Three easy pieces

Textbook: Three easy pieces

- virtualization
  - CPU
  - memory
  - *correctness intuition should be preserved*
- concurrency
  - virtualize the CPU w/o compromising intuition or correctness
- persistence
  - file systems....
- *Six Easy Pieces: Richard Feynman* - highly recommended
  - because physics is twice as hard as computer science

OS History

- Multics
  - hierarchical file systems
  - dynamic linking
  - single level store
- UNIX
  - file systems
  - protection
  - portability
  - *modularity*
    - everything is a file: files, I/O devices, pipes and sockets
    - input/output is bytes: `ls | grep pete | sort`
  - descendents
    - linux
    - macOS
    - *GeekOS*

This Course

- Very high-level
    - synchronization approaches
    - queuing theory
    - scheduling and resource allocation algorithms
    - tests and homeworks are 60% of the grade
- Also very low-level: we are hacking an operating system!
    - process creation, signals, pipes
    - file systems
    - virtual memory
    - Projects are 40% of the grade
- Two parts sometimes don't tie together well...
- Both extremely important

Today

- Quick welcome and overview
- Finish up Sections 1,2 of Geekos slides
- Sections 1,2 of Processes

Goal

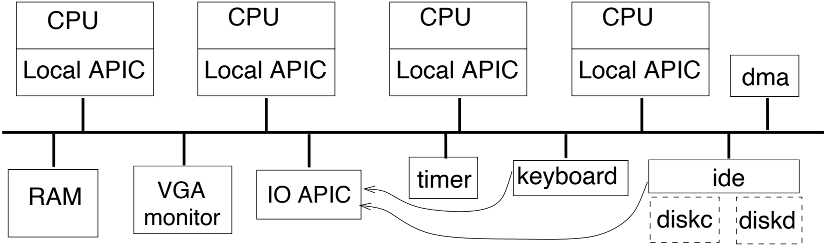
- Provide a very compact overview of GeekOS
- Much more friendly but older: geekos overview s2017.

  This has some content missing (due to latex-to-word conversion), which can be seen in geekos overview s2015.

Outline

1. Hardware and devices (drivers + interrupt handlers)
2. Booting and kernel initialization
3. Kernel threads
4. User processes
5. Interrupt-disabling and Spinlocks
6. Scheduling
7. Synchronization constructs
8. Virtual filesystem
9. PFAT
10. Blockdev
11. Bufcache

Hardware configuration



- x86 cpus in SMP (symmetric multi-processing) configuration
- apics (interrupt controllers)
  - local apic: recv intrpts from io-apic, send/recv to other cpus
  - io-apic: route interpts from io devices/timer to local apics
- diskc: kernel image; pfat filesystem with user programs
- emulated by QEMU running on linux (unix) environment

x86 (CPU)

- Has several modes: only “real” and “protected” modes relevant

- Real mode
  - Enters this mode upon power up
  - 16-bit machine (Intel 8086)
  - 20-bit segmented memory address: 1MB
  - 16-bit IO (port) address, 256 interrupts

- Protected mode
  - Enter this mode upon executing a certain instr in real mode
  - 32-bit machine with many more features
  - 4 privilege levels: 0 (kernel mode), 1, 2, 3 (user mode)
  - 32-bit segmented (+ optional paging) memory address: 4GB
  - 16-bit IO (port) address space, 256 interrupts
  - Geekos runs in this mode.
  - Rest of this section deals with protected mode

x86: Addressing

- Address space: 4GB (32-bit address)
- Segment: a contiguous chunk of address space
  - code segment, data segment, stack segment
- Address formed from 16-bit segment selector and 32-bit offset
- Segment selector indexes into a seg descriptor table
  - [gdt or ldt, index into table, protection level]
  - global descriptor table (gdt), local descriptor table (ldt)
- Yields a 64-bit segment descriptor, which points to a segment
  - [base addr, limit, privilege level, etc]

