Virtual Memory

Anne Bracy
CS 3410
Computer Science
Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

P & H Chapter 5.7
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 MIPS 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

Processor

| LB | $1 ← M[1] |
| LB | $2 ← M[5] |
| LB | $3 ← M[1] |
| LB | $3 ← M[4] |
| LB | $2 ← M[0] |
| LB | $2 ← M[12] |
| LB | $2 ← M[5] |
| LB | $2 ← M[12] |
| LB | $2 ← M[5] |
| LB | $2 ← M[12] |
| LB | $2 ← M[5] |

Memory

| 0x000...0 |
| 0x7ff...f |
| 0xfff...f |

Cache

tag | data
--- | ---

Hits:

Misses:
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?
Q: Can we relocate second program?

A: Yes, but...

- What if they don’t fit?
- What if not contiguous?
- Need to recompile/relink?
- ...

Solution? Multiple processes/processors
Takeaway

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

– David Wheeler
– or, Butler Lampson
– or, Leslie Lamport
– or, Steve Bellovin

Solution: Need a **MAP**  
To map a **Virtual Address (generated by CPU)** to a **Physical Address (in memory)**
Goals for Today: Virtual Memory

What is Virtual Memory?

How does Virtual memory Work?

• Address Translation
  • Pages, page tables, and memory mgmt unit
• Paging
• Role of Operating System
  • Context switches, working set, shared memory
• Performance
  • How slow is it
  • Making virtual memory fast
  • Translation lookaside buffer (TLB)
• Virtual Memory Meets Caching
Big Picture: (Virtual) Memory

How do we execute *more than one* program at a time?

A: Abstraction – Virtual Memory

- Memory that *appears* to exist as main memory (most of it is supported by data held in secondary storage and transferred as required = ”paging”)
- Abstraction that supports multi-tasking—the ability to run more than one process at a time
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?
Virtual Memory

Virtual Memory: A Solution for All Problems

- Program/CPU can access any address from $0...2^N$ (e.g. $N=32$)
  (so addresses $0$ to $2^{32}$; $0x0000 0000$ to $0xFFFF FFFF$)

Each process has its own virtual address space

- 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
Virtual Address Space

Physical Address Space

Address Translation
Programs load/store to virtual addresses
Actual memory uses physical addresses

Memory Management Unit (MMU)

- Responsible for translating on the fly
- Essentially, just a big array of integers:
  \[ \text{paddr} = \text{PageTable}[\text{vaddr}] \]
Virtual Memory Advantages

Advantages

Easy relocation

- Loader puts code anywhere in physical memory
- Creates 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 PageTage, that maps a vaddr (a virtual address) to a paddr (physical address):

paddr = PageTable[vaddr]
Next Goal
How do we implement that translation from a virtual address (vaddr) to a physical address (paddr)?

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

i.e. How do we implement the PageTable??
Address Translation
Pages, Page Tables, and
the Memory Management Unit (MMU)
Attempt #1: Address Translation

How large should a PageTable be for a MMU?

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

Granularity?

- Per word…
- Per block…
- Variable…..

Typical:

- 4KB – 16KB pages
- 4MB – 256MB jumbo pages
Attempt #1: Address Translation

How large should a PageTable be for a MMU?

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

Granularity?

- 2\(^{32} = 4\text{GB}\)

- 4 bytes per word -> Need 1 billion entry PageTable!
  - Per word... \(2^{32} / 4 = 2^{32} / 2^2 = 2^{30} = 1\text{ billion pagetable entries}\)
  - Per block...
  - Variable.....

