Recall: Clock Algorithm (Not Recently Used)

- **Clock Algorithm**: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approximation to 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 (VAX/VMS)

- **Directly Mapped Pages**: Marked: RW, List: FIFO
- **Marked: RW, List: FIFO**: Page-in
- **Overflow, LRU victim**: New Active Pages
- **Second Chance List**: New SC Victims

Free List

- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique (“Pageout demon”)
  - Dirty pages start copying back to disk when enter list
- **Like VAX second-chance list**
  - If page needed before reused, just return to active set
- **Advantage**: faster for page fault
  - Can always use page (or pages) immediately on fault
Reverse Page Mapping (Sometimes called “Coremap”)

• When evicting a page frame, how to know which PTEs to invalidate?
  – Hard in the presence of shared pages (forked processes, shared memory, …)
• Reverse mapping mechanism must be very fast
  – Must hunt down all page tables pointing at given page frame when freeing a page
  – Must hunt down all PTEs when seeing if pages “active”
• Implementation options:
  – For every page descriptor, keep linked list of page table entries that point to it
    » Management nightmare – expensive
  – Linux: Object-based reverse mapping
    » Link together memory region descriptors instead (much coarser granularity)

Allocation of Page Frames (Memory Pages)

• How do we allocate memory among different processes?
  – Does every process get the same fraction of memory? Different fractions?
  – Should we completely swap some processes out of memory?
• Each process needs minimum number of pages
  – Want to make sure that all processes that are loaded into memory can make forward progress
  – Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    » Instruction is 6 bytes, might span 2 pages
    » 2 pages to handle from
    » 2 pages to handle to
• Possible Replacement Scopes:
  – Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  – Local replacement – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

• Equal allocation (Fixed Scheme):
  – Every process gets same amount of memory
  – Example: 100 frames, 5 processes → process gets 20 frames
• Proportional allocation (Fixed Scheme)
  – Allocate according to the size of process
  – Computation proceeds as follows:
    \[ s_i = \text{size of process } p_i \text{ and } S = \sum s_i \]
    \[ m = \text{total number of physical frames in the system} \]
    \[ a_i = \text{(allocation for } p_i) = \frac{s_i}{S} \times m \]
• Priority Allocation:
  – Proportional scheme using priorities rather than size
    » Same type of computation as previous scheme
  – Possible behavior: If process \( p_i \) generates a page fault, select for replacement a frame from a process with lower priority number
• Perhaps we should use an adaptive scheme instead???
  – What if some application just needs more memory?

Page-Fault Frequency Allocation

• Can we reduce Capacity misses by dynamically changing the number of pages/application?
  – Establish “acceptable” page-fault rate
    – If actual rate too low, process loses frame
    – If actual rate too high, process gains frame
• Question: What if we just don’t have enough memory?
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 = 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
Adminstrivia

- Still Grading Midterm 2
  - Done tonight!!!
  - We will release solutions at the same time that we release grades
- Both Homework 4 and Project 2 are due in week after Spring break
  - Don’t wait until end of Spring break!
- Midterm 3: April 27
  - Ok, so this is a while yet…!

- Enjoy Spring Break!!!
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 setjmp/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
  - 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
    - Will be coming, but still in works

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

In a Picture

- 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

Modern I/O Systems

- 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 bus
- 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)
  - Bus speed is set to that of the slowest device

**PCI Express “Bus”**

- No longer a parallel bus
- Really a collection of fast serial channels or “lanes”
- Devices can use as many as they need to achieve a desired bandwidth
- 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 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 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

Port-Mapped I/O in Pintos Speaker Driver

- There’s more than just a CPU in there!
Chip-scale Features of 2015 x86 (Sky Lake)

- Significant pieces:
  - Four OOO cores with deeper buffers
    » Intel MPX (Memory Protection Extensions)
    » Intel SGX (Software Guard Extensions)
    » Issue up to $6 \mu$-ops/cycle
  - GPU, System Agent (Mem, Fast I/O)
  - Large shared L3 cache with on-chip ring bus
    » 2 MB/core instead of 1.5 MB/core
    » High-BW access to L3 Cache
- Integrated I/O
  - Integrated memory controller (IMC)
    » Two independent channels of DRAM
  - High-speed PCI-Express (for Graphics cards)
  - Direct Media Interface (DMI) Connection to PCH (Platform Control Hub)

Sky Lake I/O: PCH

- Platform Controller Hub
  - Connected to processor with proprietary bus
    » Direct Media Interface
- Types of I/O on PCH:
  - USB, Ethernet
  - Thunderbolt 3
  - Audio, BIOS support
  - More PCI Express (lower speed than on Processor)
  - SATA (for Disks)

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):
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

Kernel Device Structure

- 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
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete
Recall: Life Cycle of An I/O Request

User Program

Device Driver

Kernel I/O Subsystem

Device Hardware

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\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
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