CSE 120
Principles of Operating Systems
Spring 2023
Lecture 14: TLB, Swapping
Yiying Zhang
Multiple-level page tables
Multi-level page tables

- 3 Advantages?
  - L1, L2, L3 tables do not have to be consecutive
  - They do not have to be allocated before use!
  - They can be swapped out to disk!

*The power of an extra level of indirection!*

- Problems?
Efficient Translations

• Our original page table scheme already increased the cost of doing memory lookups
  ♦ Two lookups into the page table, another to fetch the data
  ♦ One lookup and one data access for original flat page table
• Now 4-level page tables require five DRAM accesses for one memory operation!
  ♦ Four lookups into the page tables, a fifth to fetch the data
• Solution: *reference locality!*
  ♦ In a short period of time, a process is likely accessing only a few pages
  ♦ Store part of the page table that is “hot” in a fast hardware unit
Translation Look-aside Buffer (TLB)

- Translation Look-aside Buffers
  - Translate VPNs into PFNs
- TLBs implemented in hardware
  - TLB hit is very fast <=1 CPU cycle
  - New entries can be inserted anywhere in the TLB
  - All entries looked up in parallel
  - TLB can’t be made very big, typically 64 – 4096 entries
Translation Look-aside Buffer (TLB)

- Virtual address
  - VPN
  - offset

- TLB
  - VPN
  - PFN
  - ...
  - ...
  - ...

- Real page table

- Physical address
  - PFN
  - offset

TCP/IP - aside Buffer (TLB)
Miss handling: Hardware-controlled TLB

• On a TLB hit, MMU checks the valid bit
  ♦ If valid, perform address translation
  ♦ If invalid (e.g. page not in memory), MMU generates a page fault
    » OS performs fault handling
    » Restart the faulting instruction

• On a TLB miss
  ♦ MMU parses page table and loads PTE into TLB
    » Needs to replace if TLB is full
    » Page table layout is fixed
  ♦ Same as hit …

• Most modern machines use hardware-controlled TLB
Miss handling:
Software-controlled TLB

• On a TLB hit, MMU checks the valid bit
  ♦ If valid, perform address translation
  ♦ If invalid (e.g. page not in memory), MMU generates a page fault
  » OS performs page fault handling
  » Restart the faulting instruction

• On a TLB miss, HW raises exception, traps to the OS
  ♦ OS parses page table and loads PTE into TLB
  » Needs to replace if TLB is full
  » Page table layout can be flexible
  ♦ Same as in a hit…
More TLB Issues

• When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  ♦ Which TLB entry should be replaced?
    » Random
    » LRU

• What happens when changing a page table entry (e.g. because of swapping, change read/write permission)?
  ♦ Change the entry in memory
  ♦ flush (eg. invalidate) the TLB entry
    » INGLPG on x86
What happens to TLB in a process context switch?

- During a process context switch, cached translations can not be used by the next process
  - Invalidate all entries during a context switch
    » Lots of TLB misses afterwards
More on coherence issues

• No hardware maintains coherence between DRAM and TLBs:
  ♦ OS needs to flush related TLBs whenever changing a page table entry in memory

• On multiprocessors, when you modify a page table entry, you need to do “TLB shoot-down” to flush all related TLB entries at all the cores
Summary so far

- Virtual memory addresses: a level of indirection to decouple static time (compiler) from run time (OS)
- Paging: avoiding external fragmentation, great flexibility
- Single-level page tables are too big
- Multi-level page tables reduce the space overhead (leveraging indirection) but increases the performance overhead
- TLB improves paging performance (leveraging locality)
- But TLB shootdown is costly (esp. on many cores)
Early-day memory system

- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically
    - dynamic memory relocation: base&bound
  - Single segment per process
    - dynamic memory relocation: segmentation, paging
  - Entire address space needs to fit in mem
The last drawback

• So far we’ve separated the process’s view of memory from the OS’s view using a mapping mechanism
  ♦ Each sees a different organization
  ♦ Allows OS to shuffle processes around
  ♦ Simplifies memory sharing
  ♦ *What is the essence of the mechanism that enables this?*

• But, a user process had to be completely loaded into memory before it could run

→ Wasteful since a process only needs a small amount of its total memory at any time (*reference locality!*).
Virtual Memory

- Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory.

- Key idea: Virtual address space translated to:
  - Physical memory (small, fast) or
  - Disk/SSD (backing store), large but slow
  - Or nothing (if it is not yet in use)

- Deep thinking – what made above possible?
Virtual Memory Implementation

• How to save memory?

1. Allocate physical memory pages (from storage or initially allocated) as late as possible, i.e., “on demand”
2. Move less used pages to storage, i.e., paging with swapping
On-Demand Allocation

- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

- How to detect this?
  - In software?
**On-Demand Allocation**

- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

- Hardware/software cooperate to make things work
  - Include a valid bit (present bit) in each PTE
  - Any page not in main memory right now has the valid bit cleared in its PTE
  - If valid bit isn’t set, a reference to the page results in a trap by the paging hardware, called *page fault*
  - What needs to happen when page fault occurs?
What happens at virtual memory allocation time and access time?

- What happens at virtual memory allocation time?
  - If demand paging (on-demand allocation) is used, the OS allocates a virtual address and establishes a PTE with no PFN and with invalid bit set.