Typical:

- e.g. \(2^{32} / 4\ \text{kB} = 2^{32} / 2^{12} = 2^{20}\)
  \(2^{20} \implies 1\text{ million entry PageTable is better}\)

- 4\text{KB} – 16\text{KB} pages

- 4\text{MB} – 256\text{MB} jumbo pages
  e.g. \(2^{32} / 256\ \text{MB} = 2^{32} / 2^{28} = 2^{4}\)
  \(2^{4} \implies 16\text{ entry PageTable!}\)
Attempt #1: For any access to virtual address:

- Calculate virtual page number and page offset
- Lookup physical page number at PageTable[vpn]
- Calculate physical address as ppn:offset
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 PageTage, 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.
Next Goal

Example

How to translate a vaddr (virtual address) generated by the CPU to a paddr (physical address) used by main memory using the PageTable managed by the memory management unit (MMU).
Next Goal

Example

How to translate a vaddr (virtual address) generated by the CPU to a paddr (physical address) used by main memory using the PageTable managed by the memory management unit (MMU).

Q: Where is the PageTable stored??
Q: Where to store page tables?

a) In memory
b) In cache
c) In registers
d) On disk
e) None of the above
Q: Where to store page tables?
A: In memory, of course...

Special *page table base register*  
(CR3:PTBR on x86)  
(Cop0:ContextRegister on MIPS)
**Cool Trick #1: Don’t map all pages**

Need valid bit for each page table entry

Q: Why?

A: e.g. access to NULL will fail

A: we might not have that much physical memory

### Physical Page Number

<table>
<thead>
<tr>
<th>V</th>
<th>Number</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0x00000000</td>
</tr>
<tr>
<td>1</td>
<td>0x10045</td>
<td>0x90000000</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0x10044000</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0x4123B000</td>
</tr>
<tr>
<td>1</td>
<td>0xC20A3</td>
<td>0xC20A3000</td>
</tr>
<tr>
<td>1</td>
<td>0x4123B</td>
<td>0x4123B000</td>
</tr>
<tr>
<td>1</td>
<td>0x10044</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Cool Trick #2: Page permissions!

Keep R, W, X permission bits for each page table entry.

Q: Why?

A: can make code read-only, executable; make data read-write but not executable; etc.
Cool Trick #3: Aliasing
Map the same physical page at several virtual addresses
Q: Why?
A: can make different views of same data with different permissions
Page Size Example

Overhead for VM Attempt #1: How large is PageTable?

Virtual address space (for each process):

• total virtual memory: $2^{32}$ bytes = 4GB
• page size: $2^{12}$ bytes = 4KB
• entries in PageTable?
• size of PageTable?

Physical address space:

• total physical memory: $2^{29}$ bytes = 512MB
• overhead for 10 processes?
Page Size Example

Overhead for VM Attempt #1: How large is PageTable?

Virtual address space (for each process):

• total virtual memory: \(2^{32}\) bytes = 4GB
• page size: \(2^{12}\) bytes = 4KB
• entries in PageTable? \(2^{20} = 1\) million entries in PageTable
• size of PageTable? PageTable Entry (PTE) size = 4 bytes
  So, PageTable size = \(4 \times 2^{20} = 4\)MB

Physical address space:

• 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
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!!
Beyond Flat Page Tables
Assume most of PageTable is empty

How to translate addresses? Multi-level PageTable

* x86 does exactly this
Beyond Flat Page Tables
Assume most of PageTable is empty
How to translate addresses? Multi-level PageTable

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
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 and significantly reduce the overhead due to PageTables.
Next Goal

Can we run process larger than physical memory?
Paging
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

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>X</th>
<th>D</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0x10045</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>invalid</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>disk sector 200</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>disk sector 25</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0x00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>

**Physical Page Numbers**

<table>
<thead>
<tr>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x10045000</td>
</tr>
<tr>
<td>0x4123B000</td>
</tr>
<tr>
<td>0x90000000</td>
</tr>
<tr>
<td>0xC20A3000</td>
</tr>
</tbody>
</table>

**Disk Sectors**

- Sector 200
- Sector 25

**Need more bits:**

- Dirty
- RecentlyUsed

Putting it Together: Symbols

Basic Parameters
- \( N = 2^n \) : Number of addresses in virtual address space
- \( M = 2^m \) : Number of addresses in physical address space
- \( P = 2^p \) : Page size (bytes)

Components of the virtual address (VA)
- \( VPO \): Virtual page offset
- \( VPN \): Virtual page number

