Isolation & Virtual Memory

Jinyang Li
Layered organization

Software

Operating System

User Applications

Hardware

Goal: Isolation -- failure of one running program must not affect another

CPU | Memory | I/O
Isolation is provided by OS using special hardware primitives.
Process

• What is a process?
  – An instance of a running program

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

• OS assigns different processes different process id
  – `getpid()` system call returns id of current process
  – Command `ps` list all processes, `kill` terminates a process
Upon program startup, OS creates a process. Process asks for OS services through system calls (\textit{getpid, read, write}).

Syscalls are listed in manual section 2 type “man 2 getpid”
Our simplified “Mental Model” of program execution

Question: how does a CPU execute >1 programs “simultaneously”? 
Sharing CPU: time multiplexing

1. OS assigns different processes different memory regions
Sharing CPU: time multiplexing

2. OS saves/restores CPU state in memory to switch execution of one process to another every ~10ms
Sharing CPU: time multiplexing

2. OS saves/restores CPU state in memory to switch execution of one process to another every ~10ms
## Sharing CPU: time multiplexing

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC: 0x00...0028</td>
<td>saved CPU state</td>
</tr>
<tr>
<td>IR: mov %rax, (%rbx)</td>
<td></td>
</tr>
<tr>
<td>RAX: 0x2</td>
<td>saved CPU state</td>
</tr>
<tr>
<td>RBX: 0x00..1234</td>
<td>Memory</td>
</tr>
</tbody>
</table>

**Challenge:** how to prevent a process from reading/writing another process’s memory?!
Processes need to share memory safely

- (Simplicity) Processes are loaded at the same addresses
  - e.g. Linker/loader can handle different processes with the same code
- (Isolation) One process cannot access another process’ memory
  - Process X cannot overwrite data in process Y
  - Process X cannot peek sensitive data in process Y

How?
- Virtual Memory
Hardware solution: 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...010
0x0...018
0x0...020
0x0...028
0x0...030
0x0...038
0x0...040
0x0...048
0x0...050
0x0...058

data: 0x1
Address Translation – Strawman

Provide MMU with 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

Question: What is the size of mapping table?
For 64-bit address, size is $2^{64}$
Address Translation – Page

Problem: Mapping table too big
Solution: map at a coarser granularity
  – Divide memory into fixed-size pages.
  – Page table: map virtual pages to physical pages.
Paging example: 4-bit address

- 4-bit virtual and physical addresses
- 4-byte page size

Question: for 32-bit address, 4KB page size, how many pages?

Answer: \( \frac{2^{32}}{2^{12}} = 2^{20} \) pages
Paging example: 4-bit address

- Not all virtual pages are used
  - On 64-bit machine, vast majority of virtual pages are unused

<table>
<thead>
<tr>
<th>Process 1’s address space</th>
<th>Process 2’s address space</th>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>1100</td>
</tr>
</tbody>
</table>

Diagram showing the address spaces and physical memory for process 1, process 2, and a section of physical memory.
Paging example: 4-bit address

- What does the mapping table look like?

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1100</td>
</tr>
<tr>
<td>0100</td>
<td>--</td>
</tr>
<tr>
<td>1000</td>
<td>--</td>
</tr>
<tr>
<td>1100</td>
<td>0100</td>
</tr>
</tbody>
</table>

Can we eliminate this column?
Paging example: 4-bit address

- What does the mapping table look like?
  - An array of integers (Page Table Entry, PTE)

Index of array is virtual page number

PTE is physical page number

For $2^x$ page size, PTE’s least significant $x$ bits can be used for other purposes e.g. indicate PTE validity

Process 2’s address space

Physical memory
Paging example: 4-bit address

- How to translate from virtual to physical address?

```
<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>??</td>
</tr>
</tbody>
</table>

CPU ➔ MMU ➔ Physical Memory

Memory Management Unit

1101
0000
0000
0101

page table of p2

physical memory

```
Paging example: 4-bit address

• How to translate from virtual to physical address?
Paging example: 4-bit address

- How to translate from virtual to physical address?

![Diagram of paging example: 4-bit address with CPU, Memory Management Unit (MMU), virtual address, physical address, page table of p2, and physical memory.]
Each process has its own page table

OS switches CPU to execute p1 instead of p2
Recap: Linux address space

What is this address?
Address Translation: 32-bit address

- Page size: 4 KB ($2^{12}$)
- How many virtual pages?
- How many bits for virtual page number (VPN)?
- How many bits for page offset?
Address Translation: 32-bit address

• Page size: 4 KB ($2^{12}$)
• What’s the size of each PTE?

PTE: 

- Physical Page #
- Present?
- Accessible by OS only?
- Writable?

4-byte PTE
Address Translation: 32-bit address

- Page size: 4 KB ($2^{12}$)
- How many PTEs in the page table? $2^{32}/2^{12} = 2^{20}$
- What’s the size of page table? $2^{20} \times 2 = 4$ MB
Address Translation: 64-bit address

- Page size: 4 KB ($2^{12}$)
- How many virtual pages?
- How many bits for virtual page number (VPN)?
- How many bits for page offset?

