10/2/23

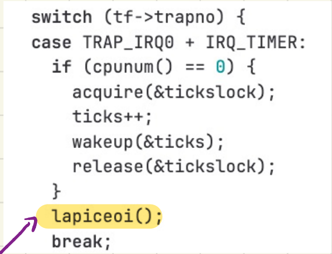
Review : Types of Mode Transfer

1). System call resumes at next instr

2). Interrupts resume on interrupted instr.

-> hw events (external interrupt)

-> can preempt syscall & exception handlers



```
switch (tf->trapno) {
case TRAP_IRQ0 + IRQ_TIMER:
  if (cpunum() == 0) {
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

-> 1 at a time, kernel sends EOI (end of interrupt) when done handling

3). Exceptions

-> caused by current instr.

-> behaviors differ depend on the type of exception

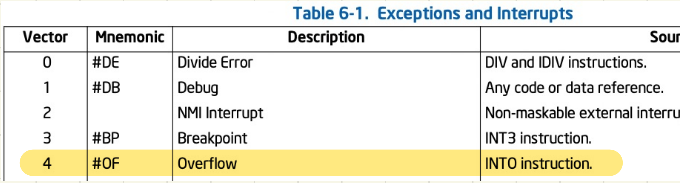


Table 6-1. Exceptions and Interrupts

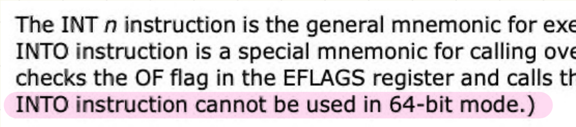
Vector	Mnemonic	Description	Sour
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interru
3	#BP	Breakpoint	INT3 instruction.
4	#OF	Overflow	INTO instruction.

-> overflow exception story * resumes on next instr.

-> arith. op sets the overflow flag (OF)

-> INTO instr. causes exception when OF flag is set

interrupt on overflow



The INT *n* instruction is the general mnemonic for exe
INTO instruction is a special mnemonic for calling ove
checks the OF flag in the EFLAGS register and calls th
INTO instruction cannot be used in 64-bit mode.)

-> page fault * resumes on the faulting instr.

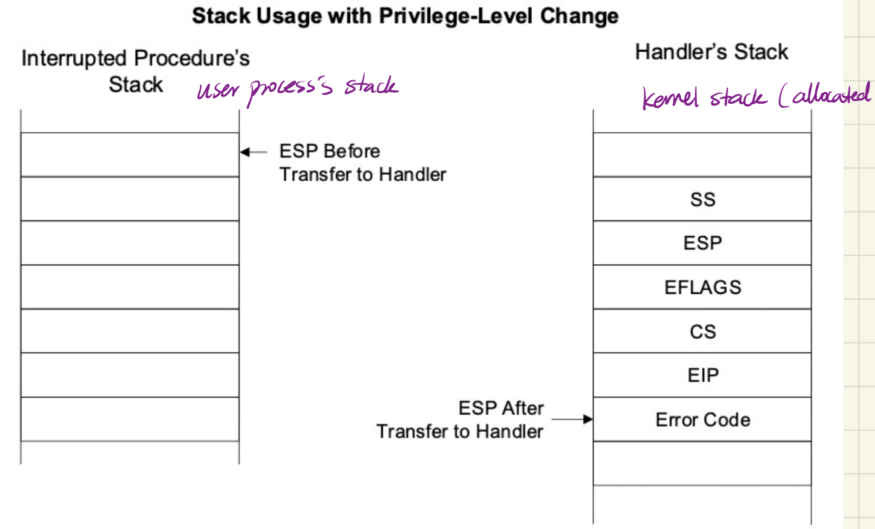
Mode Transfer Mechanism

Control Flow: enter kernel mode → switch to a kernel stack → save process's states → execute handler → restore process's states → return to user mode

Kernel stack

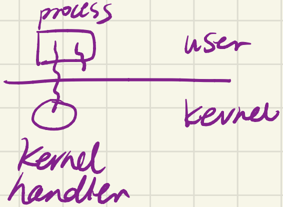
→ store local vars.
save registers
call frame (return addr)

✱ used when executing kernel code


Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

user process's stack

kernel stack (allocated from kernel memory)



Why separate kernel stack?

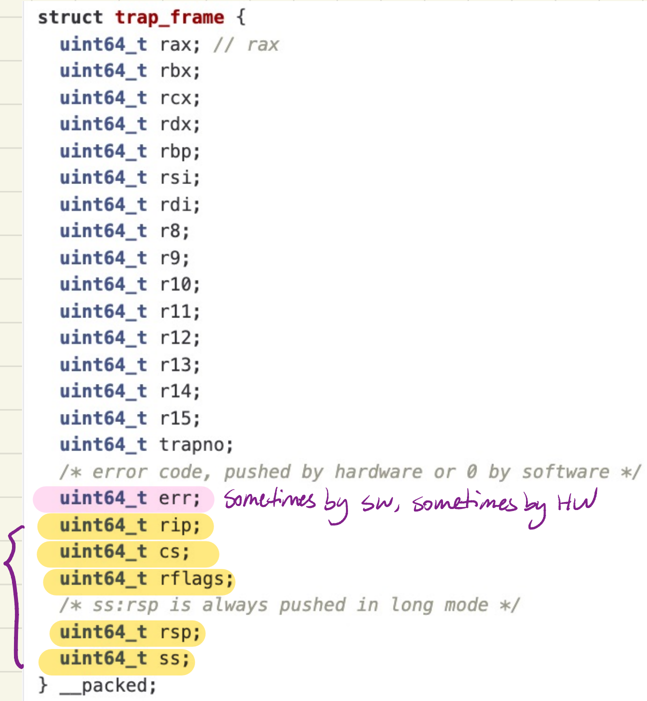
→ Security: user process/threads have read & write access to the user stack

→ what if the user stack is corrupted? kernel won't be able to service interrupts!

Control Flow: enter kernel mode → switch to a kernel stack → save process's states → execute handler → restore process's states → return to user mode

(can be found)

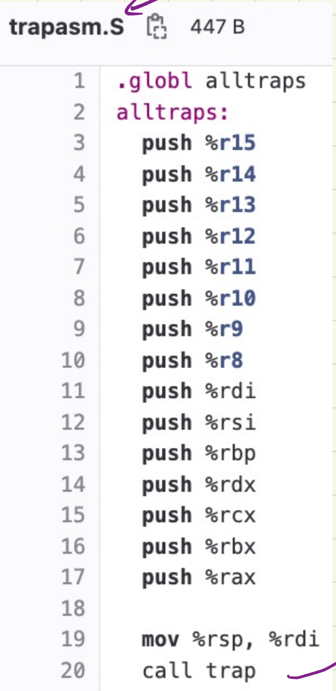
What states are saved?



