entry_*.S

A carefree stroll through kernel entry code

Borislav Petkov
SUSE Labs
bp@suse.de

Reasons for entry into the kernel

- System calls (64-bit, compat, 32-bit)
- Interrupts (NMIs, APIC, timer, IPIs...)
  - software: INT 0x0-0xFF, INT3, …
  - external (hw-generated): CPU-ext logic, async to insn exec
- Architectural exceptions (sync vs async)
  - faults: precise, reported **before** faulting insn => restartable (#GP,#PF)
  - traps: precise, reported **after** trapping insn (#BP,#DB-both)
  - aborts: imprecise, not reliably restartable (#MC, unless MCG_STATUS.RIPV)

Intr/Ex entry

- IDT, int num index into it (256 vectors); all modes need an IDT
- If handler has a higher CPL, switch stacks
- A picture is always better:

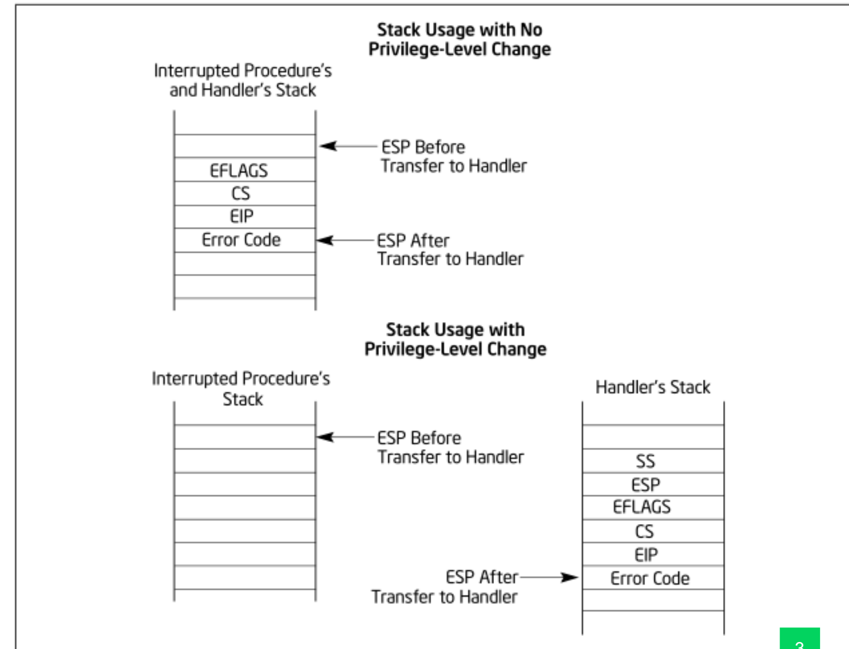
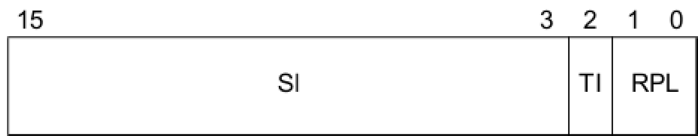
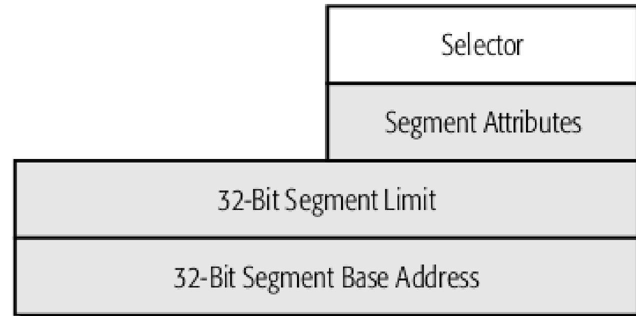


Figure 6-5. Stack Usage on Transfers to Interrupt and Exception Handling Routines

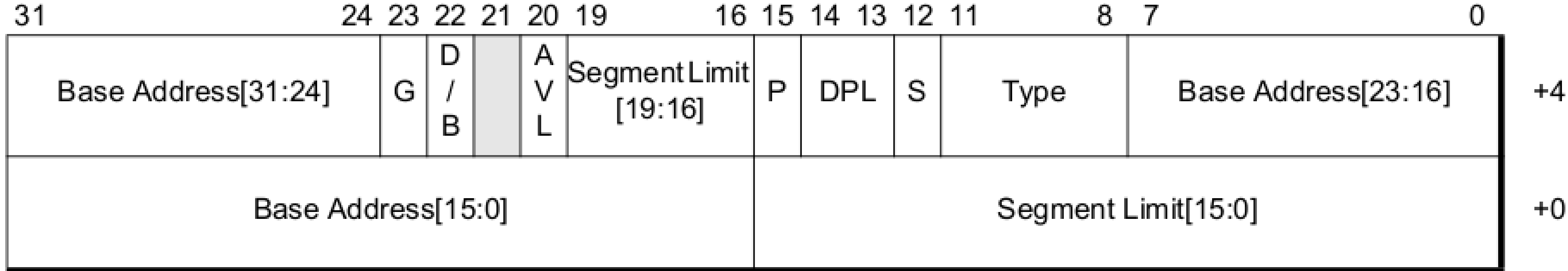
45sec guide to Segmentation

- Continuous range at an arbitrary position in VA space
- Segments described by segment descriptors
- … selected by segment selectors
- … by indexing into segment descriptor tables (GDT,LDT,IDT,...)
- … and loaded by the hw into segment registers:
  - user: CS,DS,{E,F,G}S,SS
  - system: GDTR,LDTR,IDTR,TR (TSS)





