Isolation & Virtual Memory: Concepts

Jinyang Li

based on the slides of Tiger 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

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

An instance of a computer program that being executed

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

Different processes have different process id
- `getpid()` function call returns id of current process
- Command `ps`: list all processes
To run a program, OS starts a process and provide services through system call (\textit{getpid()}, \textit{printf()}).
Our “Mental Model” of Memory System

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

Memory
- addr: 0x38
- data: 0x1

...
Processes share the same address space

The requirements:
- Different processes use the same address to store their local code/data.
- One process cannot access another process’ memory

Why
- Isolation
  - prevent process X from damaging 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
Real system – Virtual addressing

CPU

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

Memory Management Unit (MMU)

Virtual address: 0x38

Physical address: 0x10

Memory

- Physical address: 0x10
- Virtual address: 0x38
- Data: 0x1

Memory locations:
- 0x0...058
- 0x0...050
- 0x0...048
- 0x0...040
- 0x0...038
- 0x0...030
- 0x0...028
- 0x0...020
- 0x0...018
- 0x0...010
- ...
Real system – Virtual addressing

Virtual address space:
- 64-bit addresses: [0, 2^64-1]
- 32-bit addresses: [0, 2^32-1]

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

Memory Management Unit
- Virtual address 0x38
- Physical address 0x10
- Physical address 0x10 maps to memory location 0x1

Memory
- data: 0x1
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>0x58</td>
<td>0x10</td>
</tr>
<tr>
<td>0x59</td>
<td>0x11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

mapping table

**Diagram:**
- CPU
  - PC: 0x00...0058
  - IR: `movq (%rax), %rbx`
  - RAX: 0x38
  - RBX: 0x1
- MMU
  - Virtual address mapping table
  - Physical address mapping table
- Memory
  - Physical address 0x10
  - Virtual address 0x38
  - Data: 0x1

**MMU**
- Memory Management Unit
  - Map each virtual address into a physical address
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?
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>

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

Virtual Memory Space
(Conceptual memory space)

0x00000000
0x00000010
0x00000020
0x00000030
0x00000040
0x00000048
0x00000050
0x00000058

0xffff

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

0x00000000
0x00000008
0x00000010
0x00000018
0x00000020
0x00000028
0x00000030
0x00000038
0x00000040
0x00000048
0x00000050
0x00000058

0xffff

......
Address Translation – Page

Virtual Memory Space (Conceptual memory space)

0x000...
0x008...
0x010...
0x018...
0x020...
0x028...
0x030...
0x038...
0x040...
0x048...
0x050...
0x058...
0xfff...

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

0x000...
0x008...
0x010...
0x018...
0x020...
0x028...
0x030...
0x038...
0x040...
0x048...
0x050...
0x058...
0xffff...
Address Translation – Page

Virtual Memory Space (Conceptual memory space)

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

MMU

<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>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Page Table
Address Translation – Page

CPU access 0x1234

Virtual Address 0x1234

MMU

Virtual Page Number (VP0)

VPN PPN
VP0 PP1
VP1 PP10
...

Page Table

Offset in the page (0x8)

Memory Management Unit

0x0…ffff
......
......
0x0…058
0x0…050
0x0…048
0x0…040
0x0…038
0x0…030
0x0…028
0x0…020
0x0…018
0x0…010
0x0…008
0x0…000

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

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

...

Page Table

Virtual Page Number

(VP0)

Physical Page Number

(PP1)

Physical Address

0x0...fff

......

......

0x0...058

0x0...050

0x0...048

0x0...040

0x0...038

0x0...030

0x0...028

0x0...020

0x0...018

0x0...010

0x0...008

0x0...000

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

Data
Address Translation – Page

CPU

Virtual Address
0x1234

MMU

Offset in the page
(0x8)

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

Virtual Page Number
(VP0)

VPN
VP0
VP1
…

Physical Page Number
(PP1)

Physical Page Number
(PP10)

…

Page Table

VPN
PPN
VP0
PP1
VP1
PP10
…

Physical Address

Physical Memory Space
(0x0…0ff)

0x0…058
0x0…050
0x0…048
0x0…040
0x0…038
0x0…030
0x0…028
0x0…020
0x0…018
0x0…010
0x0…008
0x0…000

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 $\rightarrow$ Physical Address

- Calculate the virtual page number
- Locate the data from the according physical page

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

<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

MMU

access 0x1234

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

VPN
VP0
VP1
...

PPN
PP1
PP3
...

Page Table

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

0x0...ffffff
...0x3008
0x0...3000
......
......
0x0...208
0x0...2000
......
......
0x0...1008
0x0...1000
......
......
0x0...0008
0x0...000
0x0...000

...
Address Translation

Memory Management Unit

Virtual Address: 0x1234

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

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

Page Table

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

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

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

...
Address Translation

CPU 0x1234

Virtual Address: 0x1234

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

MMU

VPN
VP0
VP1
...

