Recall: Clock Algorithm (Not Recently Used)

- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approximation to MIN)
  - Replace an old page, not the oldest page
- Details:
  - Hardware “use” bit per physical page (called “accessed” in Intel architecture):
    » Hardware sets use bit on each reference
    » If use bit isn’t set, means not referenced in a long time
    » Some hardware sets use bit in the TLB; must be copied back to page TLB entry gets replaced
- On page fault:
  » Advance clock hand (not real time)
  » Check use bit:
    1 → used recently; clear and leave alone
    0 → selected candidate for replacement

Recall: Second-Chance List Algorithm (Rearrangement)

- Mapped Pages
  - Active List: Marked: RW, List: FIFO
- Unmapped Pages
  - Second Chance List: Marked: Invalid, List: LRU
- New Active Page
- New SC Victim

Recall: Second-Chance List Algorithm (Page in from Disk)

- Mapped Pages
  - Active List: Marked: RW, List: FIFO
- Unmapped Pages
  - Second Chance List: Marked: Invalid
- Page-in From disk
- LRU victim
- New Active Page
- New SC Victim

- 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
- 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
Thrashing

• If a process does not have “enough” pages, the page-fault rate is very high.
  This leads to:
  – low CPU utilization
  – operating system spends most of its time swapping to disk
• Thrashing = a process is busy swapping pages in and out with little or no actual progress
• Questions:
  – How do we detect Thrashing?
  – What is best response to Thrashing?

Locality In A Memory-Reference Pattern

• Program Memory Access Patterns have temporal and spatial locality
  – Group of Pages accessed along a given time slice called the “Working Set”
  – Working Set defines minimum number of pages for process to behave well
• Not enough memory for Working Set $\Rightarrow$ Thrashing
  – Better to swap out process?

Working-Set Model

• $\Delta$ = working-set window $\Rightarrow$ fixed number of page references
  – Example: 10,000 instructions
• WSi (working set of Process Pi) = total set of pages referenced in the most recent $\Delta$ (varies in time)
  – if $\Delta$ too small will not encompass entire locality
  – if $\Delta$ too large will encompass several localities
  – if $\Delta = \infty \Rightarrow$ will encompass entire program
• $D = \Sigma |WSi|$ = total demand frames
• if $D > m \Rightarrow$ Thrashing
  – Policy: if $D > m$, then suspend/swap out processes
  – This can improve overall system behavior by a lot!

What about Compulsory Misses?

• Recall that compulsory misses are misses that occur the first time that a page is seen
  – Pages that are touched for the first time
  – Pages that are touched after process is swapped out/swapped back in
• Clustering:
  – On a page-fault, bring in multiple pages “around” the faulting page
  – Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
• Working Set Tracking:
  – Use algorithm to try to track working set of application
  – When swapping process back in, swap in working set
Linux Memory Details?

- Memory management in Linux considerably more complex than the examples we have been discussing.
- 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)
- Allocation priorities
  - Is blocking allowed/etc

Linux Virtual memory map (Pre-Meltdown)

- Kernel memory not generally visible to user
  - Exception: special VDSO (virtual dynamically linked shared objects) 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 0xFFFFF800000000000

Pre-Meltdown Virtual Map (Details)

- Kernel memory not generally visible to user
  - Exception: special VDSO (virtual dynamically linked shared objects) 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 0xFFFFF800000000000

Post Meltdown Memory Map

- Meltdown flaw (2018, Intel x86, IBM Power, ARM)
  - Exploit speculative execution to observe contents of kernel memory
  1: // Set up side channel (array flushed from cache)
  2: uchar array[256 * 4096];
  3: flush(array); // Make sure array out of cache
  4: try { // … catch and ignore SIGSEGV (illegal access)
  5: uchar result = *(uchar *) kernel_address;// Try access!
  6: uchar dummy = array[result * 4096]; // Leak info!
  7: } catch(){;} // Could use signal() and setjump/longjmp
  8: // scan through 256 array slots to determine which loaded
  - Some details:
    » Reason we skip 4096 for each value: avoid hardware cache prefetch
    » Note that value detected by fact that one cache line is loaded
    » Catch and ignore page fault: set signal handler for SIGSEGV, can use setjump/longjmp….
- Patch: Need different page tables for user and kernel
  - Without PCID tag in TLB, flush TLB twice on syscall (800% overhead!)
  - Need at least Linux v 4.14 which utilizes PCID tag in new hardware to avoid flushing when change address space
- Fix: better hardware without timing side-channels
  - Mostly implemented, but related problem (Spectre) much harder to fix
### Administrivia

- Welcome back from Spring Break
- Midterm 2 is graded: Min: 19.3, Max: 91.5, Mean: 57.4, StdDev: 14.3
  - Regrade requests closed
  - Regrades finished by tomorrow (hopefully)
- Midterm 3 on April 25
  - All topics up to previous Tuesday (4/23) are in scope
  - Closed book, 3 pages, double-sided handwritten notes.
- Extensions:
  - Homework 4 ⇒ Due Wednesday (4/3)
  - Project 2 ⇒ Due Friday (4/5)

### What about I/O???

**Diagram from "Computer Organization and Design" by Patterson and Hennessy**

**Requirements of I/O**

- So far in CS 162, we have studied:
  - Abstractions: the APIs provided by the OS to applications running in a process
  - Synchronization/Scheduling: How to manage the CPU
- 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?
Example: Device Transfer Rates in Mb/s (Sun Enterprise 6000)

- Device rates vary over 12 orders of magnitude!!!
- System must be able to handle this wide range
  - Better not have high overhead/byte for fast devices
  - Better not waste time waiting for slow devices

Example: I/O System

Expansion through hierarchy of buses!
Recall: Recent Intel Chipset I/O Configuration

- Direct-connect High Speed PCIe
- Really High Speed I/O (e.g., graphics)
- High-Speed I/O devices (PCIe)
- Disks (8 x SATA3.0)
- Slower I/O (USB)
- Integrated 2.5G Ethernet
- HD Audio
- RAID 0/1/5/10
- Intel Management Engine (ME) and BIOS Support
- Integrated WiFi 6E
- Integrated WiFi 7
- Integrated Ethernet
- Lots of expansion through buses!

What’s a bus?

- Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
  - Operations: e.g., Read, Write
  - Control lines, Address lines, Data lines
  - Typically multiple devices
- Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address, length, data
- Very high BW close to processor (wide, fast, and inflexible), low BW with high flexibility out in I/O subsystem

Why a Bus?

- Buses let us connect \( n \) devices over a single set of wires, connections, and protocols
  - \( O(n^2) \) relationships with 1 set of wires (!)
- Downside: Only one transaction at a time
  - The rest must wait
  - “Arbitration” aspect of bus protocol ensures the rest wait

PCI Bus Evolution

- PCI started life out as a physical (parallel) bus
  - Example: 32-bit system \( \Rightarrow \) 32 address wires, 32 data wires, power, control
- But a parallel bus has many limitations
  - Multiplexing address/data for many requests
  - Slowest devices must be able to tell what’s happening (e.g., for arbitration)
  - Capacitance increases with each device you attach \( \Rightarrow \) Slowing down bus accesses!
  - Bus speed is set to that of the slowest device
PCI Express (PCIe) “Bus”

- No longer a parallel bus
- Really a **collection of fast serial channels** or “lanes”
- Devices can use as many serial channels as they need to achieve a desired bandwidth
  - 1X, 2X, 4X, 8X, 16X
- Slow devices don’t have to share with fast ones
- One of the successes of device abstraction in Linux was the ability to migrate from PCI to PCI Express
  - The physical interconnect changed completely, but the old API still worked

Example: PCI Architecture

**How does the Processor 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, etc.
- Processor accesses registers in two ways:
  - **Port-Mapped I/O**: in/out instructions
    - Example from the Intel architecture: out 8x21, AL
  - **Memory-mapped I/O**: load/store instructions
    - Registers/memory appear in physical address space
    - I/O accomplished with load and store instructions

Port-Mapped I/O in Pintos Speaker Driver
Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by HW jumpers or at boot time
  - Simply writing to display memory (also called the “frame buffer”) changes image on screen
    - Addr: 0x8000F000 — 0x8000FFFF
  - Writing graphics description to cmd queue
    - Say enter a set of triangles describing 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

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 transfers
    - Some devices require continual monitoring
    - Others generate interrupts when they need service

- Transfer Mechanism: Programmed IO and DMA

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 book):
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
The Goal of the I/O Subsystem

• Provide Uniform Interfaces, Despite Wide Range of Different Devices
  – This code works on many different devices:
    ```c
    FILE fd = fopen("/dev/something", "rw");
    for (int i = 0; i < 10; i++) {
      fprintf(fd, "Count %d
", 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

Conclusion

• 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
• Notification mechanisms
  – Interrupts
  – Polling: Report results through status register that processor looks at
    periodically
• Device drivers interface to I/O devices
  – Provide clean Read/Write interface to OS above
  – Manipulate devices through PIO, DMA & interrupt handling
  – Three types: block, character, and network