Virtual Memory

CS 3410
Computer System Organization & Programming

[K. Bala, A. Bracy, E. Sirer, and H. Weatherspoon]
Click any letter let me know you’re here today.

Instead of a DJ Clicker Question today, please take a minute to think about the question:

“What can I do to make Cornell a more welcoming and affirming campus?”
Picture Memory as...?

Byte Array:

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffff</td>
<td>xaa</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>x00</td>
</tr>
<tr>
<td>0x00000000</td>
<td>x00</td>
</tr>
<tr>
<td>x00</td>
<td>xef</td>
</tr>
<tr>
<td>xef</td>
<td>xcd</td>
</tr>
<tr>
<td>xcd</td>
<td>xab</td>
</tr>
<tr>
<td>xab</td>
<td>xff</td>
</tr>
<tr>
<td>xff</td>
<td>x00</td>
</tr>
</tbody>
</table>

Segments:

- system reserved
- stack
- heap
- data
- text
- system reserved

Page Array:

New! page n

- 0xffffffff00
- 0xffffffffd000
- 0xffff0000
- 0x00000000
- 0x00000400
- 0x00000200
- 0x00000100
- 0x00000000

Each segment uses some # of pages
A Little More About Pages

Suppose each page = 4KB

Anything in page 2 has address:
\[ \text{0x00002xxx} \]

Lower 12 bits specify which byte you are in the page:
\[ \text{0x00002200} = 0010 0000 0000 = \text{byte 512} \]

upper bits = page number
lower bits = page offset

Sound familiar?
Data Granularity

**ISA:** instruction specific: LB, LH, LW (MIPS)

**Registers:** 32 bits (MIPS)

**Caches:** cache line/block

*Address bits divided into:*
- **tag:** sanity check for address match
- **index:** which entry in the cache
- **offset:** which byte in the line

**Memory:** page

*Address bits divided into:*
- **page number:** which page in memory
- **index:** which byte in the page
These assumptions are embedded in the executable! If they are wrong, things will break! Recompile? Relink?

Program’s View of Memory

32-bit machine:
0x00000000 – 0xffffffff to play with (modulo system reserved)

2 Interesting/Dubious Assumptions:
The machine I’m running on has 4GB of DRAM. I am the only one using this DRAM.

64-bits: 16 EB ???
Indirection* to the Rescue!

Virtual Memory: a Solution for All Problems

• Each process has its own virtual address space
  ▪ Program/CPU can access any address from 0...2^N
  ▪ A process is a program being executed
  ▪ Programmer can code as if they own all of memory

• On-the-fly at runtime, for each memory access
  ▪ all accesses are indirect through a virtual address
  ▪ translate fake virtual address to a real physical address
  ▪ redirect load/store to the physical address

*google David Wheeler, Butler Lampson, Leslie Lamport, and Steve Bellovin
Virtual vs. Physical Address Spaces

- Not contiguous
- Page vs. Address?
Advantages of Virtual Memory

Easy relocation
• Loader puts code anywhere in physical memory
• Virtual mappings to give illusion of correct layout

Higher memory utilization
• Provide illusion of contiguous memory
• Use all physical memory, even physical address 0x0

Easy sharing
• Different mappings for different processes / cores

And more to come...
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?

- Address Translation
- Overhead
- Paging
- Performance
- Virtual Memory & Caches
Address Translator: MMU

- Processes use virtual addresses
- DRAM uses physical addresses

Memory Management Unit (MMU)
- HW structure
- Translates virtual → physical address on the fly
Address Translation: in Page Table

**OS-Managed** Mapping of Virtual $\rightarrow$ Physical Pages

```c
int page_table[2^{20}] = { 0, 5, 4, 1, ... };
...
ppn = page_table[vpn];
```

Remember:
any address `0x00001234` is `x234` bytes into Page C
both virtual & physical

VP 1 $\rightarrow$ PP 5

Assuming each page = 4KB, lower 12 bits $\rightarrow$ offset
Page Table Basics

1 Page Table per process

Lives in Memory, i.e., in a page (or more...)

Location stored in Page Table Base Register

Assuming each page = 4KB
### Simple Address Translation

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1010 1111 0000 1111 0000 1111 0000</td>
<td>0000 1111 0000</td>
</tr>
</tbody>
</table>

- **Lookup in Page Table**

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0101 1100 0011 0000 0000 0000 1111 0000</td>
<td>0000 1111 0000</td>
</tr>
</tbody>
</table>

*Assuming each page = 4KB*
Simple Page Table Translation

Assuming each page = 4KB
General Address Translation

What if the page size is not 4KB?
⇒ Page offset is no longer 12 bits

Clicker Question:
Page size is 16KB ⇒ how many bits is page offset?
(a) 12  (b) 13  (c) 14  (d) 15  (e) 16

What if Main Memory is not 4GB?
⇒ Physical page number is no longer 20 bits

Clicker Question:
Page size 4KB, Main Memory 512 MB ⇒ how many bits is PPN?
(a) 14  (b) 15  (c) 16  (d) 17  (e) 18
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?
  • Address Translation
  • Overhead
  • Paging
  • Performance
  • Virtual Memory & Caches
Page Table Overhead

• How large is a Page Table?

• Virtual address space (for each process):
  ▪ Given: total virtual memory: $2^{32}$ bytes = 4GB
  ▪ Given: page size: $2^{12}$ bytes = 4KB
  ▪ # entries in PageTable?
  ▪ size of PageTable?
    ▪ *This is one, big contiguous array, by the way!*

• Physical address space:
  ▪ Given: total physical memory: $2^{29}$ bytes = 512MB
  ▪ overhead for 10 processes?
Page Table Overhead

• How large is PageTable?

• Virtual address space (for each process):
  ▪ Given: total virtual memory: $2^{32}$ bytes = 4GB
  ▪ Given: page size: $2^{12}$ bytes = 4KB
  ▪ # entries in PageTable? $2^{20} = 1$ million entries
  ▪ size of PageTable? PTE size = 4 bytes

• Physical address space:
  PageTable size = $4 \times 2^{20} = 4$MB
  ▪ total physical memory: $2^{29}$ bytes = 512MB
  ▪ overhead for 10 processes?

  $10 \times 4$MB = 40 MB of overhead!

  • $40$ MB /$512$ MB = 7.8% overhead, space due to PageTable
But Wait… There’s more!

• Page Table Entry won’t be just an integer
• Meta-Data
  ▪ Valid Bits
    • What PPN means “not mapped”? No such number…
    • At first: not all virtual pages will be in physical memory
    • Later: might not have enough physical memory to map all virtual pages
  ▪ Page Permissions
    • R/W/X permission bits for each PTE
    • Code: read-only, executable
    • Data: writeable, not executable
### Less Simple Page Table

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>X</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0xC20A3</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0x10045</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0xC20A3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0x10044</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Aliasing:** mapping several virtual addresses → same physical page

- Process tries to access a page without proper permissions
  - **Segmentation Fault**

**Examples:**
- Write to read-only? → process killed
- Execute non-executable? → process killed
Now how big is this Page Table?

```c
struct pte_t page_table[2^{20}]
```

Each PTE = 8 bytes

How many pages in memory will the page table take up?

Clicker Question:

(a) 4 million (2^{22}) pages
(b) 2048 (2^{11}) pages
(c) 1024 (2^{10}) pages
(d) 4 billion (2^{32}) pages
(e) 4K (2^{12}) pages

Assuming each page = 4KB
Multi-Level Page Table

* Indirection to the Rescue, AGAIN!

Also referred to as Level 1 and Level 2 Page Tables
Multi-Level Page Table

Doesn’t this take up more memory than before?

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

Drawbacks
- Performance: Longer lookups
Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

• Address Translation
• Overhead
• Paging
• Performance
• Virtual Memory & Caches
Paging

What if process requirements > physical memory?

*Virtual starts earning its name*

Memory acts as a cache for secondary storage (disk)
- Swap memory pages out to disk when not in use
- Page them back in when needed

Courtesy of Temporal & Spatial Locality (again!)
- Pages used recently mostly likely to be used again

More Meta-Data:
- Dirty Bit, Recently Used, *etc.*
- OS may access this meta-data to choose a victim
Example: accessing address beginning with \texttt{0x00003} (PageTable[3]) results in a Page Fault which will page the data in from disk sector 200
Page Fault

Valid bit in Page Table = 0 → means page is not in memory

OS takes over:
- Choose a physical page to replace
  - “Working set”: refined LRU, tracks page usage
- If dirty, write to disk
- Read missing page from disk
  - Takes so long (~10ms), OS schedules another task

Performance-wise page faults are really bad!
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?
• Address Translation
• Overhead
• Paging
• Performance
• Virtual Memory & Caches
Watch Your Performance Tank!

For every instruction:

- MMU translates address (virtual \(\rightarrow\) physical)
  - Uses PTBR to find Page Table in memory
  - Looks up entry for that virtual page
- Fetch the instruction using physical address
  - Access Memory Hierarchy (I$ \(\rightarrow\) L2 \(\rightarrow\) Memory)

- Repeat at Memory stage for load/store insns
  - Translate address
  - Now you perform the load/store
Translation Looksaside Buffer (TLB)

- Small, fast cache
- Holds VPN → PPN translations
- Exploits temporal locality in pagetable
- TLB Hit: huge performance savings
- TLB Miss: invoke TLB miss handler
  - *Put translation in TLB for later*
TLB Parameters

**Typical**
- very small (64 – 256 entries) $\rightarrow$ *very fast*
- fully associative, or at least set associative
- tiny block size: why?

**Example: Intel Nehalem TLB**
- 128-entry L1 Instruction TLB, 4-way LRU
- 64-entry L1 Data TLB, 4-way LRU
- 512-entry L2 Unified TLB, 4-way LRU
**TLB to the Rescue!**

For every instruction:

• Translate the address (virtual $\rightarrow$ physical)
  - CPU checks TLB
  - That failing, walk the Page Table
    • Use PTBR to find Page Table in memory
    • Look up entry for that virtual page
    • Cache the result in the TLB

• Fetch the instruction using physical address
  - Access Memory Hierarchy (I$ \rightarrow$ L2 $\rightarrow$ Memory)

• Repeat at Memory stage for load/store insns
  - CPU checks TLB, translate if necessary
  - **Now** perform load/store
The presence of a TLB is part of the ISA.

(A) True
(B) False
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?
  • Address Translation
  • Overhead
  • Paging
  • Performance
  • Virtual Memory & Caches
    • Caches use physical addresses
    • Prevents sharing except when intended
    • Works beautifully!
Translation in Action

Clicker Question:
In this Diagram there cannot be a DRAM miss here.
(A) True (B) False

Next Topic:
Exceptional Control Flow
Takeaways

Need a map to translate a “fake” virtual address (from process) to a “real” physical Address (in memory).

The map is a **Page Table**: \( \text{ppn} = \text{PageTable}[\text{vpn}] \)

A page is constant size block of virtual memory. Often ~4KB to reduce the number of entries in a PageTable.

Page Table can enforce Read/Write/Execute permissions on a per page basis. Can allocate memory on a per page basis. Also need a valid bit, and a few others.

Space overhead due to Page Table is significant. Solution: another level of indirection! **Two-level of Page Table** significantly reduces overhead.

Time overhead due to Address Translations also significant. Solution: caching! **Translation Lookaside Buffer (TLB)** acts as a cache for the Page Table and significantly improves performance.