – Present
- Bit 1 – Write
- Bit 2 – User
- Bit 3 – Reserved write
- Bit 4 – Instruction fetch
Example kernel oops

Oops counter, followed by state of selected important kernel config options:

PREEMPT
SMP
DEBUG_PAGEALLOC
KASAN
Example kernel oops

Mostly useful when it is known which drivers are built as modules (e.g.
with standard distro kernel configs).

May also contain module taint flags:
P – proprietary
O – out-of-tree
F – force-loaded
C – staging
E – unsigned (suse)
X – external support (suse) / unsigned
N – no support (suse)
+/− – being loaded/unloaded
Example kernel oops

Information about CPU, process, kernel version, hardware.

Taint flags:
- POFCEXN – same as per-module
- R – module was force-unloaded
- M – system has reported a MCE
- B – bad page was encountered
- U – userspace-defined
- D – there was an oops before
- W – there was a warning before
- A – ACPI table was overridden
- I – firmware bug workaround
- L – soft-lockup has occurred before
- K – kernel has been live patched
- S – SMP kernel on UP machine
Example kernel oops

Information about task that's supposed to be currently running, and whose stack we are actually running on.
Example kernel oops

Kernel BUG at mm/rmap.c:399!
invalid opcode: 0000

Modules linked in: amdkfd amd_iommu_v2 radeon cfbfillrect cfbimgblt
cfbcopyarea drm_kms_helper ttm fuse

CPU: 3 PID: 5155 Comm: java Not tainted 3.19.0-rc3-kfd+ #24
Hardware name: AMD BALLINA/Ballina, BIOS WBL3B20N_Weekly_13_11_2
11/20/2013

-task: ffff8800a3b3c840 ti: ffff8800916f8000 task.ti: ffff8800916f8000

unlink_anon_vmas+0x102/0x159

Which instruction was executing, translated to function name + offset.
This may be different from where position where BUG_ON() was reported, if the Function containing BUG_ON was inlined.
Example kernel oops

[ 266.491864] ------------[ cut here ]--
[ 266.491904] kernel BUG at mm/rmap.c:399!
[ 266.491934] invalid opcode: 0000 [#1] SMP
[ 266.491962] Modules linked in: amdkfd amd_iommu_v2 radeon cfbfillrect cfbimgblt
[ 266.492043] CPU: 3 PID: 5155 Comm: java Not tainted 3.19.0-rc3-kfd+ #24
[ 266.492087] Hardware name: AMD BALLINA/Ballina, BIOS WBL3B20N_Weekly_13_11_2
[ 266.492141] 11/20/2013
[ 266.492191] RIP: 0010:[<ffffffff81126630>] [<ffffffff81126630>]
[ 266.492249] RSP: 0018:ffff8800916fbb68 EFLAGS: 00010286
[ 266.492285] RAX: ffff88008f6b3ba0 RBX: ffff88008f6b3b90 RCX: ffff8800a3b3cf30
[ 266.492331] RDX: ffff8800914b3c98 RSI: 0000000000000001 RDI: ffff8800914b3c98
[ 266.492376] RBP: ffff8800916fbba8 R08: 0000000000000000 R09: 0000000000000000
[ 266.492421] R10: 0000000000000008 R11: 0000000000000001 R12: ffff88008f686068
[ 266.492465] R13: ffff8800914b3c98 R14: ffff88008f6b3b90 R15: ffff88008f686000
[ 266.492513] FS: 00007fb8966f6700(0000) GS: ffff88011ed8000(0000)
[ 266.492566] KnlGS: 0000000000000000
[ 266.492666] CS: 0010 DS: 0000 ES: 0000 CR0: 00000000008050033
[ 266.492601] CR2: 00007f50fa190770 CR3: 0000000000b31000 CR4: 00000000000407e0
[ 266.492652] Stack:
[ 266.492665] 00000000000000 ffff88008f686078 ffff8800916fbba8 ffff88008f686000
[ 266.492714] ffff8800916fbc08 00000000000000 0000000000000000 ffff88008f686000
[ 266.492764] ffff8800916fbbf8 fffffff8111ba5d 00007fb885918000 ffff88008edf3000
...
Example kernel oops