A couple more seconds of Segmentation



- L (bit 21) new long mode attr: 1=long mode, 0=compat mode
- D (bit 22): default operand and address sizes
  - legacy: D=1b – 32bit, D=0b – 16bit
  - long mode: D=0b – 32-bit, L=1,D=1 reserved for future use
- G (bit 23): granularity: G=1b: seg limit scaled by 4K
- DPL: Descriptor Privilege Level of the segment

Legacy syscalls

- Call OS through gate descriptor (call, intr, trap or task gate)
- Overhead due to segment-based protection:
  - load new selector + desc into segment register (even with flat model due to CS/SS reloads during privilege levels switches)
  - Selectors and descriptors are in proper form
  - Descriptors within bounds of descriptor tables
  - Gate descs reference the appropriate segment descriptors
  - Caller, gate and target privs are sufficient for transfer to take place
  - Stack created by the call is sufficient for the transfer

Syscalls, long mode

- SYSCALL + SYSRET
- ¼th of the legacy CALL/RET clocks
- Flat mem model with paging (CS.base=0, ignore CS.limit)
- Load predefined CS and SS
- Eliminate a bunch of unneeded checks
  - Assume CS.base, CS.limit and attrs are unchanged, only CPL changes
  - Assume SYSCALL target CS.DPL=0, SYSRET target CS.DPL=3 (SYSCALL sets CPL=0)

Syscalls, long mode

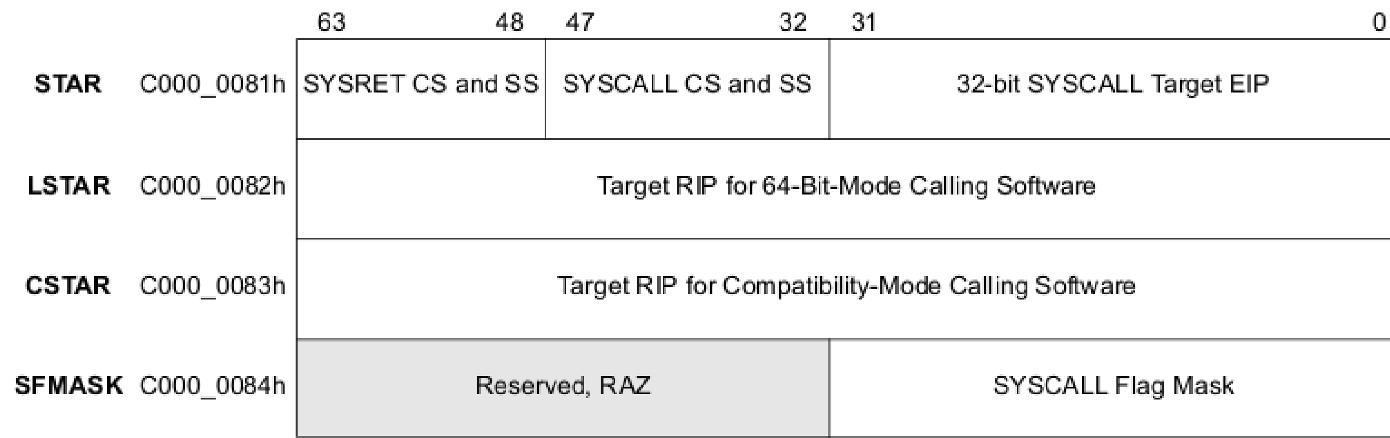


Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs

- Targets and CS/SS selectors configured through MSRs
- **L**ong/**C**ompat mode **S**yscall **T**arget **A**dd**R**ess
- SFMASK: rFLAGS to be cleared during SYSCALL

```
void syscall_init(void)
{
        wrmsr(MSR_STAR, 0, (__USER32_CS << 16) | __KERNEL_CS);
        wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);

#ifdef CONFIG_IA32_EMULATION
        wrmsrl(MSR_CSTAR, (unsigned long)entry_SYSCALL_compat);
```

SYSCALL, long mode

- %rcx = %rip + sizeof(SYSCALL==0f 05) = %rip + 2 (i.e., next_RIP)
- %rip = MSR_LSTAR(0xC000_0082) (MSR_CSTAR in compat mode)
- %r11 = rFLAGS & ~RF (so that SYSRET can reenable insn #DB)
  - RF: resume flag, cleared by CPU on every insn retire
  - RF=1b => #DB for insn breakpoints are disabled until insn retires