```
struct trap_frame {
  uint64_t rax; // rax
  uint64_t rbx;
  uint64_t rcx;
  uint64_t rdx;
  uint64_t rbp;
  uint64_t rsi;
  uint64_t rdi;
  uint64_t r8;
  uint64_t r9;
  uint64_t r10;
  uint64_t r11;
  uint64_t r12;
  uint64_t r13;
  uint64_t r14;
  uint64_t r15;
  uint64_t trapno;
  /* error code, pushed by hardware or 0 by software */
  uint64_t err;
  uint64_t rip;
  uint64_t cs;
  uint64_t rflags;
  /* ss:rsp is always pushed in long mode */
  uint64_t rsp;
  uint64_t ss;
} __packed;
```

sometimes by SW, sometimes by HW

pushed by HW



trapasm.S 447 B

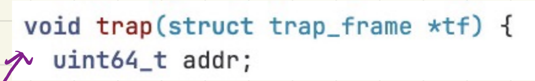
```
.globl alltraps
alltraps:
  push %r15
  push %r14
  push %r13
  push %r12
  push %r11
  push %r10
  push %r9
  push %r8
  push %rdi
  push %rsi
  push %rbp
  push %rdx
  push %rcx
  push %rbx
  push %rax

  mov %rsp, %rdi
  call trap
```

kernel code pushing (saving) states

Why save states?

to resume execution after the syscall/interrupt/exception



