CS162
Operating Systems and
Systems Programming
Lecture 16

Memory 4: Demand Paging Policies
General I/O

October 21\textsuperscript{th}, 2021
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
Recall: Example: FIFO (strawman)

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

<table>
<thead>
<tr>
<th>Ref: Page:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since need A again right away
Example: MIN / LRU

• Suppose we have the same reference stream:
  – A B C A B D A D B C B

• Consider MIN Page replacement:

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• MIN: 5 faults
  – Where will D be brought in? Look for page not referenced farthest in future

• What will LRU do?
  – Same decisions as MIN here, but won’t always be true!
Consider the following: A B C D A B C D A B C D

LRU Performs as follows (same as FIFO here):

- Every reference is a page fault!
- Fairly contrived example of working set of N+1 on N frames
When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Every reference is a page fault!
- MIN Does much better:

<table>
<thead>
<tr>
<th>Ref: Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td>D</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• One desirable property: When you add memory the miss rate drops (stack property)
  – Does this always happen?
  – Seems like it should, right?
• No: Bélády’s anomaly
  – Certain replacement algorithms (FIFO) don’t have this obvious property!
Adding Memory Doesn’t Always Help Fault Rate

• Does adding memory reduce number of page faults?
  – Yes for LRU and MIN
  – Not necessarily for FIFO! (Called Bélády’s anomaly)

<table>
<thead>
<tr>
<th>Ref: Page:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9 page faults

<table>
<thead>
<tr>
<th>Ref: Page:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10 page faults!

• After adding memory:
  – With FIFO, contents can be completely different
  – In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page
Approximating LRU: Clock Algorithm

- **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
Clock Algorithm Example

- Free frame
- Page

Diagram showing clock algorithm example with a sequence of page references and frame allocation.
Clock Algorithm Example: Page Fault
Clock Algorithm Example: Page Fault

- Free frame
- Page

Use: 0
Use: 0
Use: 0
Use: 1
Use: 1
Use: 1
Free frame
Clock Algorithm Example: Page Fault

Free frame
Page

use: 0

use: 0

use: 0

use: 0

use: 0

use: 0

use: 0
Clock Algorithm Example: Page Fault

Free frame

Page

use: 0

use: 0

use: 0

use: 0

This is "0". We can replace it!
Clock Algorithm Example: Page Fault

- Free frame
- Page

Save the page, if “dirty”; invalidate TLB and PTE
Clock Algorithm Example: Page Fault

Free frame
Page

Load page; update PTE
Clock Algorithm Example

- Free frame
- Page

Access page (red or write)
Clock Algorithm Example: Another Page Fault

- Free frame
- Page

use: 0
use: 1
use: 1
use: 0
Clock Algorithm Example: Another Page Fault

Free frame
Page

Free frame

use: 0
use: 1
use: 0
use: 0
use: 0
use: 0

use: 0
Clock Algorithm Example: Another Page Fault

Free frame; Load page; update PTE
Clock Algorithm: More details

• Will always find a page or loop forever?
  – Even if all use bits set, will eventually loop all the way around ⇒ FIFO

• What if hand moving slowly?
  – Good sign or bad sign?
    » Not many page faults
    » or find page quickly

• What if hand is moving quickly?
  – Lots of page faults and/or lots of reference bits set

• One way to view clock algorithm:
  – Crude partitioning of pages into two groups: young and old
  – Why not partition into more than 2 groups?
**N\textsuperscript{th} Chance version of Clock Algorithm**

- **N\textsuperscript{th} chance algorithm**: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1 → clear use and also clear counter (used in last sweep)
    - 0 → increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced

- **How do we pick N?**
  - Why pick large N? Better approximation to LRU
    - If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - Otherwise, might have to look a long way to find free page

- **What about “modified” (or “dirty”) pages?**
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - Clean pages, use N=1
    - Dirty pages, use N=2 (and write back to disk when N=1)
Recall: Meaning of PTE bits

- Which bits of a PTE entry are useful to us for the Clock Algorithm? Remember Intel PTE:

  - The “Present” bit (called “Valid” elsewhere):
    » P==0: Page is invalid, and a reference will cause page fault
    » P==1: Page frame number is valid and MMU is allowed to proceed with translation
  - The “Writable” bit (could have opposite sense and be called “Read-only”):
    » W==0: Page is read-only and cannot be written.
    » W==1: Page can be written
  - The “Accessed” bit (called “Use” elsewhere):
    » A==0: Page has not been accessed (or used) since last time software set A→0
    » A==1: Page has been accessed (or used) since last time software set A→0
  - The “Dirty” bit (called “Modified” elsewhere):
    » D==0: Page has not been modified (written) since PTE was loaded
    » D==1: Page has changed since PTE was loaded
Clock Algorithms Variations

