Week 8.a
CS6640
10/24 2023
https://naizhengtan.github.io/23fall/

1. OS basics and C
2. RISC-V
3. Context and scheduling
4. trap to kernel
5. isolation and protection
6. OSI and egos

OSI midterm review
---

1. OS basics and C

- everything is 0s and 1s
  * bits and bytes
  * instructions are 0s and 1s
  * data: char, int, long long
  * unsigned vs. signed types

- little endian
  * what is little endian
  * how 0xdeadbeef looks like in little endian?
  * configurable in RISC-V machines
  * we (egos) assume little endian

- a program's memory layout
  * text segment
  * data segment
  * stack
  * heap
  * Which RISC-V registers point to them?

- C pointers
  * a pointer = a memory address
  * pointer arithmetic
  * a pointer of a pointer?
    -- "void **old_sp" in context switch

- common knowledge of C
  * uninitialized local variables on stack
  * two ways to access arrays
  * keywords:
    * union
    * static variable
    * bit field
  * bit manipulation

```
10001 & 10000 = 10000   // bit-wise and
10001 | 10000 = 10001   // bit-wise or
10001 ^ 10000 = 00001   // bit-wise xor
~1000 = 0111            // bit-wise not
```

2. RISC-V

- asm(Template : OutputOperands : InputOperands)
  * Template: a string that is the template for the assembler code.
  * OutputOperands: the C variables modified by the instructions in the

Template.
    * InputOperands: C expressions read by the instructions in the Template.
    * for example,
        int mie;
        asm("csrr %0, mie" : "=r"(mie));
        asm("csrw mie, %0" ::"r"(mie | 0x80));

  - common RISC-V registers
    * execution: pc, sp, ra
    * exceptions: mepc, mtvec, mcause
    * timer interrupt: mtime, mtimcmp, mstatus, mie

  - common RISC-V assembly instructions
    * execution: addi, sw, lw, mv
    * function calls: call, ret
    * syscalls: ecall, mret
    * CSRs: csrr, csrw

3. Context and scheduling

  (a) user-level threading

  - TCB: thread control block
    * thread id
    * status
    * "main" function pointer
    * function arguments
    * context
      (stack pointer, program counter, general purposed registers)

  - ULT tasks
    * manages TCBs
    * make a new stack for each new thread
    * scheduling
    * context switch! (in user-space)

  - context switch in user space
    * how it works in egos?
      ctx_start()/ctx_switch()

  - cooperative scheduling vs. preemptive scheduling
    * lab2 ULT is cooperative
    * one can implement a preemptive ULT by using signals

 (b) kernel schedulers

  - PCB: process control block
    * process id
    * status
    * context (sp and mepc)
    * IPC (sender & receiver)
    * scheduling attributes

  - process status
    * Unused, Ready, Runnable, Running, "Waiting"
    * transition diagram
      -- definition: preemptive scheduler and nonpreemptive scheduler

  - scheduling metrics
    * turnaround time
    * response time
    * CPU time

```
    * yield numbers

  - scheduling algorithm
    * round robin (~= original scheduler)
    * MLFQ (lab3)

  - "loop &; ls"
    * the performance difference in the above two algorithms,
    * and why?

4. trap to kernel---interrupts, exceptions, and syscalls

  (a) timer interrupt

  - the backbone of handling timer interrupts

    void handler() {
        CRITICAL("Got a timer interrupt!");
        // (4) reset timer
    }

    int main() {
        CRITICAL("This is a simple timer example");
        // (1) register handler() as interrupt handler
        // (2) set a timer
        // (3) enable timer interrupt

        while(1);
    }

  - register timer handler
    * mtvec

  - set/reset CPU timer
    * mtime, mtimecmp

  - enable timer interrupt
    * mstatus, mie

  - how to trigger a timer interrupt?
    * set mtime + mtimecmp accordingly
    * set the MSIP bit of mip
      // how we initially implement syscall

  - bummers: rollover, overflow, implicit type conversion
    * rollover
    * overflow
      -- the fundamental problem:
        RV32 can atomically read 4B at a time
    * implicit type conversion

  - gcc interrupt attribute:
    void handler() __attribute__((interrupt));

  (b) exceptions

  - CPU: "I don't know how to progress".
  - Exceptions are synchronized traps.

  - RISC-V exception reason table
    Examples:
      * read/write invalid memory
      * run invalid instructions
```

```
    * run privileged instructions in unprivileged mode

  - handling exceptions:
    Similar to interrupts.
    Again, mtvec, mcause, mepc,

  (c) syscalls

  - Interfaces for user applications to "talk" to kernel.

  - Three design questions for kernel developers
    Q1: how to trap to kernel?
    Q2: what information is transferred between apps and kernel?
    Q3: how to transfer the information?

  - how to trap to kernel
    * interrupt (skeleton code)
    * ecall     (your lab4)

5. isolation and protection

  (a) privilege levels

  - defined by CPU
    * two bits, may or may not be visible
    * can be a "virtualization hole"

  - a program running in the high privilege level can
    * execute privileged instructions,
    * touch privileged registers (CSRs),
    * access memory marked as privileged

  - how to switch privilege levles
    * low-to-high: interrupts, exceptions, and ecall
    * high-to-low: set MPP, then mret

  - trap-and-emulate hypervisor
    * VM: simulation of a computer, accurate enough to run an O/S
    *  running the VM's kernel in U-Mode
    * ordinary instructions work fine
    * privileged RISC-V instructions are illegal in U-Mode
        will cause a trap, to the hypervisor
    * hypervisor trap handler emulates privileged instruction

  (b) memory protection

  - protection = isolation + access control

  - the access control problem
      subj --[access]--> obj

    * for us
        subj: instructions + context
              (privilege levels and others)
        op: r/w/x
        obj: memory address

  - solutions: ACL
    * segmentation (x86-32)
    * PMP (RISC-V)
    * paging

  - segmentation (x86-32)
```

