Recall: Some following questions

- 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

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 \( p = \text{Probability of miss}, 1-p = \text{Probability of hit} \)
  - Then, we can compute EAT as follows:
    \[
    \text{EAT} = 200\text{ns} + p \times 8\text{ ms}
    \]
    \[
    = 200\text{ns} + p \times 8,000,000\text{ns}
    \]

- 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%?
  - \( 200\text{ns} \times 1.1 < \text{EAT} \Rightarrow p < 2.5 \times 10^{-6} \)
  - This is about 1 page fault in 400000!
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

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
    same amount of time.
  – Bad – throws out heavily used pages instead of infrequently used

• 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!

   Head
   Page 6 — Page 7 — Page 1 — Page 2 — Tail (LRU)

   – 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:

<table>
<thead>
<tr>
<th>Ref.</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></td>
<td></td>
<td>D</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>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></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

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

<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>

- 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 Bélády’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 drops
  - Does this always happen?
  - Seems like it should, right?
- No: Bélády’s anomaly
  - Certain replacement algorithms (FIFO) don’t have this obvious property!
**Administrivia**

- Peer review is *NOT* optional
  - Every person must fill out the project 1 peer review
  - Due today Wed 3/16
    - We will consider taking off points for missing reviews
  - The peer review is an important part of our evaluation of partner dynamics – please take is very seriously

- Survey on Piazza: Please tell us how the course is going!
  - What is going well, what is not going well
  - What could we change?

- Project 2 has been released
  - Get started early as design doc is due Monday 3/28

---

**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
    - Some hardware sets use bit in the TLB; you have to copy this back to page table entry 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

- 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

Directly Mapped Pages
- Marked: RW
- List: FIFO

Page-in From disk
- New Active Pages
- New SC Victims

LRU victim
Second Chance List
Marked: Invalid
List: LRU
Second-Chance List Algorithm (con’t)

- 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 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
- Keep set of free pages ready for use in demand paging
- 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:
    \[
    s_i = \text{size of process } p_i \text{ and } S = \sum s_i \\
    m = \text{total number of frames} \\
    a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m
    \]

• **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

• 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?

Working-Set Model

- $\Delta = \text{working-set window} = \text{fixed number of page references}$
  - Example: 10,000 instructions
- $WS_i = \text{total set of pages referenced in the most recent } \Delta \text{ (varies in time)}$
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty$ ⇒ will encompass entire program
- $D = \Sigma |WS_i| = \text{total demand frames}$
- if $D > m$ ⇒ 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

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
  - 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 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)
- Allocation priorities
  - Is blocking allowed/etc

Recall: Linux Virtual memory map

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 0xFFFF800000000000
Summary

• 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