Isolation & Virtual Memory: Concepts

Zhaoguo Wang
Layered Organization

Hardware
- CPU
- Memory
- I/O

Software
- Operating System
  - Windows
  - Linux

User Applications
- Firefox
- Safari
- Adobe
- Skype
Isolation

User Applications

Operating System

Software

Hardware

CPU  Memory  I/O
Isolation

- Enforced separation to contain effects of failures
Isolation – Enforced separation to contain effects of failures
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
- Function call `getpid()`: get the process id of current process
- Shell command `ps`: list all processes
To run a program, OS starts a process and provide services through system call (*getpid()*, *printf()*).
Our “Mental Model” of Memory System

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

Memory
addr: 0x38
0x0...058
0x0...050
0x0...048
0x0...040
0x0...038
0x0...030
0x0...028
0x0...020
0x0...018
0x0...010
...

data: 0x1
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
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: 0x0...058
  - data: 0x1
Real system – Virtual addressing

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

Memory Management Unit
Virtual address 0x38

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

Memory
Physical address 0x10

0x0…058
0x0…050
0x0…048
0x0…040
0x0…038
0x0…030
0x0…028
0x0…020
0x0…018
0x0…010
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>...</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

<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 the 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)

0xffff...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

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
Address Translation – Page

Virtual Memory Space
(Conceptual memory space)

Physical Memory Space
(real memory space)
e.g. 4GB
Address Translation – Page

Virtual Memory Space (Conceptual memory space)

| VP0   | PP0 |
| VP1   | PP1 |
| VP2   | PP2 |
| VP3   | PP3 |

Virtual Page Number (VPN) | Physical Page Number (PPN)
---|---
VP0 | PP1
VP1 | PP10

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

Page Table

MMU
Address Translation – Page

Memory Management Unit

CPU

Virtual Address 0x1234

MMU

Virtual Page Number (VP0)

VPN  PPN
VP0   PP1
VP1   PP10
...

Page Table

Offset in the page (0x8)

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

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
Address Translation – Page

Memory Management Unit

CPU

Virtual Address

0x1234

access

0x1234

MMU

Virtual Page Number (VP0)

VPN

PPN

PP0

VP1

PP10

...

...

Page Table

Offset in the page (0x8)

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 Page Number (PP1)

Physical Address

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 → 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></td>
<td>Page offset (VPO)</td>
<td></td>
</tr>
</tbody>
</table>
Address Translation

Memory Management Unit

CPU ➔ MMU

Virtual Address: 0x1234

VPN = Vaddr >> 12
(0x1234 >> 12) ➔ 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

...
Address Translation

CPU

Virtual Address
0x1234

access 0x1234

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

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

VPN
VP0
VP1
...

PPN
PP1
PP3
...

Page Table

Memory Management Unit

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

0x0...fff
.....
.....
.....
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

PP0
PP1
PP2
PP3

...
Address Translation

Memory Management Unit

Virtual Address: 0x1234
VPN = Vaddr >> 12
(0x1234 >> 12) \(\Rightarrow\) 0x1
Offset = Vaddr & 0xffff
(0x1234 & 0xffff) \(\Rightarrow\) 0x234

CPU

Page Table

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

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

0x0...008
0x0...000
0x0...1000
0x0...1008
0x0...2000
0x0...2008
0x0...3000
0x0...3008
0x0...300f
0x0...3fff

0x0...0f
0x0...10
...
## 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>
### 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>

<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?
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 table
  - 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 has 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 are limited to $2^{48}$ bytes which is about 262 Terabytes

**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

CPU Register: CR3

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

Virtual Address: 0x80801ffffa8

<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

0x40000000

Physical address of the 1st entry at level 0

0x3466000
0x3467000
0x3468000
...
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: 0x80801fffa8

CPU Register: CR3

Physical address of the 1st entry at 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

CPU Register: CR3

0x4ffff000

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

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

Virtual Address: 0x80801fffa8

CPU Register: CR3

0x4ffff000

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

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

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

Virtual Address: 0x80801ffff8

Level 0

CPU Register: CR3

0x4ffffff000
Physical address of the 1st entry at level 0

Level 1

Physical address of the 1st entry at level 1

Level 2

Physical address of the 1st entry at level 2

Level 3

Physical address of the 1st entry at level 3

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

Virtual Address: 0x80801fffa8

CPU Register: CR3

0x4ffff000

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

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

Virtual Address: 0x80801fffa8

CPU Register: CR3

0x4ffff000
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
0x3579000
... unused