CR2: the faulting virtual address
CR3: phys. addr of top level page table
CR4: a mask for enabling various extensions
Example kernel oops

Raw contents of top of the stack starting at RSP
Example kernel oops

... 266.492815] Call Trace:
   266.492834] [ffffffff8111ba5d] free_pgtables+0x8e/0xcc
   266.492873] [ffffffff8112293d] exit_mmap+0x84/0x116
   266.492907] [ffffffff8103f789] mmput+0x52/0xe9
   266.492940] [ffffffff81043918] do_exit+0x3cd/0x9c9
   266.492975] [ffffffff8170c1ec] ? _raw_spin_unlock_irq+0x2d/0x32
   266.493016] [ffffffff81044d7f] do_group_exit+0x4c/0xc9
   266.493051] [ffffffff8104eb87] get_signal+0x58f/0x5bc
   266.493090] [ffffffff810022c4] do_signal+0x28/0x5b1
   266.493123] [ffffffff8170ca0c] ? sysret_signal+0x5/0x43
   266.493162] [ffffffff81002882] do_notify_resume+0x35/0x68
   266.493200] [ffffffff8170cc7f] int_signal+0x12/0x17
   266.493235] Code: e8 03 b7 f4 ff 49 8b 47 78 4c 8b 20 48 8d 58 f0 49 83
   266.493404] RIP  [ffffffff81126630] unlink_anon_vmas+0x102/0x159
   266.493447] RSP  [ffff8800916fb68]
   266.508877] ---[ end trace 02d28fe9b3de2e1a ]---
   266.508880] Fixing recursive fault but reboot is needed!

Backtrace reconstructed by unwinding the stack, showing the return addresses from individual call frames

“?” means a pointer to function is on stack but doesn't fit in the stack frame; could be leftover from previous execution, or heuristics failure
Example kernel oops

A bunch of instructions around the RIP. RIP position denoted by <  >

Recall that 0F 0B is opcode for UD2

We can disassemble the code listing by piping the oops into ./scripts/decodecode in the kernel source tree.
Example decodecode output

```bash
~/linux.git> ./scripts/decodecode < oops-example.txt
```

```
[ 266.493235] Code: e8 03 b7 f4 ff 49 8b 47 78 4c 8b 20 48 8d 58 f0 49 83 ec 10 48 8d 43 10 48 39 45 c8 74 55 48 8b 7b 08 83 bf 8c 00 00 00 00 74 02 <0f> 0b e8 a4 fd ff ff 48 8b 43 18 48 8b 53 10 48 89 df 48 89 42
All code
======
0:   e8 03 b7 f4          callq  0xfffffffffff4b708
  5:   49 8b 47 78           mov    0x78(%r15),%rax
  9:   4c 8b 20             mov    (%rax),%r12
  c:   48 8d 58 f0           lea    -0x10(%rax),%rbx
 10:   49 83 ec 10           sub    $0x10,%r12
 14:   48 8d 43 10           lea    0x10(%rbx),%rax
 18:   48 39 45 c8           cmp    %rax,-0x38(%rbp)
 1c:   74 55                 je     0x73
 1e:   48 8b 7b 08           mov    0x8(%rbx),%rdi
 22:   83 bf 8c 00 00 00 00  cmp1  $0x0,0x8c(%rdi)
 29:   74 02                je     0x2d
 2b:*  0f 0b                 ud2  <-- trapping instruction
2d:   e8 a4 fd ff ff         callq  0xfffffffffffffffdd6
 32:   48 8b 43 18           mov    0x18(%rbx),%rax
 36:   48 8b 53 10           mov    0x10(%rbx),%rdx
 3a:   48 89 df             mov    %rbx,%rdi
 3d:   48                   rex.W
 3e:   89                   .byte 0x89
 3f:   42                   rex.X
```

Code starting with the faulting instruction
==========================================
```
0:   0f 0b                     ud2
  2:   e8 a4 fd ff ff           callq  0xfffffffffffffffddab
  7:   48 8b 43 18             mov    0x18(%rbx),%rax
  b:   48 8b 53 10             mov    0x10(%rbx),%rdx
  f:   48 89 df               mov    %rbx,%rdi
 12:   48                     rex.W
 13:   89                     .byte 0x89
 14:   42                     rex.X
```
Example decodecode output

