Isolation & Virtual Memory: Concepts

Jinyang Li

based on the slides of Tiger Wang

Layered Organization

User Applications



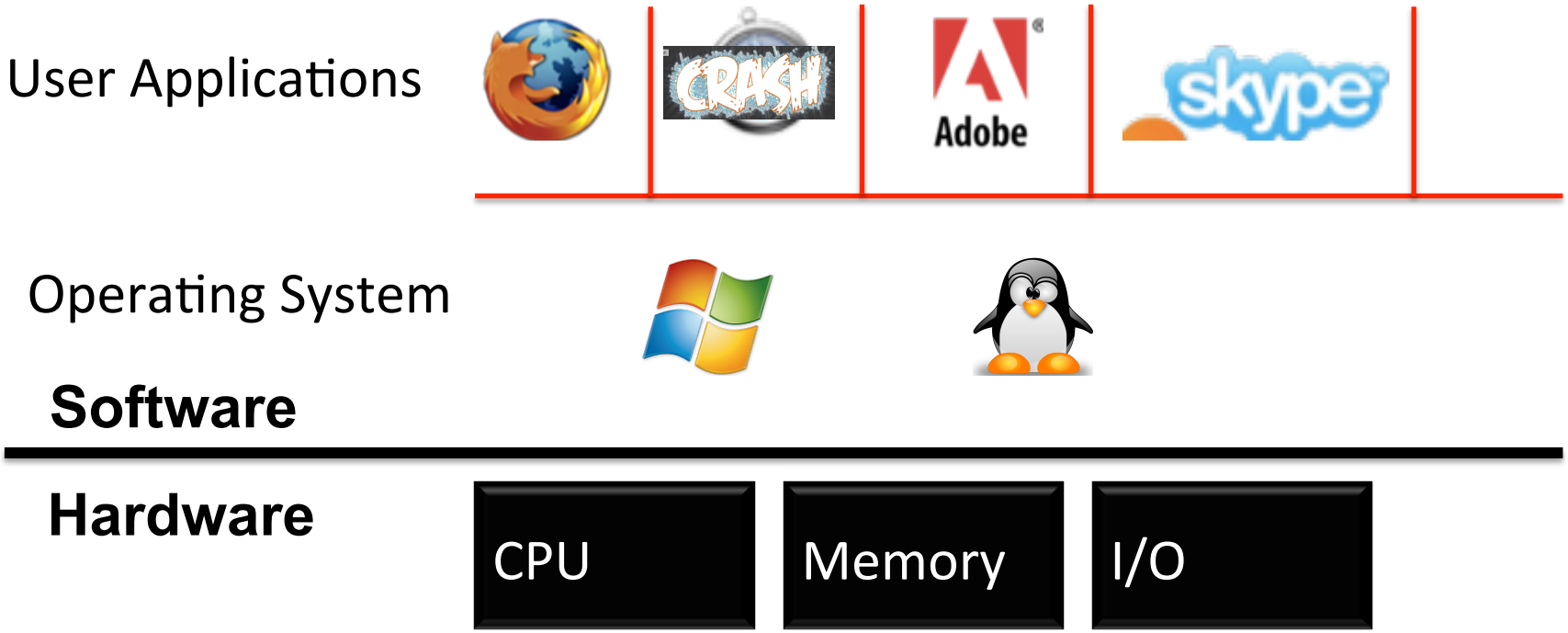
Operating System


Software

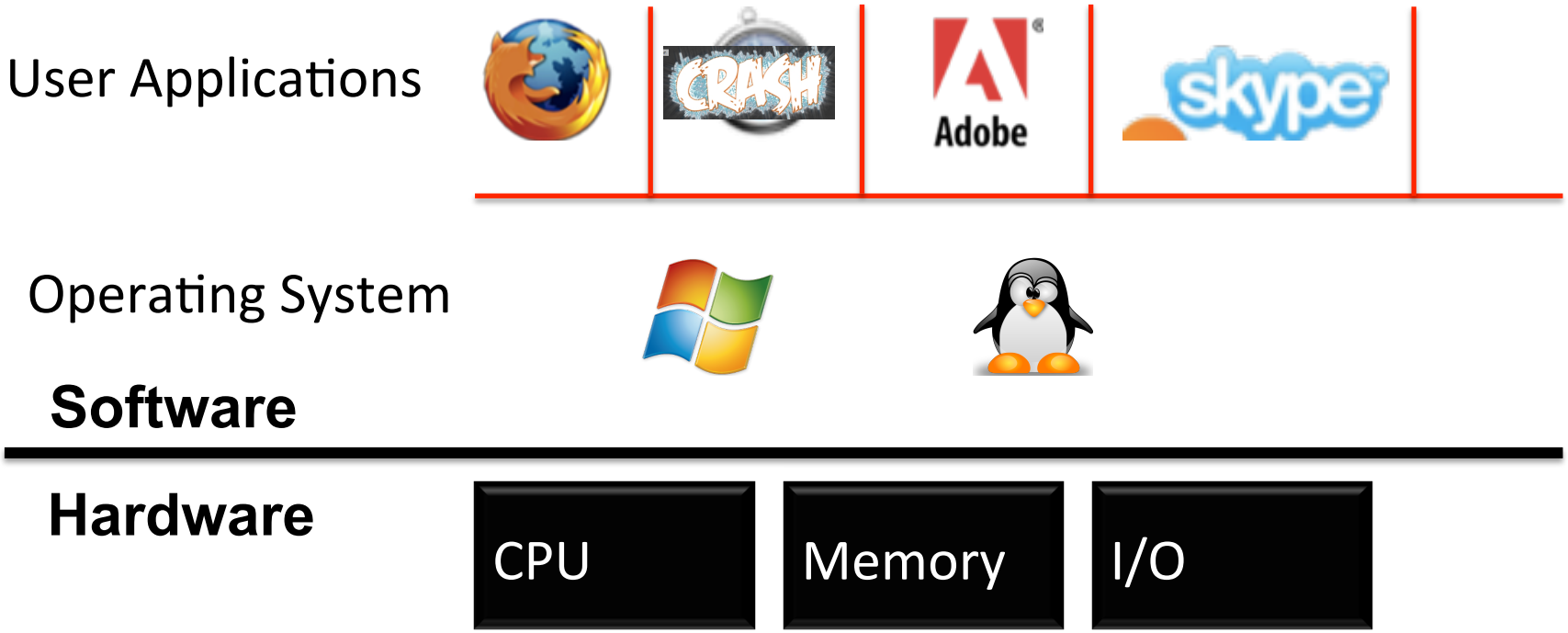
Hardware

CPU | Memory | I/O

Isolation

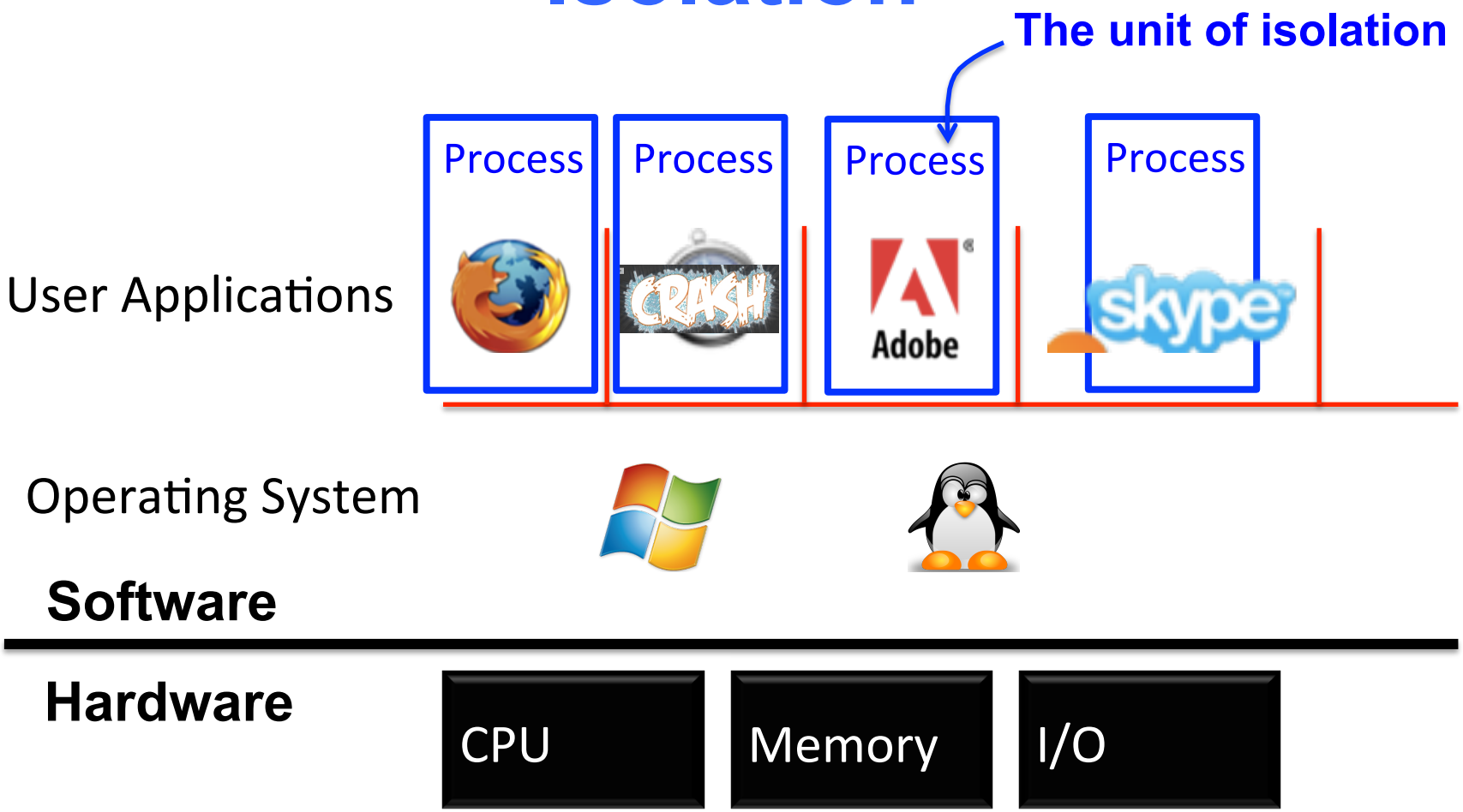

User Applications
CRASH
Adobe
skype
Operating System
Software
Hardware
CPU
Memory
I/O

Isolation



Isolation – Enforced separation to contain effects of failures

Isolation



Isolation – Enforced separation to contain effects of failures

Process

An instance of a computer program that being executed

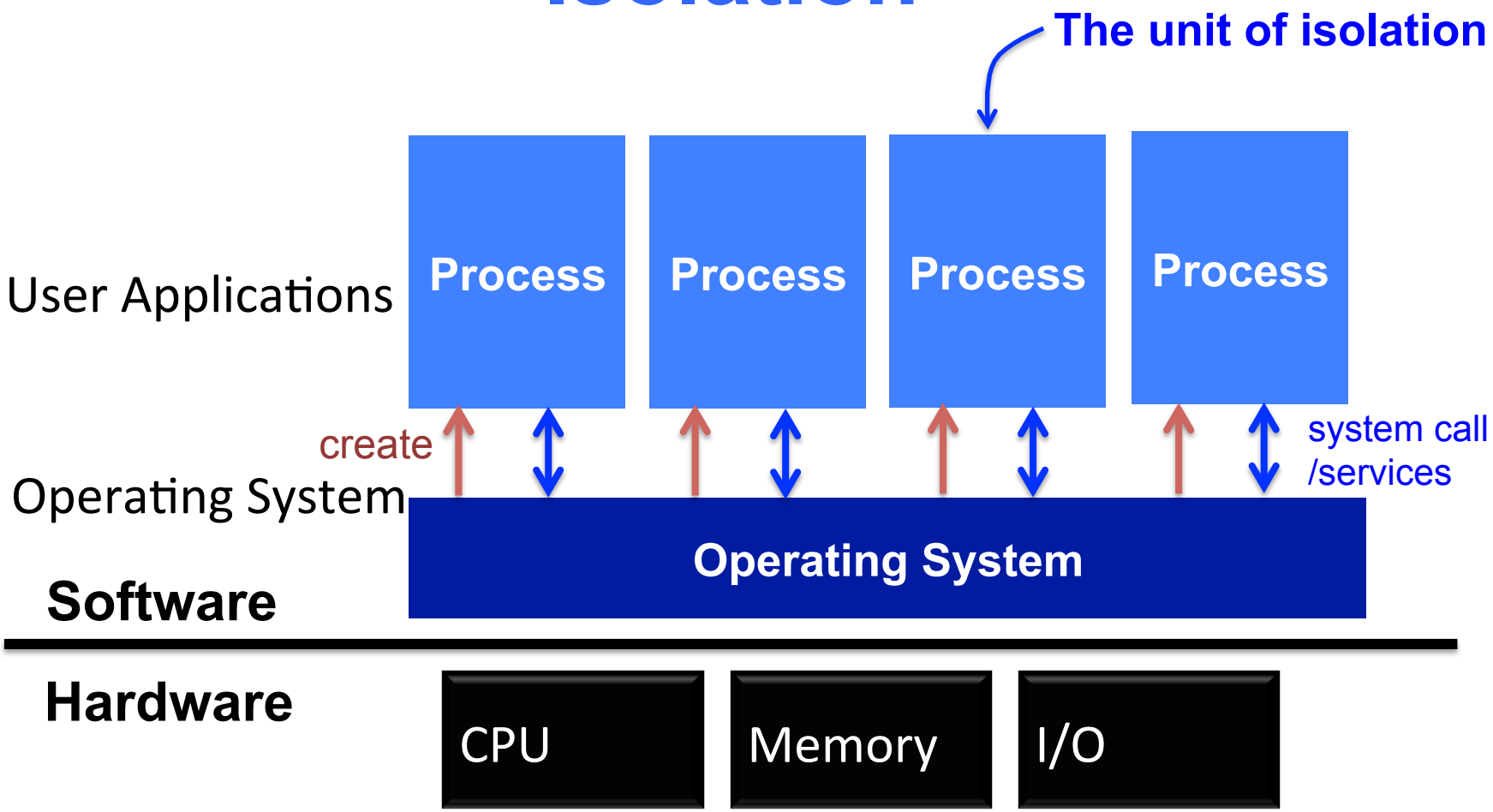
Program vs. Process

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

Different processes have different process id

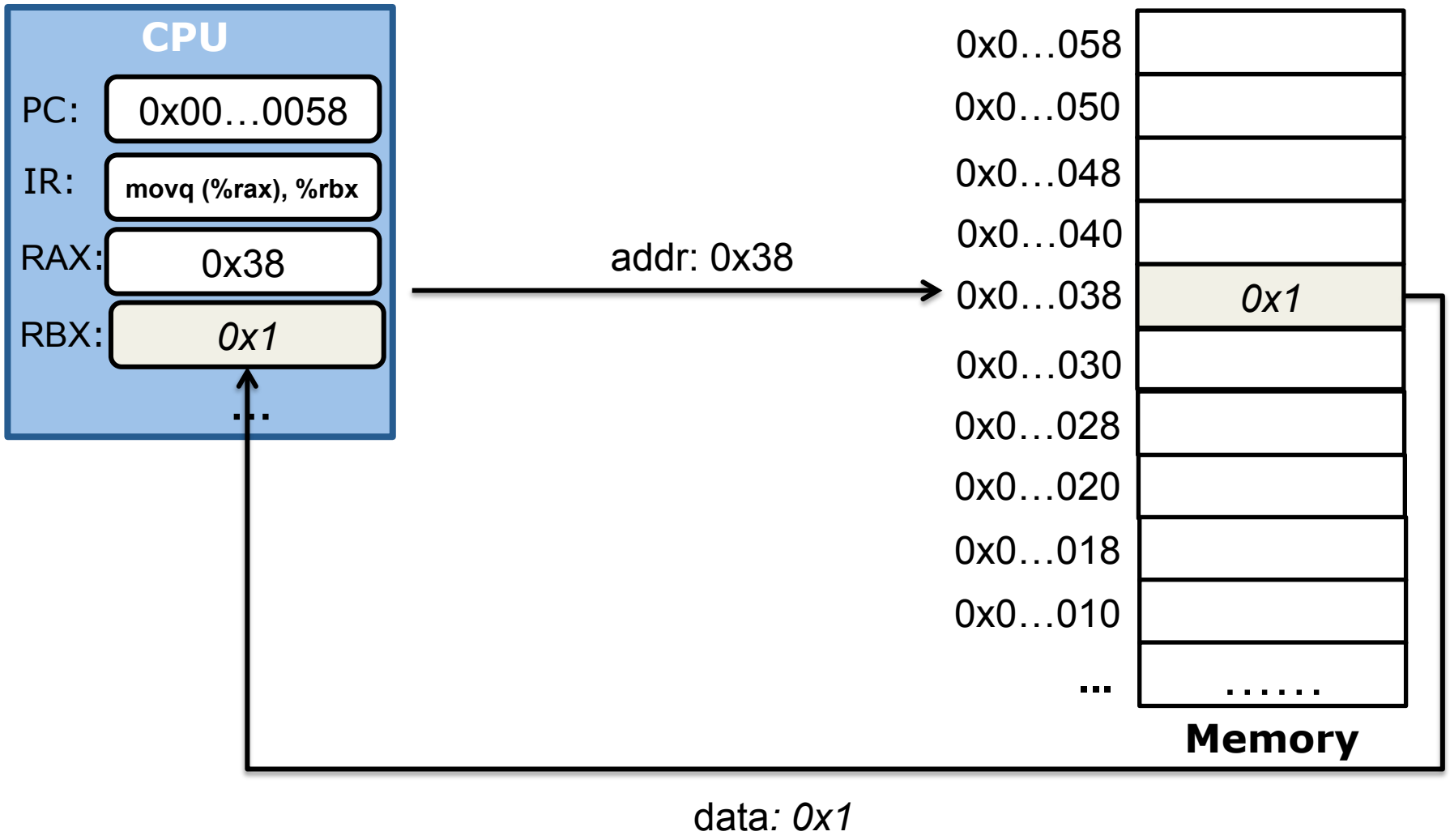
- ***getpid()*** function call returns id of current process
- Command ***ps***: list all processes

Isolation



To run a program, OS starts a process and provide services through system call (*getpid(), printf()*).

Our "Mental Model" of Memory System



Processes share the same address space

The requirements:

- Different processes use the same address to store their local code/data.
- One process can not access another process' memory

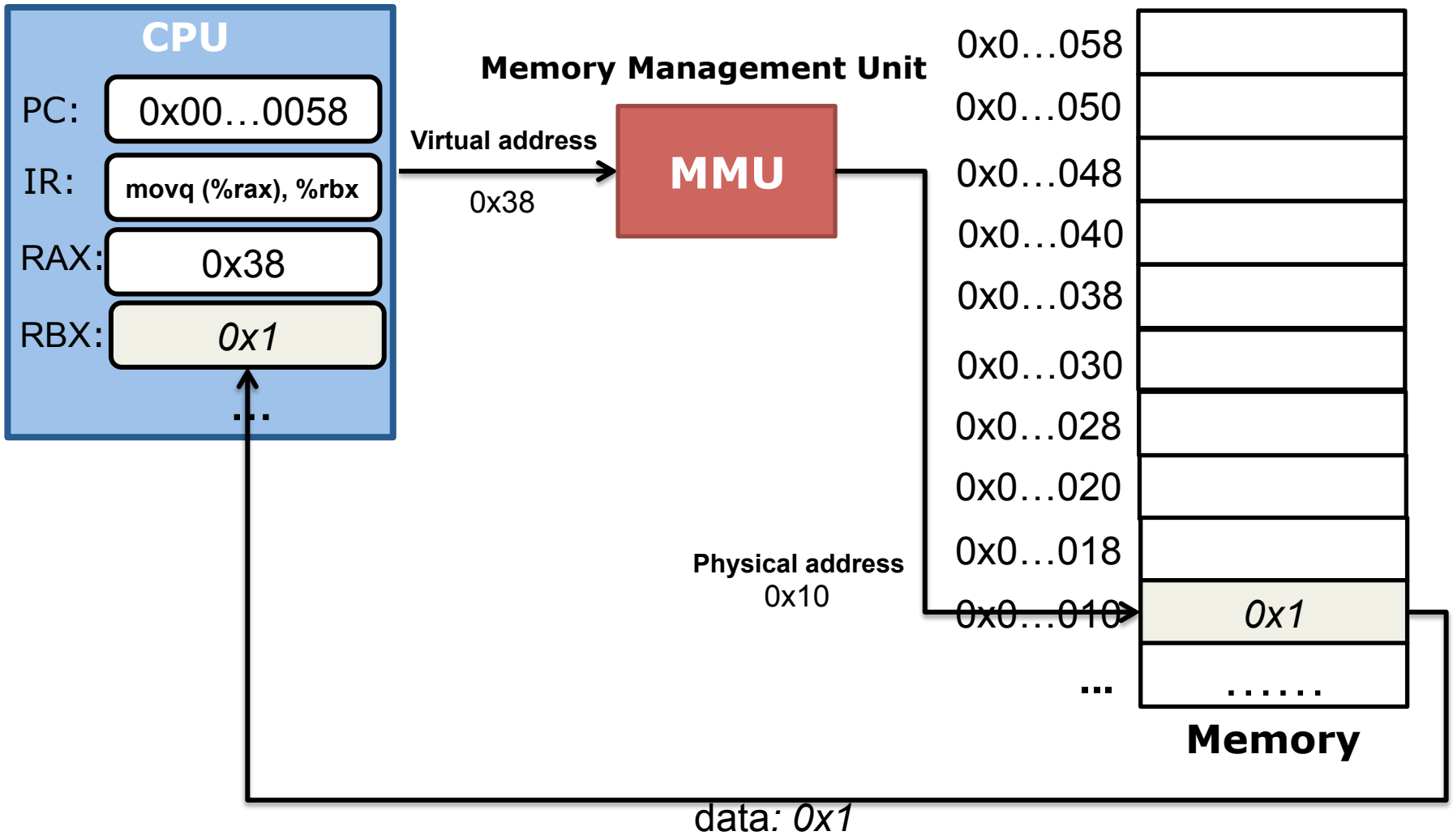
Why

- 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. (e.t.c. linking or loading)

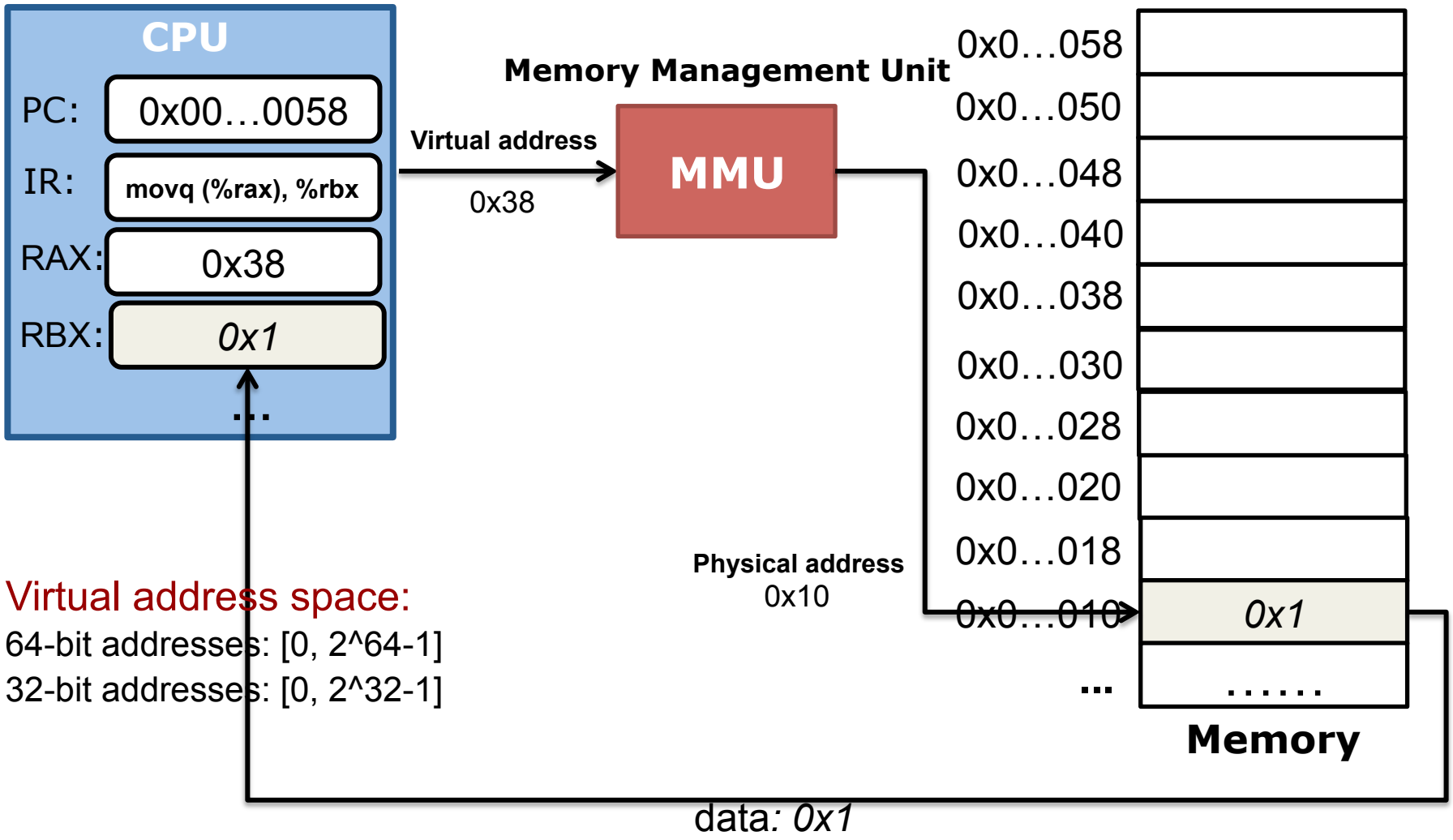
How

- **Virtual Memory**

Real system – Virtual addressing



Real system – Virtual addressing



Address Translation – Strawman

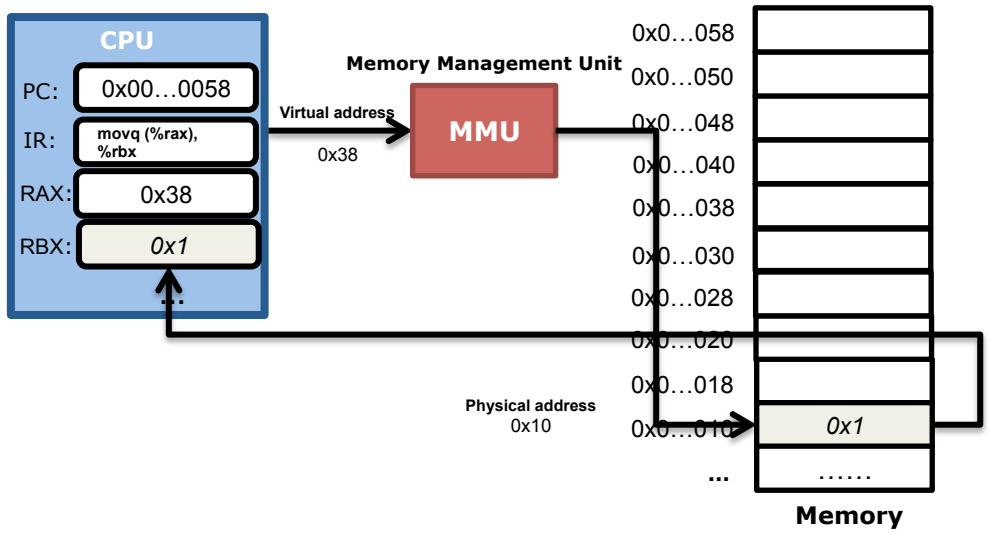
MMU has a mapping table at byte granularity

- Map each virtual address into a physical address

MMU

Virtual address	Physical address
…	…
0x58	0x10
0x59	0x11
…	…

mapping table



data: *0x1*

Address Translation – Strawman

MMU has a mapping table at byte granularity

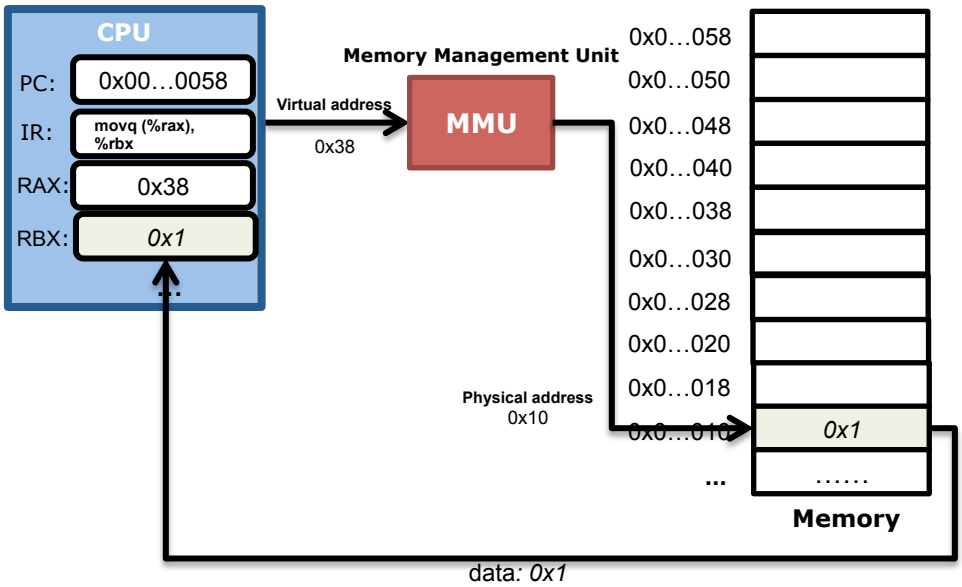
- Map each virtual address into a physical address

MMU

Virtual address	Physical address
…	…
0x58	0x10
0x59	0x11
…	…

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

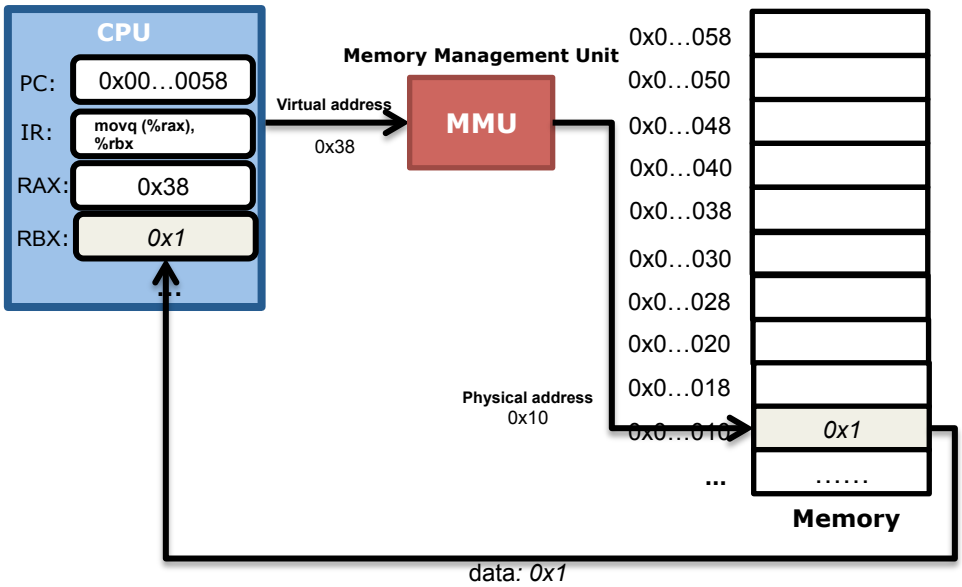
MMU

Virtual address	Physical address
…	…
0x58	0x10
0x59	0x11
…	…

mapping table

What is the size of mapping table?

