Outline

• Demand paging
• Replacement policies
  • FIFO, MIN, LRU
• Clock algorithm
• N\textsuperscript{th}-chance clock algorithm
Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30% per year
- But they don’t use all their memory most of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as cache for disk
Since Demand Paging is Caching, One Must Ask …

• What is block size?
  • One page

• What is organization of cache structure? (i.e. direct mapped, set-associative, fully associative)
  • Fully associative: Mapping arbitrary virtual page → any physical page

• How do we find pages in cache?
  • First check TLB, then page-table traversal

• What is page replacement policy? (i.e. LRU, Random…)
  • This requires more explanation… (it’s kinda LRU)

• What happens on misses?
  • Go to lower level to fill miss (i.e. disk)

• What happens on writes? (write-through, write back)
  • Write-back – need dirty bit!
Next Up: What Happens When …

Process

Instruction

Virtual address

MMU

V-Page#

PT

P-Page#

Memory

Disk

Update PT entry

load page from disk

Scheduler

Retry

Operating System

Page fault exception

Page fault handler
Recall: Page Table Entry

- What is in each Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- P: Present (same as “valid” bit in other architectures)
- W: Writeable
- U: User accessible
- PWT: Page write transparent: external cache write-through
- PCD: Page cache disabled (page cannot be cached)
- A: Accessed: page has been accessed recently
- D: Dirty bit: Page has been modified recently
- L: \( L=1 \Rightarrow 4\text{MB page} \)
Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid $\Rightarrow$ Page in memory, PTE points at physical page
  - Not Valid $\Rightarrow$ 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, run another process from ready queue
    - Suspended process sits on wait queue
Demand Paging Cost Model

- Demand paging is caching $\Rightarrow$ Can compute average access time! (Effective Access Time)
  - EAT = Hit Time + Miss Rate $\times$ Miss Penalty

- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8ms
  - Suppose $p =$ Probability of miss, $1 - p =$ Probably of hit
  - Then, we can compute EAT as follows
    \[
    EAT = 200\text{ns} + p \times 8\text{ms} = 200\text{ns} + p \times 8,000,000\text{ns}
    \]

- If one access out of 1,000 causes page fault, then EAT = 8.2 $\mu$s:
  - This is a slowdown by a factor of 40!

- What if want slowdown by less than 10%?
  - $200\text{ns} \times 1.1 > EAT \Rightarrow p < 2.5 \times 10^{-6}$
  - This is approximately 1 page fault in 400,000 accesses!
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!

• Capacity misses
  • Not enough memory; must somehow increase available memory size
  • Can we do this?
    • One option is increasing amount of DRAM (not quick fix!)
    • Another option is adjusting percentage of memory allocated to process if multiple processes are in memory

• Conflict misses
  • Technically, conflict misses don't exist in virtual memory, since it is “fully-associative” cache

• Policy misses
  • Caused when pages were in memory, but kicked out prematurely because of replacement policy
  • How to fix?
    • Better replacement policy
Page Replacement Policies

- **Random**
  - Pick random page for every replacement
  - Typical solution for TLB’s, simple hardware
  - Pretty unpredictable – makes it hard to provide any real-time guarantees

- **First In, First Out (FIFO)**
  - Throw out oldest page, fair – let every page live in memory for same amount of time
  - Bad – could throw out heavily used pages instead of infrequently used

- **Minimum (MIN)**
  - Replace page that won’t be used for the longest time
  - Great, but how can we really know future?
  - Makes good comparison case, however

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

- Suppose we have 3 p-pages, 4 v-pages, and following reference stream:
  - A B C A B D A D B C B

<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></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>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>C</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 we'll need A again right away
Example: MIN

- Consider following reference stream: A B C A B D A D B C B

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</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></td>
<td>C</td>
<td></td>
<td>D</td>
<td></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!
When Will LRU Perform Badly?

- Consider following reference stream: A B C D A B C D A B C D

<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>D</td>
<td></td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>B</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>D</td>
<td>C</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Every reference is a page fault!
When will LRU Perform Badly? (cont.)

- Consider the following: A B C D A B C D A B C D
- 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>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory Size and Page Fault Rate

• 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 policies don’t have this obvious property!
Bélády's Anomaly

After adding memory:

- With FIFO, contents can be completely different
- With LRU or MIN, contents of memory with $X$ pages are a subset of contents with $X+1$ Page
LRU Implementation

- 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 when each page is used to change its position in list
  - Many instructions for each memory access
Clock Algorithm: Practical LRU Implementation

- Arrange physical pages in circle with single clock hand
- HW sets “use” bit of PTE on each reference
  - If use bit isn’t set, it means page hasn’t been referenced in long time
- HW could set use bit in TLB
  - OS must copy this back to PTE when TLB entry 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
- Replace new page with selected candidate ⇒ replace old page, not the oldest page
- Will this algorithm always find replacement page, or does it loop forever?
  - If all use bits set, clock hand will eventually loop around ⇒ FIFO
Clock Algorithm: Not Recently Used

