Recall: Precise Exceptions

- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Recall: Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    » Resulting trap is a “Page Fault”
    - What does OS do on a Page Fault?:
      » Choose an old page to replace
      » If old page modified (“D=1”), write contents back to disk
      » Change its PTE and any cached TLB to be invalid
      » Load new page into memory from disk
      » Update page table entry, invalidate TLB for new entry
      » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    » Suspended process sits on wait queue

Summary: Steps in Handling a Page Fault
Management & Access to the Memory Hierarchy

Some questions for this lecture!

- During a page fault, where does the OS get a free frame?
  - Keeps a free list
  - Unix runs a "reaper" if memory gets too full
  - As a last resort, evict a dirty page first
- How can we organize these mechanisms?
  - Work on the replacement policy
- How many page frames/process?
  - Like thread scheduling, need to "schedule" memory resources:
    » utilization? fairness? priority?
    - allocation of disk paging bandwidth

What Factors Lead to Misses?

- **Compulsory Misses:**
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    » Prefetching: loading them into memory before needed
    » Need to predict future somehow! More later.
- **Capacity Misses:**
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    » One option: Increase amount of DRAM (not quick fix!)
    » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Conflict Misses:**
  - Technically, conflict misses don’t exist in virtual memory, since it is a “fully-associative” cache
- **Policy Misses:**
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

Demand Paging Cost Model

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - \[ \text{EAT} = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time} \]
  - \[ \text{EAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \]
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose \( p = \) Probability of miss, \( 1-p = \) Probably of hit
  - Then, we can compute EAT as follows:
    \[ \text{EAT} = 200ns + p \times 8ms \]
    \[ = 200ns + p \times 8,000,000ns \]
  - If one access out of 1,000 causes a page fault, then \( \text{EAT} = 8.2 \mu s \):
    - This is a slowdown by a factor of 40!
  - What if want slowdown by less than 10%?
    - \[ 200ns \times 1.1 < \text{EAT} \Rightarrow p < 2.5 \times 10^{-6} \]
    - This is about 1 page fault in 400000!
Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out

- FIFO (First In, First Out)
  - Throw out oldest page. Be fair - let every page live in memory for some amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages

- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future...
  - Makes good comparison case, however

- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable - makes it hard to make real-time guarantees

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.

- How to implement LRU? Use a list!
  - On each use, remove page from list and place at head
  - LRU page is at tail

- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list...
  - Many instructions for each hardware access

- In practice, people approximate LRU (more later)

Example: FIFO

- 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:

- FIFO: 7 faults.
- When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

- 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!
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</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

- Every reference is a page fault!

- MIN Does much better:

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

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 Belady's anomaly)

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

- 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

Graph of Page Faults Versus The Number of Frames

- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: Belady's anomaly
  - Certain replacement algorithms (FIFO) don't have this obvious property!

Administrivia

- Problems with website (cs162.eecs.Berkeley.edu)
  - Ran out of space/crashed yesterday
  - Restore of bad checkpoint caused phantom HW3 to appear (it was last year's version)
  - Everything should be ok now - please check
- No sections this week!
- Spring Break is next week!
  - No class!
- Still working on the grading of exams
  - No deadline yet, will let you know
  - Solutions are done!
    - Will be posted on new handout link shortly
- Checkpoint 1 moved to after Spring Break
  - Monday, 3/30
Implementing LRU

- **Perfect:**
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- **Clock Algorithm:** Arrange physical pages in circle with single clock hand
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page
- **Details:**
  - Hardware “use” bit per physical page:
    - Hardware sets use bit on each reference
    - If use bit isn’t set, means not referenced in a long time
    - Nachos hardware sets use bit in the TLB; you have to copy this back to page table when 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
  - Will always find a page or loop forever?
    - Even if all use bits set, will eventually loop around → FIFO

Clock Algorithm: Not Recently Used

- **Single Clock Hand:**
  - Advances only on page fault!
  - Check for pages not used recently
  - Mark pages as not used recently
- **Set of all pages in Memory**
- **What if hand moving slowly?**
  - Good sign or bad sign?
    - Not many page faults and/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?

Nth Chance version of Clock Algorithm

- **Nth 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 approx 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 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)

Clock Algorithms: Details

- **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!
  - **Do we really need hardware-supported “modified” bit?**
    - No. Can emulate it (BSD Unix) using read-only bit
      - Initially, mark all pages as read-only, even data pages
      - On write, trap to OS. OS sets software “modified” bit, and marks page as read-write.
      - Whenever page comes back in from disk, mark read-only
Clock Algorithms Details (continued)

- Do we really need a hardware-supported “use” bit?
  - No. Can emulate it similar to above:
    - Mark all pages as invalid, even if in memory
    - On read to invalid page, trap to OS
    - OS sets use bit, and marks page read-only
  - Get modified bit in same way as previous:
    - On write, trap to OS (either invalid or read-only)
    - Set use and modified bits, mark page read-write
  - When clock hand passes by, reset use and modified bits
    and mark page as invalid again
* Remember, however, that 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 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

Second-Chance List Algorithm (con't)

- How many pages for second chance list?
  - If 0 \(\Rightarrow\) FIFO
  - If all \(\Rightarrow\) 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 widely
    separated machines
* Question: why didn’t VAX include “use” bit?
  - 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
Demand Paging (more details)

• Does software-loaded TLB need use bit?
Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU

• Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    » Can’t push page out to disk without invalidating all PTEs

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:
    \[ a_i = \frac{s_i}{S} \times m \]
    \[ s_i = \text{size of process } p_i \text{, and } S = \sum s_i \]
    \[ m = \text{total number of frames} \]

• 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?
• 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
• 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 needed for process to behave well
- **Not enough memory for Working Set** => Thrashing
  - Better to swap out process?

---

**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
You are here ...

Course Structure: Spiral

OS Basics: I/O

In a picture

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?

• Some operational parameters:
  - Byte/Block
    » Some devices provide single byte at a time (e.g. keyboard)
    » Others provide whole blocks (e.g. disks, networks, etc)
  - Sequential/Random
    » Some devices must be accessed sequentially (e.g. tape)
    » Others can be accessed randomly (e.g. disk, cd, etc.)
  - Polling/Interrupts
    » Some devices require continual monitoring
    » Others generate interrupts when they need service
Kernel Device Structure

The System Call Interface

- Process Management
- Memory Management
- Filesystems
- Device Control
- Networking
- Concurrency, multitasking
- Virtual memory
- Files and dirs: the VFS
- TTYs and device access
- Connectivity

Architecture Dependent Code

Memory Manager

File System Types

Block Devices

Device Control

Network Subsystem

IF drivers

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

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!

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 μ-ops/cycle
  - Integrated GPU
  - System Agent (Memory and Fast I/O)
  - Shared L3 cache divided in 4 banks
  - On-chip Ring bus network
    - Both coherent and non-coherent transactions
    - 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)
**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 a 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):**

---

![Diagram of processor-memory bus](image-url)
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
  - 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

Summary (1/2)

- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started

- 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 approx LRU
  - Give pages multiple passes of clock hand before replacing

- Second-Chance List algorithm: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults

Summary (2/2)

- 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

- 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)