- If paging is on, the address is divided into [dir, page, offset]

x86: Interrupts

- 256 interrupts: 0–31 hw, rest sw (traps, exceptions, faults, etc)
- Interrupt indexes into a interrupt descriptor table (idt)
- Yields a 64-bit interrupt gate, which points to interrupt handler
  - [seg selector, offset, descriptor privilege level (dpl), etc]
- If interrupt-handler's privilege-level = cpu's privilege-level: cpu pushes on its current stack
  - its eflags, cs, eip, and an error code (for some interrupts)
- If interrupt-handler's privilege-level < cpu's privilege-level: cpu uses another stack whose location is in a task state segment (tss)
  - pushes its ss and esp // interrupted task's stack
  - pushes eflags, cs, eip, error code (if present)
- Return-from-interrupt (IRET) undoes the above (in both cases)

x86: Registers

- eax, ebx, ecx, esi, edi, edx: “general purpose” (32-bit)
- esp (32-bit): stack pointer (in ss segment)
- ebp (32-bit): frame pointer (in ss segment)
- eip (32-bit): instruction pointer (in cs segment)

- segment registers (16-bit), each holds a segment selector
  - cs (code segment), ss (stack segment)
  - ds, es, fs, gs (data segment)

- gdtr (48-bit): addr and size of current gdt
- idtr (48-bit): addr and size of current idt
- ldtr (16-bit): selector to current ldt (via gdt)
- tr (16-bit): selector to current tss (via gdt)

- eflags (32-bit): carry, overflow, sign, interrupt enable, etc
- cr0–cr4 (32-bit): paging enable, page fault, cache enable, etc.

Local APICs and IO APIC

- BIOS stores APICs config info at certain addresses
- Local APIC info starts at `0xFEE00000` (`APIC_Addr`)
  - offset `0x20` (`APIC_ID`) stores the apic id (= cpu id) // 0, 1, ...
- Get_CPU_ID(): // return cpu id of caller thread
  - disable interrupts
  - apicid ← read location `APIC_Addr + APIC_ID`
  - restore interrupts
  - return apicid
- IO APIC info starts at `0xFEC00000` (`IO_APIC_Addr`)

PIT timer + LAPIC timers

- PIT timer: interrupt `TIMER_IRQ` (=0)
- Each Local APIC has a timer: interrupt 32
- PIT timer is used only at boot to calibrate the LAPIC timers

- Global and static variables
  - `g_numTicks` // global tick counter
  - `DEFAULT_MAX_TICKS = 4` // default quantum
  - `g_Quantum = DEFAULT_MAX_TICKS`

LAPIC timer

```
Timer_Interrupt_Handler(istate):     // simplified
    id ← Get_CPU_ID()
    ct ← get_current_thread()
    if id is 0:
      ++g_numTicks
    ++ct.numTicks
    if ct.numTicks >= g_Quantum:
      g_needReschedule[id]
```

```
Init_Timer():
    Install_IRQ(32, Timer_Interrupt_Handler)
    enable interrupt 32
```

```
Init_Local_APIC(cpuid):
    Install_IRQ(39, Spurious_Interrupt_Handler)   // SMP
    enable interrupt 39
    set timer timeout value          // cpu 0 uses PIT to calibrate
```

VGA screen

- Ports: `CRT_*` regs (`0x3D4`, `0x3D5`, etc)
  - access via io instr // eg, `Out_Byte(port, value)`
  - for refresh, scan rate, blanking, cursor control, etc
- Video memory: `VIDMEM` (`0xb8000`–`0x100000`)
  - holds characters to display // `NUMROWS = 25`, `NUMCOLS = 80`
  - access via read/write instrs // eg, `VIDMEM[offset] = keycode`
- Var console_state: row, col, esc, numeric arg, etc
- `Update_Cursor()` based on console state // ports used here only
- `Put_Char_Imp(c)`: place char c at text cursor position
- `Init_Screen()`: clear screen, set "text cursor" to origin
- `Print(*fmt, ...)`

Keyboard

- Ports
  - input reg: `KB_DATA` (`0x60`)
  - control reg: `KB_CMD` (`0x64`)
  - status regs: `KB_OUTPUT_FULL` (0x01), `KB_KEY_RELEASE` (0x80)
- Interrupt: `KB_IRQ` (1)
- Static variables (for drivers, interrupt handler)
  - `s_queue` // queue for incoming keycodes
  - `s_keyboardWaitQueue` // threads waiting for kbd inputs
  - `s_kbdQueueLock` // spinlock protecting s_queue
  - scantables // map scancode to keycode
  - kbd state // shift, esc, control, alt, etc

