Recall: Clock Algorithm (Not Recently Used)

Set of all pages in Memory

Single Clock Hand:
Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

• Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    » For example for catching modifications to code pages!

• Clock Algorithm: pages arranged in a ring
  - On page fault:
    » Advance clock hand (not real time)
    » Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Crude partitioning of pages into two groups: young and old
Second-Chance List Algorithm (VAX/VMS)

- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list
Reverse Page Mapping (Sometimes called “Coremap”)

- Physical page frames often shared by many different address spaces/page tables
  - All children forked from given process
  - Shared memory pages between processes
- Whatever reverse mapping mechanism that is in place must be very fast
  - Must hunt down all page tables pointing at given page frame when freeing a page
- Implementation options:
  - For every page descriptor, keep linked list of page table entries that point to it
    » Management nightmare - expensive
  - Linux 2.6: Object-based reverse mapping
    » Link together memory region descriptors instead (much coarser granularity)
Linux Memory Details?

- Memory management in Linux considerably more complex than the previous indications
- Memory Zones: physical memory categories
  - ZONE_DMA: < 16MB memory, DMAable on ISA bus
  - ZONE_NORMAL: 16MB ⇒ 896MB (mapped at 0xC0000000)
  - ZONE_HIGHMEM: Everything else (> 896MB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
- Many different types of allocation
  - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
  - Anonymous memory (not backed by a file, heap/stack)
  - Mapped memory (backed by a file)
Recall: Linux Virtual memory map

Kernel Addresses

User Addresses

32-Bit Virtual Address Space

64-Bit Virtual Address Space

0x00000000

0xC0000000

0xFFFFFFFF

0x0000000000000000

0x00007FFFFFFF

0xFFFF800000000000

0xFFFFFFFFFFFFFFFF

0x0FFFFFFF80000000000000

"Canonical Hole"

0x000007FFFFFFF

0x0000000000000000000000
Virtual Map (Details)

- Kernel memory not generally visible to user
  - Exception: special VDSO facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as gettimeofday()).

- Every physical page described by a “page” structure
  - Collected together in lower physical memory
  - Can be accessed in kernel virtual space
  - Linked together in various “LRU” lists

- For 32-bit virtual memory architectures:
  - When physical memory < 896MB
    - All physical memory mapped at 0xC0000000
  - When physical memory >= 896MB
    - Not all physical memory mapped in kernel space all the time
    - Can be temporarily mapped with addresses > 0xCC000000

- For 64-bit virtual memory architectures:
  - All physical memory mapped above 0xFFFFFFFF80000000
Page Frame Reclaiming Algorithm (PFRA)

• Several entrypoints:
  - Low on Memory Reclaiming: The kernel detects a “low on memory” condition
  - Hibernation reclaiming: The kernel must free memory because it is entering in the suspend-to-disk state
  - Periodic reclaiming: A kernel thread is activated periodically to perform memory reclaiming, if necessary

• Low on Memory reclaiming:
  - Start flushing out dirty pages to disk
  - Start looping over all memory nodes in the system
    » try_to_free_pages()
    » shrink_slab()
    » pdflush kernel thread writing out dirty pages

• Periodic reclaiming:
  - Kswapd kernel threads: checks if number of free page frames in some zone has fallen below pages_high watermark
  - Each zone keeps two LRU lists: Active and Inactive
    » Each page has a last-chance algorithm with 2 count
    » Active page lists moved to inactive list when they have been idle for two cycles through the list
    » Pages reclaimed from Inactive list
SLAB Allocator

- Replacement for free-lists that are hand-coded by users
  - Consolidation of all of this code under kernel control
  - Efficient when objects allocated and freed frequently

- Objects segregated into “caches”
  - Each cache stores different type of object
  - Data inside cache divided into “slabs”, which are continuous groups of pages (often only 1 page)
  - Key idea: avoid memory fragmentation
SLAB Allocator Details

