Trap handling in xv6

Mythili Vutukuru
CSE, IIT Bombay

Trap handling in xv6

- The following events in xv6 cause a user process to “trap” into the kernel
  - 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
- Trap instruction has a parameter (int n), indicating type of interrupt
  - E.g., syscall has a different value of n from keyboard interrupt
  - The value of “n” is used to index into IDT, get address of kernel code to run
- xv6 trap handling code saves register context, handles trap, returns

xv6 system calls

- In xv6, system calls available to user programs are defined in user library header “user.h”
  - Equivalent to C library headers (xv6 doesn’t use standard C library)
- These system call functions invoked in user programs after including “user.h”
- The actual invoking of system call is done in usys.S

```
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

xv6 system calls

- The user library makes the actual system call to invoke OS code
- User library invokes trap instruction to make system call, code seen in usys.S
  - Defined using a macro
  - Move system call number to eax
  - Invoke int n where n is T_SYSCALL
- Other interrupts set different values of n
- The trap (int) instruction causes a jump to kernel code that handles the system call

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
```

Trap frame in xv6

- Trap frame is the structure pushed on kernel stack before trap handling, popped when returning from trap
- Contains various registers that are saved on kernel stack before trap handling
- The “int n” instruction pushes a few registers (old PC, old SP etc.) and jumps to kernel code to handle trap
- The kernel code that is run next will push remaining registers on kernel stack, and then proceed to handle the trap
- Think: why are EIP, ESP pushed by hardware and not by kernel code?

```
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 };
```

xv6 kernel trap handler

- IDT entries for all interrupts eventually redirect to the kernel trap handler “alltraps”
  - Push trap number (n) on trapframe and then call alltraps
- Alltraps assembly code pushes remaining registers to complete trapframe on kernel stack
- 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 in xv6

- C trap handler performs different actions based on kind of trap
- Different types of traps identified using value of “n” in “int n”
- For system call, “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 and calls the corresponding function (fork, exec, …)
  - Return value of syscall stored in eax

C trap handler invoking syscalls

```
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 (contd)

- If interrupt from a device, corresponding driver code called
- Timer is special hardware interrupt, 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;
```

```
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 }
```

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

Return from trap

- Assembly code “trapret”
- 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
```

xv6 trap handling: the complete story

- 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 “alltraps”
- Pre-trap CPU state is saved on kernel stack in the trap frame by int instruction + alltraps code
- Alltraps assembly code calls C trap handling function
- C trap handler handles trap suitably and returns to trapret code
- Trapret pops register context and runs “iret” instruction to return from trap to user mode of process