SYSCALL, long mode

- CS.sel = MSR_STAR.SYSCALL_CS & 0xfffc /* enforce RPL=0 */
- [47:32] = 0x10 which is __KERNEL_CS, i.e. 2*8
- CS.L=1b, CS.DPL=0b, CS.R=1b /* read/exec, 64-bit mode */
- CS.base = 0x0, CS.limit = 0xFFFF_FFFF /* seg in long mode */
- SS.sel = MSR_STAR.SYSCALL_CS + 8 /* sels are hardcoded, i.e., this is __KERNEL_DS */
- SS.W=1b, SS.E=0b /* r/w segment, expand-up */
- SS.base = 0x0, SS.limit = 0xFFFF_FFFF

```
[root@pd: ~ $> rdmsr -ac 0xc0000081 | uniq
0x23001000000000
```

```
#define GDT_ENTRY_KERNEL32_CS 1
#define GDT_ENTRY_KERNEL_CS 2
#define GDT_ENTRY_KERNEL_DS 3
```

SYSCALL, long mode

- RFLAGS &= ~MSR_SFMASK (0xC000_0084): 0x47700
  - TF (Trap Flag): do not singlestep the syscall from luserspace
  - IF (Intr Flag): disable interrupts, we do enable them a little later
  - DF (Dir Flag): reset direction of string processing insns (no need for CLD)
  - IOPL >= CPL for kernel to exec IN(S),OUT(S), thus reset it to 0 as we're in CPL0
  - NT: IRET reads NT to know whether current task is nested
  - AC: disable alignment checking (no need for CLAC)
- rFLAGS.RF=0
- CPL = 0

```
/* Flags to clear on syscall */
wrmsrl(MSR_SYSCALL_MASK,
       X86_EFLAGS_TF|X86_EFLAGS_DF|X86_EFLAGS_IF|
       X86_EFLAGS_IOPL|X86_EFLAGS_AC|X86_EFLAGS_NT);
```

SYSCALL, long mode/kernel

- entry_SYSCALL_64:
- Up to 6 args in registers:
  - RAX: syscall #
  - RCX: return address
  - R11: saved rFLAGS & ~RF
  - RDI, RSI, RDX, **R10**, R8, R9: args
  - for comparison with C ABI: RDI, RSI, RDX, **RCX**, R8, R9
    - A bit later we do movq %r10, %rcx to get it to conform to C ABI
  - R12-R15, RBP, RBX: callee preserved

SYSCALL, long mode/kernel

- Example: int stat(const char *pathname, struct stat *buf)
- %rax: syscall #, stat() → sys_newstat()
- %rip = entry_SYSCALL_64
- %rcx = caller RIP, i.e. next_RIP
- %r11 = rFLAGS
- %rdi = *pathname
- %rsi = *buf
- CS=0x10
- SS=0x18

```
RAX=0000000000000004 RBX=0000000000000000 RCX=00007f4c88204f35 RDX=00007ffd445ef0d0
RSI=00007ffd445ef0d0 RDI=0000000000407199 RBP=00007ffd445ef458 RSP=00007ffd445eef98
R8 =000000000000ffff R9 =0000ffef00000000 R10=0000000000000000 R11=0000000000000246
R12=0000000000000001 R13=0000000000000001 R14=0000000000000000 R15=ffffffffffffffff
RIP=ffffffff816fff40 RFL=00000046 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 0000000000000000 ffffffff 00800000
CS =0010 0000000000000000 ffffffff 00a09b00 DPL=0 CS64 [-RA]
SS =0018 0000000000000000 ffffffff 00c09300 DPL=0 DS   [-WA]
DS =0000 0000000000000000 ffffffff 00800000
FS =0000 00007f4c88b52800 ffffffff 00800000
GS =0000 0000000000000000 ffffffff 00800000
LDT=0000 0000000000000000 0000ffff 00000000
TR =0040 ffff88007edd2e40 00002087 00008b00 DPL=0 TSS64-busy
GDT=     ffff88007ec09000 0000007f
IDT=     fffffffff57c000 00000fff
CR0=80050033 CR2=0000000000191fbb0 CR3=000000007ba9d000 CR4=000406f0
DR0=0000000000000000 DR1=0000000000000000 DR2=0000000000000000 DR3=0000000000000000
DR6=00000000ffff0ff0 DR7=0000000000000400
EFER=0000000000000d01
```

SYSCALL, long mode/kernel

- SWAPGS_UNSAFE_STACK
- Load kernel data structures so that we can switch stacks and save user regs
- Swap GS shadow (MSR_KERNEL_GS_BASE: 0xC000_0102) with GS.base (hidden portion) (MSR_GS_BASE: 0xC000_0101)
- SWAPGS doesn't require GPRs or memory operands
- Before SWAPGS:

```
FS =0000 00007f4c88b52800 ffffffff 00800000
GS =0000 0000000000000000 ffffffff 00800000
LDT=0000 0000000000000000 0000ffff 00000000
```