Components of the physical address (PA)
- \( PPO \): Physical page offset (same as VPO)
- \( PPN \): Physical page number
- \( CO \): Byte offset within cache line
- \( CI \): Cache index
- \( CT \): Cache tag
Putting it Together: Symbols

Addressing
- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes
## Putting it Together: Page Table

Only showing first 16 entries (out of 256)

VPN is the index into the Page Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Putting it Together: Cache

16 lines, 4-byte block size
Physically addressed
Direct mapped

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Putting it Together: Example #1

Virtual Address: \(0x03D4\)

![Virtual Address Diagram]

Physical Address

![Physical Address Diagram]
Putting it Together: Example #1

Virtual Address: \texttt{0x03D4}

<table>
<thead>
<tr>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

VPN \texttt{0x0F}

Page Fault? \texttt{N} 
PPN: \texttt{0x0D}

Physical Address

<table>
<thead>
<tr>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

CO \texttt{0} 
Cl \texttt{0x5} 
CT \texttt{0x0D} 
Hit? \texttt{Y} 
Byte: \texttt{0x36}
Putting it Together: Example #2

Virtual Address: \texttt{0xB8F}

Physical Address

VPN ___ Page Fault? ___ PPN: ____

CO ___ CI___ CT ____ Hit? ____ Byte: ____
Putting it Together: Example #2

Virtual Address: 0x0B8F

Physical Address

VPN 0x2E Page Fault? Y  PPN: TBD

Byte: ___

CO ___  CI___  CT ____  Hit? ___
Putting it Together: Example #3

Virtual Address: \texttt{0x0020}

VPN \(\ldots\) Page Fault? \(\ldots\) PPN: \(\ldots\)

Physical Address

CO \(\ldots\) CI \(\ldots\) CT \(\ldots\) Hit? \(\ldots\) Byte: \(\ldots\)
Putting it Together: Example #3

Virtual Address: \(0 \times 0020\)

```
  13 12 11 10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  0  0  0  0  1  0  0  0  0  0

VPN  VPO

VPN 0x00 Page Fault? N  PPN: 0x28
```

Physical Address

```
  11 10  9  8  7  6  5  4  3  2  1  0
  1  0  1  0  0  0  1  0  0  0  0  0

CT  CI  CO

CO 0  CI 0x8  CT 0x28  Hit? N  Byte: Mem
```
Summary

Virtual Memory

• Address Translation
  • Pages, page tables, and memory mgmt unit
• Paging

Next time

• Role of Operating System
  • Context switches, working set, shared memory
• Performance
  • How slow is it
  • Making virtual memory fast
  • Translation lookaside buffer (TLB)
• Virtual Memory Meets Caching
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
A TLB in the Memory Hierarchy

START: Memory instruction

TLB Lookup

Page Table Lookup

Page Fault:
- Invalid entry
- Page Fault: Kill Process

Access Memory Hierarchy

Disk

Virtual Addr ➔ Phyiscal Addr ➔ Update ➔ Location on Disk ➔ Data

Hit

1 cycle

Miss

data in memory:
Update TLB
Goto START

data not in memory:
Update Page Table
Goto START

Page Fault:
Load from disk,
fix page table,
try again
dozens of cycles

Timing varies
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
November 1988: Internet Worm

Internet Worm attacks thousands of Internet hosts

Best Wikipedia quotes:

“According to its creator, the Morris worm was not written to cause damage, but to gauge the size of the Internet. The worm was released from MIT to disguise the fact that the worm originally came from Cornell.”

“The worm ... determined whether to invade a new computer by asking whether there was already a copy running. But just doing this would have made it trivially easy to kill: everyone could run a process that would always answer "yes". To compensate for this possibility, Morris directed the worm to copy itself even if the response is "yes" 1 out of 7 times. This level of replication proved excessive, and the worm spread rapidly, infecting some computers multiple times. Morris remarked, when he heard of the mistake, that he "should have tried it on a simulator first".”
Project 2 is finished!

MIPS

Have some Chocolate Chip Cookies!