Size of virtual address space 2^{64}



Address Translation – Page

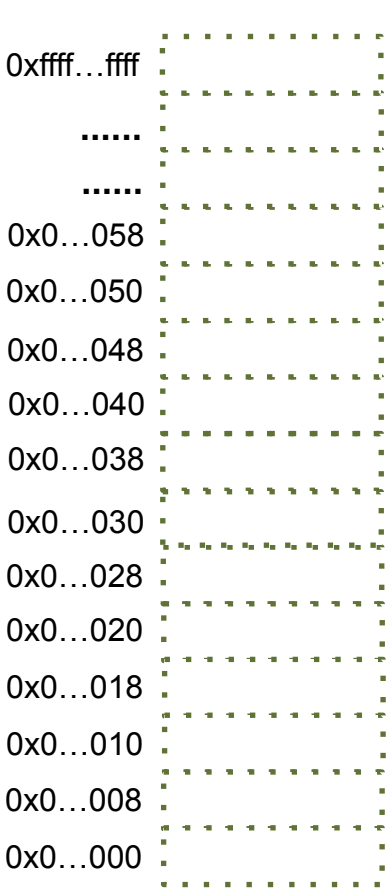
Observation

- Both virtual memory space and physical memory space are contiguous

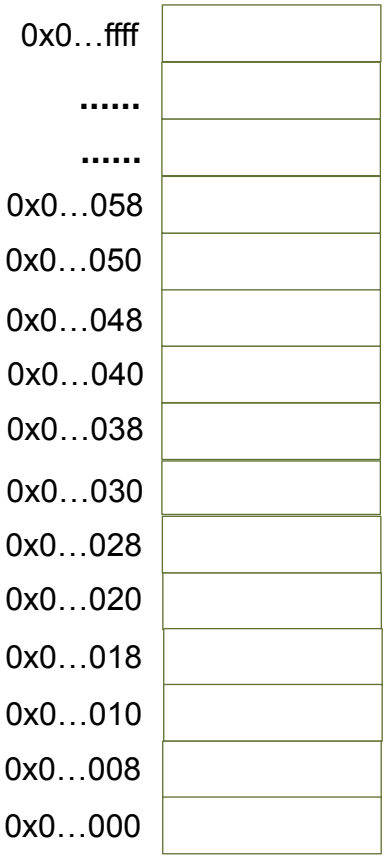
Build the mapping at coarse granularity

- 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

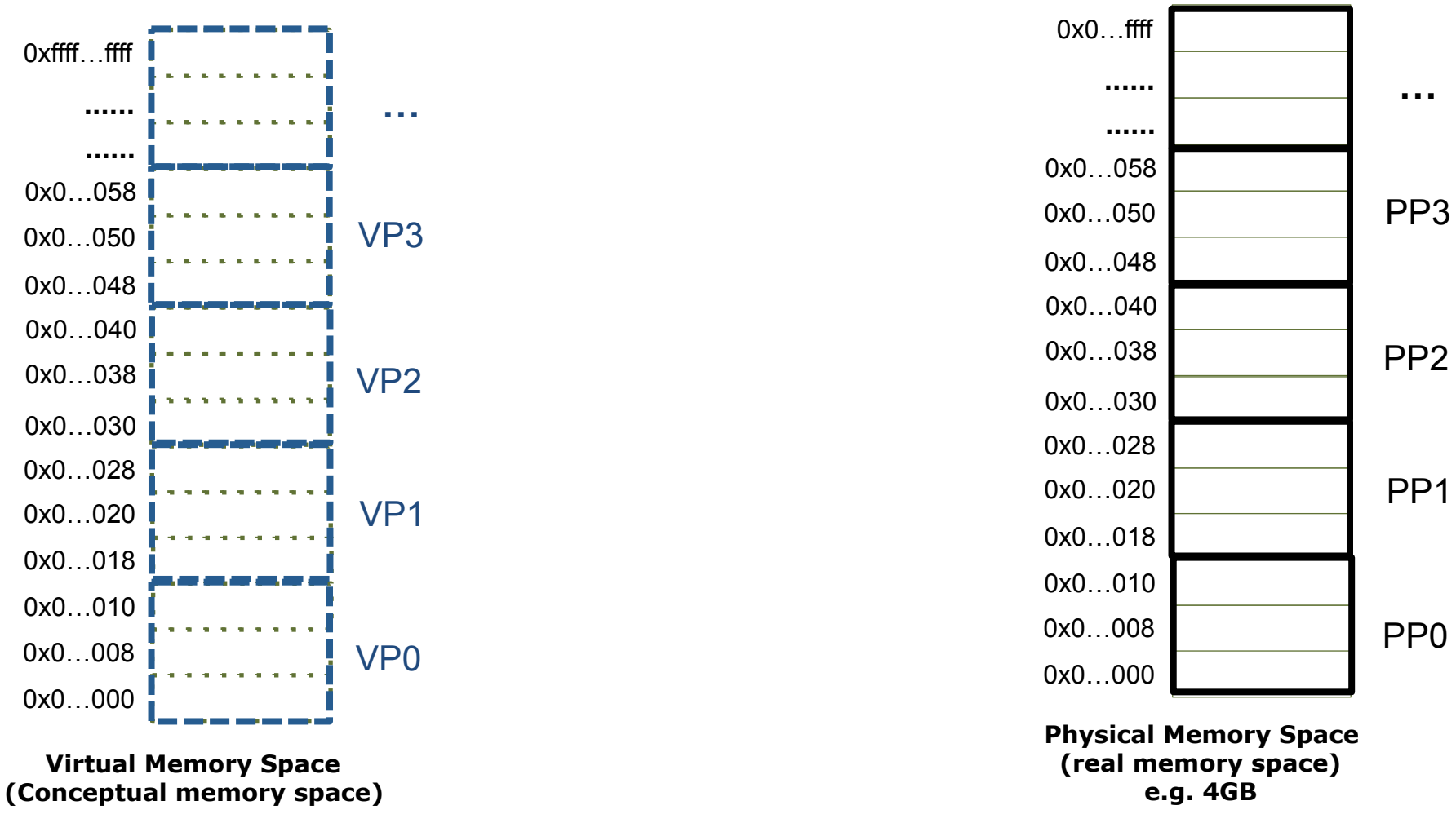


Virtual Memory Space
(Conceptual memory space)

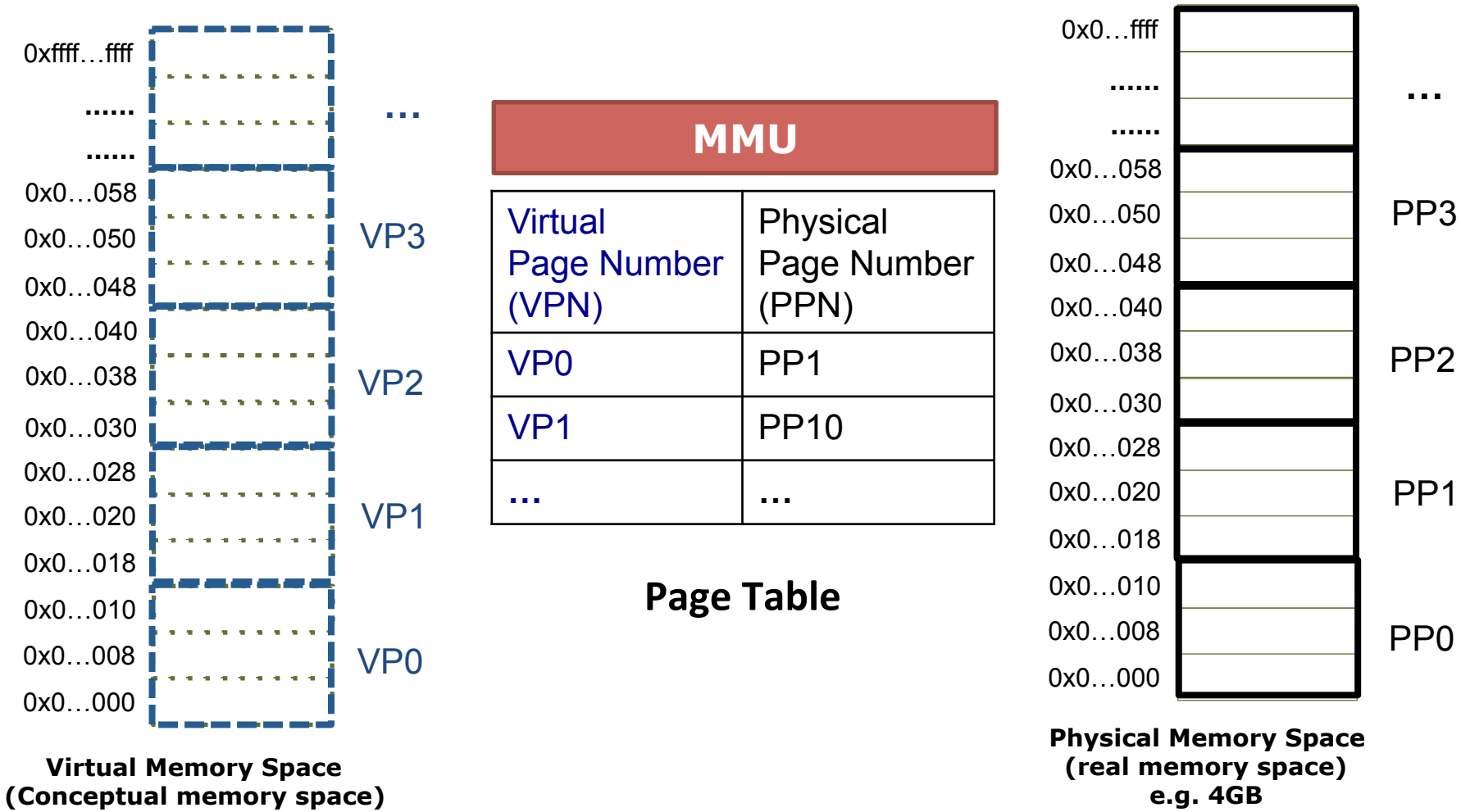


Physical Memory Space
(real memory space)
e.g. 4GB

Address Translation – Page



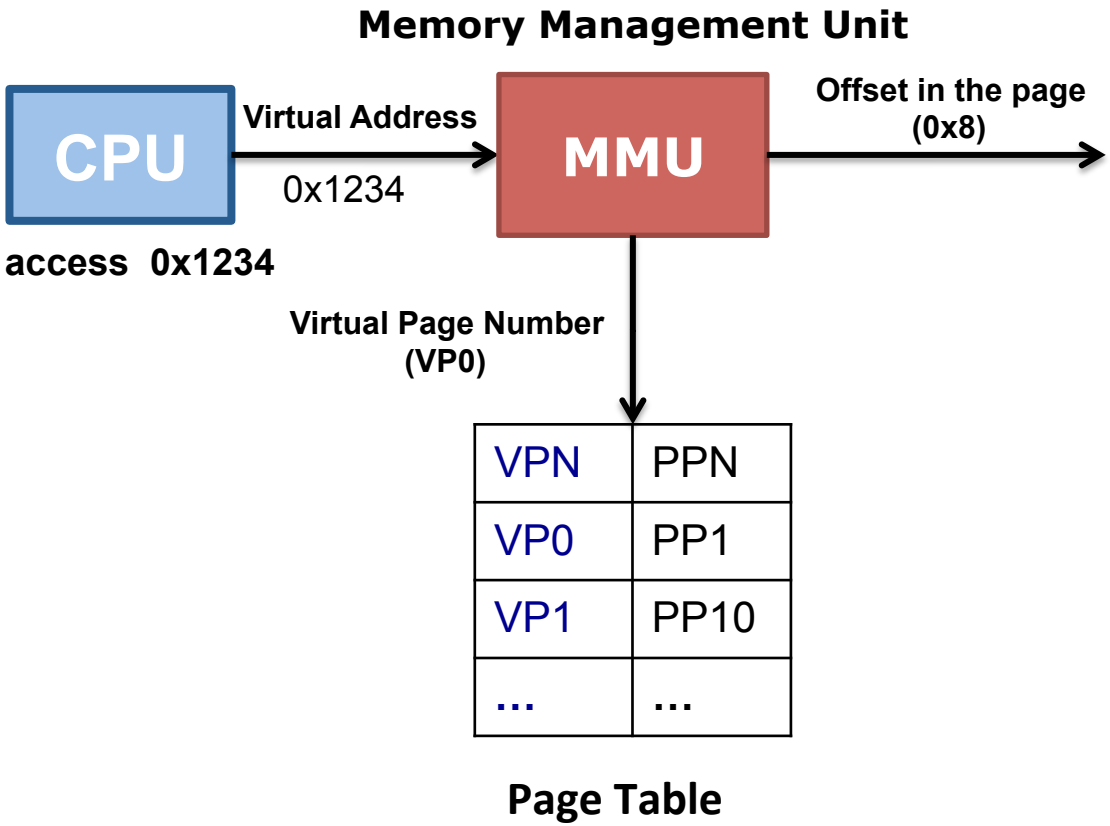
Address Translation – Page

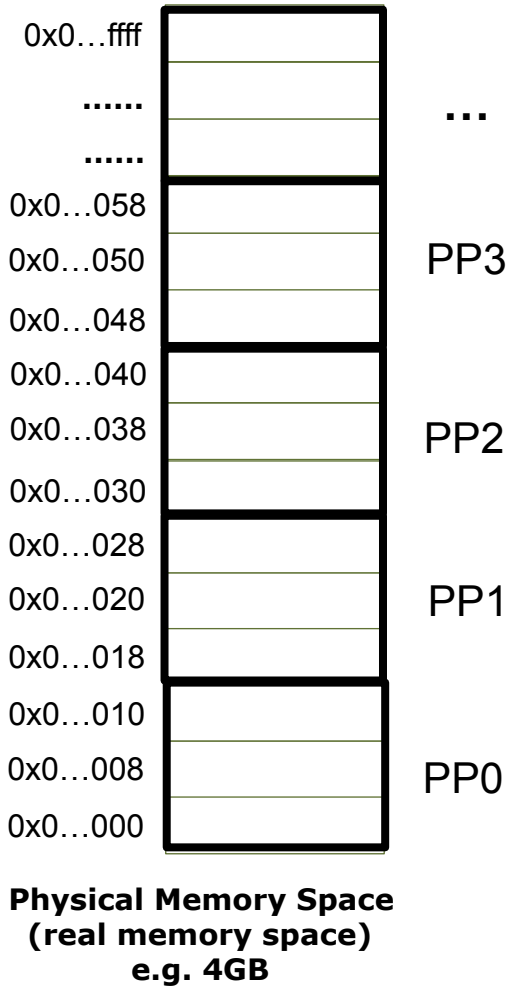


Virtual Memory Space (Conceptual memory space)

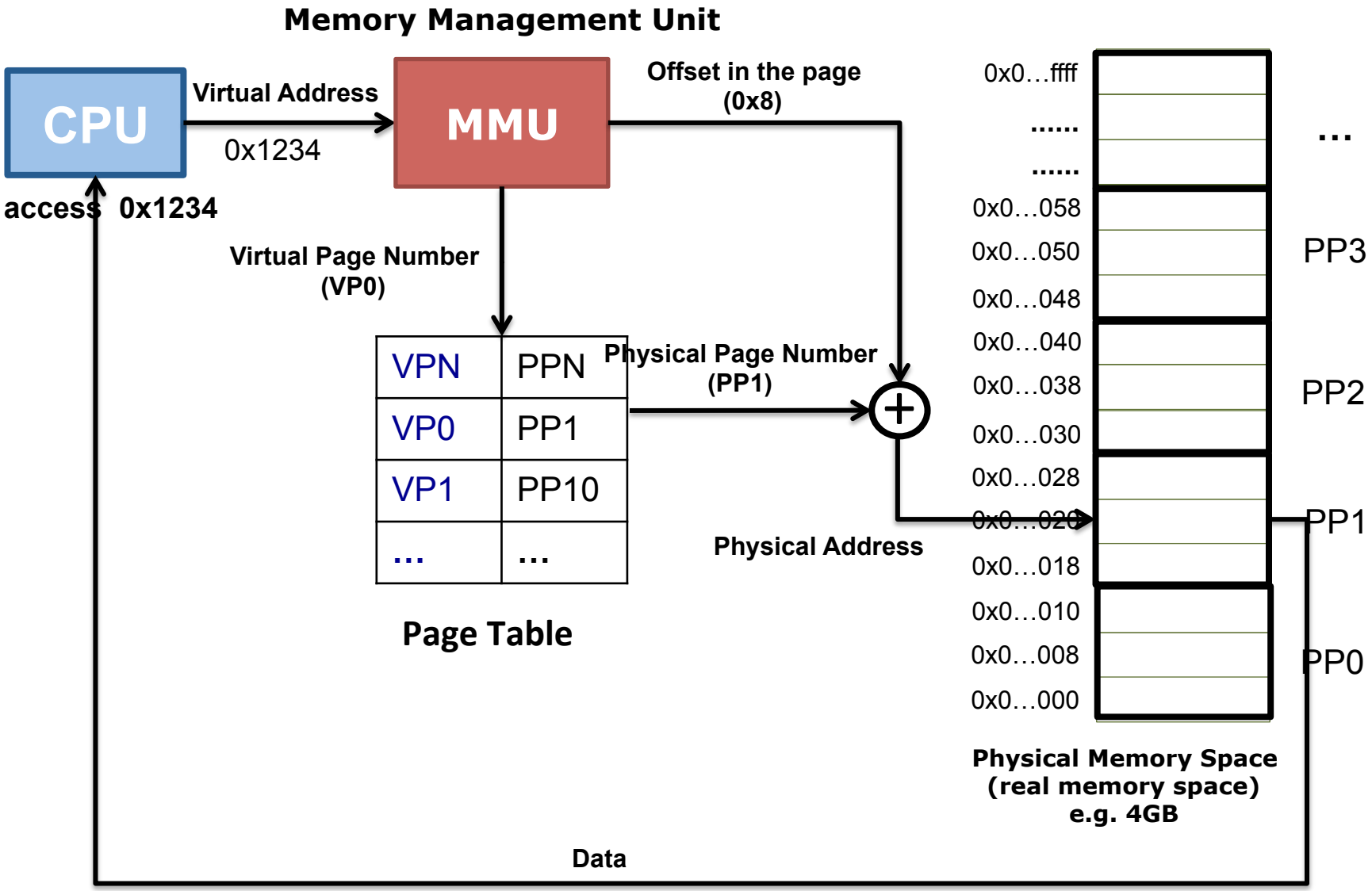
Physical Memory Space (real memory space) e.g. 4GB

Address Translation – Page

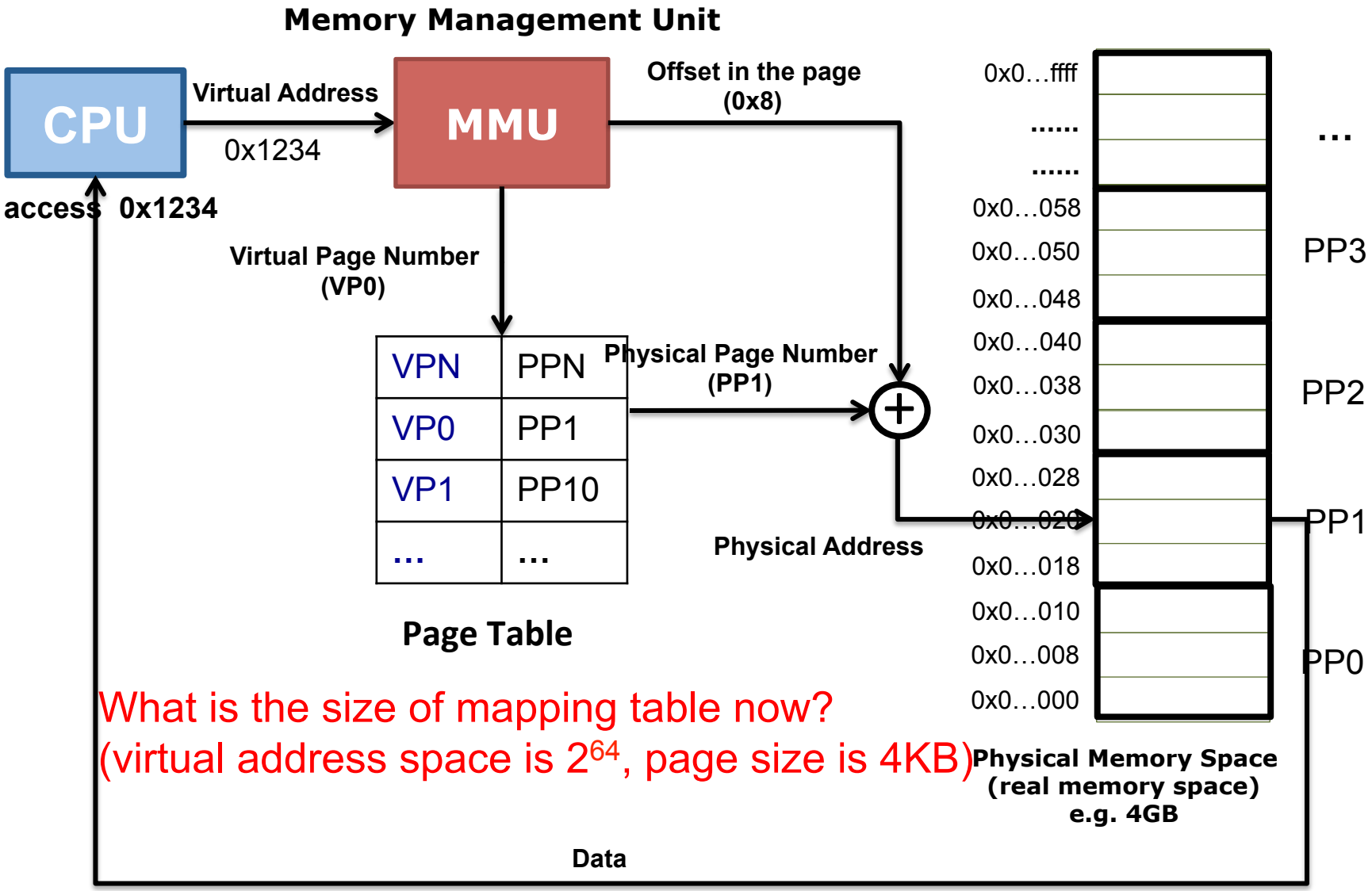




Address Translation – Page

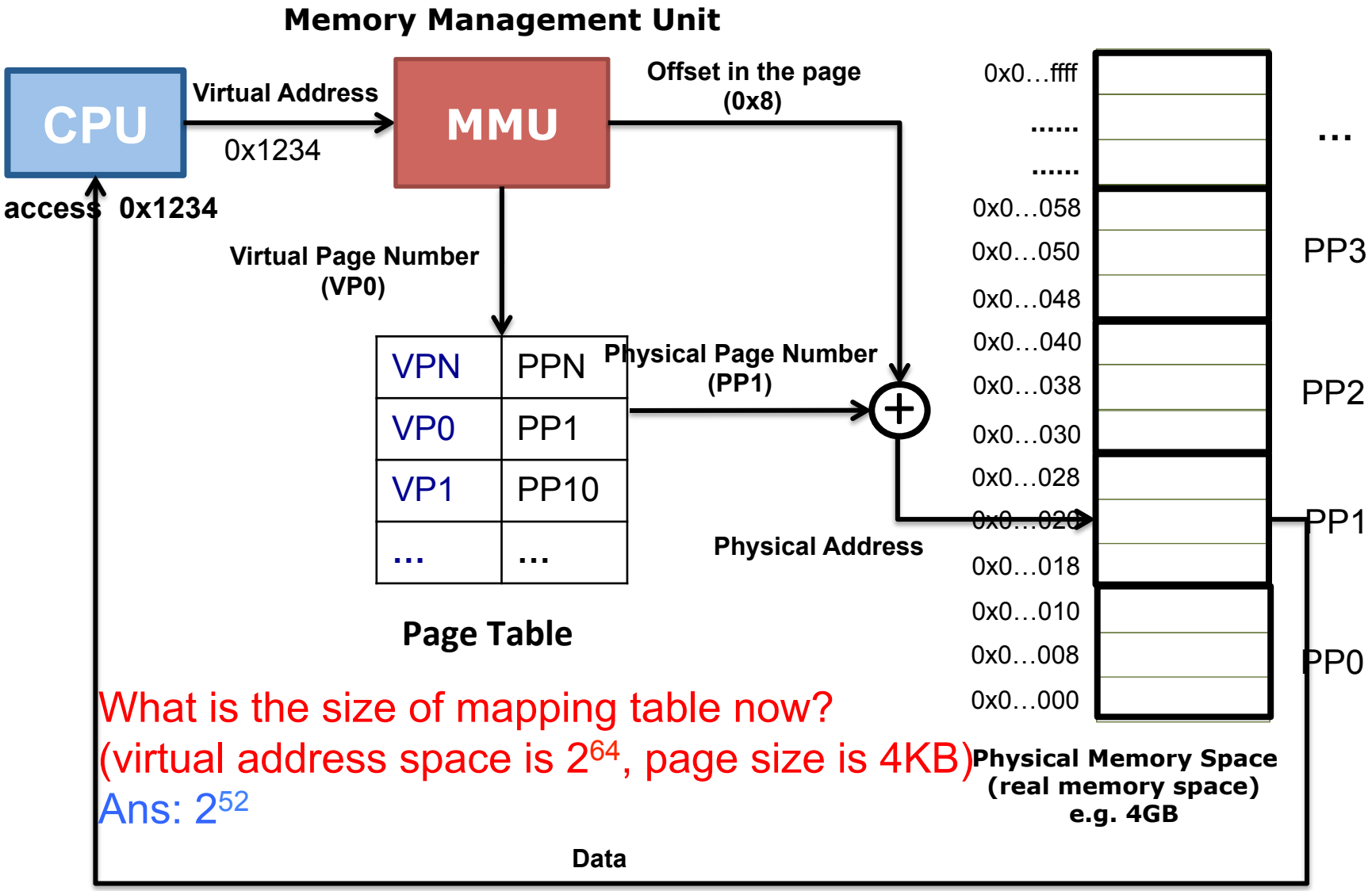


Address Translation – Page



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

Address Translation – Page



What is the size of mapping table now?
(virtual address space is 2^{64}, page size is 4KB)
Ans: 2^{52}

