CSCI-UA.0201

Computer Systems Organization

Memory Management – Virtual Memory

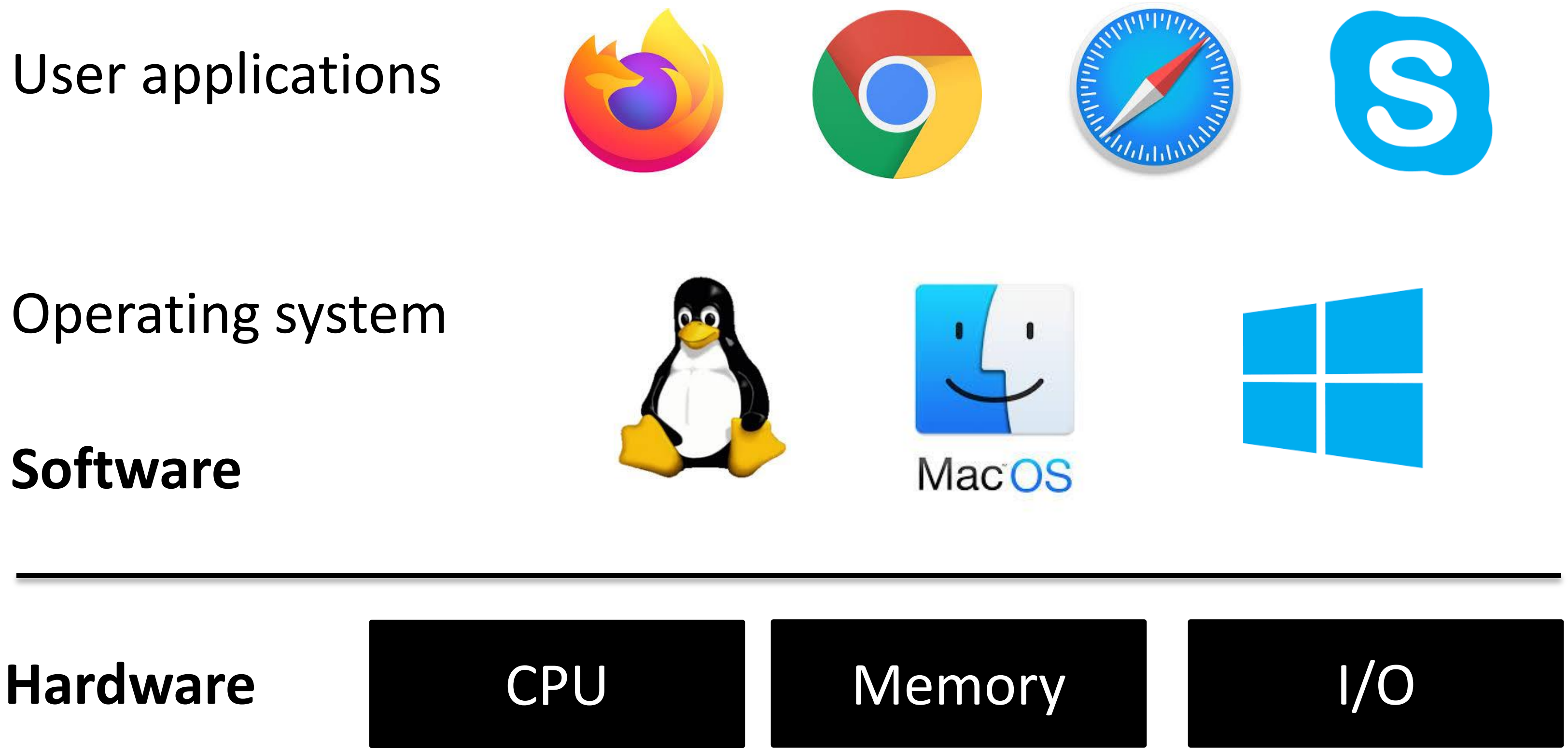
Thomas Wies

wies@cs.nyu.edu

https://cs.nyu.edu/wies

Virtual Memory and Isolation

Isolation



User applications

Operating system

Software

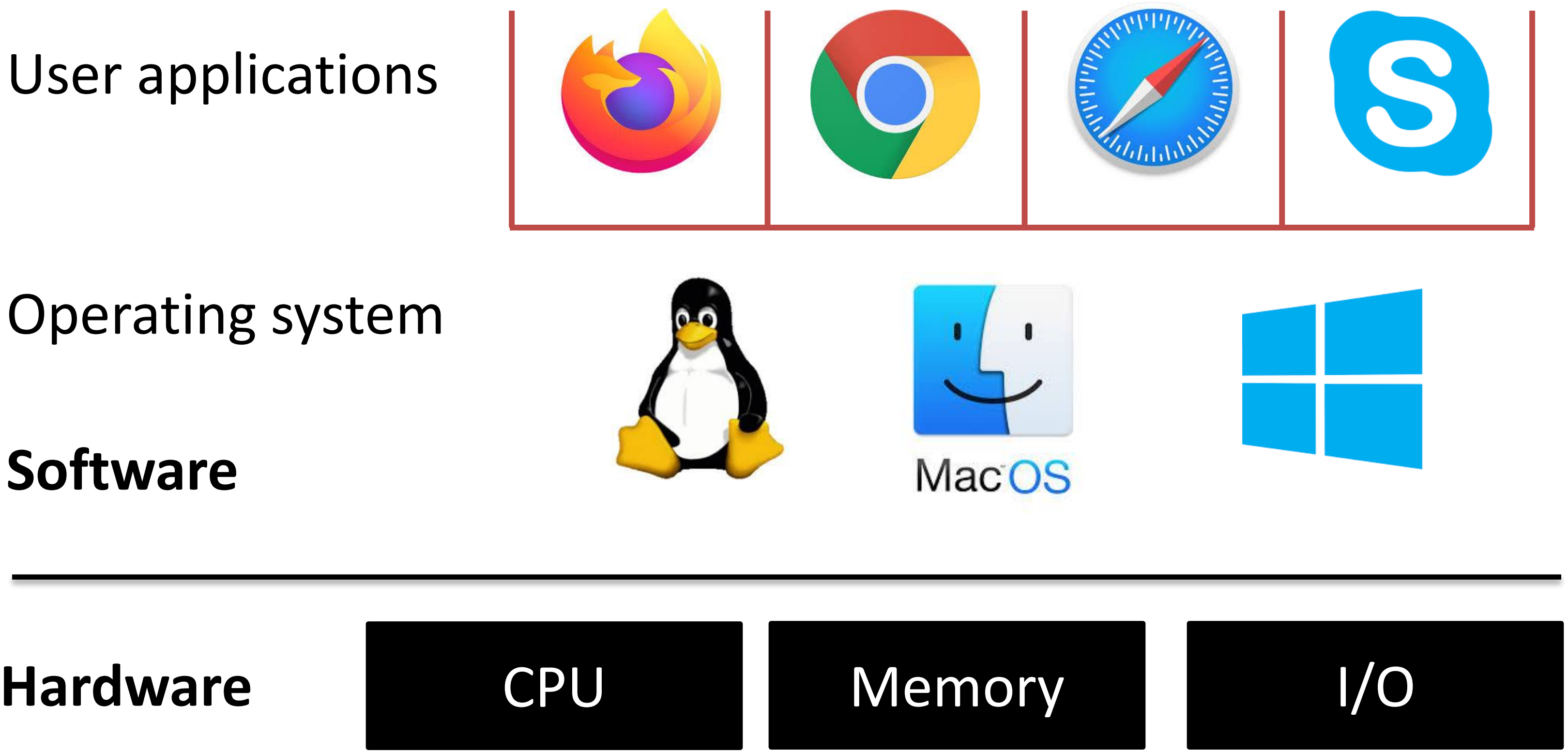
Hardware

CPU

Memory

I/O

Isolation



User applications

Operating system

Software

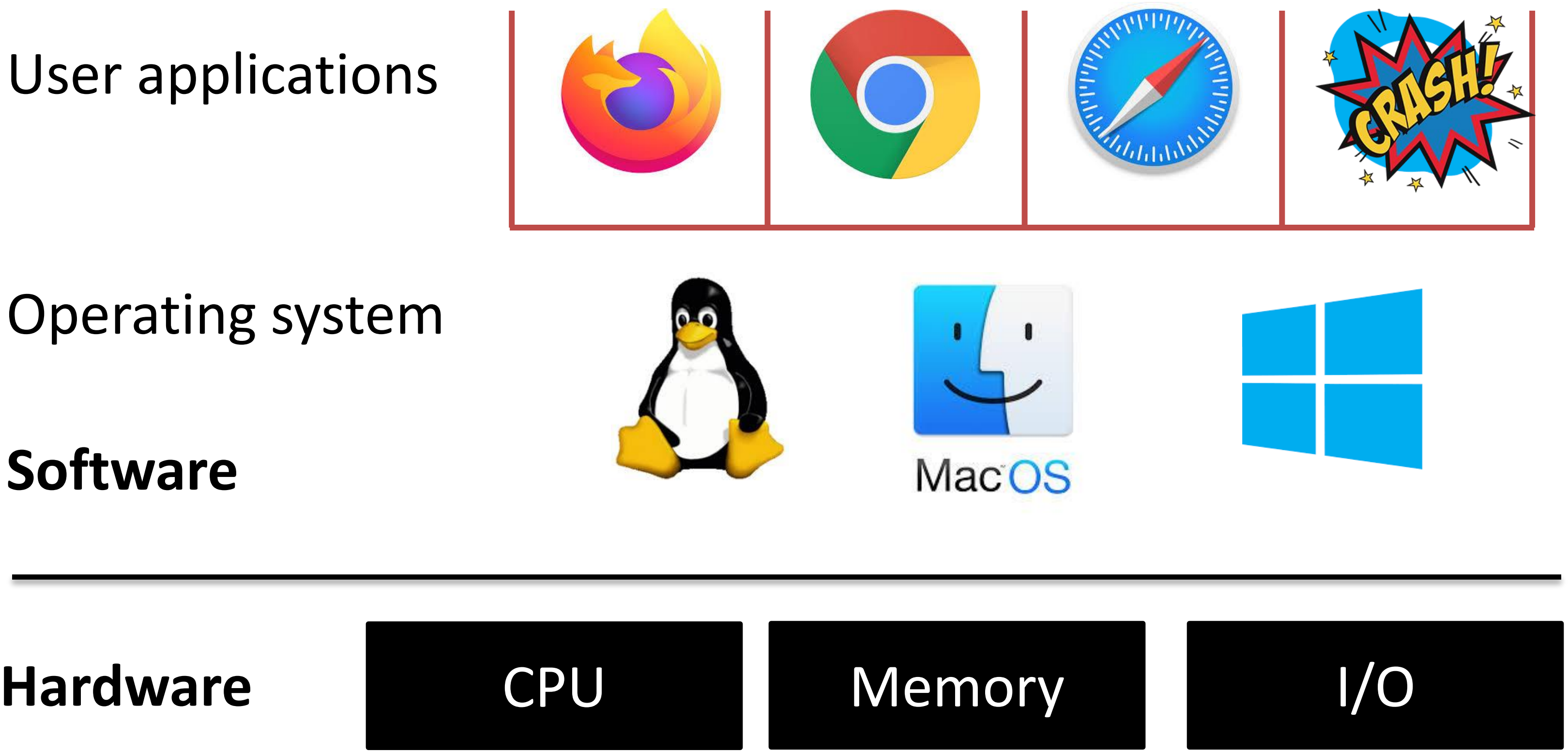
Hardware

CPU

Memory

I/O

Isolation



User applications

Operating system

Software

Hardware

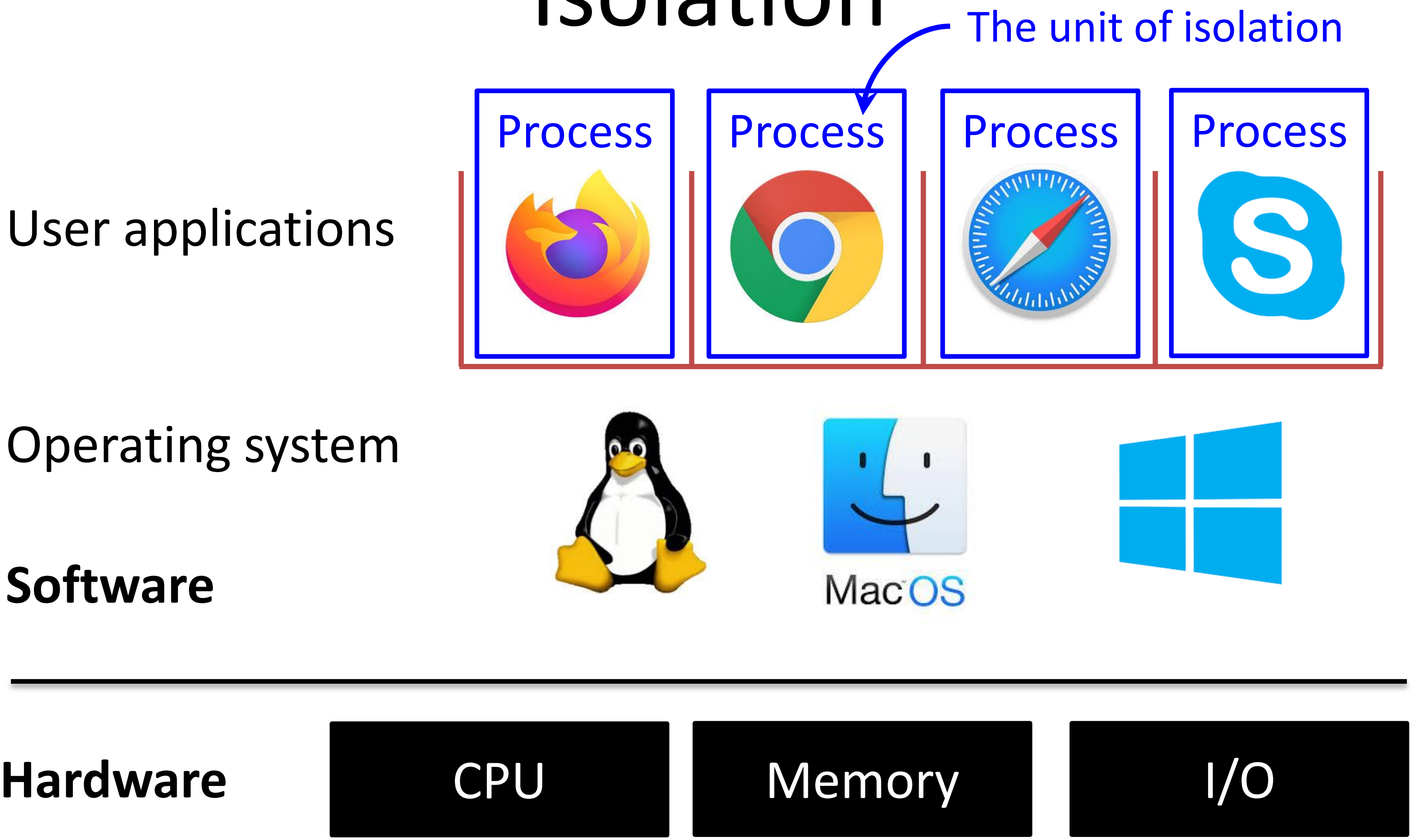
CPU

Memory

I/O

Isolation – Enforced separation to contain effects of failures

Isolation



The unit of isolation

Process

Process

Process

Process

User applications

Operating system

Software

Hardware

CPU

Memory

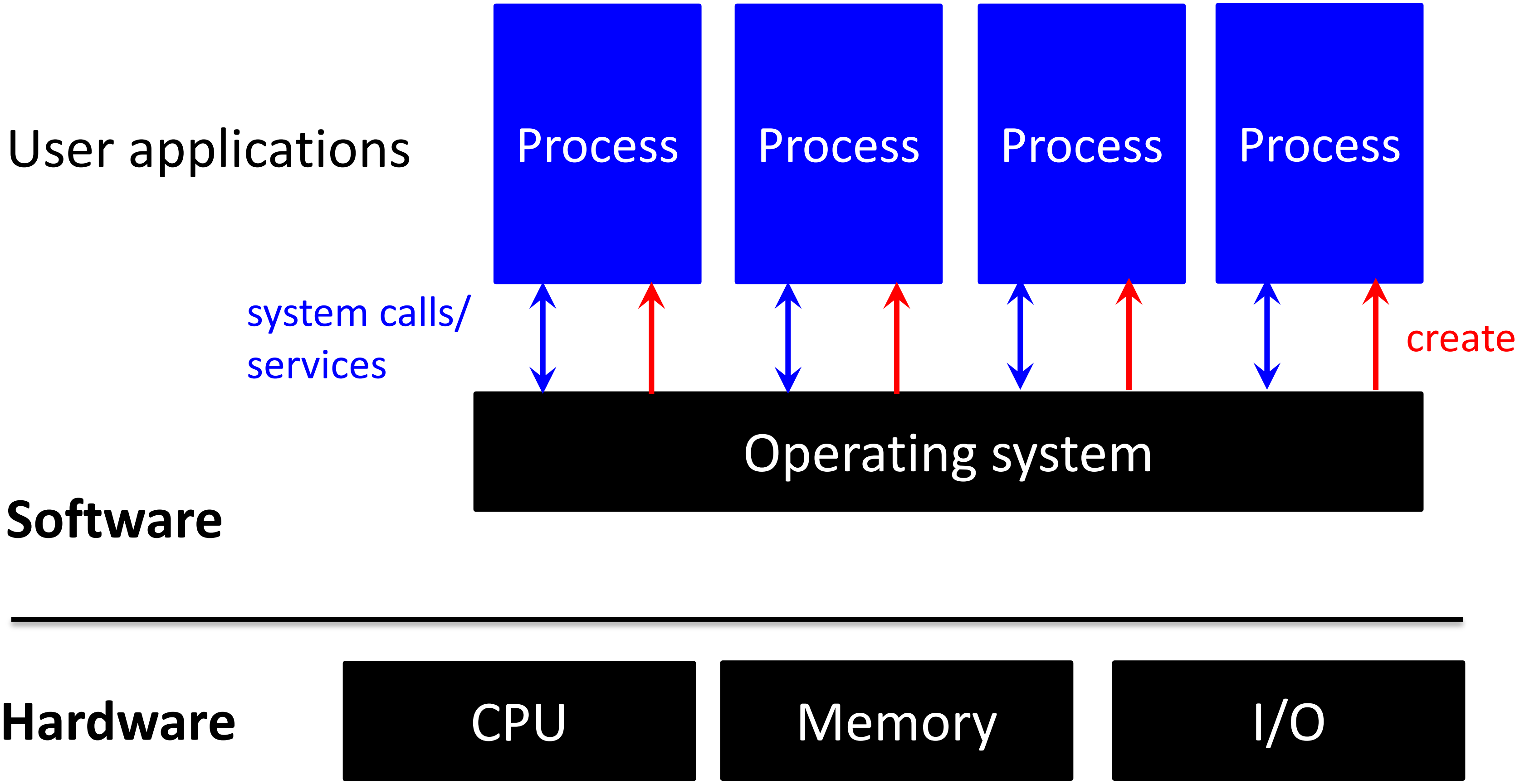
I/O

Isolation – Enforced separation to contain effects of failures

Process

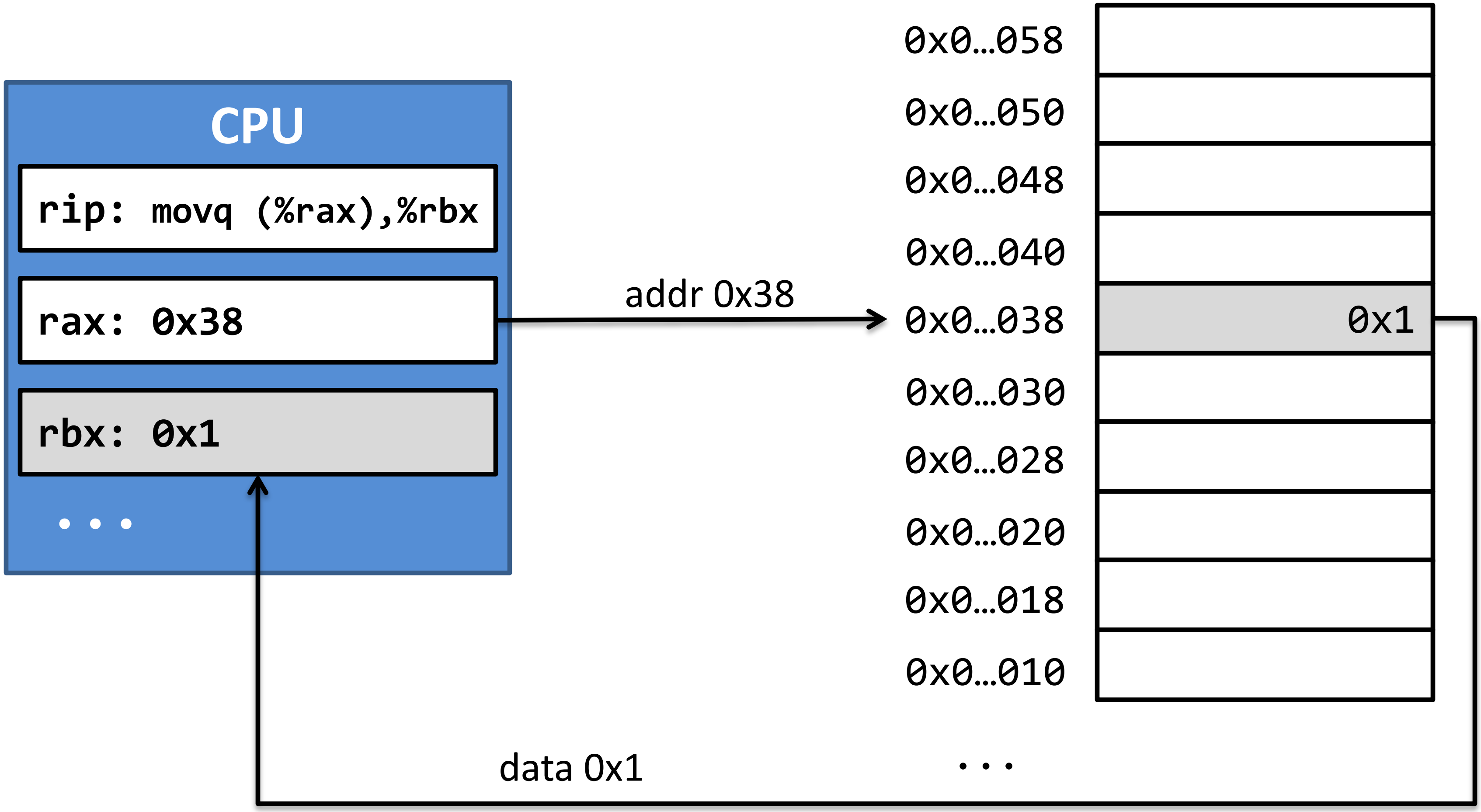
- An instance of a computer program that is being executed
- Program vs. Process
  - Program: a passive collection of instructions
  - Process: the actual execution of those instructions
- Different processes have different process id
