

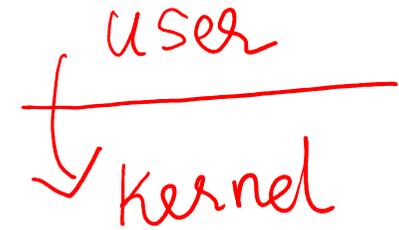
Lecture 24: Trap handling in xv6

Mythili Vutukuru

IIT Bombay

<https://www.cse.iitb.ac.in/~mythili/os/>

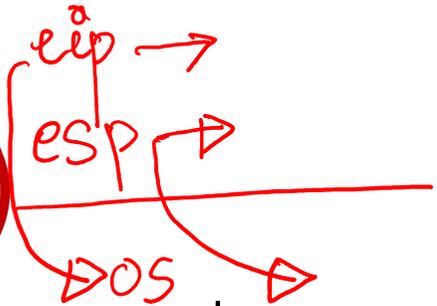
Trap handling in xv6



- The following events cause a user process to “trap” into the kernel (xv6 refers to all these events as traps)
 - System calls (requests by user for OS services)
 - Interrupts (external device wants attention)
 - Program fault (illegal action by program)
- When above events happen, CPU executes the special “int” instruction
 - Example seen in usys.S, “int” invoked to handle system calls
 - For hardware interrupts, device sends a signal to CPU, and CPU executes int instruction
- Trap instruction has a parameter (int n), indicating type of interrupt
 - E.g., syscall has a different value of n from keyboard interrupt



Trap instruction (int n)



- Before trap: eip pointing to user program instruction, esp to user stack. Suppose interrupt occurs now
- The following steps are performed by CPU as part of “int n” instruction
 - Fetch n-th entry interrupt descriptor table (CPU knows memory address of IDT)
 - Save stack pointer (esp) to internal register
 - Switch esp to kernel stack of process (CPU knows location of kernel stack of current process)
 - On kernel stack, save old esp, eip (where execution stopped before interrupt occurred, so that it can be resumed later)
 - Load new eip from IDT, points to kernel trap handler
- Result: ready to run kernel trap handler code, on kernel stack of process
- Few details omitted:
 - Stack, code segments (cs, ss) and a few other registers also saved
 - Permission checks of CPU privilege levels in IDT entries (e.g., user code can invoke IDT entry of system call, but not of disk interrupt)
 - If interrupt occurs when already handling previous interrupt (already on kernel stack), no need to save stack pointer again

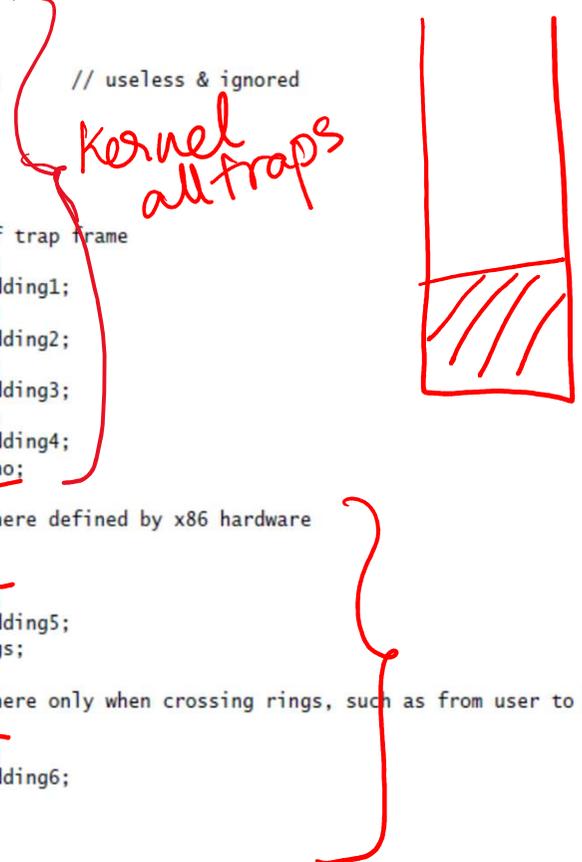
Why a separate trap instruction?

- Why can't we simply jump to kernel code, like we jump to the code of a function in a function call?
 - The CPU is executing user code in a lower privilege level, but OS code must run at higher privilege
 - User program cannot be trusted to invoke kernel code on its own correctly
 - Someone needs to change the CPU privilege level and give control to kernel code
 - Someone also needs to switch to the secure kernel stack, so that the kernel can start saving state
 - That “someone” is the CPU executing “int n”

Trap frame on the kernel stack

- Trap frame: state is pushed on kernel stack during trap handling
 - CPU context of where execution stopped is saved, so that it can be resumed after trap
 - Some extra information needed by trap handler is also saved
- The “int n” instruction has so far only pushed the bottom few entries of trap frame
 - The kernel code we are about to see next will push the rest

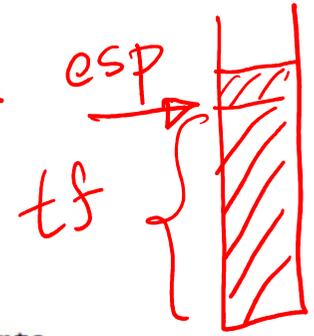
```
0600 // Layout of the trap frame built on the stack by the
0601 // hardware and by trapasm.S, and passed to trap().
0602 struct trapframe {
0603     // registers as pushed by pusha
0604     uint edi;
0605     uint esi;
0606     uint ebp;
0607     uint oesp; // useless & ignored
0608     uint ebx;
0609     uint edx;
0610     uint ecx;
0611     uint eax;
0612
0613     // rest of trap frame
0614     ushort gs;
0615     ushort padding1;
0616     ushort fs;
0617     ushort padding2;
0618     ushort es;
0619     ushort padding3;
0620     ushort ds;
0621     ushort padding4;
0622     uint trapno;
0623
0624     // below here defined by x86 hardware
0625     uint err;
0626     uint eip;
0627     ushort cs;
0628     ushort padding5;
0629     uint eflags;
0630
0631     // below here only when crossing rings, such as from user to kernel
0632     uint esp;
0633     ushort ss;
0634     ushort padding6;
0635 };
```