All code
========

0:  e8 03 b7 f4 ff          callq  0xfffffffffff4b708
5:  49 8b 47 78             mov    0x78(%r15),%rax
9:  4c 8b 20                mov    (%rax),%r12
 c:  48 8d 58 f0             lea    -0x10(%rax),%rbx
10:  49 83 ec 10             sub    $0x10,%r12
14:  48 8d 43 10             lea    0x10(%rbx),%rax
18:  48 39 45 c8             cmp    %rax,-0x38(%rbp)
1c:  74 55                   je     0x73
1e:  48 8b 7b 08             mov    0x8(%rbx),%rdi
22:  83 bf 8c 00 00 00 00    cmpl   $0x0,0x8c(%rdi)
29:  74 02                   je     0x2d
2b:*  0f 0b                   ud2             <-- trapping instruction
2d:  e8 a4 fd ff ff          callq  0xfffffffffffffd6
32:  48 8b 43 18             mov    0x18(%rbx),%rax
36:  48 8b 53 10             mov    0x10(%rbx),%rdx
3a:  48 89 df                mov    %rbx,%rdi
3d:  48                      rex.W
3e:  89 .byte 0x89
3f:  42                      rex.X
BUG_ON(anon_vma-&gt;degree);

We skip the UD2 instruction if 0x8c(%rdi) equals zero → we trap if the value is non-zero

This suggests that RDI holds the struct anon_vma pointer and degree is at offset 0x8c

However, we can't determine the value that had been compared to zero in this case...
Example decodecode output

struct anon_vma_chain *avc;
list_for_each_entry_safe(avc, ...)
  struct anon_vma *anon_vma = avc->anon_vma;
  BUG_ON(anon_vma->degree);

Suggests that RBX holds the struct anon_vma_chain pointer avc and anon_vma member is at offset 0x8
Verifying structure offsets

- We can use pahole from dwarves package
  - May depend on GCC version, .config options
    - rwsem size depends on CONFIG_DEBUG_SPINLOCK, CONFIG_DEBUG_LOCK_ALLOC

```c
> pahole --hex -C anon_vma mm/vmscan.o
struct anon_vma {
  struct anon_vma *          root;                 /*     0   0x8 */
  struct rw_semaphore        rwsem;                /*   0x8  0x80 */
  /* --- cacheline 2 boundary (128 bytes) was 8 bytes ago --- */
  atomic_t                   refcount;             /*  0x88   0x4 */
  unsigned int               degree;               /*  0x8c   0x4 */
  struct anon_vma *          parent;               /*  0x90   0x8 */
  struct rb_root             rb_root;              /*  0x98   0x8 */
  /* size: 160, cachelines: 3, members: 6 */
  /* last cacheline: 32 bytes */
}
```
Example kernel oops

... 

[266.492815] Call Trace:
[266.492834]  [<ffffffff8111ba5d>] free_pgtables+0x8e/0xcc
[266.492873]  [<ffffffff8112253e>] exit_mmap+0x84/0x116
[266.492907]  [<ffffffff8103f789>] mmput+0x52/0xe9
[266.492940]  [<ffffffff81043918>] do_exit+0x3cd/0x9c9
[266.492975]  [<ffffffff8170c1ec>] _raw_spin_unlock_irq+0x2d/0x32
[266.493016]  [<ffffffff81044d7f>] do_group_exit+0x4c/0xc9
[266.493051]  [<ffffffff810022c4>] do_signal+0x28/0x5b1
[266.493123]  [<ffffffff8170ca0c>] sysret_signal+0x5/0x43
[266.493162]  [<ffffffff8104eb87>] get_signal+0x58f/0x5bc

The most important registers again, with higher printk level, or in case the details had scrolled away.

---[ end trace 02d28fe9b3de2e1a ]---

[266.493404] RIP  [<ffffffff81126630>] unlink_anon_vmas+0x102/0x159
[266.493447] RSP  [<fff8800916bb68>
[266.508877] Fixing recursive fault but reboot is needed!
Example kernel oops

Randomization to distinguish reports of same bug instance from separate instances.
Example kernel oops

The task was already exiting when it oopsed. In this case it's clearly graceful exit (from the backtrace), but it could be exiting due to previous oops. It's safer to leave task as zombie than to risk infinite oopses in the exit path.
How is stack unwinding implemented?