  - `getpid()`: function that returns id of current process
  - Command `ps`: list all processes

Isolation



To run a program, OS starts a process and provide services through system calls (`getpid()`, `fopen()`).

Our "Mental Model" of Memory

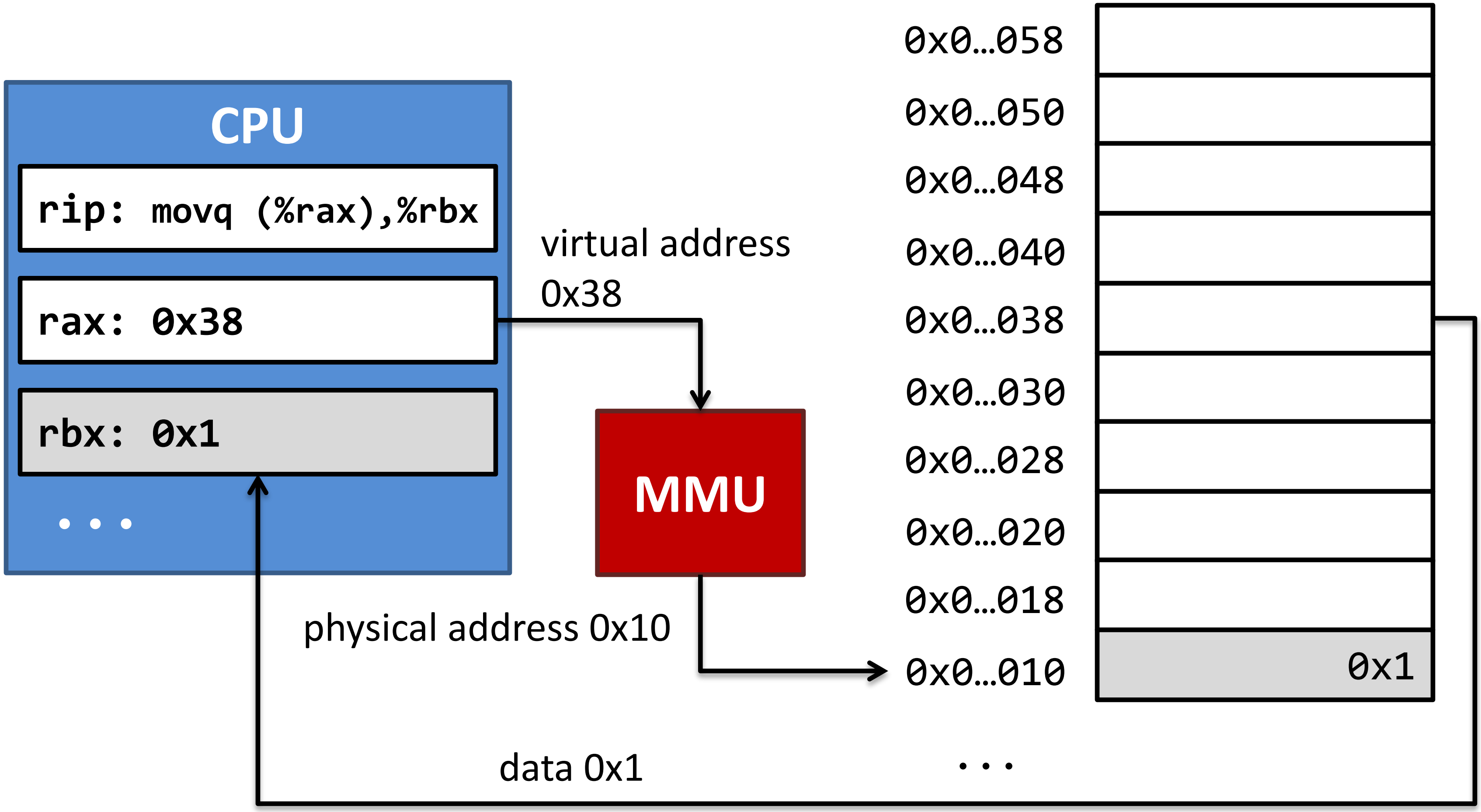


Processes Share the Same Physical Address Space

- The requirements:
  - Different processes use the same address to store their local code/data.
  - One process can not 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. (etc. linking or loading)
- How
  - **Virtual Memory**

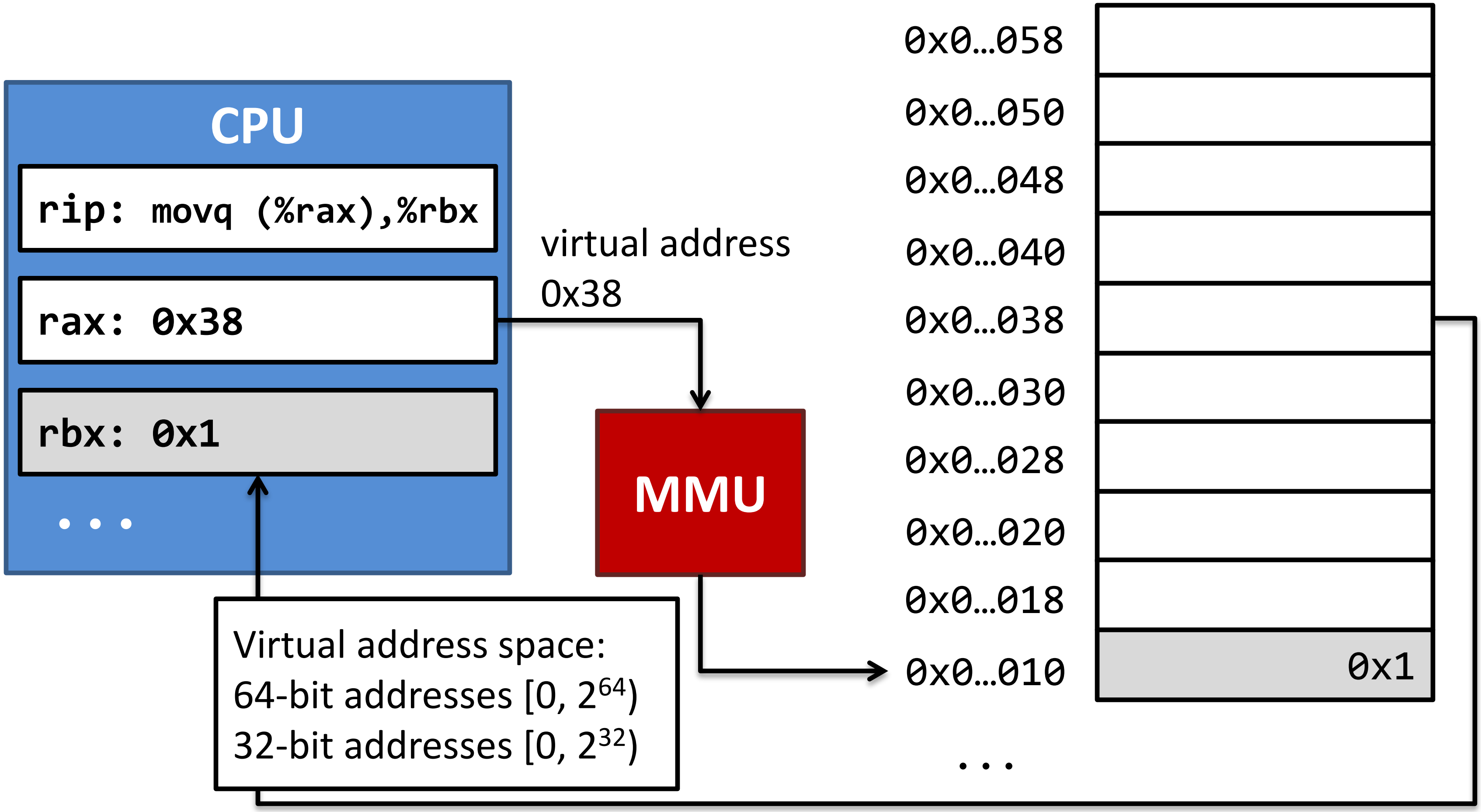
Real System – Virtual Addressing

Memory Management Unit (MMU)



Real System – Virtual Addressing

Memory Management Unit (MMU)



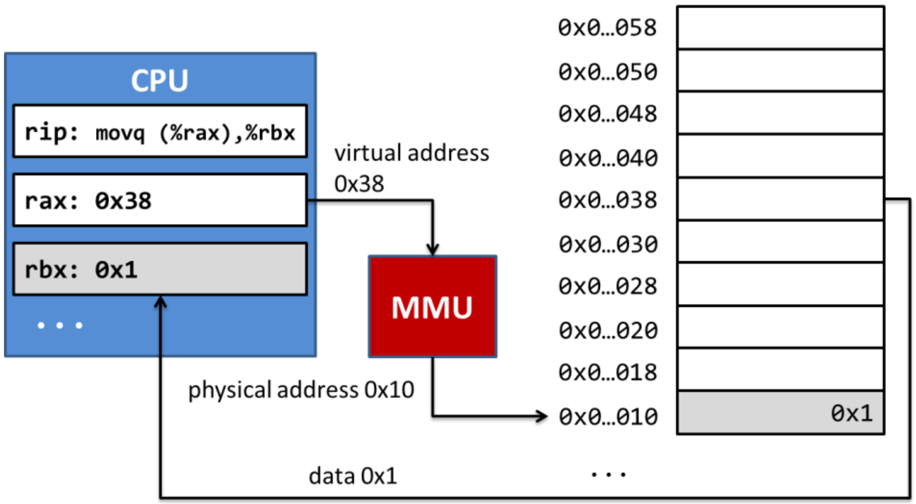
Virtual address space:
64-bit addresses [0, 2^{64})
32-bit addresses [0, 2^{32})

Address Translation – Strawman

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

MMU

Virtual address	Physical address
…	
0x39	0x11
0x38	0x10
…	



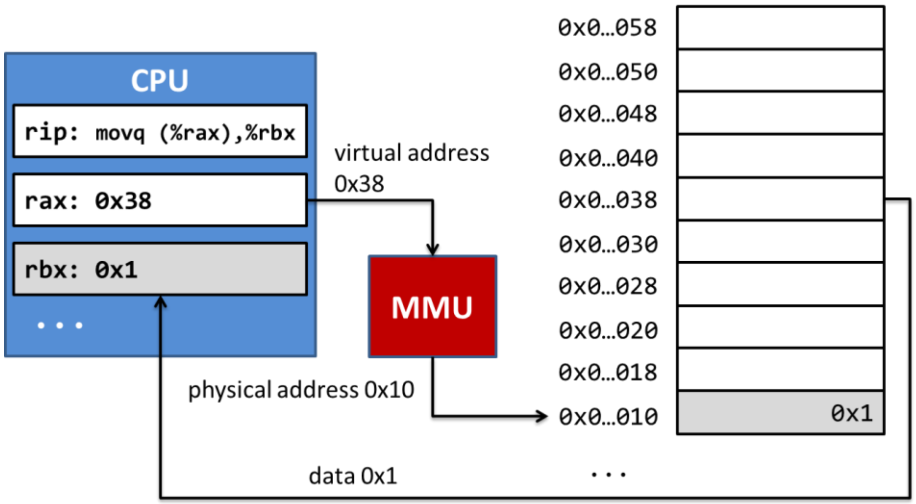
Address Translation – Strawman

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

MMU

Virtual address	Physical address
…	
0x39	0x11
0x38	0x10
…	

What is the size of mapping table?



