CS162
Operating Systems and Systems Programming
Lecture 17

Memory 4: Demand Paging Policies

March 16th, 2023
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall 61C: Average Memory Access Time

- Used to compute access time probabilistically:
  \[
  \text{AMAT} = \text{Hit Rate}_{L1} \times \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Time}_{L1}
  \]
  
  \[
  \text{Hit Rate}_{L1} + \text{Miss Rate}_{L1} = 1
  \]
  
  \[
  \text{Hit Time}_{L1} = \text{Time to get value from L1 cache.}
  \]
  
  \[
  \text{Miss Time}_{L1} = \text{Hit Time}_{L1} + \text{Miss Penalty}_{L1}
  \]
  
  Miss Penalty_{L1} = AVG Time to get value from lower level (DRAM)

  So, \( \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1} \)

- What about more levels of hierarchy?
  \[
  \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}
  \]
  
  Miss Penalty_{L1} = AVG time to get value from lower level (L2)
  
  = Hit Time_{L2} + Miss Rate_{L2} \times Miss Penalty_{L2}

  Miss Penalty_{L2} = Average Time to fetch from below L2 (DRAM)

  \[
  \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2})
  \]

- And so on ... (can do this recursively for more levels!)
Recall: Caching Applied to Address Translation

- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on same page (accesses are sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…

- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds
What Actually Happens on a TLB Miss?

- **Hardware traversed page tables (x86, many others):**
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - If PTE valid, hardware fills TLB and processor never knows
    - If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards

- **Software traversed Page tables (like MIPS):**
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - If PTE valid, fills TLB and returns from fault
    - If PTE marked as invalid, internally calls Page Fault handler

- **Most chip sets provide hardware traversal**
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - shared segments
    - user-level portions of an operating system
Transparent Exceptions: Page fault

• How to transparently restart faulting instructions?
  – (Consider load or store that gets Page fault)
  – Could we just skip faulting instruction?
    » No: need to perform load or store after reconnecting physical page!

• Hardware must help out by saving:
  – Faulting instruction and partial state
    » Need to know which instruction caused fault
    » Is single PC sufficient to identify faulting position?????
  – Processor State: sufficient to restart user thread
    » Save/restore registers, stack, etc

• What if an instruction has side-effects?

Diagram:

User

Page Faults

OS

Faulting Inst 1

Restart Inst 1

Faulting Inst 2

Restart Inst 2

Copy on Write/Update PTE

Fetch page/sleep on queue Later: Update PTE/restart

User

OS

Page Faults
Consider weird things that can happen

• What if an instruction has side effects?
  – Options:
    » Unwind side-effects (easy to restart)
    » Finish off side-effects (messy!)
  – Example 1: `mov (sp)+, 10`
    » What if page fault occurs when writing to stack pointer?
    » Did sp get incremented before or after the page fault?
  – Example 2: `strcpy (r1), (r2)`
    » Source and destination overlap: can’t unwind in principle!
    » IBM S/370 and VAX solution: execute twice – once read-only

• What about “RISC” processors?
  – For instance delayed branches?
    » Example: `bne somewhere`  
      `ld r1, (sp)`
    » Restart after page fault: need two PCs, PC and nPC!
  – Delayed exceptions:
    » Example: `div r1, r2, r3`  
      `ld r1, (sp)`
    » What if takes many cycles to discover divide by zero, but load has already caused page fault?
Precise Exceptions

• Precise $\Rightarrow$ 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, ...
    – x86 takes this position

• Imprecise $\Rightarrow$ 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
Current Intel x86 (Skylake, Cascade Lake)

- (L1) Instruction TLB
- (L2) Unified TLB
- (L1) Data TLB

- Reorder Buffer (Precise Exceptions)
Current Example: Memory Hierarchy

• Caches (all 64 B line size)
  – L1 I-Cache: 32 KiB/core, 8-way set assoc.
  – L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  – L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  – L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency

• TLB
  – L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    » 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  – L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    » 32 entries; 4-way set associative, 2 MiB / 4 MiB page translations:
      » 4 entries; 4-way associative, 1G page translations:
  – L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    » 16 entries; 4-way set associative, 1 GiB page translations:
What happens on a Context Switch?

• Need to do something, since TLBs map virtual addresses to physical addresses
  – Address Space just changed, so TLB entries no longer valid!

