Isolation & Virtual Memory: Concepts

Zhaoguo Wang
Layered Organization

User Applications

Operating System

Software

Hardware

CPU
Memory
I/O
Isolation

User Applications

Operating System

Software

Hardware

CPU | Memory | I/O
## Isolation

**Software**

- User Applications
  - Firefox
  - CRASH
  - Adobe
  - Skype

- Operating System
  - Windows
  - Linux

- Hardware
  - CPU
  - Memory
  - I/O

**Isolation** – Enforced separation to contain effects of failures
Isolation – Enforced separation to contain effects of failures
Process

An instance of a computer program that is being executed.

Program vs. Process
- Program: a passive collection of instructions
- Process: the actual execution of those instructions

Different processes have different process id
- Function call `getpid()`: get the process id of the current process
- Shell command `ps`: list all processes
To run a program, OS starts a process and provide services through system call (`getpid()`, `printf()`).
Share the memory space among different processes

Observation
– Different processes can use the same address to store their local code/data.
– One process can not access another process’s memory

Why
– Isolation
  • prevent process X from *wrecking* on process Y
– Security
  • prevent process X from *spying* on process Y
– Simplicity
  • Systems (OS/Compiler) can handle different processes with the same code. (e.t.c. Linking or Loading)

How
– *Virtual Memory*
Our “Mental Model” of Memory System

**CPU**
- **PC:** 0x00…0058
- **IR:** `movq (%rax), %rbx`
- **RAX:** 0x38
- **RBX:** 0x1

**Memory**
- **addr:** 0x38
- **data:** 0x1

The diagram shows a CPU with registers PC, IR, RAX, and RBX, and a memory block with addresses ranging from 0x0 to 0x0…058. The instruction `movq (%rax), %rbx` is executed, moving the content of memory at address 0x38 (0x1) into the RBX register.
Real system – Virtual addressing

CPU
- PC: 0x00...0058
- IR: movq (%rax), %rbx
- RAX: 0x38
- RBX: 0x1

Memory Management Unit
- Virtual address: 0x38
- Physical address: 0x10
- Memory: 0x1
- data: 0x1
Real system – Virtual addressing

Virtual address space: Set of $2^n$ virtual addresses
- $n$ is the memory address width (32 or 64)
Address Translation – Strawman

MMU has a mapping table at byte granularity
  - Map each virtual address into a physical address

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x58</td>
<td>0x10</td>
</tr>
<tr>
<td>0x59</td>
<td>0x11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

mapping table
Address Translation – Strawman

MMU has a mapping table at byte granularity
- Map each virtual address into a physical address

Memory Management Unit

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x58</td>
<td>0x10</td>
</tr>
<tr>
<td>0x59</td>
<td>0x11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

mapping table

What is the size of mapping table?
Address Translation – Strawman

MMU has a mapping table at byte granularity
- Map each virtual address into a physical address

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x58</td>
<td>0x10</td>
</tr>
<tr>
<td>0x59</td>
<td>0x11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Mapping table

What is the size of mapping table?
Size of virtual address space $2^{64}$

MMU has a mapping table at byte granularity
- Map each virtual address into a physical address

What is the size of mapping table?
Size of virtual address space $2^{64}$
Address Translation – Page

Observation

– Both virtual memory space and physical memory space are contiguous

Build the mapping at coarse granularity

– Page: split the virtual/physical memory space into blocks with the same size.
– Page table: map the virtual pages to physical pages.
### Address Translation – Page

<table>
<thead>
<tr>
<th>Virtual Memory Space (Conceptual memory space)</th>
<th>Physical Memory Space (real memory space)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>0x0000000000000008</td>
<td>0x0000000000000008</td>
</tr>
<tr>
<td>0x0000000000000010</td>
<td>0x0000000000000010</td>
</tr>
<tr>
<td>0x0000000000000018</td>
<td>0x0000000000000018</td>
</tr>
<tr>
<td>0x0000000000000020</td>
<td>0x0000000000000020</td>
</tr>
<tr>
<td>0x0000000000000028</td>
<td>0x0000000000000028</td>
</tr>
<tr>
<td>0x0000000000000030</td>
<td>0x0000000000000030</td>
</tr>
<tr>
<td>0x0000000000000038</td>
<td>0x0000000000000038</td>
</tr>
<tr>
<td>0x0000000000000040</td>
<td>0x0000000000000040</td>
</tr>
<tr>
<td>0x0000000000000048</td>
<td>0x0000000000000048</td>
</tr>
<tr>
<td>0x0000000000000050</td>
<td>0x0000000000000050</td>
</tr>
<tr>
<td>0x0000000000000058</td>
<td>0x0000000000000058</td>
</tr>
<tr>
<td>0xffff...ffff</td>
<td>0x0000000000000000</td>
</tr>
</tbody>
</table>
Address Translation – Page