Address Translation – Strawman

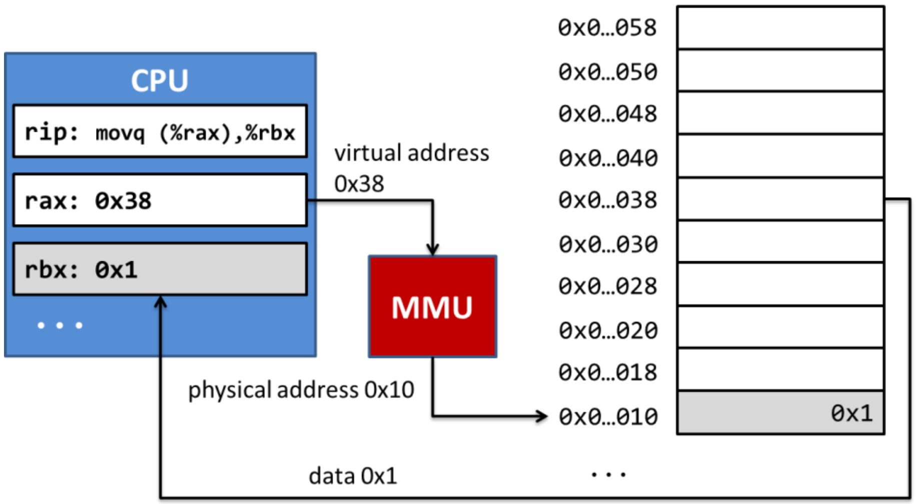
- MMU has a mapping table at byte granularity
  - Map each virtual address into a physical address

MMU

Virtual address	Physical address
…	
0x39	0x11
0x38	0x10
…	

What is the size of mapping table?

Size of virtual address space 2^{64}

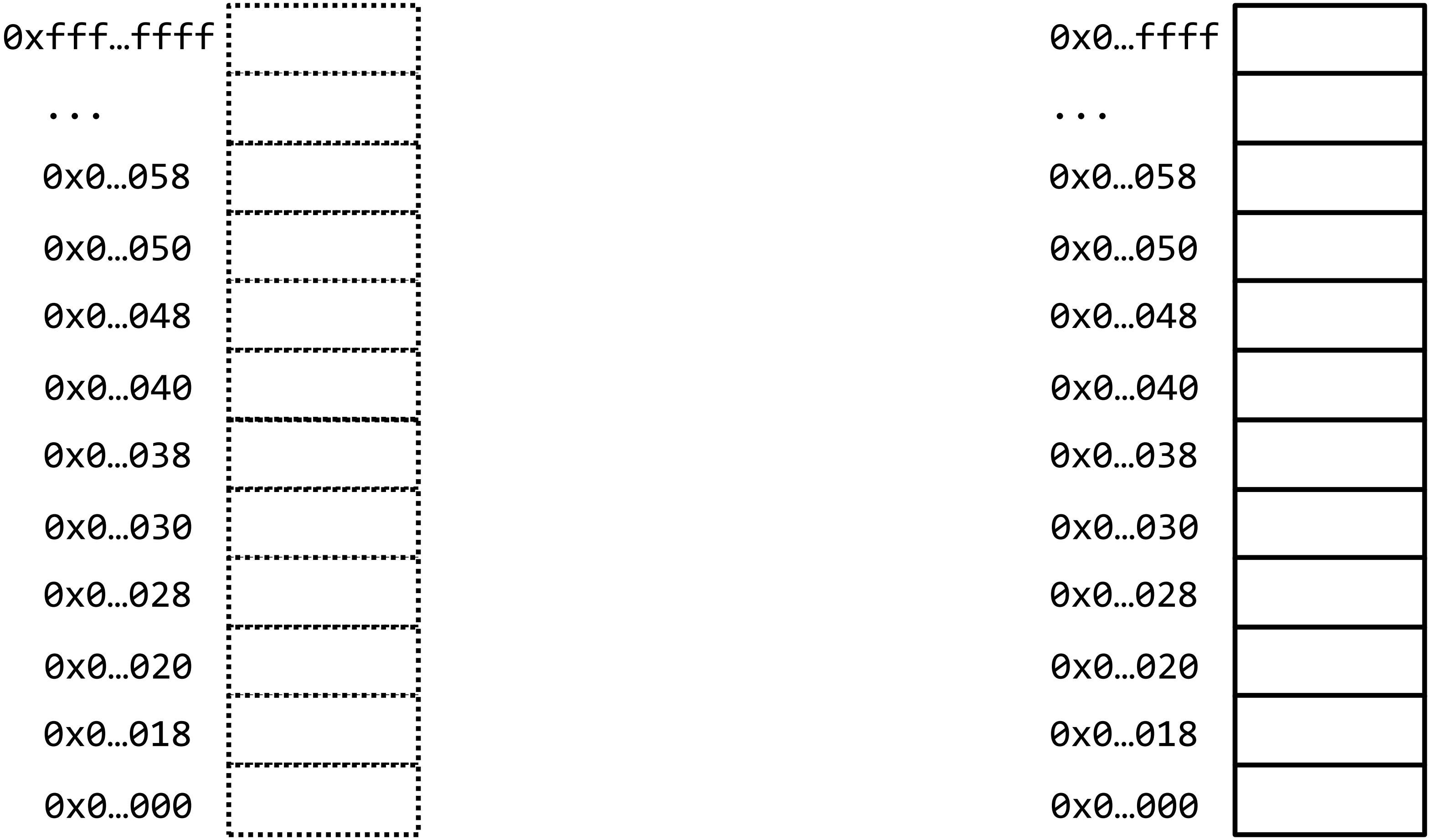


Address Translation – Page

- Observation
  - Both virtual memory space and physical memory space are contiguous

- Build the mapping at coarse granularity
  - split the virtual/physical memory space into contiguous blocks of the same size
  - blocks are called **pages**
  - **page table** maps the virtual pages to physical pages

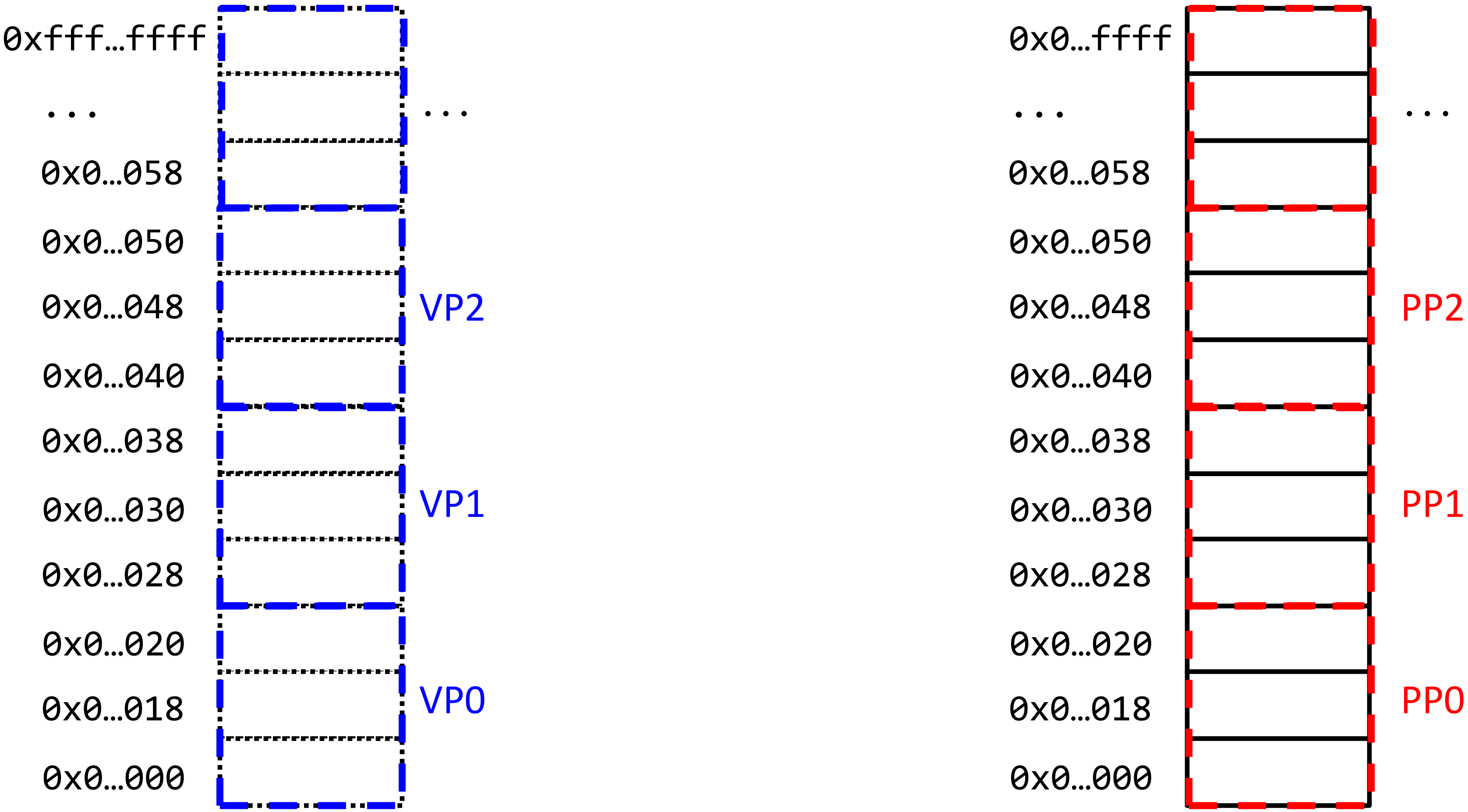
Address Translation – Page-based



Virtual Memory Space
(conceptual memory space)

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

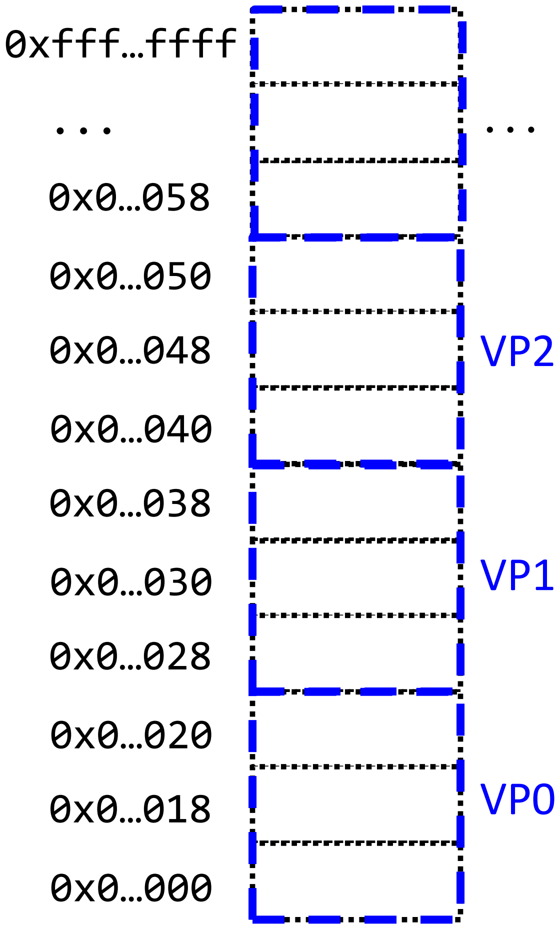
Address Translation – Page-based



Virtual Memory Space
(conceptual memory space)

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

Address Translation – Page-based

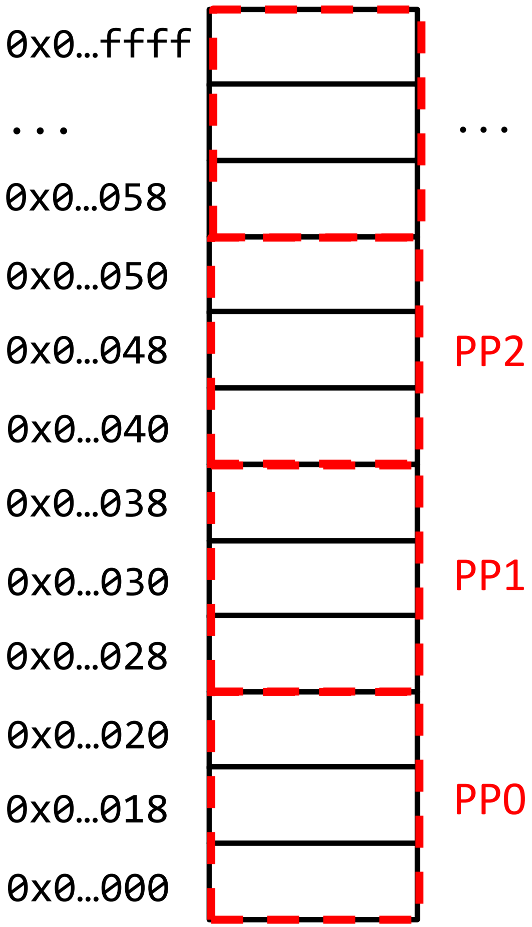


Virtual Memory Space
(conceptual memory space)

MMU

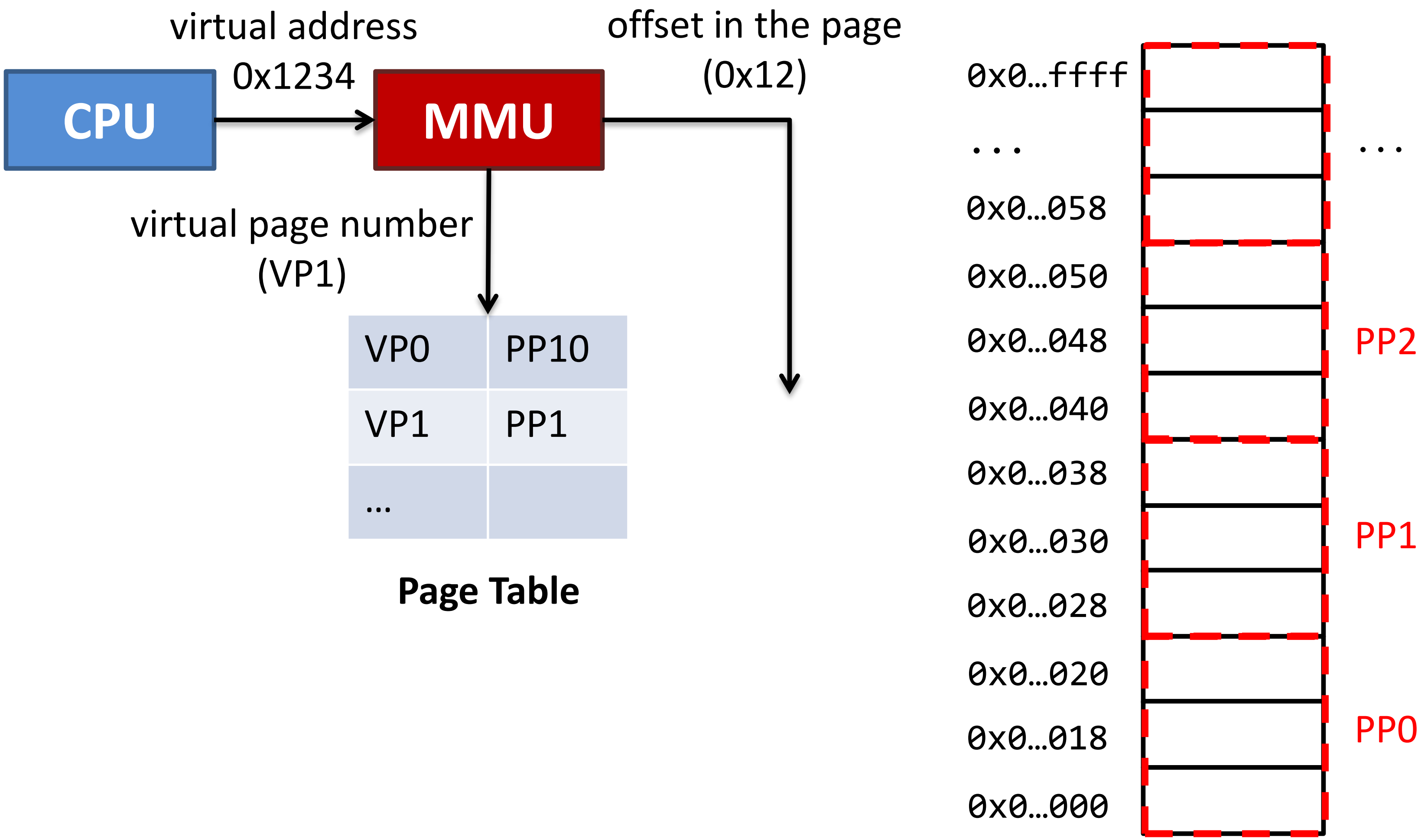
Virtual page # (VPN)	**Physical page # (PPN)**
VP0	PP10
VP1	PP0
…	

Page Table



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

Address Translation – Page-based



CPU

virtual address
0x1234

MMU

offset in the page
(0x12)

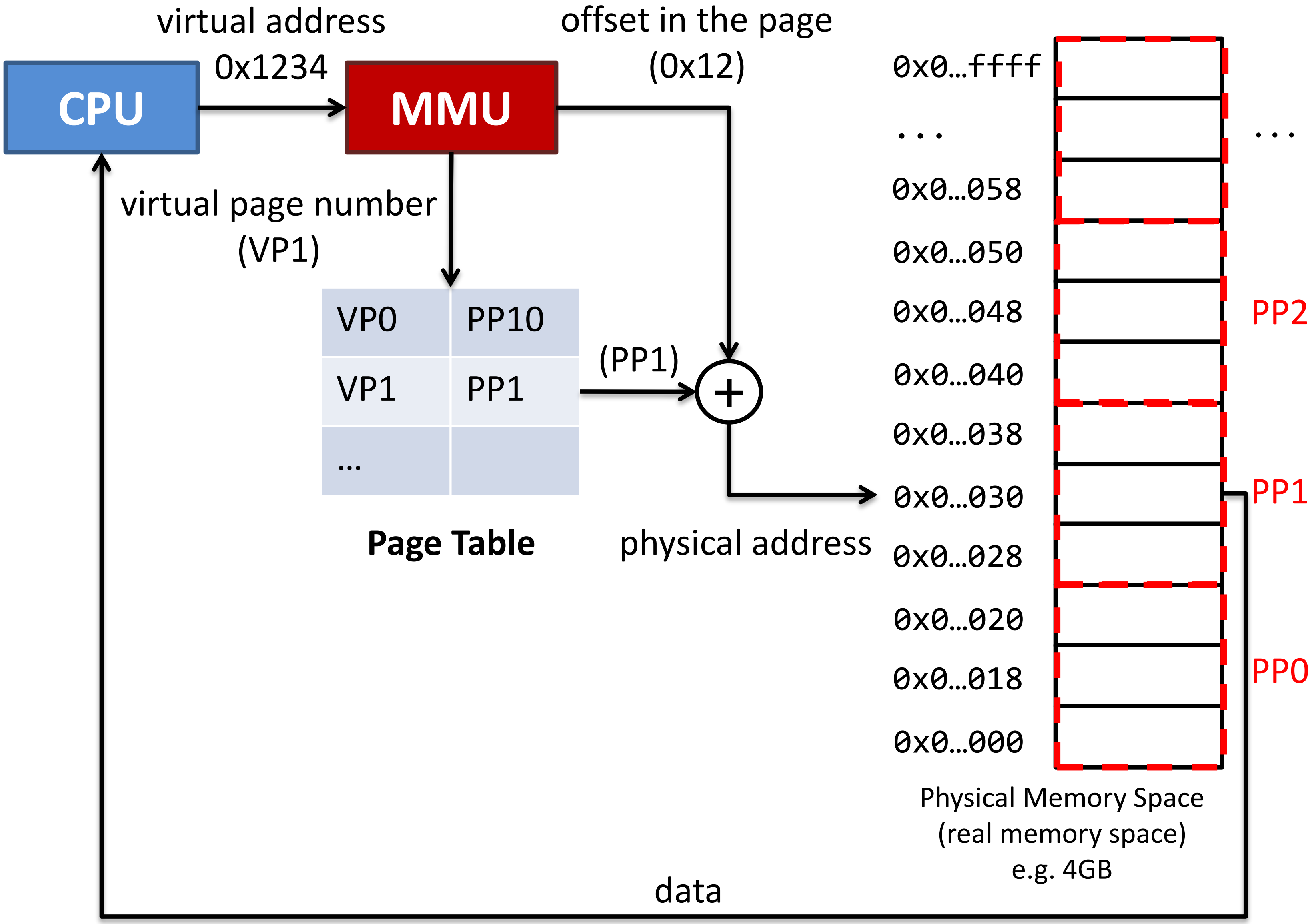
virtual page number
(VP1)

VP0	PP10
VP1	PP1
…	

Page Table

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

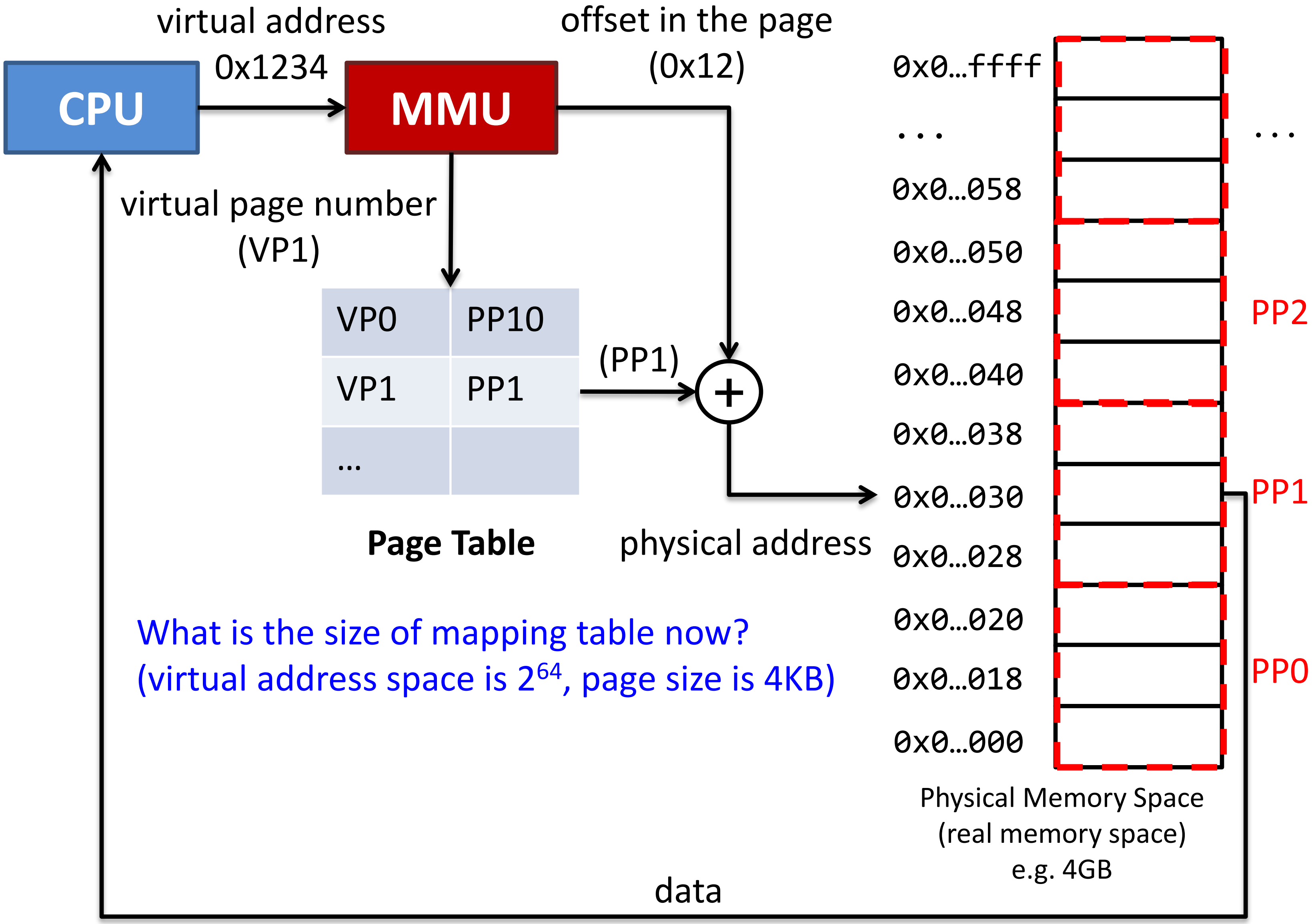
Address Translation – Page-based



VP0	PP10
VP1	PP1
…	

Page Table

Address Translation – Page-based

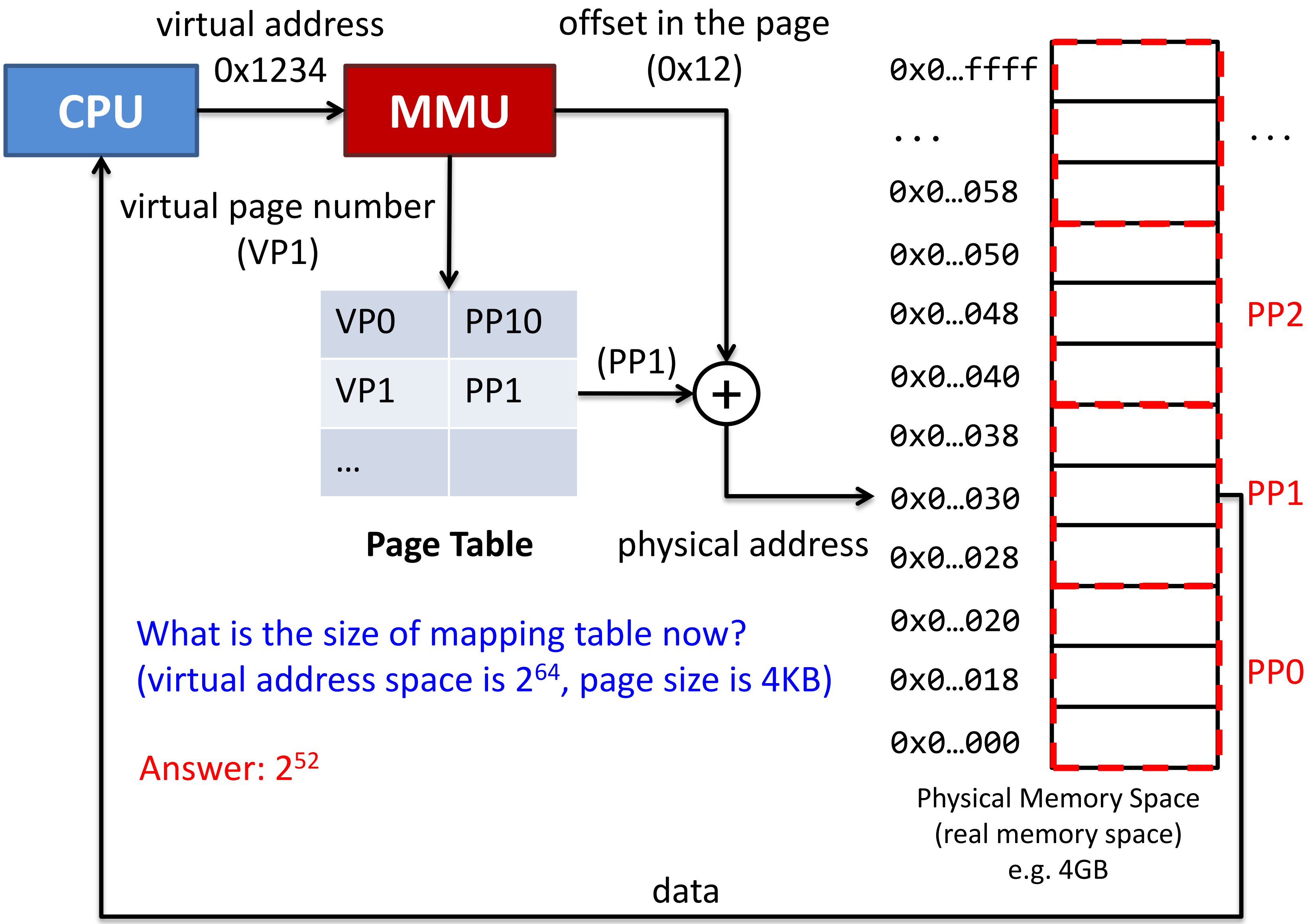


VP0	PP10
VP1	PP1
…	

Page Table

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

Address Translation – Page-based



VP0	PP10
VP1	PP1
…	

Page Table

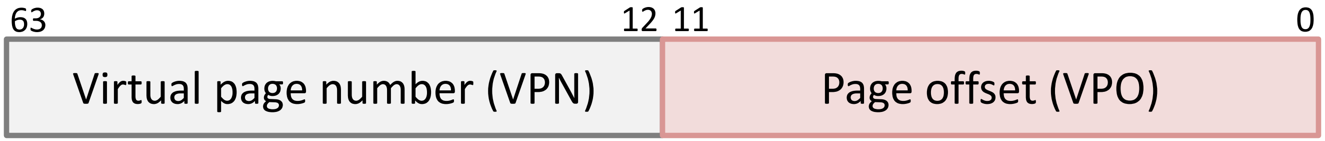
What is the size of mapping table now?
(virtual address space is 2^{64}, page size is 4KB)

Answer: 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})

Virtual address layout



63

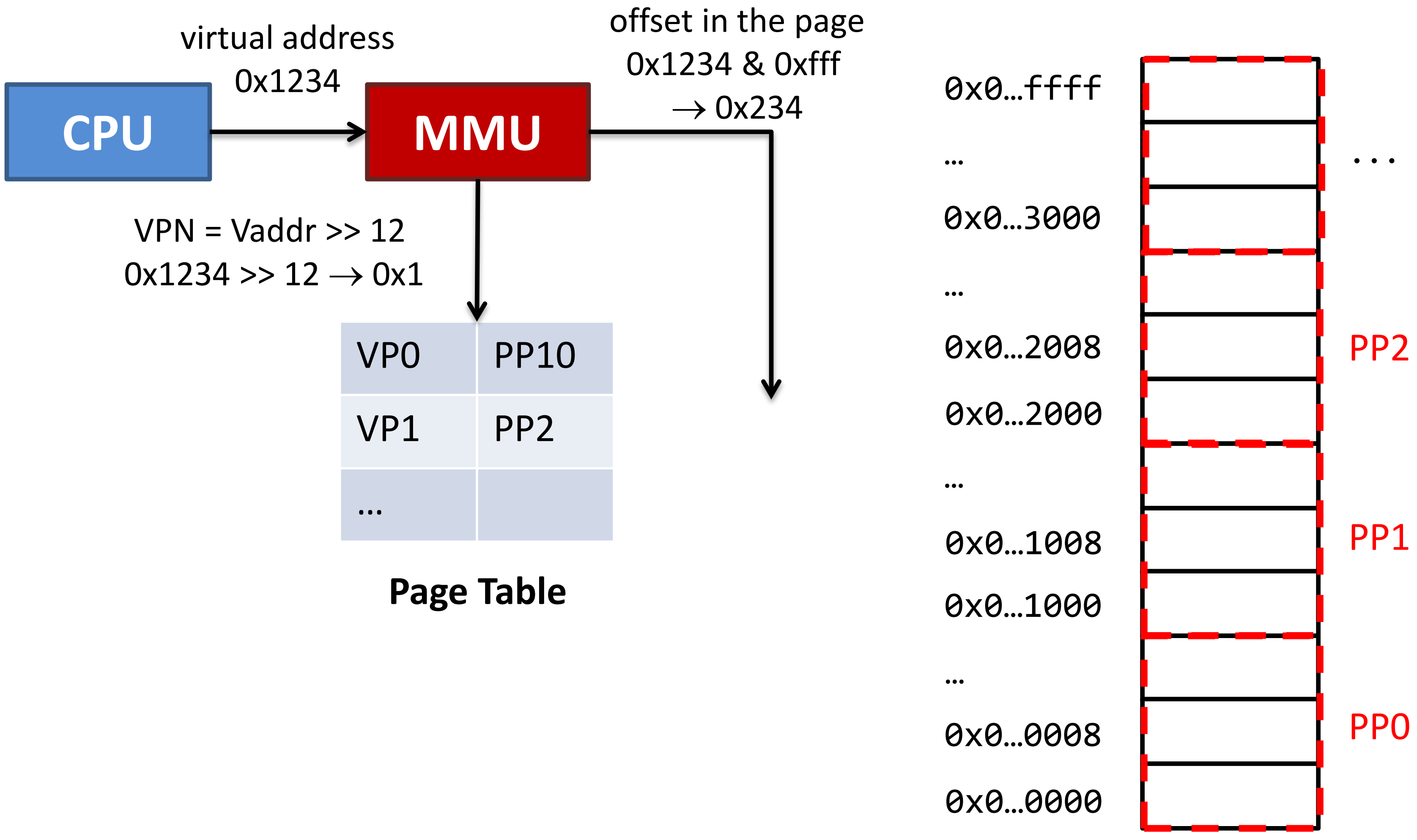
12 11

0

Virtual page number (VPN)

Page offset (VPO)

Address Translation – Page-based



virtual address
0x1234

offset in the page
0x1234 & 0xfff
→ 0x234

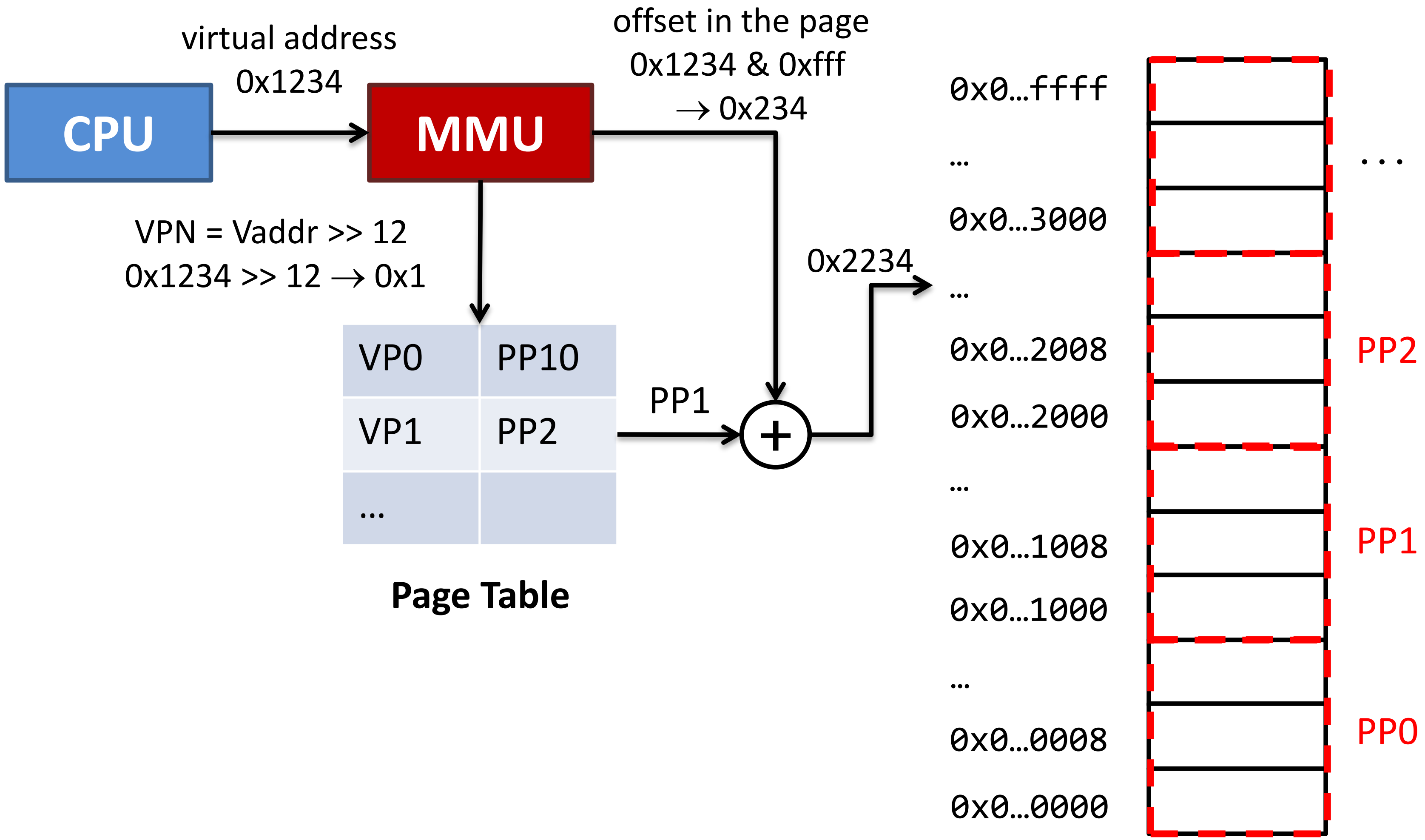
VPN = Vaddr >> 12
0x1234 >> 12 → 0x1

VP0	PP10
VP1	PP2
…	

Page Table

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

Address Translation – Page-based



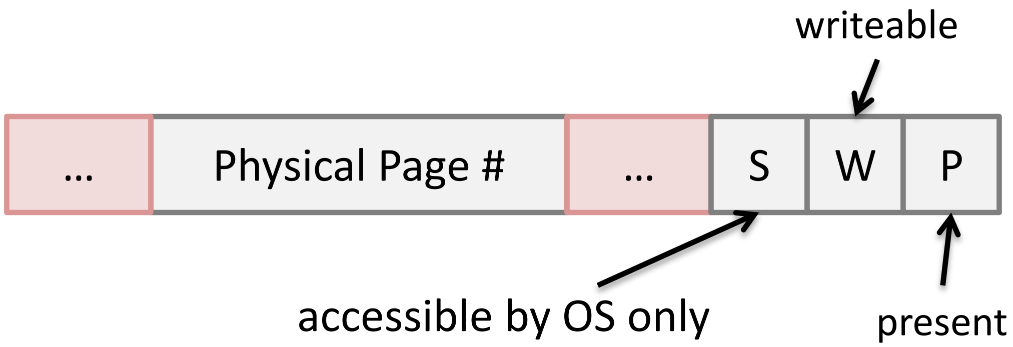
VP0	PP10
VP1	PP2
…	

Page Table

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

Page table entries encode permission information

PTE format



VP0	PP1
VP1	PP3
VP2	PP4
VP4	PP0
VP5	PP3
VP6	PP2

Conceptual Page Table

P[0]	8 byte page table entry (PTE)
P[1]	…
P[3]	
P[4]	
P[5]	

Actual Page Table

How many PTEs per page?

Advanced Topics

- 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, what is the number of page table entries required for translation?

# of bytes addressable in 64-bit address space

$$\frac{2^{64}}{2^{12}} = 2^{52} \longleftarrow$$

# of pages in 64-bit address space
= # of page table entries required

page size

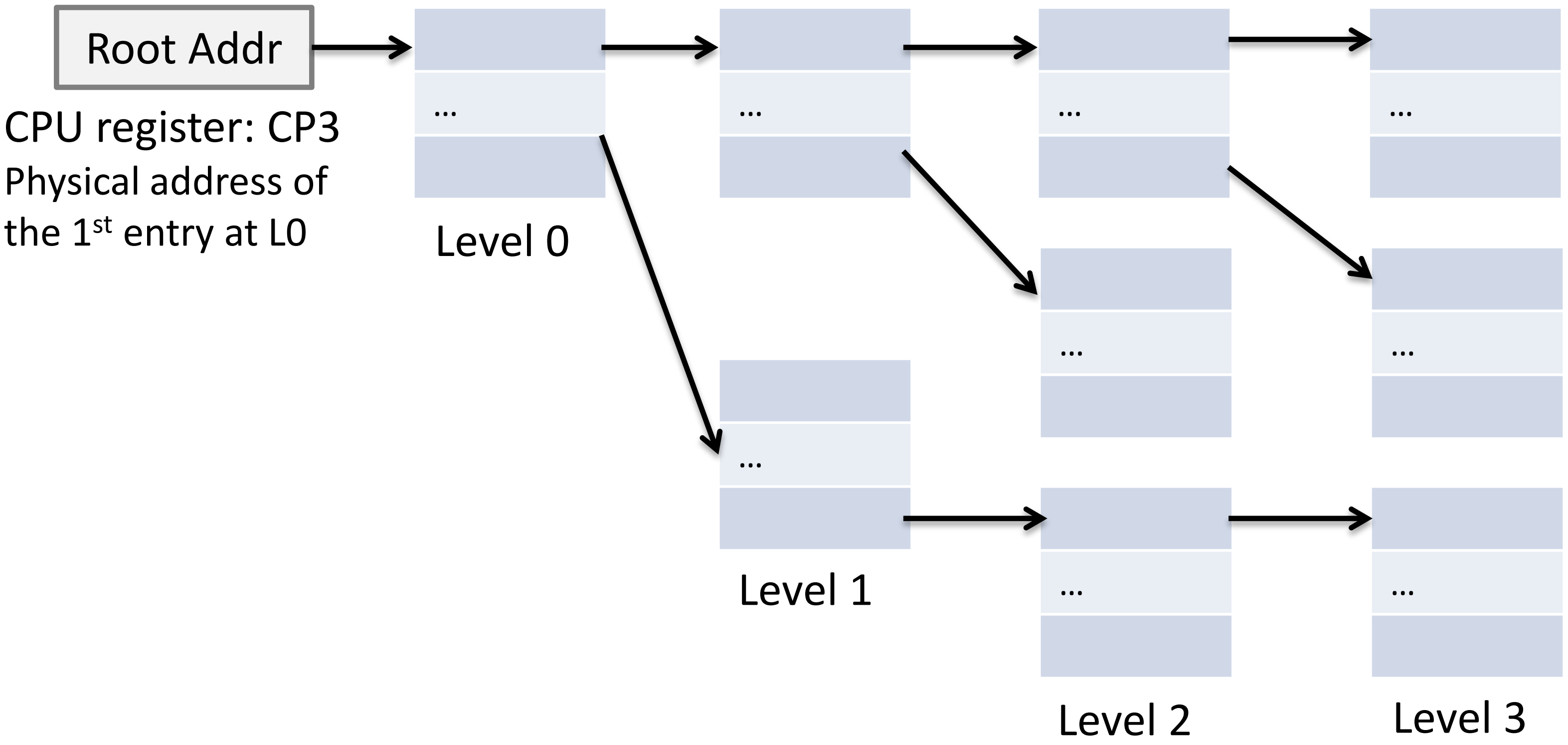
Multi-level Page Tables

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

- Solution
  - multi-level page table
  - x86-64 supports 4-level page table

Multi-level Page Tables on x86_64

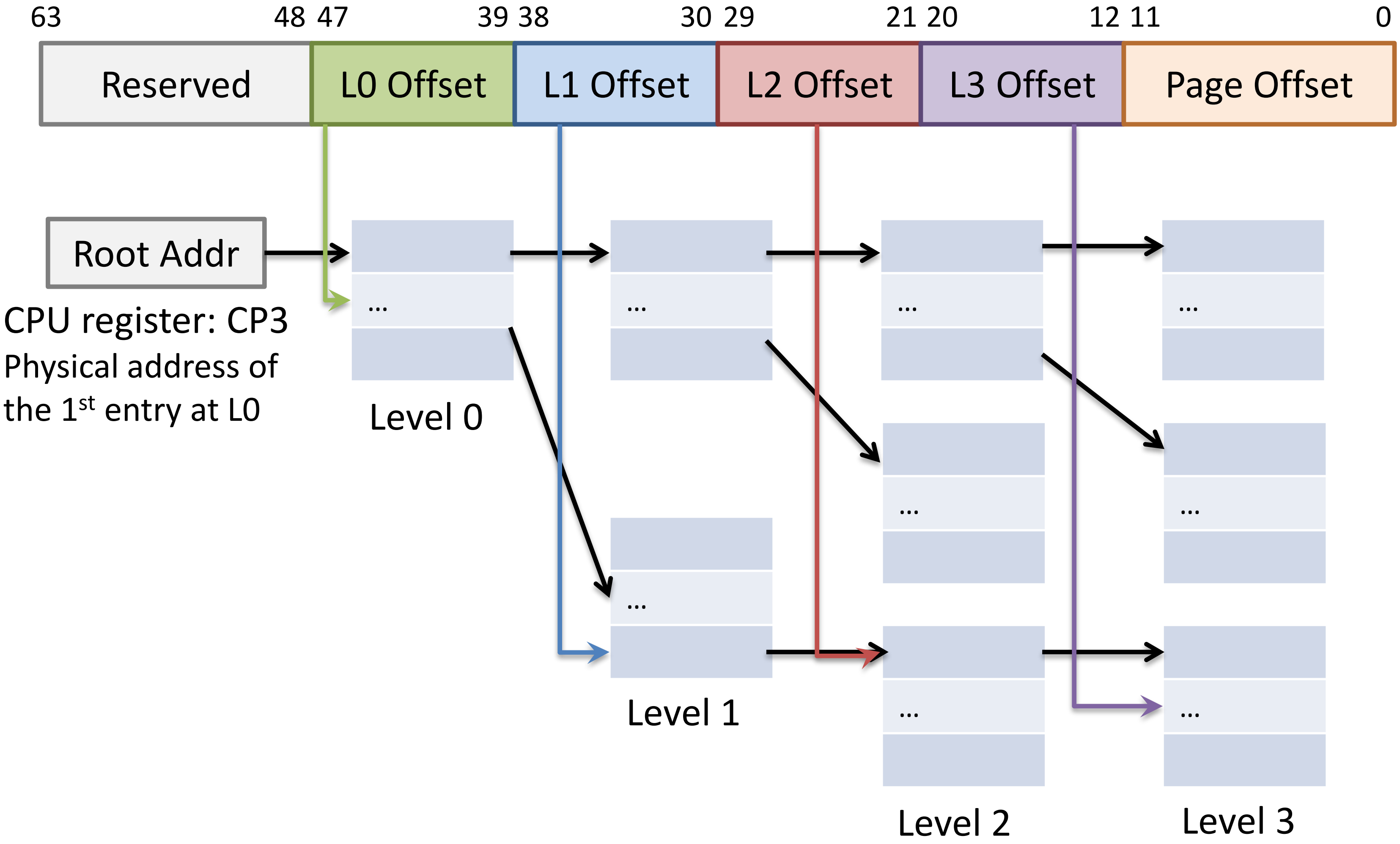
63 48	47 39	38 30	29 21	20 12	11 0
Reserved	L0 Offset	L1 Offset	L2 Offset	L3 Offset	Page Offset



CPU register: CP3

Physical address of the 1st entry at L0

Multi-level Page Tables on x86_64



63 48 47 39 38 30 29 21 20 12 11 0

Reserved

L0 Offset

L1 Offset

L2 Offset

L3 Offset

Page Offset

Root Addr

CPU register: CP3

Physical address of
the 1st entry at L0

Level 0

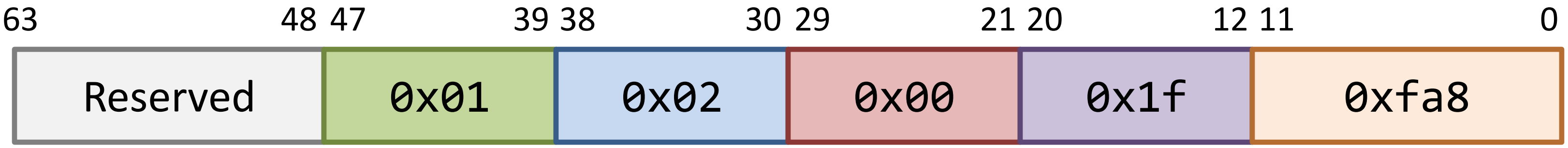
Level 1

Level 2

Level 3

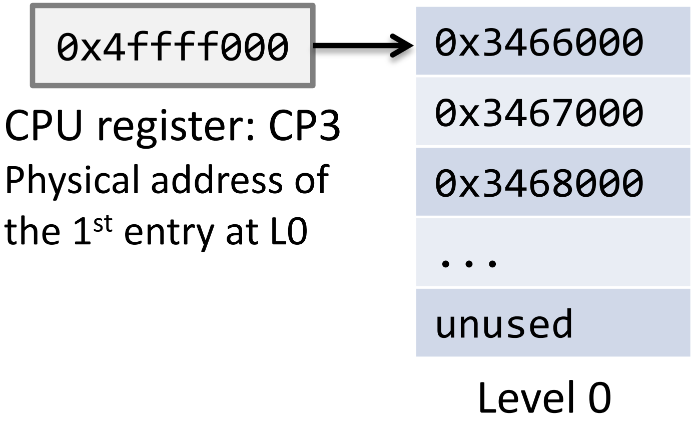
…

Multi-level Page Tables on x86_64



63 48 47 39 38 30 29 21 20 12 11 0

Reserved | 0x01 | 0x02 | 0x00 | 0x1f | 0xfa8



0x4ffff000

CPU register: CP3

Physical address of the 1st entry at L0

0x3466000

0x3467000

0x3468000

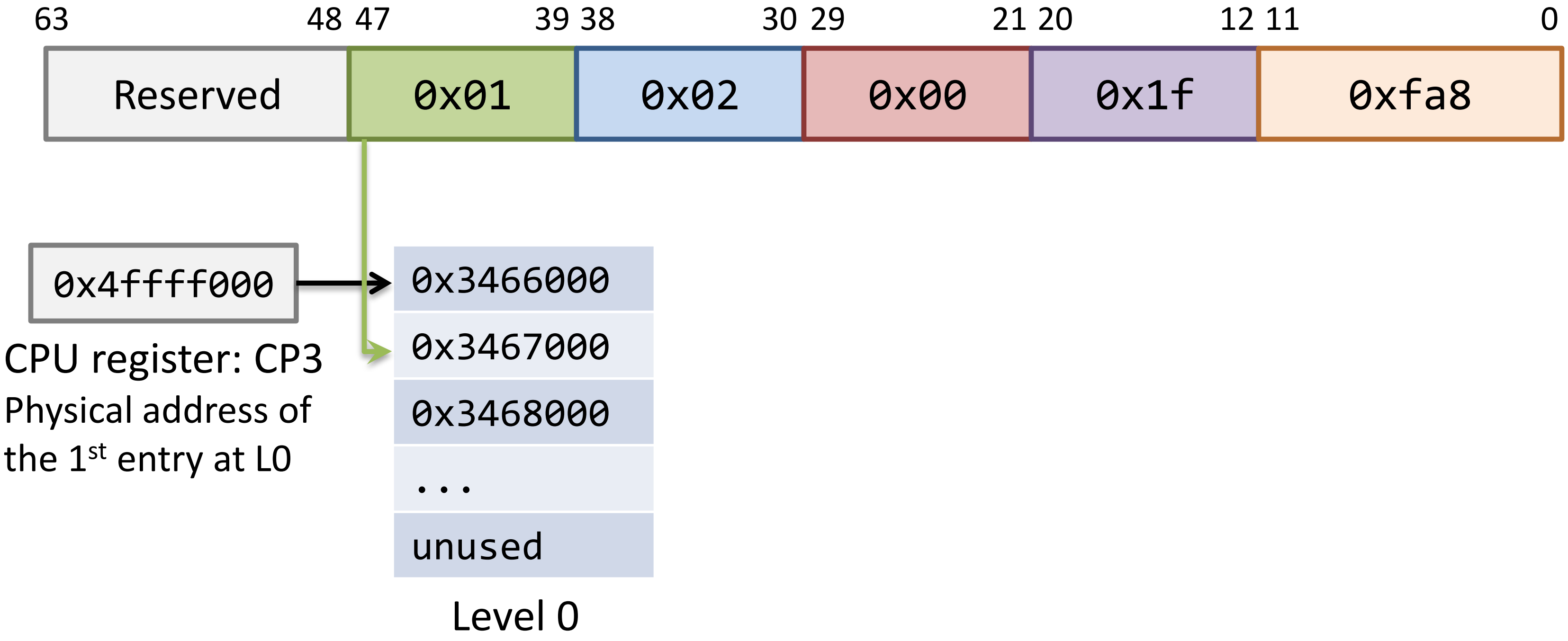
...