• Based on algorithm first introduced for SunOS
  - Observation: amount of time required to initialize a regular object in the kernel exceeds the amount of time required to allocate and deallocate it
  - Resolves around object caching
    » Allocate once, keep reusing objects

• Avoids memory fragmentation:
  - Caching of similarly sized objects, avoid fragmentation
  - Similar to custom freelist per object

• Reuse of allocation
  - When new object first allocated, constructor runs
  - On subsequent free/reallocation, constructor does not need to be re-executed
Administrivia

- Unfortunately no HW3 judge!
  - Sample test cases were emailed out
- Group Issues
Next Objective
OS Basics: I/O

Software

OS Hardware Virtualization

Hardware

ISA

Processor

Controller

Memory

Protection Boundary

Storage

Networks

Inputs

Displays

Processes

Address Spaces

Windows

Sockets

Threads

Files

OS

Protection Boundary

Windows

Sockets

Address Spaces

Processes

Threads

Operating System
• I/O devices you recognize are supported by I/O Controllers
• Processors accesses them by reading and writing IO registers as if they were memory
  - Write commands and arguments, read status and results
The Requirements of I/O

- So far in this course:
  - We have learned how to manage CPU, memory

- What about I/O?
  - Without I/O, computers are useless (disembodied brains?)
  - But... thousands of devices, each slightly different
    » How can we standardize the interfaces to these devices?
  - Devices unreliable: media failures and transmission errors
    » How can we make them reliable???
  - Devices unpredictable and/or slow
    » How can we manage them if we don't know what they will do or how they will perform?
Operational Parameters for I/O

- **Data granularity: Byte vs. Block**
  - Some devices provide single byte at a time (e.g., keyboard)
  - Others provide whole blocks (e.g., disks, networks, etc.)

- **Access pattern: Sequential vs. Random**
  - Some devices must be accessed sequentially (e.g., tape)
  - Others can be accessed “randomly” (e.g., disk, cd, etc.)
    - Fixed overhead to start sequential transfer (more later)

- **Transfer Notification: Polling vs. Interrupts**
  - Some devices require continual monitoring
  - Others generate interrupts when they need service

- **Transfer Mechanism: Programmed IO and DMA**
Kernel Device Structure

The System Call Interface

- Process Management
  - Concurrency, multitasking
  - Architecture Dependent Code

- Memory Management
  - Virtual memory
  - Memory Manager

- Filesystems
  - Files and dirs: the VFS
  - File System Types
  - Block Devices

- Device Control
  - TTYs and device access
  - Device Control

- Networking
  - Connectivity
  - Network Subsystem
  - IF drivers
The Goal of the I/O Subsystem

- Provide Uniform Interfaces, Despite Wide Range of Different Devices
  
  - This code works on many different devices:
    
    ```
    FILE fd = fopen("/dev/something","rw");
    for (int i = 0; i < 10; i++) {
        printf(fd,"Count %d\n",i);
    }
    close(fd);
    ```
  
  - Why? Because code that controls devices ("device driver") implements standard interface.

- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  
  - Can only scratch surface!
Want Standard Interfaces to Devices

- **Block Devices:** e.g. disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible

- **Character Devices:** e.g. keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing

- **Network Devices:** e.g. Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include `socket` interface
    » Separates network protocol from network operation
    » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes
How Does User Deal with Timing?

• **Blocking Interface:** “Wait”
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data

• **Non-blocking Interface:** “Don’t Wait”
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing

• **Asynchronous Interface:** “Tell Me Later”
  - When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Chip-scale features of Recent x86 (SandyBridge)

- **Significant pieces:**
  - Four OOO cores
    - New Advanced Vector eXtensions (256-bit FP)
    - AES instructions
    - Instructions to help with Galois-Field mult
    - 4 $\mu$-ops/cycle
  - Integrated GPU
  - System Agent (Memory and Fast I/O)
  - Shared L3 cache divided in 4 banks
  - On-chip Ring bus network
    - High-BW access to L3 Cache