Virtual Memory Space (Conceptual memory space)

0x00000000 ...
0x00000018
0x00000020
0x00000028
0x00000030
0x00000038
0x00000040
0x00000048
0x00000050
0x00000058
0xfffffff...

Physical Memory Space (real memory space) e.g. 4GB

0x00000000 ...
0x00000010
0x00000018
0x00000020
0x00000028
0x00000030
0x00000038
0x00000040
0x00000048
0x00000050
0x00000058
...........
0xfffffff...

0x00000000 ...
0x00000010
0x00000018
0x00000020
0x00000028
0x00000030
0x00000038
0x00000040
0x00000048
0x00000050
0x00000058
...........
0xfffffff...

VP0
VP1
VP2
VP3
# Address Translation – Page

## Virtual Memory Space

(Conceptual memory space)

<table>
<thead>
<tr>
<th>Virtual Page Number (VPN)</th>
<th>Physical Page Number (PPN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP0</td>
<td>PP1</td>
</tr>
<tr>
<td>VP1</td>
<td>PP10</td>
</tr>
</tbody>
</table>

## Physical Memory Space

(real memory space)

- e.g. 4GB

<table>
<thead>
<tr>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP0</td>
</tr>
<tr>
<td>PP1</td>
</tr>
<tr>
<td>PP2</td>
</tr>
<tr>
<td>PP3</td>
</tr>
</tbody>
</table>

## MMU

Virtual Page Number (VPN) vs Physical Page Number (PPN) mapping is performed by the Memory Management Unit (MMU).
Address Translation – Page

Memory Management Unit

CPU

Virtual Address 0x1234

MMU

Offset in the page (0x8)

Virtual Page Number (VP0)

VPN PPN

VP0 PP1

VP1 PP10

...

...

Page Table

Virtual Memory Space (real memory space) e.g. 4GB

Physical Memory Space

0x0...ffff

......

......

0x0...58

0x0...50

0x0...48

0x0...40

0x0...38

0x0...30

0x0...28

0x0...20

0x0...18

0x0...10

0x0...08

0x0...00

PP0

PP1

PP2

PP3
Address Translation – Page

Memory Management Unit

CPU

Virtual Address
0x1234

MMU

Offset in the page (0x8)

VPN
VP0
VP1
...

PPN
PP1
PP10
...

Virtual Page Number (VP0)

Page Table

Physical Page Number (PP1)

Physical Address

0x0...fff
......
0x0...058
0x0...050
0x0...048
0x0...040
0x0...038
0x0...030
0x0...028
0x0...026
0x0...018
0x0...010
0x0...008
0x0...000

Physical Memory Space (real memory space)
e.g. 4GB

Data
What is the size of mapping table now? (virtual address space is $2^{64}$, page size is 4KB)
What is the size of mapping table now? (virtual address space is $2^{64}$, page size is 4KB)

Ans: $2^{52}$
Address Translation

Virtual Address $\rightarrow$ Physical Address
- Calculate the virtual page number
- Locate the data from the according physical page

Memory address width: 64 bits
Page size: 4 KB ($2^{12}$)
Address Translation

Virtual Address → Physical Address
- Calculate the virtual page number
- Locate the data from the according physical page

Memory address width: 64 bits
Page size: 4 KB ($2^{12}$)

<table>
<thead>
<tr>
<th>63</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual page number (VPN)</td>
<td>Page offset (VPO)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Address Translation

Memory Management Unit

CPU

Virtual Address
0x1234

access 0x1234

VPN = Vaddr >> 12
(0x1234 >> 12) \(\rightarrow\) 0x1

VPN | PPN
--- | ---
VP0 | PP1
VP1 | PP3
... | ...

Page Table

Physical Memory Space
(real memory space)
e.g. 4GB

0x0…ffff
......
......
0x0…3008
0x0…3000
......
0x0…2008
0x0…2000
......
0x0…1008
0x0…1000
......
0x0…008
0x0…000

PP0
PP1
PP2
PP3
Address Translation

CPU

Virtual Address
0x1234

access 0x1234

Memory Management Unit

VPN = Vaddr >> 12
(0x1234 >> 12) \(\Rightarrow\) 0x1

Offset = Vaddr & 0xffff
(0x1234 & 0xffff) \(\Rightarrow\) 0x234

VPN

PPN

VP0

PP1

VP1

PP3

...

...

Page Table

Physical Memory Space
(real memory space)
e.g. 4GB

0x0...fff

......

......

0x0...3008

0x0...3000

......

0x0...2008

0x0...2000

......

0x0...1008

0x0...1000

......

0x0...008

0x0...000

PP0

PP1

PP2

PP3
Address Translation

Memory Management Unit

Virtual Address 0x1234

VPN = Vaddr >> 12
(0x1234 >> 12) \(\Rightarrow\) 0x1

Offset = Vaddr & 0xfff
(0x1234 & 0xfff) \(\Rightarrow\) 0x234

VPN
VP0
VP1
...

PPN
PP1
PP3
...

Page Table

PP: PP3

Physical Memory Space (real memory space)
e.g. 4GB

CPU

access 0x1234

0x1234

MMU

VPN = Vaddr >> 12
(0x1234 >> 12) \(\Rightarrow\) 0x1

Offset = Vaddr & 0xfff
(0x1234 & 0xfff) \(\Rightarrow\) 0x234

VPN = Vaddr >> 12
(0x1234 >> 12) \(\Rightarrow\) 0x1

Physical Memory Space (real memory space)
e.g. 4GB
**Exercise**

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP0</td>
<td>PP1</td>
</tr>
<tr>
<td>VP1</td>
<td>PP3</td>
</tr>
<tr>
<td>VP2</td>
<td>PP4</td>
</tr>
<tr>
<td>VP4</td>
<td>PP0</td>
</tr>
<tr>
<td>VP5</td>
<td>PP3</td>
</tr>
<tr>
<td>VP6</td>
<td>PP2</td>
</tr>
</tbody>
</table>

**Page Table**

Page size: 4KB  
Address width: 64

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td></td>
</tr>
<tr>
<td>0x4321</td>
<td></td>
</tr>
<tr>
<td>0x5678</td>
<td></td>
</tr>
<tr>
<td>0x2222</td>
<td></td>
</tr>
<tr>
<td>0x4567</td>
<td></td>
</tr>
<tr>
<td>0x5234</td>
<td></td>
</tr>
</tbody>
</table>
Exercise

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP0</td>
<td>PP1</td>
</tr>
<tr>
<td>VP1</td>
<td>PP3</td>
</tr>
<tr>
<td>VP2</td>
<td>PP4</td>
</tr>
<tr>
<td>VP4</td>
<td>PP0</td>
</tr>
<tr>
<td>VP5</td>
<td>PP3</td>
</tr>
<tr>
<td>VP6</td>
<td>PP2</td>
</tr>
</tbody>
</table>

Page Table

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td>0x3234</td>
</tr>
<tr>
<td>0x4321</td>
<td>0x0321</td>
</tr>
<tr>
<td>0x5678</td>
<td>0x3678</td>
</tr>
<tr>
<td>0x2222</td>
<td>0x4222</td>
</tr>
<tr>
<td>0x4567</td>
<td>0x0567</td>
</tr>
<tr>
<td>0x5234</td>
<td>0x3234</td>
</tr>
</tbody>
</table>

Page size: 4KB
Address width: 64
Multi-level page tables

Problem
- For 64 bit memory address and 4KB page size, the page table size is $2^{52}$, could you further reduce the size?

Solution
- Multi-level page
  - 3 level page table on 32 bit machine
  - 4 level page table on 64 bit machine
Multi-level page tables on X86_64

Entry at level \( i \) points to the physical address of page at level \( i - 1 \).
Multi-level page tables on X86_64

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>39</th>
<th>38</th>
<th>30</th>
<th>29</th>
<th>21</th>
<th>20</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>L0 Offset</td>
<td>L1 Offset</td>
<td>L2 Offset</td>
<td>L3 Offset</td>
<td>Page Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current mapping uses 48 bits, we we are limited to $2^{48}$ bytes which is about 262 Terabytes
Multi-level page tables on X86_64

CPU Register: CR0

Root Addr

Level 0

Level 1

Level 2

Level 3
Multi-level page tables on X86_64

CPU Register: CR0

Root Addr

Level 0

Level 1

Level 2

Level 3
Multilevel page tables on X86_64

Virtual Address: 0x80801fffa8

Physical Address: 0x5799fa8
Review The First Question

How can each process have the same virtual address space?

– Each process has its own page table
– When executing a process, the operating system uses its local page table to do the address translation.
Virtual Address Space For Each Process

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:
Questions After the Class

For 32 bit memory address and 4KB page size, the page table size is $2^{20}$, could you further reduce the size?
To access the data with some va (e.g., 0x1234), how many memory accesses in total?
To access the data with some va (e.g., 0x1234), how many memory accesses in total? 4 times page table access plus one time data access which is 5 times memory accesses.
### Speedup Address Translation

Translation lookaside buffer (TLB)
- Small memory cache in MMU
- Maps virtual page numbers to physical page numbers

1. Calculate VPN
   a. $\text{VPN} = \text{VA} >> 12$

2. Check TLB
   a. Index = $hash(\text{VPN})$, $\text{VPN} \mod 4$
   b. Check if $\text{TLB}[\text{Index}].\text{VPN} == \text{VPN}$
   c. On TLB hit,
      PA = $\text{TLB}[\text{Index}].\text{PPN} + \text{Offset}$
      a. On TLB miss
         Go through page table to get PPN
         Buffer the result in TLB

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP0</td>
<td>PP1</td>
</tr>
<tr>
<td>VP1</td>
<td>PP3</td>
</tr>
<tr>
<td>unused</td>
<td>unused</td>
</tr>
<tr>
<td>unused</td>
<td>unused</td>
</tr>
</tbody>
</table>

**Diagram:**
- CPU
- MMU
- TLB (4 Entries)
- VA: 0x1234
Speedup Address Translation

Translation lookaside buffer (TLB)
- Small memory cache in MMU
- Maps virtual page numbers to physical page numbers

1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = hash(VPN), VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go through page table to get PPN
      Buffer the result in TLB

Example:
1. VPN = 0x1234 >> 12 = 0x1
2. TLB_Index = 0x1 % 4 = 1
3. Check TLB[1].VPN which is VP1
4. On TLB hit, PA = 0x234 + PP3 = 0x3234
Speedup Address Translation

Translation lookaside buffer (TLB)
- Small memory cache in MMU
- Maps virtual page numbers to physical page numbers

Example:
1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = hash(VPN), VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go through page table to get PPN
      Buffer the result in TLB

Example:
1. VPN = 0x2234 >> 12 = 0x2
2. TLB_Index = 0x2 % 4 = 1
3. Check TLB[2].VPN which is Empty
4. Go through the page table
Speedup Address Translation

Translation lookaside buffer (TLB)
- Small memory cache in MMU
- Maps virtual page numbers to physical page numbers

Example:
1. Calculate VPN
   a. $\text{VPN} = \text{VA} \gg 12$
2. Check TLB
   a. Index = hash($\text{VPN}$), $\text{VPN} \% 4$
   b. Check if $\text{TLB}[\text{Index}].\text{VPN} == \text{VPN}$
   c. On TLB hit,
      - PA = $\text{TLB}[\text{Index}].\text{PPN} + \text{Offset}$
      - On TLB miss
        - Go through page table to get PPN
        - Buffer the result in TLB

Example:
1. $\text{VPN} = 0x2234 \gg 12 = 0x2$
2. $\text{TLB\_Index} = 0x2 \% 4 = 1$
3. Check TLB[2].VPN which is Empty
4. Go through the page table
5. Buffer the result in TLB
Speedup Address Translation

1. Calculate VPN
   a. VPN = VA >> 12

2. Check TLB
   a. Index = hash(VPN), VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

Access pattern
- access 0x1234
- access 0x5234
- access 0x1234
- access 0x5234

TLB:
- access 0x1234, TLB Miss
1. Calculate VPN
   a. $\text{VPN} = \text{VA} >> 12$