• Options?
  – Invalidate (“Flush”) TLB: simple but might be expensive
    » What if switching frequently between processes?
  – Include ProcessID in TLB
    » This is an architectural solution: needs hardware

• What if translation tables change?
  – For example, to move page from memory to disk or vice versa…
  – Must invalidate TLB entry!
    » Otherwise, might think that page is still in memory!
  – Called “TLB Consistency”

• Aside: with Virtually-Indexed, Virtually-Tagged cache, need to flush cache!
  – Everyone has their own version of the address “0” and can’t distinguish them
  – This is one advantage of Virtually-Indexed, Physically-Tagged caches..
Putting Everything Together: Address Translation
Putting Everything Together: TLB

Virtual Address: Virtual P1 index Virtual P2 index Offset

PageTablePtr

Virtual P1 index

Virtual P2 index

Page Table (1st level)

Physical Page #

Offset

Page Table (2nd level)

TLB:

Physical Address: Physical Page # Offset

Physical Memory:

Virtual Address:

Offset
Putting Everything Together: Cache
Page Fault Handling

- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Privilege Level Violation, Access violation, or does not exist
  - Causes an Fault / Trap
    - Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically terminate the process
- Other Page Faults engage operating system to fix the situation and retry the instruction
  - Allocate an additional stack page, or
  - Make the page accessible – (Copy on Write),
  - Bring page in from secondary storage to memory – demand paging
- Fundamental inversion of the hardware / software boundary
  - Need to execute software to allow hardware to proceed!
Page Fault → Demand Paging

Process

virtual address

MMU

physical address

page fault

update PT entry

load page from disk

scheduler

retry

exception

Page Fault Handler

Operating System

instruction

Page Fault
Administrivia

• Still grading exam!
  – Hopefully have it by early next week
• Project 2 in full swing
  – Stay on top of this one. Don’t wait until last moment to get pieces together
• Homework 4 also in full swing
• Make sure to fill out survey! Due today!
  – We really want to hear how you think we are doing
  – Also, will get a chance to suggest topics for the special topics lecture
Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all 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
### Management & Access to the Memory Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Access</th>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Accessed in Hardware</td>
<td>0.3</td>
<td>100Bs</td>
</tr>
<tr>
<td>L1 Cache</td>
<td></td>
<td>1</td>
<td>10kBs</td>
</tr>
<tr>
<td>L2 Cache (shared)</td>
<td></td>
<td>3</td>
<td>100kB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td></td>
<td>10-30</td>
<td>MBs</td>
</tr>
<tr>
<td>Main Memory (DRAM)</td>
<td>Accessed in Software - OS</td>
<td>100</td>
<td>100GBs</td>
</tr>
<tr>
<td>Secondary Storage (SSD)</td>
<td></td>
<td>100,000 (0.1 ms)</td>
<td>100GBs</td>
</tr>
<tr>
<td>Secondary Storage (Disk)</td>
<td></td>
<td>10,000,000 (10 ms)</td>
<td>TBs</td>
</tr>
</tbody>
</table>

**Secondary Storage**
- Disk
- SSD

**Managed in Hardware**
- TLB
- Registers

**Managed in Software - OS**
- PT
Demand Paging as Caching, …

- What “block size”? - 1 page (e.g., 4 KB)
- What “organization” i.e., direct-mapped, set-associ., fully-associative?
  - Fully associative since arbitrary virtual → physical mapping
- How do we locate a page?
  - First check TLB, then page-table traversal
- What is page replacement policy? (i.e., LRU, Random…)
  - This requires more explanation… (kinda LRU)
- What happens on a miss?
  - Go to lower level to fill miss (i.e., disk)
- What happens on a write? (write-through/write back?)
  - Definitely write-back – need dirty bit!
Illusion of Infinite Memory

- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    » More programs fit into memory, allowing more concurrency
- Principle: **Transparent Level of Indirection** (page table)
  - Supports flexible placement of physical data
    » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    » Performance issue, not correctness issue
Review: What is in a PTE?

- What is in a 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:
  - 2-level page table (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>P</th>
<th>W</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>
<td></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 (PTE only): page has been modified recently
- PS: Page Size: PS=1⇒4MB page (directory only).
  Bottom 22 bits of virtual address serve as offset
Demand Paging Mechanisms

• PTE makes demand paging implementatable
  – 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
Origins of Paging

- Disks provide most of the storage.
- Keep most of the address space on disk.
- Actively swap pages to/from.
- Keep memory full of the frequently accesses pages.
- Relatively small memory, for many processes.
- Many clients on dumb terminals running different programs.
Very Different Situation Today

Powerful system
Huge memory
Huge disk
Single user
A Picture on one machine

- Memory stays about 75% used, 25% for dynamics
- A lot of it is shared 1.9 GB
Many Uses of Virtual Memory and “Demand Paging” …

• Extend the stack
  – Allocate a page and zero it

• Extend the heap (sbrk of old, today mmap)

• Process Fork
  – Create a copy of the page table
  – Entries refer to parent pages – NO-WRITE
  – Shared read-only pages remain shared
  – Copy page on write

• Exec
  – Only bring in parts of the binary in active use
  – Do this on demand

• MMAP to explicitly share region (or to access a file as RAM)
Classic: Loading an executable into memory

- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization: `crt0` (C runtime init)
Create Virtual Address Space of the Process

- Utilized pages in the VAS are backed by a page block on disk
  - Called the backing store or swap file
  - Typically in an optimized block store, but can think of it like a file
Create Virtual Address Space of the Process

- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - swapped into and out of memory as needed
- For every process
Create Virtual Address Space of the Process

- User Page table maps entire VAS
  - Resident pages to the frame in memory they occupy
  - The portion of it that the HW needs to access must be resident in memory
Provide Backing Store for VAS

- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk
  - Many ways to do this, but might use remaining bits of PTE when P=0
What Data Structure Maps Non-Resident Pages to Disk?

