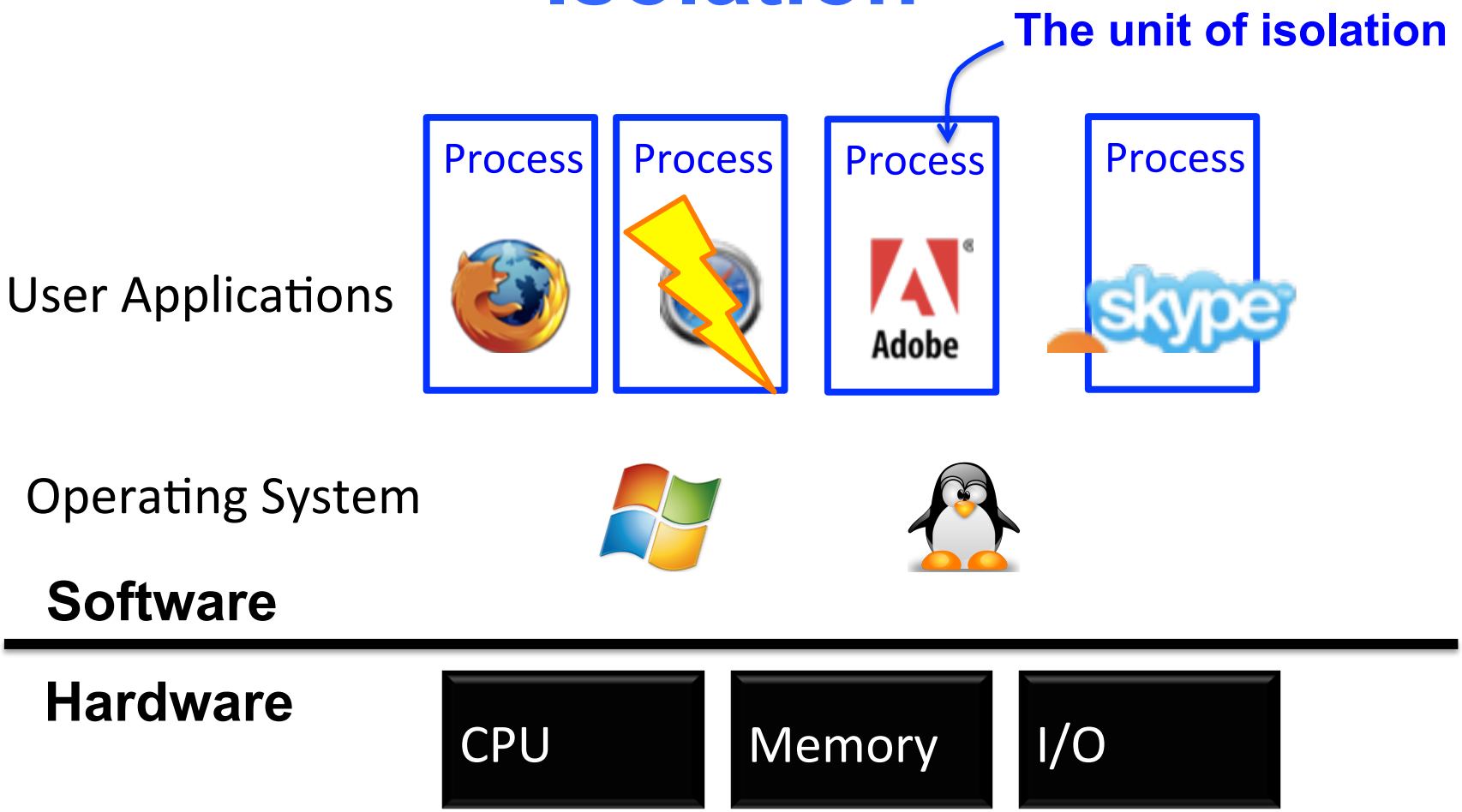
Isolation & Virtual Memory

Jinyang Li

Layered organization

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

Isolation

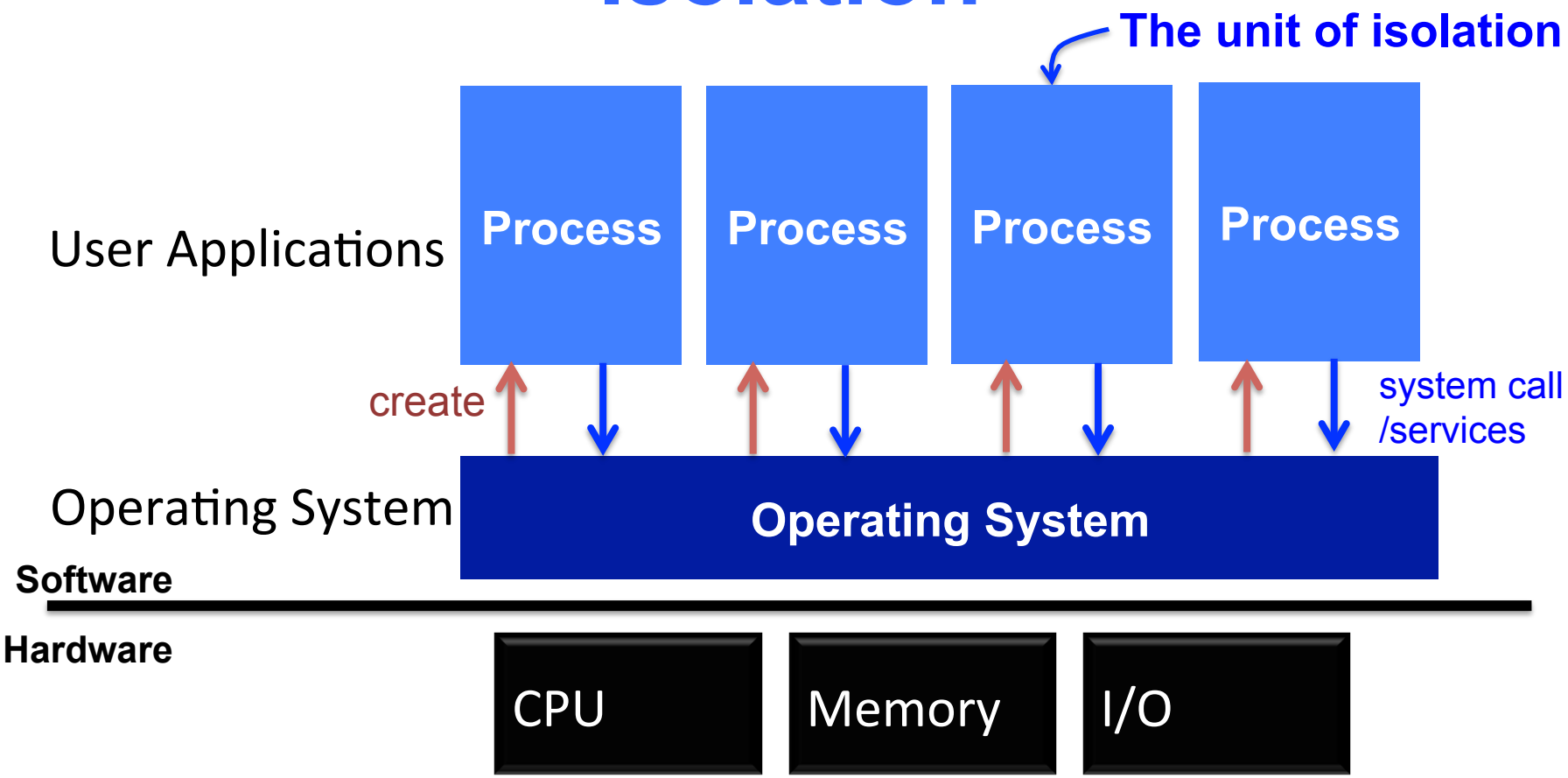


Isolation is provided by OS using special hardware primitives

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*** lists all processes, **kill** terminates a process

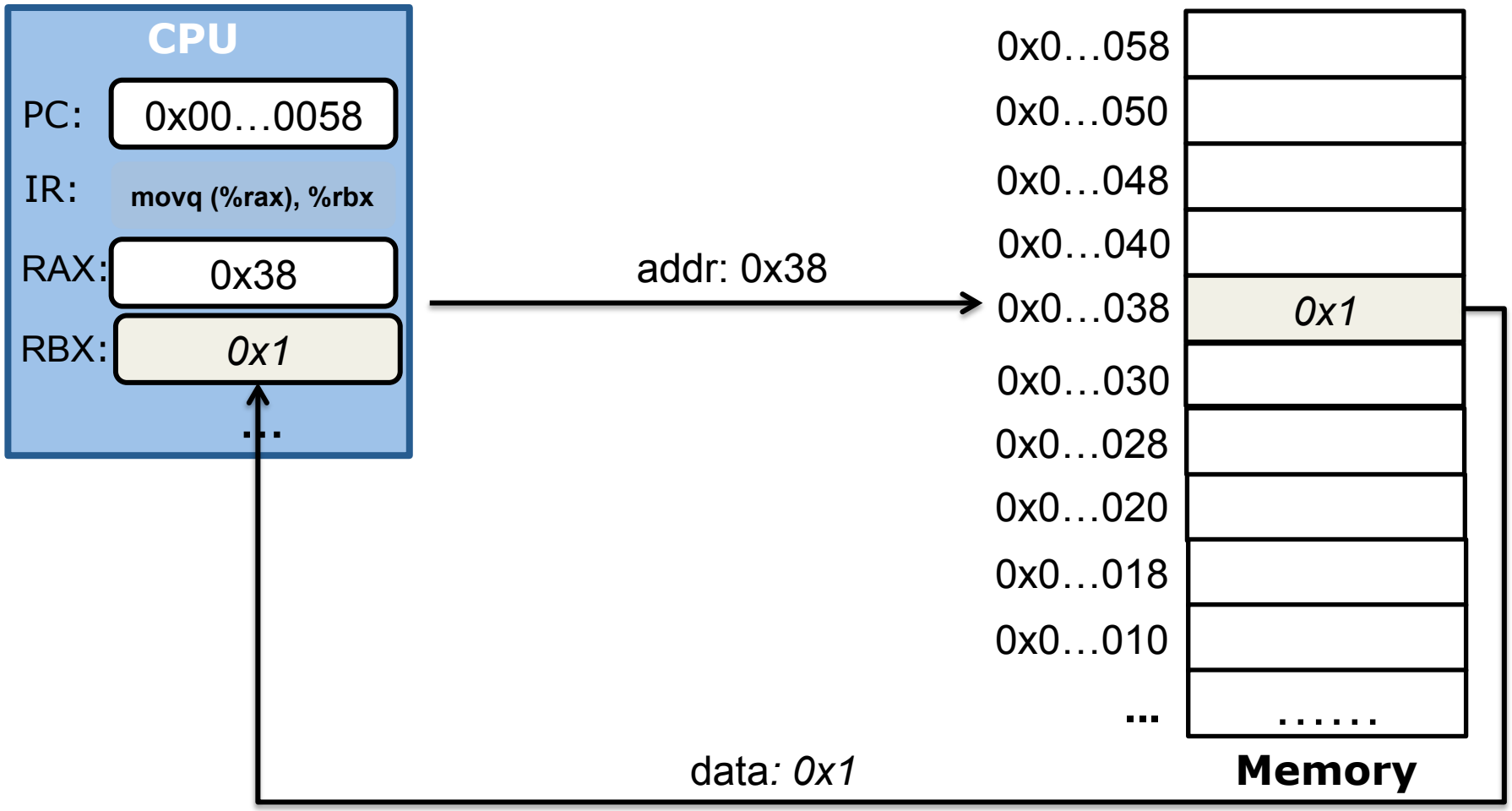
Isolation



Upon program startup, OS creates a process. Process asks for OS services through system calls (*getpid, read, write*).

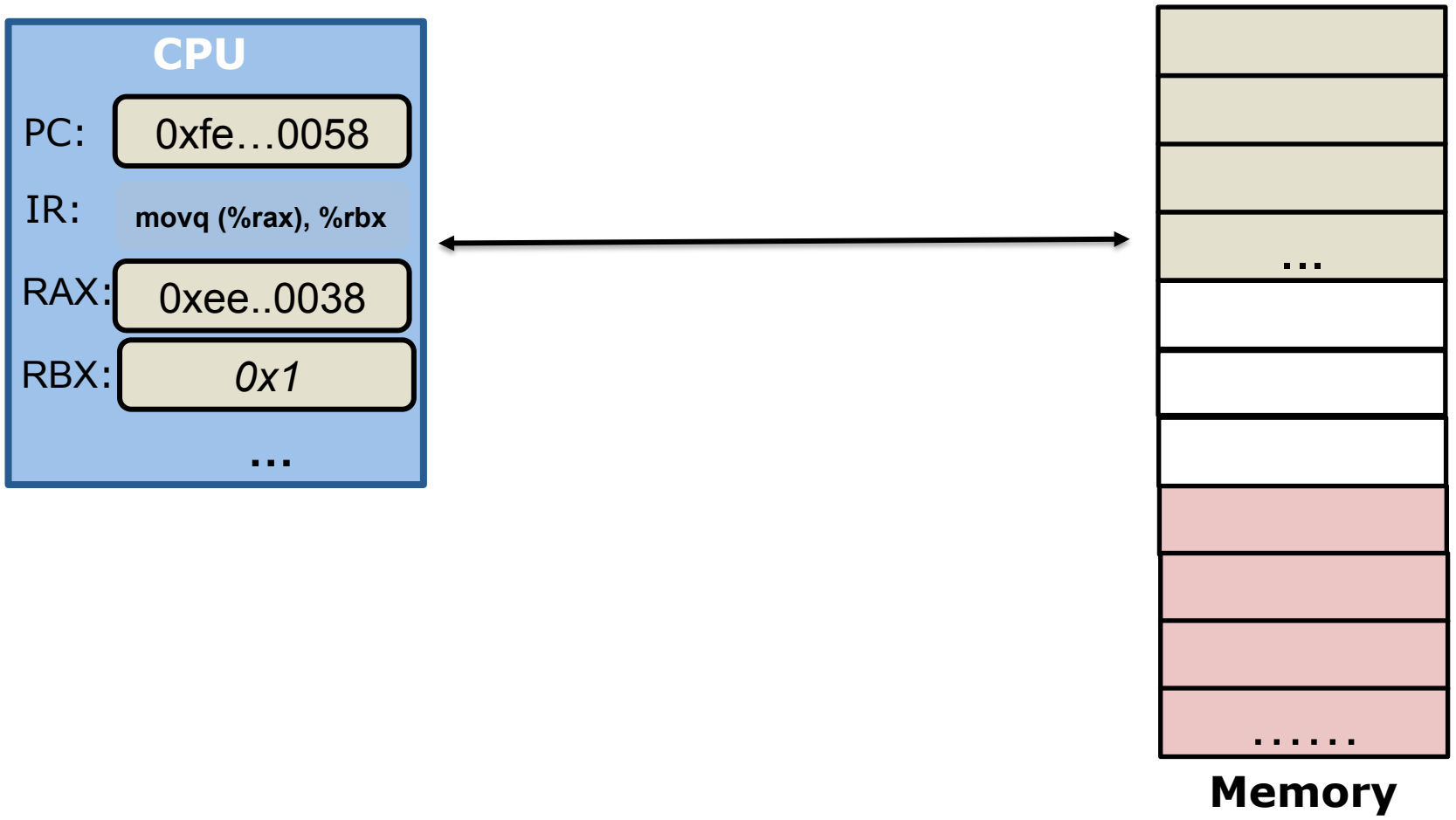
Syscalls are listed in manual section 2
type “man 2 getpid”

Our simplified "Mental Model" of program execution



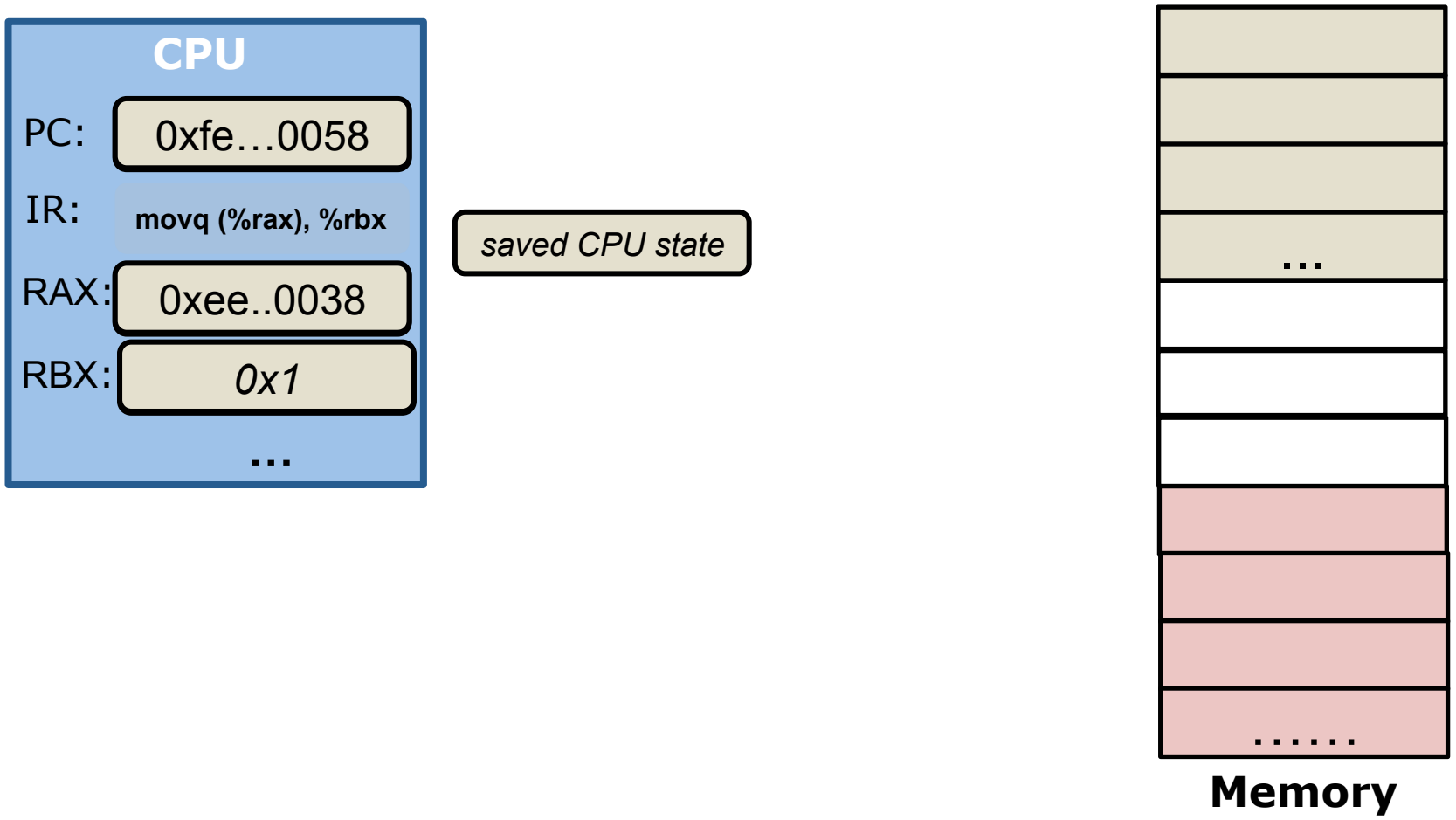
Question: how does a CPU execute >1 programs "simultaneously"?

Sharing CPU: time multiplexing



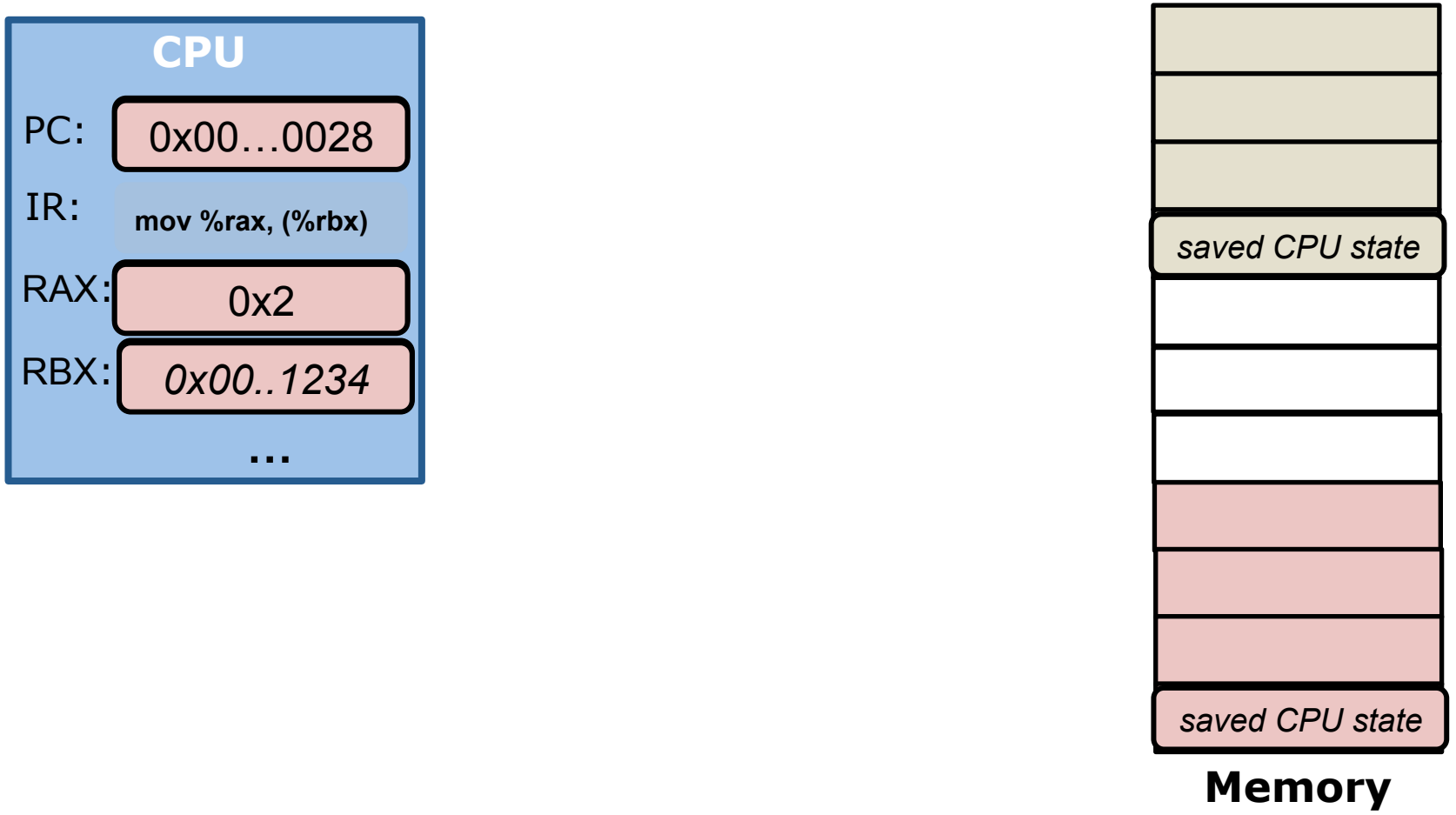
1. OS assigns different processes different memory regions

Sharing CPU: time multiplexing



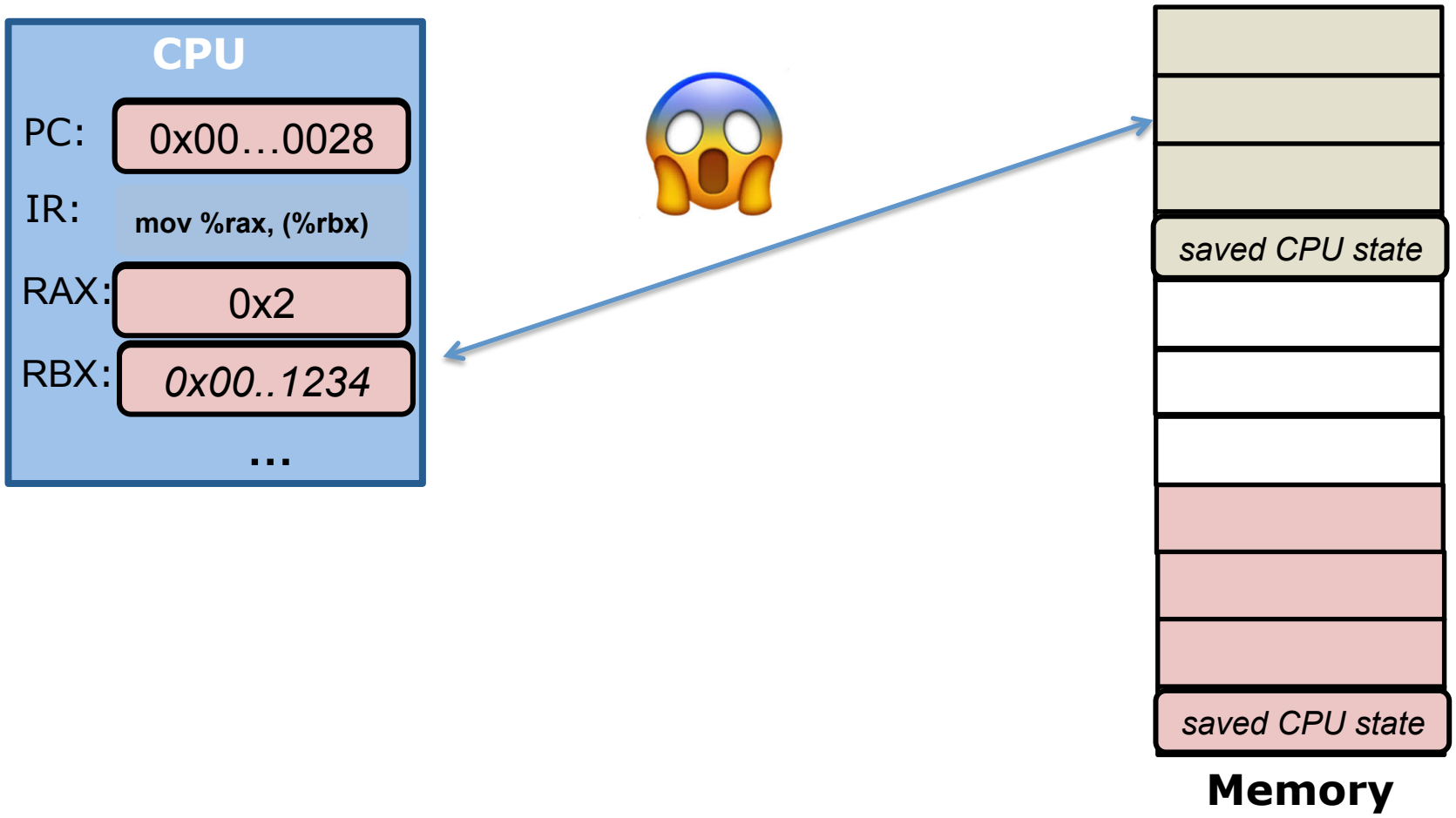
2. OS saves/restores CPU state in memory to switch execution of one process to another every ~10ms

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



Challenge: how to prevent a process from reading/writing another process's memory?!