unused

Level 0

Virtual Address: 0x0102001ffa8

Multi-level Page Tables on x86_64



63 48 47 39 38 30 29 21 20 12 11 0

Reserved

0x01

0x02

0x00

0x1f

0xfa8

0x4ffff000

CPU register: CP3

Physical address of the 1st entry at L0

0x3466000

0x3467000

0x3468000

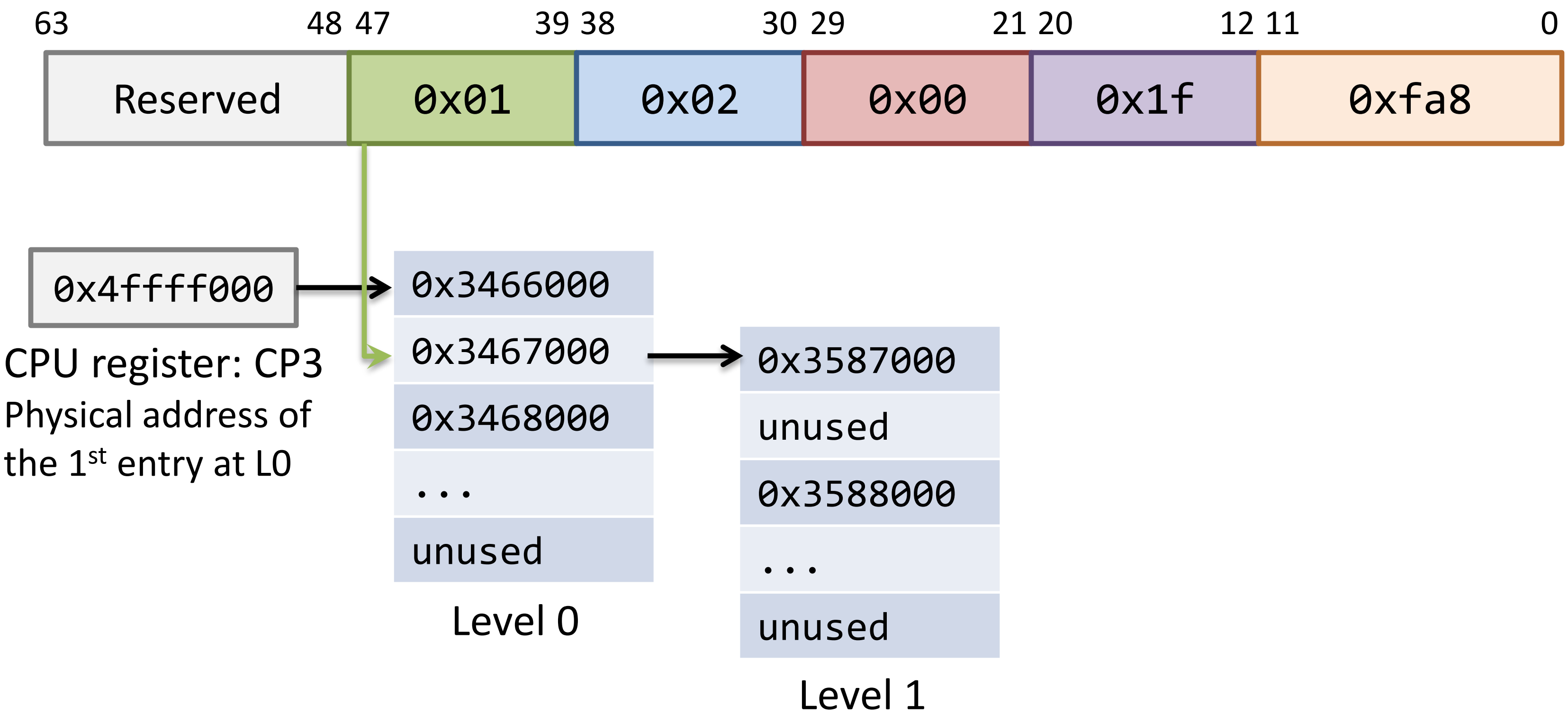
...

unused

Level 0

Virtual Address: 0x0102001ffa8

Multi-level Page Tables on x86_64



63 48	47 39	38 30	29 21	20 12	11 0
Reserved	0x01	0x02	0x00	0x1f	0xfa8

0x4ffff000

CPU register: CP3

Physical address of the 1st entry at L0

0x3466000

0x3467000

0x3468000

...

unused

Level 0

0x3587000

unused

0x3588000

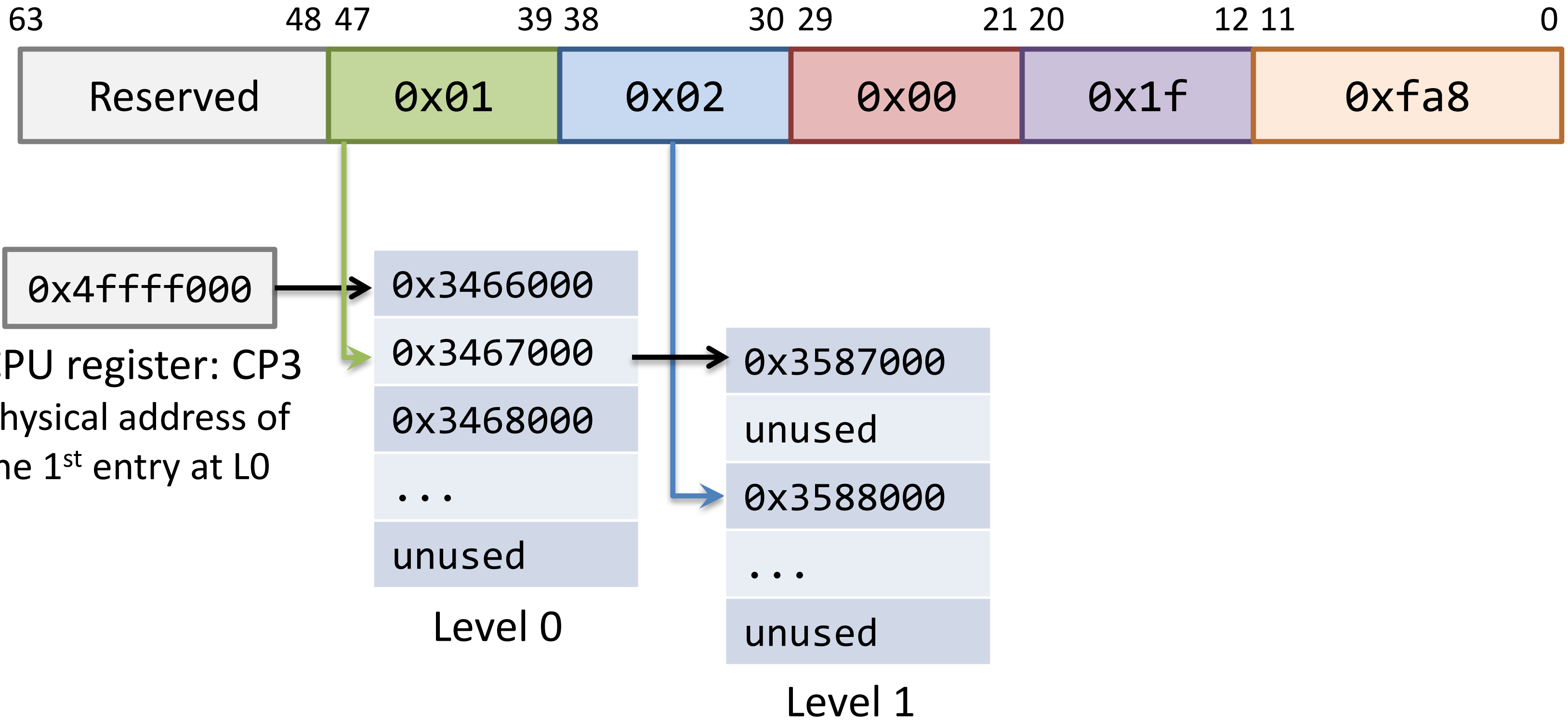
...

unused

Level 1

Virtual Address: 0x0102001ffa8

Multi-level Page Tables on x86_64



63 48	47 39	38 30	29 21	20 12	11 0
Reserved	0x01	0x02	0x00	0x1f	0xfa8

0x4ffff000

CPU register: CP3

Physical address of the 1st entry at L0

Level 0:

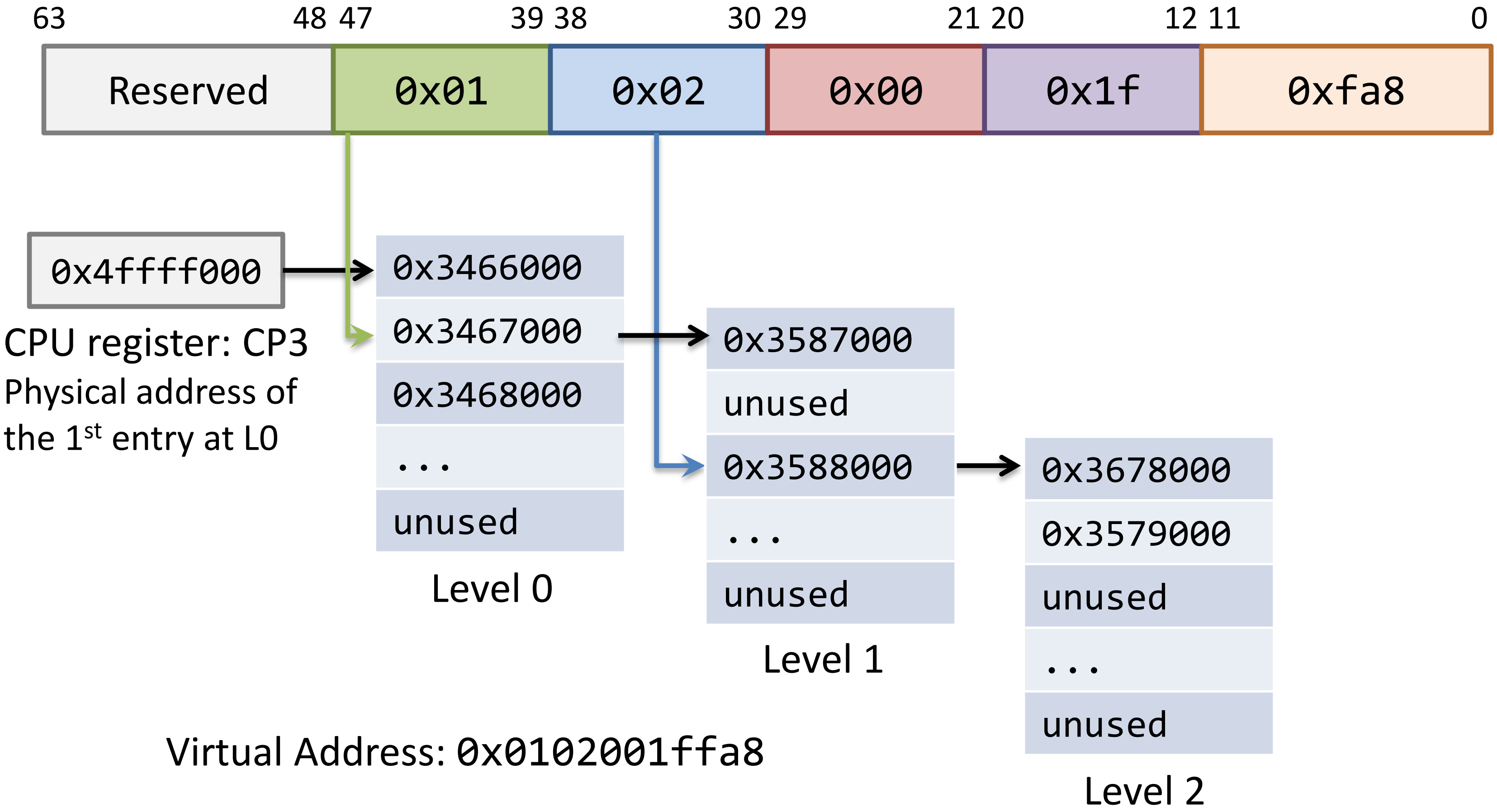
0x3466000
0x3467000
0x3468000
...
unused

Level 1:

0x3587000
unused
0x3588000
...
unused

Virtual Address: 0x0102001ffa8

Multi-level Page Tables on x86_64



63 48 47 39 38 30 29 21 20 12 11 0

Reserved	0x01	0x02	0x00	0x1f	0xfa8

0x4ffff000

CPU register: CP3

Physical address of the 1st entry at L0

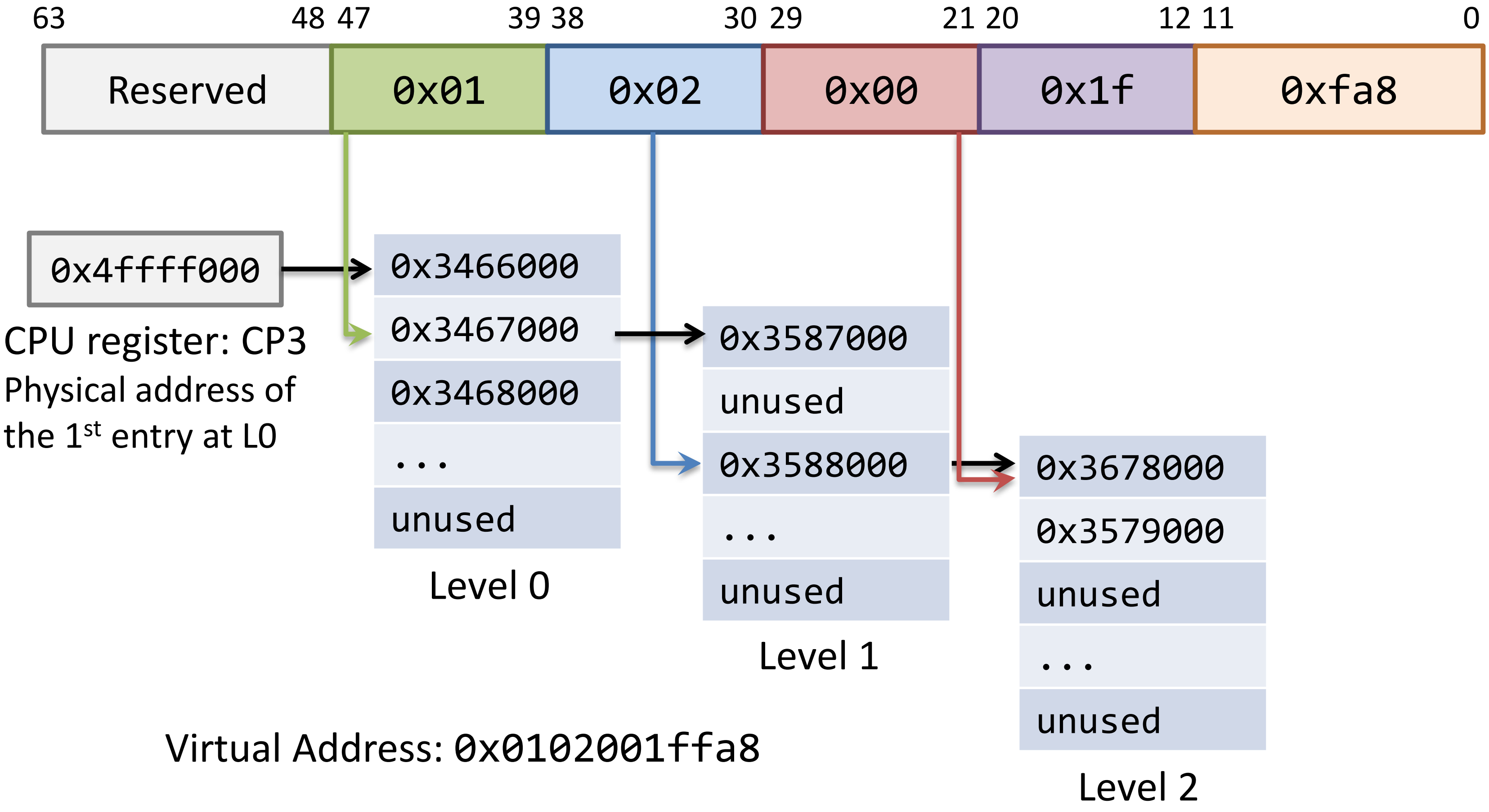
Level 0:
- 0x3466000
- 0x3467000
- 0x3468000
- ...
- unused