    * protecting granularity: a segment, defined by "base" and "limit"
    * segments are defined in descriptor tables in memory
    * access right bits are in descriptors

  - PMP (RISC-V)
    * protecting granularity: memory regions
    * regions are defined by pmpcfg and pmpaddr CSRs
    * access right bits are in pmpcfg registers

  - paging
    * isolation: different roots of page tables provide different address spaces
    * access control:
      -- protecting granularity: pages
      -- access right bits are in page tables, in particular, page table enries

6. OSI and egos

  - OS organizations:
    * monolithic OS
    * microkernel OS (egos-2k+)
    * exokernel (LibOS)
    * multikernel

  - a bird view of egos-2k+
    * three layers: earth, grass, and apps
    * earth/grass/user apps run in M/S/U-Mode
    * earth abstracts HW
    * grass provides scheduling, IPC, and syscalls

  - earth and grass interfaces
    * a list of function pointers in a well-known memory location

  - system apps (part of OS)
    * sys_proc (GPID_PROC)
    * sys_file (GPID_FILE)
    * sys_dir   (GPID_DIR)
    * sys_shell (GPID_SHELL)

  - two perspectives to see OS
    (a) static: disk layout + memory layout
    (b) dynamic: OS booting + responding "requests"

  (a) static view

  - OS "disk" layout
    * read mkfs.c
    * 1MB for 8 OS elfs, including grass.elf and four system services
    * readonly fs for user apps

  - elf binary too big?
    * too many floating-point operations
    * too much debugging info
    * too many macros

  - OS memory layout
    * see "memory layout" on reference page

  (b) dynamic view

  - OS booting
    * CPU jmps to 0x20400000

```
      +-> earth.S:_enter
          +-> earth.c:main
              +-> grass.S:_enter
                  +-> grass.c:main
                      +-> app.S:_enter
                          +-> sys_proc.c:main
                              |
                              ...
                              |
                              +-> sys_shell.c:main
```

- kernel ~= 3 handlers
  * handling interrupts (scheduling)
  * handling exceptions (killing apps)
  * handling syscalls   (run syscalls)

- egos exception handler:

```
    trap_entry (cpu_intr.c)
    +-> trap_handler (kernel.c)
        (ctx_start)
        +-> ctx_entry()
            +-> excp_entry(id)
                +->[your code]
                   +-> ctx_switch() // back to user stack
                       +-> [end of trap_handler]
                           +-> [end of trap_entry]
                               +-> [user space]
```

- handling syscalls:
  * see "syscall workflow" on reference page

notes about exam

- 90min
  -- a lot to read
  -- don't linger

- 3min
  -- for you to turn in your exam
  -- will reject your submission after 3min

- when something is ambiguous:
  -- always write down your assumptions

- 11 pages; A4; one-sided
  -- stapled
  -- feel free to disassemble (easier to read code and answer questions)
  -- if you do so, remember to write down your name at the bottom of each page

Northeastern University
CS6640: Operating System Implementation: Fall 2023
Midterm Exam

- This exam is **90 minutes**.
- Stop writing when "time" is called. *You must turn in your exam; we will not collect it.* Do not get up or pack up in the final ten minutes. The instructor will leave the room 93 minutes after the exam begins and will not accept exams outside the room.
- There are **8** problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.
- **This exam is closed book and notes. You may not use electronics: phones, tablets, calculators, laptops, etc.**
- If you find a question unclear or ambiguous, be sure to write any assumptions you make.
- If the questions impose a sentence limit, we will not read past that limit. In addition, *a response that includes the correct answer, along with irrelevant or incorrect content, will lose points.*
- Don't linger. If you know the answer, give it, and move on.

Do not write in the boxes below.

I (xx/30)	II (xx/12)	III (xx/32)	IV (xx/26)	Total (xx/100)

Name: