Virtual Memory 2

Prof. Hakim Weatherspoon
CS 3410, Spring 2015
Computer Science
Cornell University

P & H Chapter 5.7
Announcements

Lab3: Available today, and due by next Wednesday

HW2: Do up to Problem5 this week. Do it now.
    Do Problem9, coding a hashtable in C, now.
## Announcements

Next five weeks

- Week 10 (Apr 7): **Lab3** (calling convention) release
- Week 11 (Apr 14): **Proj3** (caches) release, Lab3 due Wed
- Week 12 (Apr 21): **Lab4** (virtual memory) release, due in-class, Proj3 due Fri, **HW2 due Sat**
- Week 13 (Apr 28): **Proj4** (multi-core/parallelism) release, Lab4 due in-class, **Prelim2**
- Week 14 (May 5): **Proj3 Tournament**, Proj4 design doc due

Final Project for class

- Week 15 (May 12): Proj4 due Wed
Goals for Today

Virtual Memory
• Address Translation
  • Pages, page tables, and memory mgmt unit
• Paging
• Performance
  • How slow is it
  • Making virtual memory fast
  • Translation lookaside buffer (TLB)
• Virtual Memory Meets Caching

• Role of Operating System
  • Context switches, working set, shared memory
Virtual Memory Summary

All problems in computer science can be solved by another level of indirection.

Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

Virtual memory is implemented via a “Map”, a PageTable, that maps a vaddr (a virtual address) to a paddr (physical address):
\[ paddr = PageTable[vaddr] \]

A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits. But, overhead due to PageTable is significant.
Next Goal

How do we reduce the size (overhead) of the PageTable?
Next Goal

How do we reduce the size (overhead) of the PageTable?

A: Another level of indirection!!
A: can make different views of same data with different permissions
Q: Benefits?
A1: Don’t need 4MB contiguous physical memory
A2: Don’t need to allocate every PageTable, only those containing valid PTEs
Q: Drawbacks?
A: Longer lookups.
<table>
<thead>
<tr>
<th>Multi-level Page Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume most of PageTable is empty</td>
</tr>
<tr>
<td>How to translate addresses? Multi-level PageTable</td>
</tr>
</tbody>
</table>

Q: Benefits?
A: Don’t need 4MB contiguous physical memory
A: Don’t need to allocate every PageTable, only those containing valid PTEs

Q: Drawbacks
A: Performance: Longer lookups

Q: Benefits?
A1: Don’t need 4MB contiguous physical memory
A2: Don’t need to allocate every PageTable, only those containing valid PTEs
Q: Drawbacks?
A: Longer lookups.
Takeaway

All problems in computer science can be solved by another level of indirection.

Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

Virtual memory is implemented via a “Map”, a PageTable, that maps a vaddr (a virtual address) to a paddr (physical address):
\[ paddr = \text{PageTable}[vaddr] \]

A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits.

But, overhead due to PageTable is significant.

Another level of indirection, two levels of PageTables, can significantly reduce the overhead due to PageTables.
Next Goal

Can we run process larger than physical memory?
Paging

Can we run process larger than physical memory?
  • The “virtual” in “virtual memory”

View memory as a “cache” for secondary storage
  • Swap memory pages out to disk when not in use
  • Page them back in when needed

Assumes Temporal/Spatial Locality
  • Pages used recently most likely to be used again soon
Cool Trick #4: Paging/Swapping

Need more bits:
Dirty, RecentlyUsed, ...

A: can make code read-only, executable; make data read-write but not executable; etc.
Performance
Performance

Virtual Memory Summary

PageTable for each process:

- Page table tradeoffs
  - Single-level (e.g. 4MB contiguous in physical memory)
  - or multi-level (e.g. less mem overhead due to page table),
  - ...
- every load/store translated to physical addresses
- page table miss = \textit{page fault}
  - load the swapped-out page and retry instruction,
    or kill program if the page really doesn’t exist,
    or tell the program it made a mistake
Page Table Review

x86 Example: 2 level page tables, assume...
32 bit vaddr, 32 bit paddr
4k PDir, 4k PTables, 4k Pages

Q: How many bits for a physical page number?
A: 20

Q: What is stored in each PageTableEntry?
A: ppn, valid/dirty/r/w/x/...