- **Integrated I/O**
  - Integrated memory controller (IMC)
    - Two independent channels of DDR3 DRAM
  - High-speed PCI-Express (for Graphics cards)
  - DMI Connection to SouthBridge (PCH)
SandyBridge I/O: PCH

- **Platform Controller Hub**
  - Used to be “SouthBridge,” but no “NorthBridge” now
  - Connected to processor with proprietary bus
    - Direct Media Interface
  - Code name “Cougar Point” for SandyBridge processors

- **Types of I/O on PCH:**
  - USB
  - Ethernet
  - Audio
  - BIOS support
  - More PCI Express (lower speed than on Processor)
  - SATA (for Disks)
Modern I/O Systems

network

PCI bus

monitor

graphics controller

bridge/memory controller

cache

SCSI bus

disk
disk
disk

general

IDE disk controller

expansion bus interface

keyboard

eexpansion bus

parallel port

serial port

disk
disk
disk

disk
Example: PCI Architecture

- CPU
- RAM
- Memory Bus
- Host Bridge
- PCI #0
- PCI #1
- ISA Bridge
- ISA Controller
- Legacy Devices
- PCI Slots
- USB Controller
- SCSI Controller
- CD ROM
- Scanner
- Hard Disk
- Root Hub
- Hub
- Webcam
- Mouse
- Keyboard
Example Device-Transfer Rates in Mb/s
(Sun Enterprise 6000)

- Device Rates vary over 12 orders of magnitude !!!
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices
How does the processor actually talk to the device?

- **CPU** interacts with a **Controller**
  - Contains a set of registers that can be read and written
  - May contain memory for request queues or bit-mapped images

- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - **I/O instructions**: in/out instructions
    » Example from the Intel architecture: `out 0x21, AL`
  - **Memory mapped I/O**: load/store instructions
    » Registers/memory appear in physical address space
    » I/O accomplished with load and store instructions
Example: Memory-Mapped Display Controller

- **Memory-Mapped:**
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the “frame buffer”) changes image on screen
    - Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - Say enter a set of triangles that describe some scene
      - Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - Say render the above scene
      - Addr: 0x0007F004

- Can protect with address translation
Transferring Data To/From Controller

• Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size

• Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly

• Sample interaction with DMA controller (from OSC):

1. Device driver is told to transfer disk data to buffer at address X
2. Device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. Disk controller initiates DMA transfer
4. Disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. When C = 0, DMA interrupts CPU to signal transfer completion
I/O Device Notifying the OS

• The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error

• **I/O Interrupt:**
  - Device generates an interrupt whenever it needs service
  - **Pro:** handles unpredictable events well
  - **Con:** interrupts relatively high overhead

• **Polling:**
  - OS periodically checks a device-specific status register
    - I/O device puts completion information in status register
  - **Pro:** low overhead
  - **Con:** may waste many cycles on polling if infrequent or unpredictable I/O operations

• Actual devices combine both polling and interrupts
  - For instance – High-bandwidth network adapter:
    - Interrupt for first incoming packet
    - Poll for following packets until hardware queues are empty
Device Drivers

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call

- **Device Drivers typically divided into two pieces**:
  - Top half: accessed in call path from system calls
    » implements a set of **standard, cross-device calls** like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    » This is the kernel’s interface to the device driver
    » Top half will start I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    » Gets input or transfers next block of output
    » May wake sleeping threads if I/O now complete
Summary

- **I/O Devices Types:**
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns:
    » Block Devices, Character Devices, Network Devices
  - Different Access Timing:
    » Blocking, Non-blocking, Asynchronous

- **I/O Controllers:** Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
  - Report their results through either interrupts or a status register that processor looks at occasionally (polling)

- **Notification mechanisms**
  - Interrupts
  - Polling: Report results through status register that processor looks at periodically

- **Drivers interface to I/O devices**
  - Provide clean Read/Write interface to OS above
  - Manipulate devices through PIO, DMA & interrupt handling
  - 2 types: block, character, and network