Level 2
Physical address of the 1st entry at level 3

0x5788000
0x5789000
0x578a000
... unused

Level 3
Physical Page Address

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

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: 0x5799000

Physical Address: 0x5799fa8

4-level page table
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:
Michael’s Question

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

Answer: Multi-level page table enables demand paging
- Construct page table on demand
Demand Paging

Memory Allocation (e.g., \( p = \text{malloc}(1024) \))

Step 1.

Declare a virtual address range from \( p \) to \( p + 1024 \) for current process.

Step 2.

Allocate the physical page and construct the page table on demand.
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

CPU Register: CR3

```
0x4ffff000
```

Level 0

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

current process’ page table
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)*
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*)
Demand Paging

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

CPU Register: CR3

0x4ffff000

Current process’ page table

New pages allocated by OS

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

char *p = (char *)malloc(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 the mapping has been built successfully.
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)
5. CPU asks MMU to translate the address again and access the physical memory.

```c
char *p = (char *)malloc(8192); // p is 0x80801fffa8
p[0] = ‘c’
p[4096] = ‘s’
```
1. MMU tells OS entry 0x200 is missing in the page at level 3. (Page fault)
2. OS constructs the mapping for the address. *(Page fault handler)*
char *p = (char *)malloc(8192);  // p is 0x80801fff8
p[0] = 'c'
p[4096] = 's'  // 0x8080200fa8

3. OS tells the CPU the mapping has been built successfully.
Demand Paging

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

4. CPU ask MMU to translate the address again and access the physical memory.
Questions

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

What is the minimal page table size on 64 bit machine? 4

If a process’ page table size is 4, how many physical pages it can have?
Questions

What is the minimal page table size on 64 bit machine? 4

If a process’ page table size is 4, how many physical pages it can have? $2^9$
Jeffrey’s Question

• Where is the **segmentation fault** from?
  – Step 1. *Declare a virtual address range from p to p + size for current process.*
  – Step 2. *Allocate the physical page and construct the page table on demand.*

1. MMU tells OS that page table entry of that address is missing. *Page fault*
2. OS checks the vaddr has not been declared, tells the application this address is illegal.
Memory Access Cost

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

With pipeline, but no memory access
- Instructions per cycle \( \geq 1 \)
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 page table accesses plus one time data access which is 5 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. 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
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

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 = 2
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. 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

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
Speedup Address Translation

1. Calculate VPN
   a. \( \text{VPN} = \text{VA} \gg 12 \)

2. Check TLB
   a. Index = \( \text{hash(VPN)} \), \( \text{VPN} \mod 4 \)
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      \[ \text{PA} = \text{TLB[Index].PPN} + \text{Offset} \]
     a. On TLB miss
        Go through 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 = 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

Both 0x1234 and 0x5234 go to the entry 1

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

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

Both 0x1234 and 0x5234 go to the entry 1
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 through 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. $\text{VPN} = \text{VA} >> 12$

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

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 hash conflict!
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 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 hash conflict! → Multi-set associative TLB
Multi-set associative TLB

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)
Example

- access 0x11234, TLB Miss
- access 0x21234
- access 0x11234
- access 0x21234

TLB (16 sets, 4 way associative)
Example

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

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

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

TLB (16 sets, 4 way associative)

MMU
Example

access \texttt{0x11234}, TLB Miss, buffer the translation result
access \texttt{0x21234}, TLB Miss, buffer the translation result
access \texttt{0x11234}
access \texttt{0x21234}

TLB (16 sets, 4 way associative)
Example

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

TLB (16 sets, 4 way associative)
Latency

Memory access
  – Hundreds of CPU cycles

TLB access
  – Be parallel with instruction execution
  – Only a couple of CPU cycles
Summary

Virtual Address → TLB Access

Hit? (Yes/No)

No → Walk Though Page Table

Yes → PPN + Offset

Physical Address
Kevin’s Question

Answer: This is the virtual memory space of a single process.