Level 1:
- 0x3587000
- unused
- 0x3588000
- ...
- unused

Level 2:
- 0x3678000
- 0x3579000
- unused
- ...
- unused

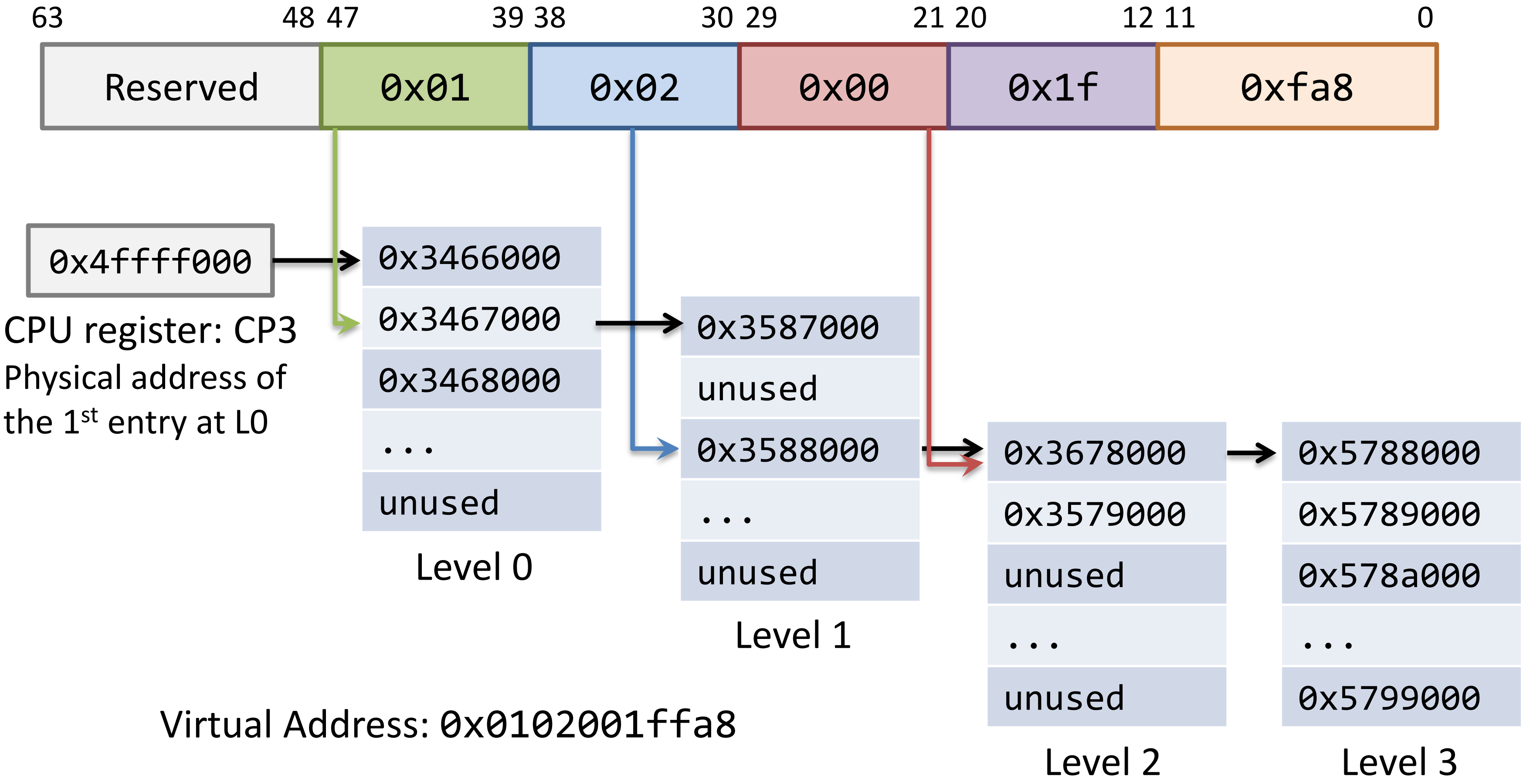
Virtual Address: 0x0102001ffa8

Multi-level Page Tables on x86_64



Virtual Address: 0x0102001ffa8

Multi-level Page Tables on x86_64



63 48 47 39 38 30 29 21 20 12 11 0

Reserved | 0x01 | 0x02 | 0x00 | 0x1f | 0xfa8

0x4ffff000

CPU register: CP3
Physical address of
the 1st entry at L0

Level 0:
0x3466000
0x3467000
0x3468000
...
unused

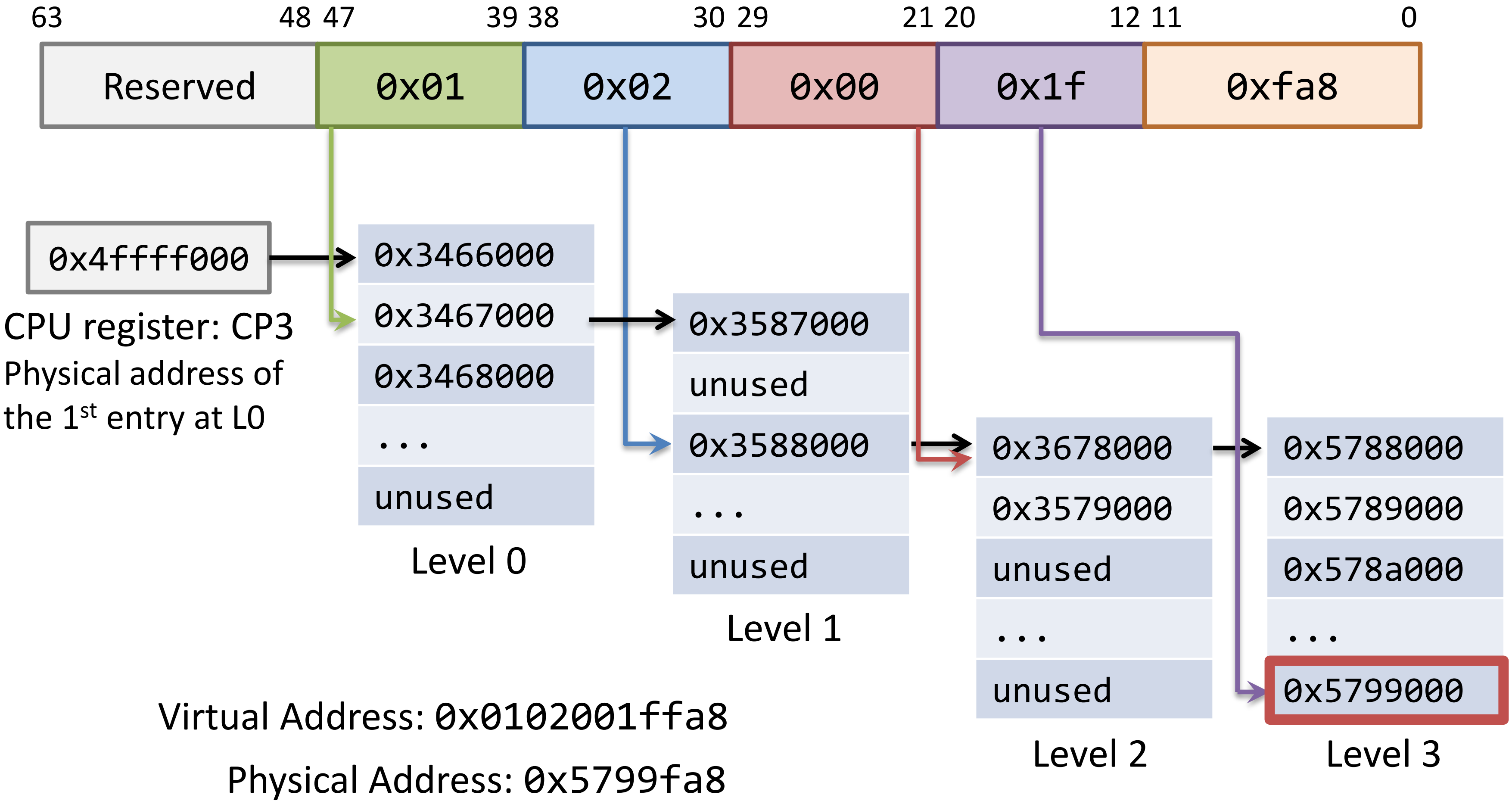
Level 1:
0x3587000
unused
0x3588000
...
unused

Level 2:
0x3678000
0x3579000
unused
...
unused

Level 3:
0x5788000
0x5789000
0x578a000
...
0x5799000

Virtual Address: 0x0102001ffa8

Multi-level Page Tables on x86_64



63 48 47 39 38 30 29 21 20 12 11 0

Reserved	0x01	0x02	0x00	0x1f	0xfa8

0x4ffff000

CPU register: CP3

Physical address of the 1st entry at L0

Level 0	Level 1	Level 2	Level 3
0x3466000	0x3587000	0x3678000	0x5788000
0x3467000	unused	0x3579000	0x5789000
0x3468000	0x3588000	unused	0x578a000
...	...	...	...
unused	unused	unused	0x5799000

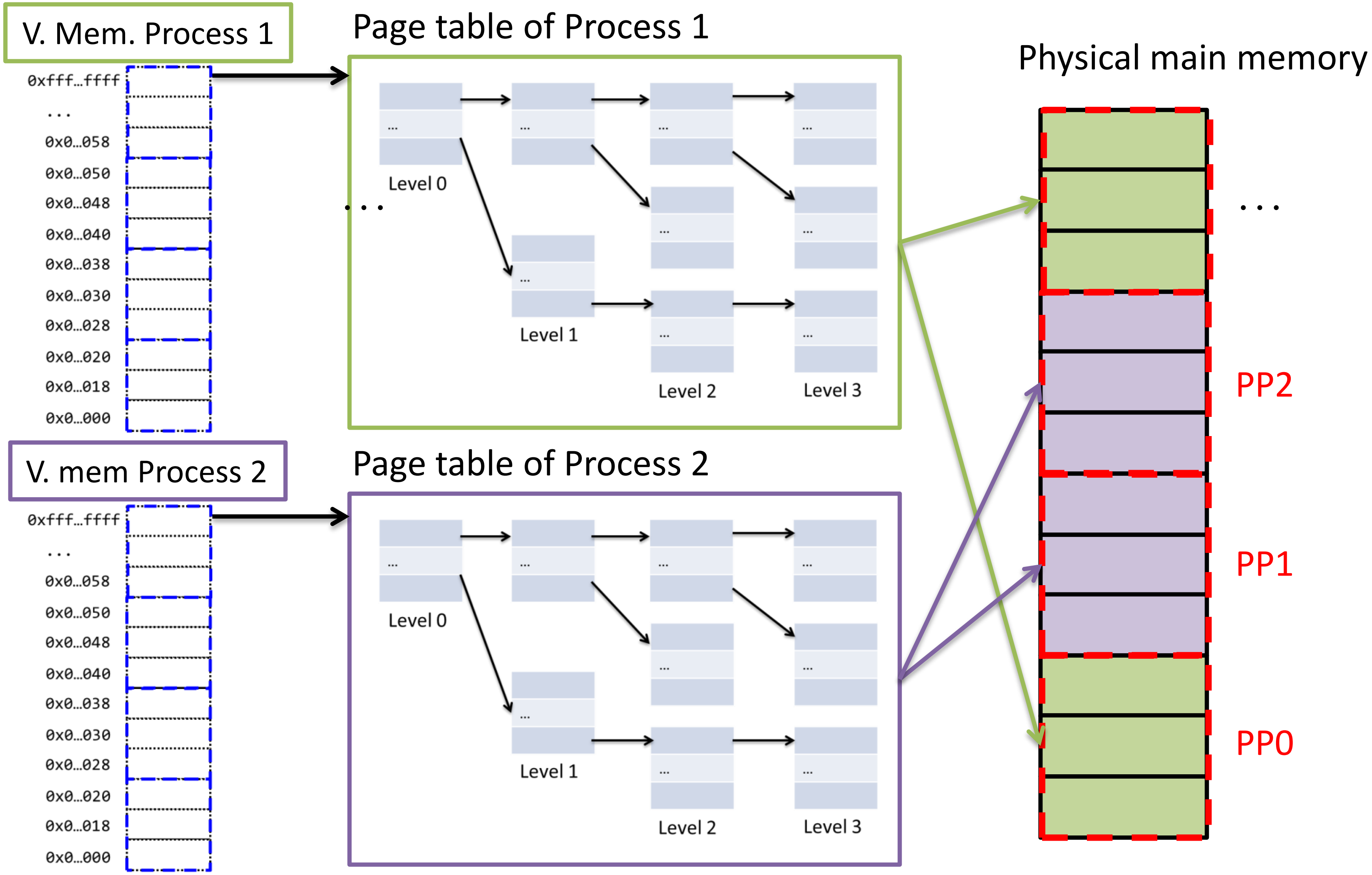
Virtual Address: 0x0102001ffa8

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



Question

Question

- Why does multi-level page table reduce page table size?

Question

- Why does multi-level page table reduce page table size?

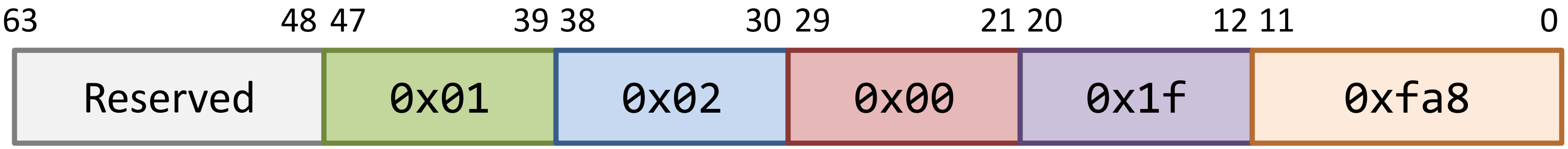
- **Answer**:
  - 4-level page table is not fully occupied.
  - Demand paging: OS constructs page table on demand.

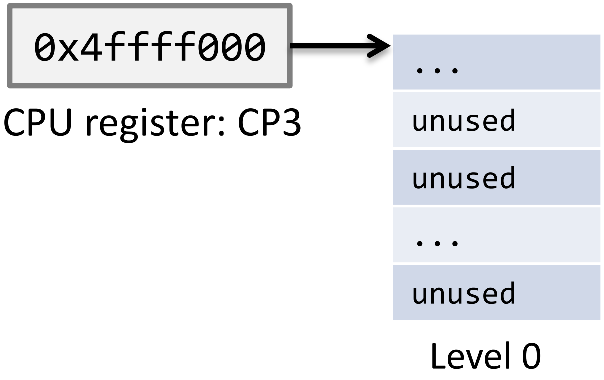
Demand Paging

- Memory Allocation (e.g., `p = 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

```
→ char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
  p[0] = 'c'
  p[4096] = 's'
```