![Diagram of 64-bit virtual address]

- Virtual page number (VPN)
- Page offset (VPO)

64-bit virtual address
Address Translation: 64-bit address

- Page size: 4 KB ($2^{12}$)
- What’s the size of each PTE?
- How many PTEs in the page table?

$$2^{64}/2^{12} = 2^{52}$$

4 Petabyte!!

Possible solution: Enlarge page size? e.g. 256MB ($2^{28}$) page size
This lecture

• Multi-level page tables
• Demand paging
• Accelerating address translation
Multi-level page tables

Problem with 1-level page table:
- For 64-bit address space and 4KB page size, page table is much too large

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

- number of bytes addressable in 64-bit address space
- \# of pages in 64-bit address space
- = \# of page table entries required
- page size
Multi-level page tables

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

Solution
• Multi-level page table
  – A tree of “page tables”
2-level Paging Example

- 6-bit virtual and physical address, 4-byte page
- 2-level page table
2-level Paging Example

- 6-bit virtual and physical address, 4-byte page
- 2-level page table

Suppose a process has only two 2 mapped virtual pages

```
2^4 PTEs
1-level Page Table

2^2 L0 PTEs

2^2 L1 PTEs
```

010000 is physical address of L1 page table
2-level Paging Example

- how to perform address translation?

Index to
page table

Index to
L0 table

Index to
L1 table

Page
offset

Page
offset

1-level Page Table

11 11 10

11 11 10

110001
000000
...
000000
010001

011001
000000
000000
110101

110001
000000
000000
010000

000000
000000
000000
010001
X86_64 supports 4-level page table

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

<table>
<thead>
<tr>
<th>Level</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>L0 Offset</td>
</tr>
<tr>
<td>2</td>
<td>L1 Offset</td>
</tr>
<tr>
<td>3</td>
<td>L2 Offset</td>
</tr>
<tr>
<td>4</td>
<td>L3 Offset</td>
</tr>
<tr>
<td></td>
<td>Page Offset</td>
</tr>
</tbody>
</table>

CPU Register: CR3

- **Root Addr**

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

4-level page table

CPU Register: CR3

Root Addr

Level 0

Level 1

Level 2

Level 3
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

Physical address of the 1st entry at level 0

0x4ffff000

Physical address of the 1st entry at level 1

0x3466001
0x3467001
0x3468001
...
unused

Level 0

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

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

Virtual Address: 0x80801fffba8

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: 0x80801fff8

CPU Register: CR3

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

Virtual Address: 0x80801fff8

CPU Register: CR3

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

Virtual Address: \(0x80801ffffa8\)

### 4-level page table

<table>
<thead>
<tr>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3466001</td>
<td>...</td>
<td>unused</td>
<td>0x5788001</td>
</tr>
<tr>
<td>0x3467001</td>
<td>...</td>
<td>unused</td>
<td>0x5789001</td>
</tr>
<tr>
<td>0x3468001</td>
<td>...</td>
<td>unused</td>
<td>0x578a001</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>unused</td>
<td>...</td>
<td>unused</td>
<td>0x5799001</td>
</tr>
</tbody>
</table>

**CPU Register: CR3**

- Physical address of the 1st entry at level 0: 0x4ffff000
- Physical address of the page: 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 Address Space for Process 2:
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

```c
0x4ffff000
```

```c
0x3466001
unused
unused
...
unused
```

**Level 0**

**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)

![Diagram showing demand paging and CPU register CR3]
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)*
1. \( \text{Page fault} \)
2. OS constructs the mapping for the address. \( \text{(Page fault handler)} \)
3. OS constructs the mapping for the address. \( \text{(Page fault handler)} \)

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

**Current process’ page table**

**Level 0**
- Unused
- 0x3668001
- Unused
- Unused
- ... unused

**Level 1**
- Unused
- 0x3588001
- Unused
- ... unused

**Level 2**
- Unused
- 0x3678001
- Unused
- Unused
- ... unused

**Level 3**
- Unused
- 0x5799001
- Unused
- Unused
- ... unused

**Physical Page**

**CPU Register: CR3**
- 0x4ffff000

**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)
5. MMU translates address again and access the physical memory.

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

![Diagram of demand paging and page tables]
1. MMU tells OS entry 0x200 is missing in the page at level 3. (Page fault)

```c
char *p = (char *)sbrk(8192); // p is 0x80801fff4a8
p[0] = 'c'
p[4096] = 's' //0x8080200fa8
```
Demand Paging

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

2. OS constructs the mapping for the address. *(Page fault handler)*
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = ‘c’
p[4096] = ‘s’ //0x8080200fa8
4. MMU translates the address again and access the physical memory.

```
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?

\[ \frac{2^{12}}{2^3} \text{ page size} \]

\[ \frac{2^{12}}{2^3} \text{ size of each page table entry} \]
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.
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 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 + Offset
   a. On TLB miss
      Go though 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
Translation lookaside buffer (TLB)
- Small cache in MMU
- Maps virtual page numbers to physical page numbers

Example:
1. 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

CPU

MMU

<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>

TLB (4 Entries)

Virtual Address

Level 0
Level 1
Level 2
Level 3

VA: 0x2234

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

Example:
1. VPN = VA >> 12
   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? (Yes/No)

Walk Though Page Table → PPN + Offset

Physical Address
More on TLBs
### Speedup Address Translation

1. Calculate VPN
   - VPN = VA >> 12
2. Check TLB
   - Index = VPN % 4
   - Check if TLB[Index].VPN == VPN
   - On TLB hit,
     - PA = TLB[Index].PPN + Offset
   - 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[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
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
Speedup Address Translation

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

2. Check TLB
   a. Index = $\text{VPN} \% 4$
   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 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
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!
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

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

TLB (16 sets, 4 way associative)
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

TLB (16 sets, 4 way associative)

Set 0
- Tag: 0x2
- PPN: 0x6

Set 1
- Tag: 0x1
- PPN: 0x4

Set 15

…

MMU
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