PPN
PP1
PP3
...

Page Table

Offset = Vaddr & 0xffff
(0x1234 & 0xffff) => 0x234

0x0...fff

0x3234

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

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

```
<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 size: 4KB  
Address width: 64  

**Exercise**

<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**

Page size: 4KB
Address width: 16

<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 table entries encode permission information

Conceptual Page Table

<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>

Actual Page Table

8-byte Page Table Entry (PTE)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page #</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

PTE format

- S: present?
- W: writable?
- accessible by OS only?

Question: how many PTEs per page?

$4KB/8 = 2^{12}/2^3 = 2^9$
What we have learnt: VM memory translation

CPU

Virtual Address

0x1234

MMU

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

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

Page Table

VPN
VP0
VP1
...

PPN
PP1
PP3
...

0x0...008
0x0...000

0x0...3008
0x0...3000

0x0...2008
0x0...2000

0x0...1008
0x0...1000

0x0...008
0x0...000

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

- Multi-level page tables
- Demand paging
- Accelerating address translation
Problem with 1-level page table:

- For 64-bit address space and 4KB page size, what is the number of page table entries required for translation?

\[
\frac{2^{64}}{2^{12}} = 2^{52}
\]

- number of bytes addressable in 64-bit address space
- # of pages in 64-bit address space
- page size

= # of page table entries required
Multi-level page tables

Problem
• how to reduce # of page table entries required?

Solution
• Multi-level page table
  – x86-64 supports 4-level page table
Multi-level page tables on X86_64

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

Current mapping uses 48 bits, programs can address \(2^{48}\) bytes, i.e. \(~256\) TB

4-level page table
Multi-level page tables on X86_64

CPU Register: CR3

4-level page table
Multi-level page tables on X86_64

4-level page table

CPU Register: CR3

Root Addr

Level 0

Reserved L0 Offset L1 Offset L2 Offset L3 Offset Page Offset

Level 1

Level 2

Level 3
Multi-level page tables on X86_64

Virtual Address: \texttt{0x80801fffa8}

<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>0x0</td>
<td>0x1</td>
<td>0x2</td>
<td>0x0</td>
<td>0x1ff</td>
<td>0xfa8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reserved | L0 Offset | L1 Offset | L2 Offset | L3 Offset | Page Offset

CPU Register: CR3

\texttt{0x4fff000} -> \texttt{0x3466000}

Physical address of the 1st entry at level 0

\texttt{0x3467000}
\texttt{0x3468000}
\texttt{...}

unused

Level 0

Physical address of the 1st entry at level 1

4-level page table
Multi-level page tables on X86_64

Virtual Address: 0x80801fffda8

CPU Register: CR3

Physical address of the 1st entry at level 0

0x4ffff000

Physical address of the 1st entry at level 1

0x3466000
0x3467000
0x3468000
...
unused

Level 0

4-level page table
Multi-level page tables on X86_64

Virtual Address: 0x80801fffa8

CPU Register: CR3

0x4ffff00

Physical address of the 1st entry at level 0

Level 0

Physical address of the 1st entry at level 1

4-level page table
Multi-level page tables on X86_64

Virtual Address: 0x80801fffa8

Physical address of the 1st entry at level 0

Level 0

Physical address of the 1st entry at level 1

Level 1

Physical address of the 1st entry at level 2

4-level page table
Multi-level page tables on X86_64

Virtual Address: 0x80801fff8

CPU Register: CR3

Physical address of the 1st entry at level 0: 0x3466000

Level 0

Physical address of the 1st entry at level 1: 0x3587000

Level 1

Physical address of the 1st entry at level 2: 0x3678000

Level 2

4-level page table
Multi-level page tables on X86_64

Virtual Address: 0x80801fff8

CPU Register: CR3

Level 0
- Physical address of the 1st entry at level 0
  - 0x3466000
  - 0x3467000
  - 0x3468000
  - ... unused

Level 1
- Physical address of the 1st entry at level 1
  - 0x3587000
  - unused
  - 0x3588000
  - ... unused

Level 2
- Physical address of the 1st entry at level 2
  - 0x3578000
  - unused

4-level page table
Multi-level page tables on X86_64

Virtual Address: 0x80801fffa8

CPU Register: CR3

0x4ffffff00

Physical address of the 1st entry at level 0

0x3466000
0x3467000
0x3468000

... unused

Level 0

Physical address of the 1st entry at level 1

0x3587000

unused
0x3588000

... unused

Level 1

Physical address of the 1st entry at level 2

0x3678000

unused
0x3579000

... unused

Level 2

Physical address of the 1st entry at level 3

0x5788000

0x5789000
0x578a000

... unused

Level 3

Physical address of the page
Multi-level page tables on X86_64

Virtual Address: 0x80801fffa8

CPU Register: CR3

Physical address of the 1st entry at level 0

Level 0

Physical address of the 1st entry at level 1

Level 1

Physical address of the 1st entry at level 2

Level 2

Physical address of the 1st entry at level 3

Level 3

4-level page table
Multi-level page tables on X86_64

Virtual Address: 0x80801fffa8

CPU Register: CR3

Physical Address of the 1st entry at level 0: 0x3466000

Physical Address of the 1st entry at level 1: 0x3587000

Physical Address of the 1st entry at level 2: 0x3678000

Physical Address of the 1st entry at level 3: 0x5788000

Physical Address: 0x5799fa8
Review Virtual Address

How can each process have the same virtual address space?

- OS sets up a separate page table for each process
- When executing a process $p$, MMU uses $p$’s page table to do address translation.
Virtual Address Space For Each Process

Virtual Address Space for Process 1:

Virtual memory

vaddr1
vaddr2
vaddr3
...

Virtual Address Space for Process 2:

Virtual memory

vaddr1
vaddr2
vaddr3
...

Page Table 1

Main memory

PP 0
PP 1
PP 2
PP k-2
PP k-1
PP k
Question: why does multi-level PT save PT memory overhead?

Question: why does multi-level page table save page table size?

Answer:
• 4-level page table is not fully occupied.
• Demand paging:
  – OS constructs page table on demand
Demand Paging

Memory Allocation (e.g., \( p = \text{sbrk}(8192) \))

User program to OS:
- Declare a virtual address range from \( p \) to \( p + 8192 \) for use by the current process.

OS’ actions:
- Allocate the physical page and populate the page table.
Demand Paging