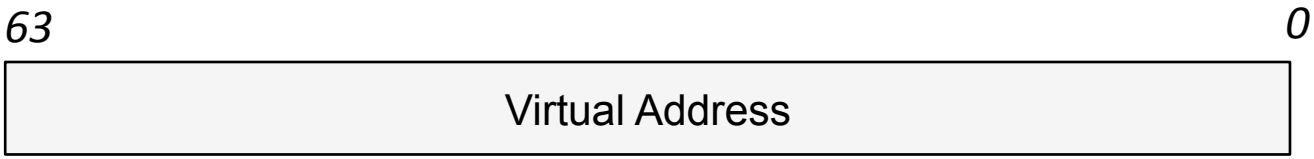
Address Translation

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})



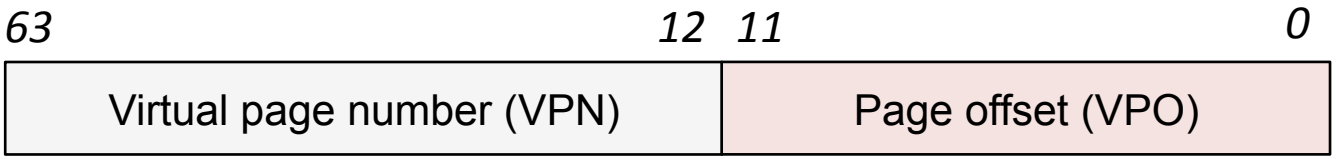
Address Translation

Virtual Address → Physical Address

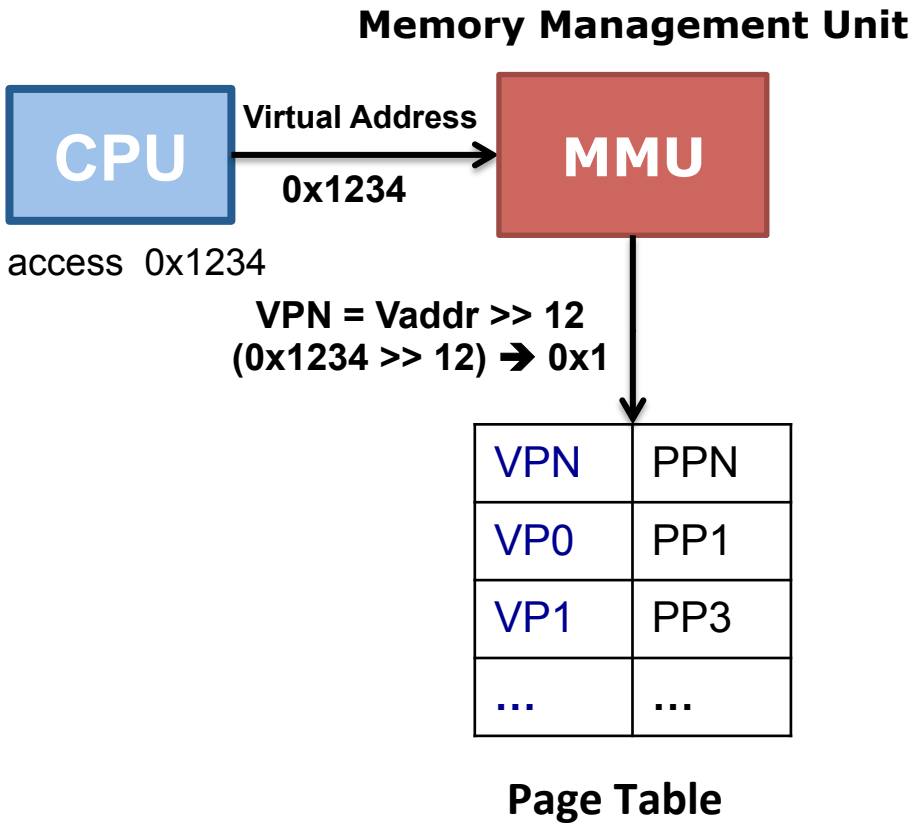
- Calculate the virtual page number
- Locate the data from the according physical page

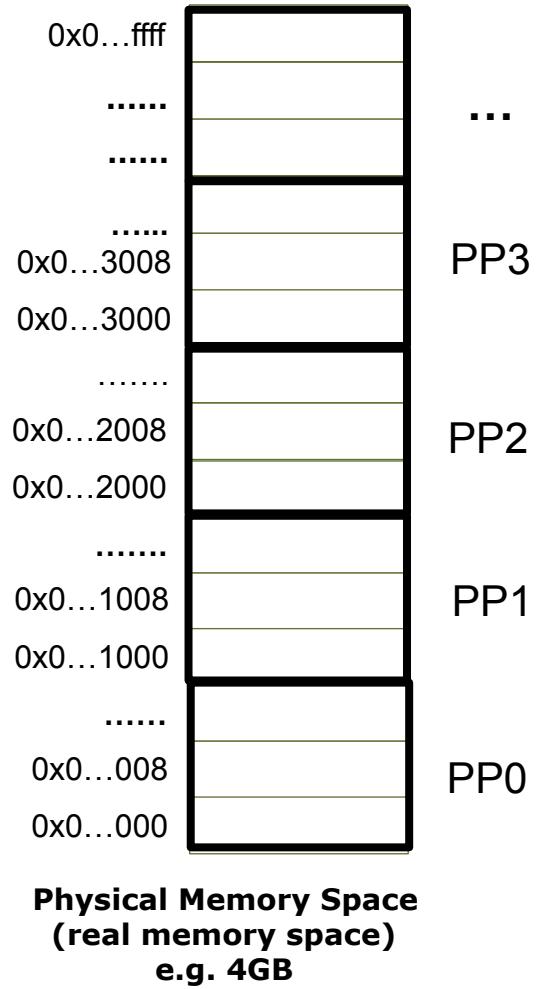
Memory address width: 64 bits

Page size: 4 KB (2^{12})

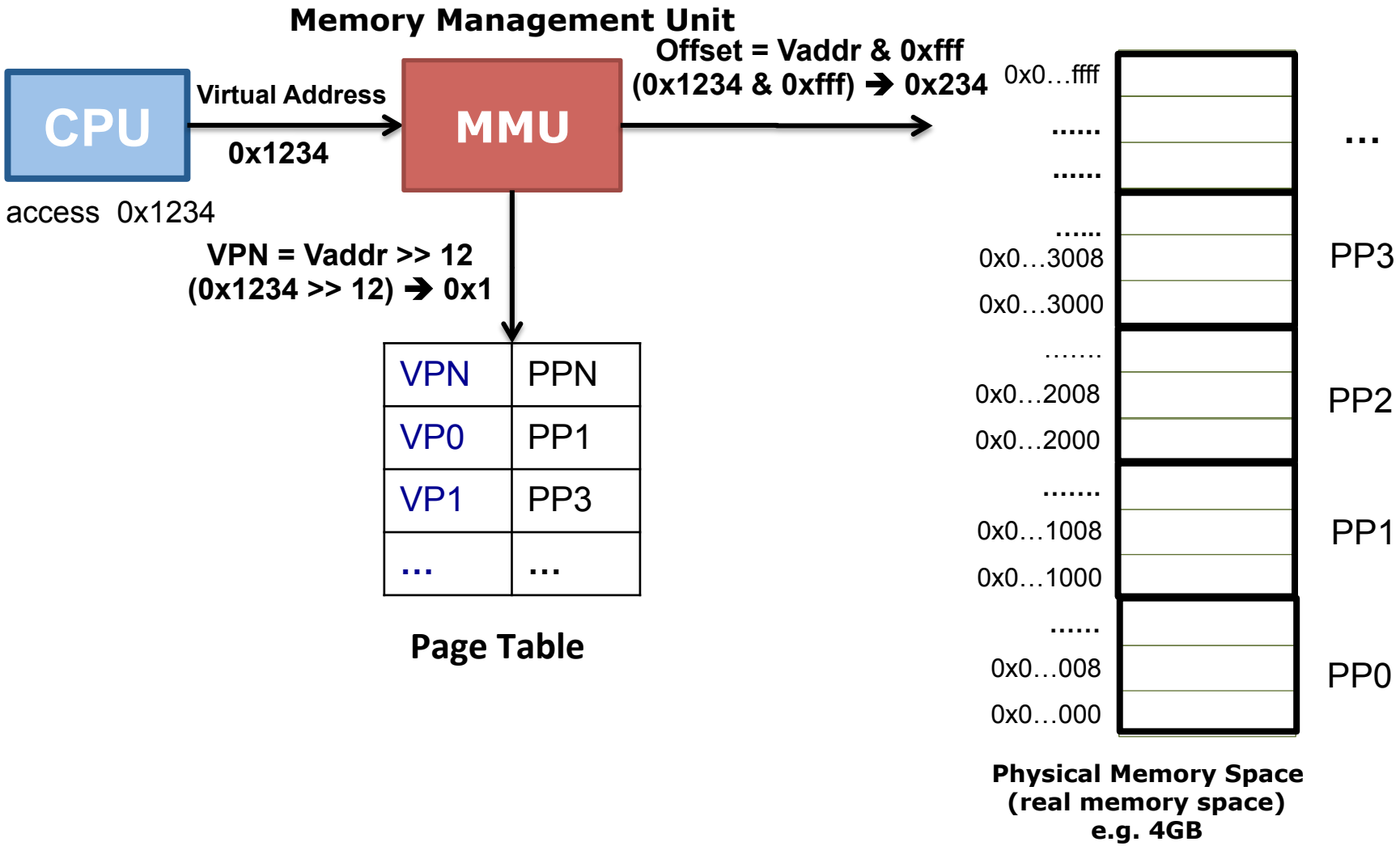


Address Translation

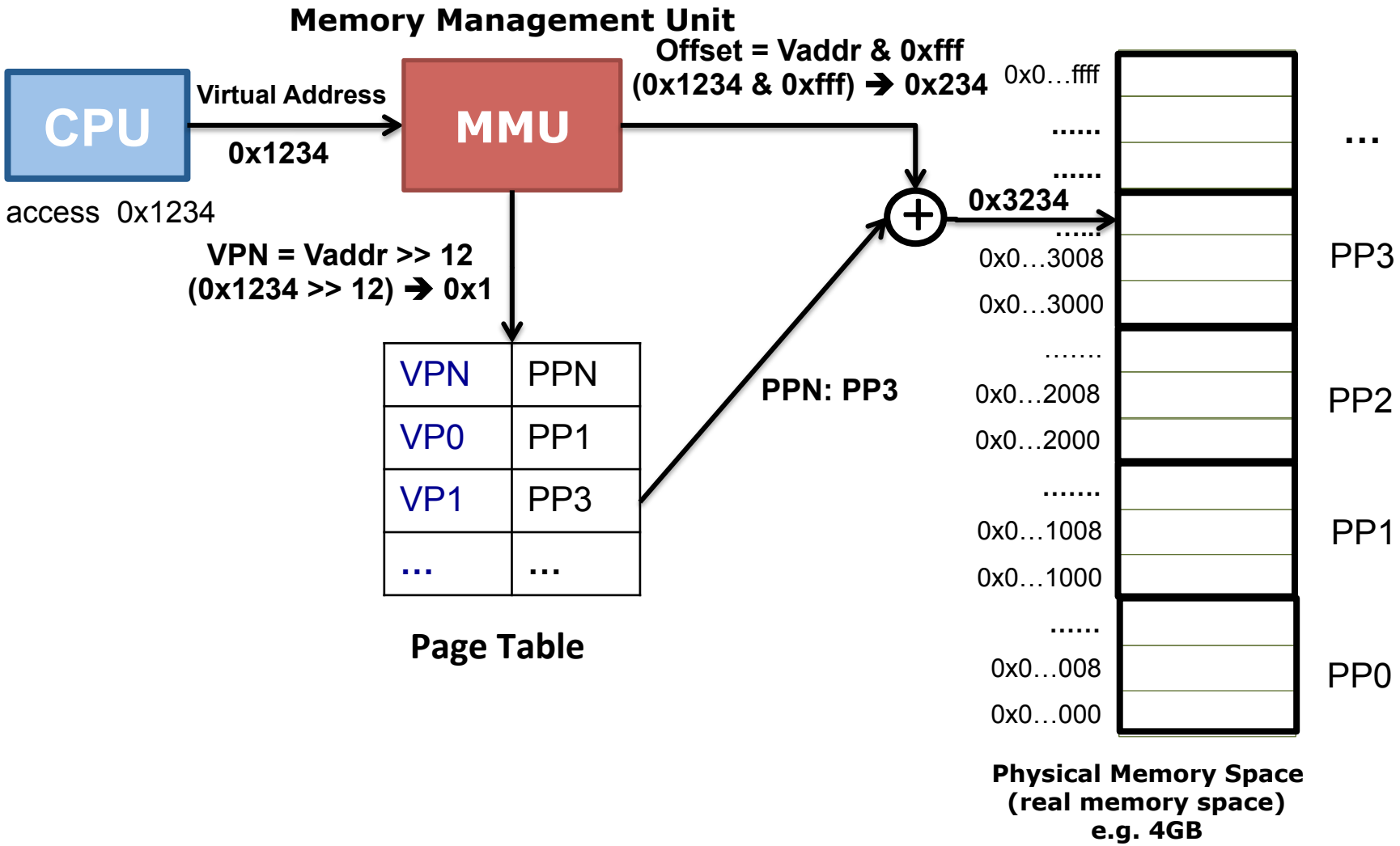




Address Translation



Address Translation



Exercise

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

Page Table

Page size: 4KB
Address width: 64

Virtual Address	Physical Address
0x1234	
0x4321	
0x5678	
0x2222	
0x4567	
0x5234	

Exercise

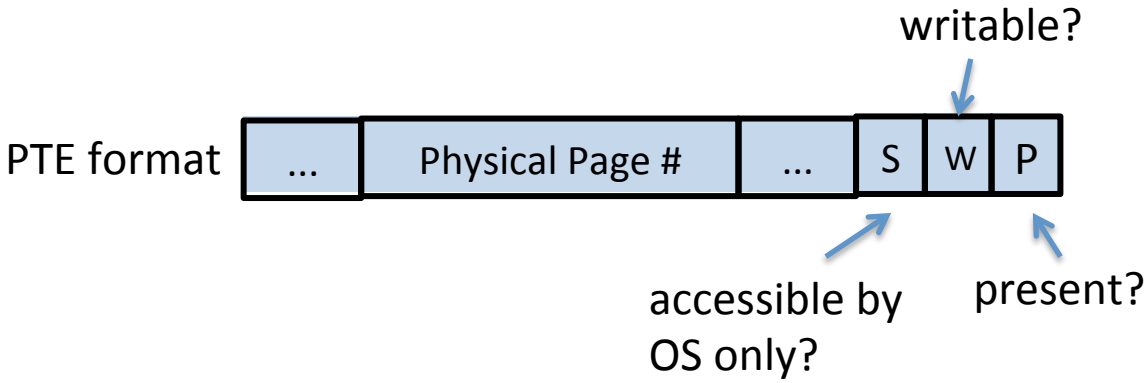
VPN	PPN
VP0	PP1
VP1	PP3
VP2	PP4
VP4	PP0
VP5	PP3
VP6	PP2

Page Table

Page size: 4KB
Address width: 16

Virtual Address	Physical Address
0x1234	0x3234
0x4321	0x0321
0x5678	0x3678
0x2222	0x4222
0x4567	0x0567
0x5234	0x3234

Page table entries encode permission information



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

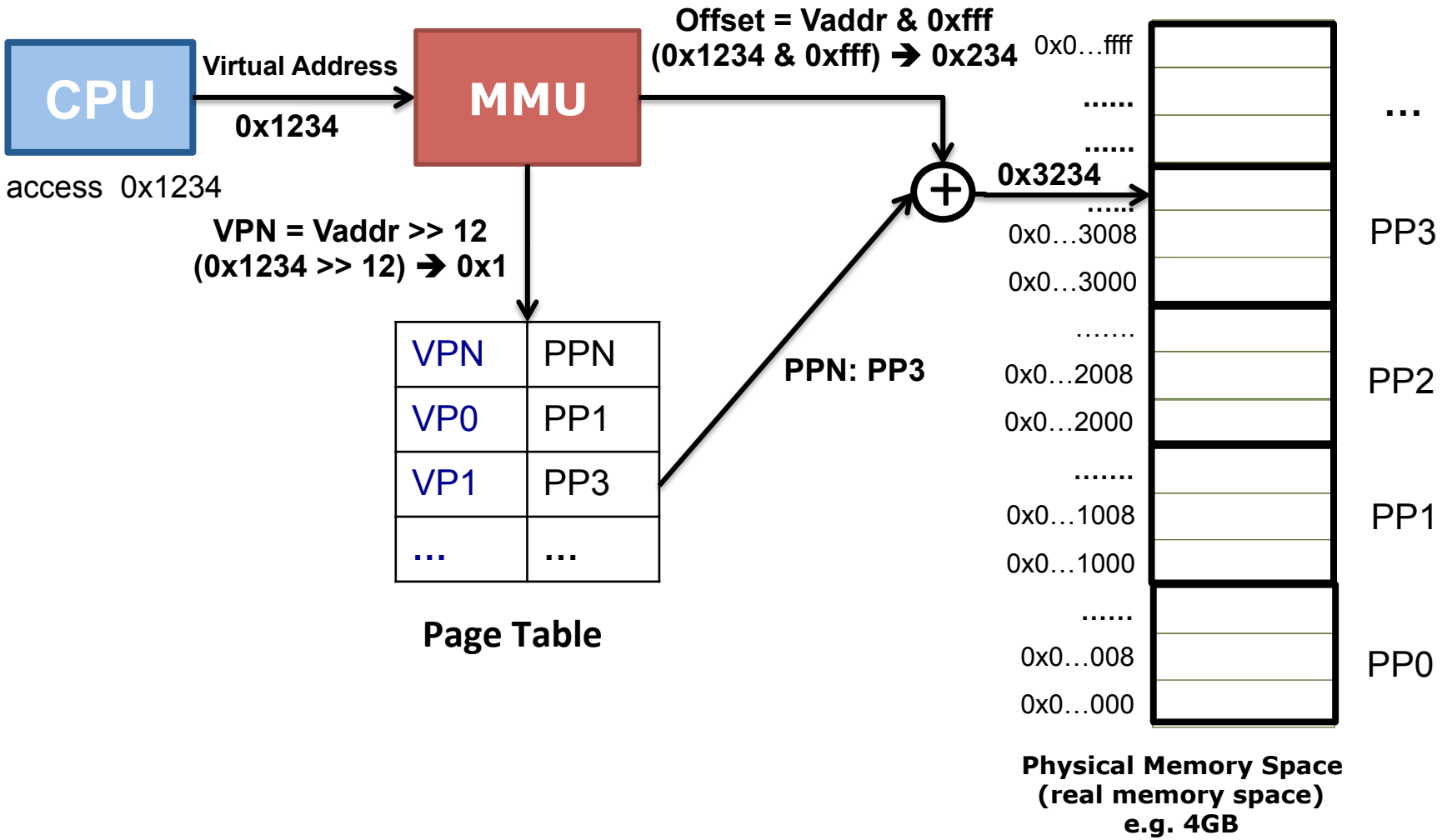
Conceptual Page Table

pt[0]	8-byte Page Table Entry (PTE)
pt[1]	...
pt[2]	...
pt[3]	...
pt[4]	...
pt[5]	...

Actual Page Table

Question: how many PTEs per page? 4KB/8 = 2^12/2^3 = 2^9

What we have learnt: VM memory translation



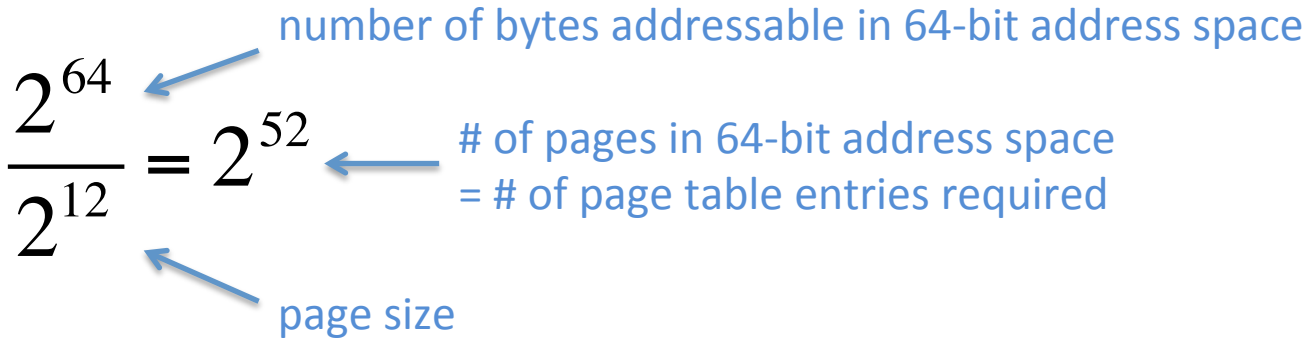
This lecture

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



Multi-level page tables

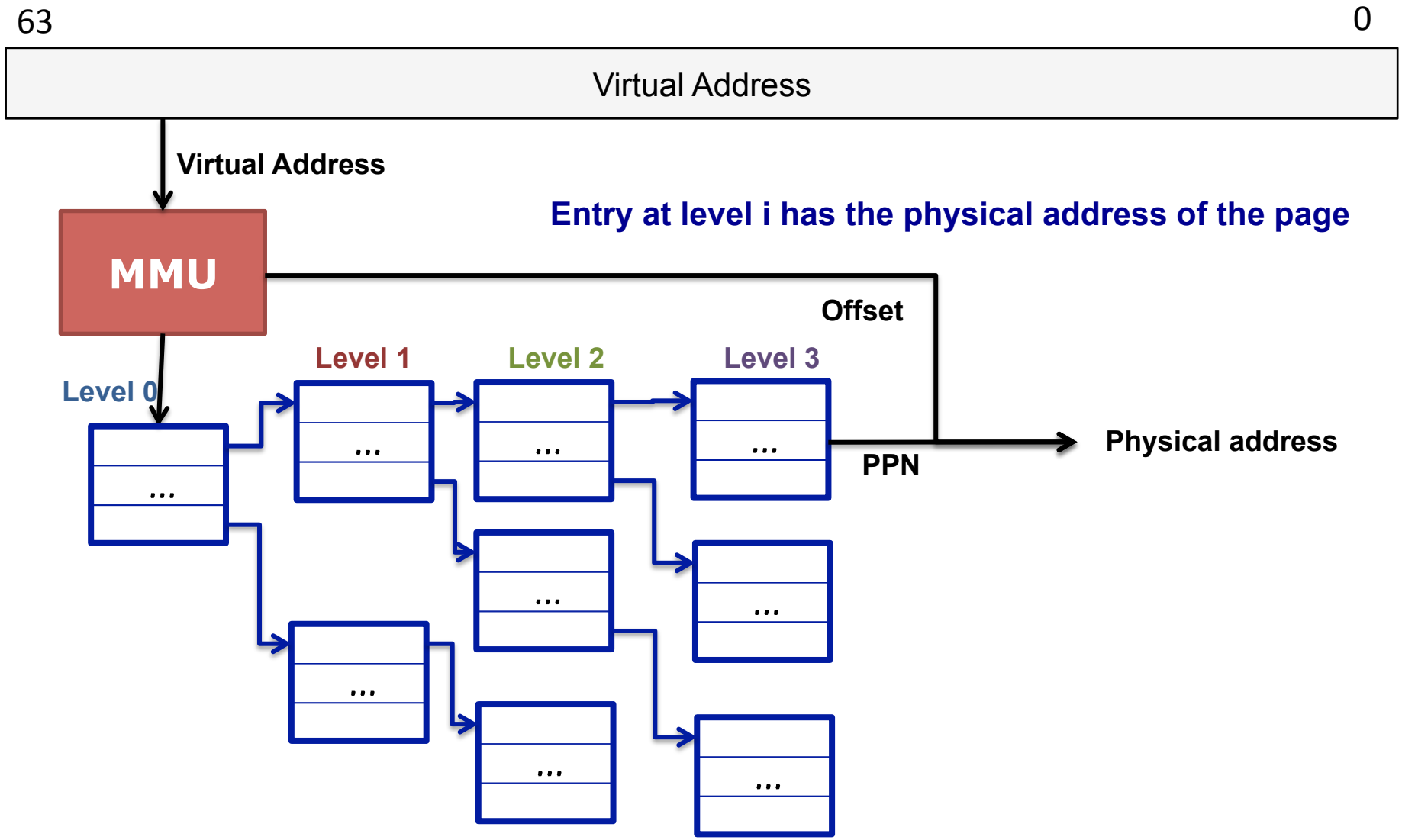
Problem

- how to reduce # of page table entries required?