- What happens when the virtual address is first accessed?
  - The OS should allocate physical memory for it.
  - How to capture the first write to a virtual page?
    - e.g. want to trap into page fault handler
      - Use valid bit
  - In page fault handler handler, check if the virtual page is allocated (and access permitted)
    - If not, segmentation fault
    - Otherwise, allocate physical page and update PTE.
**Virtual Memory**

- **Objective:**
  - To produce the illusion of memory as big as necessary
  - Without suffering a huge slowdown of execution

- **What makes this possible?**
  - *Principle of locality*
    - Knuth’s estimation of 90% of the time in 10% of the code
    - There is also significant locality in data references
What happens when main memory is not big enough?

- Processes running on a machine have collectively used more memory than what the physical main memory has.

- Some memory pages need to be put to storage (swap out)

- What happens when a swapped out page is accessed?
  - Need to swap in the page
  - How to detect that a swapped out page is accessed?
Page Fault Handling with Swapping

1. MMU (TLB)
2. Page fault
3.1 (if phys mem full) swap out a victim page to disk
3.2 Update PTE of victim pg, flush TLB
4.1 Swap in access page from disk
4.2 Update PTE of access pg
5. Resume faulting intr
Page fault handling (cont)

• On a page fault
  ♦ Find an unused phy page. If no unused, find a used phy. page (policy on which used one to pick in future lecture)
  ♦ If the phy. page is used
    » If it has been modified (how to know?), write it to disk
    » Invalidate its current PTE and TLB entry (how?)
  ♦ Load the new page from disk
  ♦ Update the faulting PTE and its TLB entry (how?)
  ♦ Restart the faulting instruction

• Supporting data structure that an OS uses
  ♦ For speed: A list of unused physical pages (more later)
  ♦ Data structure to map a phy. page to its pid(s) and virtual address(es)
  ♦ Data structure to remember where a swapped out page is on disk
Address Translation Redux

• We started this topic with the high-level problem of translating virtual addresses into physical addresses

• We’ve covered all of the pieces
  ♦ Virtual and physical addresses
  ♦ Virtual pages and physical page frames
  ♦ Multi-level page tables and page table entries (PTEs)
  ♦ TLBs
  ♦ On-demand allocation and swapping

• Now let’s put it together, bottom to top
The Common Case

- The compiler compiles source code into binaries (containing memory instructions)
- OS loads the executable (a.out) into memory and starts its execution

- Process is executing on the CPU, and it issues a read to an address
  - What kind of address is it, virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning the physical page frame and protection bits for this address
  3. TLB validates that the protection bits allows reads (in this example)
  4. MMU combines the PFN and offset into a physical address
  5. MMU then reads from that physical address, returns value to CPU

- Note: The above execution is all done by the hardware
**TLB Misses**

- At this point, two other things can happen
  1. TLB does not have this virtual address
  2. Mapping in TLB, but memory access violates protection bits or the invalid bit is set
- We’ll consider each in turn
Reloading the TLB

• If the TLB does not have mapping, two possibilities:
  MMU loads PTE from page table in memory (a page table walk)
  » Hardware managed TLB, OS not involved in this step

• When TLB has PTE, it restarts translation
  ♦ Common case is that the PTE refers to a valid page in memory
    » Hardware just reads PTE from the page table and loads it into TLB
  ♦ Uncommon case is that TLB faults again on PTE because of PTE protection/valid bits (e.g., page is invalid (not in memory))
    » Becomes a page fault…
Page Faults

• PTE can indicate the type of a page fault
  ♦ Read/write/execute – operation not permitted on page
  ♦ Invalid – page not in physical memory
• Traps to the OS (software takes over)
  ♦ R/W/E – OS usually will send fault back up to user process, or use for other purposes (e.g., copy on write, more next time)
  ♦ Invalid
    » Page not in physical memory because this is the first access
      ▪ OS allocates physical frame and sets up the PTE (and flush TLB)
    » Page not in physical memory because it has been swapped out
      ▪ Finds an empty frame in physical memory (if none, need to swap out something first), reads the page from disk, sets up the PTE to point to the new physical frame (and flush TLB)
Next time

- Swapping, memory allocation, memory sharing
Backup Slides
Memory Hierarchy Revisited

What does this imply about L1 addresses?

Where do we hope requests get satisfied?
Physical (Address) Caches

- Memory hierarchy so far: **physical caches**
  - Indexed and tagged by PAs
    - Physically Indexed (PI)
    - Physically Tagged (PT)
  - Translate to PA to VA at the outset
  + Cached inter-process communication works
    - Single copy indexed by PA
    - Slow: adds at least one cycle to $t_{hit}$
Virtual Caches (VI/VT)

- **Alternative: virtual caches**
  - Indexed and tagged by VAs (VI and VT)
  - Translate to PAs only to access L2
  + Fast: avoids translation latency in common case
  - Problem: VAs from *different processes* are distinct physical locations (with different values) (call homonyms)

- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags

- Does inter-process communication work?
  - **Synonyms**: multiple VAs map to same PA
    » Can’t allow same PA in the cache twice
    » Also a problem for DMA I/O
  - Can be handled, but very complicated
Memory Hierarchy Re-Revisited

What does this imply about L1 addresses?

Any speed benefits?
Any drawbacks?

CPU

L1

TLB

L2

Main Memory
Parallel TLB/Cache Access (VI/PT)

- Compromise: access TLB in parallel
  - In small caches, index of VA and PA the same
    - $VI == PI$
  - Use the VA to index the cache
  - Tagged by PAs
  - Cache access and address translation in parallel
    + No context-switching/aliasing problems
    + Fast: no additional $t_{hit}$ cycles
  - Common organization in processors today