Keyboard

- `Keyboard_Interrupt_Handler(istate)`:

```
if ports indicate byte available:
    get byte; convert to keycode or update kbdstate
    add keycode to s_queue              // drop if full; spinlock ops
    wakeup(s_keyboardWaitQueue)
```

- `Init_Keyboard()`:

```
initialize static variables
Install_IRQ(KB_IRQ, Keyboard_Interrupt_Handler)
enable kbd interrupt
```

- `Wait_For_Key()`:

```
disable intrpt
repeat
    if s_queue has key, get it                        // spinlock ops
    else wait(s_keyboardWaitQueue)
until got key
restore intrpt
```

IDE

- 16-bit transfer unit
- 2 hard disks
- PIO and DMA modes
- 256-byte blocks

- Ports
  - IDE_identify regs // show disk features
  - IDE_drive/cylinder/head/sector regs // target disk block
  - IDE_command reg // read/write
  - IDE_data reg // successive words of io block show up here
  - IDE_status/control/etc regs // busy, dma, interrupt, etc

IDE: drivers

- Static variables
  - `s_ideWaitQueue`: ide server thread waits here
  - `s_ideRequestQueue`: io requests queued here
- `IDE_Read(drive, blocknum, *buffer)`:

```
    convert blocknum to cylinder, head, sector
    update control and command regs
    read 256 words from data reg into buffer
```

- `IDE_Write(...)`: *like* `IDE_Read` *except* write to data reg
- `IDE_Request_Thread()`:

```
    forever: req = dequeue from request queue     // blocking
             IDE_Read/Write(req)                  // synchronous, pio
```

- `Init_IDE()`:

```
    register drives as block devices
    start kernel thread executing IDE_Request_Thread()
```

DMA controller (currently not used)

- Registers
  - memory addr
  - byte count
  - control regs (source, destination, transfer unit, etc)

- Usage for ide io
  - cpu sets up ide interface to initiate data transfer
  - cpu sets up dma interface

- `Init_DMA()`
- `Reserve_DMA(chan)`
- `Setup_DMA(direction, chan, *addr, size)`

Outline

Boot

- At power up, BIOS configures
  - one cpu-lapic as primary, with id 0
  - other cpu-lapics as secondaries, halted, with ids 1, 2, ...
  - MP config table in memory
  - loads diskc/block 0 (bootsect.asm) into memory
  - cpu 0 (in real mode) starts executing it

- bootsect.asm // executed by cpu 0
  - load the kernel image (from diskc) into memory and start executing it (setup.asm)

- setup.asm // executed by cpu 0
  - get memory size, redirect interrupts (bypass BIOS)
  - enter protected mode, set cs to `KERNEL_CS`
  - set ds, es, fs, gs, ss to `KERNEL_DS`, jump to `main.c:Main`

Kernel initialization: `Main()–1` // executed by cpu 0

- blank VGA screen
- init cpu 0's gdt, gdtr // `s_GDT[0]`; 1: code seg, 2: data seg
  // `NUM_GDT_ENTRIES = 32`
- organize memory into 4K pages // `g_pageList`, `s_freeList`
- init kernel heap
- init cpu 0's tss, tr, gdt`[3?]` // `s_theTSS[0]`; one tss per cpu
- init cpu 0's idt, idtr // `s_IDT[0]`
  - syscall entry's dpl at user level, others at kernel level
  - addresses of interrupt handlers in `g_interruptTable[0]`;
    set them to dummy interrupt handler
- init SMP: for each secondary cpu *i*
  - allocate a page for cpu *i*'s kernel stack (`CPUs[i].stack`)
  - start cpu *i* executing start_secondary_cpu (in setup.asm)
    // cpu *i* does its initialization, then spins until cpu 0 releases it

Kernel initialization: `Main()`–2 // executed by cpu 0

- init scheduler(0): create threads // with `Kernel_Thread` objects
    - current thread `{Main}` // `g_currentThreads[0]`
    - idle thread `{Idle-#0}` // `s_runQueue`
    - reaper thread `{Reaper}` // `s_runQueue`
- init traps: 12: stack exception; 13: GPF; 0x90: syscall
- init devices: Local_APIC(0), keyboard, IDE, DMA
- init PFAT: register filesystem PFAT with vfs
- release SMP
    - allow each secondary cpu to exit its initialization; wait for that
- mount root filesystem
    - mount `ide0` as PFAT fs at path "/a"
- spawn initial process // shell program
- hardware shutdown