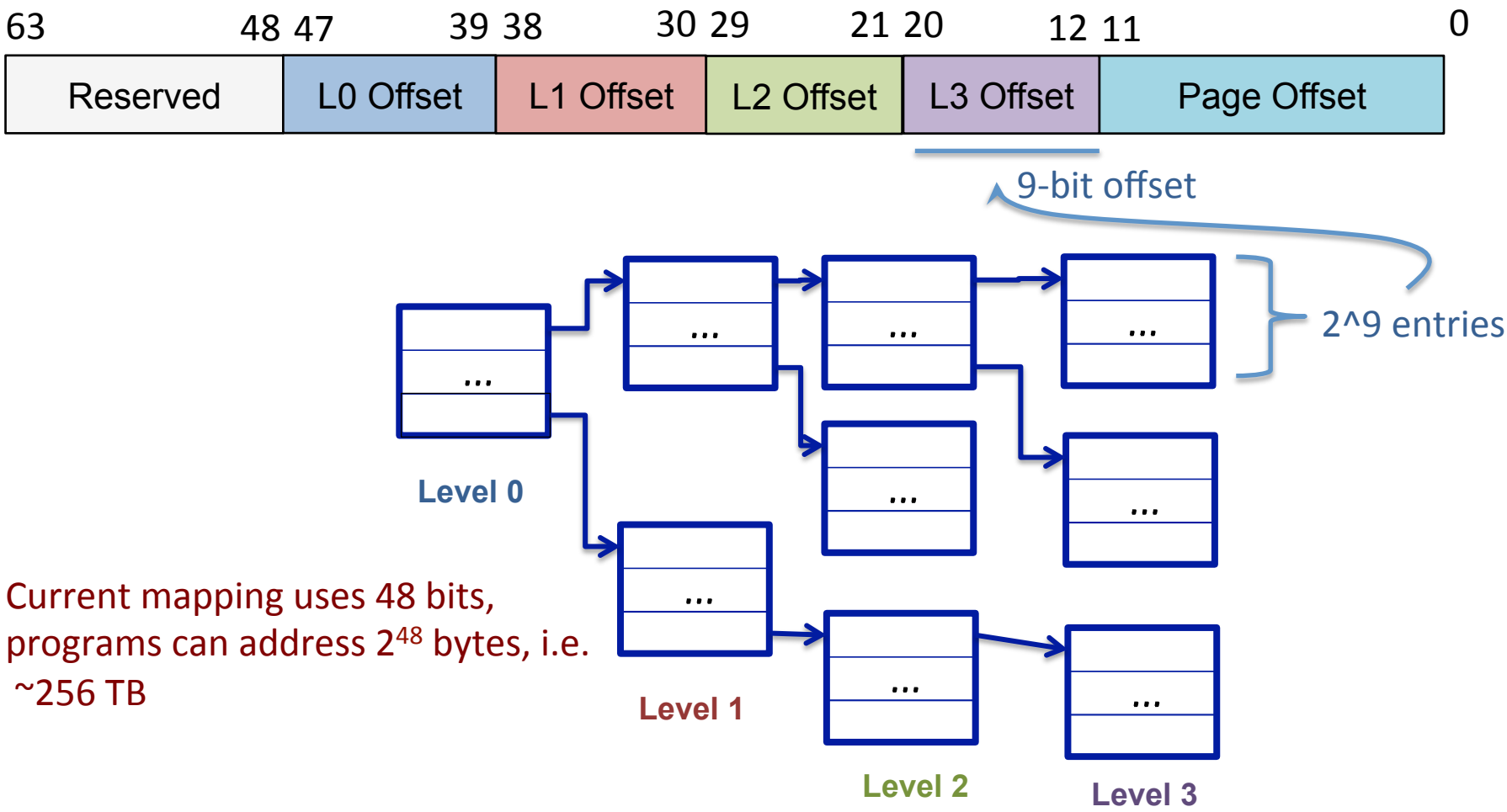
Solution

- Multi-level page table
  - x86-64 supports 4-level page table

Multi-level page tables on X86_64



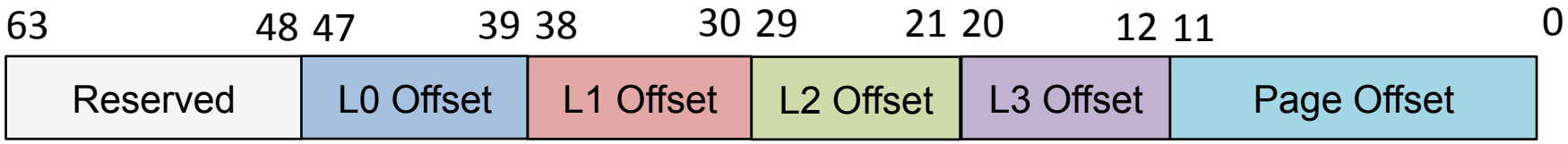
Multi-level page tables on X86_64

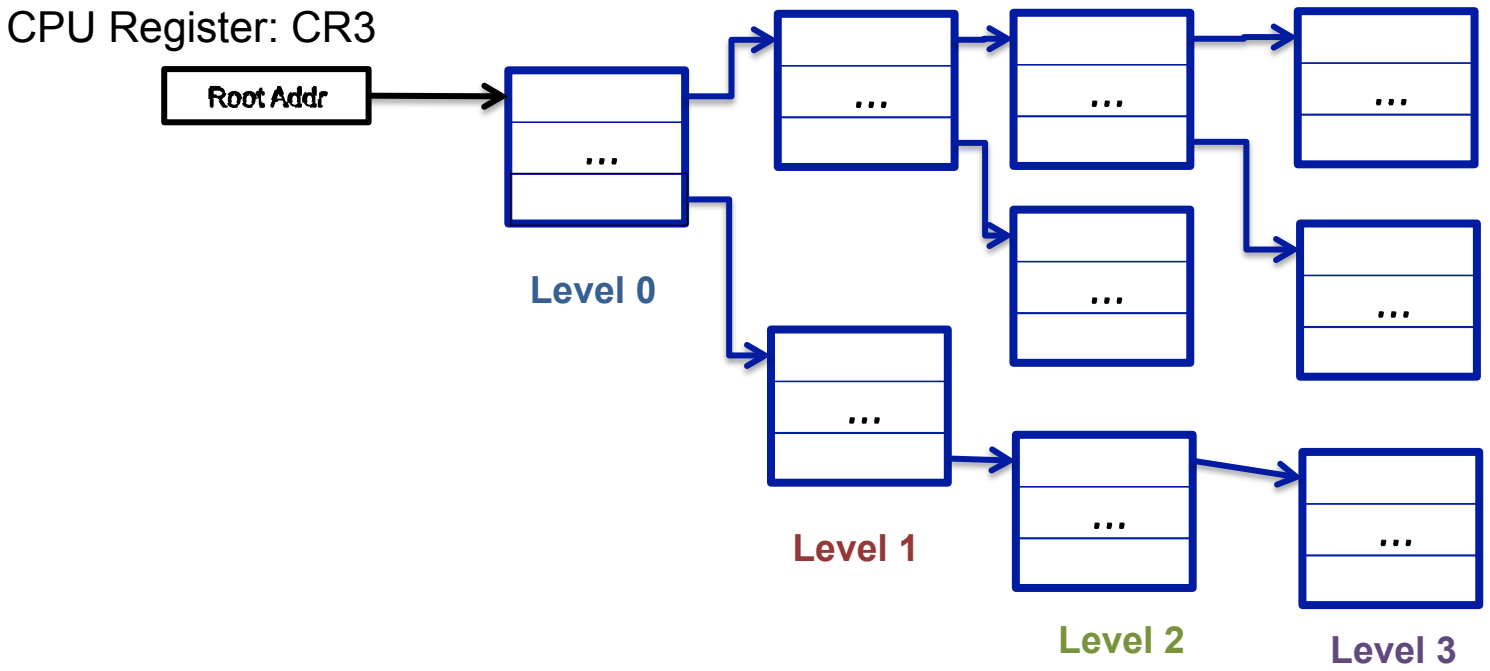


Current mapping uses 48 bits, programs can address 2^{48} bytes, i.e. ~256 TB

4-level page table

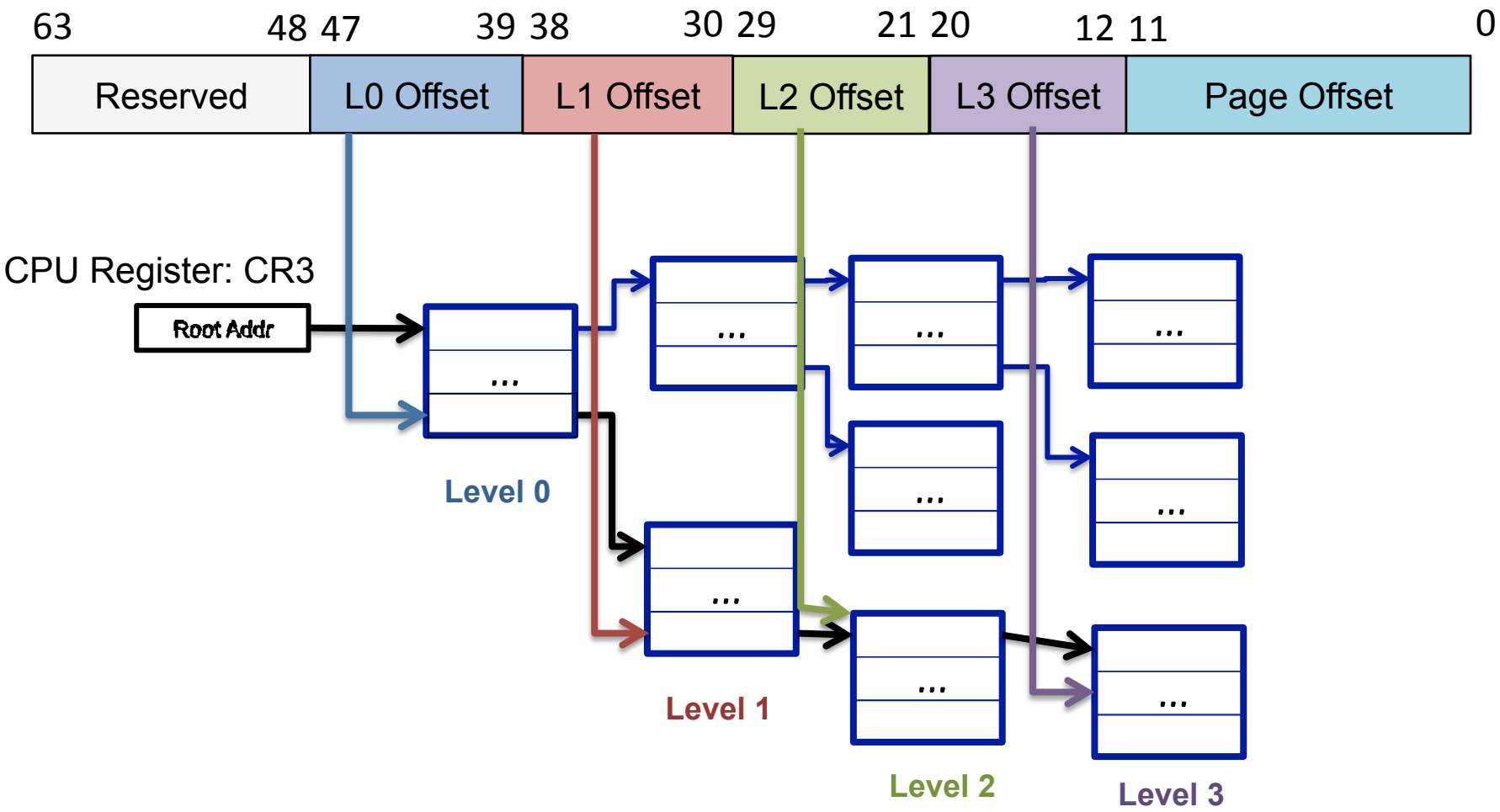
Multi-level page tables on X86_64





4-level page table

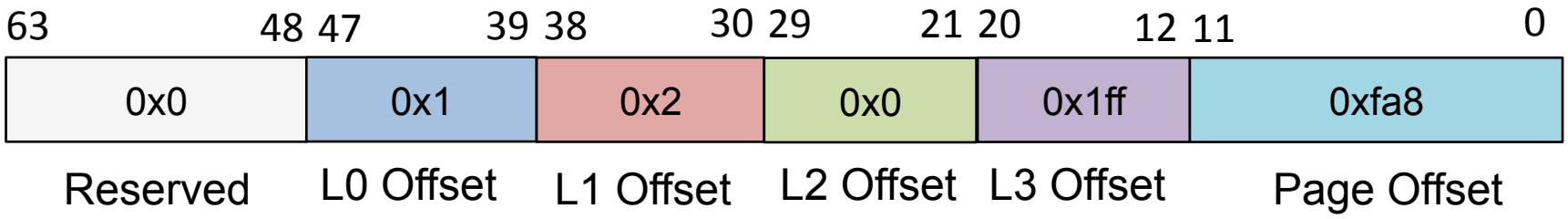
Multi-level page tables on X86_64



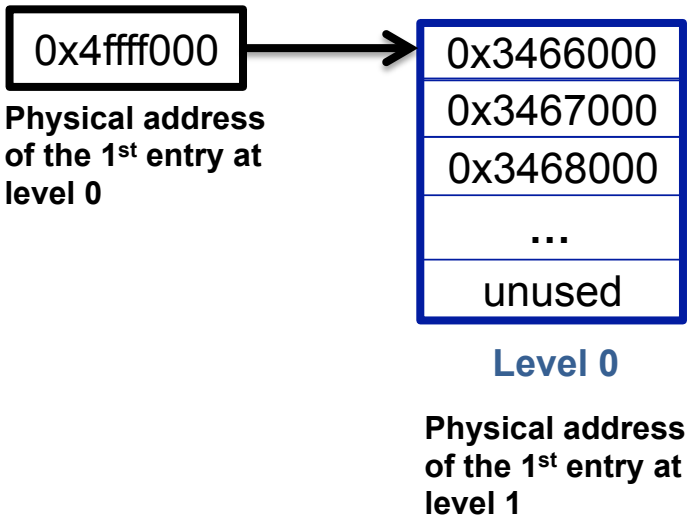
4-level page table

Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*

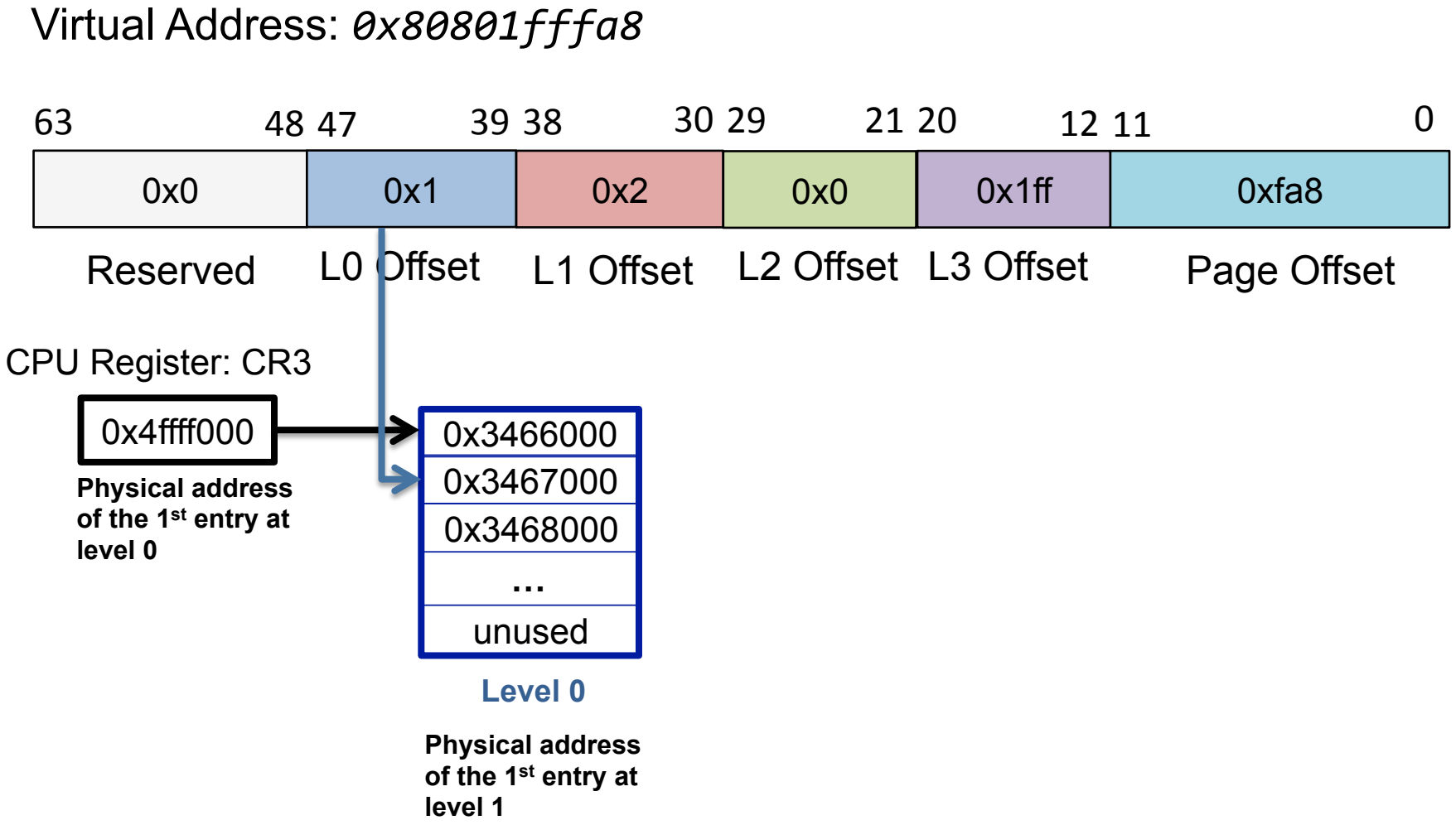


CPU Register: CR3



4-level page table

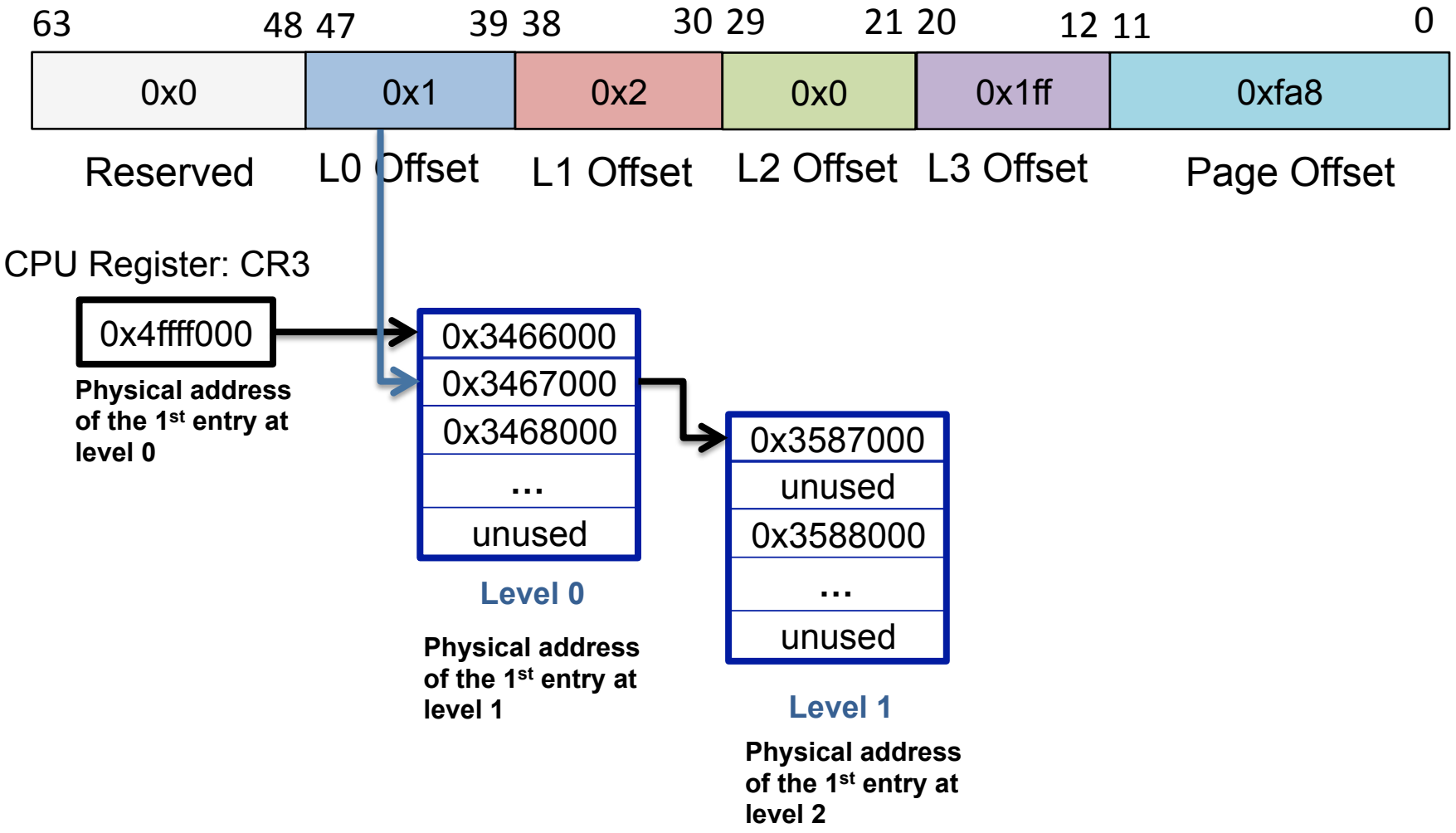
Multi-level page tables on X86_64



4-level page table

Multi-level page tables on X86_64

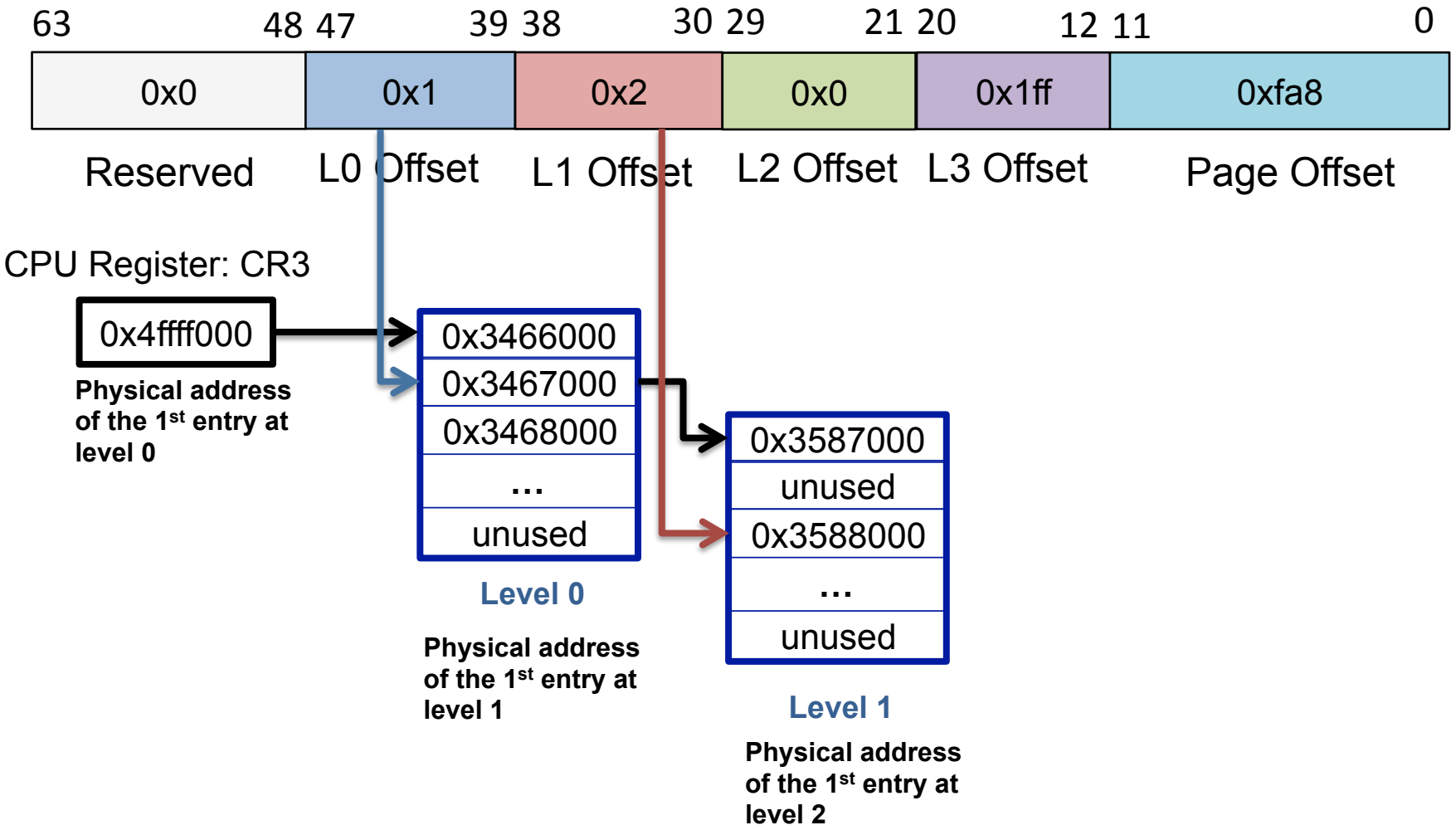
Virtual Address: *0x80801fffa8*



4-level page table

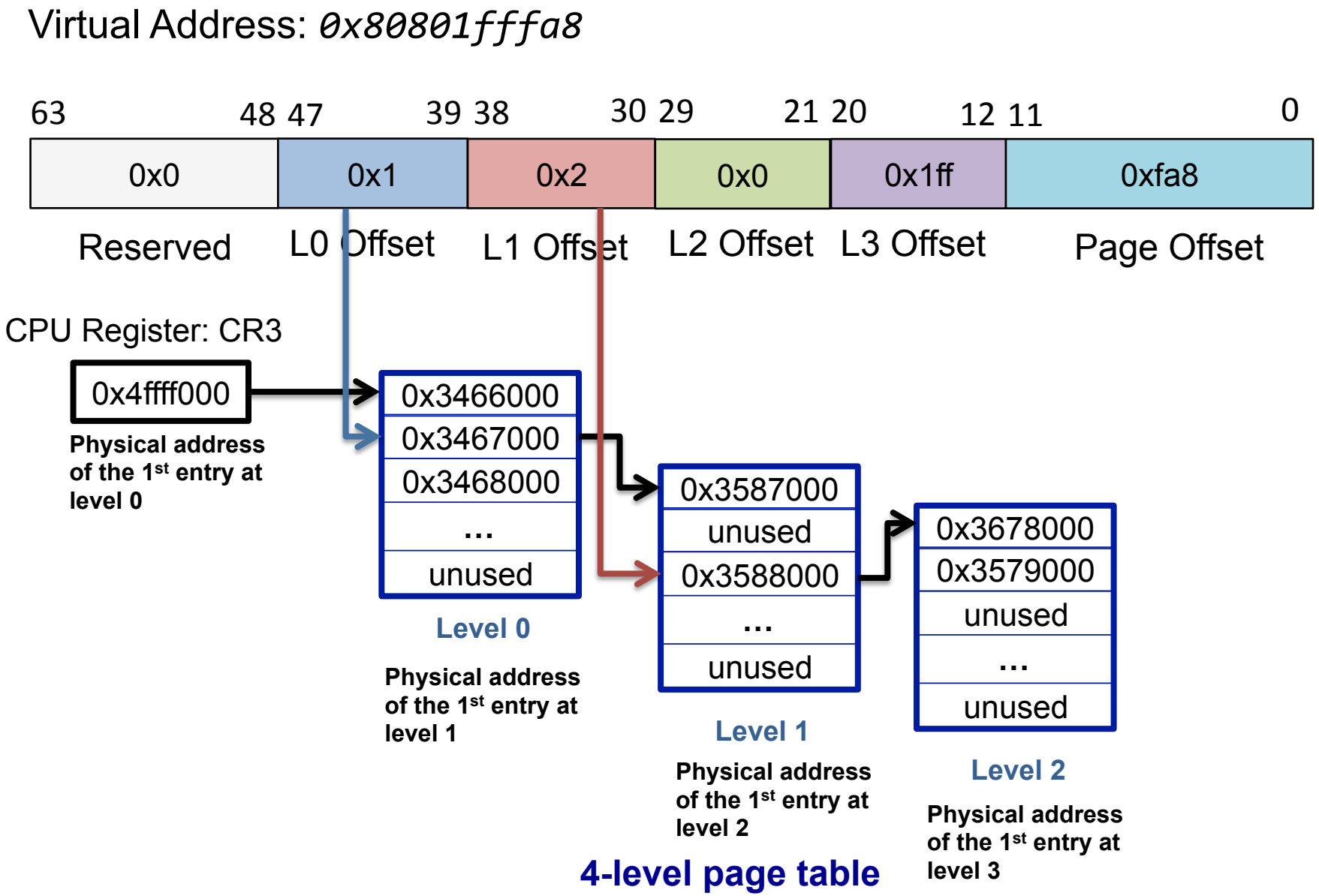
Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*