```
void trap(struct trap_frame *tf) {
  uint64_t addr;
```

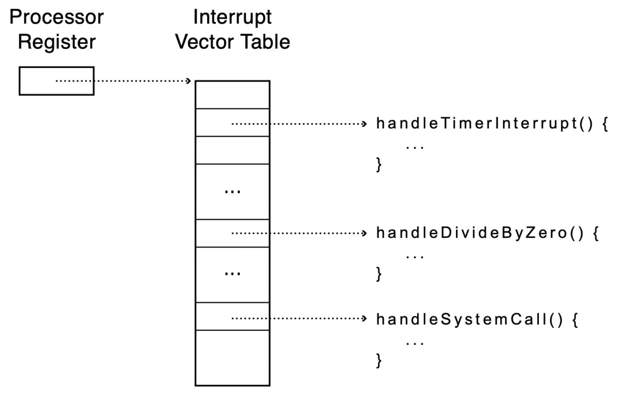
Control Flow: enter kernel mode → switch to a kernel stack → save process's states → execute handler → restore process's states → return to user mode

Interrupt Vector Table

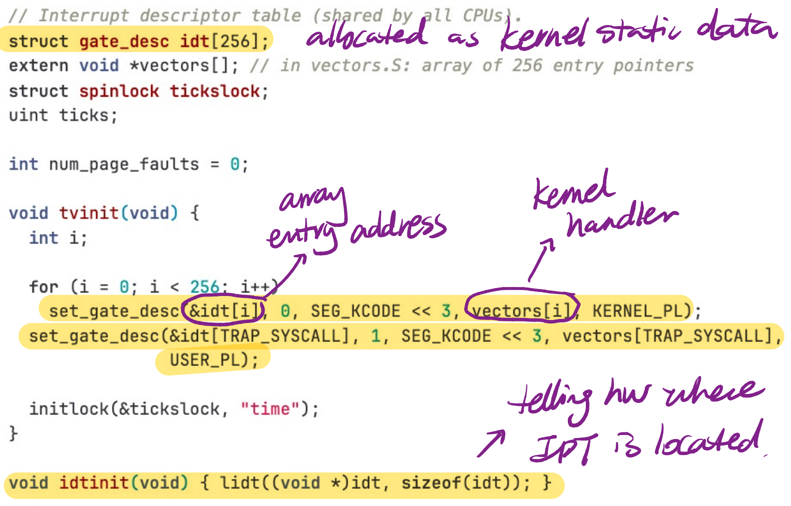
→ x86 Interrupt Descriptor Table [architecture support]

→ Table of 256 entries

→ array index = interrupt #
array entry = handler location



initialized by OS on startup



```
// Interrupt descriptor table (shared by all CPUs).
struct gate_desc idt[256];
extern void *vectors[]; // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;

int num_page_faults = 0;

void tvinit(void) {
  int i;

  for (i = 0; i < 256; i++)
    set_gate_desc(&idt[i], 0, SEG_KCODE << 3, vectors[i], KERNEL_PL);
  set_gate_desc(&idt[TRAP_SYSCALL], 1, SEG_KCODE << 3, vectors[TRAP_SYSCALL],
                USER_PL);