- Start at value of RSP and increment in a loop
  - Check if stack contains kernel text address
  - Print with translation to function name+offset
  - When RSP matches RBP + sizeof(long), consider address **reliable** (i.e. without “?”) and update RBP from the address it points to

- Not fully reliable, even with frame pointers
  - Cannot be relied upon functionally (live patching?)
  - Assembler functions now audited for missing frame pointers
    - Planned: runtime checks + DWARF validations
Stack Trace Example

```c
int c(int i) { return i; }
int b(int i) { return c(i); }
int a(int i) { return b(i); }
int main(int argc, char *argv[]) {
    return a(argc);
}
```

- Compile using gcc -O1 -m64 for AMD64
- Disassemble and single-step main() and a()
- Observe the stack
AMD64 Stack and Code Example

```
a:        pushq  %rbp
a+1:      movq   %rsp,%rbp
a+4:      movl   $0x0,%eax
a+9:      call   +0x2    <b>
a+0xe:    leave
a+0xf:    ret

main:     pushq  %rbp
main+1:   movq   %rsp,%rbp
main+4:   call   -0x2c    <a>
main+9:   leave
main+0xa: ret
```
Initial state

- No instructions executed
- Inherited stack pointer from main()'s caller

```
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     call   -0x2c <a>
main+9:     leave
main+0xa:   ret
```

rsp=0xffffffff7fffffff8
rbp=0xffffffff7fffffffc00

0xffffffff7fffffff8:  _start+0x6c
### AMD64 Stack and Code Example

- **Save previous frame pointer on the stack**

```assembly
main:       pushq  %rbp
main+1:     movq   %rsp,%rbp
main+4:     call   -0x2c <a>
main+9:     leave
main+0xa:   ret
```

```
0xffffffff7fffffff8f0:  0xffffffff7fffffff8f0
0xffffffff7fffffff8f8:  _start+0x6c
```

- `rsp=0xffffffff7fffffff8f0`
- `rbp=0xffffffff7fffffff8f0`
Establish new fixed frame pointer in RBP

- It points to where we saved the previous one

```assembly
main:    pushq  %rbp
main+1:  movq   %rsp,%rbp
main+4:  call   -0x2c   <a>
main+9:  leave
main+0xa: ret
```

```
0xffffffff7fffffff: _start+0x6c
0xffffffff7fffffff: 0xffffffff7fffffff8
rsp=0xffffffff7fffffff
rbp=0xffffffff7fffffff
```
Call a()

The argument is passed in RDI

```
main:    pushq  %rbp
main+1:  movq   %rsp,%rbp
main+4:  call   -0x2c    <a>
main+9:  leave
main+0xa: ret
```

rsp=0xffffffff7fffffffbe8
rbp=0xffffffff7fffffffbe8

0xffffffff7fffffffbe8: main+9
0xffffffff7fffffffbe8: _start+0x6c
**AMD64 Stack and Code Example**

```assembly
a:       pushq %rbp
a+1:     movq %rsp,%rbp
a+4:     movl $0x0,%eax
a+9:     call +0x2  <b>
a+0xe:   leave
a+0xf:   ret
```

- **Save the previous frame pointer to the stack**

```
0xfffffd7fffdffbe0:  0xfffffd7fffdffbf0
0xfffffd7fffdffbe8:  main+9
0xfffffd7fffdffbf0:  0xfffffd7fffdffc00
0xfffffd7fffdffbf8:  _start+0x6c
```

```
rsp=0xfffffffff7fffdffbe0
rbp=0xfffffffff7fffdffbf0
```
Establish new frame pointer in RBP

