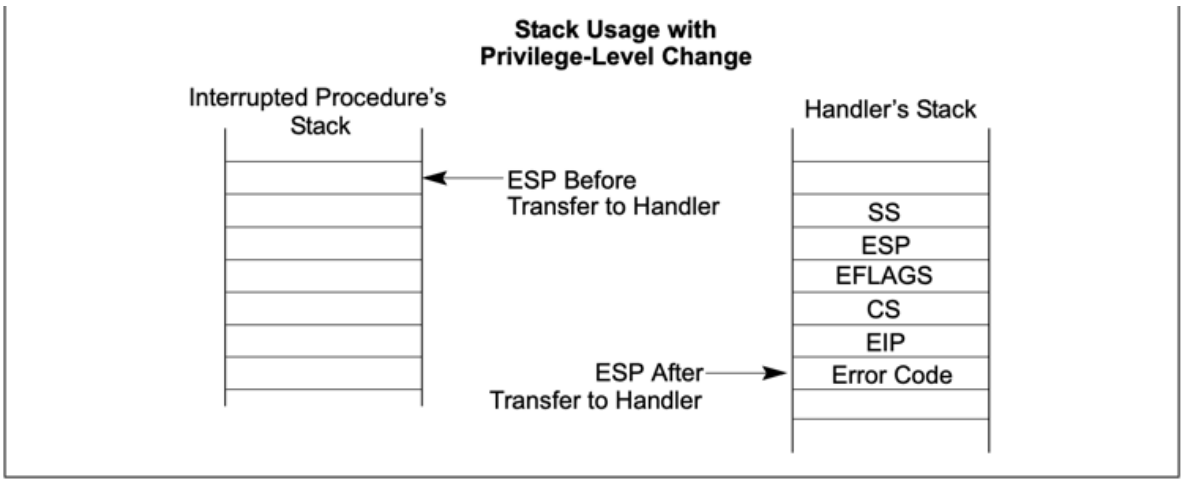
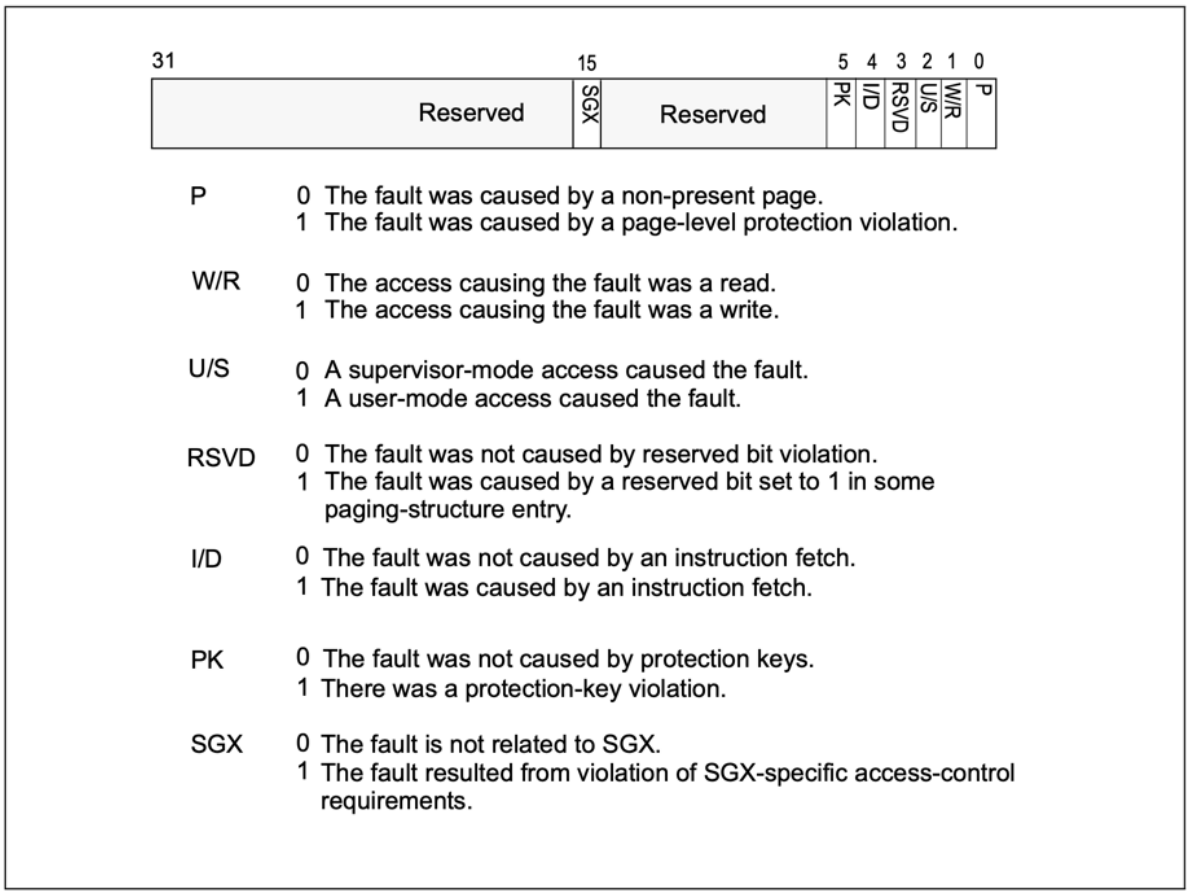