  initlock(&tickslock, "time");
}

void idtinit(void) { lidt((void *)idt, sizeof(idt)); }
```

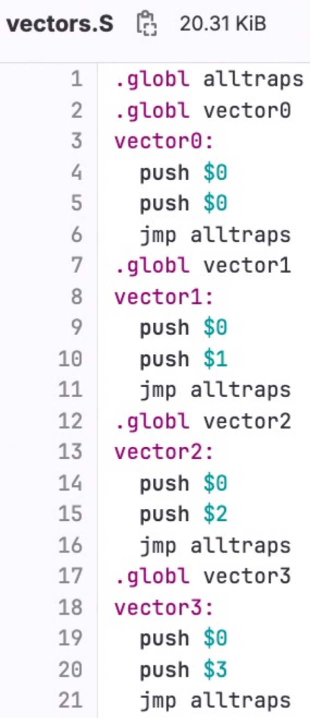
allocated as kernel static data

array entry address

kernel handler

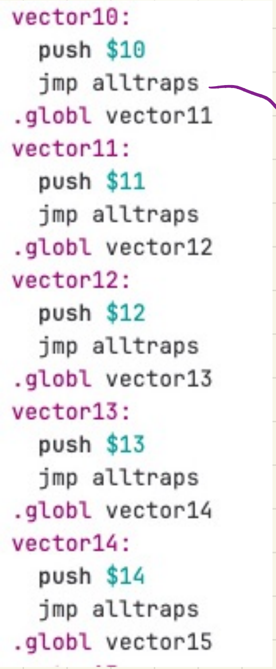
telling hw where IDT is located

(see next page for kernel handler examples)



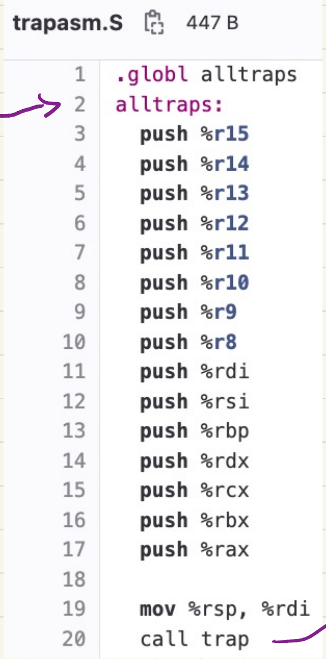
vectors.S 20.31 KiB

```
.globl alltraps
.globl vector0
vector0:
  push $0
  push $0
  jmp alltraps
.globl vector1
vector1:
  push $0
  push $1
  jmp alltraps
.globl vector2
vector2:
  push $0
  push $2
  jmp alltraps
.globl vector3
vector3:
  push $0
  push $3
  jmp alltraps
```



```
vector10:
  push $10
  jmp alltraps
.globl vector11
vector11:
  push $11
  jmp alltraps
.globl vector12
vector12:
  push $12
  jmp alltraps
.globl vector13
vector13:
  push $13
  jmp alltraps
.globl vector14
vector14:
  push $14
  jmp alltraps
.globl vector15
```

pushes errcode & trap number (IDT array index)

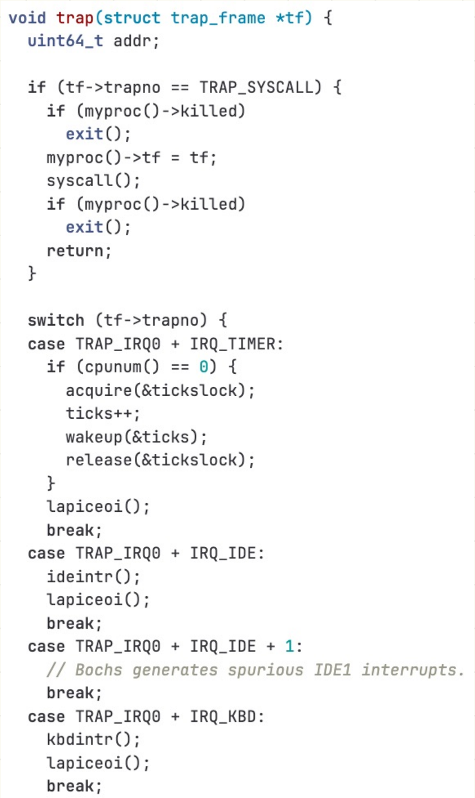


trapasm.S 447 B

```
.globl alltraps
alltraps:
  push %r15
  push %r14
  push %r13
  push %r12
  push %r11
  push %r10
  push %r9
  push %r8
  push %rdi
  push %rsi
  push %rbp
  push %rdx
  push %rcx
  push %rbx
  push %rax

  mov %rsp, %rdi
  call trap
```

generic trap handler, checks which trap it is & runs the specific handler



```
void trap(struct trap_frame *tf) {
  uint64_t addr;

  if (tf->trapno == TRAP_SYSCALL) {
    if (myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if (myproc()->killed)
      exit();
    return;
  }

  switch (tf->trapno) {
  case TRAP_IRQ0 + IRQ_TIMER:
    if (cpunum() == 0) {
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case TRAP_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case TRAP_IRQ0 + IRQ_IDE + 1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case TRAP_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
```