```c
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's'
```

1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process’ virtual address info

CPU Register: CR3

```
0x4ffff000
```

```
0x3466000
unused
unused
...
unused
```

current process’ page table
Demand Paging

```c
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's'
```

1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process’ virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)

**current process’ page table**
Demand Paging

```c
char *p = (char *)malloc(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's'
```

1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process’ virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)
3. OS constructs the mapping for the address. (*Page fault handler*)

---

**Current process’ page table**
Demand Paging

```c
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's'
```

Level 0 Offset

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x3668000</td>
</tr>
<tr>
<td>0x1</td>
<td>unused</td>
</tr>
<tr>
<td>0x2</td>
<td>unused</td>
</tr>
<tr>
<td>0x0</td>
<td>0x3588000</td>
</tr>
<tr>
<td>0x1ff</td>
<td>unused</td>
</tr>
<tr>
<td>0xfa8</td>
<td>unused</td>
</tr>
</tbody>
</table>
```

CPU Register: CR3

```
0x4ffff000
```

New pages allocated by OS

```
Level 0

Level 1

Level 2

Level 3
```

current process’ page table

OS constructs the mapping for the address. (Page fault handler)
Demand Paging

char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's'

1. OS adds [p, p+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (Page fault)
3. OS constructs the mapping for the address. (Page fault handler)
4. OS tells the CPU to resume execution

**current process’ page table**
1. OS adds \([p, p+8192)\) to the process' virtual address info.
2. MMU tells OS entry 1 is missing in the page at level 0. (Page fault)
3. OS constructs the mapping for the address. (Page fault handler)
4. MMU translates address again and access the physical memory.

```c
char *p = (char *)sbrk(8192); // p is 0x800801fffa8
p[0] = 'c'
p[4096] = 's'
```

**Demand Paging**

**current process’ page table**
1. MMU tells OS entry 0x200 is missing in the page at level 3. (Page fault)

```c
char *p = (char *)bsrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's' //0x8080200fa8
```
2. OS constructs the mapping for the address. (*Page fault handler*)

```c
char *p = (char *)malloc(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's' //0x8080200fa8
```
Demand Paging

```
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's' //0x8080200fa8
```

3. OS tells the CPU to resume execution
4. MMU translates the address again and access the physical memory.

```c
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's' //0x8080200fa8
```
Questions

What is the minimal page table size on 64 bit machine?