- After:

```
FS =0000 00007f4c88b52800 ffffffff 00800000
GS =0000 ffff88007ec00000 ffffffff 00800000
LDT=0000 0000000000000000 0000ffff 00000000
```

- dmesg:

```
[    0.000000] setup_percpu: NR_CPUS:8 nr_cpumask_bits:8 nr_cpu_ids:1 nr_node_ids:1
[    0.000000] PERCPU: Embedded 481 pages/cpu @ffff88007ec00000 s1929880 r8192 d32104 u2097152
[    0.000000] pcpu-alloc: s1929880 r8192 d32104 u2097152 alloc=1*2097152
[    0.000000] pcpu-alloc: [0] 0
```

SYSCALL, long mode/kernel

- movq %rsp, PER_CPU_VAR(rsp_scratch) →
  mov %rsp, %gs:0xb7c0
- Let's see what's there:

```
[boris@pd: ~/kernel/linux> readelf -a vmlinux | grep rsp_scratch
 77378: 000000000000b780     8 OBJECT  GLOBAL DEFAULT   17 rsp_scratch
```

- per_cpu area starts at 0xffff_8800_7ec0_0000
- So what's at 0xffff_8800_7ec0_b780?

```
(gdb) x/g 0xffff88007ec0b780
0xffff88007ec0b780:     0x00007ffff5350d28
```

- That must be the user stack pointer:

```
(qemu) info registers
RAX=00000000000000ca RBX=000000000066eb98 RCX=00007ffff79b0755 RDX=0000000000000001
RSI=0000000000000085 RDI=000000000067f5e4 RBP=0000000000000001 RSP=00007ffff5350d28
R8 =000000000067f5e0 R9 =0000000004000001 R10=0000000000000001 R11=0000000000000283
R12=000000000065e460 R13=0000000000000001 R14=00000000000000e8 R15=000000000066eb80
RIP=ffffffff81700ec3 RFL=00000083 [--S---C] CPL=0 II=0 A20=1 SMM=0 HLT=0
```

-
-
- Ok, persuaded! :-)

SYSCALL, long mode/kernel

- movq PER_CPU_VAR(cpu_current_top_of_stack), %rsp
- cpu_current_top_of_stack is:
  - cpu_tss + OFFSET(TSS_sp0,tss_struct, x86_tss.sp0)
  - i.e., CPL0 stack ptr in TSS
- tss_struct contains CPL[0-3] stacks, io perms bitmap and temporary SYSENTER stack
- TRACE_IRQS_OFF: CONFIG_TRACE_IRQFLAGS - trace when we enable and disable IRQs
- #define TRACE_IRQS_OFF call trace_hardirqs_off_thunk;
- THUNKing: stash callee-clobbered regs before calling C functions

SYSCALL, long mode/kernel

- Construct user pt_regs on stack. Hand them down to helper functions, see later
- __USER_DS: user stack, sel must be between 32- and 64-bit CS
- user RSP we just saved in rsp_scratch
- __USER_CS: user code segment's selector
- -ENOSYS: non-existent syscall
- Prepare full IRET frame in case we have to IRET

```
/* Construct struct pt_regs on stack */
pushq   $__USER_DS                  /* pt_regs->ss */
pushq   PER_CPU_VAR(rsp_scratch)    /* pt_regs->sp */
pushq   %r11                        /* pt_regs->flags */
pushq   $__USER_CS                  /* pt_regs->cs */
pushq   %rcx                        /* pt_regs->ip */
pushq   %rax                        /* pt_regs->orig_ax */
pushq   %rdi                        /* pt_regs->di */
pushq   %rsi                        /* pt_regs->si */
pushq   %rdx                        /* pt_regs->dx */
pushq   %rcx                        /* pt_regs->cx */
pushq   $-ENOSYS                    /* pt_regs->ax */
pushq   %r8                         /* pt_regs->r8 */
pushq   %r9                         /* pt_regs->r9 */
pushq   %r10                        /* pt_regs->r10 */
pushq   %r11                        /* pt_regs->r11 */
sub     $(6*8), %rsp                /* pt_regs->bp, bx, r12-15 not saved */
```

IRET frame

Always push SS to allow return to compat mode (SS ignored in long mode).

