Virtual Memory

Hakim Weatherspoon
CS 3410
Computer Science
Cornell University

[Weatherspoon, Bala, Bracy, McKee, and Sirer]
Where are we now and where are we going?

- How many programs do you run at once?
  - a) 1
  - b) 2
  - c) 3-5
  - d) 6-10
  - e) 11+
Big Picture: Multiple Processes

• Can we execute *more than one* program at a time with our current RISC-V processor?

  • a) Yes, no problem at all
  • b) No, because memory addresses will conflict
  • c) Yes, caches can avoid memory address conflicts
  • d) Yes, our modified Harvard architecture avoids memory address conflicts
  • e) Yes, because we have multiple processors (multiple cores)
Big Picture: Multiple Processes

How to run multiple processes?

• *Time-multiplex* a single CPU core (*multi-tasking*)
  • Web browser, skype, office, … all must co-exist

• Many cores per processor (*multi-core*) or many processors (*multi-processor*)
  • Multiple programs run *simultaneously*
Processor & Memory

- CPU address/data bus...
- ... routed through caches
- ... to main memory
  - Simple, fast, but...
Multiple Processes

• Q: What happens when another program is executed concurrently on another processor?

• A: The addresses will conflict
  - Even though, CPUs may take turns using memory bus
Multiple Processes

• Q: Can we relocate second program?
Solution? Multiple processes-processors

- Q: Can we relocate second program?
- A: Yes, but…
  - What if they don’t fit?
  - What if not contiguous?
  - Need to recompile/relink?
  - …
Big Picture: (Virtual) Memory

Give each process an **illusion** that it has exclusive access to entire main memory.
But In Reality…

Process 1

Process 2

Physical Memory
How do we create the illusion?

Process 1

Process 2

Physical Memory
How do we create the illusion?

Process 1

Process 2

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

– David Wheeler

Physical Memory
How do we create the illusion?

Process 1

Virtual address

Process 2

Map virtual address to physical address

Memory management unit (MMU) takes care of the mapping

Virtual Memory (just a concept; does not exist physically)

Physical Memory
How do we create the illusion?

Virtual Memory
(just a concept; does not exist physically)

Physical Memory

Process 1

Virtual address

Process 2

Virtual address

Process 1 wants to access data C

Process 1 thinks it is stored at addr 1

So CPU generates addr 1

This addr is intercepted by MMU

MMU knows this is a virtual addr

MMU looks at the mapping

Virtual addr 1 -> Physical addr 9

Data at Physical addr 9 is sent to CPU

And that data is indeed C!!!
How do we create the illusion?

Process 1

Process 2

Map virtual address to physical address

Memory management unit (MMU) takes care of the mapping

Virtual Memory

Disk

Physical Memory
Big Picture: (Virtual) Memory

- From a process’s perspective –
  - Process only sees the virtual memory
    - Contiguous memory
Big Picture: (Virtual) Memory

• From a process’s perspective –

• Process only sees the virtual memory
  ✓ Contiguous memory
  ✓ No need to recompile - only mappings need to be updated
Big Picture: (Virtual) Memory

- From a process’s perspective –

  - Process only sees the virtual memory
    - Contiguous memory
    - No need to recompile - only mappings need to be updated
Big Picture: (Virtual) Memory

- From a process’s perspective –
  - Process only sees the virtual memory
    - Contiguous memory
    - No need to recompile - only mappings need to be updated
    - When run out of memory, MMU maps data on disk in a transparent manner
Next Goal

• How does Virtual Memory work?

• i.e. How do we create the “map” that maps a virtual address generated by the CPU to a physical address used by main memory?
Next Goal (after spring break!)

• How does Virtual Memory work?

• i.e. How do we create the “map” that maps a virtual address generated by the CPU to a physical address used by main memory?
Have a great Spring Break!!!
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?

• Address Translation
• Overhead
• Paging
• Performance
Virtual Memory: Recap

Process 1
Store B at 2
Load data at 2

Virtual address

Process 2
Store H at 2

Hidden from the process

Virtual Memory
{just a set of numbers (addresses); does not exist physically}

Physical Memory

Virtual address

Physical address
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>0x80000000</td>
<td>...</td>
</tr>
<tr>
<td>x00</td>
<td>x00</td>
</tr>
<tr>
<td>x00</td>
<td>xef</td>
</tr>
<tr>
<td>x00</td>
<td>xcd</td>
</tr>
<tr>
<td>x00</td>
<td>xab</td>
</tr>
<tr>
<td>x00</td>
<td>xff</td>
</tr>
<tr>
<td>0x00000000</td>
<td>x00</td>
</tr>
</tbody>
</table>

Segments:

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

Page Array:

- page 0
- page 1
- page 2
- ...
A Little More About Pages

Memory size = depends on system say 4GB

Page size = 4KB (by default)

Then, # of pages = $2^{20}$

Any data in a page # 2 has address of the form: $0x00002xxx$

Lower 12 bits specify which byte you are in the page:

$0x00002200 = 0010 0000 0000$

= byte 512

upper bits = page number (PPN)
lower bits = page offset
Page Table: Data structure to store mapping

1 Page Table per process
Lives in Memory, i.e. in a page (or more...)
Location stored in Page Table Base Register

Part of program state (like PC)

PTBR 0x00008000

Assuming each page = 4KB
Address Translator: MMU

- Programs use virtual addresses
- Actual memory uses physical addresses

Memory Management Unit (MMU)
- HW structure
- Translates virtual → physical address on the fly
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) 15  (b) 16  (c) 17  (d) 18  (e) 19
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-1 (N=number of bits in address register)
  • 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
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 programs / cores

And more to come…
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):
  \[ paddr = \text{PageTable}[\text{vaddr}] \]
Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
- Overhead
- Paging
- Performance
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
    $\rightarrow$ PageTable size = $4 \times 2^{20} = 4$MB
• **Physical address space:**
  • total physical memory: $2^{29}$ bytes = 512MB
  • overhead for 10 processes?

10 x 4MB = 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>1</td>
<td>1</td>
<td>0</td>
<td>0xC20A3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0xC20A3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0xC20A3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0x4123B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0x10044</td>
</tr>
</tbody>
</table>

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

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

**Example:**
- Write to read-only? → process killed
Now how big is this Page Table?

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
Now how big is this Page Table?

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
Wait, how big is this Page Table?

\[ \text{page\_table}[2^{20}] = 8 \times 2^{20} = 2^{23} \text{ bytes} \]
(Page Table = 8 MB in size)

How many pages in memory will the page table take up? \(2^{23} / 2^{12} = 2^{11} \) 2K pages!

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
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.
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!!
Single-Level Page Table

20 bits

12 bits

vaddr

31 12 11 0

Where is my physical page?

PTBR

Page Table

PTEntry

PPN

Total size = $2^{20} \times 4$ bytes

= 4MB
Multi-Level Page Table

![Diagram of multi-level page table with annotations:]

- **vaddr**: 31 bits
- **PTBR**: 10 bits
- **10 bits**: 22-21
- **12 bits**: 12-0

*Indirection to the Rescue, AGAIN!*

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

Where is my page translation?

Page Directory

PTBR

PDEntry

Page Table

PTEntry

PPN

vaddr

10 bits 10 bits 12 bits

31 22 21 12 11 0

Assuming each entry is 4 bytes, what is the size of Page Directory?

A: 2KB  B: 2MB  C: 4KB  D: 4MB

Also referred to as Level 1 and Level 2 Page Tables

* Indirection to the Rescue, AGAIN!

Also referred to as Level 1 and Level 2 Page Tables

46
**Multi-Level Page Table**

![Diagram of Multi-Level Page Table]

- **vaddr**: 31 bits
- **PTBR**: 10 bits
- **12 bits vaddr**: 12 bits
- **PDEntry**: Page Directory Entry
- **PTEntry**: Page Table Entry
- **PPN**: Physical Page Number

**Where is my physical page?**

```
31  22  21  12  11  0
```

**Assuming each entry is 4 bytes, what is the total size of all Page tables?**

- A: 2KB
- B: 2MB
- C: 4KB
- D: 4MB

*Indirection to the Rescue, AGAIN!*
Multi-Level Page Table

- **vaddr**: 10 bits, 12 bits
- **PTBR**: 10 bits
- **PDEntry**: 31, 22, 21, 12, 11, 0
- **Page Directory**
  - Size = $2^{10} \times 4$ bytes = 4KB
- **Page Table**
  - Size = $2^{10} \times 2^{10} \times 4$ bytes = 4MB
  - \# page tables * \# entries per page table

**Where is my physical page?**

**Where is my translation?**

**PPN**
Multi-Level Page Table

Doesn’t this take up more memory than before?
- YES, but..

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
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 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
For every instruction:

- MMU translates address (virtual → 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$ → L2 → Memory)

- Repeat at Memory stage for load/store insns
  - Translate address
  - Now you perform the load/store
Performance

- **Virtual Memory Summary**
- **Page Table 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
Next Goal

• How do we speedup address translation?
Translation Lookaside 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

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
Translation in Action

Virtual Address

TLB Access

TLB Hit?

Physical Address

$ Access

$ Hit?

DRAM Access

DRAM Hit?

deliver Data back to CPU

TLB miss handler (HW or OS)

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 \( \sim 4\text{KB} \) 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.