Q: What is stored in each PageDirEntry?
A: ppn, valid/?/...

Q: How many entries in a PageDirectory?
A: 1024 four-byte PDEs

Q: How many entries in each PageTable?
A: 1024 four-byte PTEs
**Page Table Example**

x86 Example: 2 level page tables, assume...

- 32 bit vaddr, 32 bit paddr
- 4k PDir, 4k PTables, 4k Pages
- PTBR = 0x10005000 (physical)

Write to virtual address, vaddr= 0x7192a44c...

Q: Byte offset in page? 0x44c PT Index? 0x12a PD Index? 0x1c6

\[ vaddr = vaddr \ll 22 \]

1. PageDir is at 0x10005000, so...
   - Fetch PDE from physical address 0x1005000+(4*PDI)
     - suppose we get \( \{0x12345, v=1, \ldots\} \)

2. PageTable is at 0x12345000, so...
   - Fetch PTE from physical address 0x12345000+(4*PTI)
     - suppose we get \( \{0x14817, v=1, d=0, r=1, w=1, x=0, \ldots\} \)

3. Page is at 0x14817000, so...
   - Write data to physical address? 0x1481744c
   - Also: update PTE with d=1

byte offset = 0x44c
PTI = 0x12a
PDI = (0x719 >> 2) = 0x1c6
Performance
Virtual Memory Summary
PageTable for each process:

- Page
  - Single-level (e.g. 4MB contiguous in physical memory)
  - or multi-level (e.g. less mem overhead due to page table),
  - ...
- every load/store translated to physical addresses
- page table miss: load a swapped-out page and retry instruction, or kill program

Performance?

- How many memory references required to access memory?
  a) 1  b) 2
  c) 3  d) 4
  e) 5

4x slower!
Performance

Virtual Memory Summary

PageTable for each process:

- Page
  - Single-level (e.g. 4MB contiguous in physical memory)
  - or multi-level (e.g. less mem overhead due to page table),
  - ...
- every load/store translated to physical addresses
- page table miss: load a swapped-out page and retry instruction, or kill program

Performance?

- terrible: memory is already slow
  translation makes it slower

Solution?

- A cache, of course

4x slower!
Next Goal

How do we speed up address translation?
Making Virtual Memory Fast
The Translation Lookaside Buffer (TLB)
Translation Lookaside Buffer (TLB)

Hardware Translation Lookaside Buffer (TLB)

A small, very fast cache of recent address mappings

• TLB hit: avoids PageTable lookup
• TLB miss: do PageTable lookup, cache result for later
TLB miss in hardware usually, but not always
PageTable stuff in software
Takeaway

The TLB is a fast cache for address translations. A TLB hit is fast, miss is slow.
Next Goal

How do we keep TLB, PageTable, and Cache consistent?
TLB Coherency

TLB Coherency: What can go wrong?

A: PageTable or PageDir contents change
   • swapping/paging activity, new shared pages, ...

A: Page Table Base Register changes
   • context switch between processes
Translation Lookaside Buffers (TLBs)

When PTE changes, PDE changes, PTBR changes....

Full Transparency: TLB coherency in hardware

- Flush TLB whenever PTBR register changes [easy – why?]
- Invalidate entries whenever PTE or PDE changes [hard – why?]

TLB coherency in software

If TLB has a no-write policy...

- OS invalidates entry after OS modifies page tables
- OS flushes TLB whenever OS does context switch

process ID could just be the PTBR for the process
TLB Parameters

TLB parameters (typical)
- very small (64 – 256 entries), so very fast
- fully associative, or at least set associative
- tiny block size: why?

Intel Nehalem TLB (example)
- 128-entry L1 Instruction TLB, 4-way LRU
- 64-entry L1 Data TLB, 4-way LRU
- 512-entry L2 Unified TLB, 4-way LRU

Tiny block size because there is no spacial locality
Takeaway

The TLB is a fast cache for address translations.
A TLB hit is fast, miss is slow.

TLB Coherency –
• in HW – flush TLB when PTBR changes (context switch) and invalidate entry when PTE or PDE changes (may need processID).
• In SW–OS invalidates TLB entry after change page tables or OS flushes TLB whenever OS does context switch.
Next Goal

Virtual Memory meets Caching
Virtually vs. physically addressed caches
Virtually vs. physically tagged caches
Virtual Memory meets Caching
Virtually vs. physically addressed caches
Virtually vs. physically tagged caches
Recall TLB in the Memory Hierarchy

TLB is passing a physical address so we can load from memory.

What if the data is in the cache?
A: have to flush entire cache on context switch
A:
Doing synonym updates requires significant hardware – essentially an associative lookup on the physical address tags to see if you have multiple hits
A1: Physically-addressed: nothing; Virtually-addressed: need to flush cache
A2: Physically-addressed: nothing; Virtually-addressed: problems
Indexing vs. Tagging

Physically-Addressed Cache
- slow: requires TLB (and maybe PageTable) lookup first

Virtually-Addressed Cache
- fast: start TLB lookup before cache lookup finishes
- PageTable changes (paging, context switch, etc.)
  - need to purge stale cache lines (how?)
- Synonyms (two virtual mappings for one physical page)
  - could end up in cache twice (very bad!)

Virtually-Indexed, Physically Tagged Cache
- ~fast: TLB lookup in parallel with cache lookup
- PageTable changes → no problem: phys. tag mismatch
- Synonyms → search and evict lines with same phys. tag
Typical L1: On-chip virtually addressed, physically tagged
Typical L2: On-chip physically addressed
Typical L3: On-chip ...
Design Decisions of Caches/TLBs/VM

Caches, Virtual Memory, & TLBs
Where can block be placed?
• Direct, n-way, fully associative

What block is replaced on miss?
• LRU, Random, LFU, ...

How are writes handled?
• No-write (w/ or w/o automatic invalidation)
• Write-back (fast, block at time)
• Write-through (simple, reason about consistency)

Where?
Caches: direct/n-way/fa
VM: fa, but with a table of contents to eliminate searches
TLB: fa
Replacement?
varied
Writes?
Caches: usually write-back, or maybe write-through, or maybe no-write w/ invalidation
VM: write-back
TLB: usually no-write
### Summary of Caches/TLBs/VM

Caches, Virtual Memory, & TLBs

**Where can block be placed?**
- Caches: direct/n-way/fully associative (fa)
- VM: fa, but with a table of contents to eliminate searches
- TLB: fa

**What block is replaced on miss?**
- varied

**How are writes handled?**
- Caches: usually write-back, or maybe write-through, or maybe no-write w/ invalidation
- VM: write-back
- TLB: usually no-write

---

**Where?**
- Caches: direct/n-way/fa
- VM: fa, but with a table of contents to eliminate searches
- TLB: fa

**Replacement?**
- varied

**Writes?**
- Caches: usually write-back, or maybe write-through, or maybe no-write w/ invalidation
- VM: write-back
- TLB: usually no-write
<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>Paged Memory</th>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size (blocks)</strong></td>
<td>1/4k to 4k</td>
<td>16k to 1M</td>
<td>64 to 4k</td>
</tr>
<tr>
<td><strong>Size (kB)</strong></td>
<td>16 to 64</td>
<td>1M to 4G</td>
<td>2 to 16</td>
</tr>
<tr>
<td><strong>Block size (B)</strong></td>
<td>16-64</td>
<td>4k to 64k</td>
<td>4-32</td>
</tr>
<tr>
<td><strong>Miss rates</strong></td>
<td>2%-5%</td>
<td>10^-4 to 10^-5%</td>
<td>0.01% to 2%</td>
</tr>
<tr>
<td><strong>Miss penalty</strong></td>
<td>10-25</td>
<td>10M-100M</td>
<td>100-1000</td>
</tr>
</tbody>
</table>
Role of the Operating System
Context switches, working set, shared memory
Next Goal

How many programs do you run at once?

How does the Operating System (OS) help?
Role of the Operating System

The operating systems (OS) manages and multiplexes memory between processes. It...

- Enables processes to (explicitly) increase memory:
  - sbrk and (implicitly) decrease memory
- Enables sharing of physical memory:
  - multiplexing memory via context switching, sharing memory, and paging
- Enables and limits the number of processes that can run simultaneously
sbrk (more memory)

Suppose Firefox needs a new page of memory

1) Invoke the Operating System
   
   void *sbrk(int nbytes);

2) OS finds a free page of physical memory
   
   • clear the page (fill with zeros)
   
   • add a new entry to Firefox’s Page Table
Context Switch (sharing CPU)
Suppose Firefox is idle, but Skype wants to run
(1) Firefox invokes the Operating System
    int sleep(int nseconds);
(2) OS saves Firefox’s registers, load Skype’s
    • (more on this later)
(3) OS changes the CPU’s Page Table Base Register
    • Cop0:ContextRegister / CR3:PDBR
(4) OS returns to Skype
Shared Memory

Suppose Firefox and Skype want to share data

(1) OS finds a free page of physical memory

- clear the page (fill with zeros)
- add a new entry to Firefox’s PageTable
- add a new entry to Skype’s PageTable
  - can be same or different vaddr
  - can be same or different page permissions
Multiplexing

Suppose Skype needs a new page of memory, but Firefox is hogging it all

(1) Invoke the Operating System
    void *sbrk(int nbytes);

(2) OS can’t find a free page of physical memory
    • Pick a page from Firefox instead (or other process)

(3) If page table entry has dirty bit set...
    • Copy the page contents to disk

(4) Mark Firefox’s page table entry as “on disk”
    • Firefox will fault if it tries to access the page

(5) Give the newly freed physical page to Skype
    • clear the page (fill with zeros)
    • add a new entry to Skype’s PageTable
Takeaway

The OS assists with the Virtual Memory abstraction

- sbrk
- Context switches
- Shared memory
- Multiplexing memory
How can the OS optimize the use of physical memory?

What does the OS need to beware of?
Paging Assumption 1
OS multiplexes physical memory among processes

- assumption # 1:
  processes use only a few pages at a time

- working set = set of process's recently actively pages
Thrashing (excessive paging)

Q: What if working set is too large?

Case 1: Single process using too many pages

Case 2: Too many processes
Thrashing

Thrashing b/c working set of process (or processes) greater than physical memory available

- Firefox steals page from Skype
- Skype steals page from Firefox

- I/O (disk activity) at 100% utilization
  - But no useful work is getting done

Ideal: Size of disk, speed of memory (or cache)
Non-ideal: Speed of disk
Paging Assumption 2
OS multiplexes physical memory among processes

- assumption # 2: recent accesses predict future accesses
- working set usually changes slowly over time

![Graph showing working set over time](image)
More Thrashing

Q: What if working set changes rapidly or unpredictably?

A: Thrashing b/c recent accesses don’t predict future accesses
Preventing Thrashing

How to prevent thrashing?

- User: Don’t run too many apps
- Process: efficient and predictable mem usage
- OS: Don’t over-commit memory, memory-aware scheduling policies, etc.
Takeaway

The OS assists with the Virtual Memory abstraction

- sbrk
- Context switches
- Shared memory
- Multiplexing memory
- Working set
- Thrashing
Virtual Memory Summary

All problems in computer science can be solved by another level of indirection.

Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

Virtual memory is implemented via a “Map”, a PageTable, that maps a vaddr (a virtual address) to a paddr (physical address):

\[ \text{paddr} = \text{PageTable}[\text{vaddr}] \]

A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits. But, overhead due to PageTable is significant.
Summary of Caches/TLBs/VM

Caches, Virtual Memory, & TLBs

Where can block be placed?
  • Caches: direct/n-way/fully associative (fa)
  • VM: fa, but with a table of contents to eliminate searches
  • TLB: fa

What block is replaced on miss?
  • varied

How are writes handled?
  • Caches: usually write-back, or maybe write-through, or maybe no-write w/ invalidation
  • VM: write-back
  • TLB: usually no-write

Where?
Caches: direct/n-way/fa
VM: fa, but with a table of contents to eliminate searches
TLB: fa
Replacement?
varied
Writes?
Caches: usually write-back, or maybe write-through, or maybe no-write w/ invalidation
VM: write-back
TLB: usually no-write
<table>
<thead>
<tr>
<th>Summary of Cache Design Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Size (blocks)</strong></td>
</tr>
<tr>
<td>1/4k to 4k</td>
</tr>
<tr>
<td><strong>Size (kB)</strong></td>
</tr>
<tr>
<td><strong>Block size (B)</strong></td>
</tr>
<tr>
<td><strong>Miss rates</strong></td>
</tr>
<tr>
<td><strong>Miss penalty</strong></td>
</tr>
</tbody>
</table>