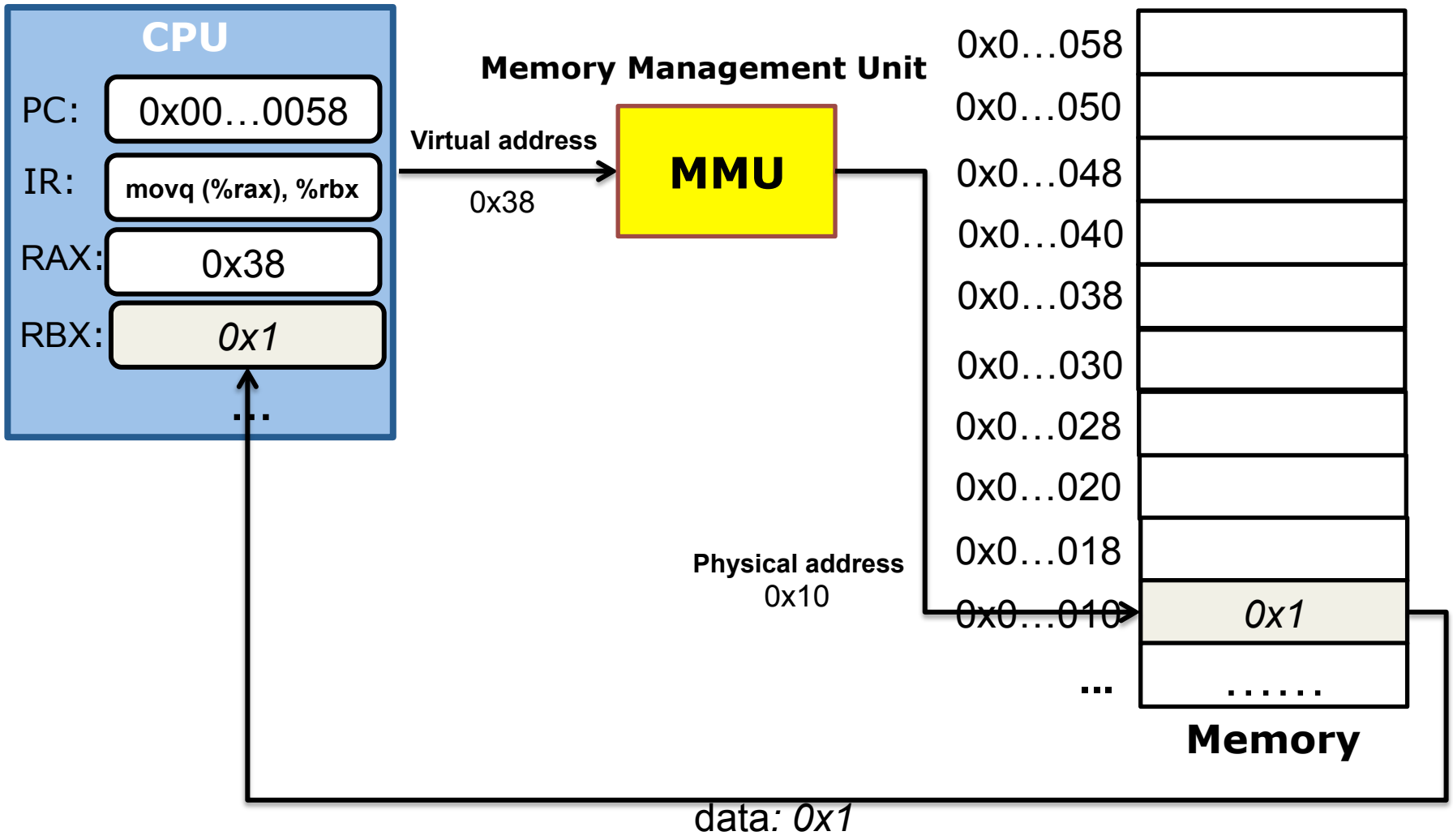
Processes need to share memory safely

- (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



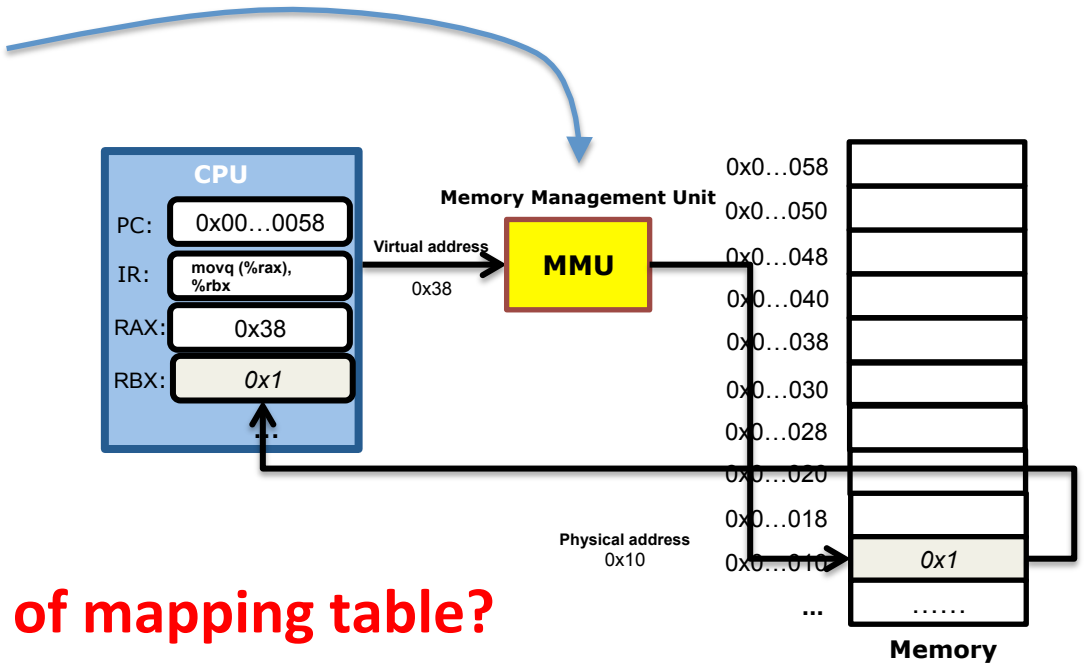
Address Translation – Strawman

Provide MMU with a mapping table at byte granularity

- Map each virtual address into a physical address

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

mapping table



Question: What is the size of mapping table?

For 64-bit address, size is 2^{64}

data: *0x1*

Address Translation – Page

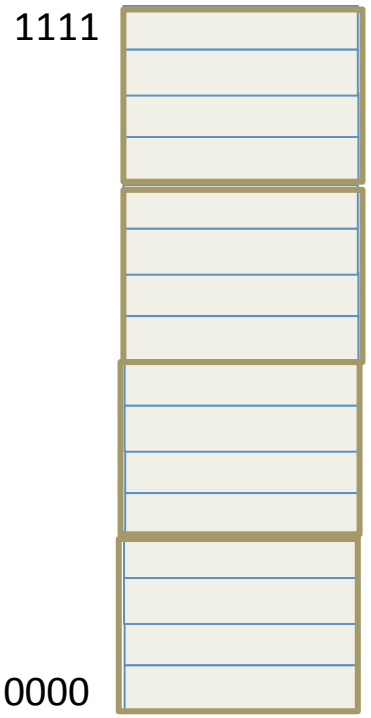
Problem: Mapping table too big

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

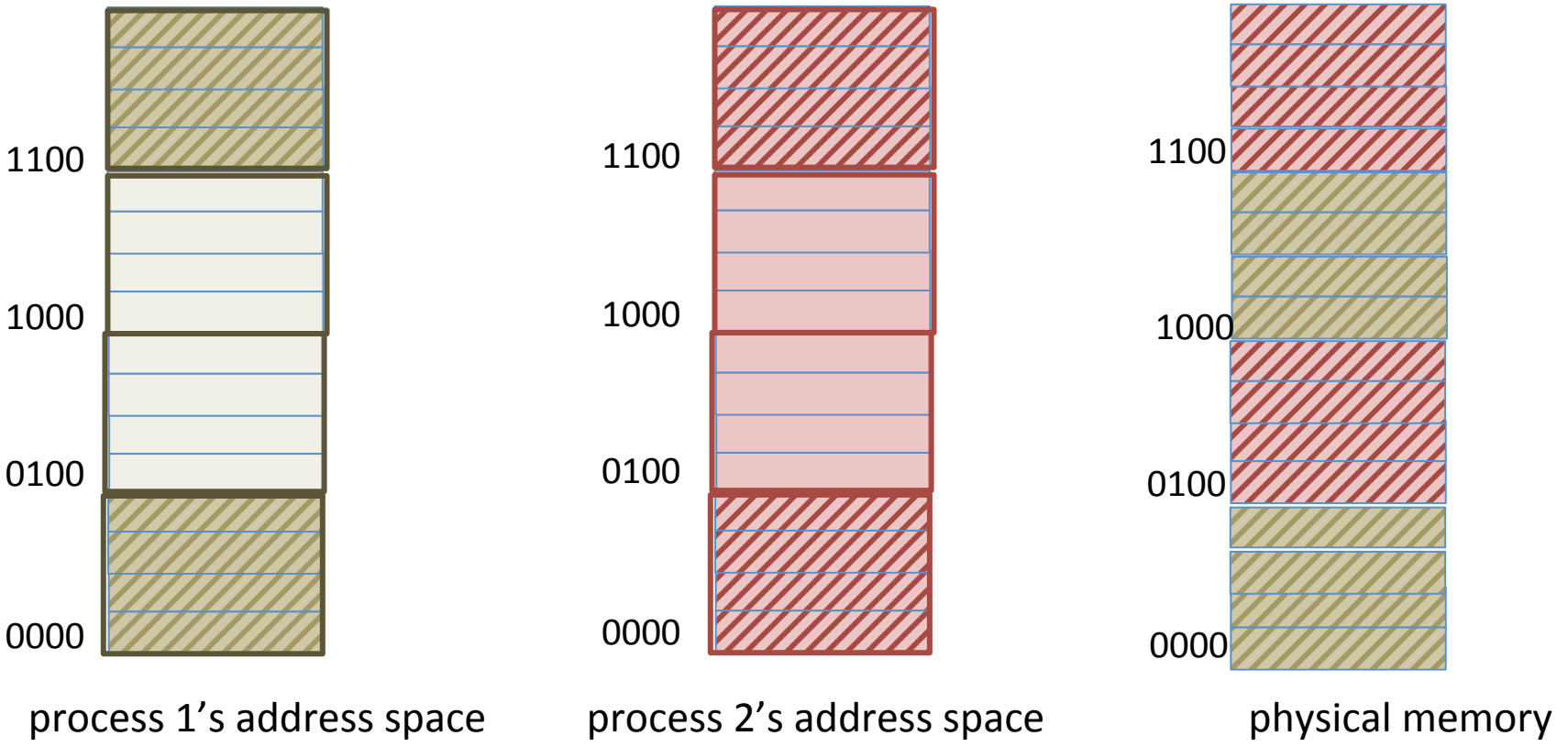


Question: for 32-bit address, 4KB page size, how many pages?

Answer: $2^{32}/2^{12}=2^{20}$ pages

Paging example: 4-bit address

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



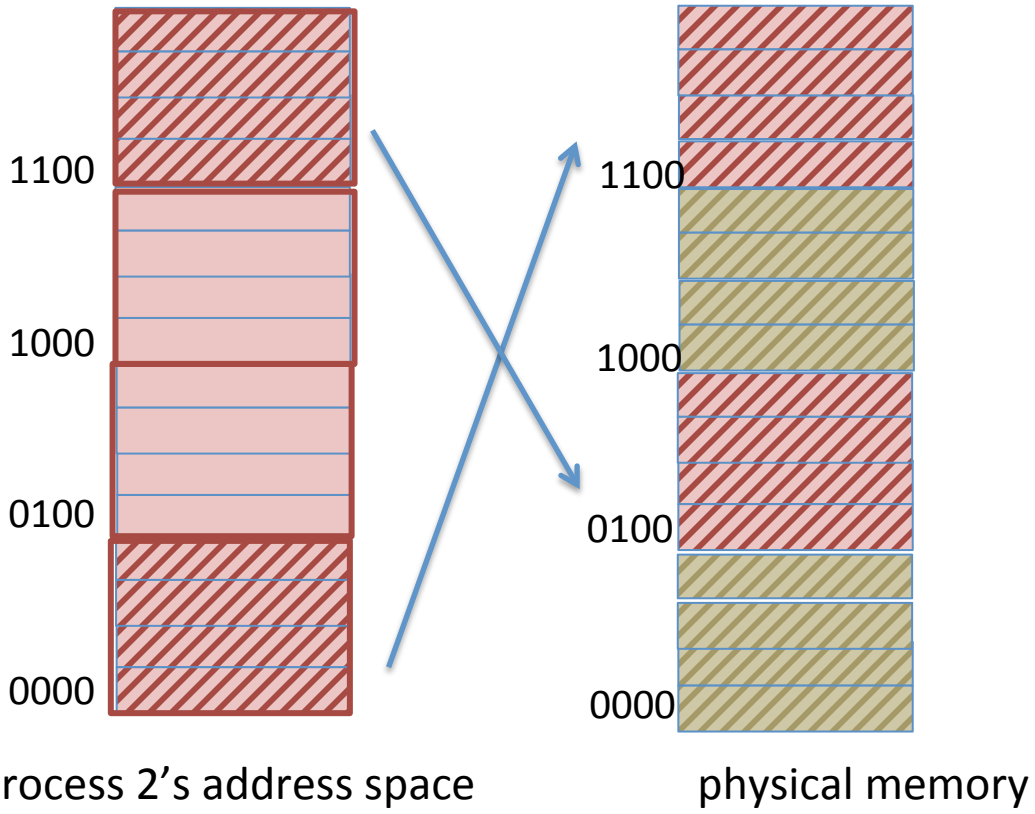
Paging example: 4-bit address

- What does the mapping table look like?

mapping for process 2:

Virtual Page	Physical Page
0000	1100
0100	--
1000	--
1100	0100

Can we eliminate this column?



Paging example: 4-bit address

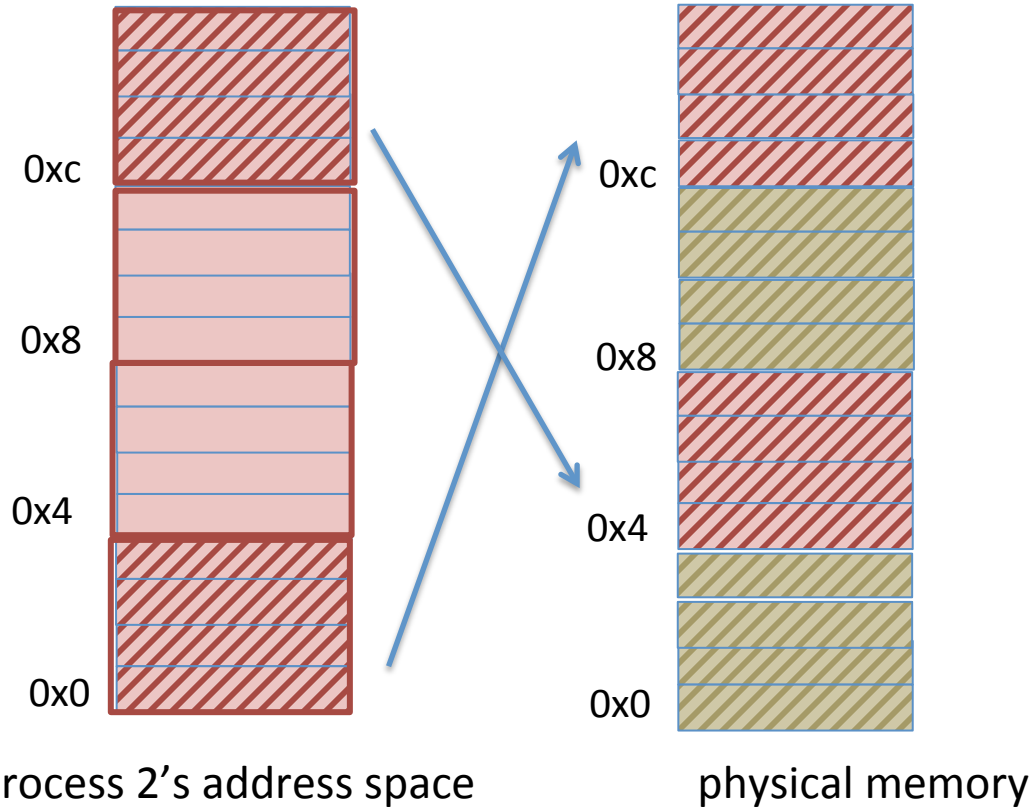
- What does the mapping table look like?
  - An array of integers (Page Table Entry, PTE)

index of array is
virtual page number

0	1101
1	0000
2	0000
3	0101

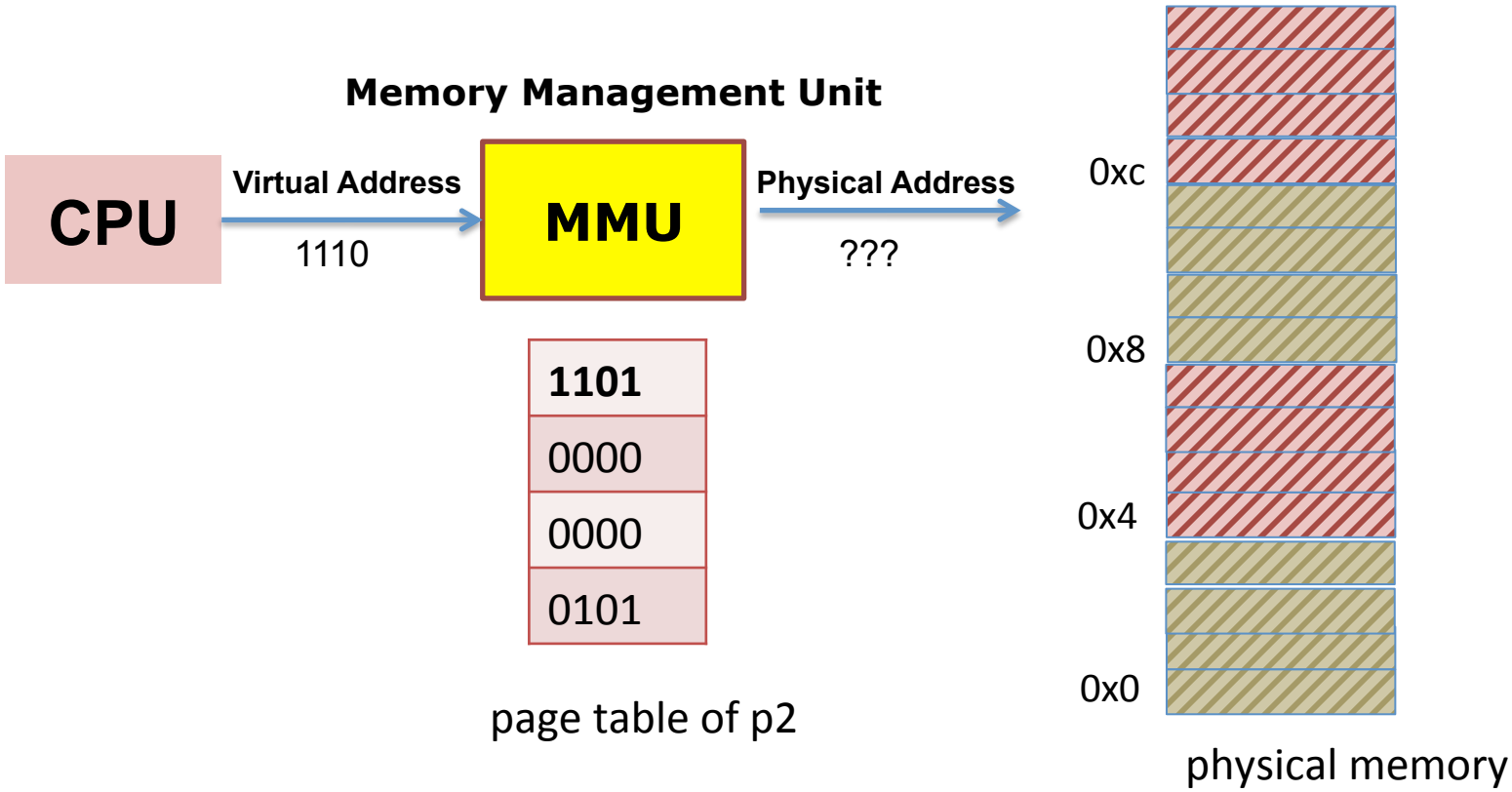
PTE is physical page number

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



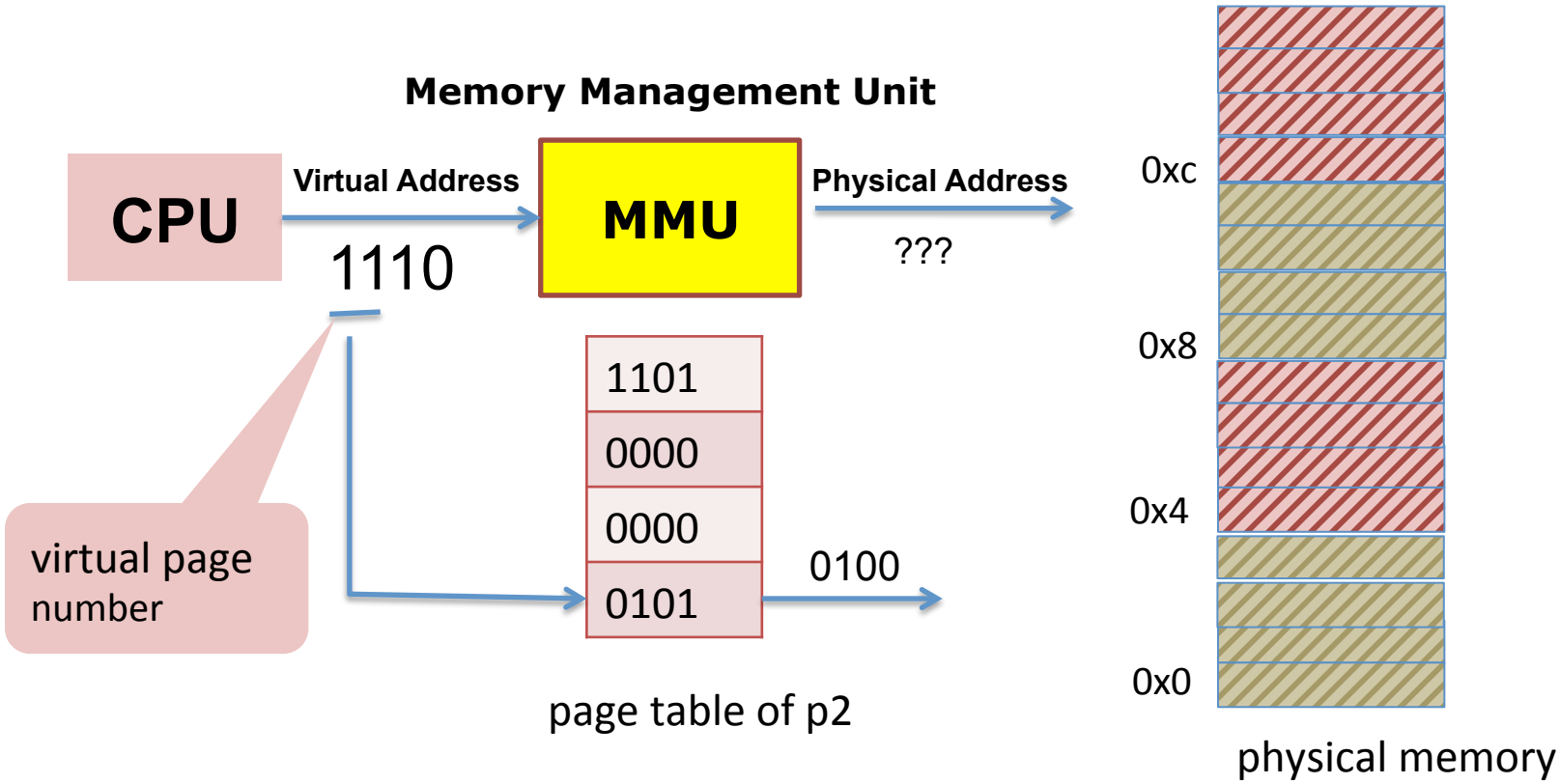
Paging example: 4-bit address

- How to translate from virtual to physical address?



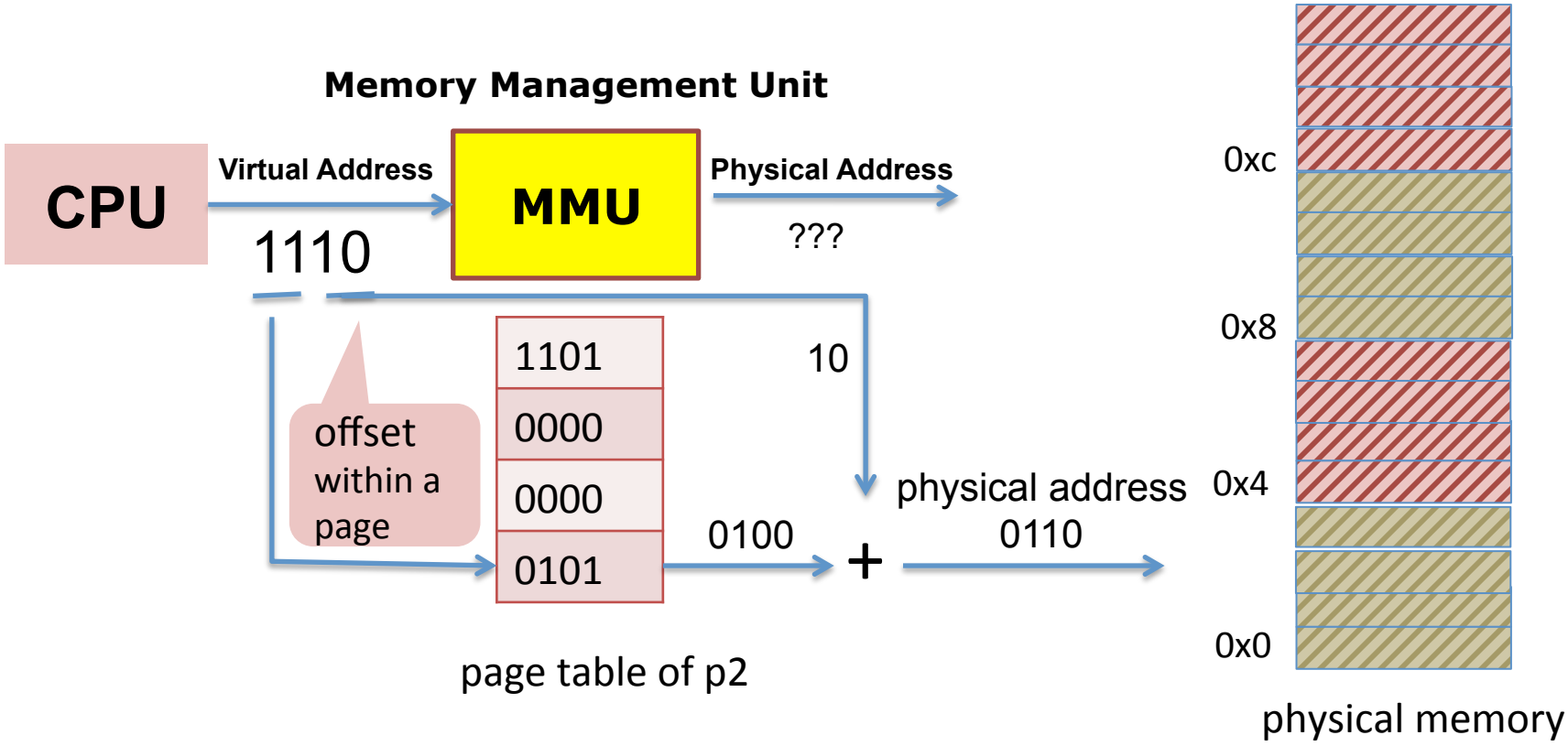
Paging example: 4-bit address

- How to translate from virtual to physical address?



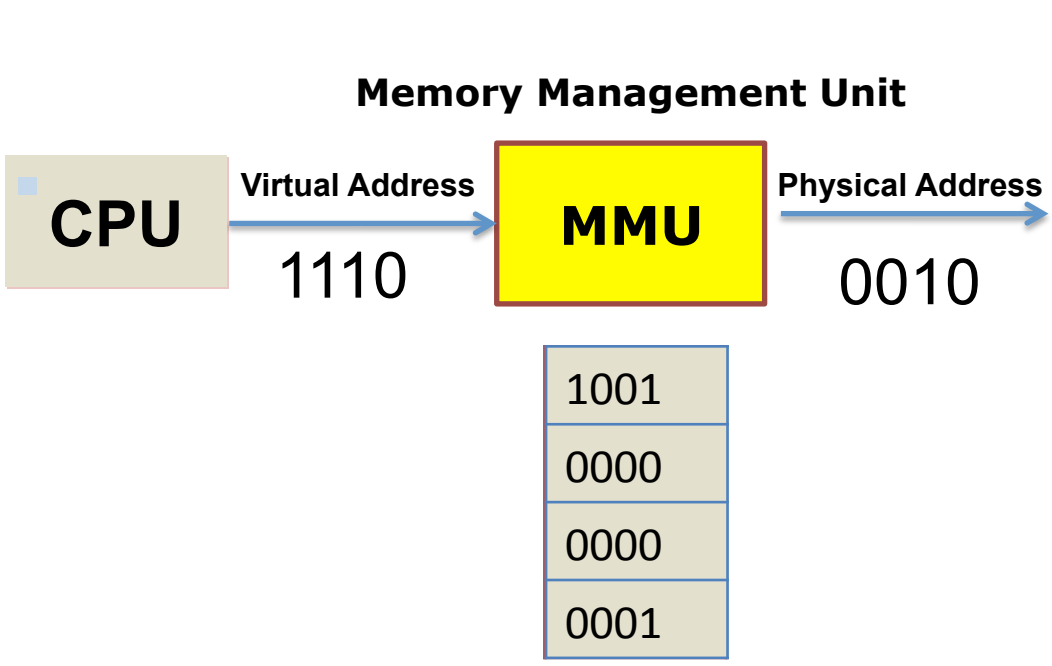
Paging example: 4-bit address

- How to translate from virtual to physical address?

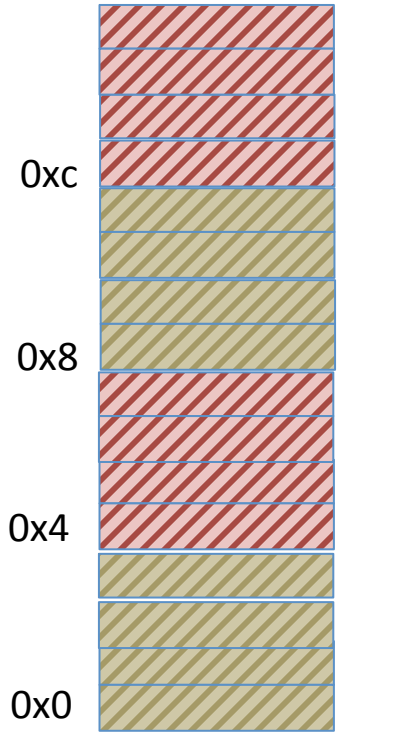


Each process has its own page table

OS switches CPU to execute p1 instead of p2

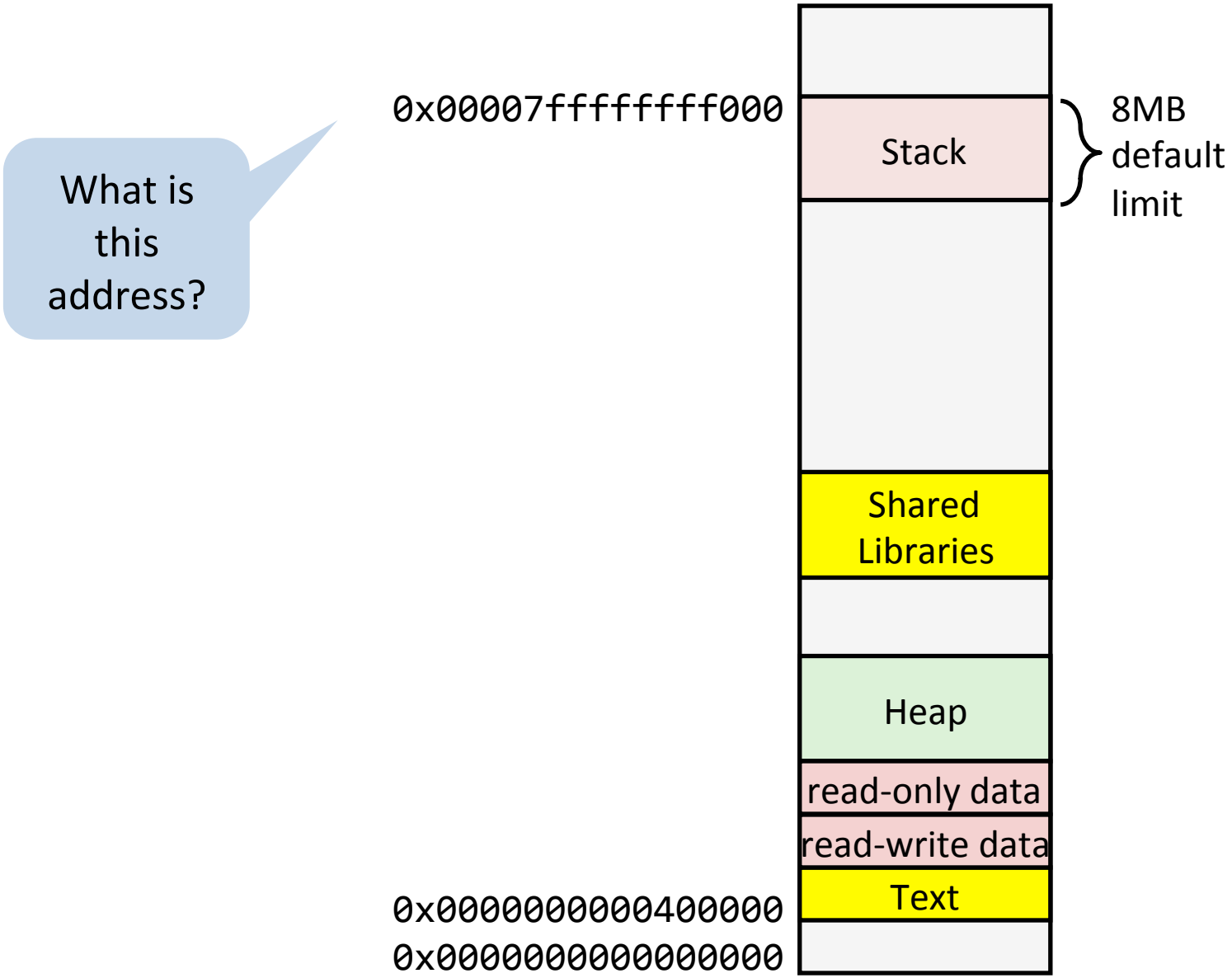


page table of p1



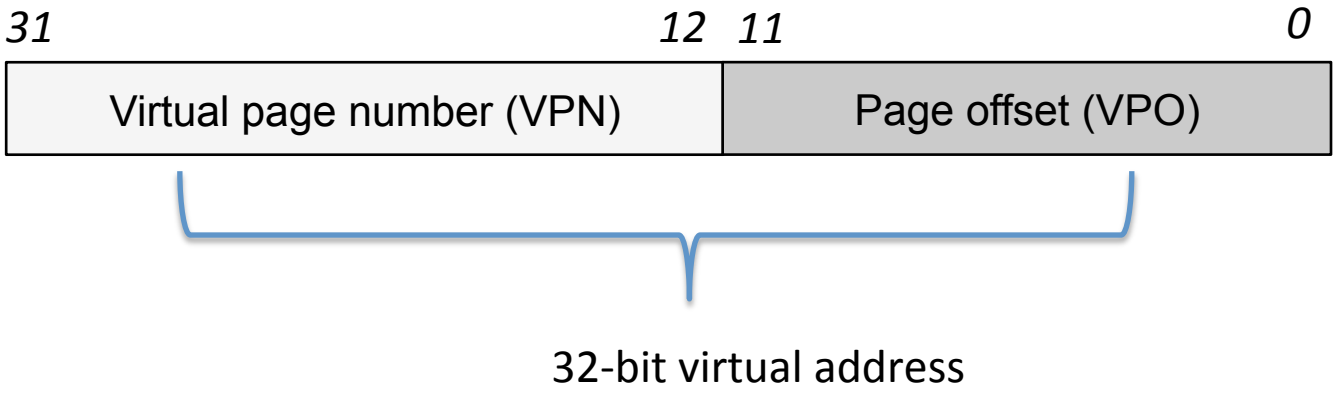
physical memory

Recap: Linux address space



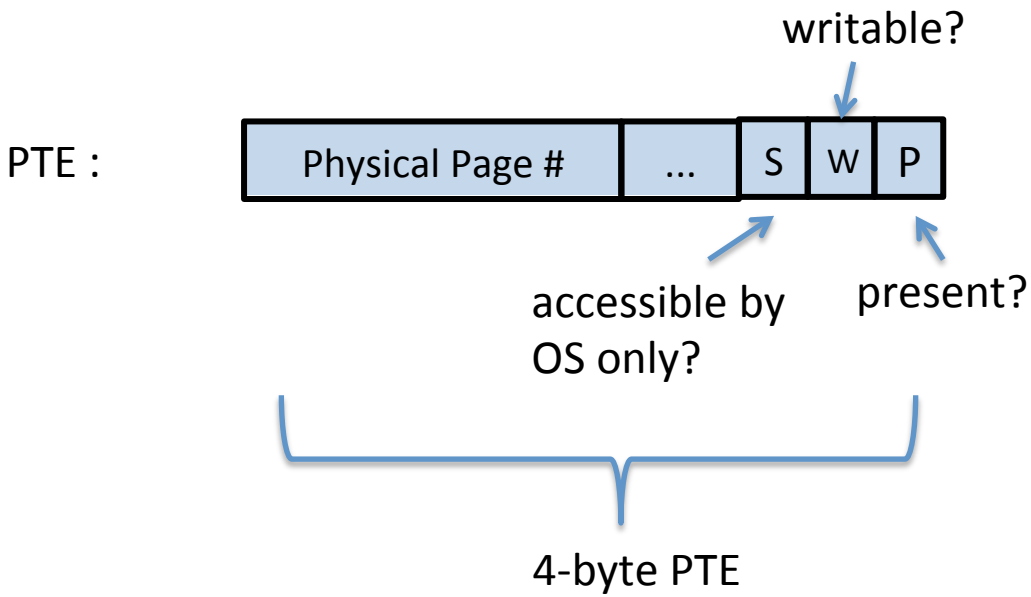
Address Translation: 32-bit address

- 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?

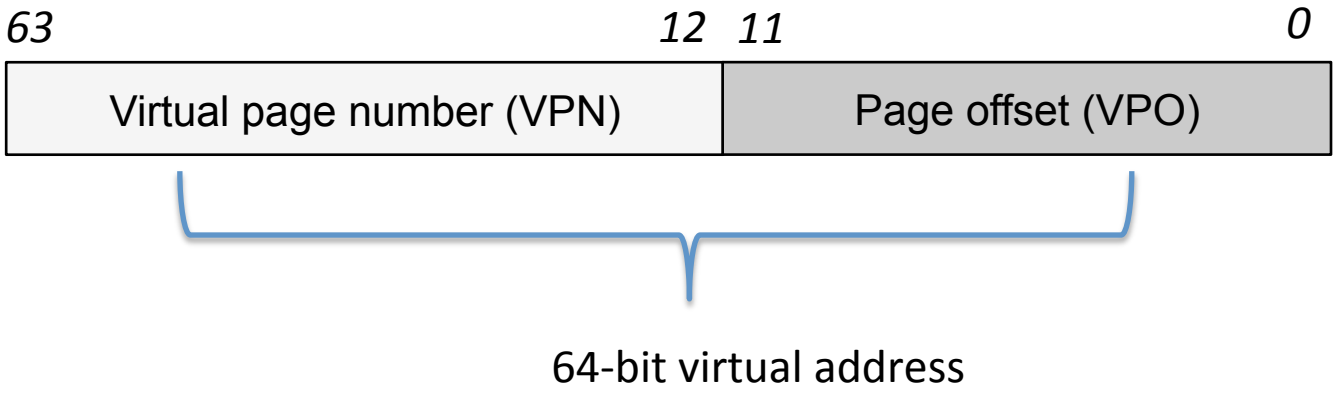


Address Translation: 32-bit address

- 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? $2^{20}*2 = 4MB$

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?



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

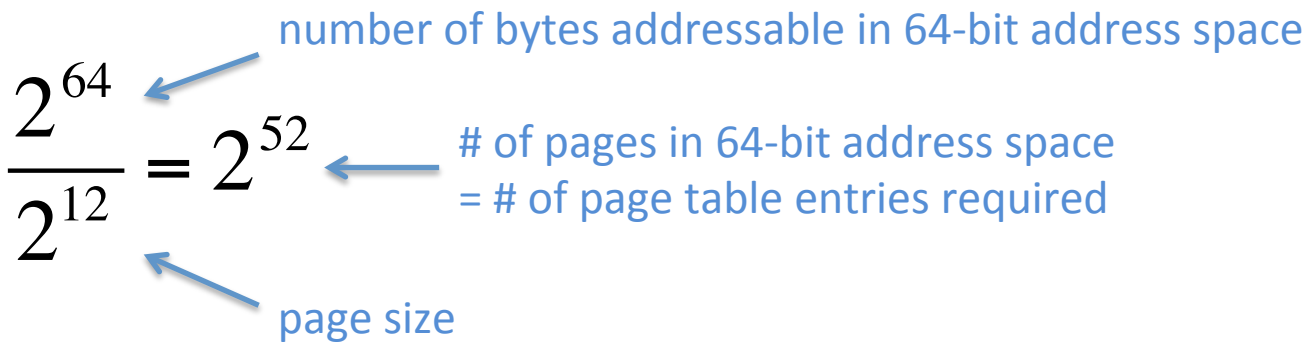
This lecture

- 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



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 Paging Example

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

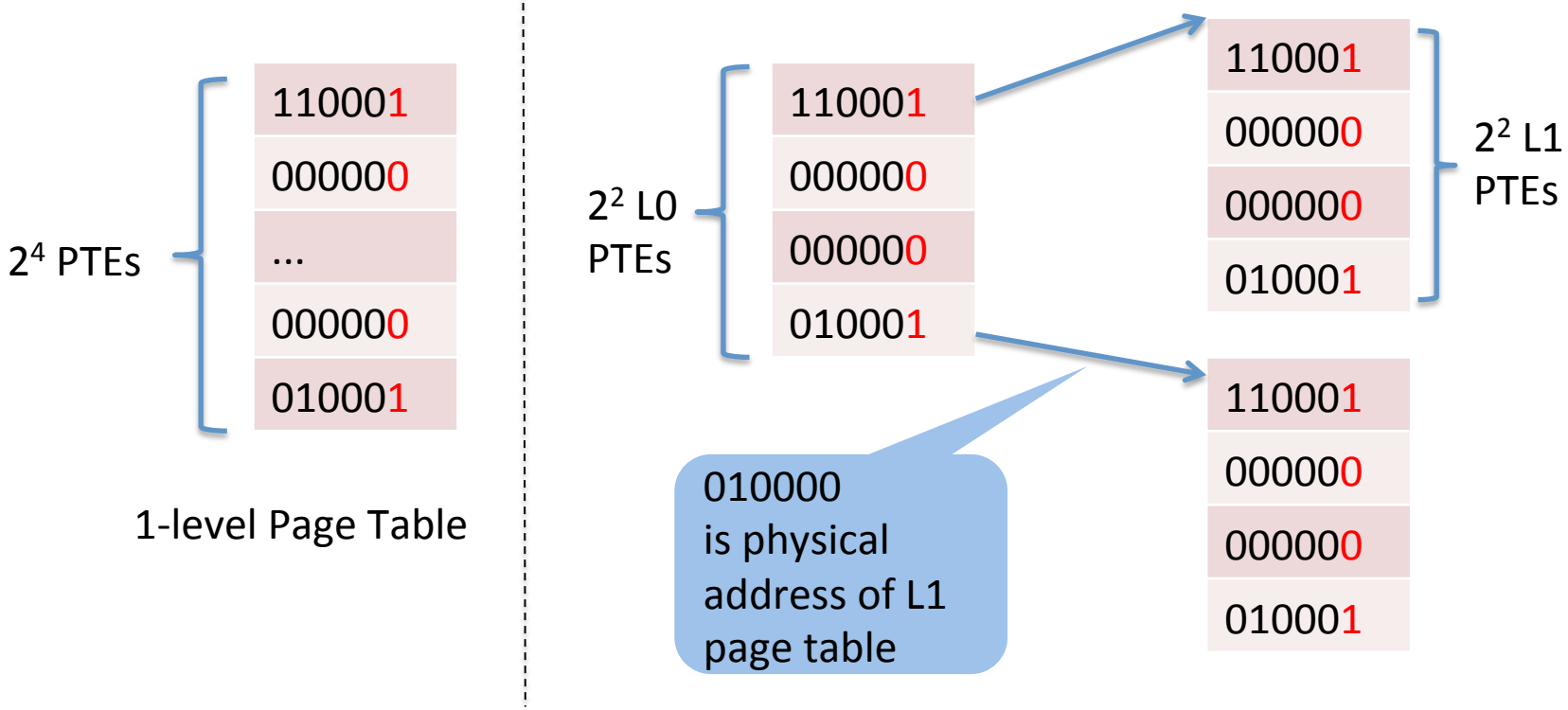
Suppose a process has only two 2 mapped virtual pages

1-level Page Table

2-level Paging Example

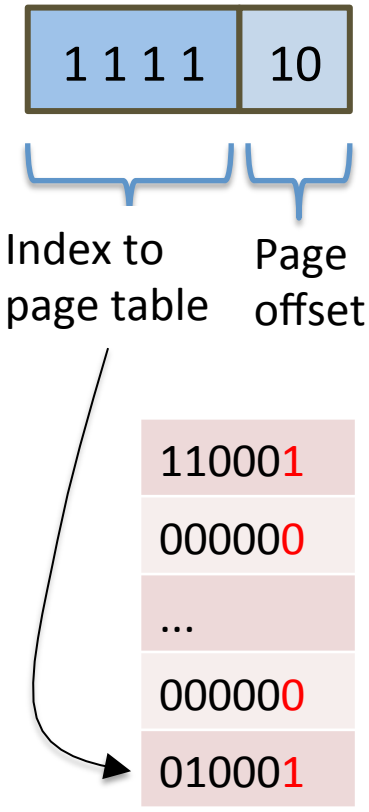
- 6-bit virtual and physical address, 4-byte page
- 2-level page table

Suppose a process has only two 2 mapped virtual pages



2-level Paging Example

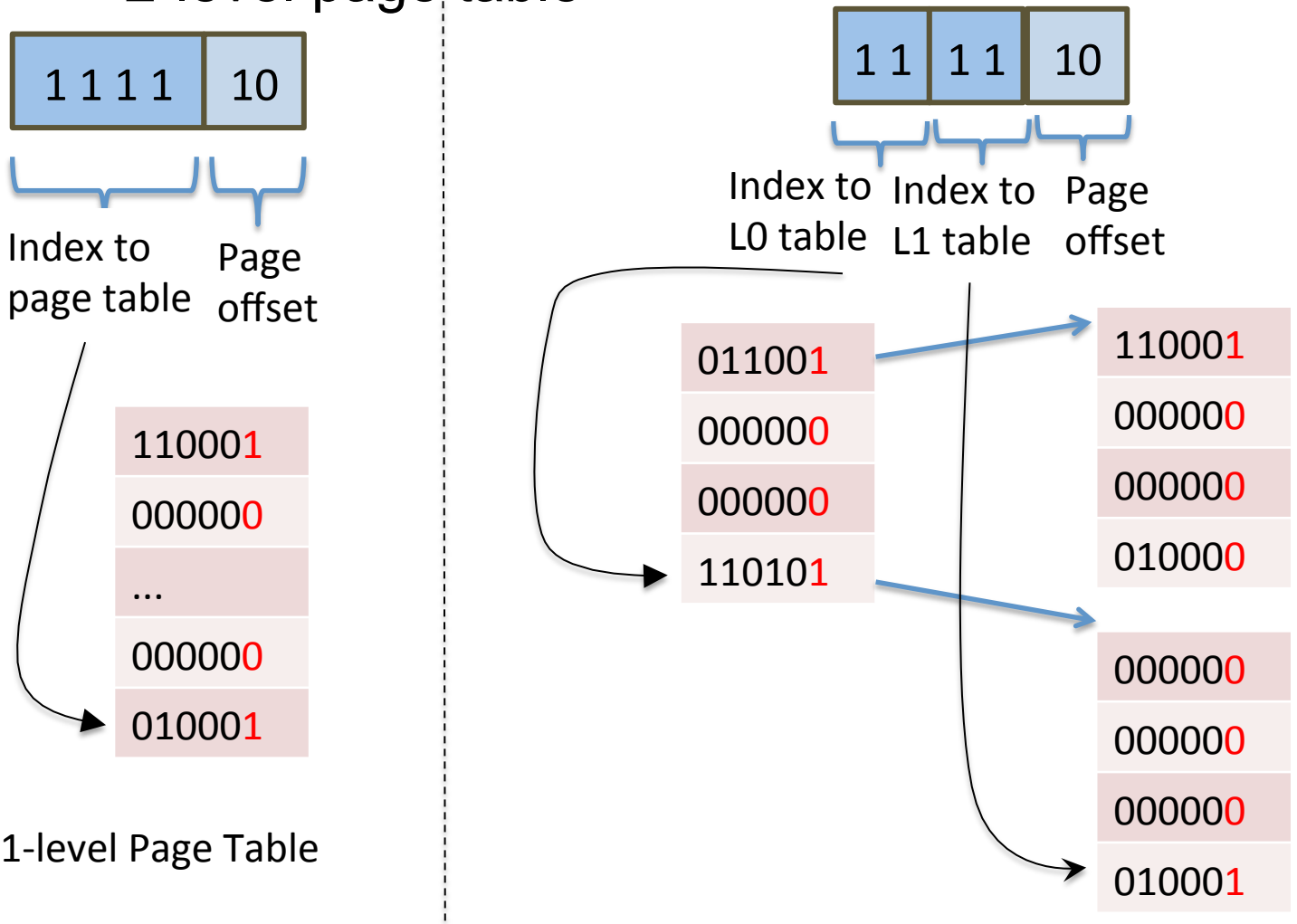
- how to perform address translation?
  - 1-level page table



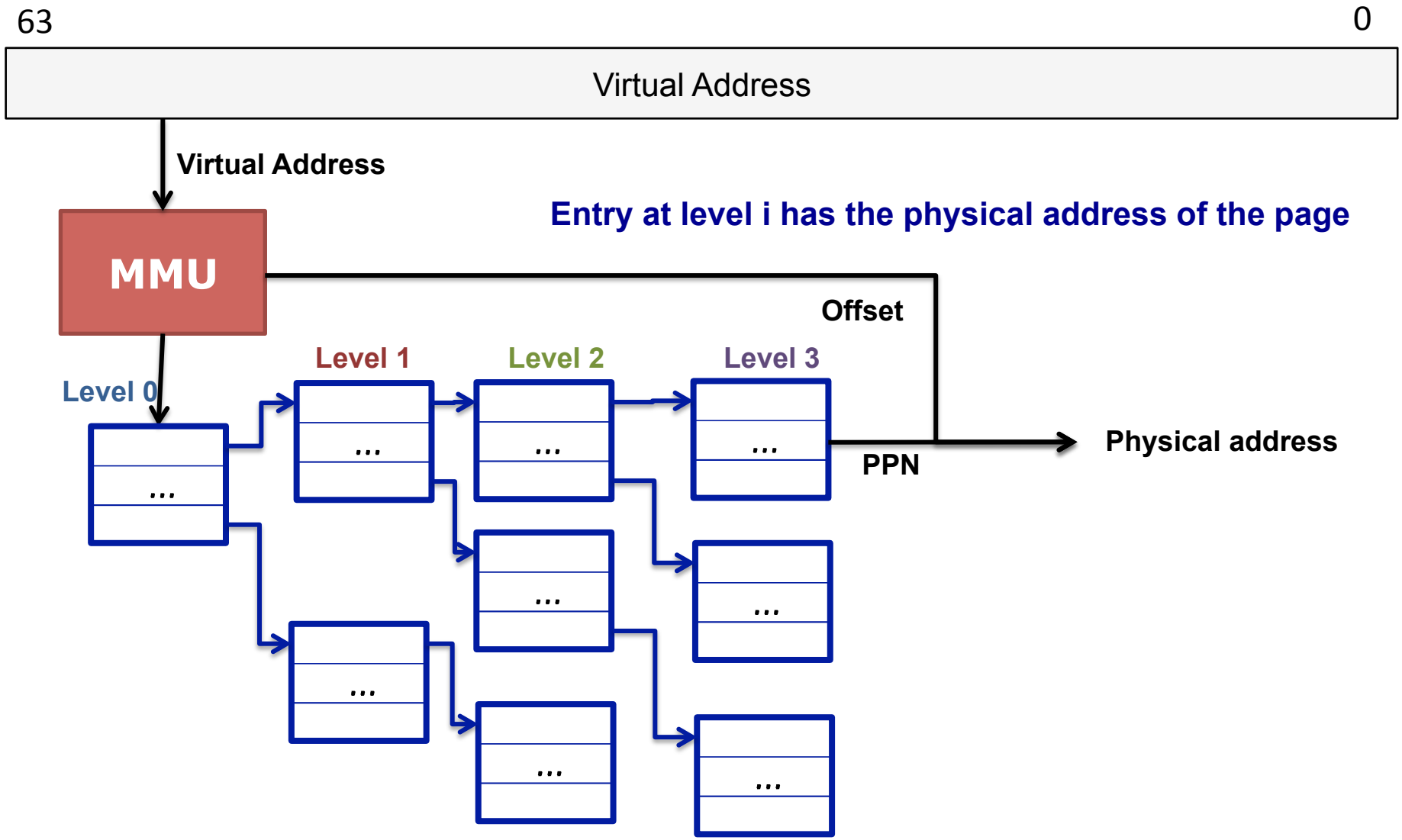
1-level Page Table

2-level Paging Example

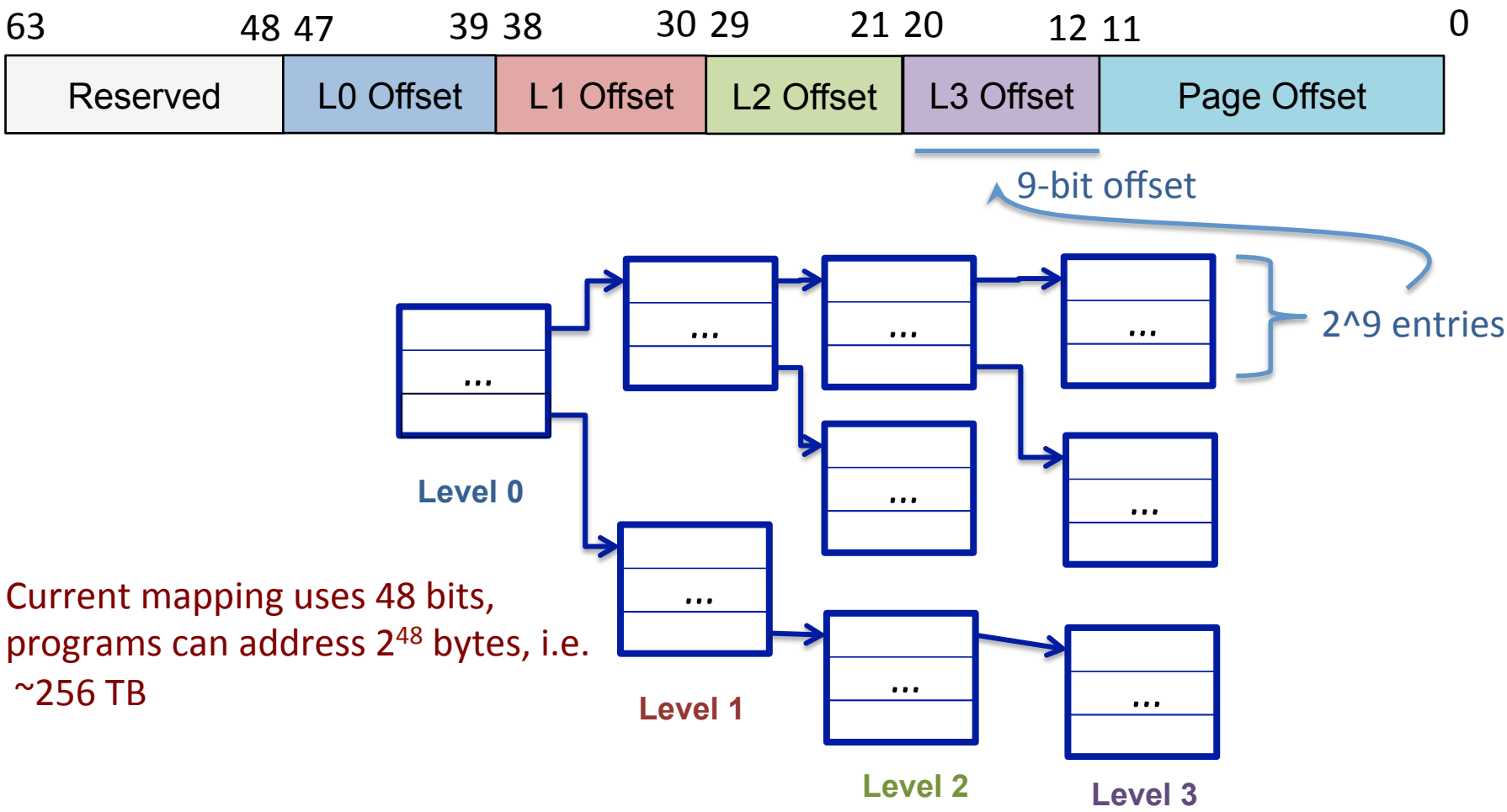
- how to perform address translation?
  - 2-level page table



X86_64 supports 4-level page table



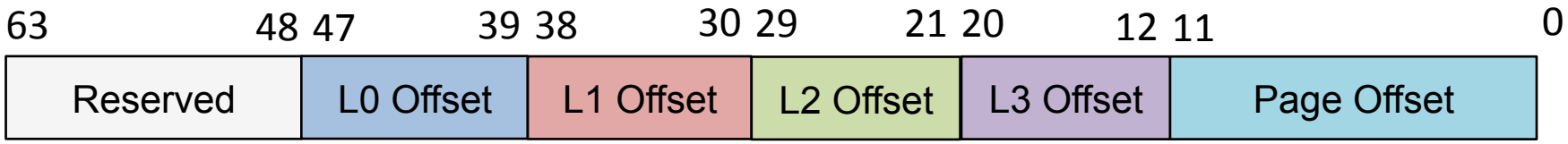
Multi-level page tables on X86_64

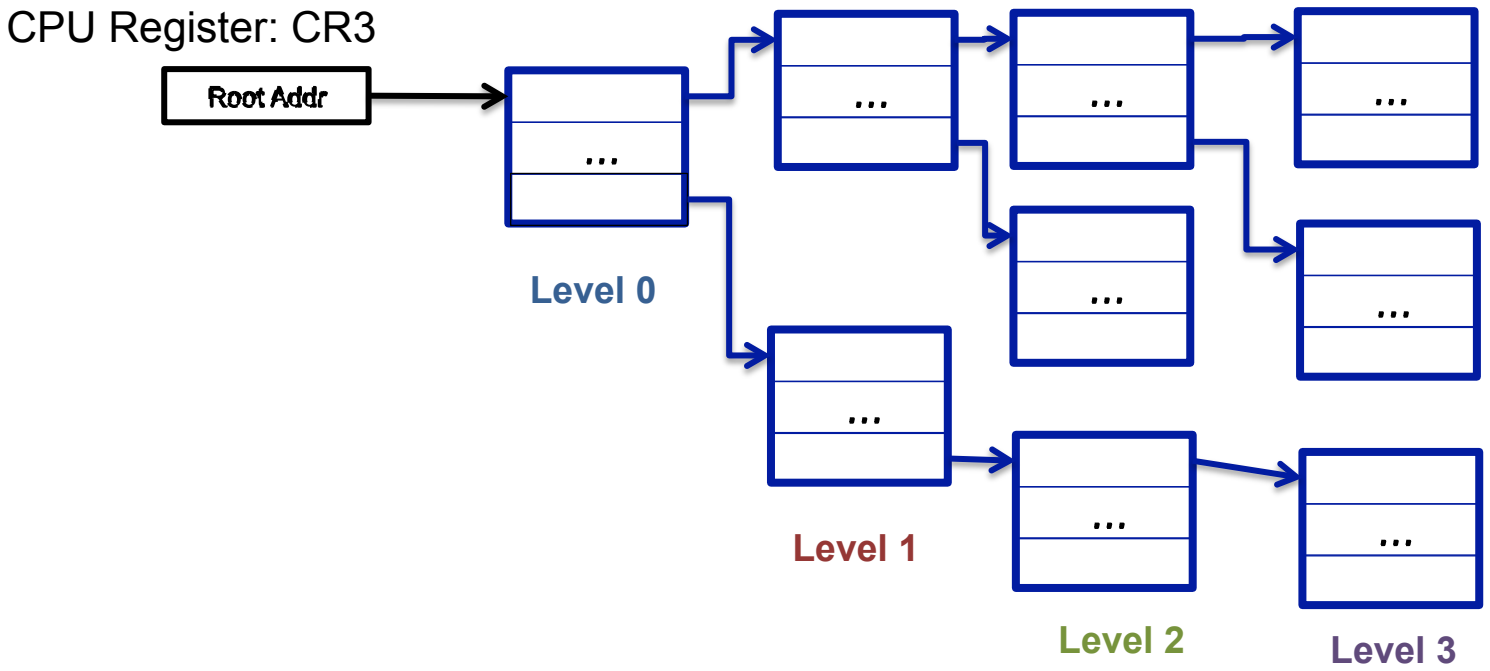


Current mapping uses 48 bits, programs can address 2^{48} bytes, i.e. ~256 TB

4-level page table

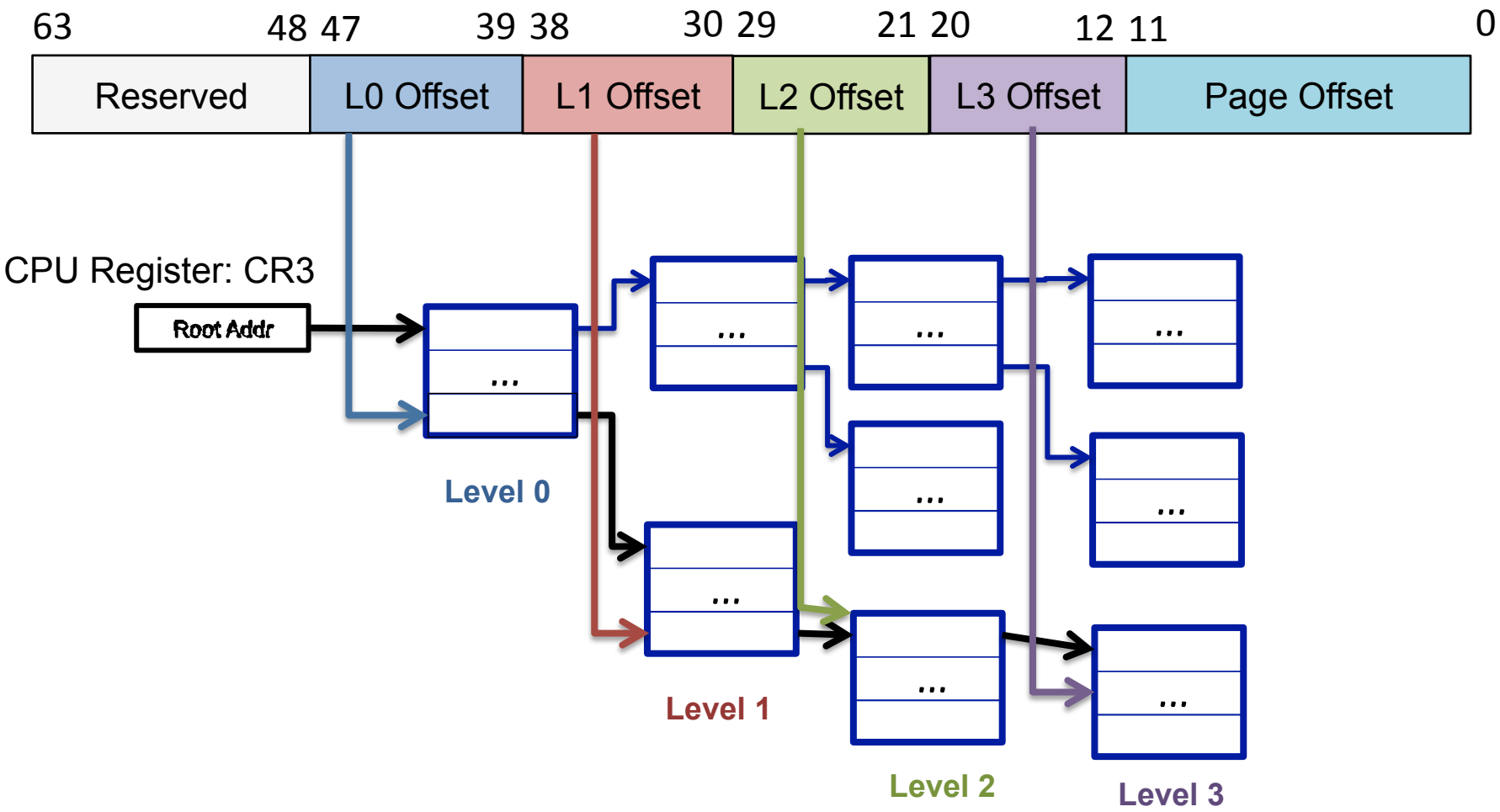
Multi-level page tables on X86_64





4-level page table

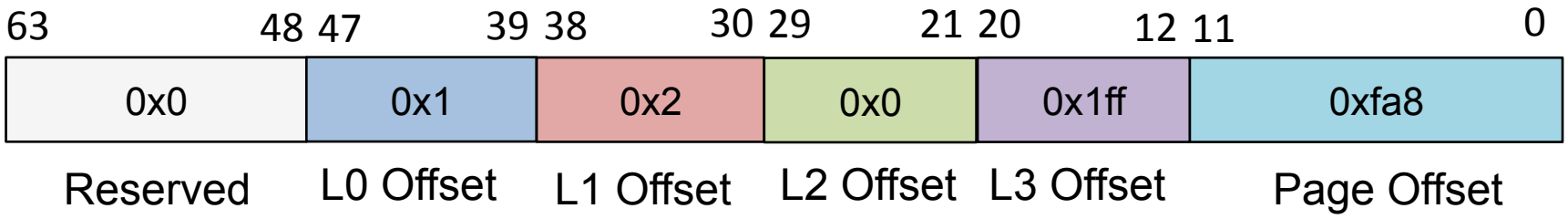
Multi-level page tables on X86_64



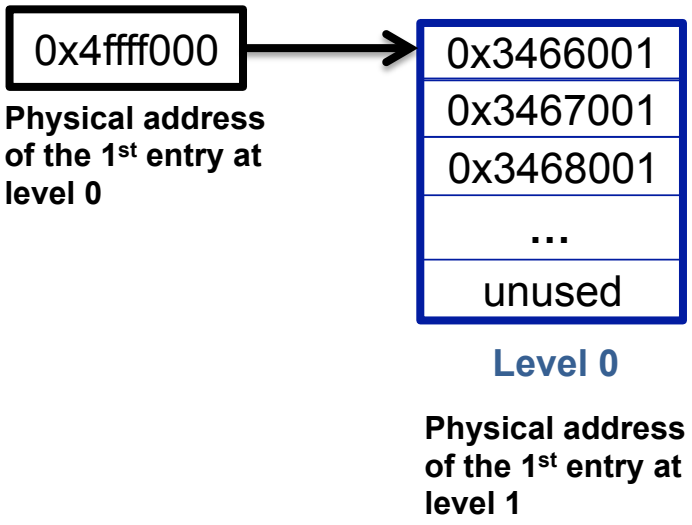
4-level page table

Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*

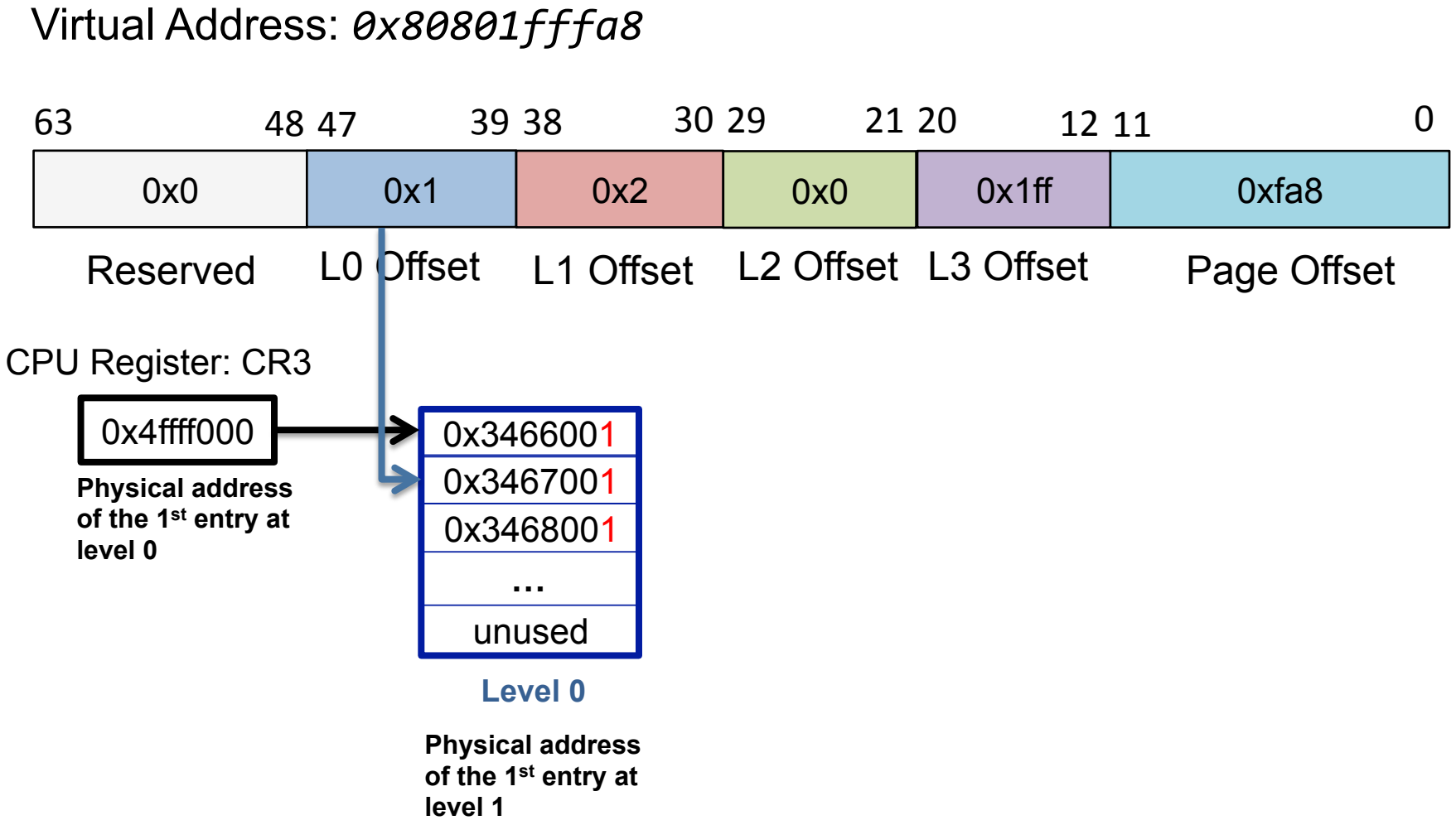


CPU Register: CR3



4-level page table

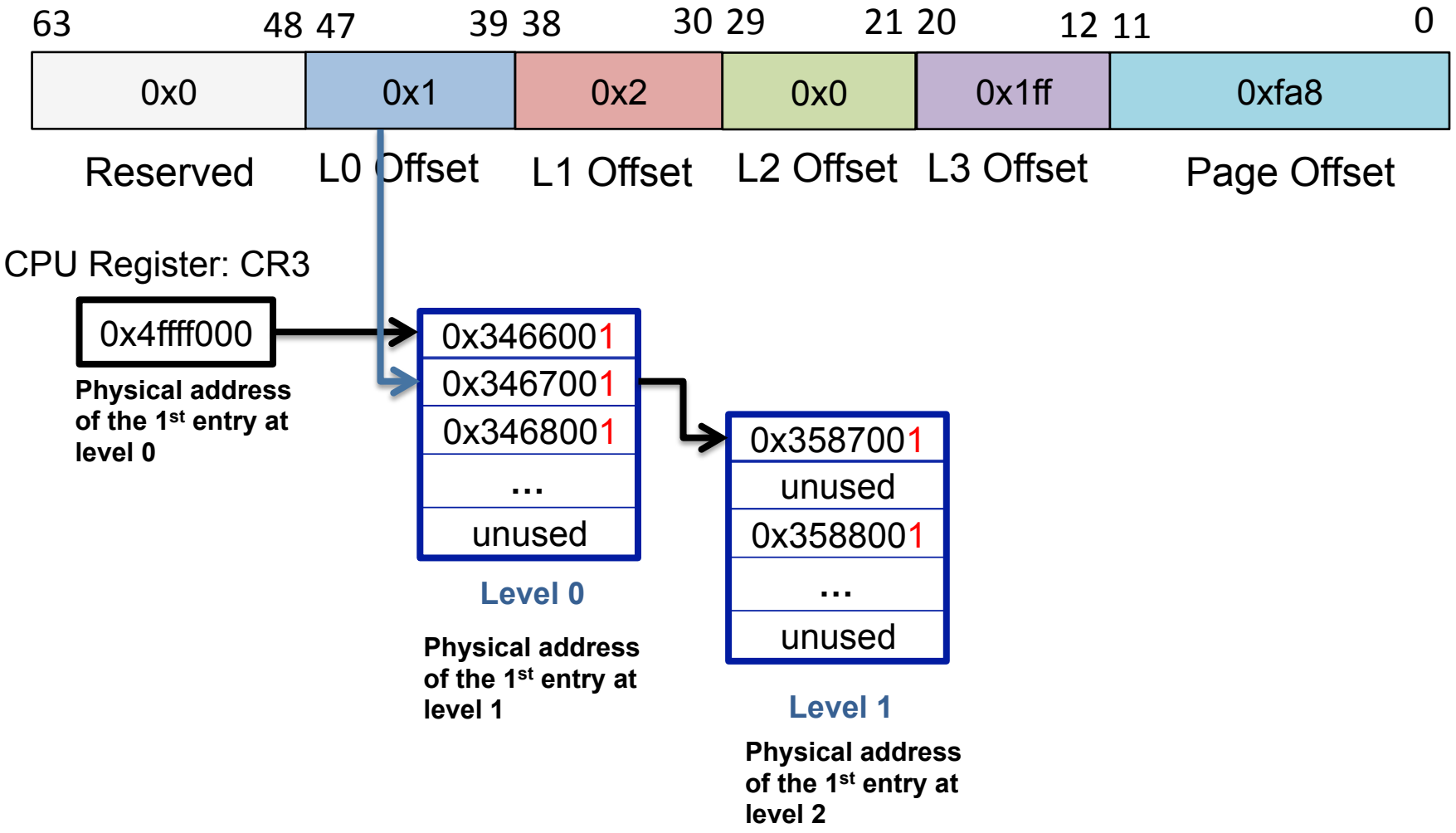
Multi-level page tables on X86_64



4-level page table

Multi-level page tables on X86_64

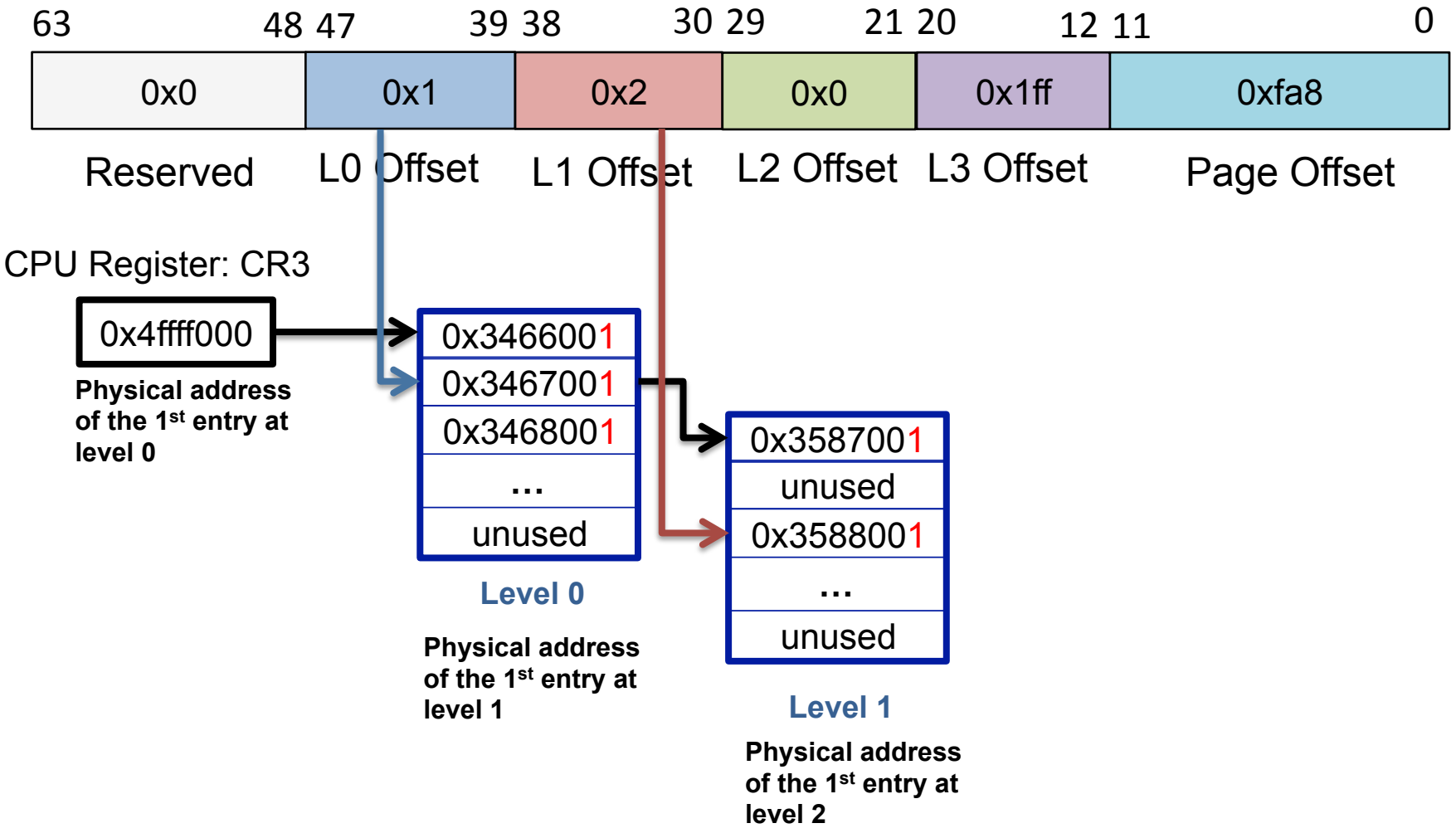
Virtual Address: *0x80801fffa8*



4-level page table

Multi-level page tables on X86_64

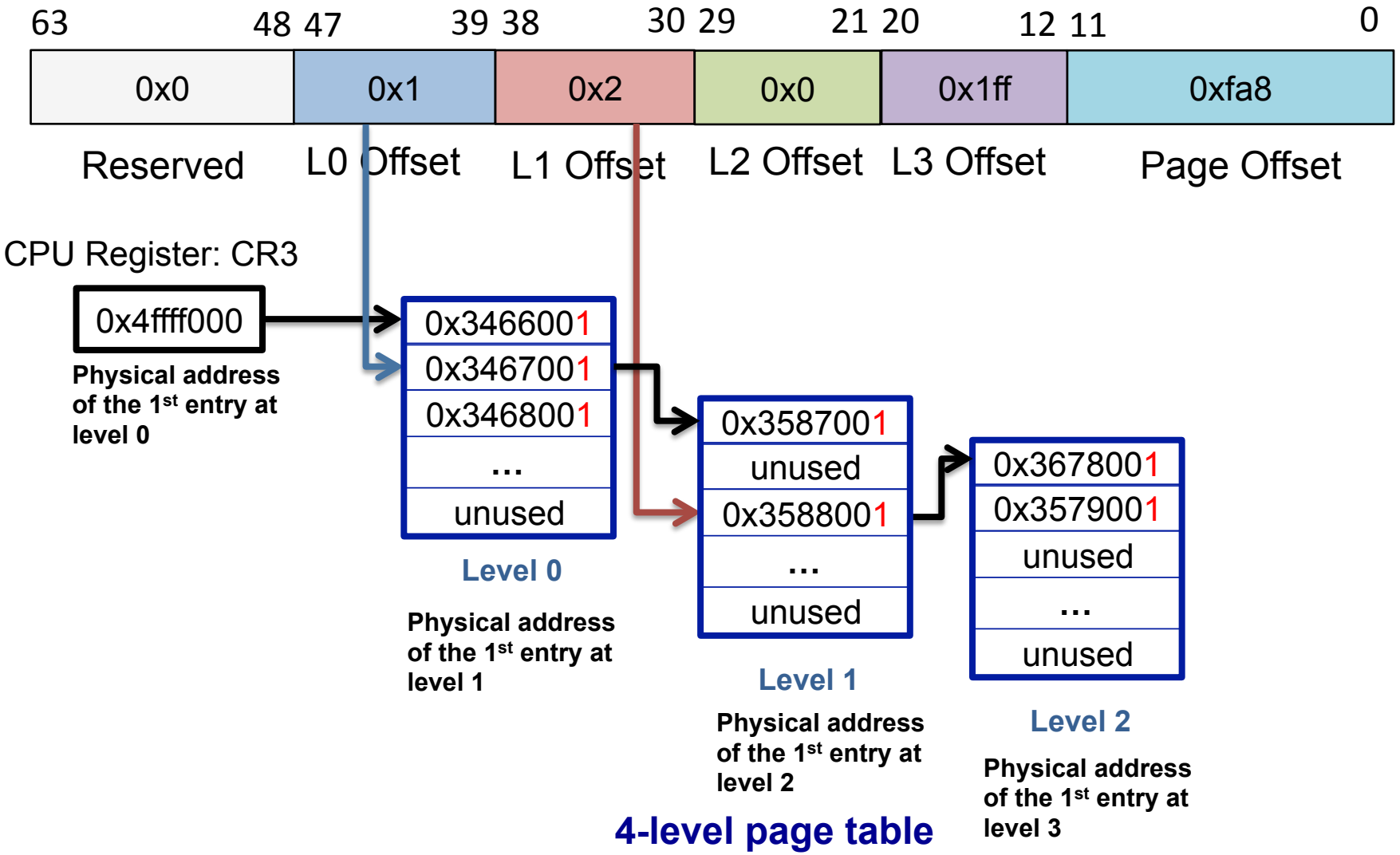
Virtual Address: *0x80801fffa8*



4-level page table

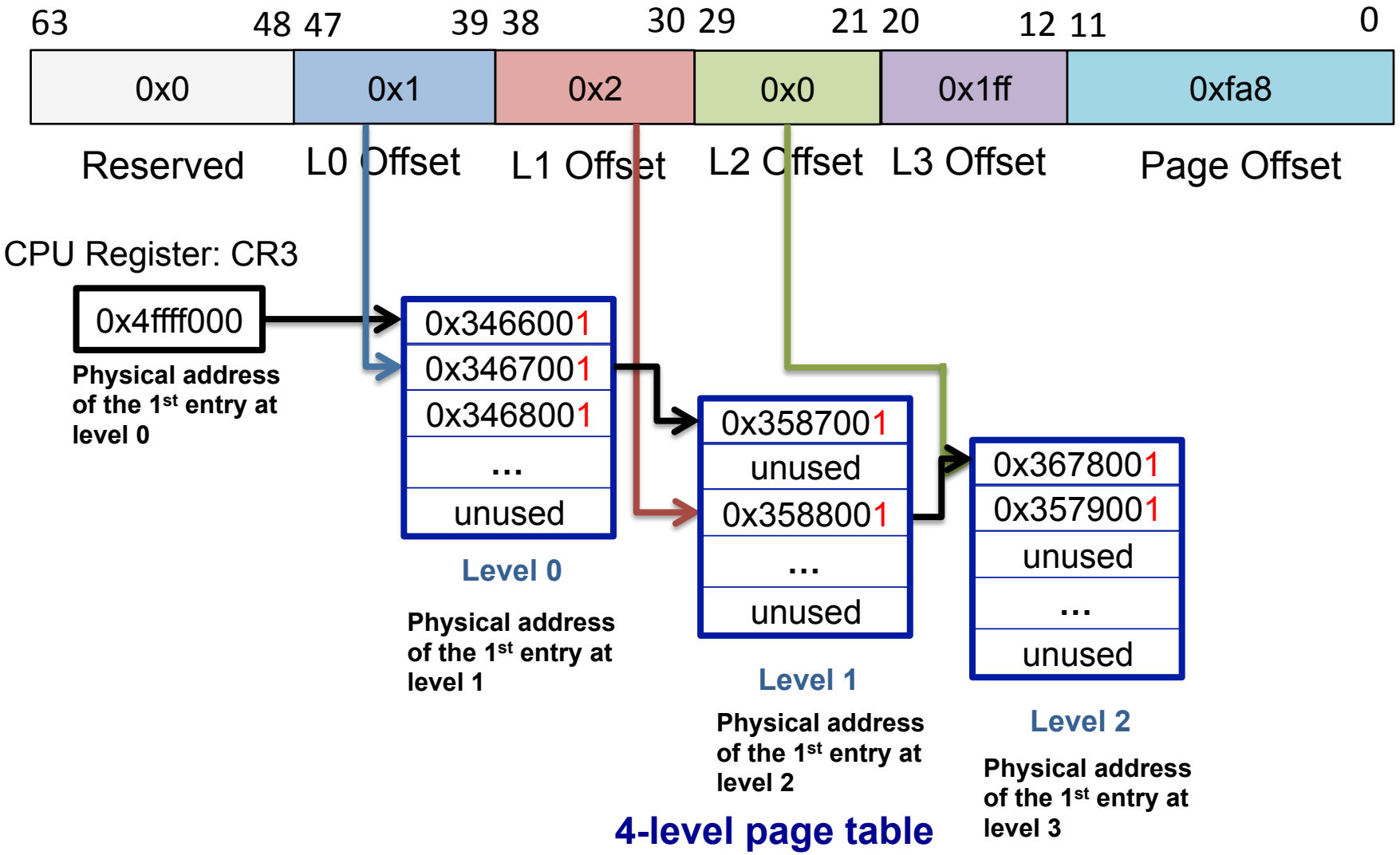
Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*



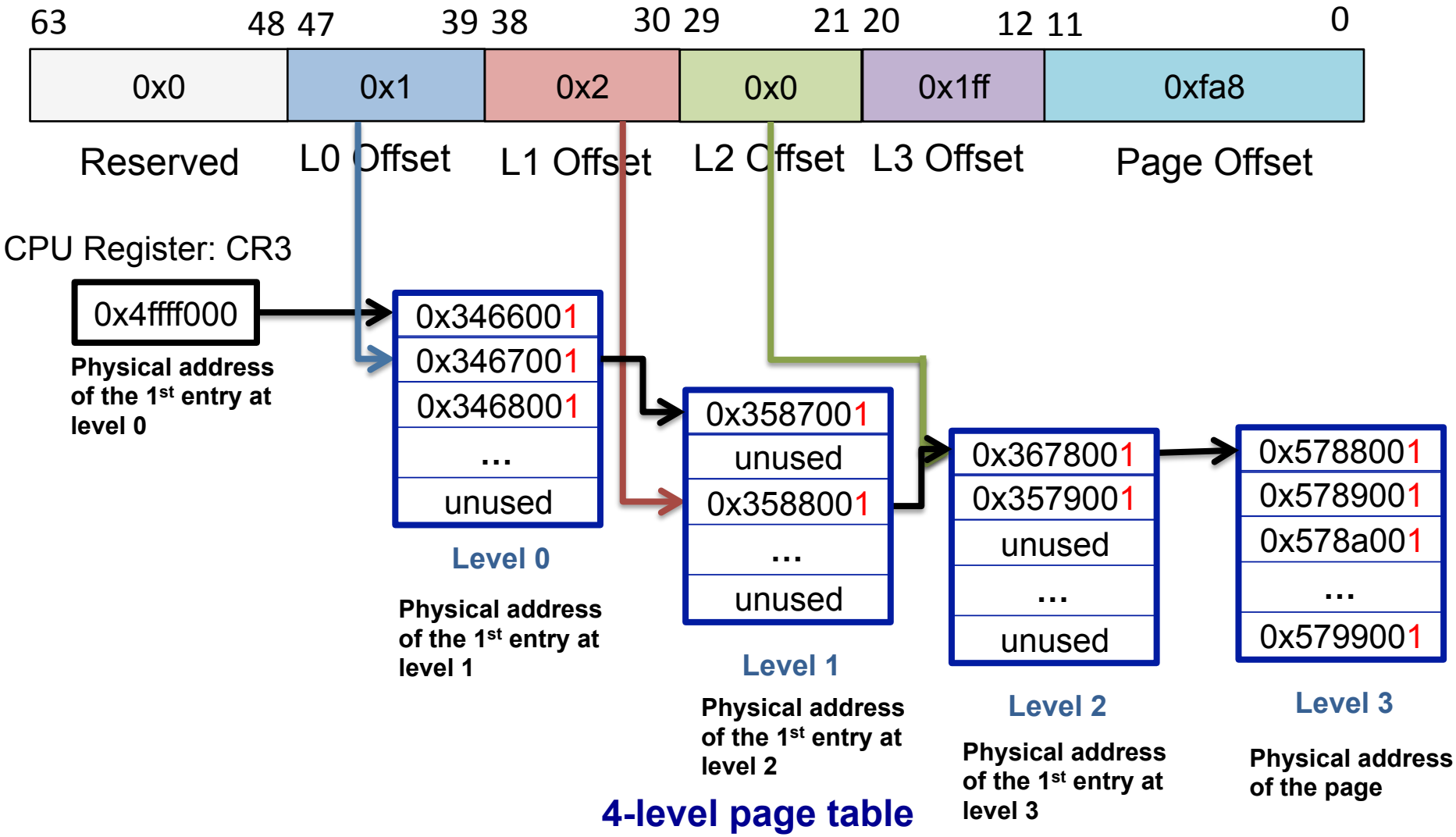
Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*

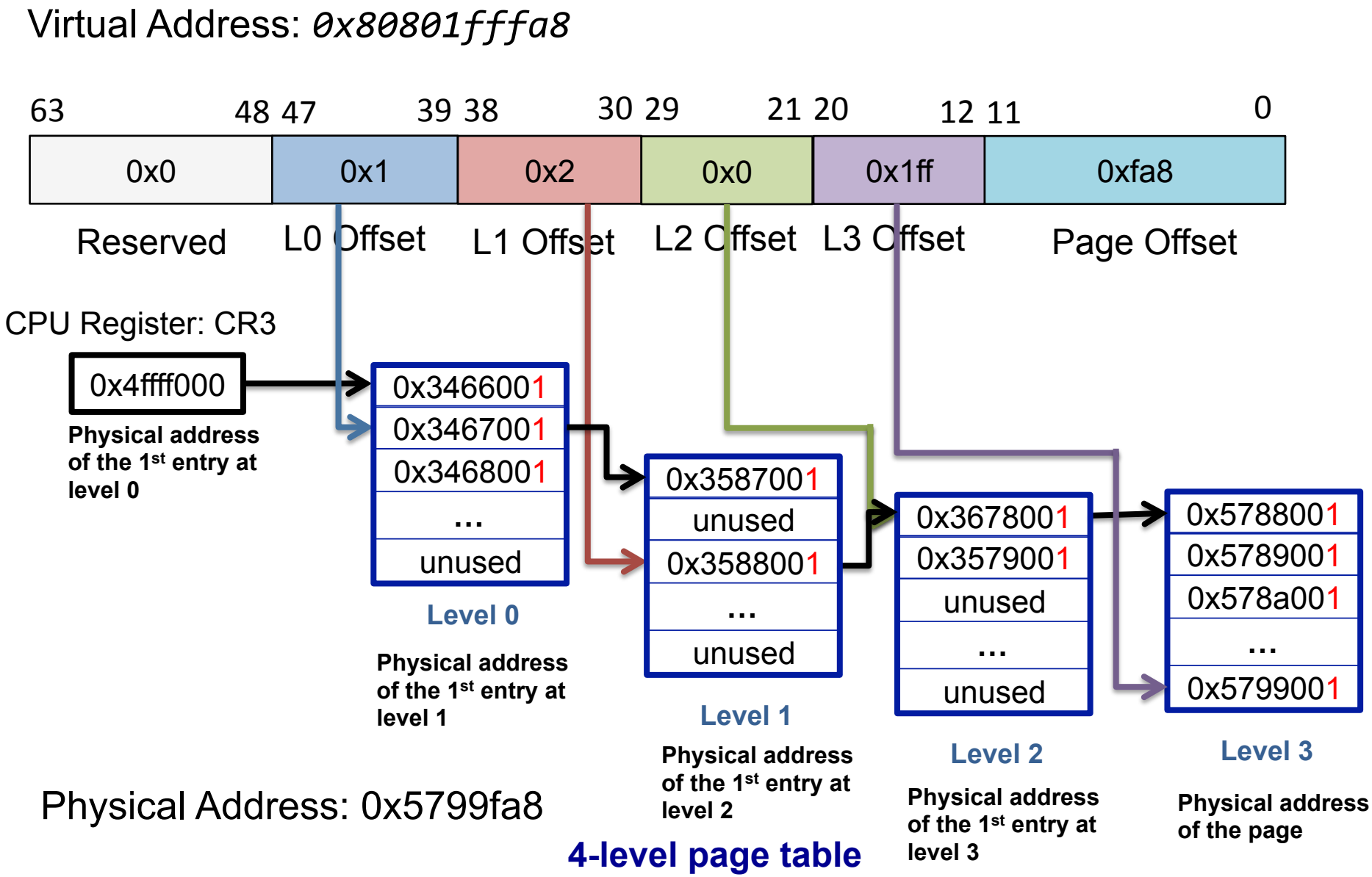


Multi-level page tables on X86_64

Virtual Address: *0x80801fffa8*



Multi-level page tables on X86_64

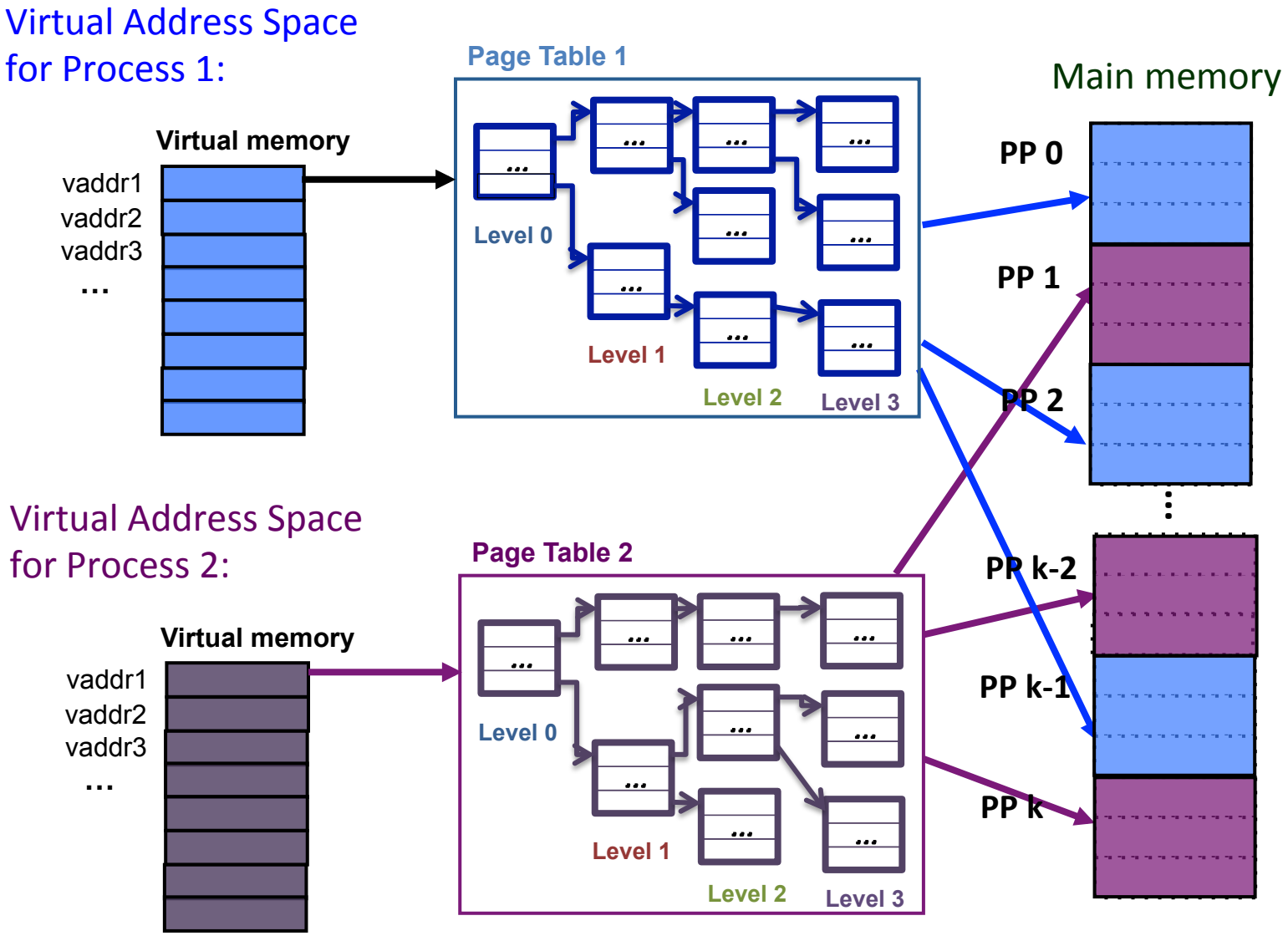


Review Virtual Address

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



Demand Paging

Memory Allocation (e.g., `p = sbrk(8192))`

User program to OS:

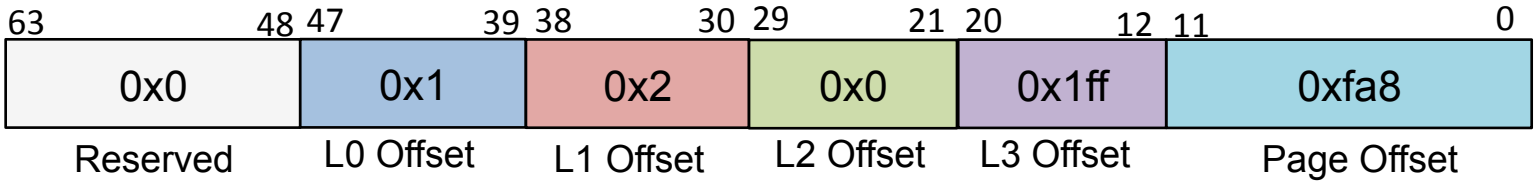
Declare a virtual address range from p to p + 8192 for use by the current process.

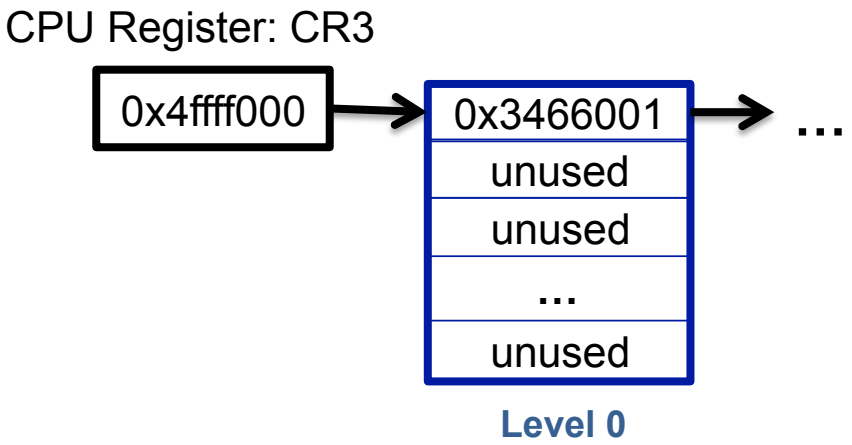
OS' actions:

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'
```



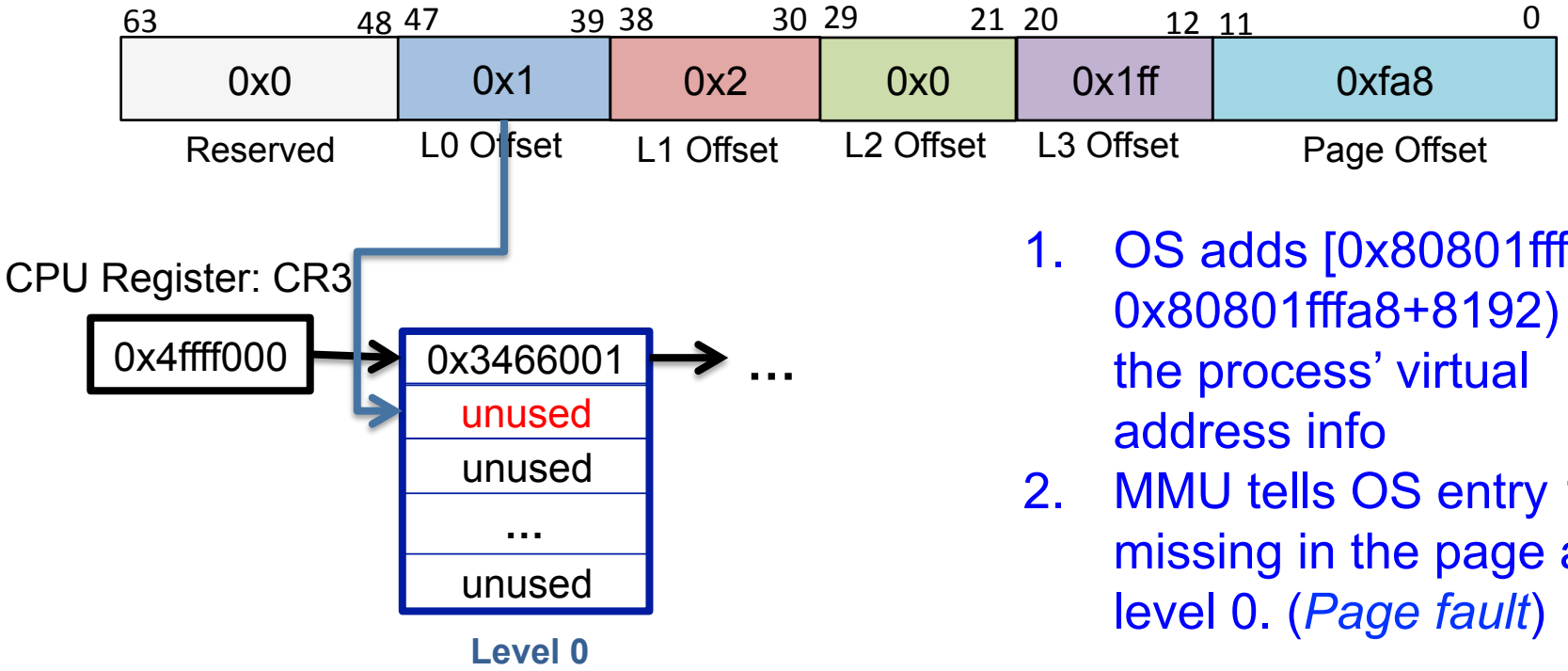


1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process' virtual address info

current process' page table

Demand Paging

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

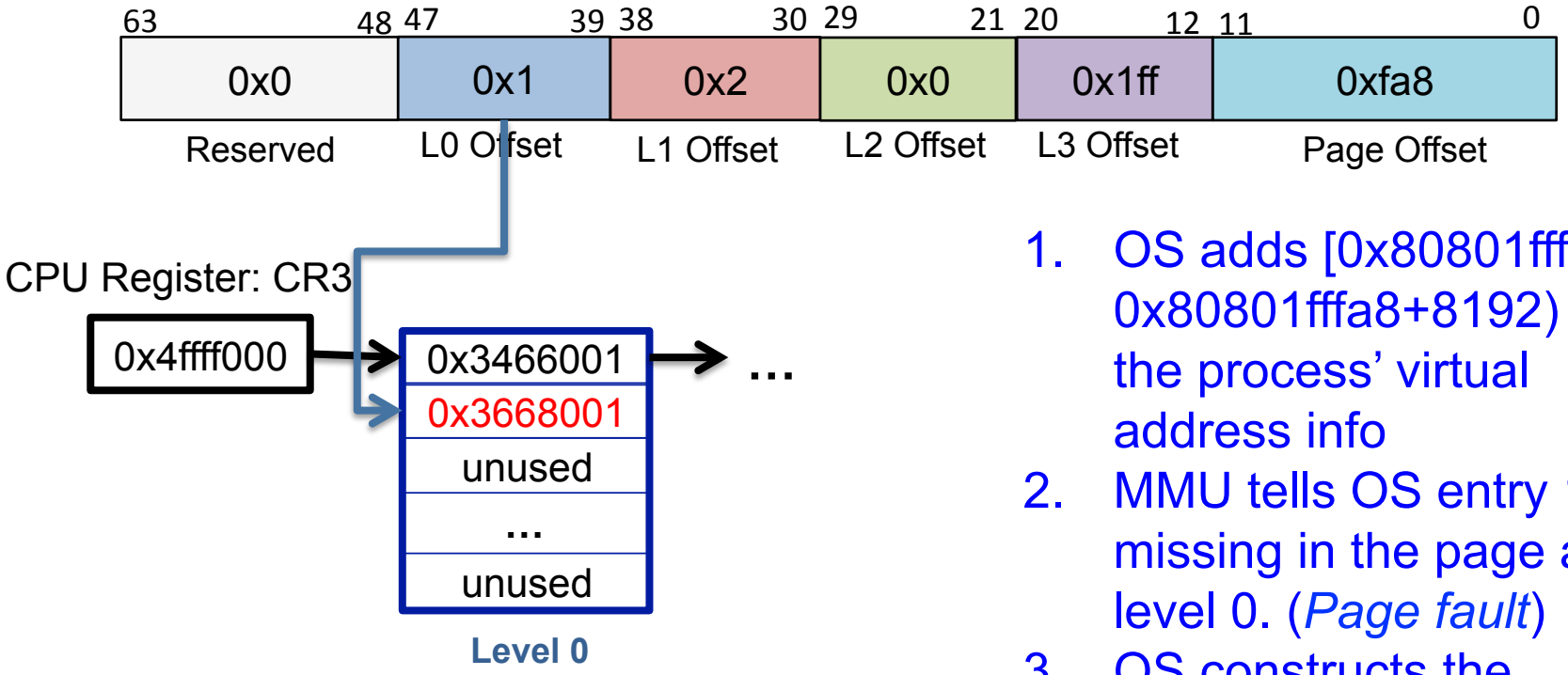


1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)

current process' page table

Demand Paging

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

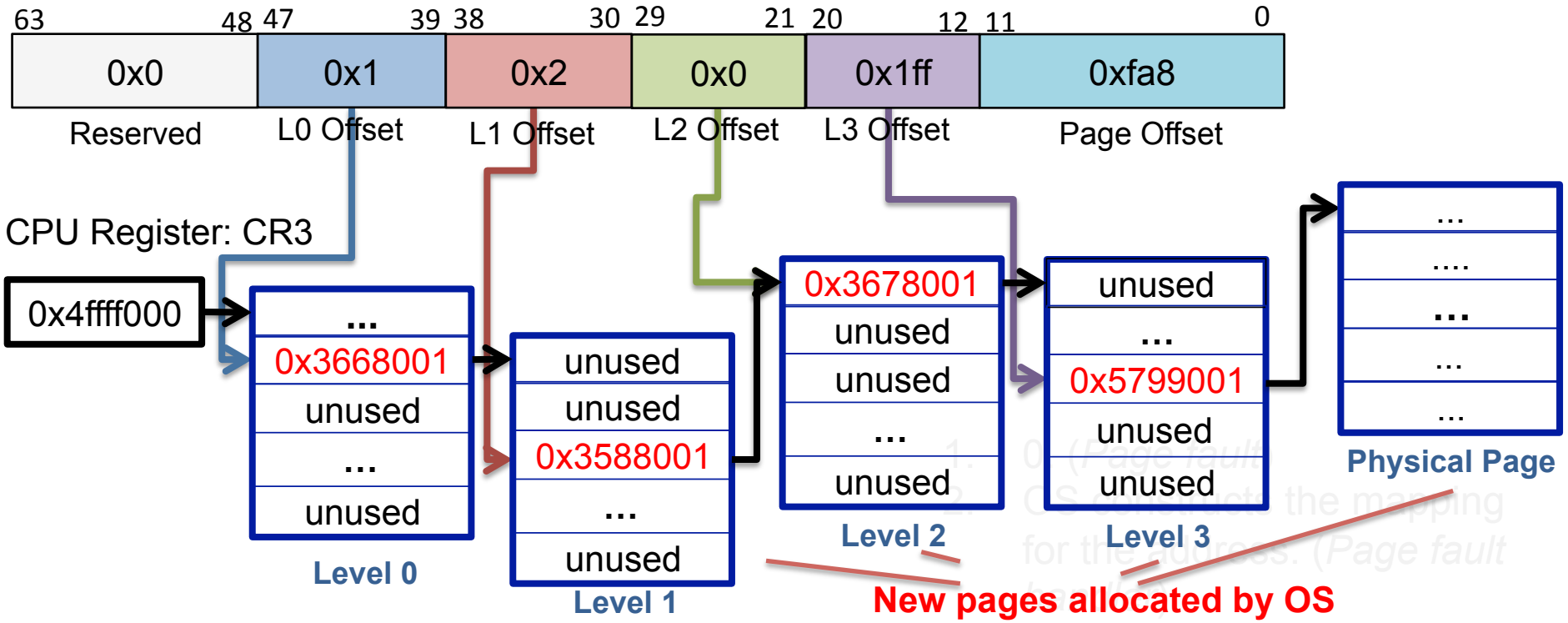


1. OS adds [0x80801fffa8, 0x80801fffa8+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (*Page fault*)
3. OS constructs the mapping for the address. (*Page fault handler*)

current process' page table

Demand Paging

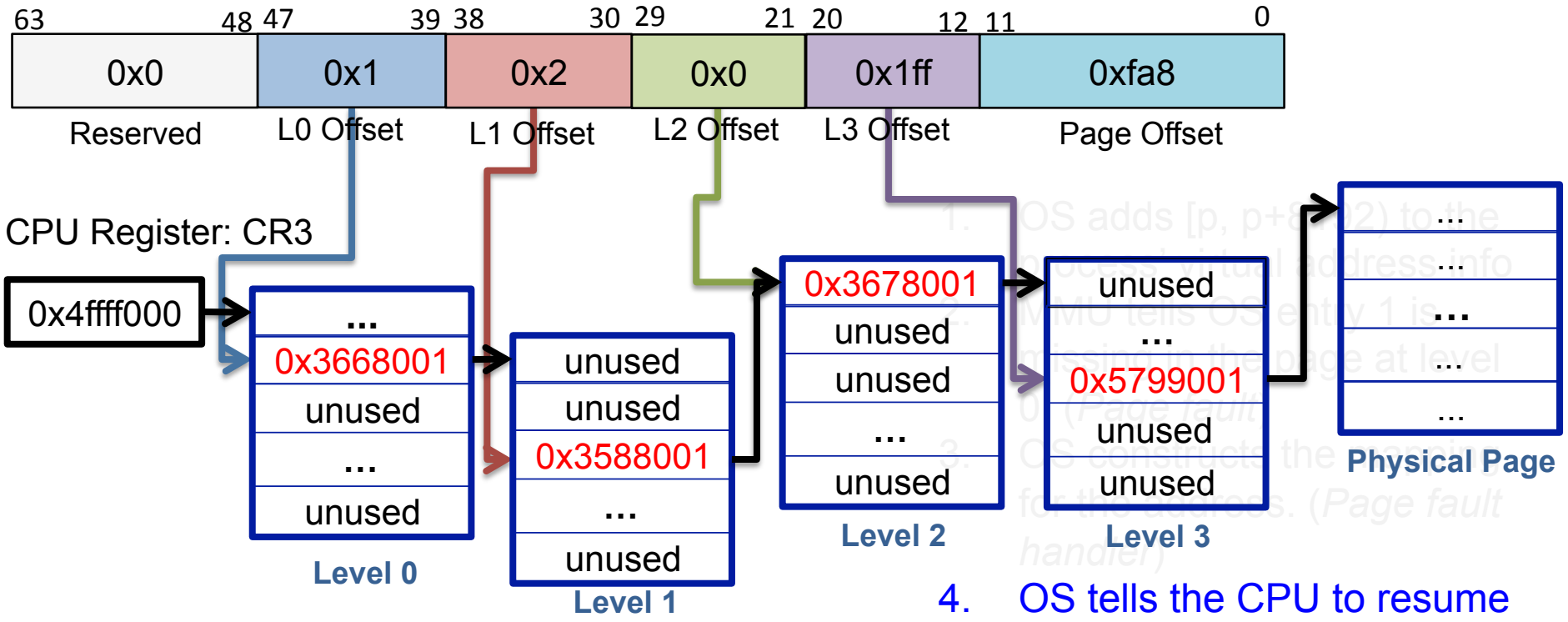
```
char *p = (char *)sbrk(8192); // p is 0x80801fffa8
p[0] = 'c'
p[4096] = 's'
```



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'
```

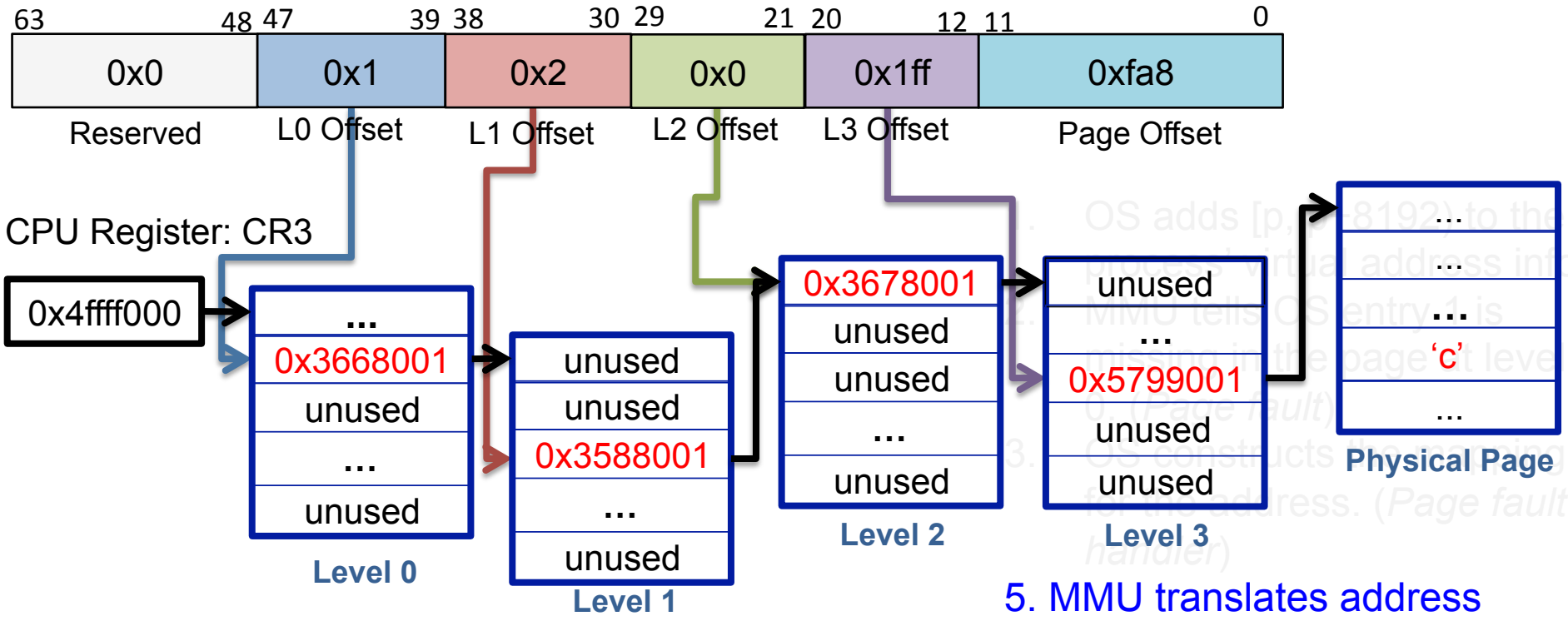


4. OS tells the CPU to resume execution

current process' page table

Demand Paging

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

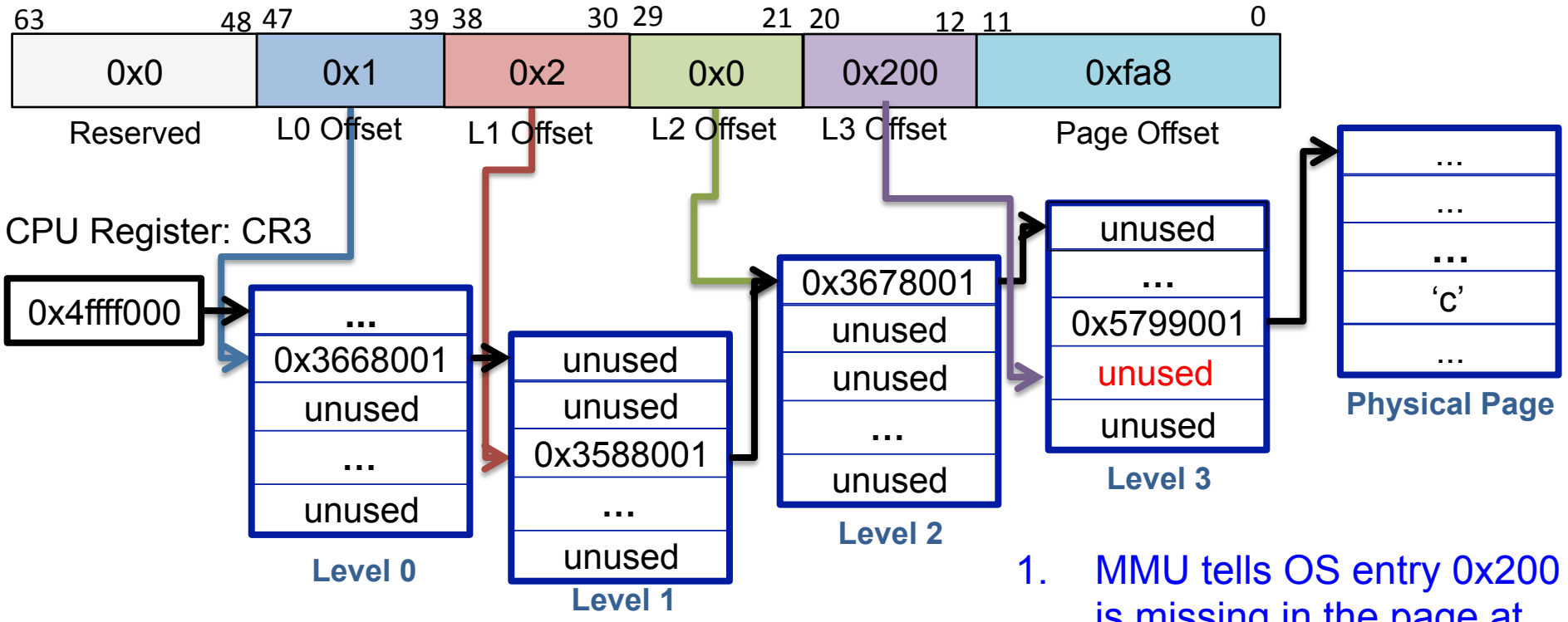


current process' page table

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

Demand Paging

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

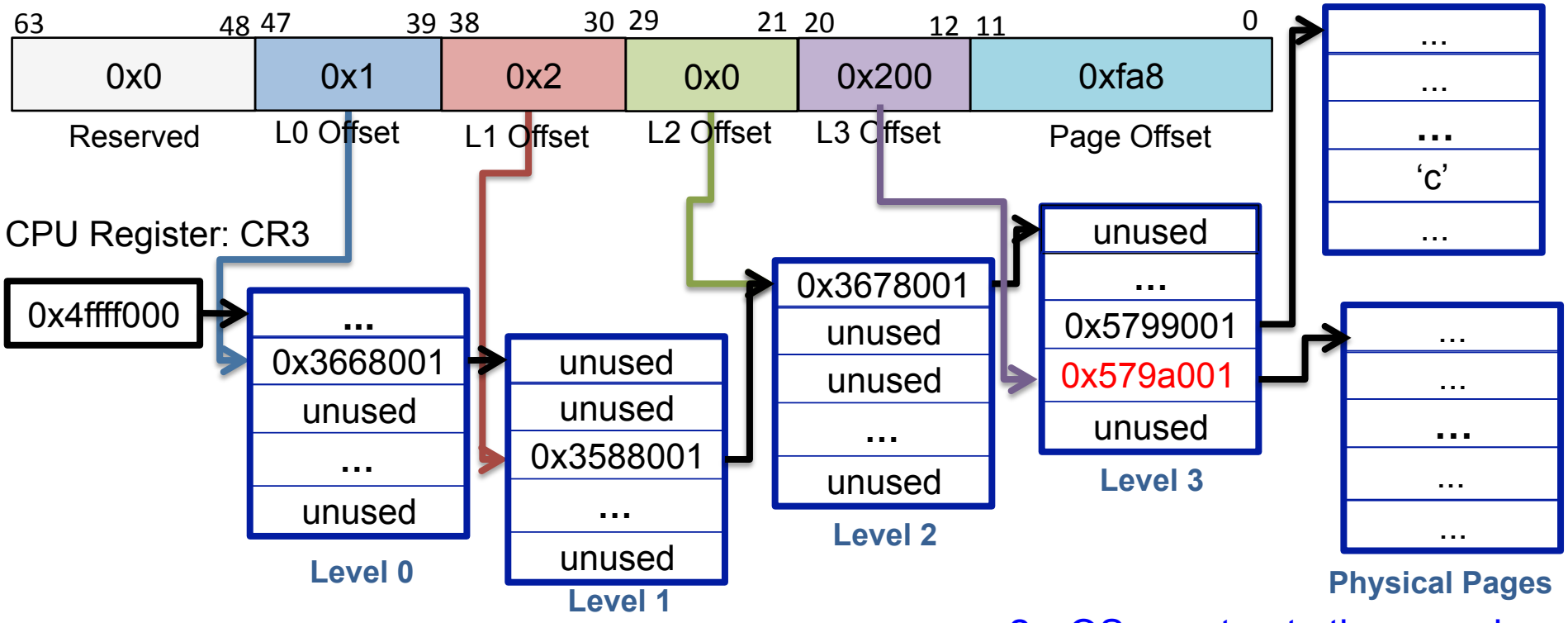


1. MMU tells OS entry 0x200 is missing in the page at level 3. (*Page fault*)

current process' page table

Demand Paging

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

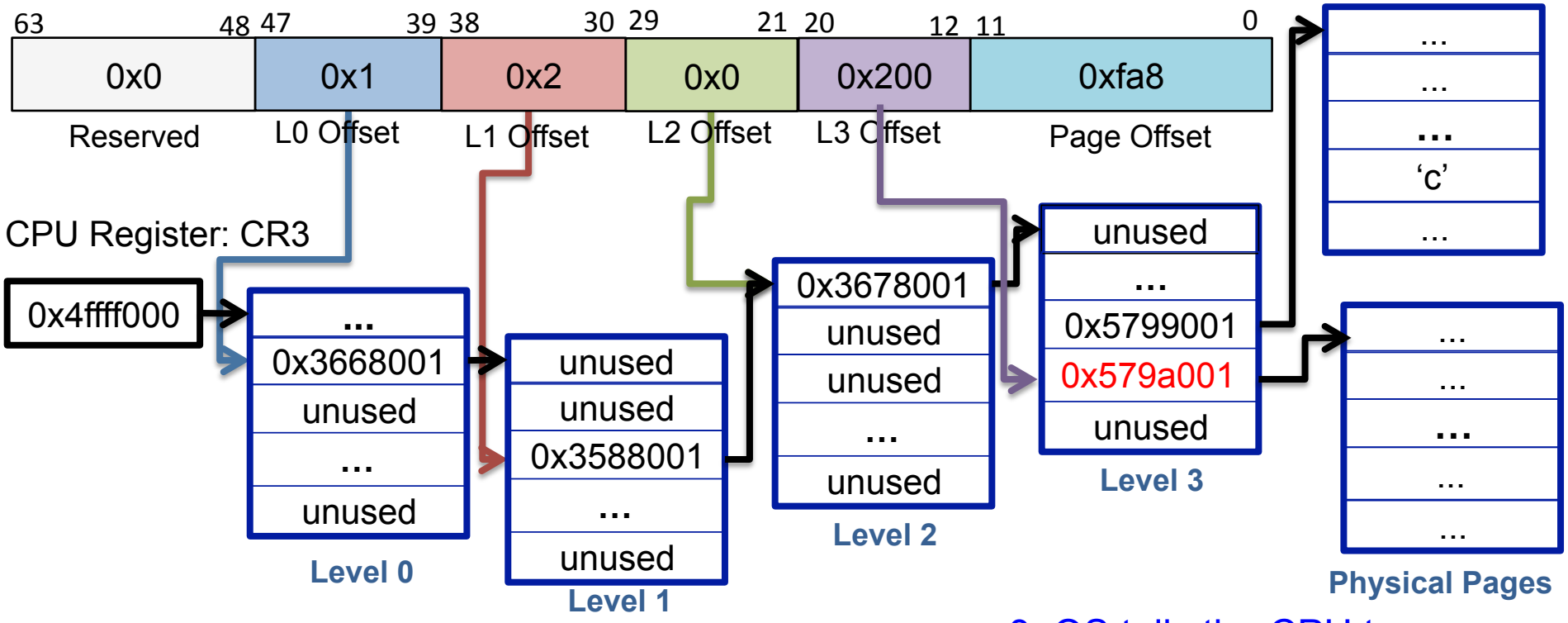


current process' page table

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
```

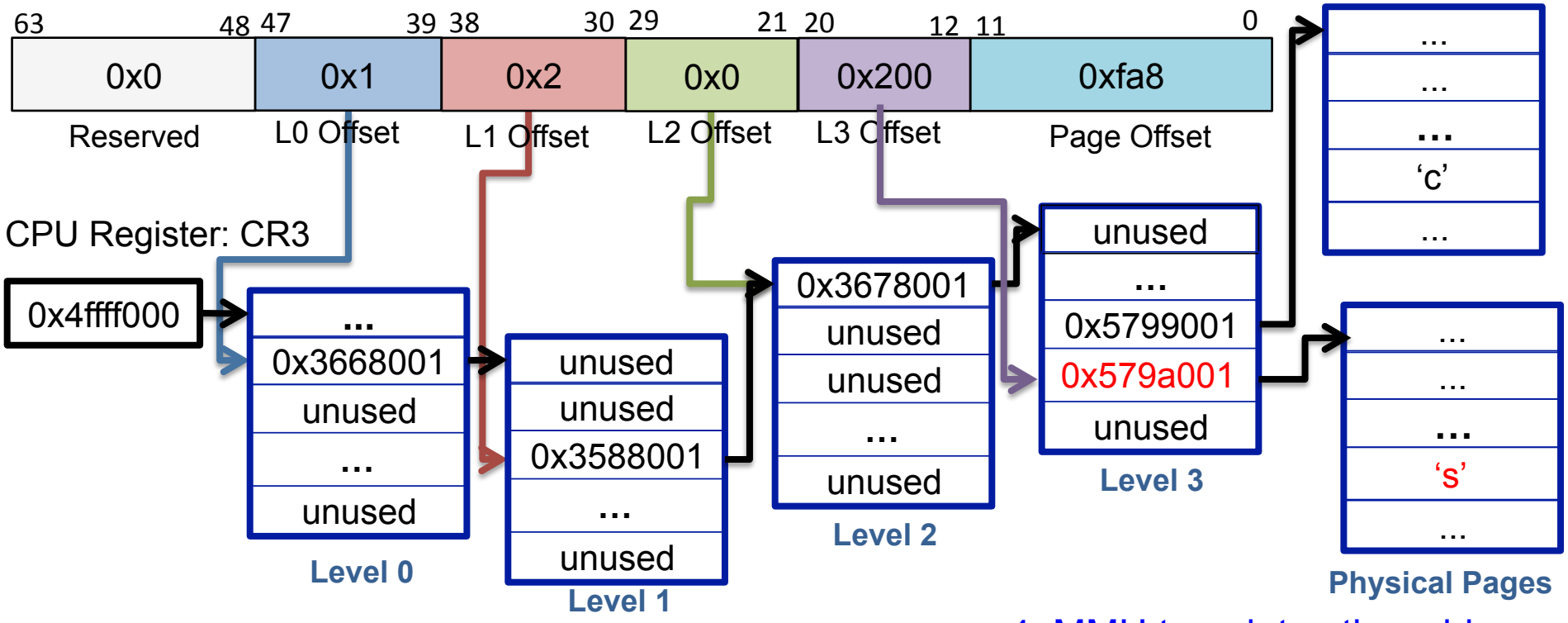


current process' page table

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

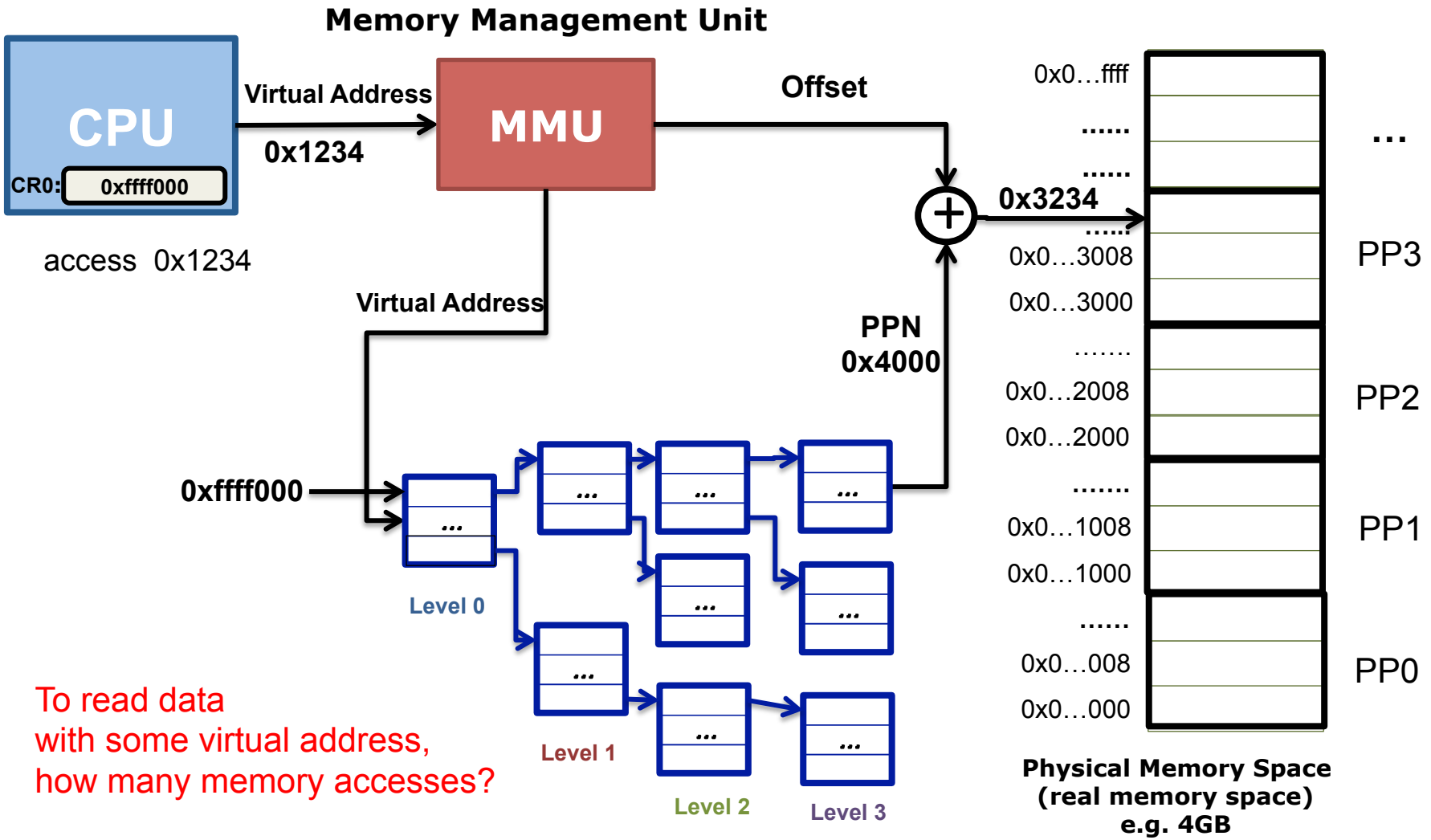
Memory access latency

- 100 ns
- 160 ~ 200 CPU cycles

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

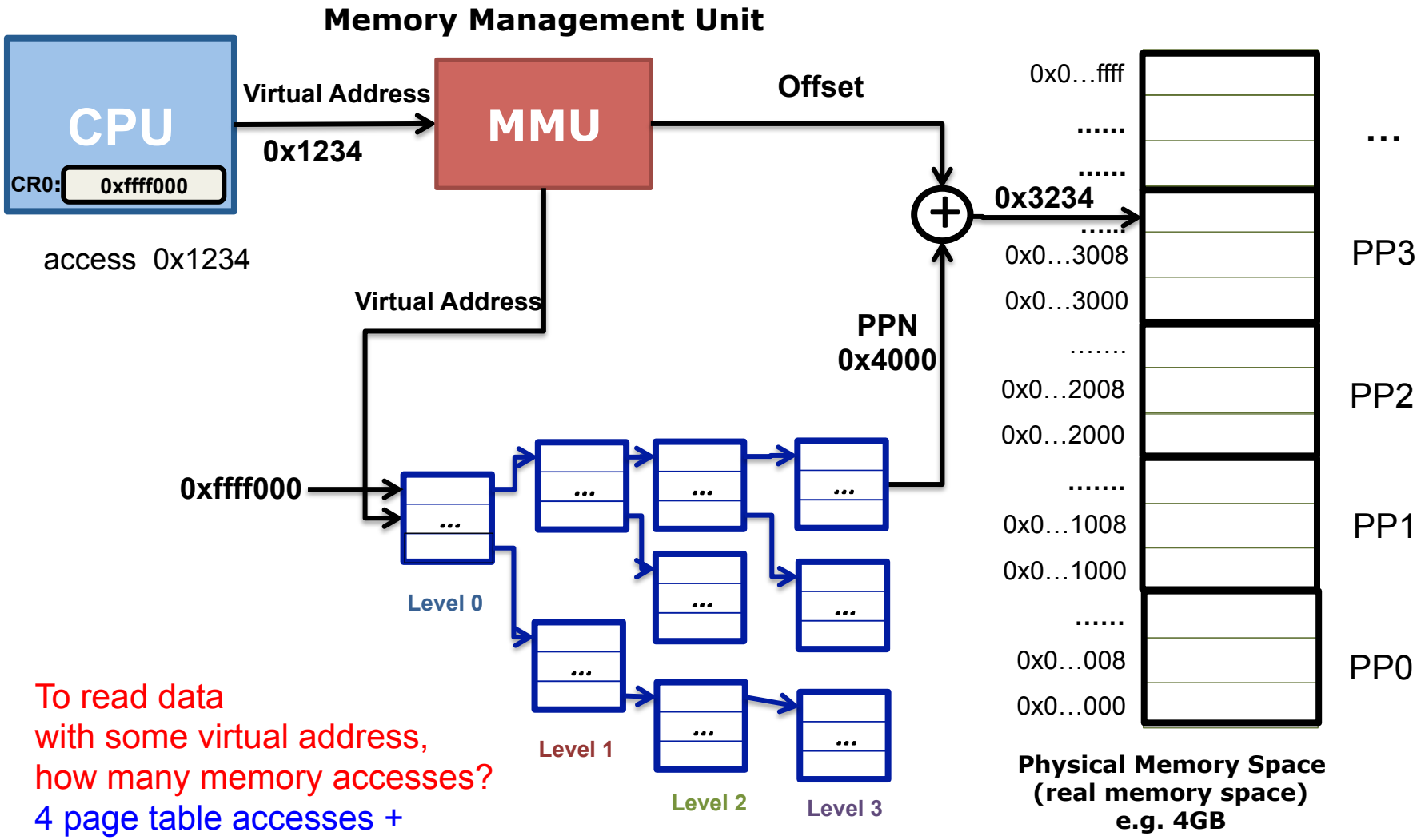
- Instructions per CPU cycle >= 1

Address translation is potentially very costly



To read data
with some virtual address,
how many memory accesses?

Address translation is potentially very costly

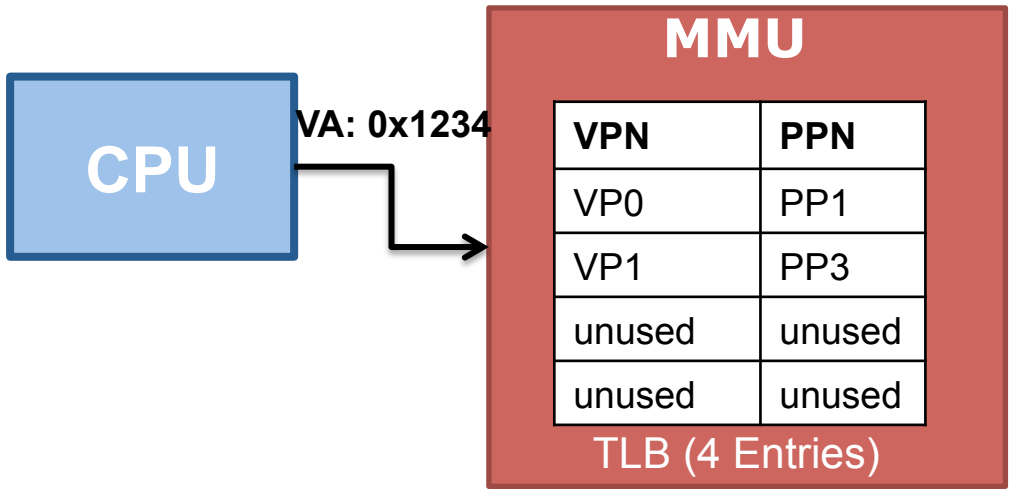


To read data
with some virtual address,
how many memory accesses?
4 page table accesses +
one data access = **5** total memory accesses.

Speedup Address Translation

Translation lookaside buffer (TLB)

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

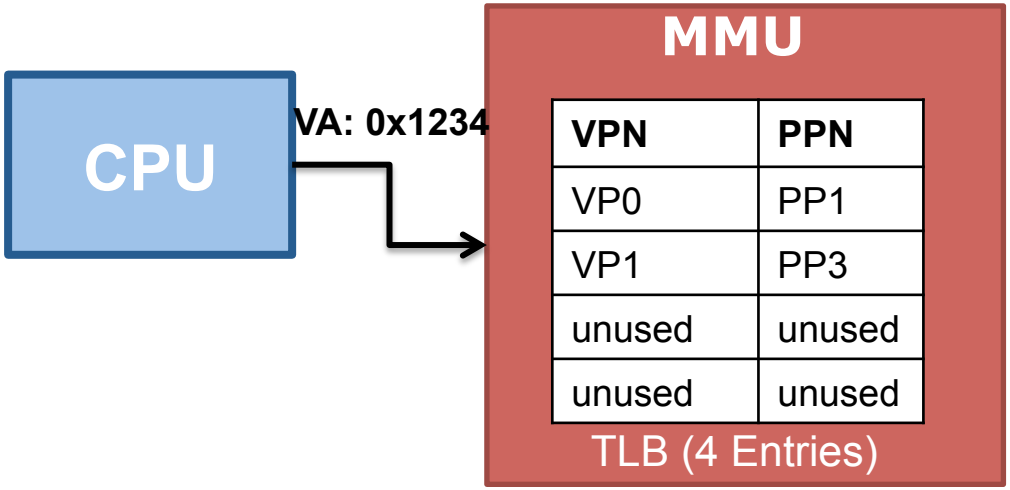


1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = *VPN % 4*
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

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

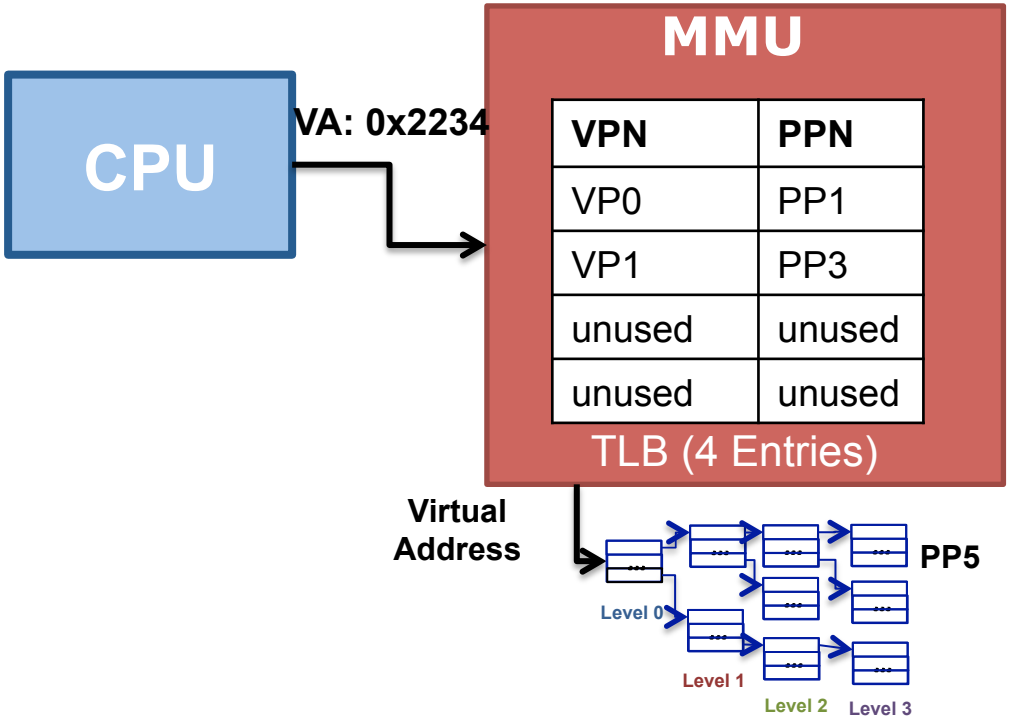
Example:

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

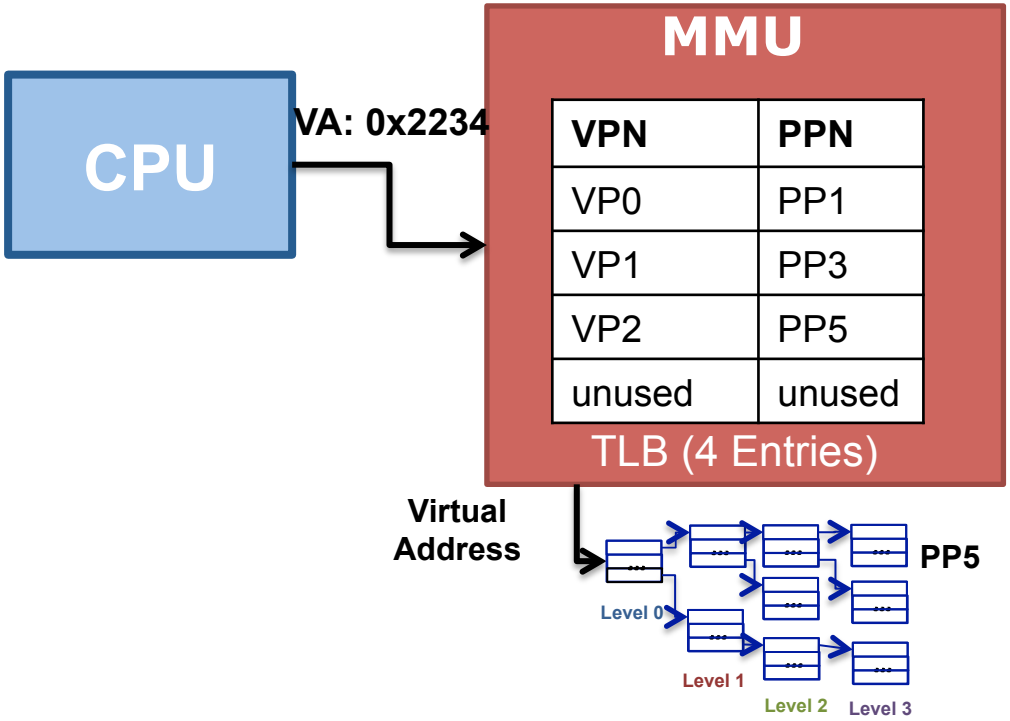
Example:

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

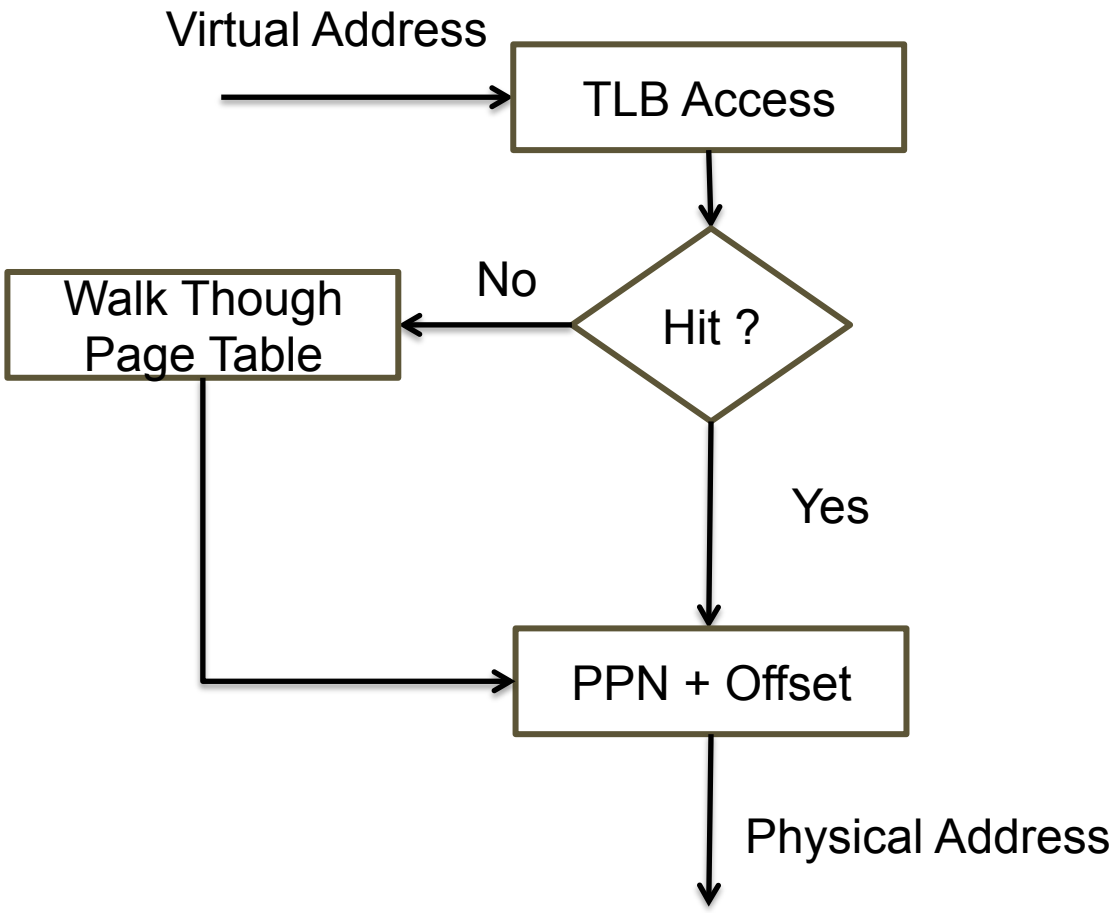
Memory access

- Hundreds of CPU cycles

TLB access

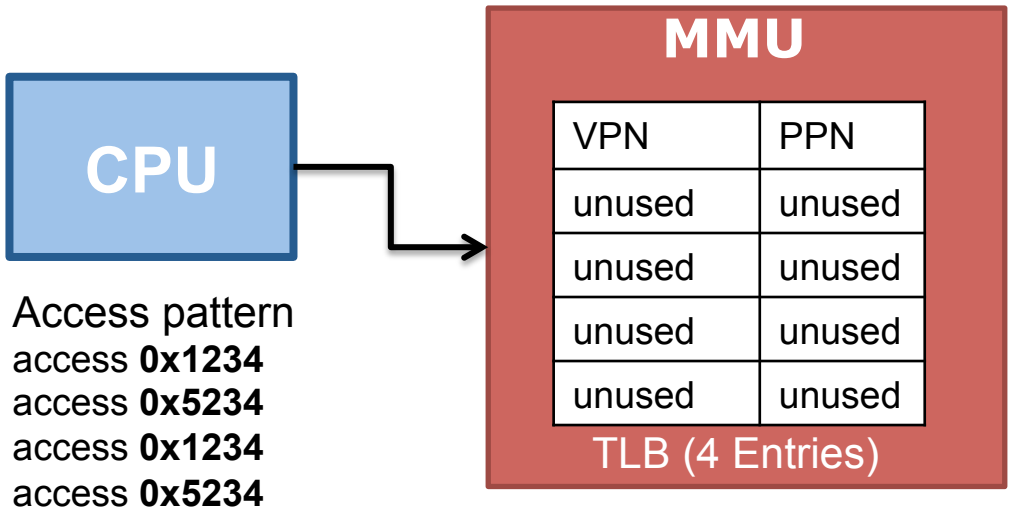
- Only a couple of CPU cycles

Summary


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

More on TLBs

Speedup Address Translation



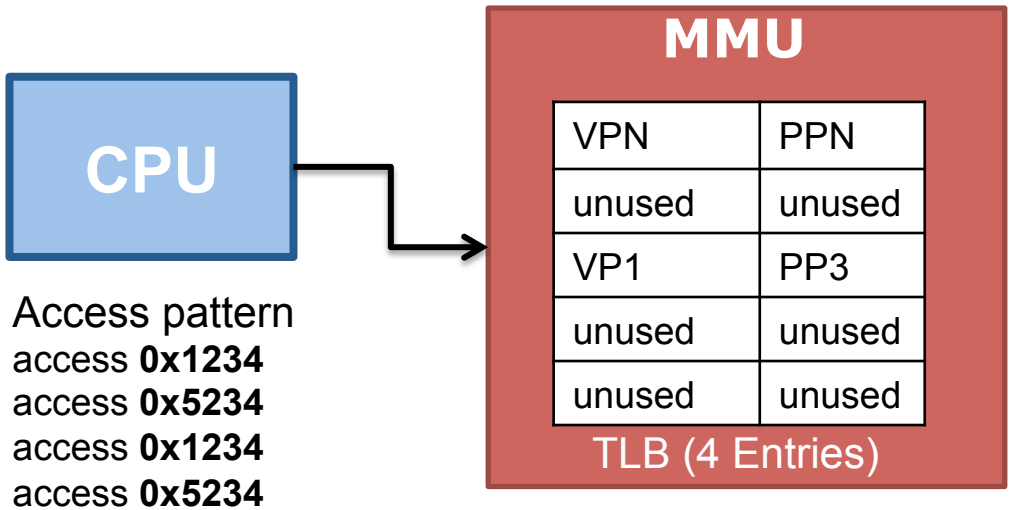
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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



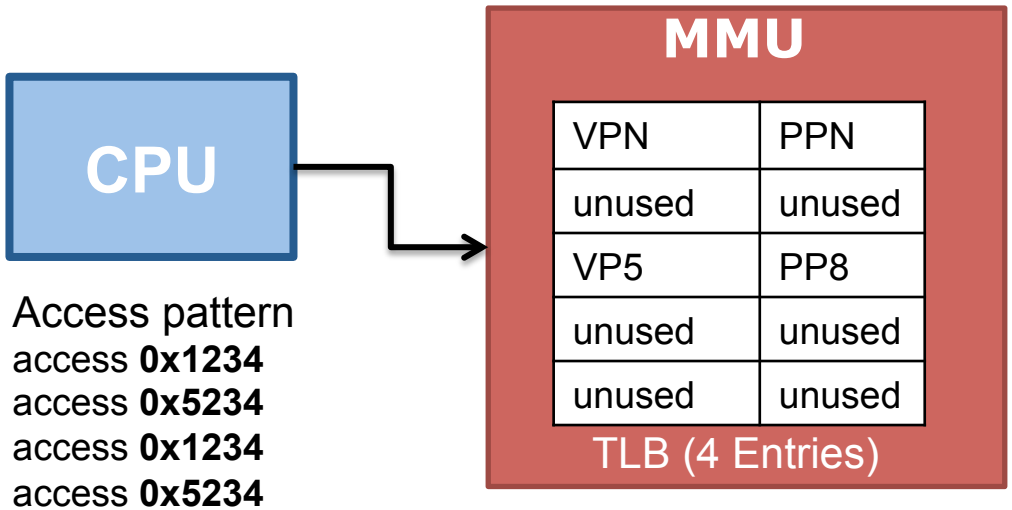
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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



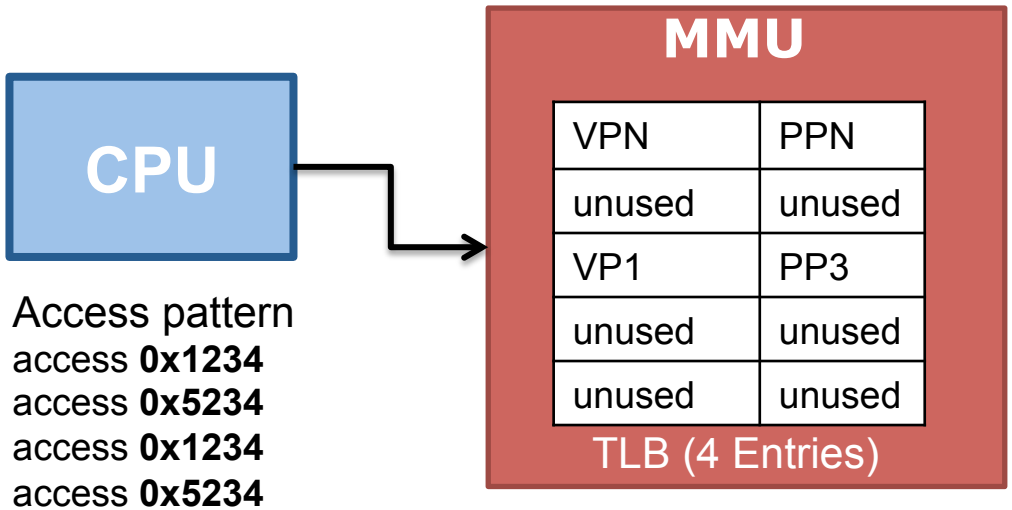
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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



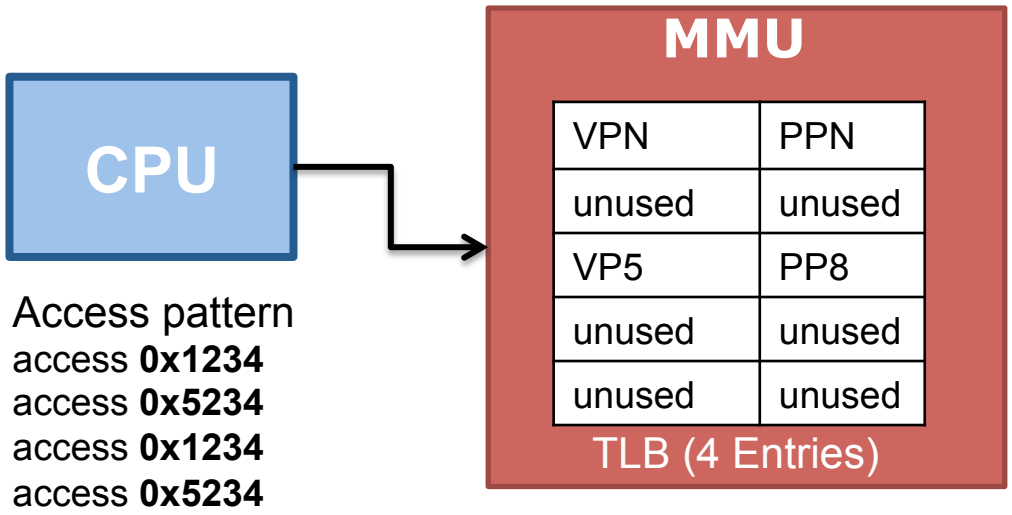
Access pattern
access **0x1234**
access **0x5234**
access **0x1234**
access **0x5234**

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**

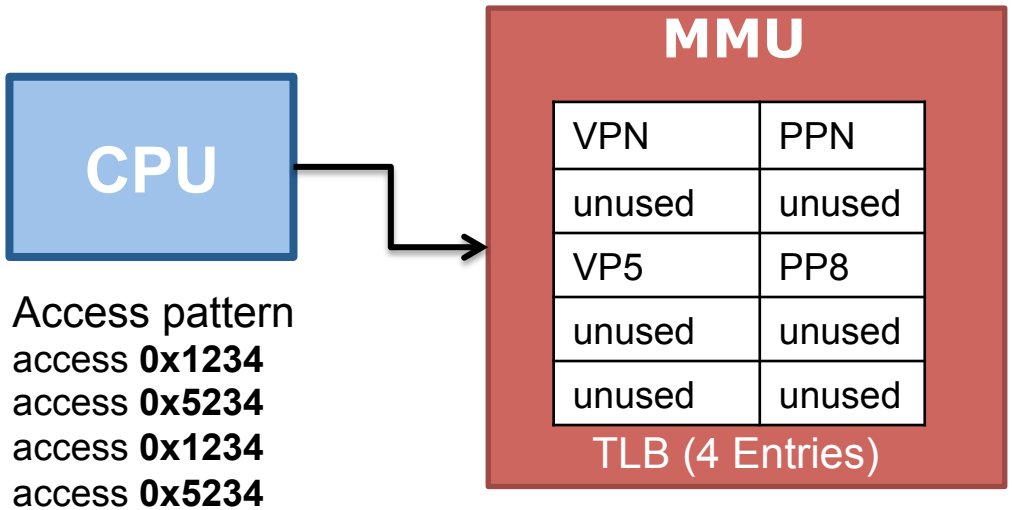
1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = *VPN % 4*
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

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**

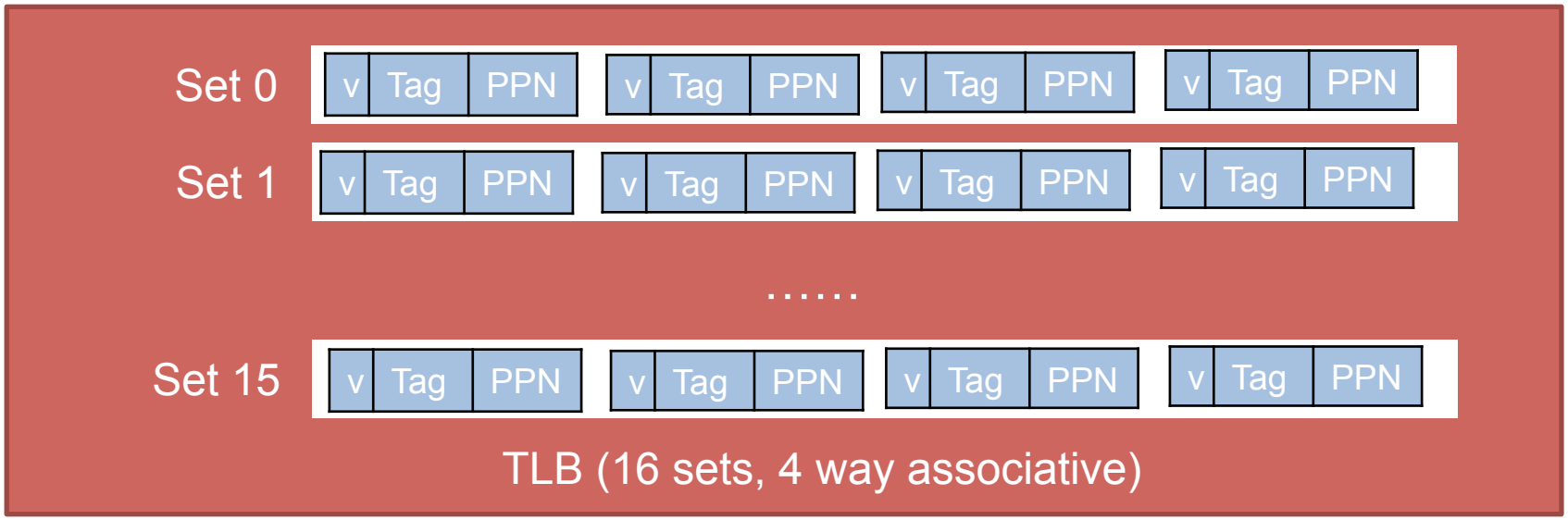
1. Calculate VPN
   a. VPN = VA >> 12
2. Check TLB
   a. Index = *VPN % 4*
   b. Check if TLB[Index].VPN == VPN
   c. On TLB hit,
      PA = TLB[Index].PPN + Offset
   a. On TLB miss
      Go though page table to get PPN
      Buffer the result in TLB

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

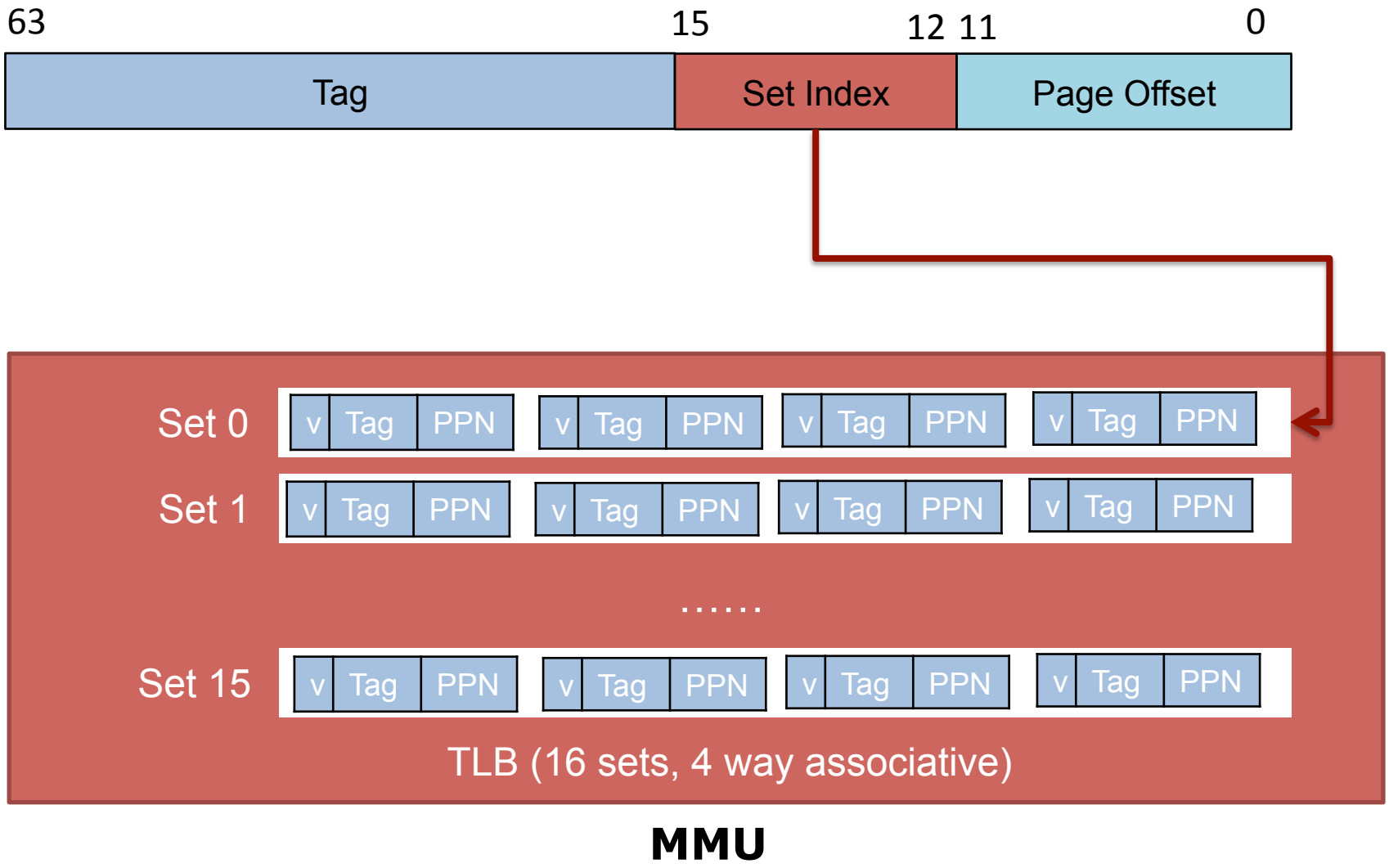
TLB eviction due to conflict! → Multi-set associative TLB

Multi-set associative TLB



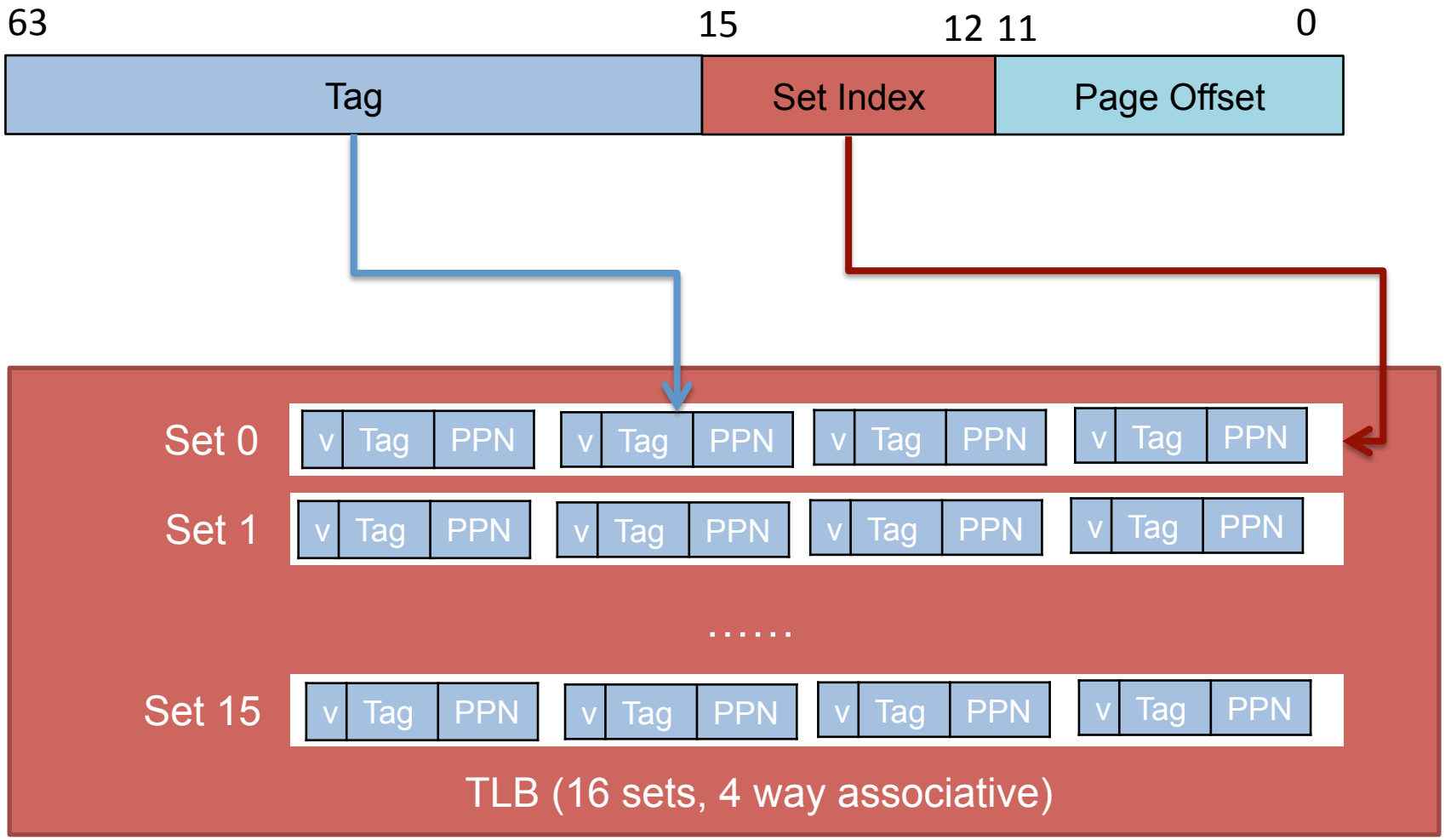
MMU

Multi-set associative TLB



MMU

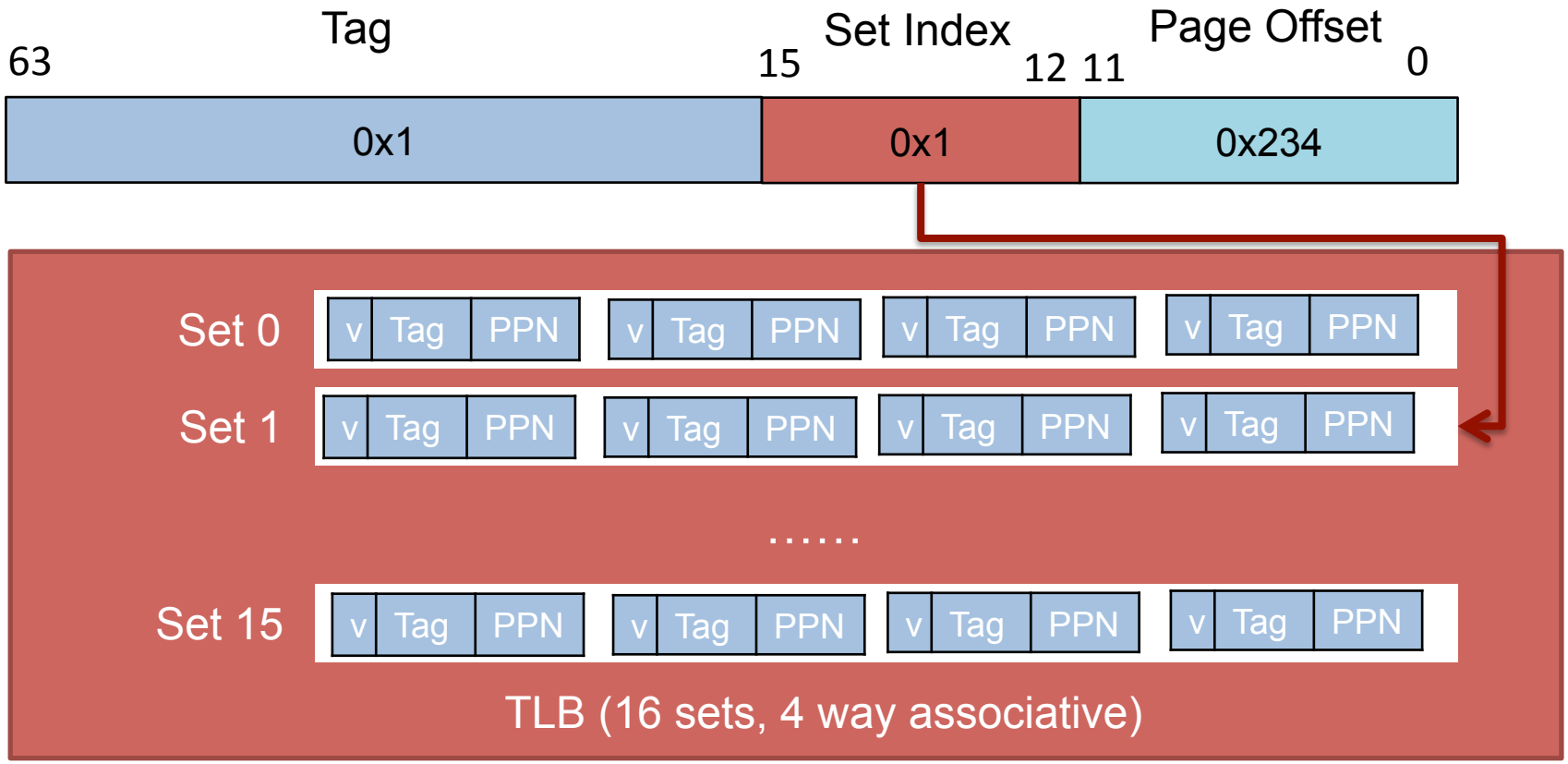
Multi-set associative TLB



MMU

Example

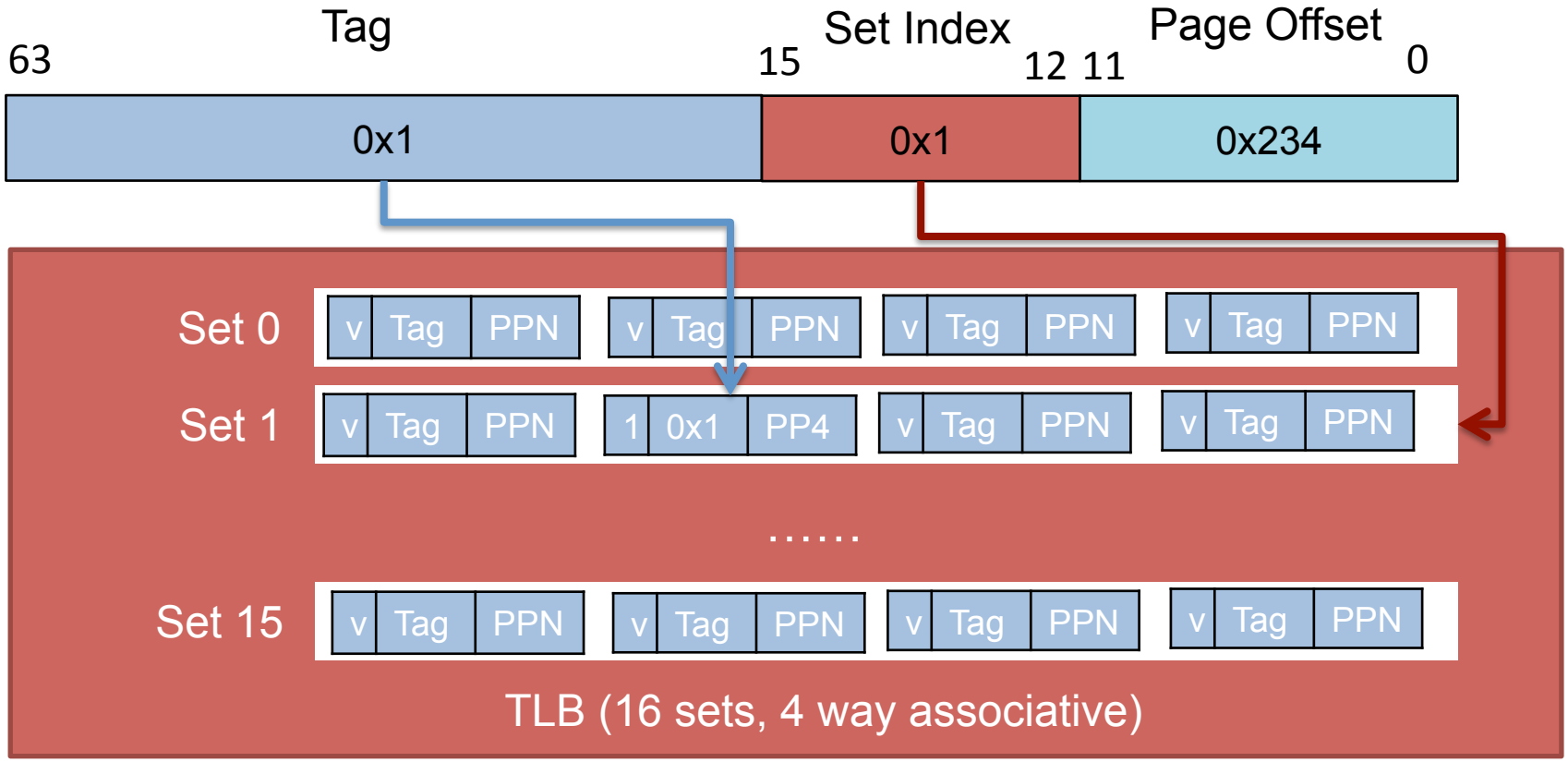
→ access **0x11234, TLB Miss**
access **0x21234**
access **0x11234**
access **0x21234**



MMU

Example

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



MMU

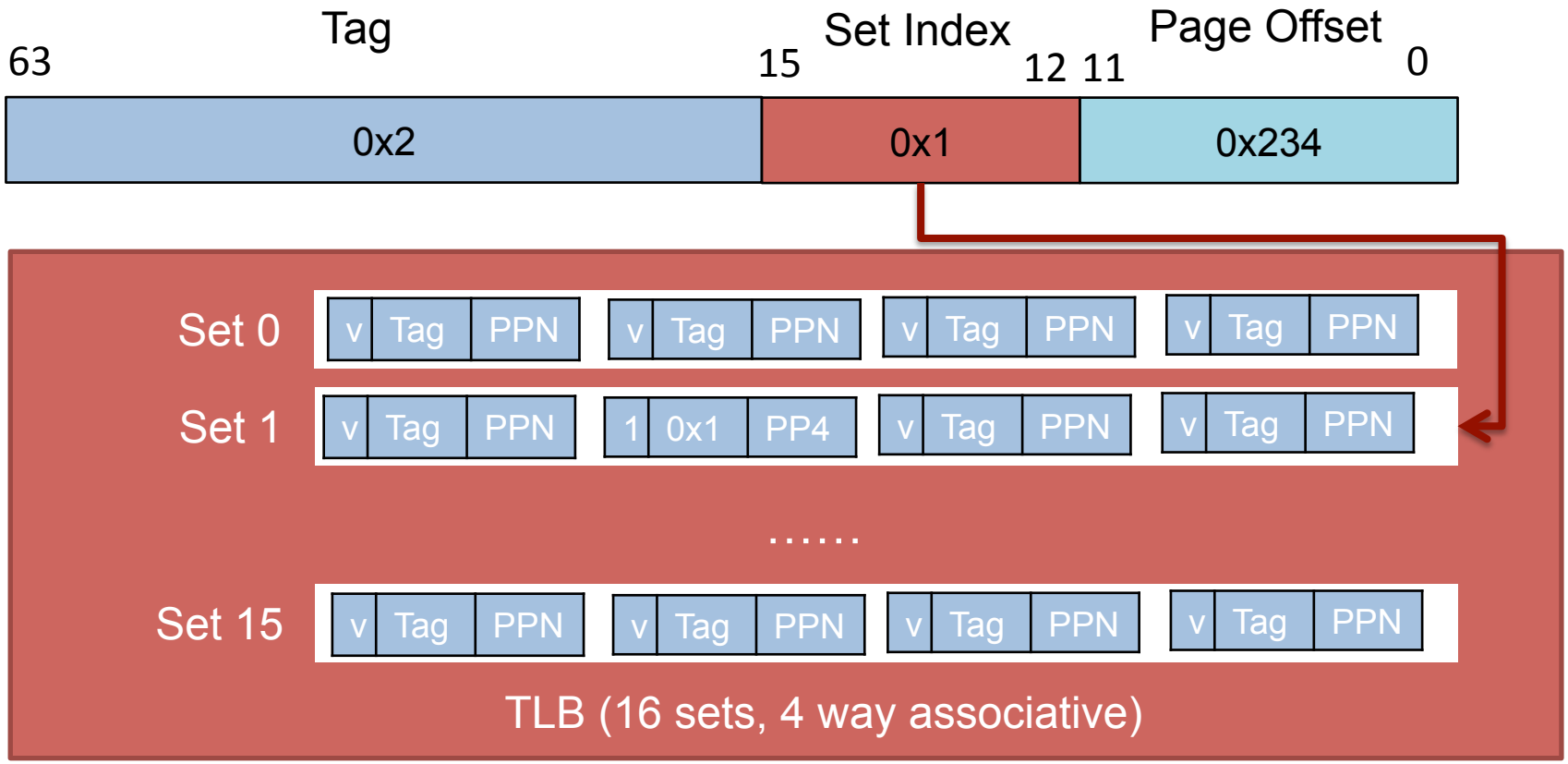
Example

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

→ access **0x21234, TLB Miss,**

access **0x11234**

access **0x21234**



MMU

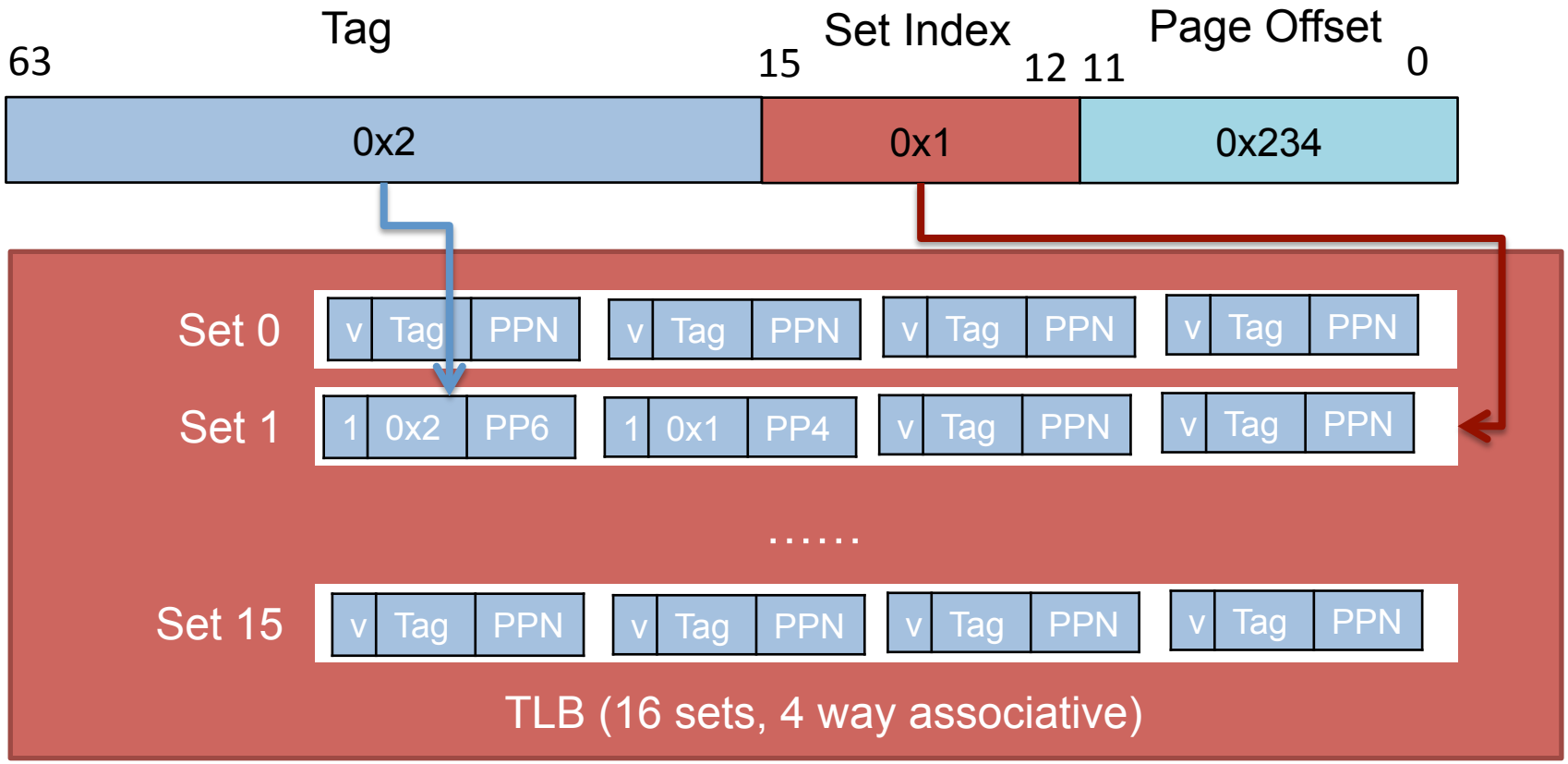
Example

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

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

access **0x11234**

access **0x21234**



MMU

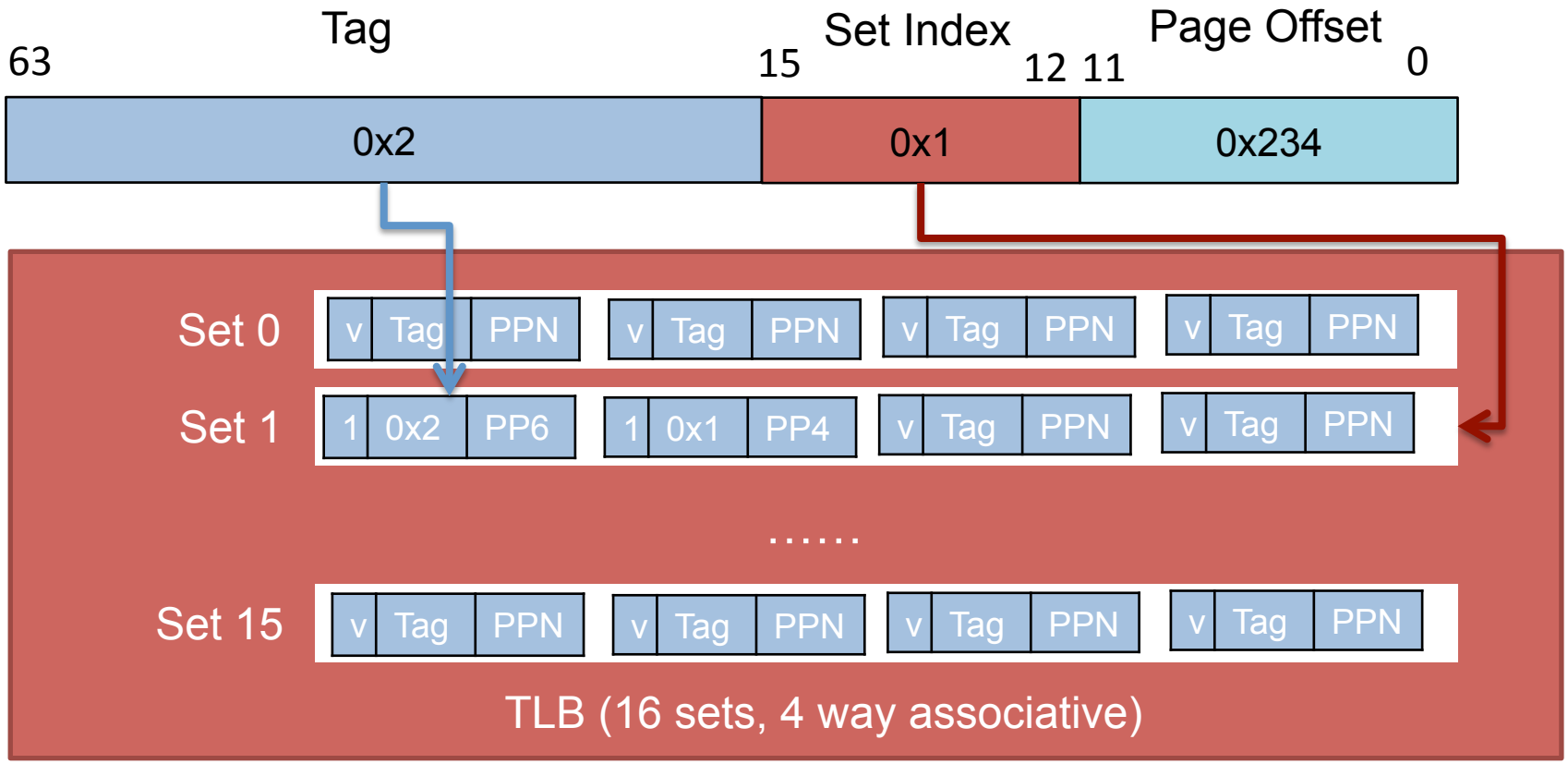
Example

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

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

access **0x11234**

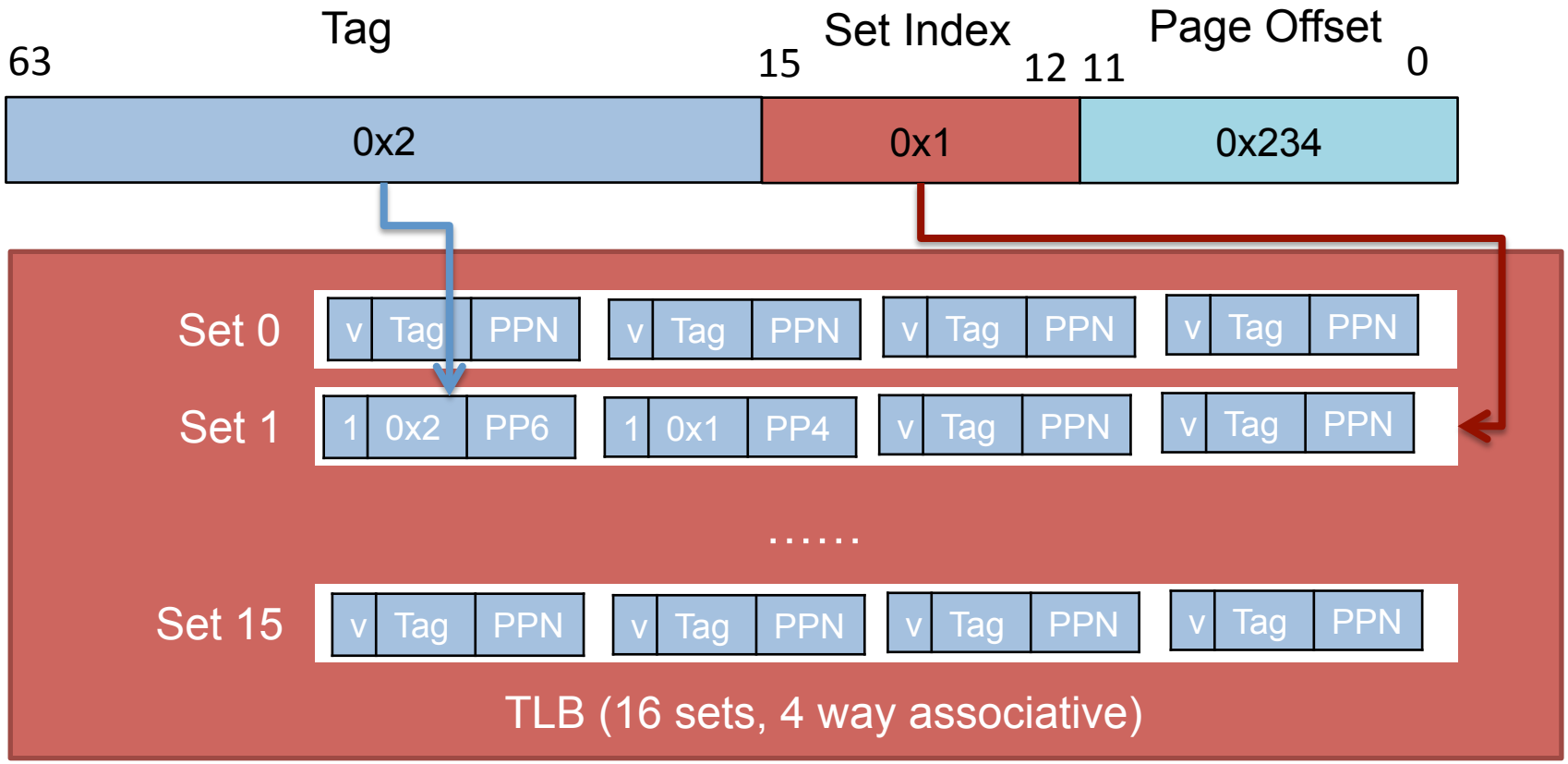
access **0x21234**



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