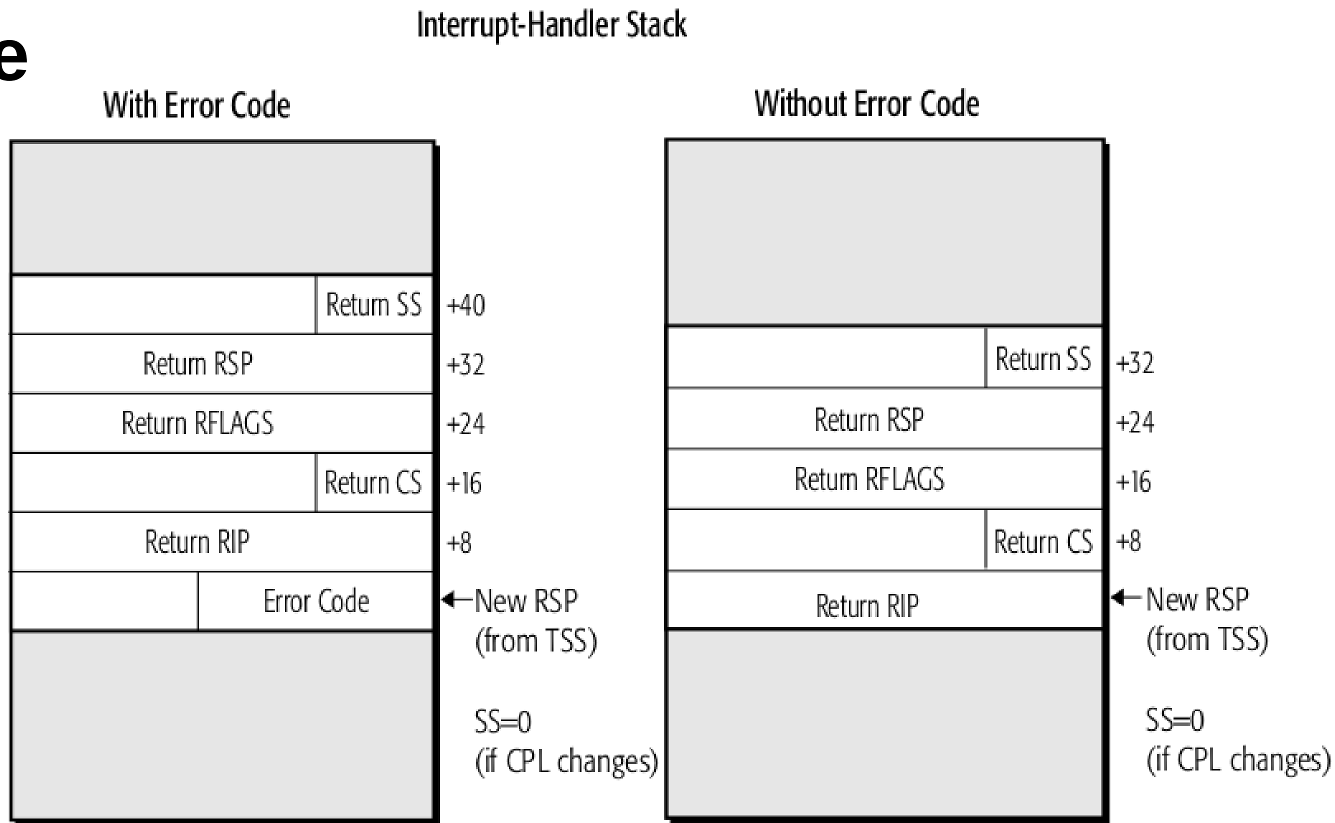


Figure 8-14. Long-Mode Stack After Interrupt—Higher Privilege

SYSCALL, long mode/kernel

```
testl	$_TIF_WORK_SYSCALL_ENTRY | _TIF_ALLWORK_MASK,
	ASM_THREAD_INFO(TI_flags, %rsp, SIZEOF_PTREGS)
```

- ASM_THREAD_INFO: get the offset to thread_info->flags on the bottom of the kernel stack
- test if we need to do any work on syscall entry:
  - TIF_SYSCALL_TRACE: ptrace(PTRACE_SYSCALL, …), f.e., examine syscall args of tracee
  - TIF_SYSCALL_EMU: ptrace(PTRACE_SYSEMU, …), UML emulates tracee's syscalls

SYSCALL, long mode/kernel

- TIF_SYSCALL_AUDIT: syscall auditing, pass args to auditing framework, see CONFIG_AUDITSYSCALL and userspace tools
- TIF_SECCOMP: secure computing. Syscalls filtering with BPFs, see Documentation/prctl/seccomp_filter.txt
- TIF_NOHZ: used in context tracking, eg. userspace ext. RCU
- TIF_ALLWORK_MASK: all TIF bits [15-0] for pending work are in the LSW

- Thus, if any work needs to be done on SYSCALL entry, we jump to the slow path

SYSCALL, long mode/kernel

- TRACE_IRQS_ON: counterpart to *OFF with the thunk
- ENABLE_INTERRUPTS: wrapper for paravirt, plain STI on baremetal
- __SYSCALL_MASK == ~__X32_SYSCALL_BIT:
  - share syscall table with X32
  - __X32_SYSCALL_BIT is bit 30; userspace sets it if X32 syscall
  - we clear it before we look at the system call number
  - see fca460f95e928

```
entry_SYSCALL_64_fastpath:
	/*
	 * Easy case: enable interrupts and issue the syscall.  If the syscall
	 * needs pt_regs, we'll call a stub that disables interrupts again
	 * and jumps to the slow path.
	 */
	TRACE_IRQS_ON
	ENABLE_INTERRUPTS(CLBR_NONE)
#if __SYSCALL_MASK == ~0
	cmpq	$__NR_syscall_max, %rax
#else
	andl	$__SYSCALL_MASK, %eax
	cmpl	$__NR_syscall_max, %eax
#endif
	ja	1f				/* return -ENOSYS (already in pt_regs->ax) */
	movq	%r10, %rcx
```