- It points to the address where the previous one is stored

\[
\begin{align*}
a: & \quad \text{pushq } \%rbp \\
a+1: & \quad \text{movq } \%rsp,\%rbp \\
a+4: & \quad \text{movl } 0x0,\%eax \\
a+9: & \quad \text{call } +0x2 \quad <b> \\
a+0xe: & \quad \text{leave} \\
a+0xf: & \quad \text{ret}
\end{align*}
\]
AMD64 Stack and Code Example

a:        pushq  %rbp
a+1:      movq   %rsp,%rbp
a+4:      movl   $0x0,%eax
a+9:      call   +0x2 <b>
a+0xe:    leave
a+0xf:    ret

- Zero EAX
  - Zero-extend to upper 32bits of RAX
    - Clears the whole RAX
  - Not needed

\[ \text{rsp}=0xfffffd7fffdfbbe0 \]
\[ \text{rbp}=0xfffffd7fffdfbbe0 \]
AMD64 Stack and Code Example

```
a:       pushq  %rbp
a+1:     movq   %rsp,%rbp
a+4:     movl   $0x0,%eax
a+9:     call   +0x2  <b>
a+0xe:   leave
a+0xf:   ret
```

- **Call b()**
  - The argument is still in RDI

```
rsp=0xffffffff7fffffff8
rbp=0xffffffff7fffffff8
0xffffffff7fffffff8: a+0xe
0xffffffff7fffffffbe0: 0xffffffff7fffffffbe0
0xffffffff7fffffffbe8: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
```

```
0xffffffff7fffffffbe8: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
```

```
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
0xffffffff7fffffffbe0: main+9
```
AMD64 Stack and Code Example

a:        pushq  %rbp
a+1:      movq   %rsp,%rbp
a+4:      movl   $0x0,%eax
a+9:      call   +0x2    <b>
a+0xe:    leave
a+0xf:    ret

• Stack Unwinding
  • rbp is a head of a linked list on the stack
  • each list member has function pointer at 8byte offset

rsp=0xfffffd7ffffdfbe0
rbp=0xfffffd7ffffdfbe0
AMD64 Stack and Code Example

a:     pushq  %rbp
a+1:   movq  %rsp,%rbp
a+4:   movl  $0x0,%eax
a+9:   call  +0x2  <b>
a+0xe:  leave
a+0xf:  ret

- Leaving a()
- `leave` is equivalent to
  - `mov rsp, rbp`
  - `pop rbp`

```
0xfffffd7fffdffbd8:  a+0xe
0xfffffd7fffdffbe0:  0xfffffd7fffdffbf0
0xfffffd7fffdffbe8:  main+9
0xfffffd7fffdffbf0:  0xfffffd7fffdfffc00
0xfffffd7fffdffbf8:  _start+0x6c
```

rsp=0xfffffd7fffdffbd8
rbp=0xfffffd7fffdffbe0
AMD64 Stack and Code Example

a: pushq %rbp
a+1: movq %rsp,%rbp
a+4: movl $0x0,%eax
a+9: call +0x2 <b>
a+0xe: leave
a+0xf: ret

Leaving a()

- mov rsp, rbp
- pop rbp
AMD64 Stack and Code Example

a: pushq %rbp
a+1: movq %rsp,%rbp
a+4: movl $0x0,%eax
a+9: call +0x2
a+0xe: leave
a+0xf: ret

- Leaving a()
  - mov rsp, rbp
  - pop rbp

rsp=0xffffffff7ffffffb8
rbp=0xffffffff7ffffffb0
AMD64 Stack and Code Example

a:     pushq  %rbp
a+1:   movq   %rsp,%rbp
a+4:   movl   $0x0,%eax
a+9:   call   +0x2 <b>
a+0xe: leave
a+0xf: ret

Leaving a()

- mov rsp, rbp
- pop rbp
- ret

rsp=0xffffffff7ffffdffb0f0
rbp=0xffffffff7ffffdffb0f0
How is stack unwinding implemented?

- For perf callgraph sampling, this would be slow
  - Therefore, fully rely on frame pointer walk there

- Alternative approach: use DWARF2 exception handler (EH) frame info
  - Patch in SUSE kernels, rejected upstream
  - Also not fully reliable, and more complex
What to do about the oops anyway?