• Do we really need hardware-supported “modified” bit?
  – No. Can emulate it using read-only bit
    » Need software DB of which pages are allowed to be written (needed this anyway)
    » We will tell MMU that pages have more restricted permissions than the actually do to force page faults (and allow us notice when page is written)
  – Algorithm (Clock-Emulated-M):
    » Initially, mark all pages as read-only \((W\rightarrow 0)\), even writable data pages.
      Further, clear all software versions of the “modified” bit \(\rightarrow 0\) (page not dirty)
    » Writes will cause a page fault. Assuming write is allowed, OS sets software “modified” bit \(\rightarrow 1\), and marks page as writable \((W\rightarrow 1)\).
    » Whenever page written back to disk, clear “modified” bit \(\rightarrow 0\), mark read-only
Clock Algorithms Variations (continued)

• Do we really need a hardware-supported “use” bit?
  – No. Can emulate it similar to above (e.g. for read operation)
    » Kernel keeps a “use” bit and “modified” bit for each page
  – Algorithm (Clock-Emulated-Use-and-M):
    » Mark all pages as invalid, even if in memory.
      Clear emulated “use” bits $\rightarrow 0$ and “modified” bits $\rightarrow 0$ for all pages (not used, not dirty)
    » Read or write to invalid page traps to OS to tell use page has been used
    » OS sets “use” bit $\rightarrow 1$ in software to indicate that page has been “used”.
      Further:
      1) If read, mark page as read-only, $W \rightarrow 0$ (will catch future writes)
      2) If write (and write allowed), set “modified” bit $\rightarrow 1$, mark page as writable ($W \rightarrow 1$)
    » When clock hand passes, reset emulated “use” bit $\rightarrow 0$ and mark page as invalid again
    » Note that “modified” bit left alone until page written back to disk

• Remember, however, clock is just an approximation of LRU!
  – Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  – Need to identify an old page, not oldest page!
  – Answer: second chance list
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 in SC List: move to it front of Active list, mark it RW
  - Not in SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list
Second-Chance List Algorithm (continued)

• How many pages for second chance list?
  – If 0 ⇒ FIFO
  – If all ⇒ LRU, but page fault on every page reference

• Pick intermediate value. Result is:
  – Pro: Few disk accesses (page only goes to disk if unused for a long time)
  – Con: Increased overhead trapping to OS (software / hardware tradeoff)

• With page translation, we can adapt to any kind of access the program makes
  – Later, we will show how to use page translation / protection to share memory between threads on different machines

• History: The VAX architecture did not include a “use” bit. Why did that omission happen???
  – Strecker (architect) asked OS people, they said they didn’t need it, so didn’t implement it
  – He later got blamed, but VAX did OK anyway
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:
    \[
    s_i = \text{size of process } p_i \quad \text{and} \quad S = \sum s_i \\
    m = \text{total number of physical frames in the system} \\
    a_i = (\text{allocation for } p_i) = \frac{s_i}{S} \times m
    \]

• 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 ⇒ Thrashing
  - Better to swap out process?
Working-Set Model

- \( \Delta \equiv \) working-set window \( \equiv \) 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
- \( D = \Sigma |WSi| \equiv \) 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
Recall: Five Components of a Computer

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?
Recall: OS Basics: I/O

- OS provides common services in form of I/O
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

- Monitor
- Graphics controller
- Bridge/memory controller
- IDE disk controller
- Disk
- Disk
- Disk
- Cache
- SCSI controller
- Expansion bus interface
- Expansion bus
- Keyboard
- Parallel port
- Serial port
- Network
- Printer
- Monitor
- Mouse
- Hard drives
- CD/DVD drives
- System unit
- PCI bus
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 (Peripheral Component Interconnect) 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

- RAM
- CPU
- Memory Bus
- Host Bridge
- ISA Bridge
- ISA Controller
- Legacy Devices
- PCI Slots
- Root Hub
- USB Controller
- Hub
- Webcam
- SATA Controller
- DVD ROM
- Scanner
- Hard Disk
- Mouse
- Keyboard
- PCI #0
- PCI #1
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
- Can protect with address translation
Summary (1/2)

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not “in use”
  - If page not “in use” for one pass, than can replace
- Nth-chance clock algorithm: Another approximate LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approximate LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn’t fit in memory
  - Need to swap out a process
Summary (2/2)

• 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