SYSCALL, long mode/kernel

- RAX contains the syscall number, index into the sys_call_table
- Some syscalls need full pt_regs and we end up calling stubs:
  `__SYSCALL_64(15, sys_rt_sigreturn, ptregs) → ptregs_sys_rt_sigregurn`
- Stub puts real syscall (sys_rt_sigreturn()) addr into %rax and calls stub_ptregs_64
- Check we're on the fast path by comparing ret addr to label below
- If so, we disable IRQs and jump to entry_SYSCALL64_slow_path
- Slow path saves extra regs for a full pt_regs and calls do_syscall_64():

```
if (likely((nr & __SYSCALL_MASK) < NR_syscalls)) {
        regs->ax = sys_call_table[nr & __SYSCALL_MASK](
                regs->di, regs->si, regs->dx,
                regs->r10, regs->r8, regs->r9);
}
```

```
        /*
         * This call instruction is handled specially in stub_ptregs_64.
         * It might end up jumping to the slow path.  If it jumps, RAX
         * and all argument registers are clobbered.
         */
        call    *sys_call_table(, %rax, 8)
.Lentry_SYSCALL_64_after_fastpath_call:

        movq    %rax, RAX(%rsp)
```

SYSCALL, long mode/kernel

- Retest if we need to do some **exit** work with IRQs off. If not
  - check locks are held before returning to userspace for lockdep (thunked)
  - mark IRQs on
  - restore user RIP for SYSRET
  - rFLAGS too
  - remaining regs
  - user stack
  - SWAPGS
  - … and finally SYSRET!

```
/*
 * If we get here, then we know that pt_regs is clean for SYSRET64.
 * If we see that no exit work is required (which we are required
 * to check with IRQs off), then we can go straight to SYSRET64.
 */
DISABLE_INTERRUPTS(CLBR_NONE)
TRACE_IRQS_OFF
testl   $_TIF_ALLWORK_MASK, ASM_THREAD_INFO(TI_flags, %rsp, SIZEOF_PTREGS)
jnz     1f

LOCKDEP_SYS_EXIT
TRACE_IRQS_ON           /* user mode is traced as IRQs on */
movq    RIP(%rsp), %rcx
movq    EFLAGS(%rsp), %r11
RESTORE_C_REGS_EXCEPT_RCX_R11
movq    RSP(%rsp), %rsp
USERGS_SYSRET64
```

SYSRET, long mode

- SYSCALL counterpart, low-latency return to userspace
- CPL0 insn, #GP otherwise
- CPL=3, regardless of MSR_STAR[49:48] (SYSRET_CS)
- Can return to 2½ modes depending on operand size
- 64-bit mode if operand size is 64-bit (EFER.LMA=1b, CS.L=1b)
  - CS.sel = MSR_STAR.SYSRET_CS + 16
  - CS.attr = 64-bit code, DPL3
  - RIP = RCX

```
 * GDT layout to get 64-bit SYSCALL/SYSRET support right. SYSRET hardcodes
 * selectors:
 *
 *   if returning to 32-bit userspace: cs = STAR.SYSRET_CS,
 *   if returning to 64-bit userspace: cs = STAR.SYSRET_CS+16,
 *
 * ss = STAR.SYSRET_CS+8 (in either case)
 *
 * thus USER_DS should be between 32-bit and 64-bit code selectors:
 */
#define GDT_ENTRY_DEFAULT_USER32_CS     4
#define GDT_ENTRY_DEFAULT_USER_DS       5
#define GDT_ENTRY_DEFAULT_USER_CS       6
```

SYSRET, long mode

- 32-bit (compat) mode, operand-size 32-bit (LMA=1, CS.L=0)
  - CS.sel = MSR_STAR.SYSRET_CS
  - CS.attr = 32-bit code, DPL3
  - RIP = ECX (zero-extended to a 64-bit write)
- For both modes: rFLAGS = R11 & ~(RF | VM)
  - reenable #DB
  - disable virtual 8086 mode

SYSRET, long mode

- 32-bit legacy prot mode: CS.L=0b, CS.D=1b
  - CS = MSR_STAR.SYSRET_CS
  - CS.attr = 32-bit code, DPL=3
  - RIP = ECX
  - rFLAGS.IF=1b
  - CPL=3
- In all 2½ cases:
  - SS.sel = MSR_STAR.SYSRET_CS + 8
  - CS.base = 0x0, CS.limit = 0xFFFF_FFFF

SYSRET, long mode

- SYSRET.CS = 0x23 = GDT_ENTRY_DEFAULT_USER32_CS*8 + 3 = 4*8 + 3

```
[root@pd: ~ $> rdmsr -ac 0xc0000081 | uniq
0x23001000000000
```

