CSE 120
Principles of Operating Systems

Fall 2021

Lecture 11: TLB, Swapping
Yiying Zhang
Announcements

- Midterm graded

- Tomorrow’s discussion section will go over PR2 and some of HW3 (if have time)
Page Table Entries (PTEs)

- Page table entries control mapping
  - The **Modify** bit says whether or not the page has been written
    » It is set when a write to the page occurs
  - The **Reference** bit says whether the page has been accessed
    » It is set when a read or write to the page occurs
  - The **Valid** bit says whether or not the PTE can be used
    » It is checked each time the virtual address is used
  - The **Protection** bits say what operations are allowed on page
    » Read, write, execute
  - The **page frame number** (PFN) determines physical page
x86 Page Table Entry

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>G</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>Cd</th>
<th>Wt</th>
<th>O</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>12 Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Valid (present)
- Read/write
- Owner (user/kernel)
- Write-through
- Cache disabled
- Accessed (referenced)
- Dirty
- PDE maps 4MB
- Global
Paging implementation – how does it really work?

- Where to store page table?
- How to use MMU?
  - Even small page tables are too large to load into MMU
  - Page tables kept in mem and MMU only has their base addresses
- What happens at context switches?
How can we reduce page table space overhead?

- Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)

How can we be flexible?

"All computer science problems can be solved with an extra level of indirection."

two-level page tables
Two-Level Page Tables

Virtual address

Dir | Page | offset
---|------|------

Directory table addr

PFN | ...
---|------

Page table

PFN

Physical address
Multiple-level page tables

- Page-map L4
- Page-dir-ptr
- Page-directory
- Page-table
- Page offset

Diagram:

- Page-map L4 table
- Page-dir-ptr entry
- Page-dir entry
- Page-table entry
- Physical address
- Main memory
- Physical page frame number
- Page offset

Page-map L4 base addr (CR3)
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?
[lec10] 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
  - Fully associative cache => least conflict misses
  - 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
- Optional (useful) bits
  - ASIDs -- Address-space identifiers (process tags)
Translation Look-aside Buffer (TLB)

Virtual address

VPN
offset

PFN
PFN
PFN
PFN

Real page table

PFN
offset

Physical address

VPN
PFN
...

VPN
PFN
...

VPN
PFN
...

TLB

Hit

Miss
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 …
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…
Hardware vs. software controlled

- **Hardware approach**
  - Efficient – TLB misses handled by hardware
  - OS intervention is required only in case of page fault
  - Page structure prescribed by MMU hardware -- rigid

- **Software approach**
  - Less efficient -- TLB misses are handled by software
  - MMU hardware very simple, permitting larger, faster TLB
  - OS designer has complete flexibility in choice of MM data structure
Deep thinking

- Without TLB, how MMU finds PTE is fixed
- With TLB, it can be flexible, e.g. software-controlled is possible
- What enables this?
- TLB is an extra level of indirection!
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
  - Tag each entry with an ASID
    » Add a HW register that contains the process id of the current executing process
    » TLB hits if an entry’s process id matches that register
Cache vs. TLB

• Similarities:
  ♦ Both cache a part of the physical memory

• Differences:
  ♦ Associatively
    » TLB is usually fully associative
    » Cache can be direct mapped
  ♦ Coherence
    » No hardware provided coherence between TLB and main memory
    » Software needs to flush TLB entries for coherence
    » Cache: hardware-provided (via snooping bus) coherence across multiple cores and main memory
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 shutdown is costly (esp. on many cores)
Remaining of This Lecture

We’ll cover more virtual memory topics:

• Optimizations
  ♦ Managing page tables (space)
  ♦ Efficient translations (TLBs) (time)
  ♦ Demand paged virtual memory (swapping) (space)

• Memory allocation

• Kernel address space (if have time)
[lec9] Sharing main memory

• 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
  » More need for swapping
  » Need to swap whole, very expensive!
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 either
  - Physical memory (small, fast) or
  - Disk/SSD (backing store), large but slow

- Deep thinking – what made above possible?

- Objective:
  - To produce the illusion of memory as big as necessary
Virtual Memory

• “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
Virtual Memory Implementation

• Virtual memory is typically implemented via demand paging

• demand paging:
  ♦ Load memory pages (from storage or initially allocated) “on demand”
  ♦ paging with swapping, e.g., physical pages are swapped in and out of memory
Demand Paging
(paging with swapping)

• 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?
Demand Paging  
(paging with swapping)

• 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 (more later today) 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
    - Else allocate physical page and update PTE
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?
Next time...

- Chapter 22