Kernel trap handler (alltraps)

- IDT entries for all interrupts will set eip to point to the kernel trap handler “alltraps”
 - Omit details of IDT construction
- Alltraps assembly code pushes remaining registers to complete trapframe on kernel stack
 - “pushal” pushes all general purpose registers
- Invokes C trap handling function named “trap”
 - Push pointer to trapframe (current top of stack, esp) as argument to the C function

```
3300 #include "mmu.h"
3301
3302 # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305 # Build trap frame.
3306     pushl %ds
3307     pushl %es
3308     pushl %fs
3309     pushl %gs
3310     pushal
3311
3312 # Set up data segments.
3313     movw $(SEG_KDATA<<3), %ax
3314     movw %ax, %ds
3315     movw %ax, %es
3316
3317     # Call trap(tf), where tf=%esp
3318     pushl %esp
3319     call trap
3320     addl $4, %esp
3321
3322 # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325     popal
3326     popl %gs
3327     popl %fs
3328     popl %es
3329     popl %ds
3330     addl $0x8, %esp # trapno and errcode
3331     iret
```



C trap handler function (1)

- C trap handler performs different actions based on kind of trap
- If system call, “int n” is invoked with “n” equal to a value T_SYSCALL (in usys.S), indicating this trap is a system call
- Trap handler invokes common system call function
 - Looks at system call number stored in eax (whether fork or exec or) and calls the corresponding function
 - Return value of syscall stored in eax

usys.S

```
3400 void
3401 trap(struct trapframe *tf)
3402 {
3403     if(tf->trapno == T_SYSCALL) {
3404         if(myproc()->killed)
3405             exit();
3406         myproc()->tf = tf;
3407         syscall();
3408         if(myproc()->killed)
3409             exit();
3410         return;
3411     }
```

```
3700 void
3701 syscall(void)
3702 {
3703     int num;
3704     struct proc *curproc = myproc();
3705
3706     num = curproc->tf->eax;
3707     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3708         curproc->tf->eax = syscalls[num]();
3709     } else {
3710         cprintf("%d %s: unknown sys call %d\n",
3711             curproc->pid, curproc->name, num);
3712         curproc->tf->eax = -1;
3713     }
3714 }
```

C trap handler function (2)

- If interrupt from a device, corresponding device-related code is called
 - The trap number (value of “n” in “int n”) is different for different devices
- Timer is special hardware interrupt, and is generated periodically to trap to kernel

```
3413  switch(tf->trapno){
3414  case T_IRQ0 + IRQ_TIMER:
3415      if(cpuid() == 0){
3416          acquire(&tickslock);
3417          ticks++;
3418          wakeup(&ticks);
3419          release(&tickslock);
3420      }
3421      lapiceoi();
3422      break;
3423  case T_IRQ0 + IRQ_IDE:
3424      ideintr();
3425      lapiceoi();
3426      break;
3427  case T_IRQ0 + IRQ_IDE+1:
3428      // Bochs generates spurious IDE1 interrupts.
3429      break;
3430  case T_IRQ0 + IRQ_KBD:
3431      kbdintr();
3432      lapiceoi();
3433      break;
```

C trap handler function (3)

- On timer interrupt, a process “yields” CPU to scheduler
 - Ensures a process does not run for too long

```
3471 // Force process to give up CPU on clock tick.
3472 // If interrupts were on while locks held, would need to check nlock.
3473 if(myproc() && myproc()->state == RUNNING &&
3474     tf->trapno == T_IRQ0+IRQ_TIMER)
3475     yield();
3476
```

```
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830     acquire(&ptable.lock);
2831     myproc()->state = RUNNABLE;
2832     sched();
2833     release(&ptable.lock);
2834 }
```

Return from trap

- Pop all state from kernel stack
- Return from trap instruction “iret” does the opposite of int
 - Pop values pushed by “int”
 - Change back privilege level
- Execution of pre-trap code can resume

```
3300 #include "mmu.h"
3301
3302 # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305 # Build trap frame.
3306     pushl %ds
3307     pushl %es
3308     pushl %fs
3309     pushl %gs
3310     pushal
3311
3312 # Set up data segments.
3313     movw $(SEG_KDATA<<3), %ax
3314     movw %ax, %ds
3315     movw %ax, %es
3316
3317 # Call trap(tf), where tf=%esp
3318     pushl %esp
3319     call trap
3320     addl $4, %esp
3321
3322 # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325     popal
3326     popl %gs
3327     popl %fs
3328     popl %es
3329     popl %ds
3330     addl $0x8, %esp # trapno and errcode
3331     iret
```

Summary of xv6 trap handling

- System calls, program faults, or hardware interrupts cause CPU to run “int n” instruction and “trap” to OS
- The trap instruction (int n) causes CPU to switch esp to kernel stack, eip to kernel trap handling code
- Pre-trap CPU state is saved on kernel stack in the trap frame (by int instruction + alltraps code)
- Kernel trap handler handles trap and returns from trap to whatever was running before the trap