6.828 2012 Lecture 5: Interrupts, exceptions

plan:
  entering the kernel (interrupts, system calls, &c)
  returning to user space

where can the system be executing?
  diagram: u/k, user stacks, kernel stacks

what are the transitions?
  u -> k: sys call / interrupt / exception
  k -> u: return from sys call / interrupt / exception
  k -> k: context switch
  k -> k: interrupt in kernel mode

interrupts, exceptions, and system calls all use the same mechanism!
we'll talk about context switch next week

Q: why per-process kernel stack?
   what would go wrong if syscall used a single global stack?

*** entering the kernel

why do we need to take special care for user -> kernel?
  we want to maintain isolation
  we want transparency (esp for device interrupts)

remember how x86 privilege levels work
  CPL in low 2 bits of CS
  CPL=0 -> kernel mode
  CPL=3 -> user mode

what has to happen in a system call?
  save user state for future resume
  set up for execution in kernel (stack, CPL=0)
  choose a place to execute in kernel
  get at system call arguments

let's look at what happens during a system call
  an sbrk() call in sh
  b *0xf48
  c -- to mov $0xc, %eax -- 0xc = 12 = SYS_sbrk
  stepi -- to int $0x40
  x/4x $esp -- return PC, sbrk argument, ...
  stepi -- to vector64, pushl $0x40

where are we? how did we get here?

the INT instruction jumps into the kernel

where does INT jump to?
  the $0x40 is a vector number
  a vector is an allowed entry point

x86 has 256 vectors, for different uses (devices, exceptions, &c)
kernel knows why interrupt occured by looking at vector #
vector is index of a descriptor in the "IDT"
IDTR register has IDT's base address
each IDT descriptor has seg selector, offset in segment
see handout
for xv6, seg selector always SEG_KCODE, offset is address of vector fn
IDT seg selector will be the code segment
so IDT seg selector determines CPL
print idt[0x40]

INT instruction steps (similar for interrupts and exceptions):
fetch vector's descriptor from IDT
if seg selector's PL < CPL:
it's a cross-ring interrupt
save ESP and SS in a CPU-internal register
load SS and ESP from TSS
push user SS
push user ESP
push user EFLAGS
push user CS
push user EIP
clear some EFLAGS bits (XXX what?)
set CS and EIP from IDT descriptor's segment selector and offset

Q: does INT really need all those steps?
e.g. why does it save SS and ESP?

what's the current CPL?
print $cs
why can't user code abuse INT to get privilege?

xv6 details:
vectors.S
tvinit() (trap.c) sets up IDT during boot
switchuvm() (vm.c) sets ss and esp in TSS

x/6x $esp
Q: which stack?
what's on it?
user registers: fake error, eip, cs, eflags, esp, ss

x/3i vector64
why push 0x40?
why not have IDT point directly to alltraps?

stepi to pushl %esp (trapasm.S, just before call trap)

x/19x $esp
these are all saved *user* registers
compare with struct trapframe, x86.h
ss // hw pushes these:
esp
eflags

```
  cs
  eip
  (error)
  trapno   // vector pushes this
  ds       // alltraps pushes these:
  es
  fs
  gs
  eax
  ecx
  edx
  ebx
  xxesp
  ebp
  esi
  edi
```

Q: where does trap(tf) argument come from?

Q: where did tf->trapno come from?

*** system call handling

if T_SYSCALL (0x40), trap() calls syscall()
  syscall() gets system call number from tf->eax
    Q: where was %eax set?
  Q: where are the system call arguments?
  sys_sbrk() fetches system call argument from user stack
    pushed by user-level C call to sbrk()
    via tf->esp
    argint()

syscall() puts sys_sbrk() return value in tf->eax

syscall() returns to trap()
  trap() returns to alltraps (trapasm.S)

*** kernel -> user

use "finish" to return to trapasm.S
si (until popal)
x/19x $esp

Q: what has changed in the trapframe?

si until iret
x/5x $esp
  eip cs eflags esp ss

si (into user space)

Q: where are we now?
Q: what are we returning from?

x/4x $esp
Q: what stack is that?
Q: what's on the stack?

*** other points about interrupts/exceptions

Faults, like page fault and divide by zero, work much the same, but to different vectors.

The kernel handles some exceptions internally, e.g. lazy-allocate page fault.

The program can arrange to get control after some exception -- via UNIX signal handlers.

Faults, interrupts, &c can occur while the kernel is running
  old CPL == new CPL
  h/w doesn't switch stacks
  h/w doesn't push old esp/ss
  so trapframe is a bit different
  you can tell: look at old CPL in low bits of tf->cs

device interrupts
  hardware generates them
  examples: timer, disk, console
  vector per device

example device: timer
  trap.c, look for IRQ_TIMER
  add a cprintf, you'll see it's called a lot

*** fork

a process's kernel stack is originally set up in fork()

we want to know setup of child
  user stack, registers, EIP
  kernel stack, registers, EIP

struct proc, in proc.h
  each has to be initialized in child

allocproc()
  kstack allocation
  space for trapframe
  *sp = trapret
  space for context
  context->eip = forkret

fork()
  copyuvm() -- copy user stack, instructions, heap
  *np->tf = *proc->tf -- copy user registers
  np->tf->eax = 0 -- child return value

6.828 Operating System Engineering

Fall 2012