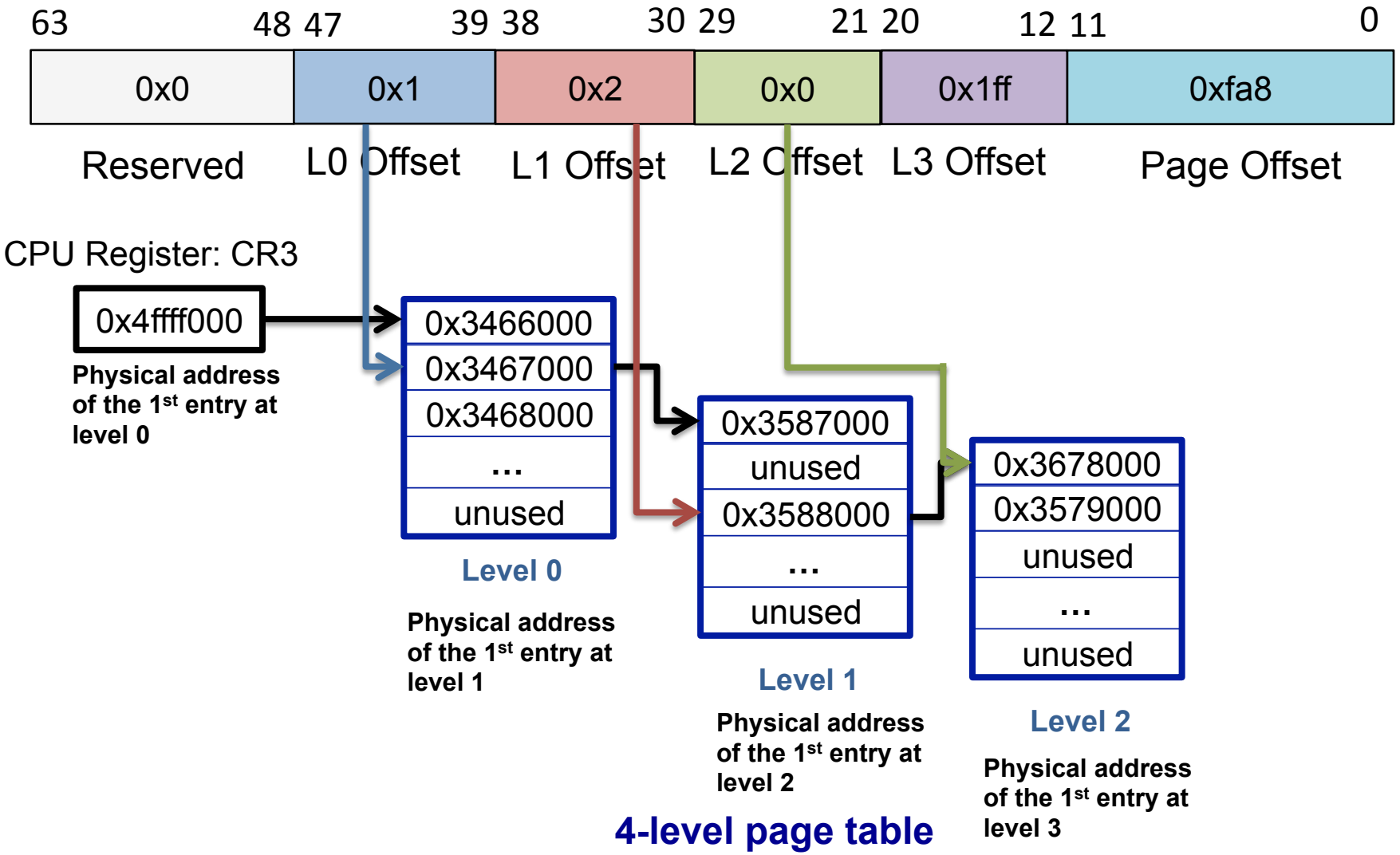
4-level page table

Multi-level page tables on X86_64



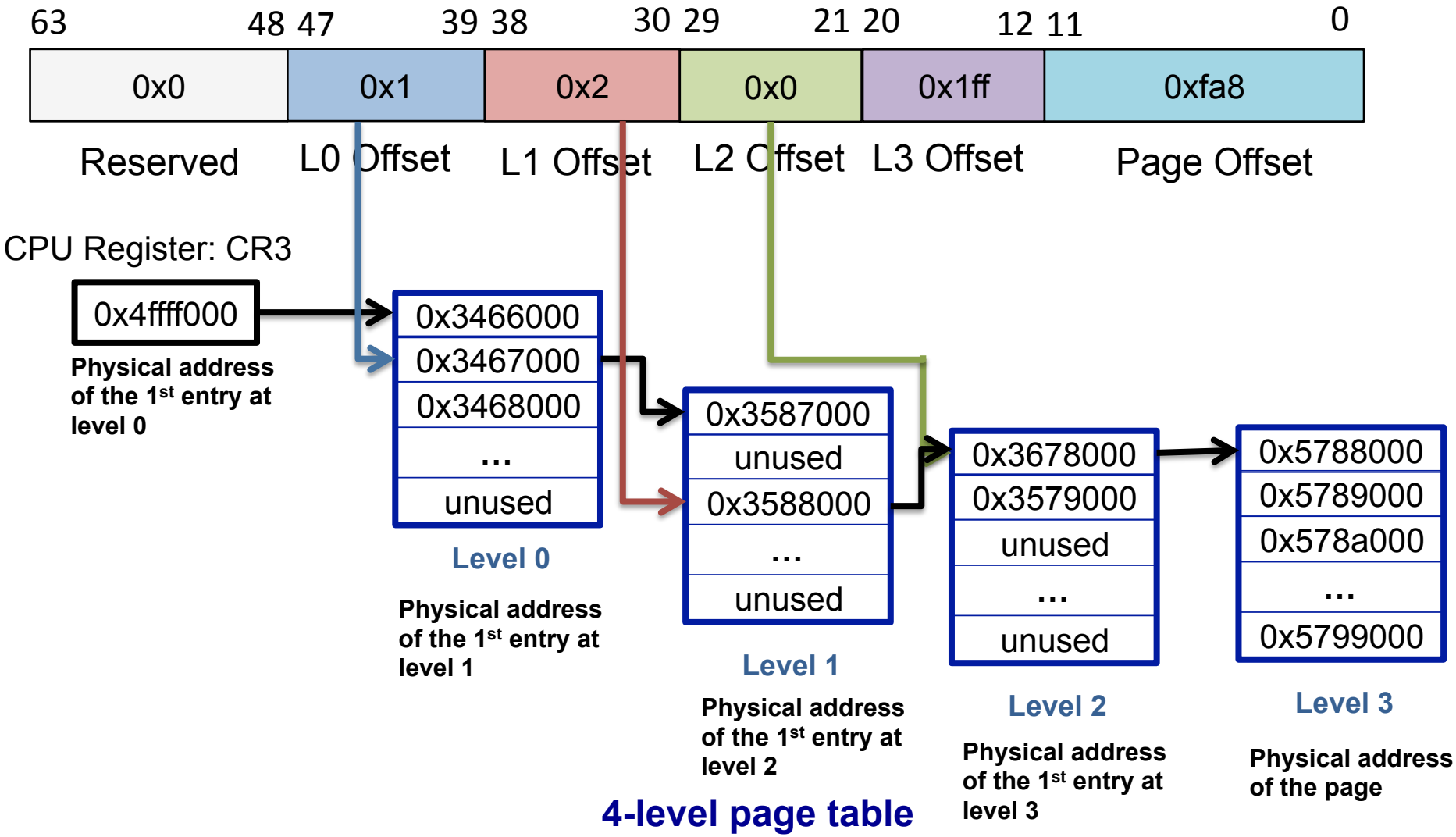
Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*



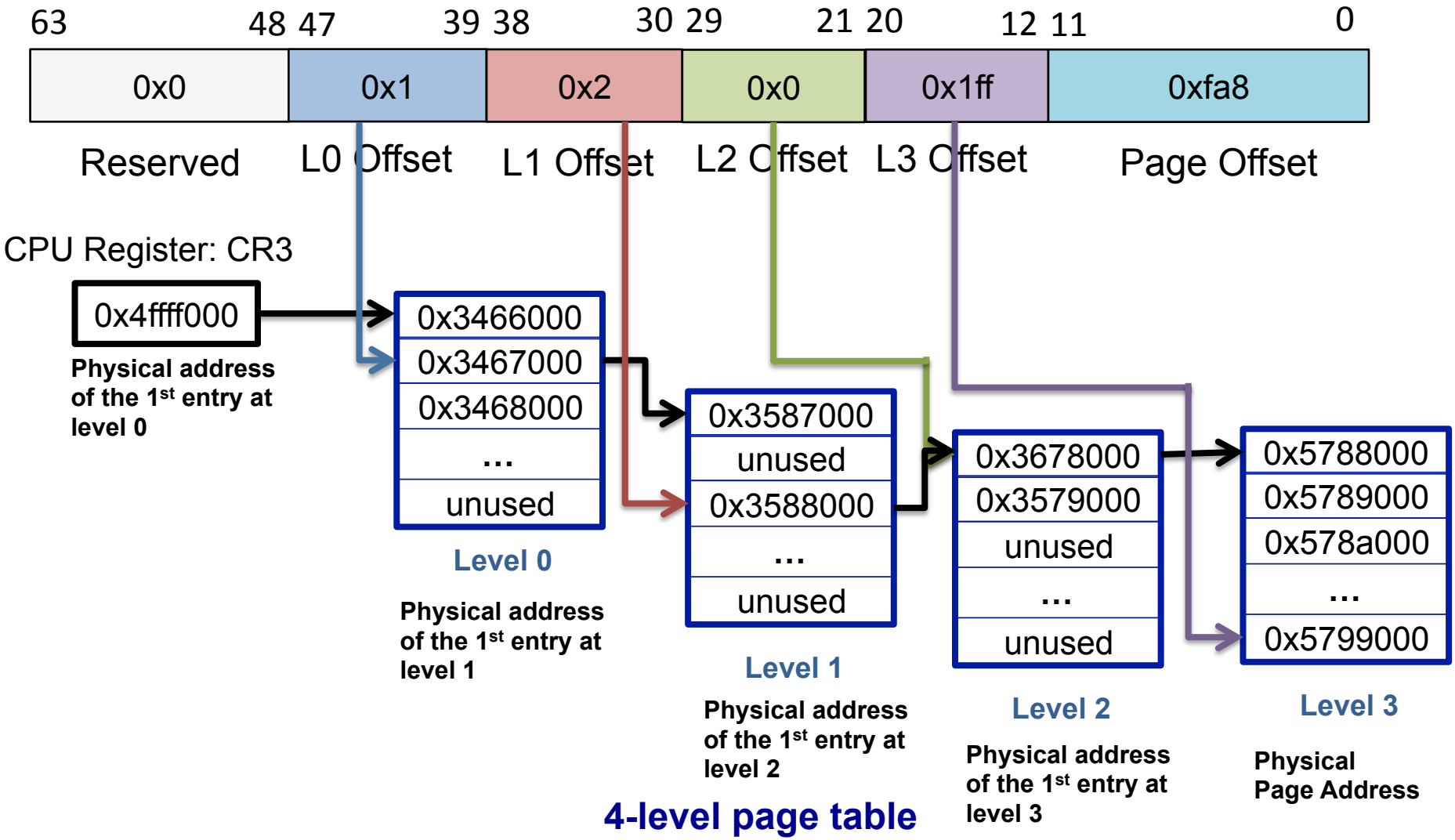
Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*

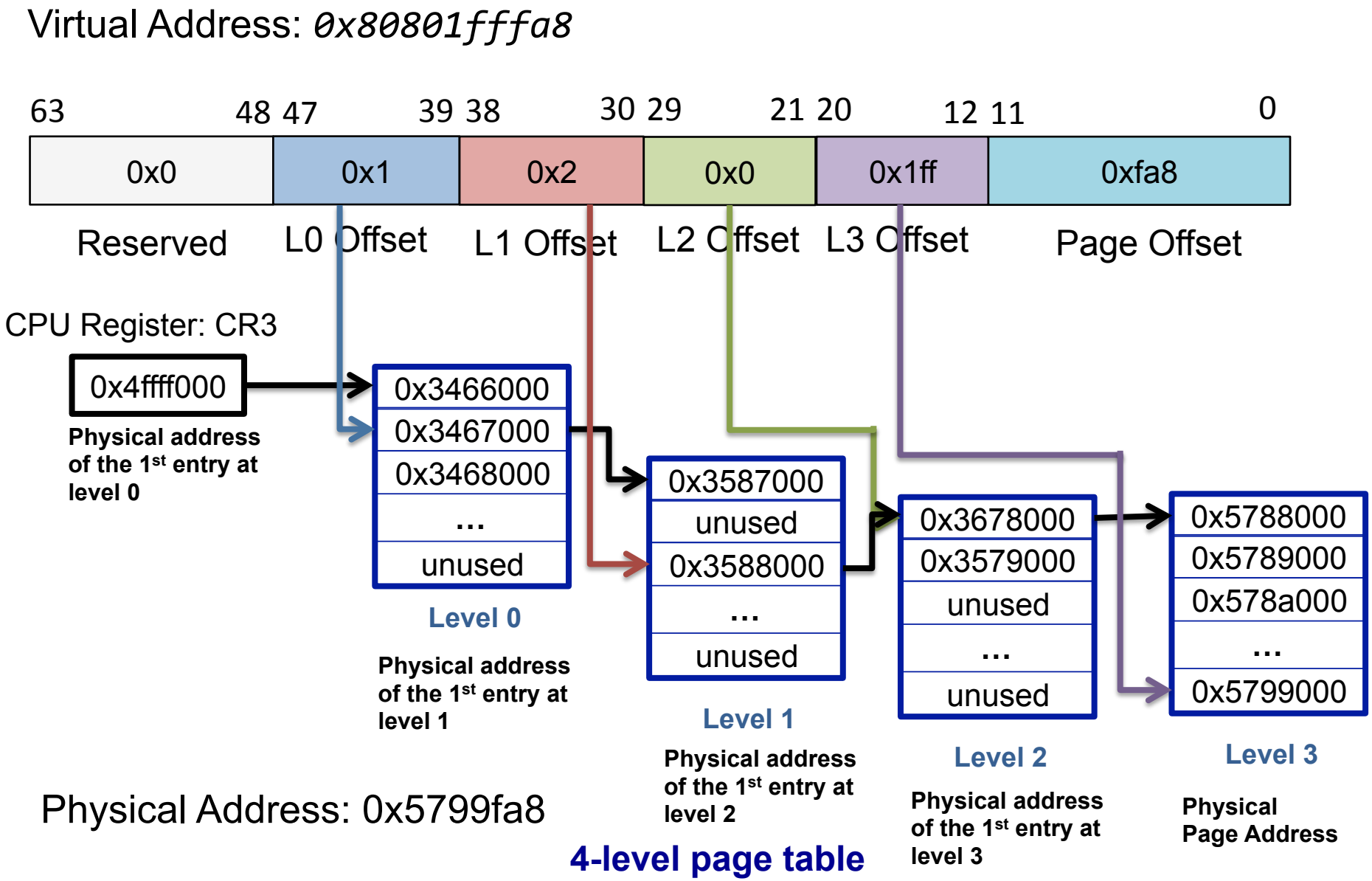


Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*



Multi-level page tables on X86_64

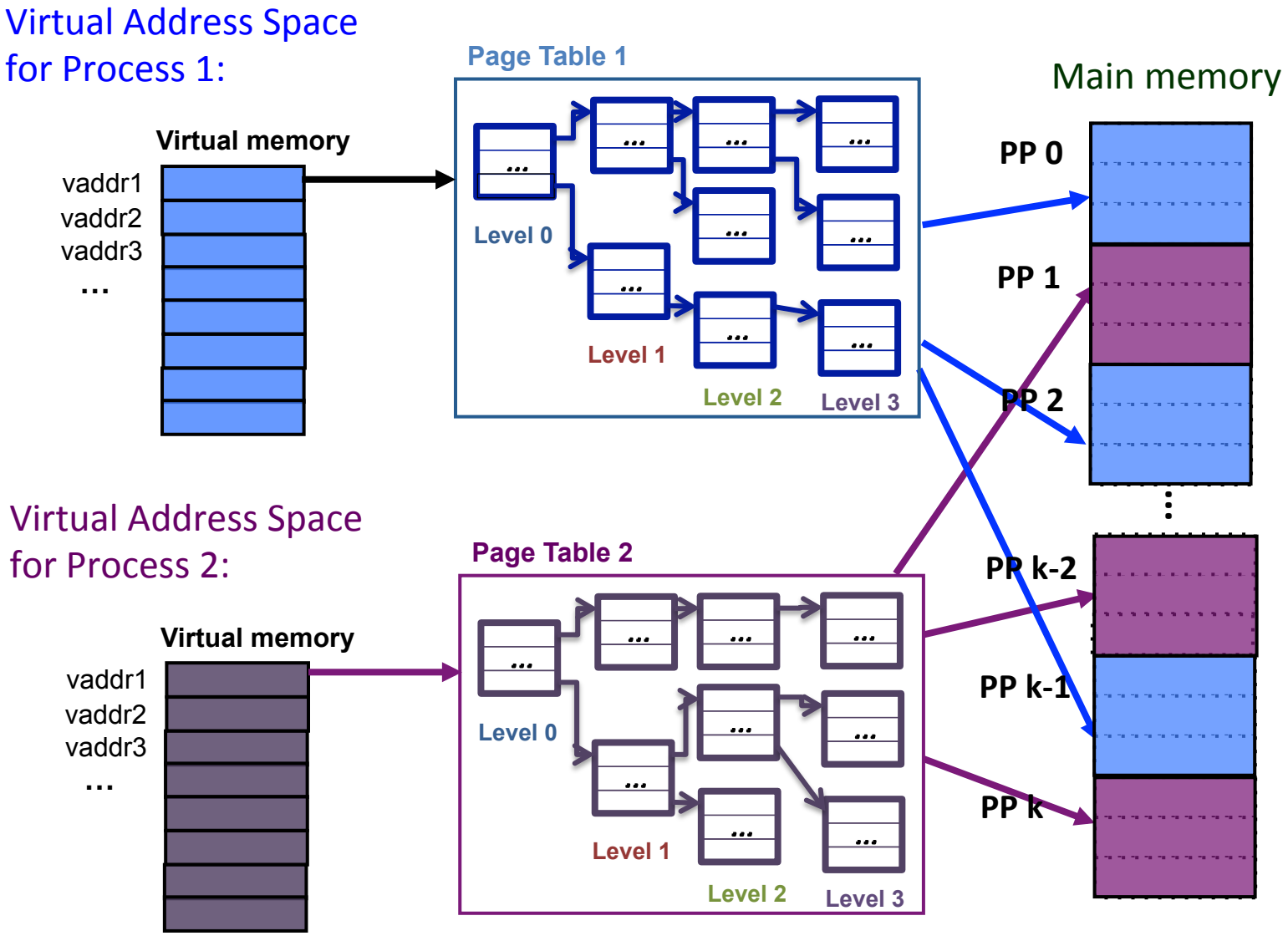


Review Virtual Address

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: 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 = sbrk(8192))`

User program to OS:

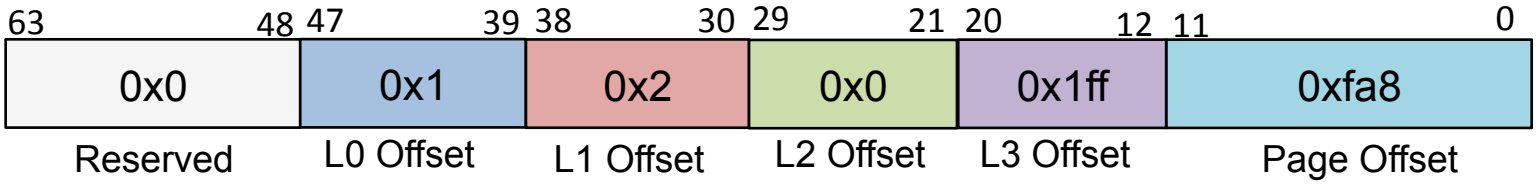
Declare a virtual address range from p to p + 8192 for use by the current process.

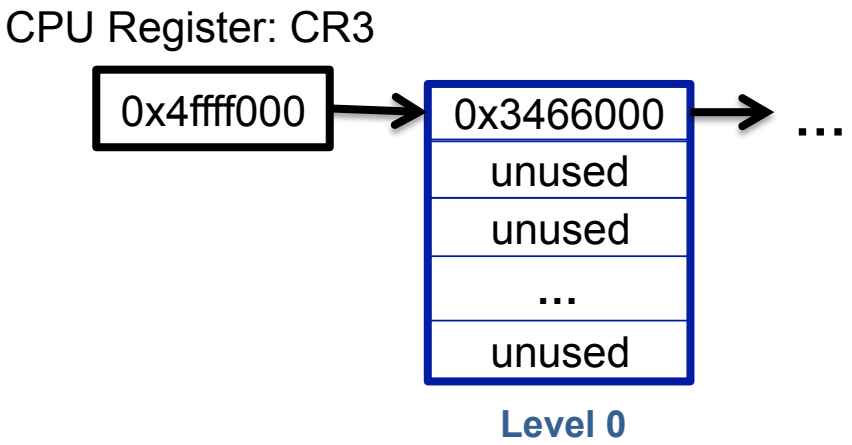
OS' actions:

Allocate the physical page and populate the page table.

Demand Paging

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



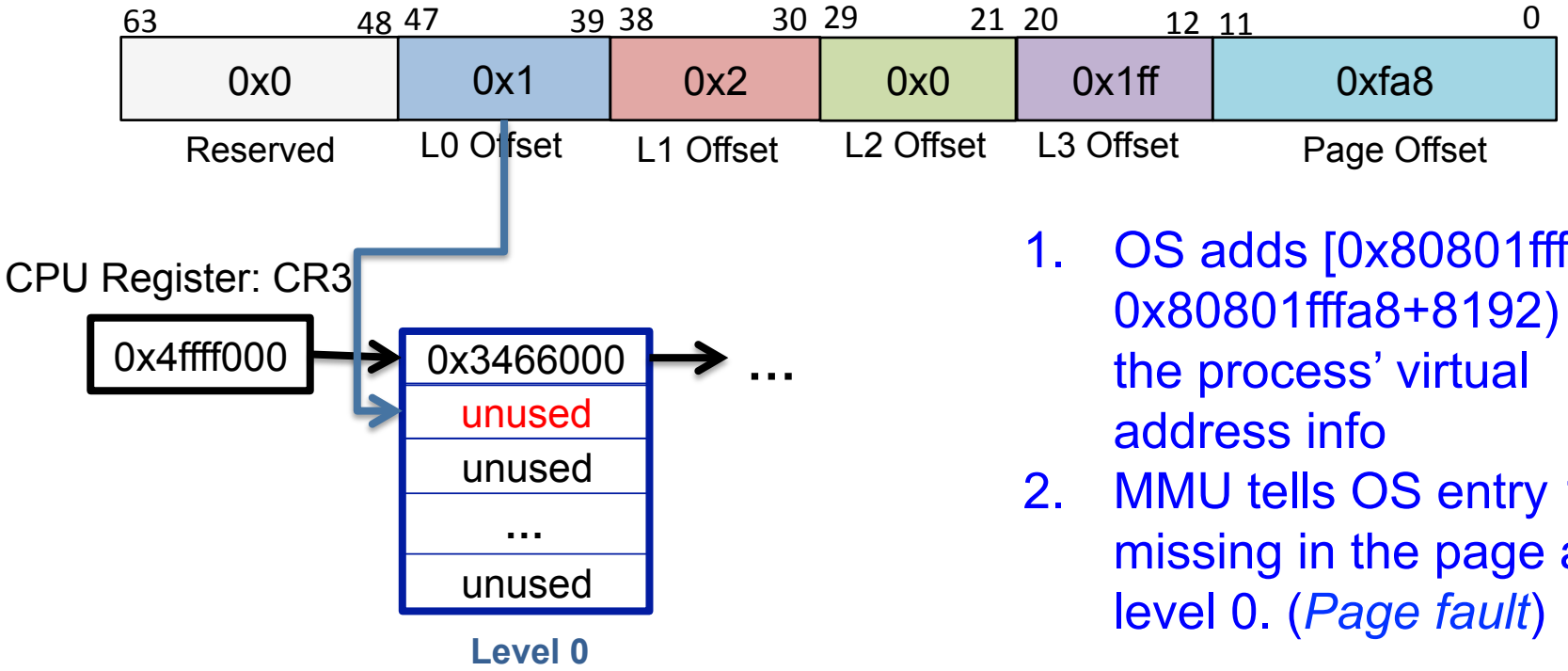


1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process' virtual address info

current process' page table

Demand Paging

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

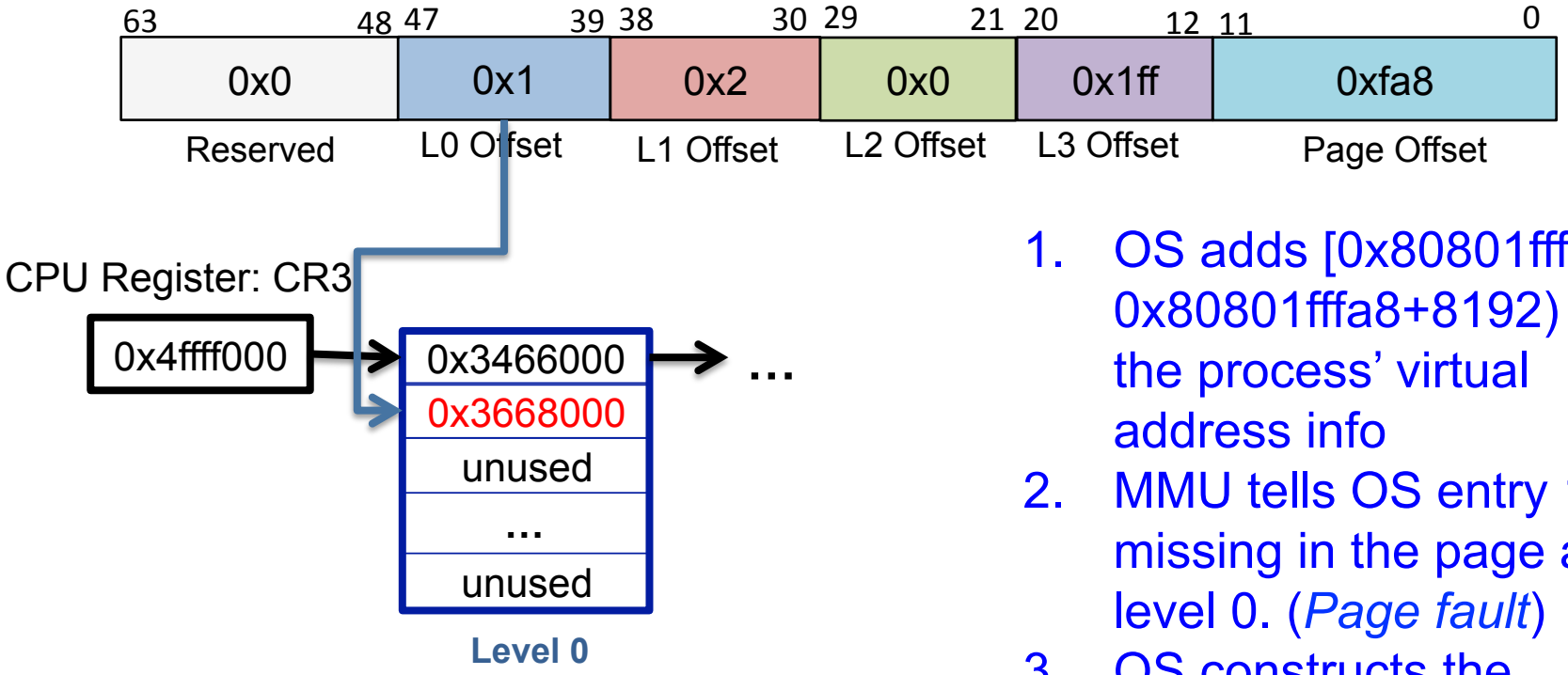


1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)

current process' page table

Demand Paging

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

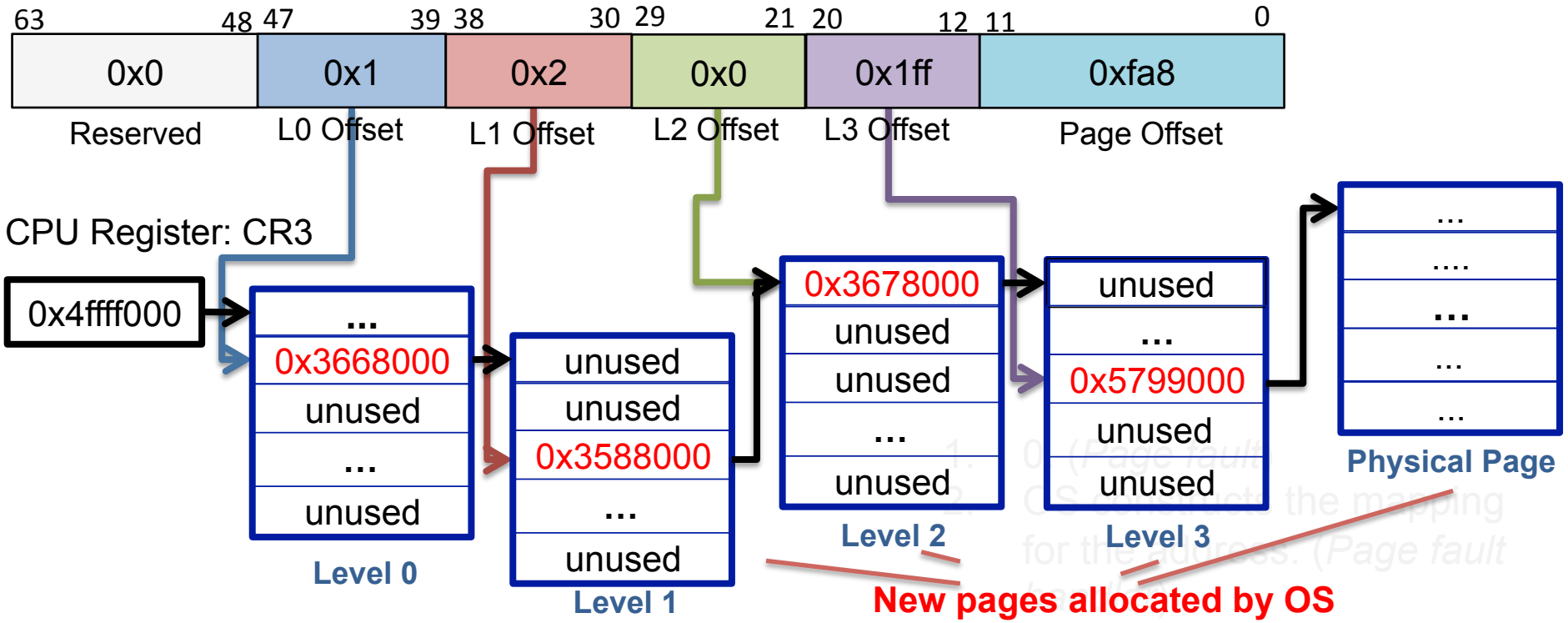


1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)
3. OS constructs the mapping for the address. (*Page fault handler*)

current process' page table

Demand Paging

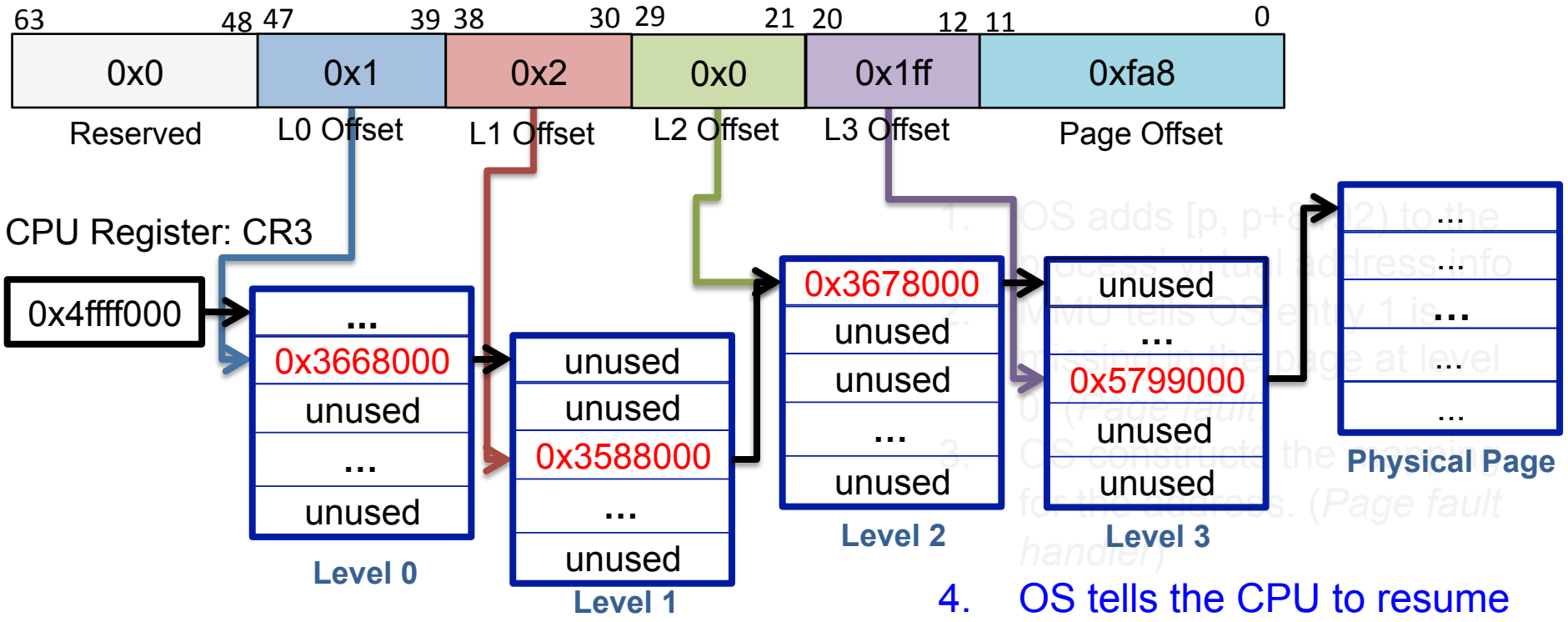
```
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's'
```



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

Demand Paging

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

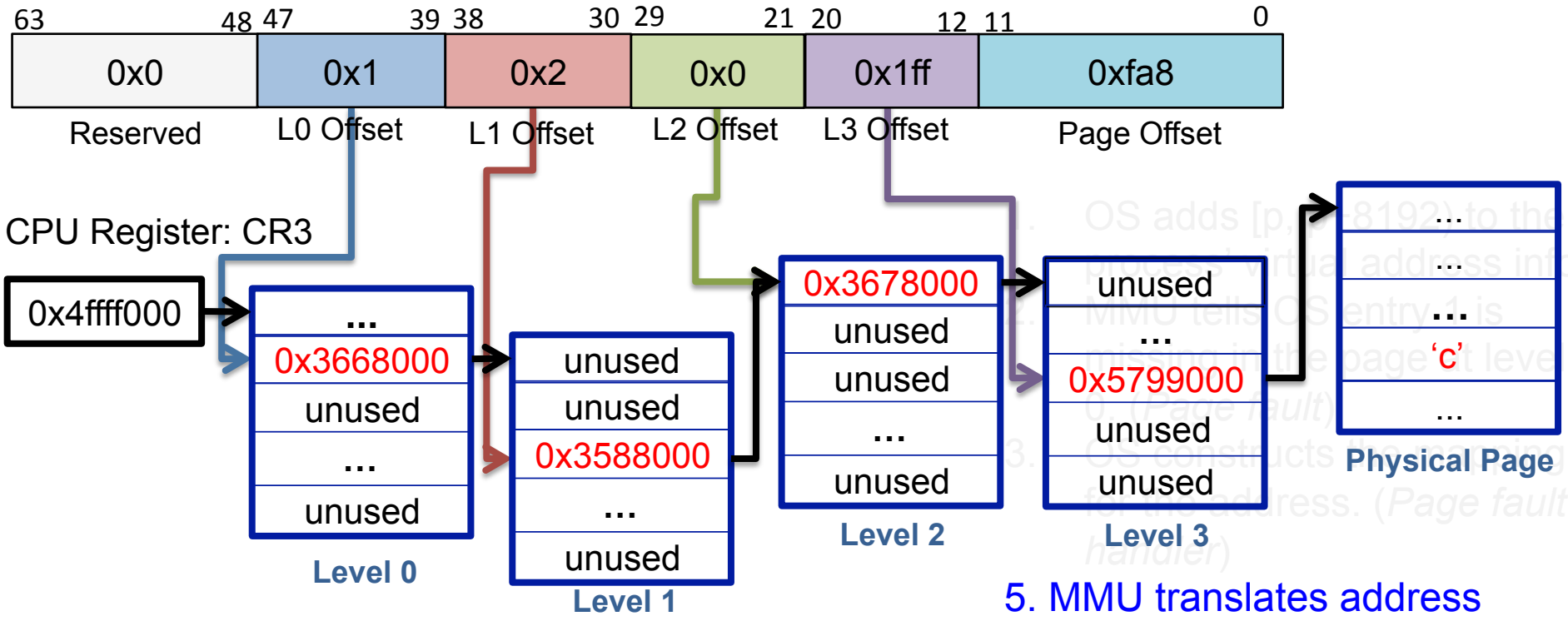


4. OS tells the CPU to resume execution

current process' page table

Demand Paging

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

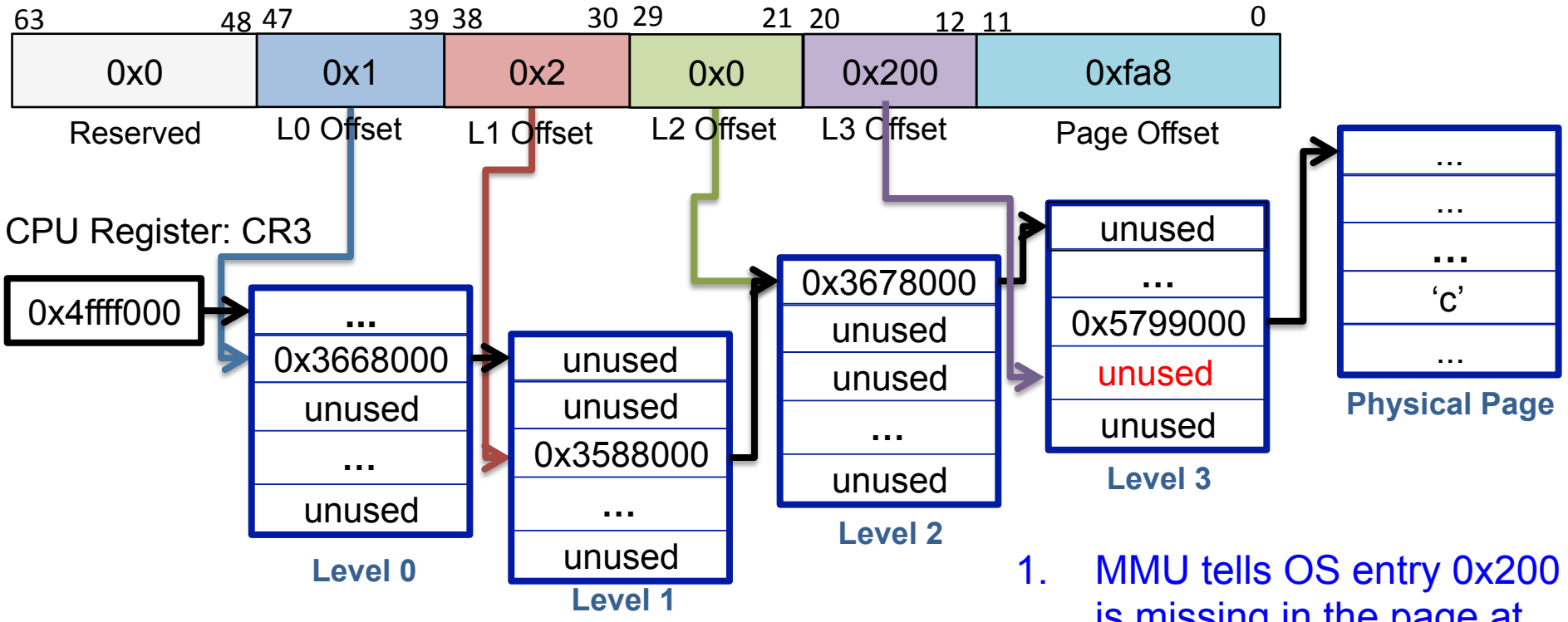


current process' page table

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

Demand Paging

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

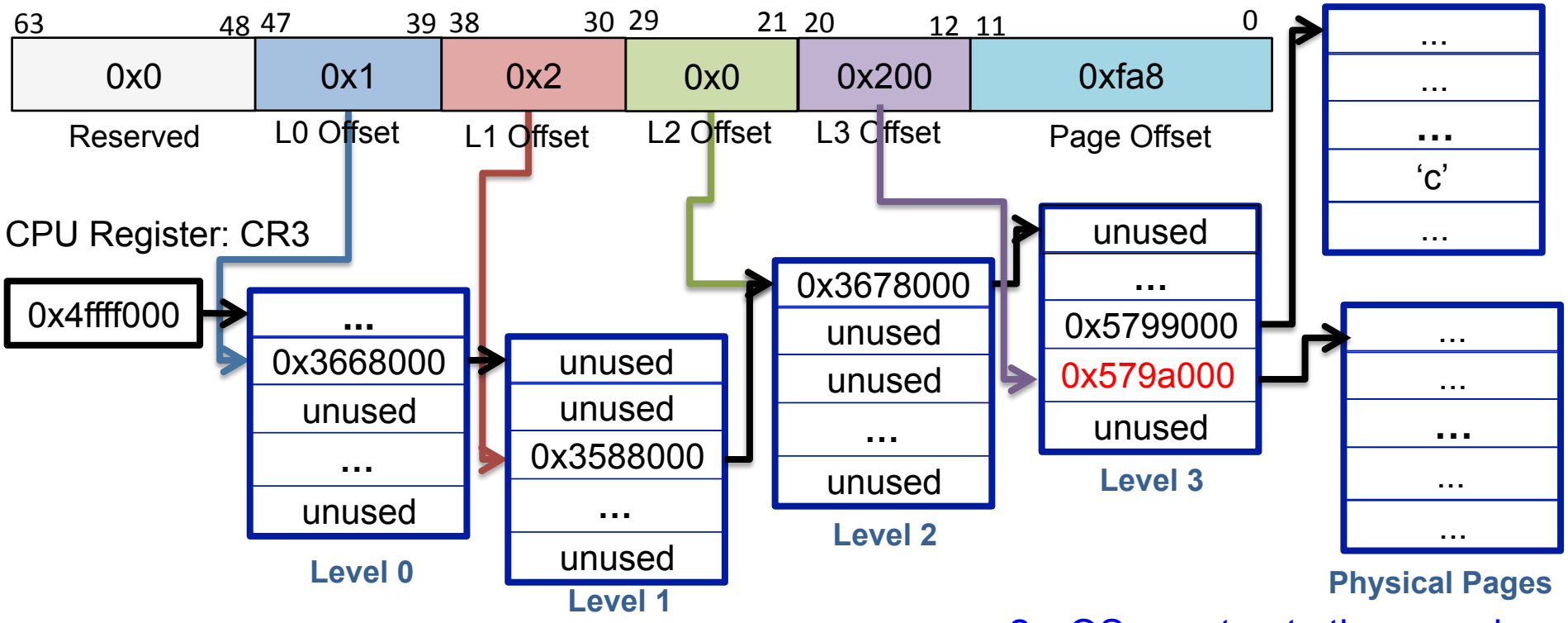


1. MMU tells OS entry 0x200 is missing in the page at level 3. (*Page fault*)

current process' page table

Demand Paging

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

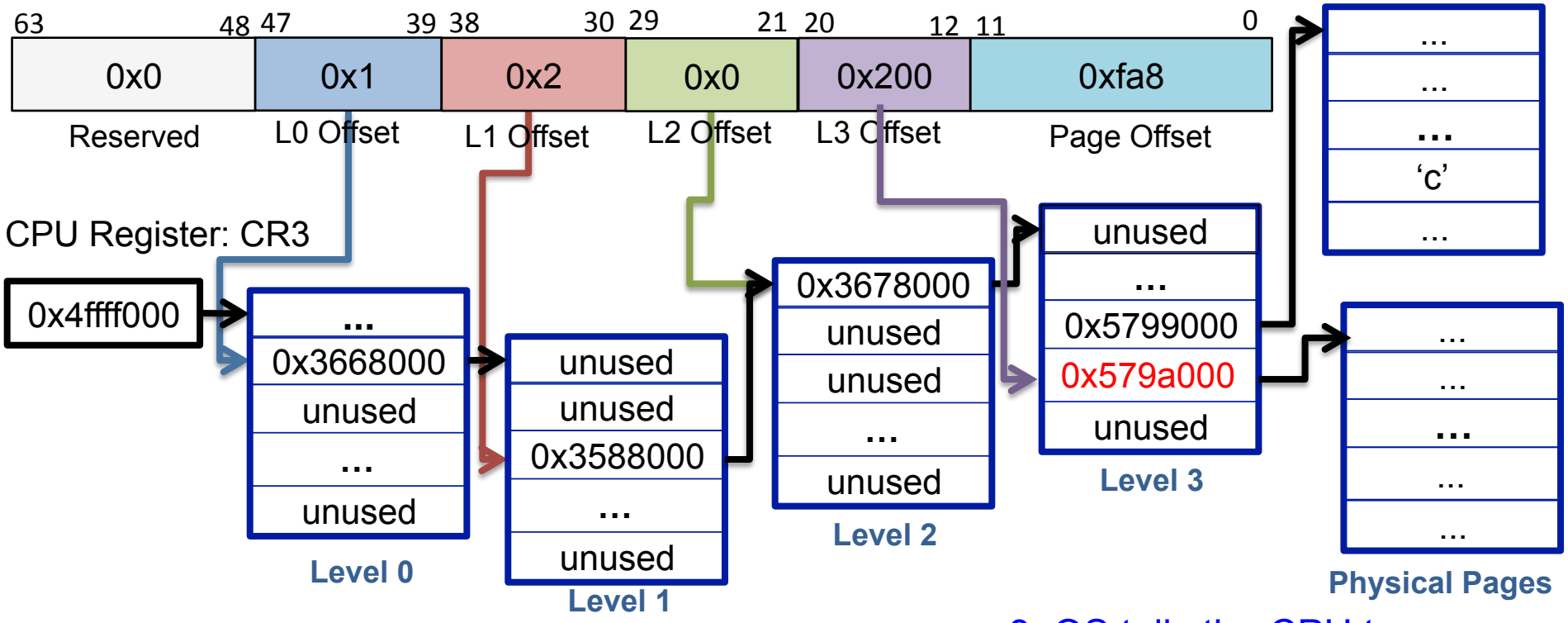


current process' page table

2. 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' //0x8080200fa8
```

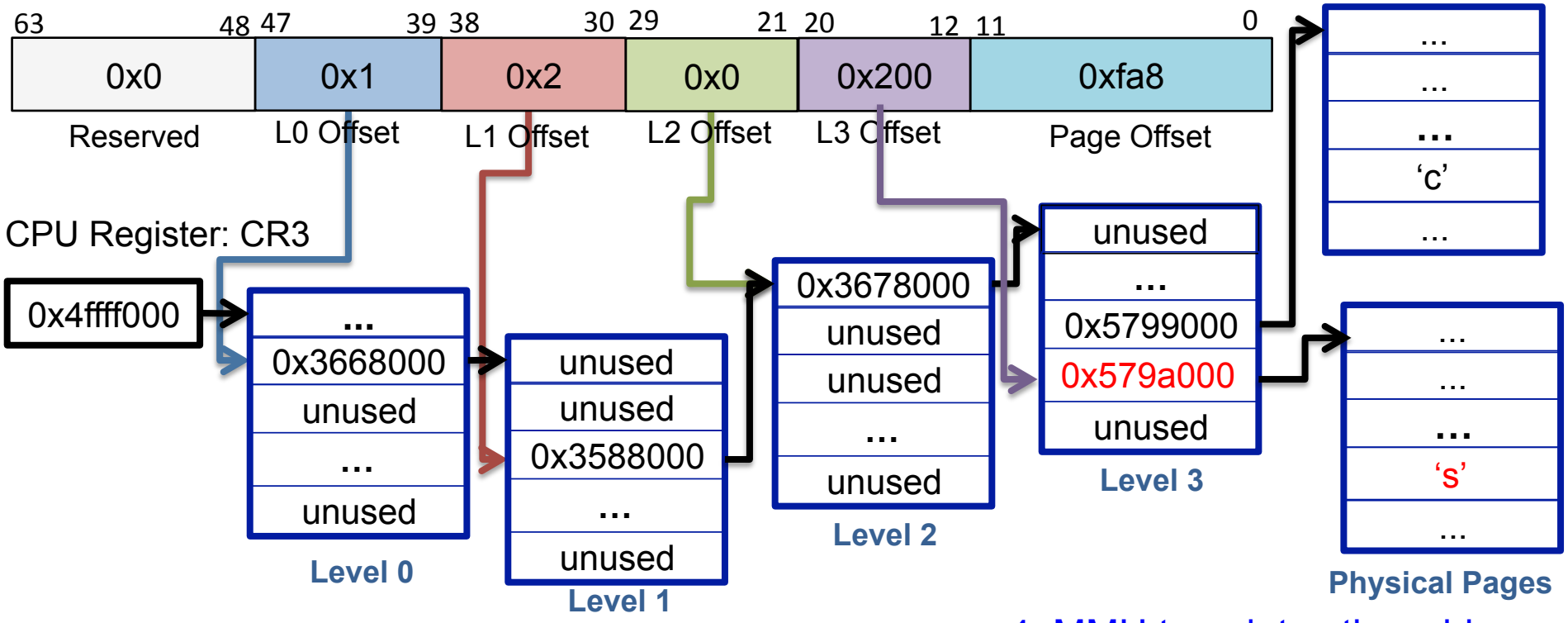


current process' page table

3. OS tells the CPU to resume execution

Demand Paging

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



current process' page table

4. MMU translates the address again and access the physical memory.

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?

$$\frac{2^{12}}{2^{3}}$$

2^{12} ← page size

2^{3} ← 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

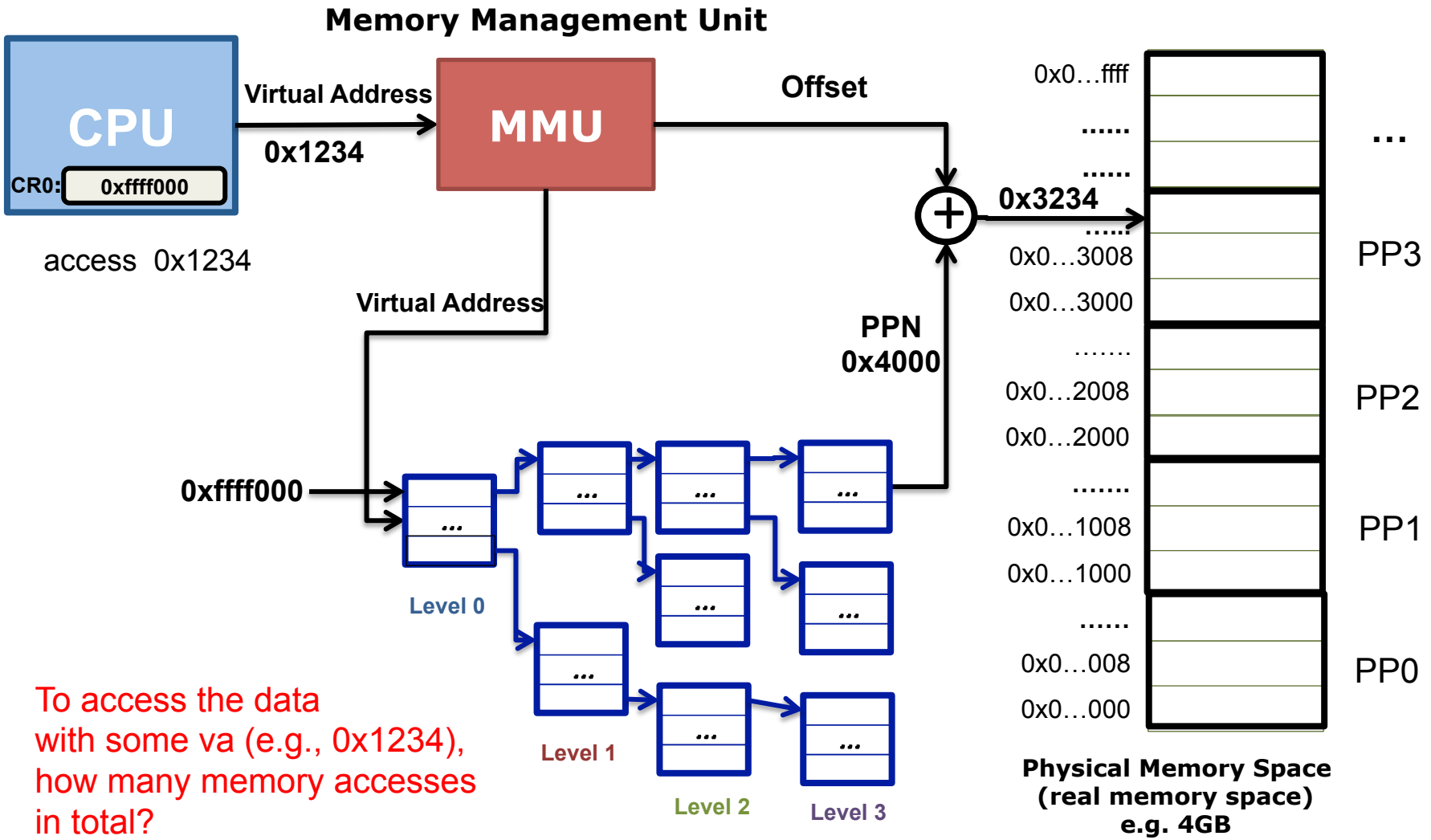
Memory access latency

- 100 ns
- 160 ~ 200 CPU cycles

Instructions that do not involve memory access can execute very quickly:

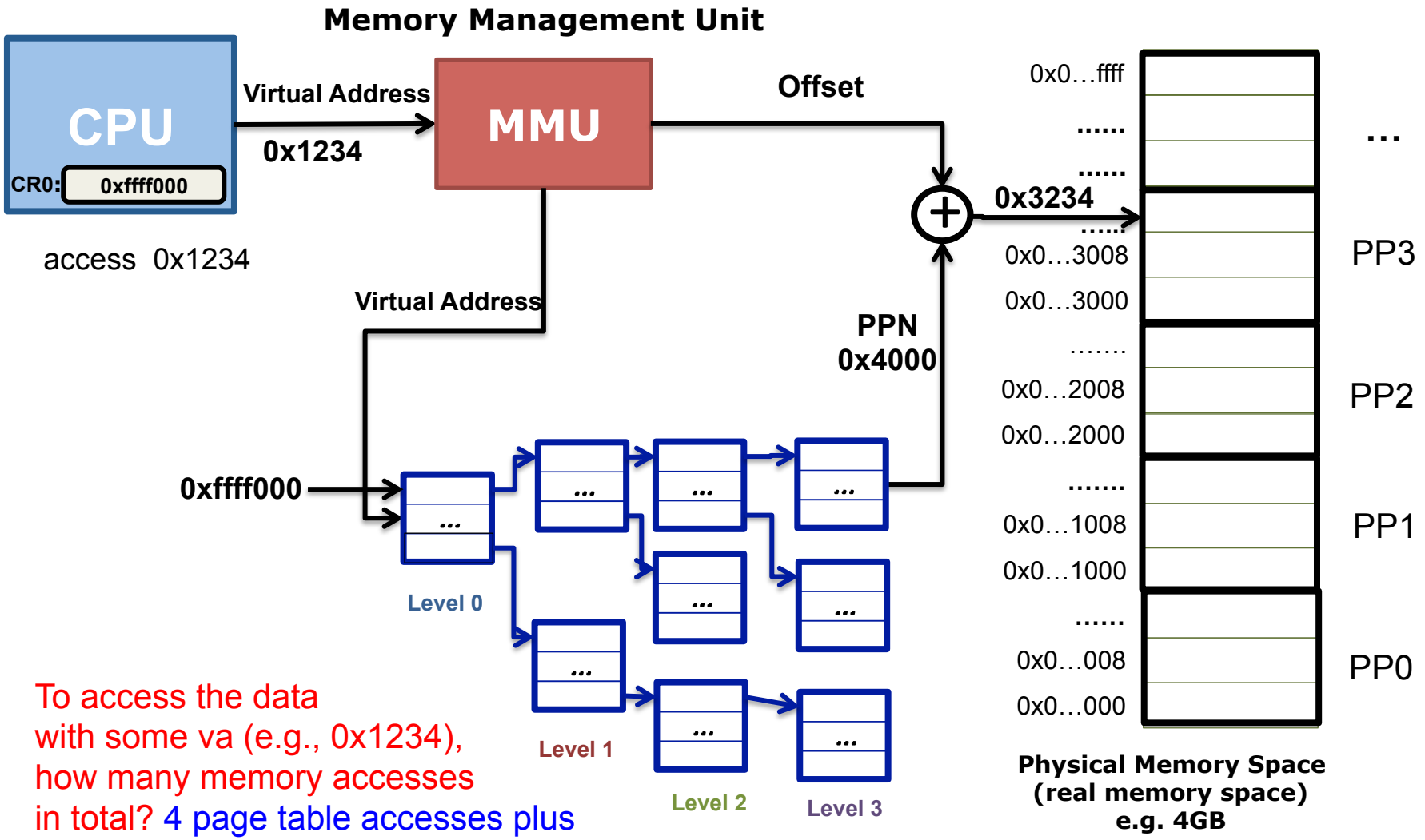
- Instructions per CPU cycle >= 1

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

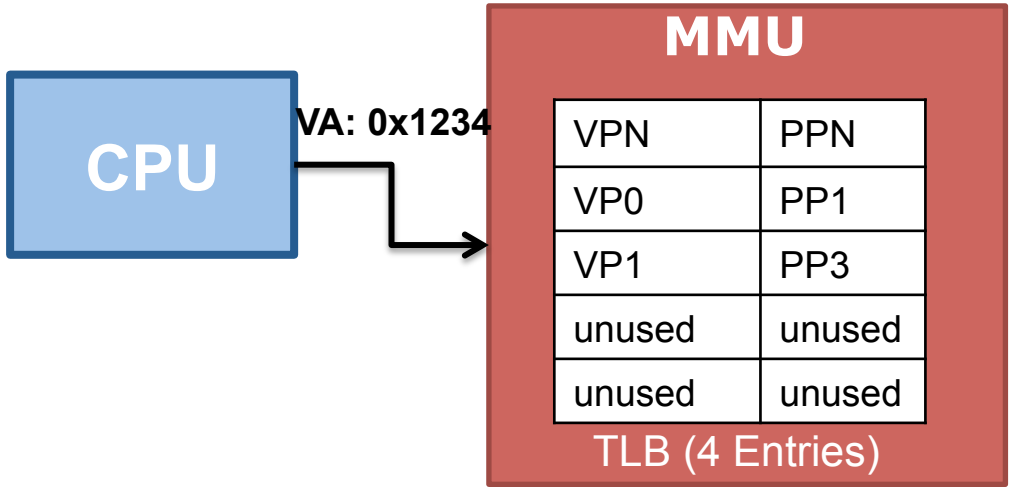


To access the data with some va (e.g., 0x1234), how many memory accesses in total? 4 page table accesses plus one time data access which is **5** memory accesses.

Speedup Address Translation

Translation lookaside buffer (TLB)

- Small cache in MMU
- Maps virtual page numbers to physical page numbers

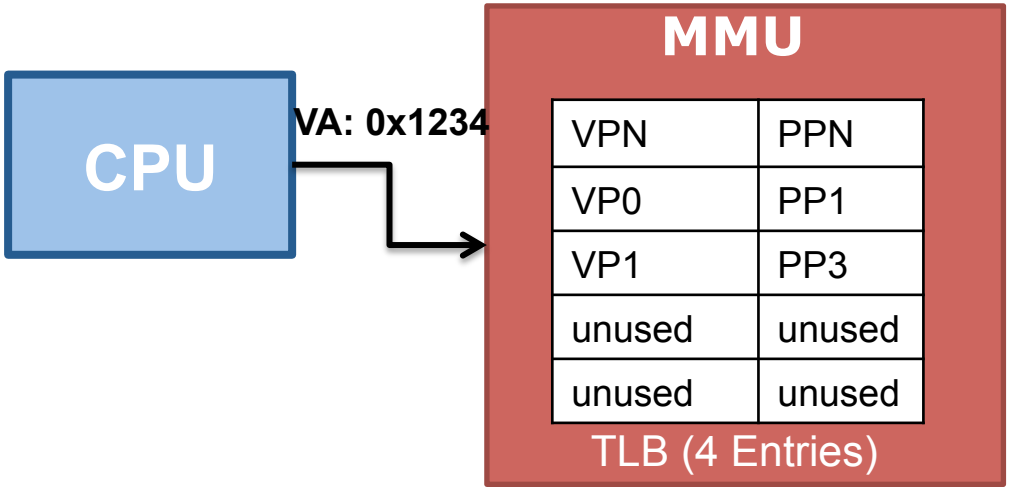


1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = *VPN % 4*
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

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

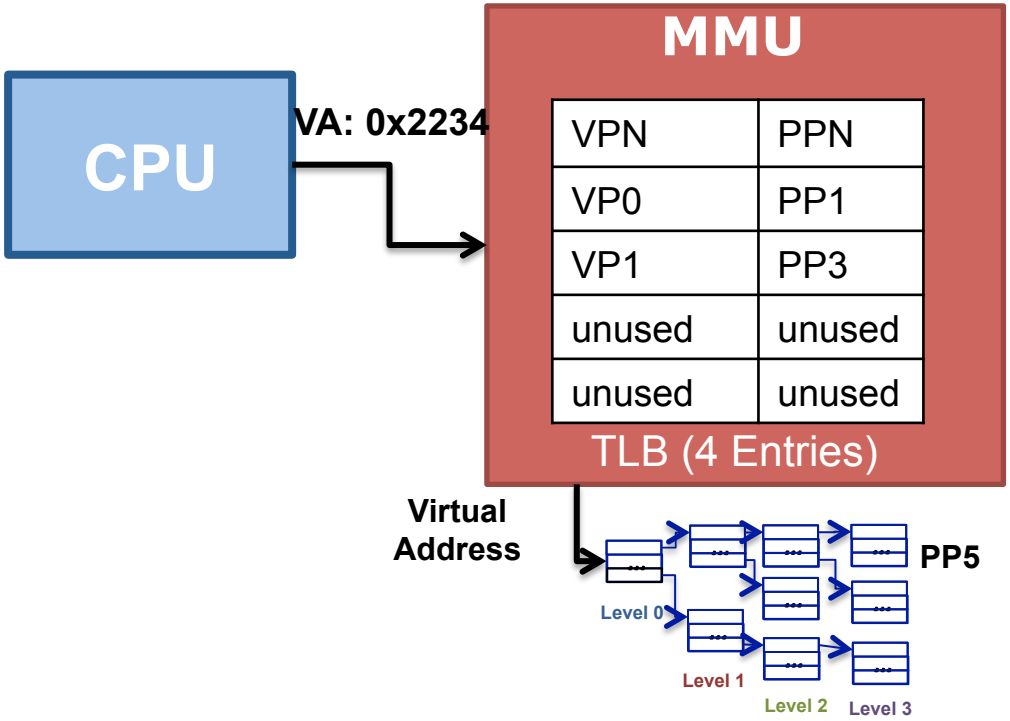
Example:

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

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

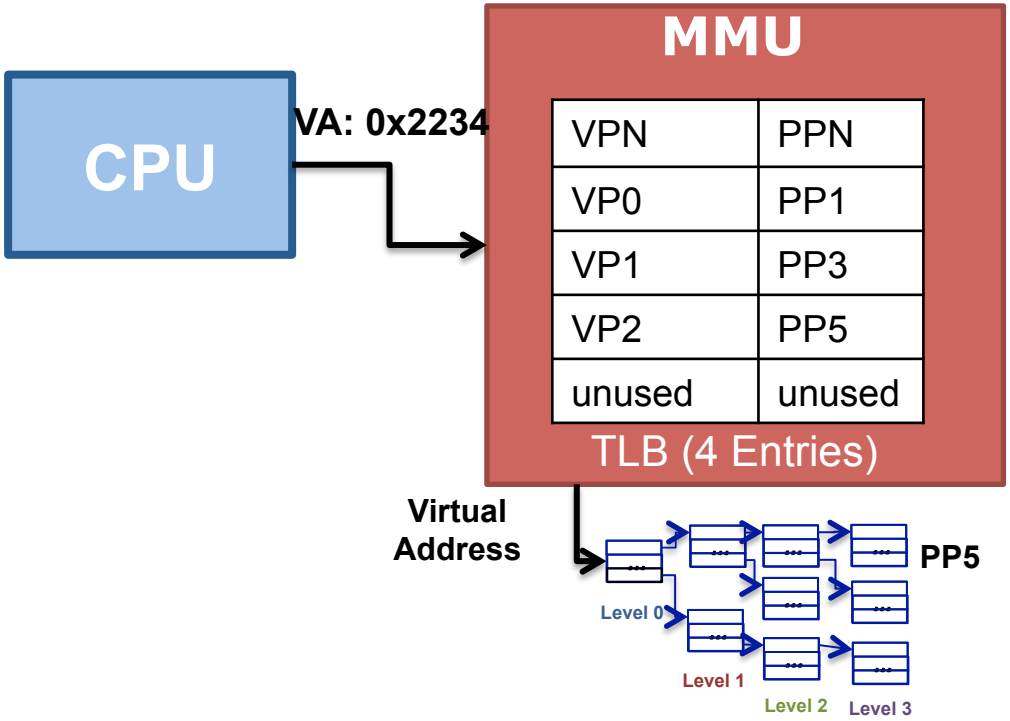
Example:

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

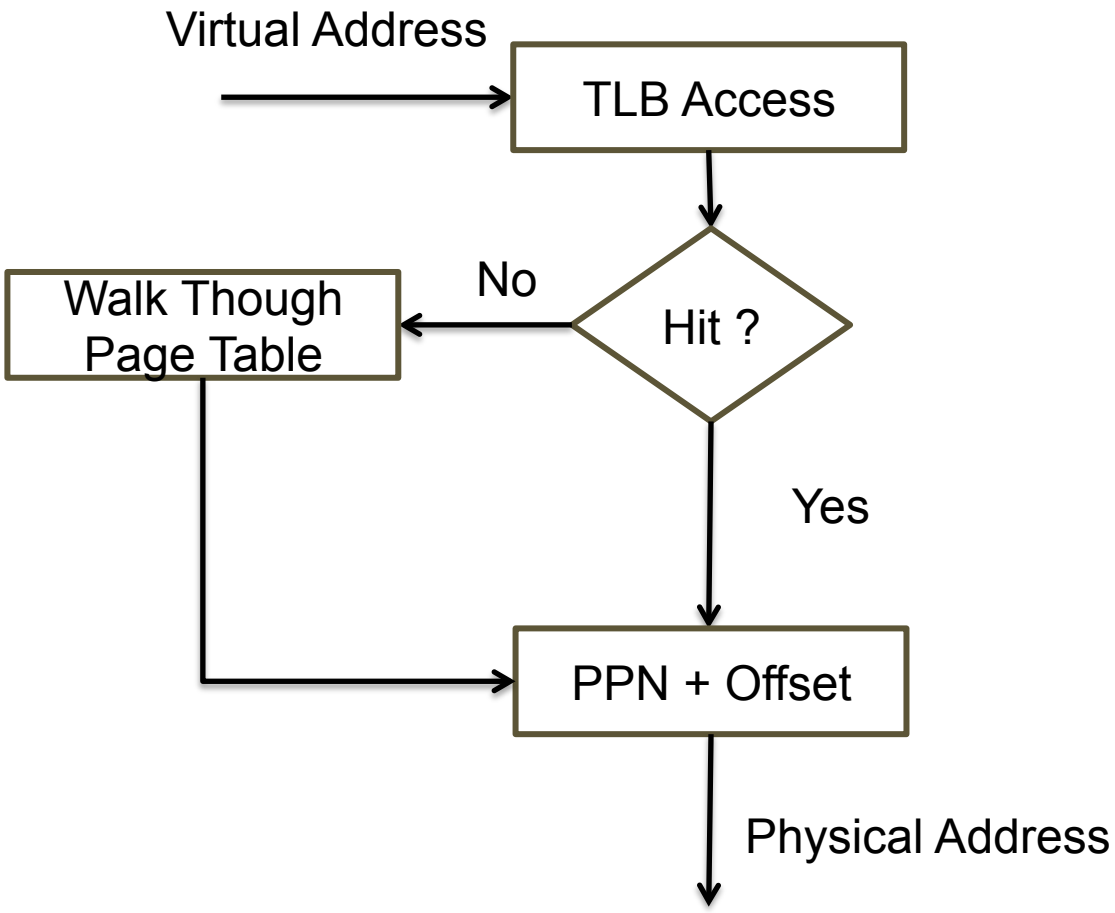
Memory access

- Hundreds of CPU cycles

TLB access

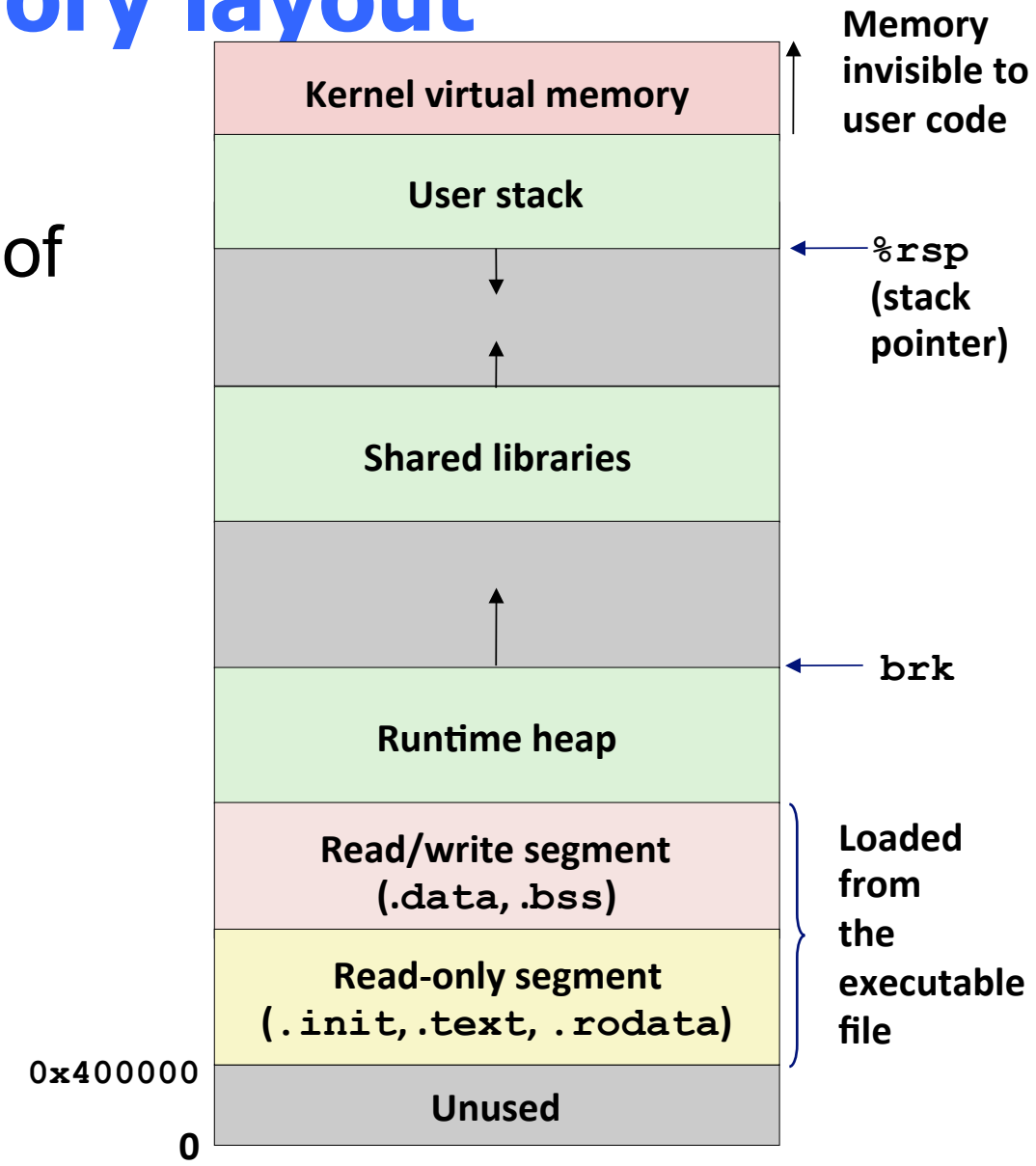
- Only a couple of CPU cycles

Summary


Virtual Address
TLB Access
Hit ?
No
Walk Though
Page Table
Yes
PPN + Offset
Physical Address

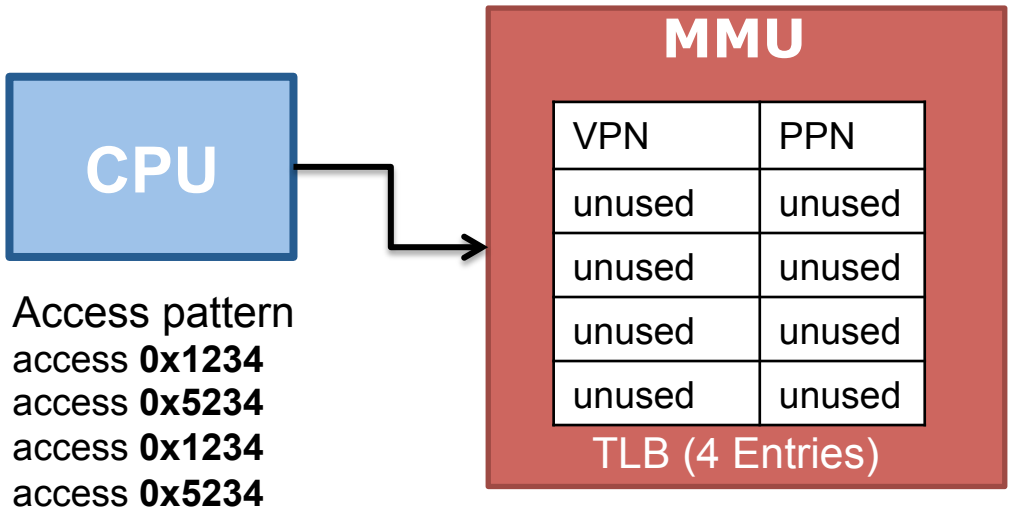
Recall the graph of a program's memory layout

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



More on TLBs

Speedup Address Translation



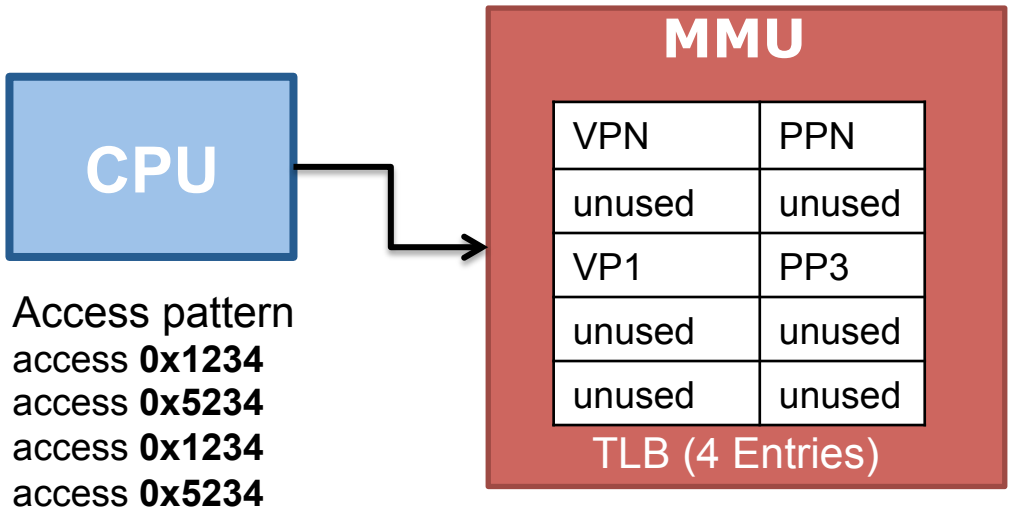
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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

Speedup Address Translation



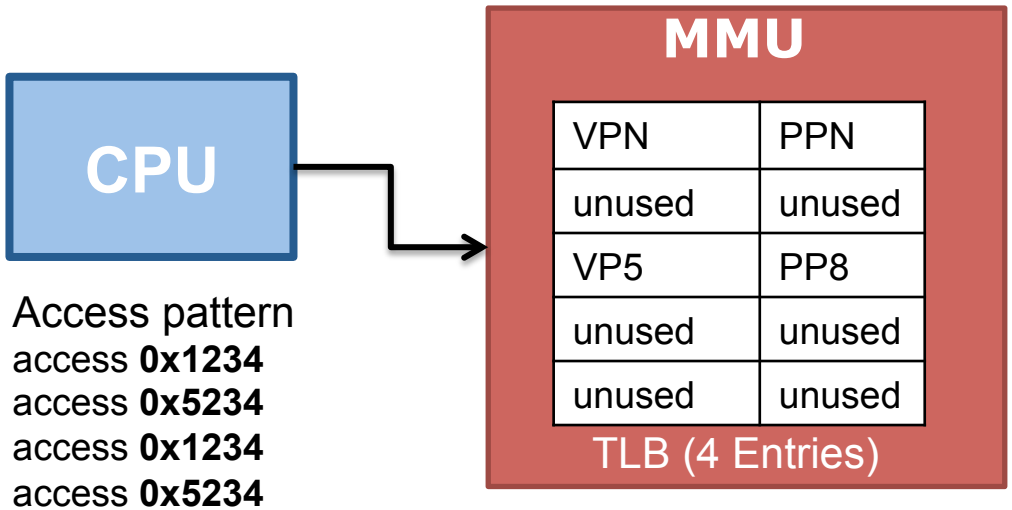
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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

Speedup Address Translation



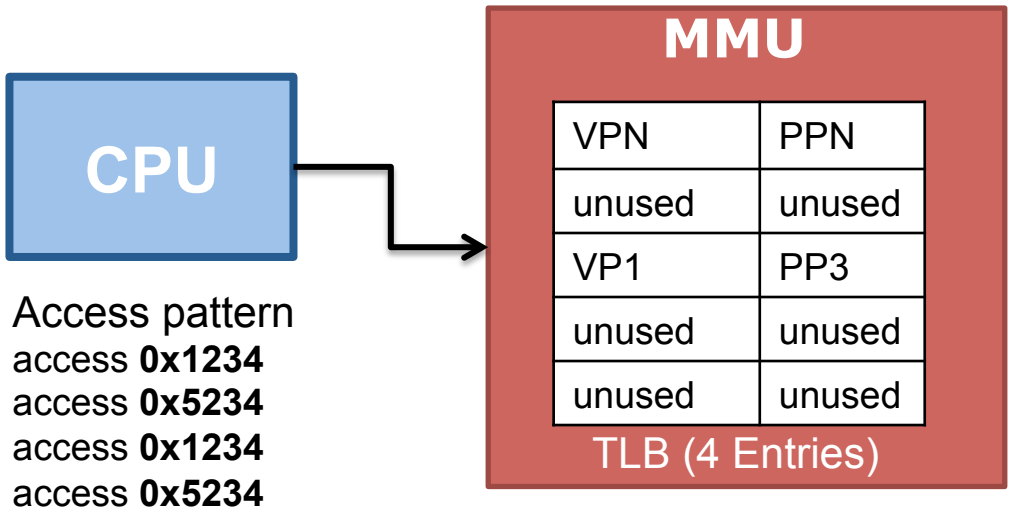
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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

Speedup Address Translation



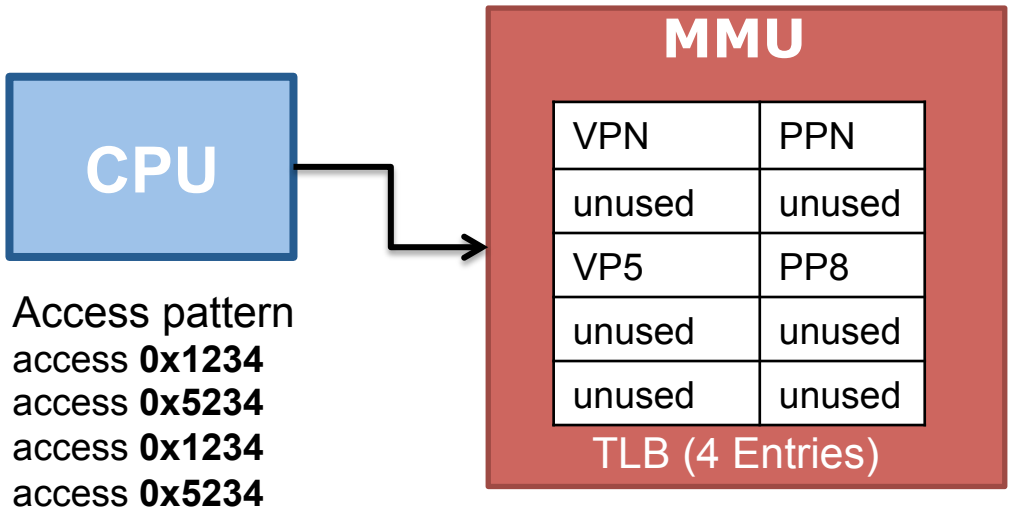
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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



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

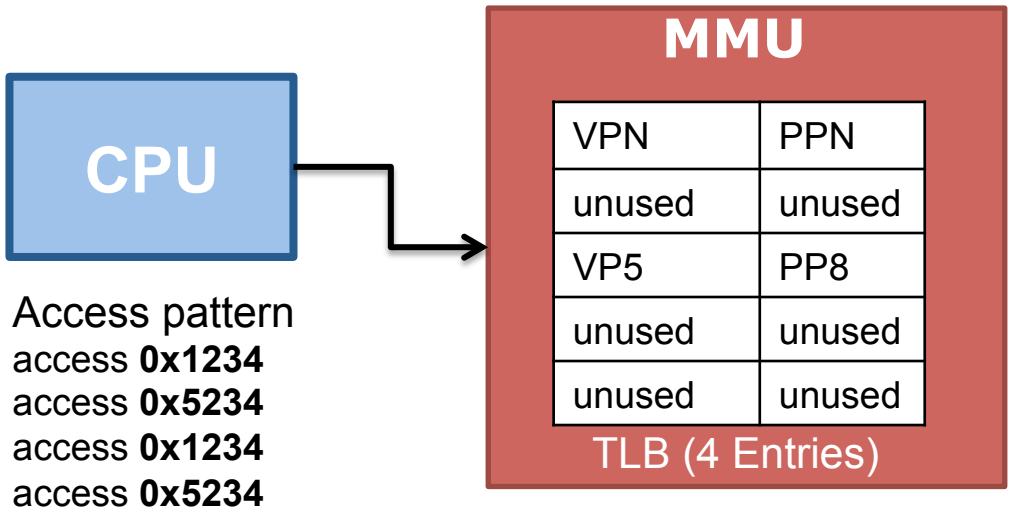
1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = *VPN % 4*
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

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

TLB eviction due to conflict!

Speedup Address Translation



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

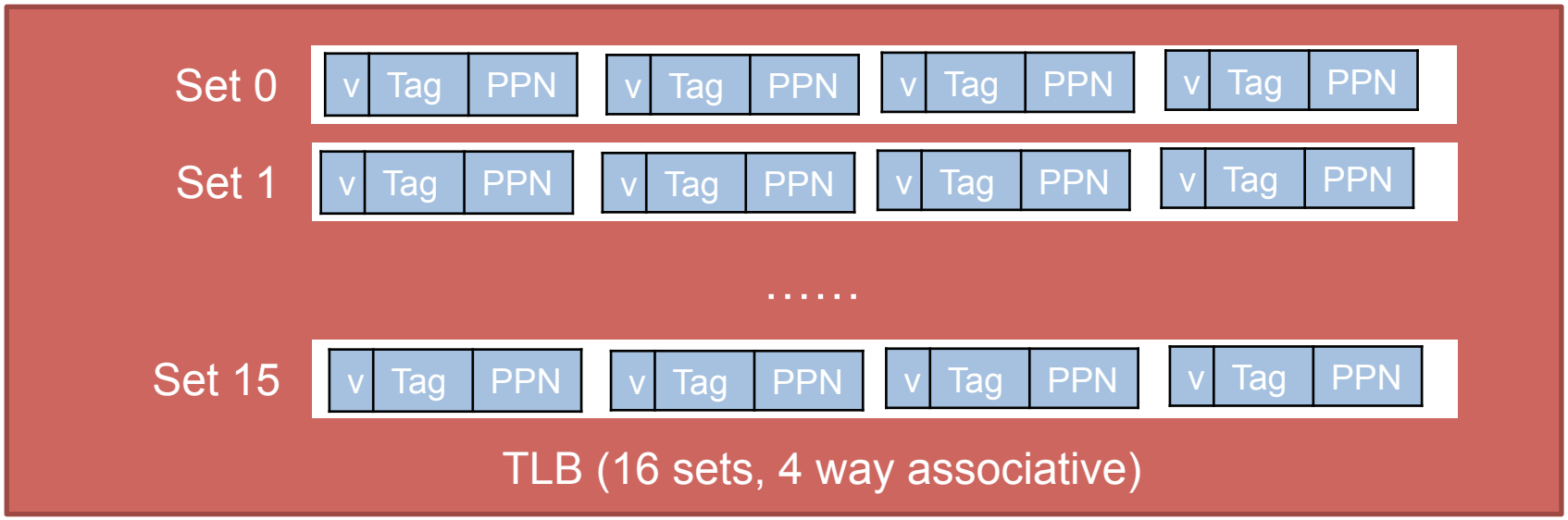
1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = *VPN % 4*
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

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

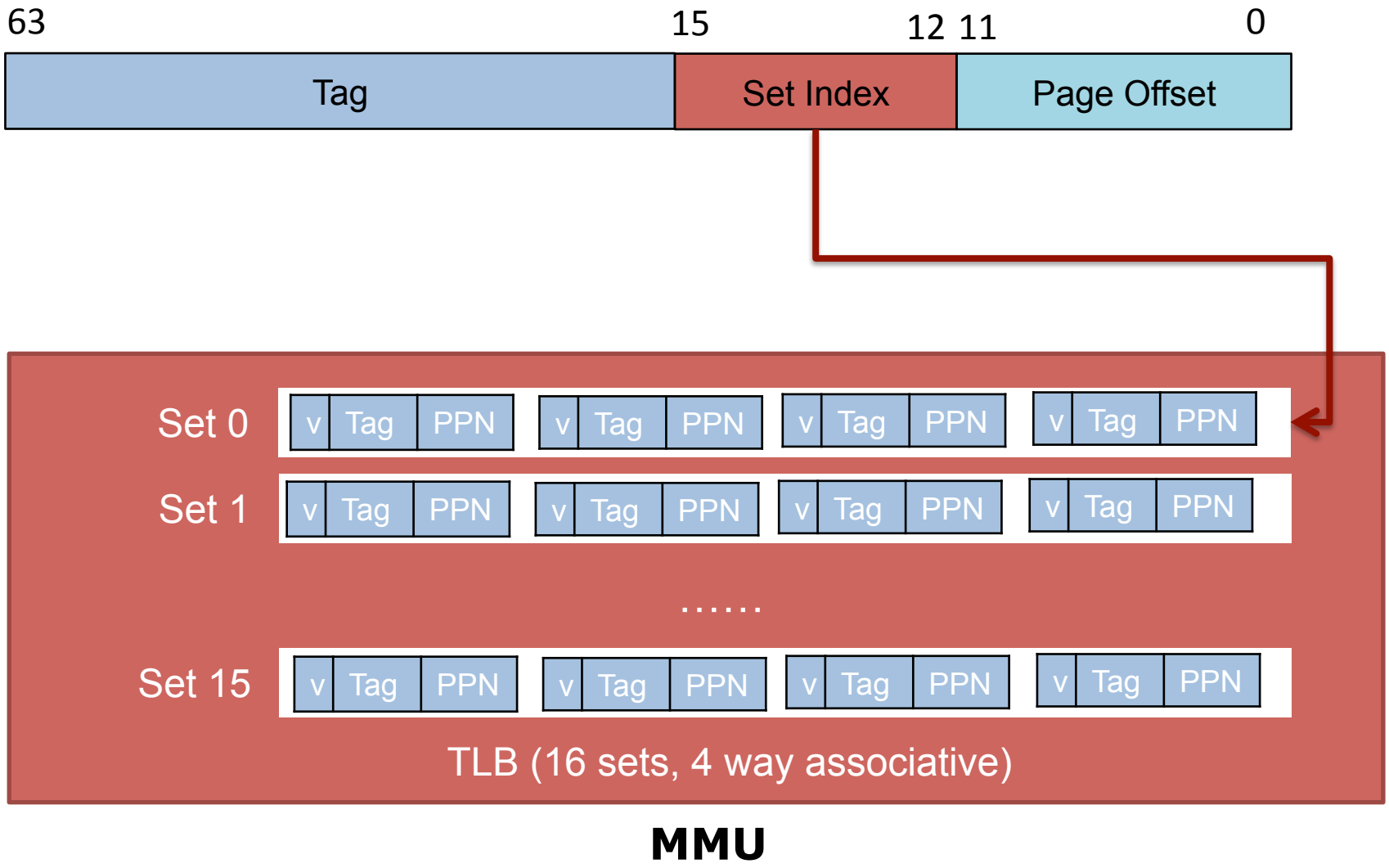
TLB eviction due to conflict! → Multi-set associative TLB

Multi-set associative TLB



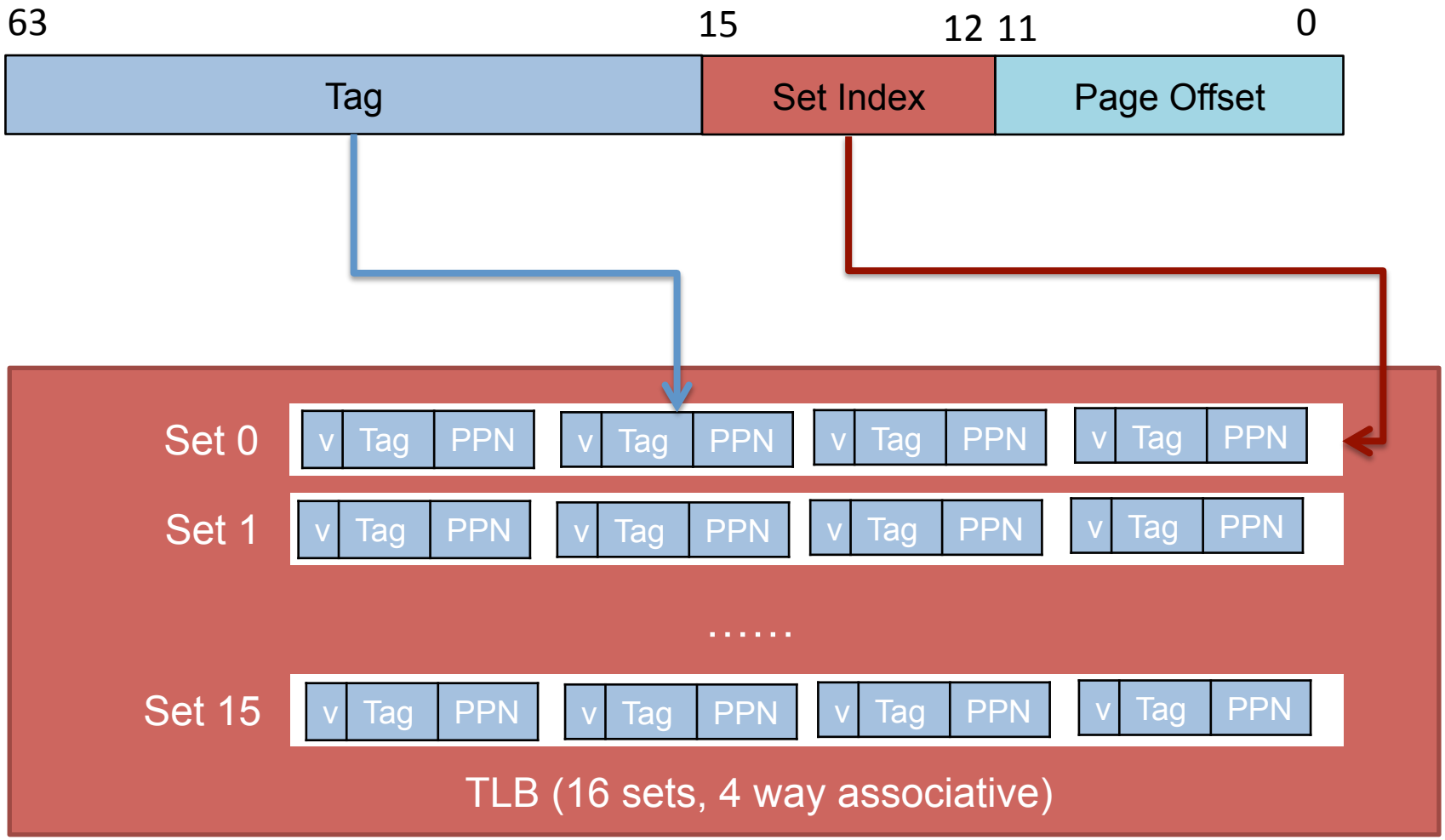
MMU

Multi-set associative TLB



MMU

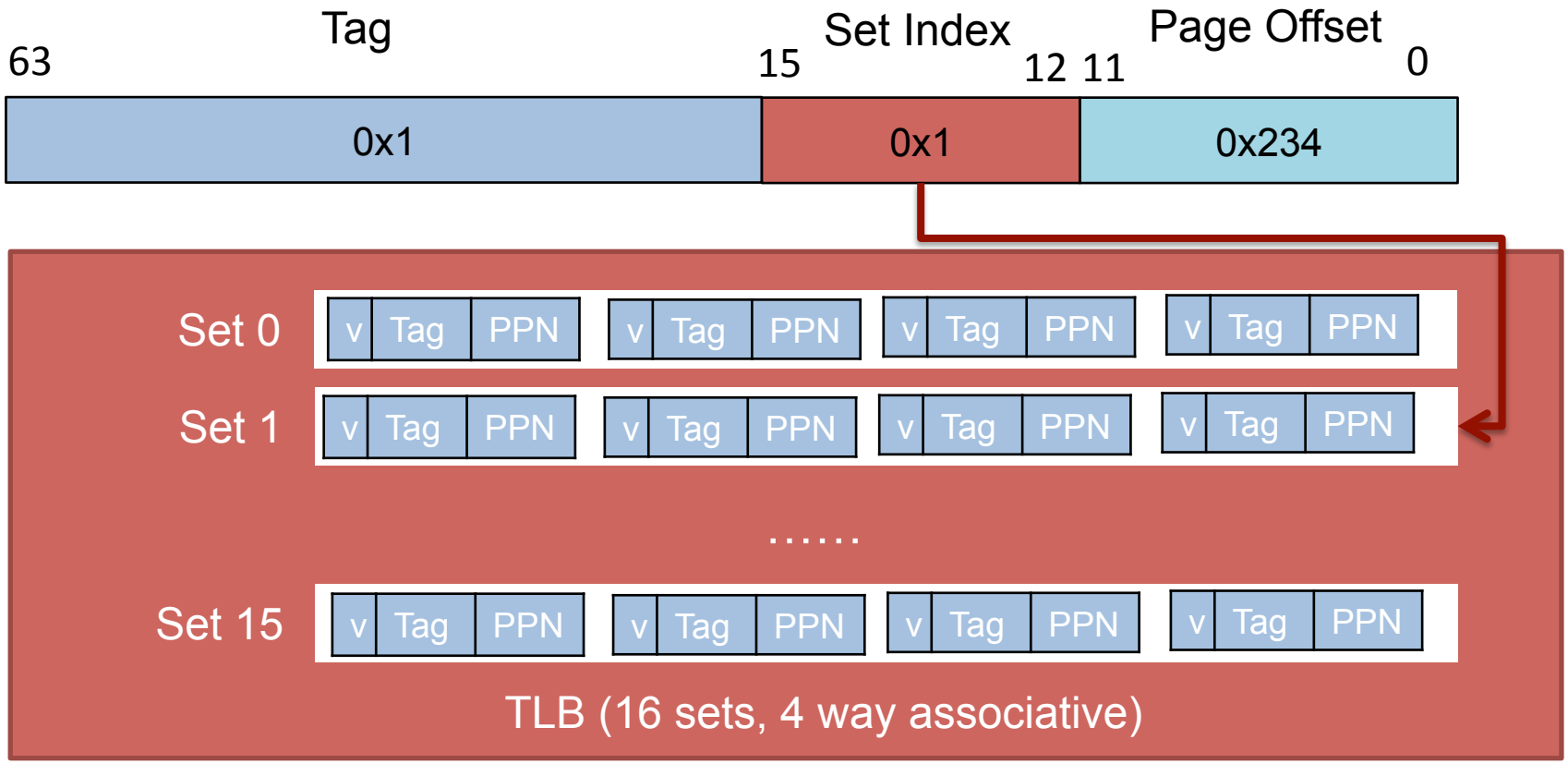
Multi-set associative TLB



MMU

Example

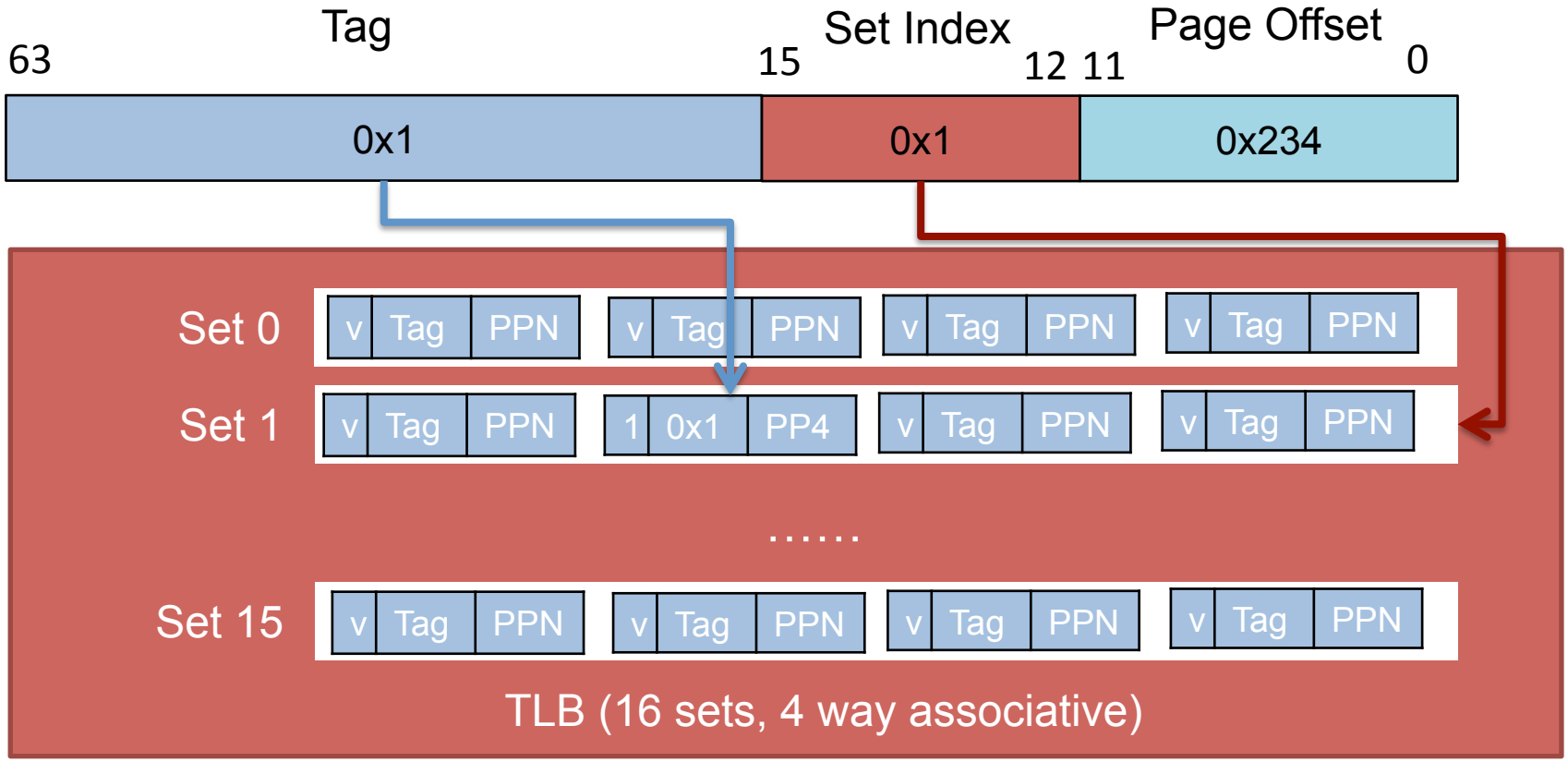
→ access **0x11234, TLB Miss**
access **0x21234**
access **0x11234**
access **0x21234**



MMU

Example

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



MMU

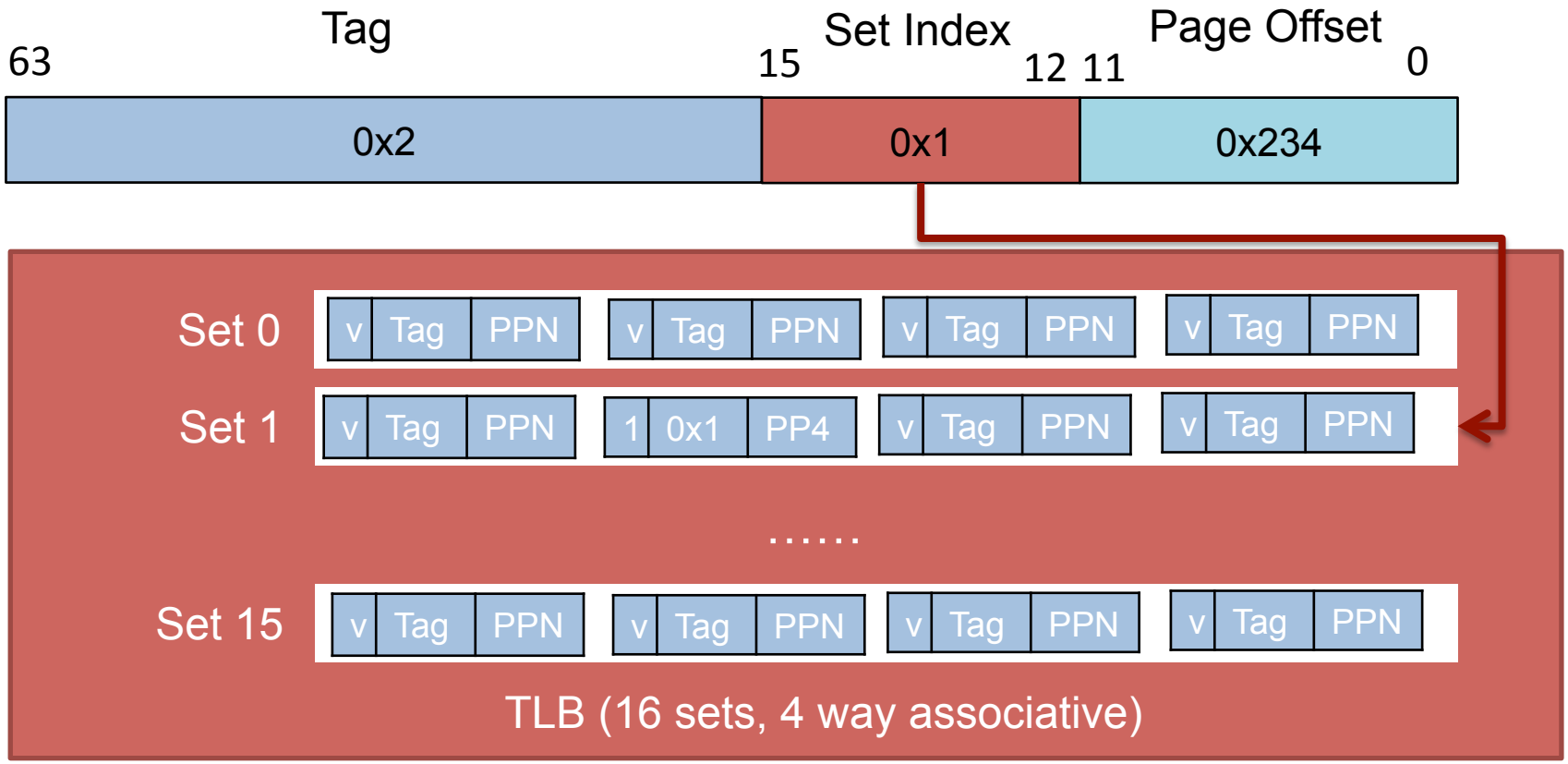
Example

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

→ access **0x21234, TLB Miss,**

access **0x11234**

access **0x21234**



MMU

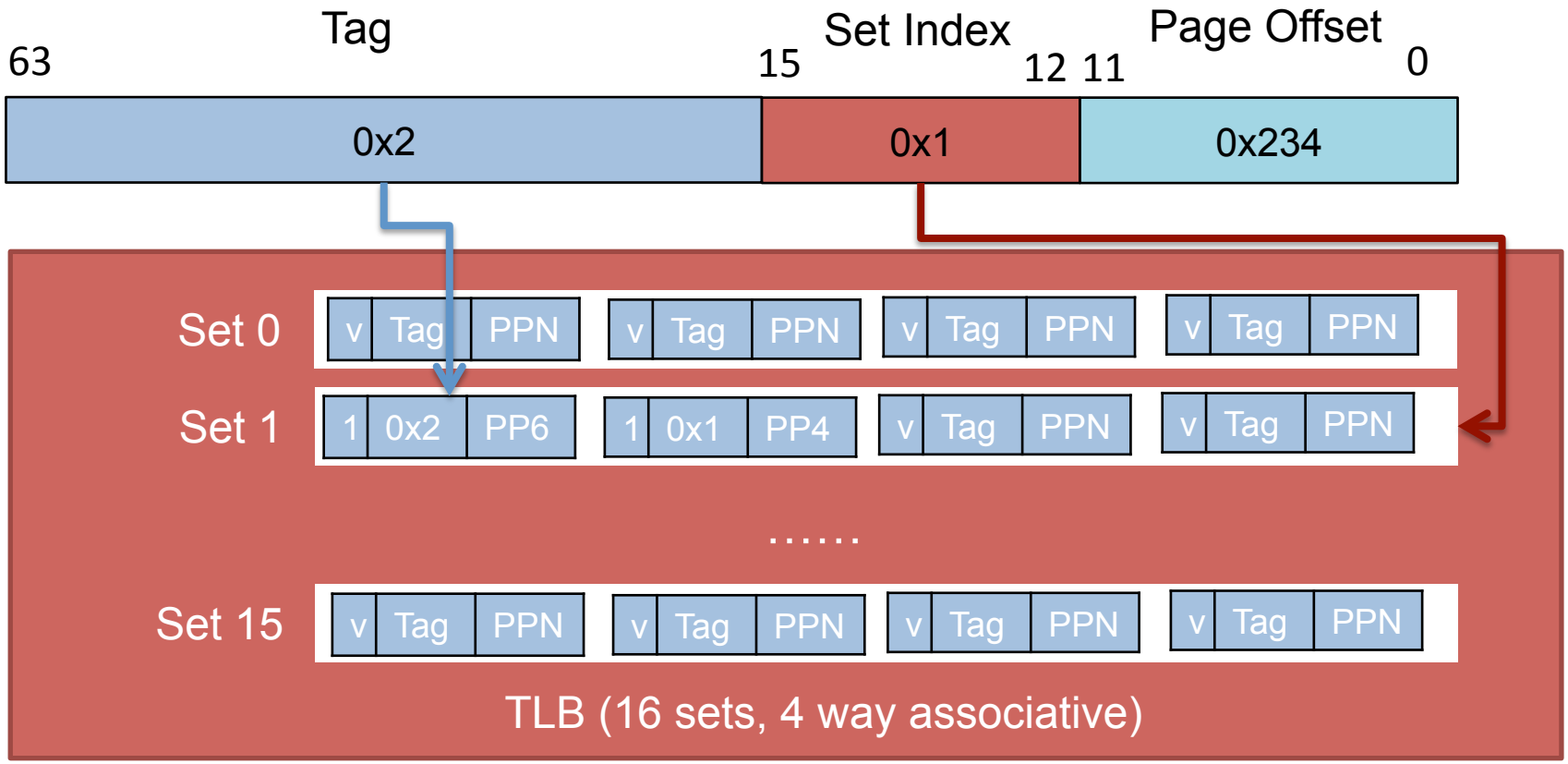
Example

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

→ access **0x21234, TLB Miss, cache the translation result**

access **0x11234**

access **0x21234**



MMU

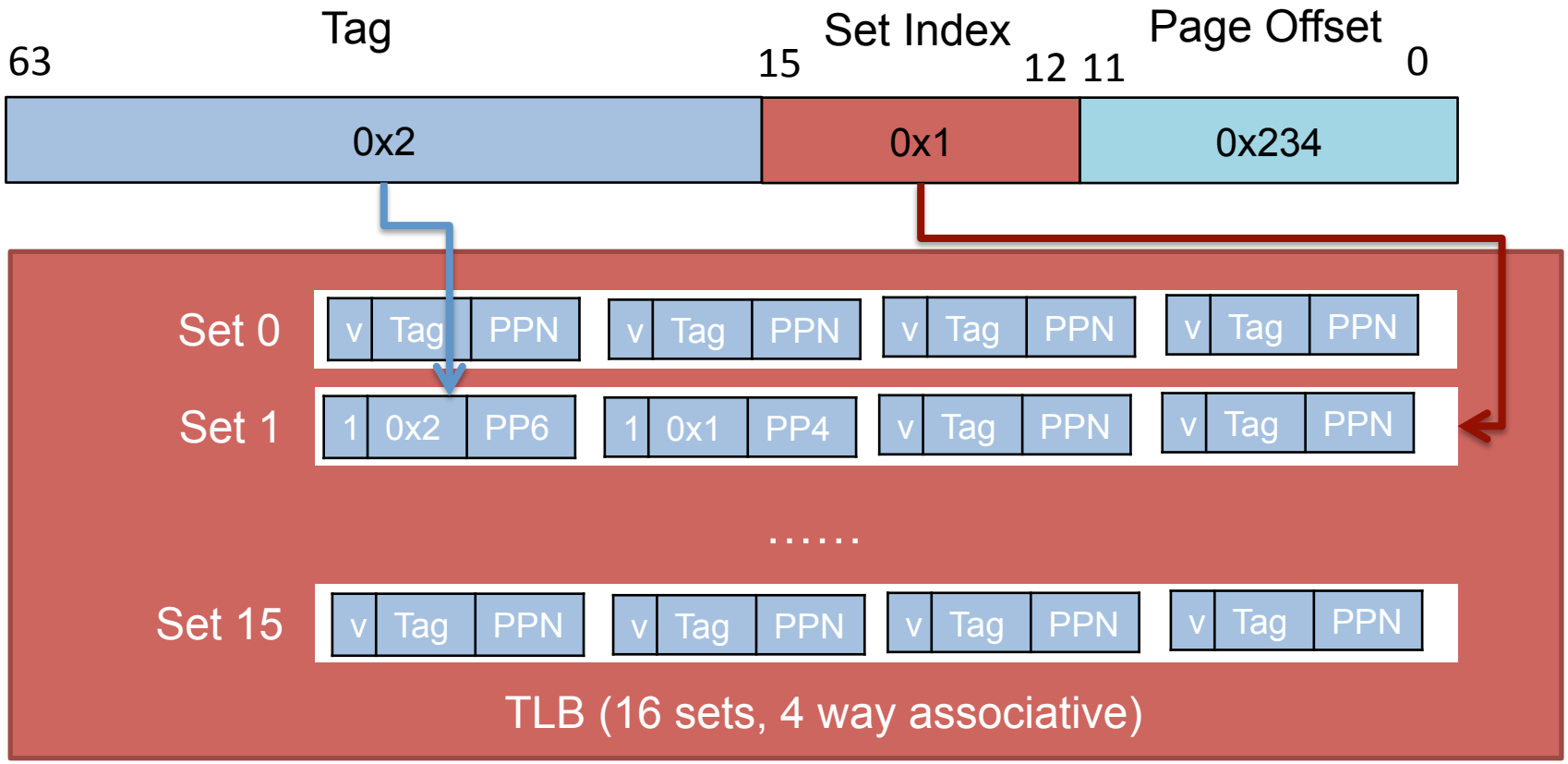
Example

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

→ access **0x21234, TLB Miss, cache the translation result**

access **0x11234**

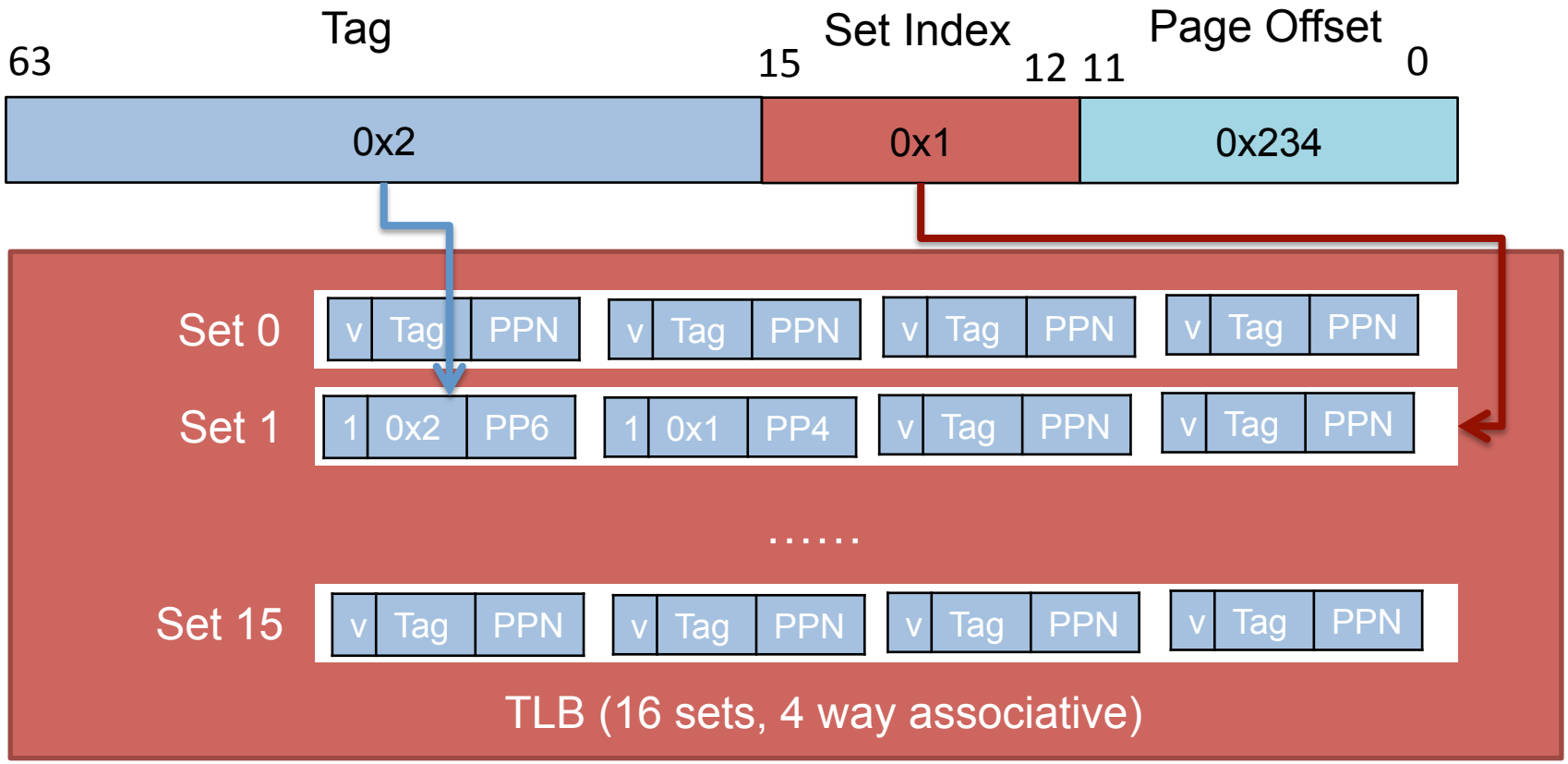
access **0x21234**



MMU

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**



MMU