SYSCALL, long mode/kernel

- Looks like we need to do some exit work, go the slow path
- … raise(3) will trigger this because of TIF_SIGPENDING
- SAVE_EXTRA_REGS: stash callee-preserved R12-R15, RBP, RBX
- move pt_regs on stack ptr for arg of syscall_return_slowpath() which...
  - does some sanity-checking
  - does syscall exit work (tracing/auditing/...)
  - rejoins return path

```
/*
 * The fast path looked good when we started, but something changed
 * along the way and we need to switch to the slow path.  Calling
 * raise(3) will trigger this, for example.  IRQs are off.
 */
TRACE_IRQS_ON
ENABLE_INTERRUPTS(CLBR_NONE)
SAVE_EXTRA_REGS
movq    %rsp, %rdi
call    syscall_return_slowpath /* returns with IRQs disabled */
jmp     return_from_SYSCALL_64
```

SYSCALL, opportunistic SYSRET

- See 2a23c6b8a9c4 ("x86_64, entry: Use sysret to return to userspace when possible")
- IRET is damn slow; most syscalls don't touch pt_regs
- Even with exit work pending, we can try to avoid IRET-ting and try SYSRET → 80ns gain in syscall overhead
- Conditions we test:
  - RCX==RIP? Did the slowpath reroute us somewhere else instead of next_RIP
  - RIP(%rsp) == Return RIP in IRET frame

```
/*
 * Try to use SYSRET instead of IRET if we're returning to
 * a completely clean 64-bit userspace context.
 */
movq    RCX(%rsp), %rcx
movq    RIP(%rsp), %r11
cmpq    %rcx, %r11                      /* RCX == RIP */
jne     opportunistic_sysret_failed
```

SYSCALL, opportunistic SYSRET

- __VIRTUAL_MASK_SHIFT = 47
- 0x0000_7FFF_FFFF_FFFF – highest user address
- Do canonicality check: zaps non-canonical bits
- If it changed, fail SYSRET instead of getting pwned
- No such check on AMD

```
/*
 * On Intel CPUs, SYSRET with non-canonical RCX/RIP will #GP
 * in kernel space.  This essentially lets the user take over
 * the kernel, since userspace controls RSP.
 *
 * If width of "canonical tail" ever becomes variable, this will need
 * to be updated to remain correct on both old and new CPUs.
 */
.ifne __VIRTUAL_MASK_SHIFT - 47
.error "virtual address width changed -- SYSRET checks need update"
.endif

/* Change top 16 bits to be the sign-extension of 47th bit */
shl     $(64 - (__VIRTUAL_MASK_SHIFT+1)), %rcx
sar     $(64 - (__VIRTUAL_MASK_SHIFT+1)), %rcx

/* If this changed %rcx, it was not canonical */
cmpq    %rcx, %r11
jne     opportunistic_sysret_failed
```

SYSCALL, opportunistic SYSRET

- Comment explains it all:
- Except the trap shadow:
- STI with IF=0
  - one insn shadow, INTR happens in the caller
- IRET with TF/RF
  - #DB realized with 1 insn shadow

```
/*
 * SYSCALL clears RF when it saves rFLAGS in R11 so SYSRET cannot
 * restore RF properly. If the slowpath sets it for whatever reason, we
 * need to restore it correctly.
 *
 * SYSRET can restore TF, but unlike IRET, restoring TF results in a
 * trap from userspace immediately after SYSRET.  This would cause an
 * infinite loop whenever #DB happens with register state that satisfies
 * the opportunistic SYSRET conditions.  For example, single-stepping
 * this user code:
 *
 *           movq      $stuck_here, %rcx
 *           pushfq
 *           popq %r11
 *   stuck_here:
 *
 * would never get past 'stuck_here'.
 */
testq   $(X86_EFLAGS_RF|X86_EFLAGS_TF), %r11
jnz     opportunistic_sysret_failed

/* nothing to check for RSP */
```

SYSCALL, opportunistic SYSRET

- Finally check SS
- We win
- Restore C user regs
- Restore user stack ptr
- SWAPGS; SYSRET

```
	cmpq	$__USER_DS, SS(%rsp)		/* SS must match SYSRET */
	jne	opportunistic_sysret_failed

	/*
	 * We win! This label is here just for ease of understanding
	 * perf profiles. Nothing jumps here.
	 */
syscall_return_via_sysret:
	/* rcx and r11 are already restored (see code above) */
	RESTORE_C_REGS_EXCEPT_RCX_R11
	movq	RSP(%rsp), %rsp
	USERGS_SYSRET64
```

SYSCALL, IRET

```
opportunistic_sysret_failed:
	SWAPGS
	jmp	restore_c_regs_and_iret
END(entry_SYSCALL_64)
```