- Fix it and become a kernel developer :)
  - Or report it, a good report is very welcome
    - You can get “Reported-by:” credit
- Guidelines are in ./REPORTING-BUGS
  - Try googling the contents of the report (the oops title and names of the functions...), chances are you are not the first one to see it
  - Check the kernel version – listed on kernel.org?
    - If not, try reproducing with one of those (i.e. latest)
    - On distro kernel? Reproduce with vanilla, or report to distro only
What to do about the oops anyway?

- Make sure the oops is whole (“cut here...”)
  - And that it's not only a secondary oops
- If possible, avoid tainting modules
- Identify which subsystem is likely responsible
  - Sending to LKML only is possible, but might get lost in the thousands of mails per day
  - Match functions/modules mentioned in backtrace to files, then `./scripts/get_maintainer.pl -f <file>`
    - Look for maintainers and specific mailing lists
What else can produce oops/panic?

- BUG_ON seen in the example – hard assertion
- Memory paging related faults – check CR2!
  - “BUG: unable to handle kernel paging request”
    - “… handle NULL pointer dereference” (when bad_addr < PAGE_SIZE)
- Corrupted page table
- Kernel trying to execute NX-protected page
- Kernel trying to execute userspace page (Intel SMEP)
- Failed bounds check in kernel mode (Intel MPX feature)
- General protection fault, unhandled double fault
Example: NULL pointer dereference

```
[  526.950444] BUG: unable to handle kernel NULL pointer dereference at 0000000000000048
[  526.950450] IP: [<ffffffff81137e81>] dequeue_hwpoisoned_huge_page+0x121/0x1c0
[  526.950457] PGD 135863067 PUD 1249a9067 PMD 0
[  526.950460] Oops: 0000 [#1] SMP
[  526.950464] CPU 0
[  526.950465] Modules linked in: (...)  
[  526.950506] Supported: Yes
[  526.950514] RIP: 0010:[<ffffffff81137e81>] [<ffffffff81137e81>]
  dequeue_hwpoisoned_huge_page+0x121/0x1c0
[  526.950523] RDX: 0000000000000000 RSI: 0000000000000000 RDI: ffffffff81e5fe08
[  526.950525] RBP: ffffea0003f8c000 R08: 0000000000001000 R09: 0000000000013a78
[  526.950529] R13: 0000000000000000 R14: ffffffff8136965e68 R15: ffffffff80136965ef8
[  526.950532] FS: 00007fca3e257700(0000) GS:fffffff88013fc0000(0000)
knlGS:0000000000000000
[  526.950535] CS: 0010 DS: 0000 ES: 0000 CR0: 000000000805003b
[  526.950542] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[  526.950546] DR3: 0000000000000000 DR6: 00000000ffff0ff0 DR7: 00000000000000400
```
Example: NULL pointer dereference

[ 526.950548] Process thugetlb_overco (pid: 6226, threadinfo ffff880136964000, task ffff8801356f2440)
[ 526.950551] Stack:
[ 526.950552]  ffffea0003f8c000 000000000122800 ffffea0003f8c000 ffffea0003f8115a3c2
[ 526.950556]  ffff880136965e68 fff880136965e68 ffffea0003f8c000 ffffea0003f8c000
[ 526.950560]  0000160000000000 0000000000000001 0000000000000065 fff880136965e6b
[ 526.950563] Call Trace:
[ 526.950571]  [<ffffffff8115a3c2>] soft_offline_huge_page+0x132/0x240
[ 526.950576]  [<ffffffff8115a5be>] soft_offline_page+0xee/0x330
[ 526.950580]  [<ffffffff8111e23e>] madvise_hwpoison+0x8e/0x120
[ 526.950584]  [<ffffffff8111e794>] sys_madvise+0x44/0x280
[ 526.950589]  [<ffffffff8146f3f2>] system_call_fastpath+0x16/0x1b
[ 526.950595]  [<00007fca3ddcd307>] 0x7fca3ddcd306
[ 526.950597] Code: 48 d3 e0 48 39 f0 74 0e 48 81 c2 68 70 00 00 48 39 fa 72 e6 31 d2 48
8b 45 00 48 c1 e8 36 48 39 c1 e0 04 48 8d 44 02 40 48 8d 70 08
8> [ 526.950611] 8b 40 08 48 8d 50 d8 48 39 c6 48 8b 4a 28 74 27 48 39 d5 74
[ 526.950618] RIP [<ffffffff81137e81>] dequeue_hwpoisoned_huge_page+0x121/0x1c0
[ 526.950622] RSP ffff880136965e48
[ 526.950623] CR2: 0000000000000048

The relevant subsystem will be typically the files containing the executing function (RIP) and the more immediate callers on the stack. There might be exceptions! Think a list manipulating function being passed NULL.
What else can produce oops/panic?