- **FindBlock**(PID, page#) → disk_block
  - Some OSs utilize spare space in PTE for paged blocks
  - Like the PT, but purely software

- Where to store it?
  - In memory – can be compact representation if swap storage is contiguous on disk
  - Could use hash table (like Inverted PT)

- Usually want backing store for resident pages too

- May map code segment directly to on-disk image
  - Saves a copy of code to swap file

- May share code segment with multiple instances of the program
Provide Backing Store for VAS

- disk (huge, TB)
- VAS 1
  - kernel
  - stack
  - heap
  - data
  - code
- VAS 2
  - kernel
  - stack
  - heap
  - data
  - code
- PT 1
  - memory
  - user page frames
  - user pagetable
  - kernel code & data
- PT 2
  - user page frames
  - user pagetable
  - kernel code & data
On page Fault …
On page Fault … find & start load
On page Fault ... update PTE

memory
user page frames
user pagetable
kernel code & data

active process & PT

disk (huge, TB)

stack
heap
data
code

stack
heap
data
code

VAS 1
PT 1

kernel
stack
heap
data
code

VAS 2
PT 2

kernel
stack
heap
data
code
Eventually reschedule faulting thread
Summary: Steps in Handling a Page Fault

1. Trap
2. Page is on backing store
3. Restart instruction
4. Bring in missing page
5. Reset page table
6. Physical memory

Operating system
Reference
Load M
Some questions we need to answer!

• 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
    » Schedule dirty pages to be written back on disk
    » Zero (clean) pages which haven’t been accessed in a while
  – 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
Working Set Model

- As a program executes it transitions through a sequence of “working sets” consisting of varying sized subsets of the address space.
Cache Behavior under WS model

- Amortized by fraction of time the Working Set is active
- Transitions from one WS to the next
- Capacity, Conflict, Compulsory misses
- Applicable to memory caches and pages. Others?
Another model of Locality: Zipf

$P\text{ access}(\text{rank}) = \frac{1}{\text{rank}}$

- Likelihood of accessing item of rank $r$ is $\alpha \frac{1}{r^a}$
- Although rare to access items below the top few, there are so many that it yields a “heavy tailed” distribution
- Substantial value from even a tiny cache
- Substantial misses from even a very large cache
Demand Paging Cost Model

• Since Demand Paging like caching, can compute average access time! (“Effective Access Time”)  
  – EAT = Hit Rate x Hit Time + Miss Rate x Miss Time  
  – EAT = Hit Time + Miss Rate x 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:
    \[ EAT = 200\text{ns} + p \times 8\text{ ms} \]
    \[ = 200\text{ns} + p \times 8,000,000\text{ns} \]

• If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  – This is a slowdown by a factor of 40!

• What if want slowdown by less than 10%?
  – EAT < 200ns x 1.1 \( \Rightarrow \) p < 2.5 \( \times \) 10\(^{-6}\)
  – This is about 1 page fault in 400,000!
What Factors Lead to Misses in Page Cache?

- **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 available memory 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

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

• MIN (Minimum):
  – Replace page that won’t be used for the longest time
  – Great (provably optimal), but can’t really know future…
  – But past is a good predictor of the future …
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 page used so that can change position in list…
  – Many instructions for each hardware access

• In practice, people approximate LRU (more later)
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>C</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></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 / 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</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></td>
<td>C</td>
<td></td>
<td></td>
<td>D</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!
Is LRU guaranteed to perform well?

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

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

- Every reference is a page fault!

- MIN Does much better:

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

<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></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td>E</td>
<td></td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></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
Approximating LRU: Clock Algorithm

- **Clock Algorithm**: Arrange physical pages in circle with single clock hand
  - Approximate LRU \((\text{approximation to approximation to MIN})\)
  - 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 PTE 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
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 \rightarrow clear use and also clear counter (used in last sweep)
    - 0 \rightarrow 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 \sim 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:

<table>
<thead>
<tr>
<th>PTE:</th>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>PS</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></td>
<td></td>
<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>
</tr>
</tbody>
</table>

- 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 → 0 and “modified” bits → 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 → 1 in software to indicate that page has been “used”.
      Further:
      1) If read, mark page as read-only, W → 0 (will catch future writes)
      2) If write (and write allowed), set “modified” bit → 1, mark page as writable (W → 1)
    » When clock hand passes, reset emulated “use” bit → 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 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 (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 widely separated 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
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