- opportunistic SYSRET failed, do IRET
- SWAPGS to user before jumping to IRET label: shared path
- We did restore callee-clobbered R12-R15,RBX,RBP earlier
- Restore remaining C regs
- Remove pt_regs from stack, leave IRET frame: SUB -(15*8+8), %rsp
  - +8: kill syscall# too, IRET frame with error code
- paravirt wrapper, `jmp native_iret` on baremetal

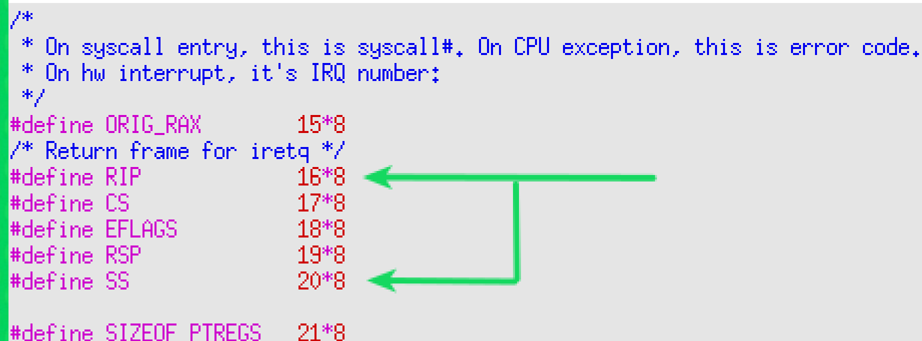
```
/*
 * At this label, code paths which return to kernel and to user,
 * which come from interrupts/exception and from syscalls, merge.
 */
GLOBAL(restore_regs_and_iret)
	RESTORE_EXTRA_REGS
restore_c_regs_and_iret:
	RESTORE_C_REGS
	REMOVE_PT_GPREGS_FROM_STACK 8
	INTERRUPT_RETURN
```

ESPFIX

- When we return to a 16-bit stack segment:
  - IRET restores only the lower word of rSP
  - causing a leak of the upper word with kernel stack contents
- We fix this with per-CPU ministacks of 64B (cacheline sized), mapped 2^16 times (128K max CPUs), 64K apart (stride jumps over [15:0])
  - on IRET, we copy IRET frame to the ministack and use that alias for luserspace
  - ministacks are RO-mapped so that a #GP during IRET gets promoted to a #DF: an IST-exception with its own stack
  - we then do the fixup in the #DF handler

ESPFIX

- See 3891a04aafd6 ("x86-64, espfix: Don't leak bits 31:16 of %esp returning to 16-bit stack")
- Test SS.TI=1b: are we returning to a SS in the LDT, i.e., a task's private SS
- SS-RIP because we have only IRET frame on the stack now




```
ENTRY(native_iret)
        /*
         * Are we returning to a stack segment from the LDT?  Note: in
         * 64-bit mode SS:RSP on the exception stack is always valid.
         */
#ifdef CONFIG_X86_ESPFIX64
        testb   $4, (SS-RIP)(%rsp)
        jnz     native_irq_return_ldt
#endif

.global native_irq_return_iret
native_irq_return_iret:
        /*
         * This may fault.  Non-paranoid faults on return to userspace are
         * handled by fixup_bad_iret.  These include #SS, #GP, and #NP.
         * Double-faults due to espfix64 are handled in do_double_fault.
         * Other faults here are fatal.
         */
        iretq
```

ESPFIX

- SWAPGS to kernel for percpu vars
- Move the writable espfix_waddr stack address into RDI
- Copy IRET frame there
- Clear [15:0] of RSP
- OR in the RO espfix_stack address
- SWAPGS to user
- Stick stack pointer into RSP
- IRET

```
#ifdef CONFIG_X86_ESPFIX64
native_irq_return_ldt:
        pushq   %rax
        pushq   %rdi
        SWAPGS
        movq    PER_CPU_VAR(espfix_waddr), %rdi
        movq    %rax, (0*8)(%rdi)               /* RAX */
        movq    (2*8)(%rsp), %rax               /* RIP */
        movq    %rax, (1*8)(%rdi)
        movq    (3*8)(%rsp), %rax               /* CS */
        movq    %rax, (2*8)(%rdi)
        movq    (4*8)(%rsp), %rax               /* RFLAGS */
        movq    %rax, (3*8)(%rdi)
        movq    (6*8)(%rsp), %rax               /* SS */
        movq    %rax, (5*8)(%rdi)
        movq    (5*8)(%rsp), %rax               /* RSP */
        movq    %rax, (4*8)(%rdi)
        andl    $0xffff0000, %eax
        popq    %rdi
        orq     PER_CPU_VAR(espfix_stack), %rax
        SWAPGS
        movq    %rax, %rsp
        popq    %rax
        jmp     native_irq_return_iret
#endif
END(common_interrupt)
```

To be continued...

References

Presentation contains snippets/images from

- AMD's Application Programming Manuals:

http://support.amd.com/en-us/search/tech-docs

- Intel's Software Developers' Manuals:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

SUSE
We adapt. You succeed.