- Soft lockup
  - CPU spent 20s in kernel without reaching a schedule point
  - A warning, unless config/bootparam softlockup_panic enabled
    - Soft lockup can be harmless, so not good idea in production
- Hard lockup
  - CPU spent 10s with disabled interrupts
- Detection of both combines several generic mechanisms
  - High priority kernel watchdog thread updates soft lockup timestamp
  - hrtimer set to deliver periodic interrupts, increments hard lockup counter and wakes up the watchdog thread
  - NMI perf event checks the timestamps and thus knows if hrtimers interrupts were processed and if watchdog thread was scheduled
Example: soft lockup (32bit x86)

kernel: NMI watchdog: BUG: soft lockup - CPU#1 stuck for 23s! [kworker/u8:2:31788]
kernel: Modules linked in:
kernel: CPU: 1 PID: 31788 Comm: kworker/u8:2 Not tainted 3.18.0-rc3 #7
12/23/2009
kernel: Workqueue: khelper __call_usermodehelper
kernel: task: e0f35580 ti: c0f12000 task.ti: c0f12000
kernel: EIP: 0060:[<c40b82bd>] EFLAGS: 00000206 CPU: 1
kernel: EIP is at __zone_watermark_ok+0xd/0xa0
kernel: EAX: c4bcb1c0 EBX: 00001aca ECX: 00000472 EDX: 00000001
kernel: ESI: 00000001 EDI: c4bcb1c0 EBP: c0f13c4c ESP: c0f13c40
kernel: DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
kernel: CR0: 8005003b CR2: 0834af00 CR3: 04c6f000 CR4: 00040790
kernel: Stack:
kernel: 00001aca 00000000 c0f13ccc c0f13c68 c40b9511 00000000 00000000 00001aca
kernel: 00037400 c4bcb1c0 c0f13ca8 c40ce958 00000000 00000000 00000000 c4bcb3ec
kernel: 00000000 00000004 00000000 e0f35580 c0f13fec c4bcb1c0 00037400 c4bcb1c0
Example: soft lockup (32bit x86)

The problem was an infinite loop in this function.
Example: NULL pointer on SPARC

Unable to handle kernel NULL pointer dereference

```c
tsk->{mm,active_mm}->context = 0000000000000000
```

```c
swapper(0): Oops [#1]
CPU: 0 PID: 0 Comm: swapper Not tainted 3.16.0-4-sparc64 #1 Debian 3.16.7-2
```

```c
task: 0000000000a14470 ti: 0000000009fc000 task.ti: 0000000009fc000
```

```c
TSTATE: 0000004480e01601 TPC: 00000000004a36fc TNPC: 00000000004a3700 Y: 00000000 Not tainted
```

```c
TPC: <kstat_incr_irq_this_cpu+0x1c/0x60>
g0: 0000000000a161a0 g1: 0000000000a2eaf8 g2: 0000000000000001 g3: 0000000000000000
```

```c
g4: 0000000000a14470 g5: 0000000000000000 g6: 00000000009fc000 g7: 000000000000001
```

```c
o0: 0000000000000000 o1: 0000000000000000 o2: 0000000000000000 o3: 0000000000000020
```

```c
o4: 0000000000000000e o5: 0000000000a8f000 sp: 0000000009fef31 ret_pc: 00000000004a36f4
```

```c
RPC: <kstat_incr_irq_this_cpu+0x14/0x60>
l0: 0000000000b30320 l1: 0000000000b0f650 l2: 0000000000000001 l3: 0000000000b30448
```

```c
l4: 0000000000b10b18 l5: 0000000000b30300 l6: 0000000000b30310 l7: 0000000000000000
```

```c
i0: 0000000000000000 i1: 0000000000b0f648 i2: 0000000000b30310 i3: 0000000000000000e
```

```c
i4: 0000000000b3000 i5: 0000000000b30000 i6: 00000000009fefe1 i7: 00000000008aaf58
```

```c
I7: <timer_interrupt+0x38/0xa0>
```
Example: NULL pointer on SPARC