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines



P
0 The fault was caused by a non-present page.
1 The fault was caused by a page-level protection violation.

W/R
0 The access causing the fault was a read.
1 The access causing the fault was a write.

U/S
0 A supervisor-mode access caused the fault.
1 A user-mode access caused the fault.

RSVD
0 The fault was not caused by reserved bit violation.
1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.

I/D
0 The fault was not caused by an instruction fetch.
1 The fault was caused by an instruction fetch.

PK
0 The fault was not caused by protection keys.
1 There was a protection-key violation.

SGX
0 The fault is not related to SGX.
1 The fault resulted from violation of SGX-specific access-control requirements.

Figure 4-12. Page-Fault Error Code

```
// WeensyOS (Lab4): x86-64.h

typedef struct regstate {
    // General-purpose registers
    uint64_t reg_rax;
    uint64_t reg_rcx;
    uint64_t reg_rdx;
    uint64_t reg_rbx;
    uint64_t reg_rbp;
    uint64_t reg_rsi;
    uint64_t reg_rdi;
    uint64_t reg_r8;
    uint64_t reg_r9;
    uint64_t reg_r10;
    uint64_t reg_r11;
    uint64_t reg_r12;
    uint64_t reg_r13;
    uint64_t reg_r14;
    uint64_t reg_r15;
    uint64_t reg_fs;
    uint64_t reg_gs;

    uint32_t reg_intno;   // Interrupt number, -1 for syscall
    uint32_t reg_swapgs;  // Whether to `swapgs` on resume
    // Error code (supplied by hardware for some x86-64 exceptions)
    uint64_t reg_errcode;

    // Task status (pushed by exception mechanism, read by `iret`)
    uint64_t reg_rip;
    uint64_t reg_cs;
    uint64_t reg_rflags;
    uint64_t reg_rsp;
    uint64_t reg_ss;
} regstate;


// WeensyOS (Lab4): k-exception.S

// Exception entry point
//    Most exception handlers jump here.
.globl _Z15exception_entryv
_Z15exception_entryv:
        push %gs
        push %fs
        pushq %r15
        pushq %r14
        pushq %r13
        pushq %r12
        pushq %r11
        pushq %r10
        pushq %r9
        pushq %r8
        pushq %rdi
        pushq %rsi
        pushq %rbp
        pushq %rbx
        pushq %rdx
        pushq %rcx
        pushq %rax
        movq %rsp, %rdi

        // load kernel page table
        movq $kernel_pagetable, %rax
        movq %rax, %cr3

        call _Z9exceptionP8regstate
        // `exception` should never return.
```



```
// WeensyOS (Lab4): kernel.cc

void exception(regstate* regs) {
    // Copy the saved registers into the `current` process descriptor.
    current->regs = *regs;
    regs = &current->regs;

    …

    // Actually handle the exception.
    switch (regs->reg_intno) {

    case INT_IRQ + IRQ_TIMER:
        ++ticks;
        lapicstate::get().ack();
        schedule();
        break;                  /* will not be reached */

    case INT_PF: {
        // Analyze faulting address and access type.
        uintptr_t addr = rdcr2();
        const char* operation = regs->reg_errcode & PTE_W
                ? "write" : "read";
        const char* problem = regs->reg_errcode & PTE_P
                ? "protection problem" : "missing page";

        if (!(regs->reg_errcode & PTE_U)) {
            proc_panic(current, "Kernel page fault on %p (%s %s, rip=%p)!\n",
                       addr, operation, problem, regs->reg_rip);
        }
        error_printf(CPOS(24, 0), 0x0C00,
                     "Process %d page fault on %p (%s %s, rip=%p)!\n",
                     current->pid, addr, operation, problem, regs->reg_rip);
        current->state = P_FAULTED;
        break;
    }

    …

    // Return to the current process (or run something else).
    if (current->state == P_RUNNABLE) {
        run(current);
    } else {
        schedule();
    }
}
```