CS162
Operating Systems and
Systems Programming
Lecture 15

Memory 3: Caching and TLBs (Con’t), Demand Paging

March 11th, 2021
Profs. Natacha Crooks and Anthony D. Joseph
http://cs162.eecs.Berkeley.edu
Recall: The two-level page table

- Tree of Page Tables
  - “Magic” 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e. CR3)
- Valid bits on Page Table Entries
  - Don’t need every 2\textsuperscript{nd}-level table
  - Even when exist, 2\textsuperscript{nd}-level tables can reside on disk if not in use
Recall: Caching Applied to Address Translation

- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses 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
Recall: A Summary on Sources of Cache Misses

• **Compulsory** (cold start or process migration, first reference): first access to a block
  – “Cold” fact of life: not a whole lot you can do about it
  – Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant

• **Capacity:**
  – Cache cannot contain all blocks access by the program
  – Solution: increase cache size

• **Conflict** (collision):
  – Multiple memory locations mapped to the same cache location
  – Solution 1: increase cache size
  – Solution 2: increase associativity

• **Coherence** (Invalidation): other process (e.g., I/O) updates memory
Recall: What TLB Organization Makes Sense?

- Needs to be really fast
  - Critical path of memory access
    - In simplest view: before the cache
    - Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high! (PT traversal)
  - Cost of Conflict (Miss Time) is high
  - Hit Time – dictated by clock cycle
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - First page of code, data, stack may map to same entry
    - Need 3-way associativity at least?
  - What if use high order bits as index?
    - TLB mostly unused for small programs
Physically-Indexed vs Virtually-Indexed Caches

- Physically-Indexed Caches
  - Address handed to cache after translation
  - Page Table holds physical addresses
  - Benefits:
    » Every piece of data has single place in cache
    » Cache can stay unchanged on context switch
  - Challenges:
    » TLB is in critical path of lookup!
  - Pretty Common today (e.g., x86 processors)

- Virtually-Indexed Caches
  - Address handed to cache before translation
  - Page Table holds virtual addresses (one option)
  - Benefits:
    » TLB not in critical path of lookup, so can be faster
  - Challenges:
    » Same data could be mapped in multiple places of cache
    » May need to flush cache on context switch

- We will stick with Physically Addressed Caches for now!
Reducing translation time for physically-indexed caches

- As described, TLB lookup is in serial with cache lookup
  - Consequently, speed of TLB can impact speed of access to cache

- Machines with TLBs go one step further: overlap TLB lookup with cache access
  - Works because offset available early
  - Offset in virtual address exactly covers the “cache index” and “byte select”
  - Thus can select the cached byte(s) in parallel to perform address translation

```
virtual address: Virtual Page # Offset
physical address: tag / page # index byte
```
Overlapping TLB & Cache Access

• Here is how this might work with a 4K cache:

- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252

- Another option: Virtual Caches would make this faster
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses
Putting Everything Together: Address Translation

Virtual Address:

- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1<sup>st</sup> level)

Page Table (2<sup>nd</sup> level)

PageTablePtr

Physical Address:

- Physical Page #
- Offset

Physical Memory:
Putting Everything Together: TLB

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:
Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Cache:
- tag
- index
- byte
Administrivia

- Midterm 2: Coming up on next Thursday 3/18 5-6:30PM (one weeks from today)
  - Topics: up and including Lecture 16: Scheduling, Deadlock, Address Translation, Virtual Memory, Caching, TLBs, Demand Paging
  - Lecture 16 will be released early
  - Will REQUIRE you to have your zoom proctoring setup working again
    » You must have screen sharing (entire desktop not just the browser), audio unmuted, and your camera working
    » Make sure to get your setup debugged and ready!
- Review Session: 3/16 (5-7pm)
  - Details TBA
Recall: Page Fault

• The Virtual-to-Physical Translation fails
  – PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  – Causes a Fault / Trap
    » Not an interrupt because synchronous to instruction execution
  – May occur on instruction fetch or data access
  – Protection violations typically terminate the instruction

• 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
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
Page Fault $\Rightarrow$ Demand Paging

Process

- Instruction
- Exception
- Retry

Operating System

Page Fault Handler

- Scheduler
- Load page from disk
- Update PT entry

virtual address

MMU

physical address

PT

page#

frame#

offset

scheduler
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”

| Page Frame Number (Physical Page Number) | Free (OS) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | P | W | U | PWT | PCD | A | D |
|-----------------------------------------|-----------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31-12                                   | 11-9      | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |   |   |   | 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 implementable
  – 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
The View 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 anExecutable 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 mapped to the frame in memory they occupy
- The portion of page table 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 the frame in memory they occupy
- For all other pages, OS must record where to find them on disk
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)

stack
heap
data
code

stack
heap
data
code

VAS 1

stack
heap
data
code

VAS 2

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 …

disk (huge, TB)

stack
heap
data
code

heap
data
code

data
code

stack
heap
data
code

kernel
stack
heap
data
code

user
page frames
user page table
kernel
code & data

active process & PT

VAS 1
PT 1

VAS 2
PT 2

memory
On Page Fault ... Find & Start Load

disk (huge, TB)

memory

kernel
data

user

page frames

user

pagetable

kernel
code & data

active process & PT

VAS 1

PT 1

code

data

heap

stack

VAS 2

PT 2

code

data

heap

stack

active process & PT

user

page frames

user

pagetable

kernel
code & data

3/11/21
On page Fault ... schedule other P or T
On Page Fault … Update PTE
Eventually reschedule faulting thread
Summary: Steps in Handling a Page Fault

1. reference
2. trap
3. page is on backing store
4. bring in missing page
5. free frame
6. reset page table
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 \( \alpha \frac{1}{r^\alpha} \)
- 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")
  - \( \text{EAT} = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time} \)
  - \( \text{EAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \)

- 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 \( \text{EAT} \) 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 \( \text{EAT} = 8.2 \ \mu\text{s} \):
  - This is a slowdown by a factor of 40!

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

• “Translation Lookaside Buffer” (TLB)
  – Small number of PTEs and optional process IDs (< 512)
  – Often Fully Associative (Since conflict misses expensive)
  – On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  – On change in page table, TLB entries must be invalidated

• Demand Paging: Treating the DRAM as a cache on disk
  – Page table tracks which pages are in memory
  – Any attempt to access a page that is not in memory generates a page fault, which causes OS to bring missing page into memory

• 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