Call Trace:
[00000000008aaf58] timer_interrupt+0x38/0xa0
[00000000004209d4] tl0_irq14+0x14/0x20
[00000000004db6c0] touch_softlockup_watchdog+0x0/0x20
[0000000000ab8568] memblock_virt_alloc_try_nid+0x84/0x94
[0000000000ab9754] sparse_init+0x1c/0x224
[0000000000abcb8] paging_init+0xd80/0xeb0
[0000000000aa7068] setup_arch+0x2d4/0x718
[0000000000aa4668] start_kernel+0x78/0x420
[0000000000aa6d84] start_early_boot+0x1bc/0x1cc
[0000000000897410] tlb_fixup_done+0x4c/0x5c
[0000000000000000] (null)

Disabling lock debugging due to kernel taint
Caller[00000000008aaf58]: timer_interrupt+0x38/0xa0
Caller[00000000004209d4]: tl0_irq14+0x14/0x20
Caller[00000000008984e4]: panic+0x1c8/0x208
Caller[0000000000ab8568]: memblock_virt_alloc_try_nid+0x84/0x94
Caller[0000000000ab9754]: sparse_init+0x1c/0x224
Caller[0000000000abcb8]: paging_init+0xd80/0xeb0
Caller[0000000000aa7068]: setup_arch+0x2d4/0x718
Caller[0000000000aa4668]: start_kernel+0x78/0x420
Caller[0000000000aa6d84]: start_early_boot+0x1bc/0x1cc
Caller[0000000000897410]: tlb_fixup_done+0x4c/0x5c
Caller[0000000000000000]: (null)

Instruction DUMP: 92100018 40076593 901222f8 <c45a2048> 030028b1 c6008000 8600e001 c6208000 c4586160
---[ end Kernel panic - not syncing: Aiee, killing interrupt handler! ]---
What else can produce oops/panic?

- Hung task check
  - "INFO: task ... blocked for more than 120 seconds"
  - khungtaskd periodically processes tasks in uninterruptible sleep and checks if their switch count changed

- RCU stall detector
  - Detects when RCU grace period is too long (21s)
    - CPU looping in RCU critical section or disabled interrupts, preemption or bottom halves, no scheduling points in non-preempt kernels
    - RT task preempting non-RT task in RCU critical section

- Several other related to debugging config options
Kernel debugging config options

- Kernel can be built with additional debugging options enabled
  - Extra checks that can catch errors sooner, or provide extra information, at the cost of CPU and/or memory overhead
  - Can also hide errors such as race conditions...
- Many of them under “Kernel hacking” in make menuconfig
  - Others placed in the given subsystem/driver
Kernel debugging config options (VM)

- **DEBUG_VM** – enable VM BUG ON( cond ) checks
- **PAGE_OWNER** – track who allocated which pages in order to find a memory leak
- **DEBUG_PAGEALLOC** – unmap (or poison) pages after they are freed
- **DEBUG_SLAB** – detect some cases of double free, or use-after-free (by poisoning)
  - SLUB_DEBUG variant can enable/disable debugging in runtime
- **DEBUG_KMEMLEAK** – detect leaks with a conservative garbage collection based algorithm
- **KASAN** – Find out of bounds accesses and use-after-free bugs at the cost of 1/8 memory and 3x slower performance
Kernel debugging config options

- DEBUG_STACKOVERFLOW – check if random corruption involving struct thread_info is caused by too deep call chains
- DEBUG_SPINLOCK and others for different locks – catch missing init, freeing of live locks, some deadlocks
- LOCK_STAT – for lock contention, perf lock
- PROVE_LOCKING - “lockdep” mechanism for online proving that deadlocks cannot happen and report that deadlock can occur before it actually does
Thank you!

- Crash dump analysis course
  - http://d3s.mff.cuni.cz/cda

- We are hiring!
  - https://www.suse.com/careers/