LECTURE 8: TRAP-SCHEDULING

Trap Handling- pg 39-44 of
Book : xv6Book

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)

System call interrupt Program fault.

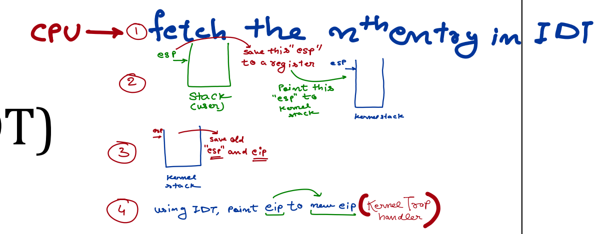
Trap

- 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*)

The IDT allows the CPU to quickly and efficiently determine how to handle various interrupts and exceptions

- 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 ↳ esp, eip etc
  - The kernel code we are about to see next will push the rest

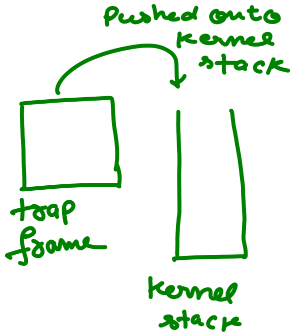
    ↳ (all traps)

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```



Kernel trap handler (alltraps)

builds the trap frame

- IDT entries for all interrupts will set eip to point to the kernel trap handler "alltraps"

  kernel trap handler.

  - 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

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp   # trapno and errcode
  iret
```

Pushing on to the Stack

esp

takes "esp" as argument.

trap frame (for saving the previous state)

kernel stack

Kernel trap C function. trap.c

Reversing (popping from stack)

reverse of "int" instruction

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

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;        system call Value
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

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

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
case T_IRQ0 + IRQ_IDE:
  ideintr();
  lapiceoi();
  break;
case T_IRQ0 + IRQ_IDE+1:
  // Bochs generates spurious IDE1 interrupts.
  break;
case T_IRQ0 + IRQ_KBD:
  kbdintr();
  lapiceoi();
  break;
```

C trap handler function (3)

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

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

// Check if the process has been killed since we yielded
if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
  exit();
}
```

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- Process set itself to "Ready"
- calls scheduler.

Return from trap

- Pop all state from kernel stack
- Return from trap instruction "iret" does the opposite of int
  - Pop values pushed by "int" → esp, eip, etc.
  - Change back privilege level
- Execution of pre-trap code can resume

```
#include "mmu.h"

  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  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 and returns from trap to whatever was running before the trap

Thank You