4 pages

Given the minimal page table, how many physical pages it can refer to?

$$\frac{2^{12}}{2^3} = \frac{4096}{8} = 512$$

page size

size of each page table entry
Understanding Seg Fault

• Where does segmentation fault come from?
• Address translation fails due to 2 reasons
  – MMU reads a missing page table entry (PTE)
    • PTE’s present bit is unset
  – MMU reads a PTE with wrong permission for the access
    • write bit is unset for a write access
    • OS bit is set for user program access
• MMU generates “page fault”, to be handled by OS
• OS either fixes the problem (e.g. demand paging) or aborts process with “segmentation fault”
Memory Access Cost

Memory access latency
  – 100 ns
  – 160 ~ 200 CPU cycles

Instructions that do not involve memory access can execute very quickly:
  – Instructions per CPU cycle >= 1
Address translation is potentially very costly

To access the data with some va (e.g., 0x1234), how many memory accesses in total?
Address translation is potentially very costly

To access the data with some va (e.g., 0x1234), how many memory accesses in total? 4 page table accesses plus one time data access which is 5 memory accesses.
Translation lookaside buffer (TLB)
- Small cache in MMU
- Maps virtual page numbers to physical page numbers

1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = 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
Speedup Address Translation

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

1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = VPN % 4
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      \[ PA = TLB[Index].PPN + \text{Offset} \]
      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 cache in MMU
- Maps virtual page numbers to physical page numbers

1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = 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

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

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

1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = 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

Example:
1. VPN = 0x2234 >> 12 = 0x2
2. TLB_Index = 0x2 % 4 = 2
3. Check TLB[2].VPN which is Empty
4. Go through the page table
5. Buffer the result in TLB
Latency

Memory access
  – Hundreds of CPU cycles

TLB access
  – Only a couple of CPU cycles
Summary

Virtual Address

TLB Access

Hit ?

No

Walk Though Page Table

Yes

PPN + Offset

Physical Address
Recall the graph of a program’s memory layout

Answer: This is the virtual memory space of single process.
More on TLBs
Speedup Address Translation

1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = 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

Both 0x1234 and 0x5234 go to the entry 1
Speedup Address Translation

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

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

Both 0x1234 and 0x5234 go to the entry 1

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

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

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

Both 0x1234 and 0x5234 go to the entry 1

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 = 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

Both 0x1234 and 0x5234 go to the entry 1

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

1. Calculate VPN
   a. $VPN = VA \gg 12$

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

Both 0x1234 and 0x5234 go to the entry 1

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

TLB eviction due to conflict!
Speedup Address Translation

1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = 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

Both 0x1234 and 0x5234 go to the entry 1

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

TLB eviction due to conflict! → Multi-set associative TLB
Multi-set associative TLB

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>……</th>
<th>Set 15</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Tag PPN" /></td>
<td><img src="image2.png" alt="Tag PPN" /></td>
<td><img src="image3.png" alt="Tag PPN" /></td>
<td><img src="image4.png" alt="Tag PPN" /></td>
</tr>
<tr>
<td><img src="image1.png" alt="Tag PPN" /></td>
<td><img src="image2.png" alt="Tag PPN" /></td>
<td><img src="image3.png" alt="Tag PPN" /></td>
<td><img src="image4.png" alt="Tag PPN" /></td>
</tr>
<tr>
<td><img src="image1.png" alt="Tag PPN" /></td>
<td><img src="image2.png" alt="Tag PPN" /></td>
<td><img src="image3.png" alt="Tag PPN" /></td>
<td><img src="image4.png" alt="Tag PPN" /></td>
</tr>
<tr>
<td><img src="image1.png" alt="Tag PPN" /></td>
<td><img src="image2.png" alt="Tag PPN" /></td>
<td><img src="image3.png" alt="Tag PPN" /></td>
<td><img src="image4.png" alt="Tag PPN" /></td>
</tr>
</tbody>
</table>

TLB (16 sets, 4 way associative)
Multi-set associative TLB

TLB (16 sets, 4 way associative)
Multi-set associative TLB

TLB (16 sets, 4 way associative)

MMU
Example

- access \(0x11234\), TLB Miss
- access \(0x21234\)
- access \(0x11234\)
- access \(0x21234\)

TLB (16 sets, 4 way associative)

---

**Set 0**
- v Tag PPN
- v Tag PPN
- v Tag PPN
- v Tag PPN

**Set 1**
- v Tag PPN
- v Tag PPN
- v Tag PPN
- v Tag PPN

**Set 15**
- v Tag PPN
- v Tag PPN
- v Tag PPN
- v Tag PPN

**MMU**
Example

- access 0x11234, TLB Miss, cache the translation result
- access 0x21234
- access 0x11234
- access 0x21234
Example

access 0x11234, TLB Miss, cache the translation result
access 0x21234, TLB Miss,
access 0x11234
access 0x21234

TLB (16 sets, 4 way associative)
Example

access 0x11234, TLB Miss, cache the translation result
access 0x21234, TLB Miss, cache the translation result
access 0x11234
access 0x21234
Example

access 0x11234, TLB Miss, cache the translation result
access 0x21234, TLB Miss, cache the translation result
access 0x11234
access 0x21234
Example

access 0x11234, TLB Miss, cache the translation result
access 0x21234, TLB Miss, cache the translation result
access 0x11234, TLB Hit
access 0x21234, TLB Hit