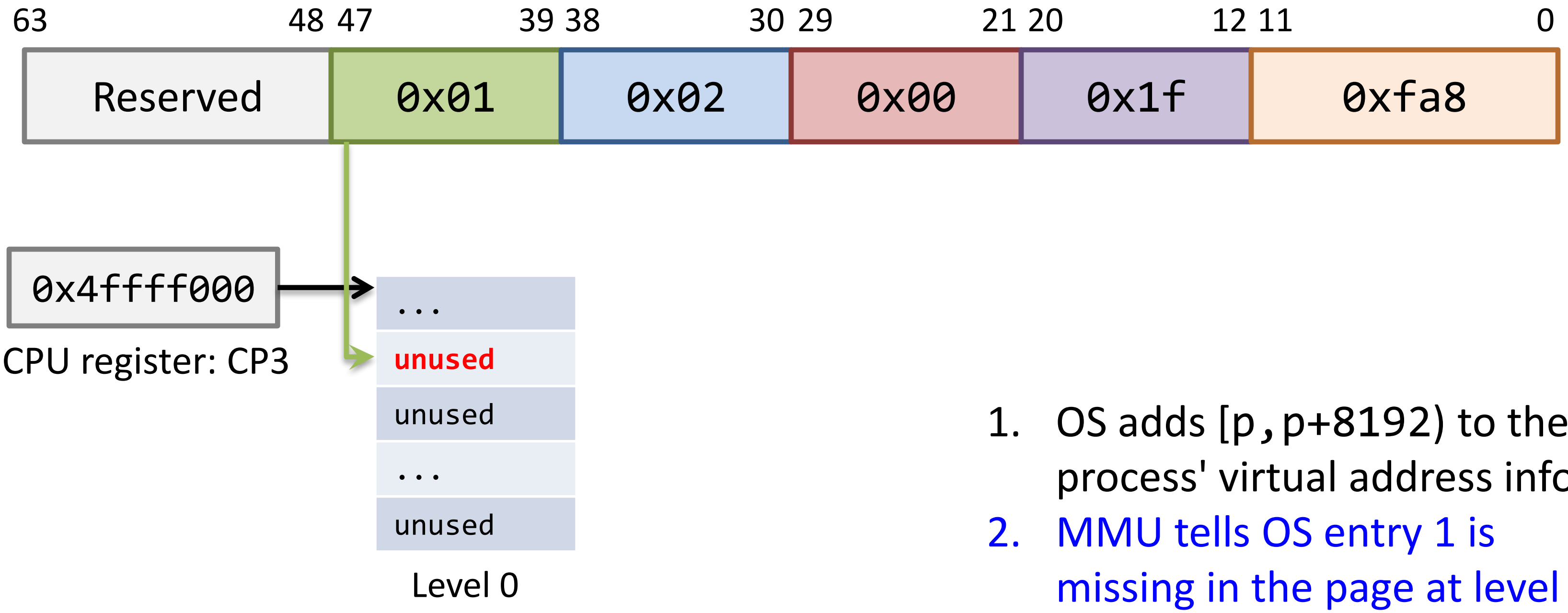


current process' page table

1. OS adds [p,p+8192) to the process' virtual address info

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

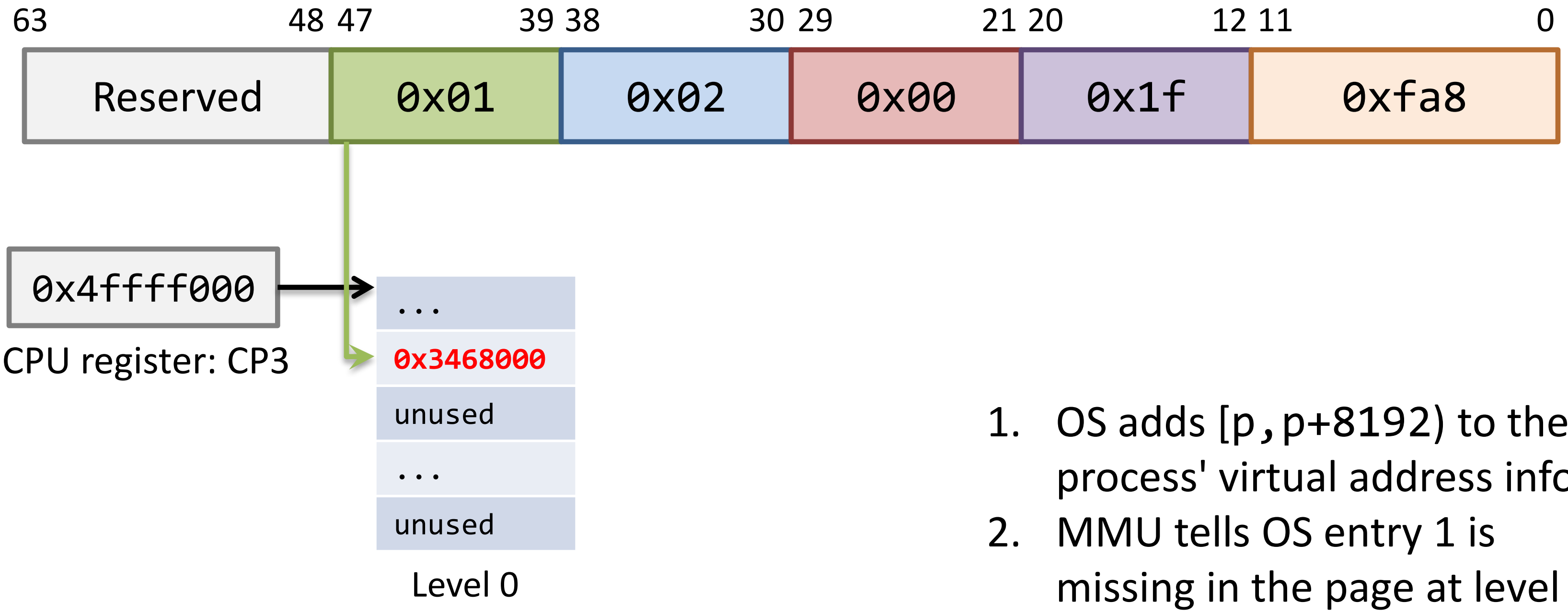


current process' page table

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*)

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

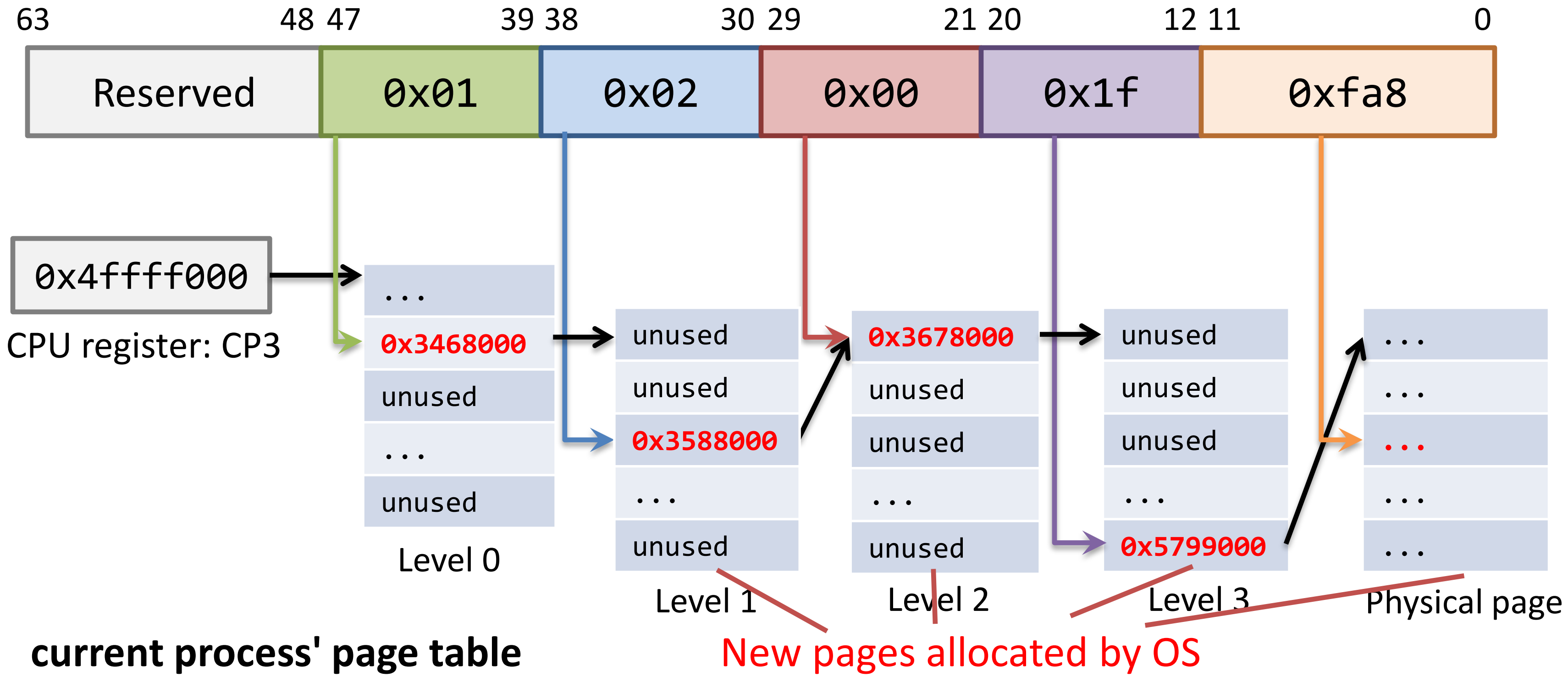


current process' page table

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*)

Demand Paging

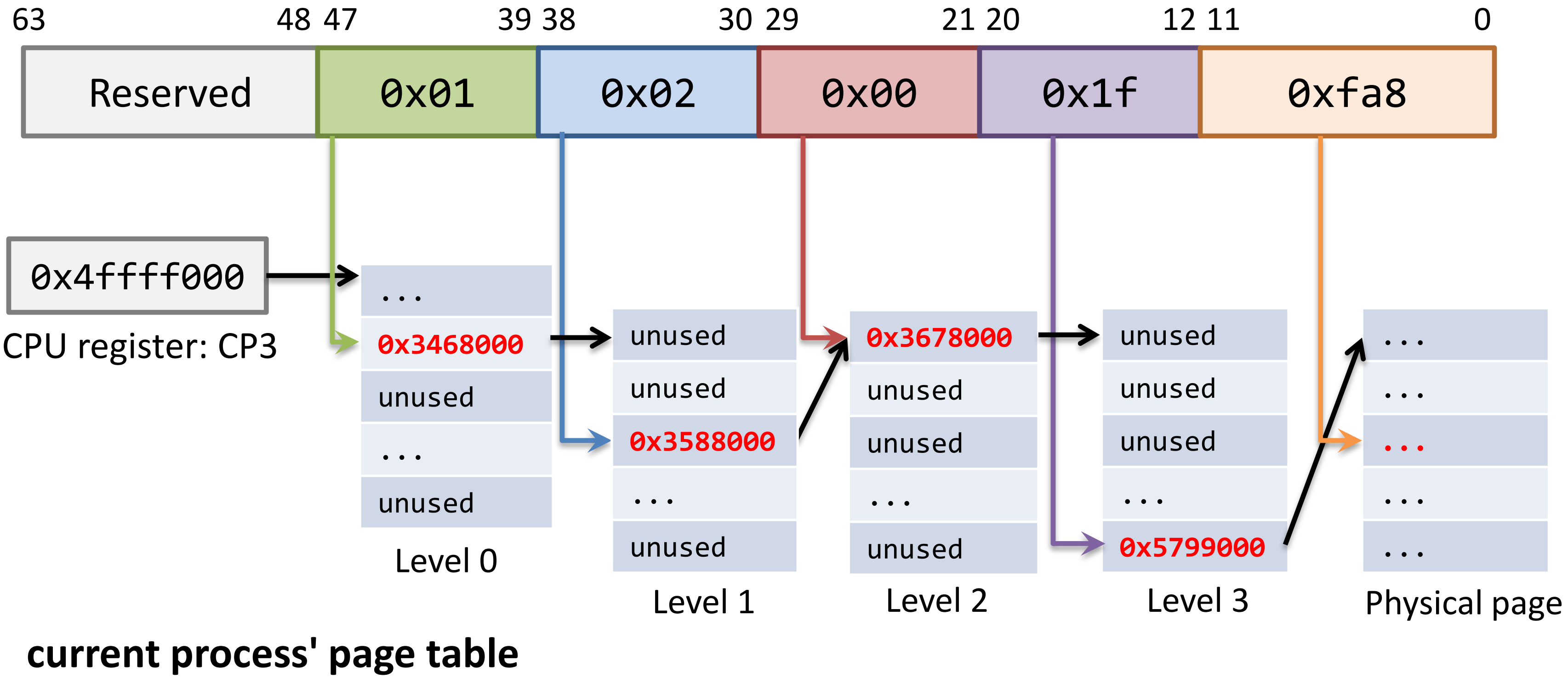
```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```



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

Demand Paging

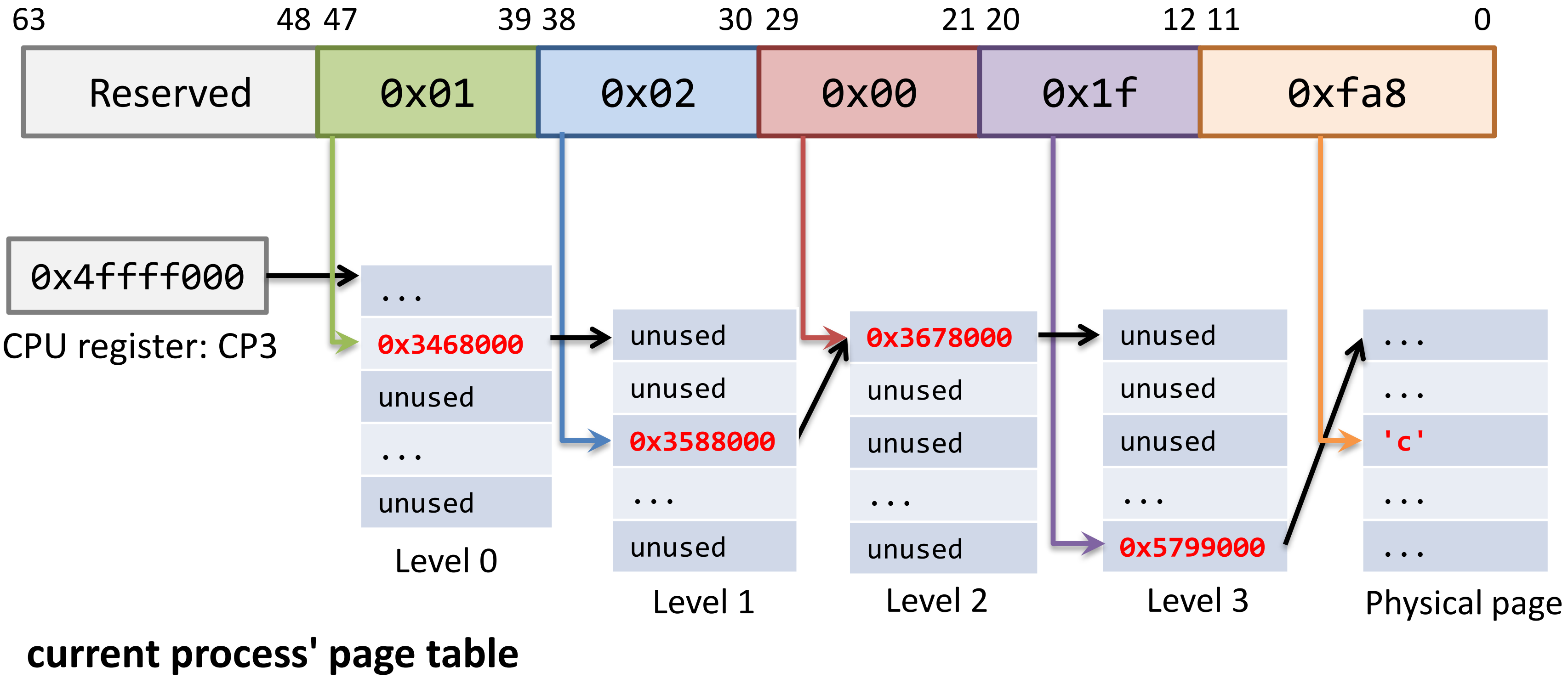
```
    char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
→  p[0] = 'c'
    p[4096] = 's'
```



4. OS tells the CPU to resume execution

Demand Paging

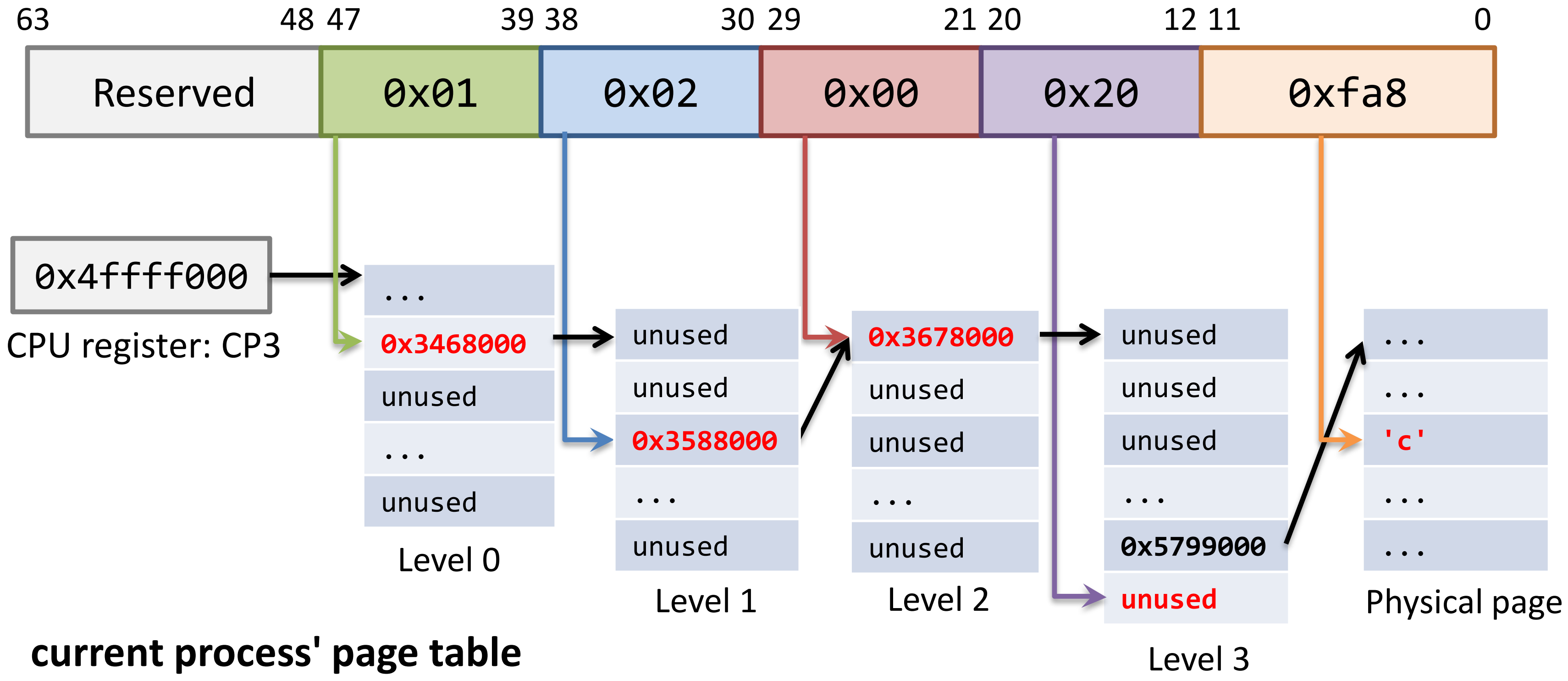
```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```



5. MMU translates address again and accesses the physical memory.

Demand Paging

```
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'   // p+4096 is 0x01020020fa8
```



2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)