- What if hand is moving slowly? Is it a good sign or a 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?
Clock Algorithms: Details

• Which bits of PTE entry are useful?
  • **Use**: Set by HW when page is referenced, cleared by clock algorithm
  • **Dirty**: set by HW when page is modified, cleared by clock algorithm when page is written back to disk
  • **Valid**: OK for program to reference this page
  • **Read-only**: OK for program to read page, but not modify

• Do we really need hardware-supported “dirty” 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 “dirty” bit, and marks page as read-write
    • Whenever page comes back in from disk, mark read-only
Clock Algorithms: Details (cont.)

- Do we really need a hardware-supported “use” bit?
  - No, we can emulate it like what we did with “dirty” bit
    - Mark all pages as invalid, even if in memory
    - On read to invalid page, trap to OS
    - OS sets software “use” bit, and marks page read-only
    - On write, trap to OS (either invalid or read-only)
    - Set software “use” and “dirty” bits, mark page read-write
    - When clock hand passes by, reset “use” and “dirty” bits and mark as invalid again

- Do we need reverse mapping (i.e. physical page → virtual page, core map)?
  - Yes. clock algorithm runs through physical pages
  - Multiple virtual pages could be mapped to the same physical page
  - We can’t push physical page out to disk without invalidating all PTEs

- Clock algorithm is just approximation of LRU, can we do a better?
  - Answer: Nth chance algorithm
Nth Chance Algorithm

- Give each page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1 → clear use AND clear counter (used in last sweep)
    - 0 → increment counter; if count = N, select as replacement candidate
  - Clock hand must sweep by N times without page being used before page is replaced

- How do we pick N?
  - Large N: Better approximation to LRU, if N ~ 1K, very good approximation
  - 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 dirty page, so give dirty pages extra chance!
    - Clean pages, use N = 1
    - Dirty pages, use N = 2 (and write back to disk when N = 1)
Implementation Notes

• Clock and N\textsuperscript{th} chance algorithms can run \textit{synchronously}
  • In page fault handler, run algorithm to find next page to evict
  • Might require writing changes back to disk first

• Or \textit{asynchronously}
  • Run algorithms in the background
  • Maintain pool of candidates for eviction
  • Write dirty pages back to disk
  • On page fault, check if requested page is in pool!
  • If not, evict a page from the pool
Allocation of Physical Pages

• How do we allocate memory among different processes?
  • Does every process get same fraction of memory?
  • Should we completely swap some processes out of memory?

• Each process needs minimum number of pages
  • All processes loaded into memory should make progress

• Possible replacement scopes
  • Global replacement – to make space for one process's page, replacement is selected from all processes’ pages
  • Local replacement – to make space for one process's page, replacement is selected from process’ set of allocated pages
Fixed/Priority Allocation

- **Equal allocation** (fixed scheme)
  - Every process gets same amount of memory
  - Example: 100 physical pages, 5 processes → Each process gets 20 pages

- **Proportional allocation** (fixed scheme)
  - Allocate according to size of process
  - Computation proceeds as follows:
    - \( s_i = \text{size of process } p_i \) and \( S = \text{sum of } s_i \)'s for all \( p_i \)'s
    - \( m = \text{total number of physical pages} \)
    - \( a_i = \text{allocation for } p_i = (s_i \times m) / S \)

- **Priority allocation**
  - Proportional scheme using priorities rather than size
  - Possible behavior: If process \( p_i \) generates page fault, select for replacement page from process with lower priority number

- Perhaps we should use an *adaptive* scheme instead?
  - What if some application just needs more memory?
Page-Fault Rate: Capacity Misses

- Can we reduce capacity misses by dynamically changing # of pages per application?

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses page
  - If actual rate too high, process gains page

- Question: What if we just don’t have enough memory?
Thrashing

- If process does not have “enough” pages, page-fault rate is very high which leads to
  - Low CPU utilization
  - OS spends most of its time swapping pages to disk
- Thrashing ≡ process is busy swapping pages in and out disk
- Questions:
  - How do we detect thrashing?
  - What is best response to thrashing?
Locality in Memory-Reference Pattern

- Working set defines minimum number of pages needed for process to behave well
- Not enough memory for working set $\Rightarrow$ Thrashing
  - Better to swap out process?
Working-Set Model

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

- Recall that compulsory misses are misses that occur first time that 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 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 track working set of applications
  - When swapping process back in, swap in working set
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, then can replace

• Nth-chance clock algorithm: Another approximate LRU
  • Give pages multiple passes of clock hand before replacing

• Thrashing: 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
Questions?
Acknowledgment

• Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny