Lecture 1, Problemset 1
Agenda

1. X86 Refresher
2. xv6 Code Reading
3. Discussion of Problemset
X86: Basics

- Up to 3 operating modes: Real Mode (16 bit), Protected Mode (32 bit), Long Mode (64 bit)
- 4 Privilege rings: 0 (highest), 1, 2, 3
- 8 32-bit general purpose registers: eax, ebx, ecx, edx, esi, edi, ebp, esp
- 6 segment registers: cs, ds, es, fs, gs, ss
- 2 special purpose registers: eip, eflags
- Calling convention: arguments passed on stack
X86: Privileged Features

- Mostly controlled with control registers: cr0, cr1, cr2, cr3, cr4

- All memory accesses are relative to a segment
  - Can be implicit or explicit
  - Controlled by General Descriptor Table (GDT):
    - segment offset, length, privileges
  - Additional special task state segment

- 256 Trap vectors for Interrupts, Exceptions and Syscalls
  - Controlled by Interrupt Descriptor Table (IDT):
    - handler, type (int or trap), privileges
  - Some differences between user and kernel space
X86: Traps

- From kernel for vector $n$:
  1. Fetch $n$th descriptor from IDT
  2. Push eflags, cs, eip
  3. For some exceptions: push error code
  4. Clear some flags in eflags (including IF if int gate)
  5. Set cs and eip to values from descriptor
X86: Traps (cont'd)

- From user space for vector n:
  1. Fetch nth descriptor from IDT
  2. If SW interrupt: check descriptor privilege level (DPL) >=
     current privilege level (CPL)
     - otherwise general protection fault
  3. Remember esp, ss
  4. Load esp[DPL], ss[DPL] from task state segment (TSS)
  5. Push old esp, old ss, eflags, cs, eip
  6. For some exceptions: push error code
  7. Clear some flags in eflags (including IF if int gate)
  8. Set cs and eip to values from descriptor
X86: Trap Return

- `iret` instruction used to return from trap
  - As well as initially entering user space

- Partially inverts what CPU does on trap
  1. Pop eip, cs, eflags
  2. If change to lower privilege: pop ss, esp
Code Reading

1. Makefile
2. Bootloader and initialization
3. Trap handling
4. Skim exec implementation
Debugging

- **GDB example**
  - Some useful commands: `break foo`, `si`, `c`, `finish`, `info registers`, ...

https://courses.cs.washington.edu/courses/cse451/16au/labs/tools.html#gdb

- **Qemu**:
  - Logs: `make QEMUEXTRA='-d int -D qemu.log' qemu-nox-gdb`
  - Console: `Ctrl + A C`, then `info registers`, `info tlb`,...
Problem set: Code reading questions

1) Identify the first line of xv6 code that is executed in the kernel when a system call occurs, when an interrupt occurs, and when an exception occurs.

2) A system call, such as UNIX open, ultimately leads to a trap into the operating system kernel. Find where in xv6 the system call is invoked.
Problem set: Questions 3, 4

3) Why can’t we use the native C compiler libraries to build user programs to run on xv6? Likewise, why can’t we use those libraries in xv6 kernel mode?

4) xv6 provides a C library printf function for use by the xv6 applications, and a separate cprintf function for use by the kernel. Why?
Problem set: Code reading questions (cont'd)

5) Where is the first line of code for constructing an xv6 trapframe? How large is an xv6 trapframe? Why?

6) In xv6, when a user program (such as the shell) returns from main, what is done with the value it returns?
Problem set: Debugging questions

- 7) Do xv6 chapter 1, problem 1.

- 8) Add a tracing utility to xv6 to print (to the console) every system call as it occurs and its return value.
Problem set: Question 9

- 9) Add an upcall mechanism to xv6 to call up to user space. Add a system call, alarm(procptr, interval), that sets up a periodic upcall to procptr every interval time ticks, in other words, the user-level equivalent of a hardware timer.