2. Check TLB
   a. $\text{Index} = \text{hash(VPN)}, \text{VPN} \% 4$
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      
      $\text{PA} = \text{TLB[Index].PPN} + \text{Offset}$
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

**TLB:**
access $0x1234$, TLB Miss, cache VP1<->PP3
1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = hash(VPN), VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit, PA = TLB[Index].PPN + Offset
      a. On TLB miss
         Go though page table to get PPN
         Buffer the result in TLB

TLB:
access 0x1234, TLB Miss, cache VP1<->PP3
access 0x5234, TLB Miss, evict VP1<->PP3, cache VP5<->PP8
Speedup Address Translation

1. Calculate VPN
   a. \( VPN = VA >> 12 \)

2. Check TLB
   a. Index = \( \text{hash}(VPN) \), \( VPN \% 4 \)
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      \[ PA = \text{TLB[Index].PPN} + \text{Offset} \]
      a. On TLB miss
      Go through page table to get PPN
      Buffer the result in TLB

TLB:
access 0x1234, TLB Miss, cache VP1<->PP3
access 0x5234, TLB Miss, evict VP1<->PP3, cache VP5<->PP8
access 0x5234, TLB Miss, evict VP5<->PP8, cache VP1<->PP3
Speedup Address Translation

1. Calculate VPN
   a. \( VPN = VA >> 12 \)

2. Check TLB
   a. \( Index = hash(VPN), VPN \% 4 \)
   b. Check if \( TLB[Index].VPN == VPN \)
   c. On TLB hit,
      \[ PA = TLB[Index].PPN + Offset \]
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

TLB:
access 0x1234, TLB Miss, cache VP1<->PP3
access 0x5234, TLB Miss, evict VP1<->PP3, cache VP5<->PP8
access 0x5234, TLB Miss, evict VP5<->PP8, cache VP1<->PP3
access 0x5234, TLB Miss, evict VP5<->PP8, cache VP1<->PP3

TLB eviction due to hash conflict!
Speedup Address Translation

1. Calculate VPN
   a. $\text{VPN} = \text{VA} >> 12$

2. Check TLB
   a. $\text{Index} = \text{hash(VPN)}, \text{VPN} \% 4$
   b. Check if $\text{TLB[Index].VPN} == \text{VPN}$
   c. On TLB hit,
      $\text{PA} = \text{TLB[Index].PPN} + \text{Offset}$
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

Access pattern
access $0x1234$
access $0x5234$
access $0x1234$
access $0x5234$

TLB:
access $0x1234$, TLB Miss, cache VP1<->PP3
access $0x5234$, TLB Miss, evict VP1<->PP3, cache VP5<->PP8
access $0x5234$, TLB Miss, evict VP5<->PP8, cache VP1<->PP3
access $0x5234$, TLB Miss, evict VP5<->PP8, cache VP1<->PP3

TLB eviction due to hash conflict! $\Rightarrow$ Double hashing
TLB with two level hash

TLB (16 sets, 4 way associative)

Set 0

Set 1

......

Set 15

MMU
TLB with two level hash

TLB (16 sets, 4 way associative)
TLB with two level hash

TLB (16 sets, 4 way associative)

MMU
Example

access 0x11234, TLB Miss
access 0x21234
access 0x31234
access 0x11234
Example

access 0x11234, TLB Miss
access 0x21234, TLB Miss
access 0x31234, TLB Miss
access 0x11234

TLB (16 sets, 4 way associative)

MMU
Example

access 0x11234, TLB Miss
access 0x21234, TLB Miss
access 0x31234, TLB Miss
access 0x11234

TLB (16 sets, 4 way associative)
Example

access 0x11234, TLB Miss
access 0x21234, TLB Miss
access 0x31234, TLB Miss
access 0x11234, TLB Hit

TLB (16 sets, 4 way associative)

MMU
Summary

Virtual Address → TLB Access

Hit? →

No → Walk Though Page Table

Yes →

PPN + Offset → Physical Address
Simplifying Linking and Loading

• Linking
  – Each program has similar virtual address space
  – Code, data, and heap always start at the same addresses.

• Loading
  – \texttt{os} allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  – The .text and .data sections are copied, page by page, on demand by the virtual memory system