

# Virtual Memory

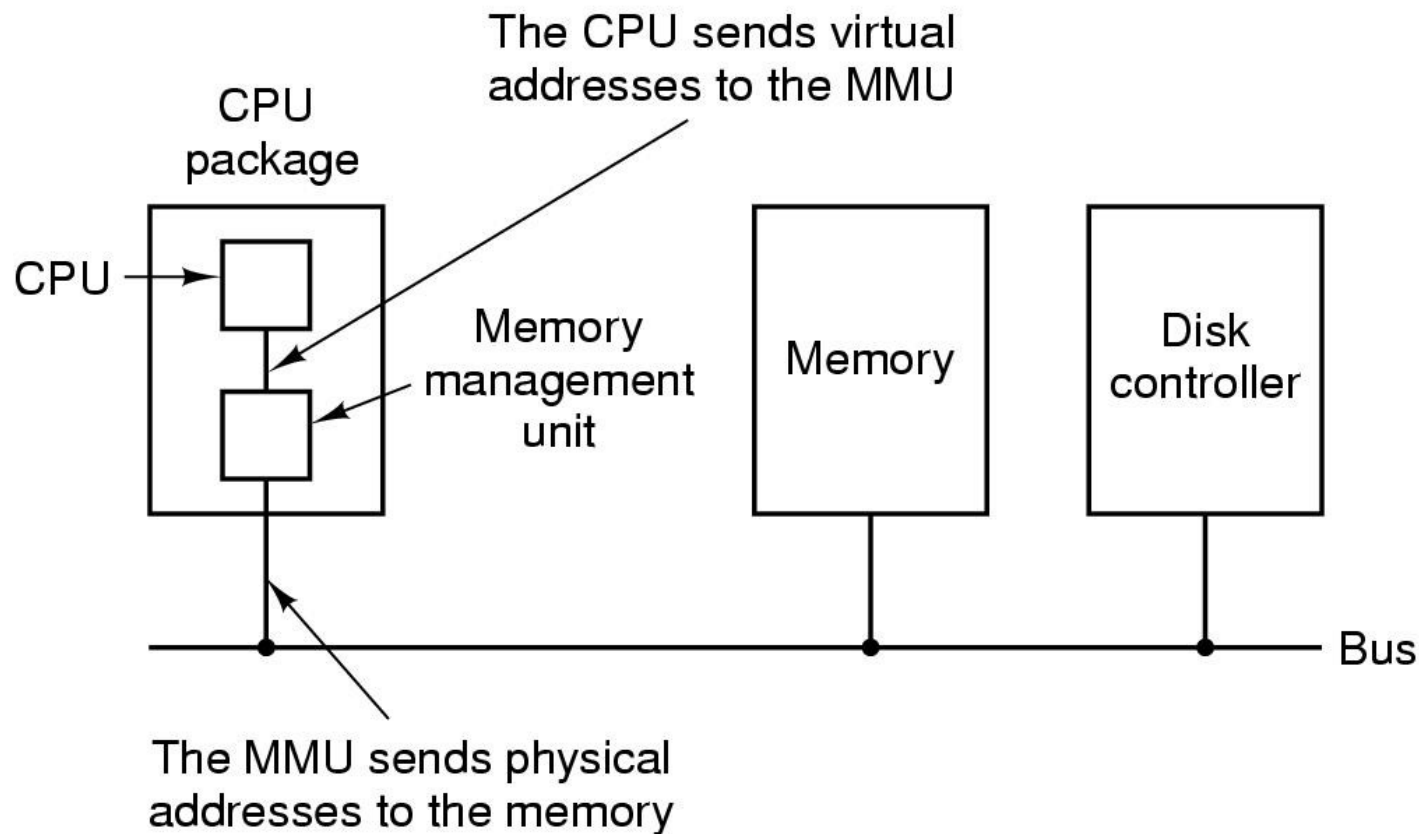


# Learning Outcomes

- An understanding of page-based virtual memory in depth.
  - Including the R3000's support for virtual memory.



# Memory Management Unit (or TLB)



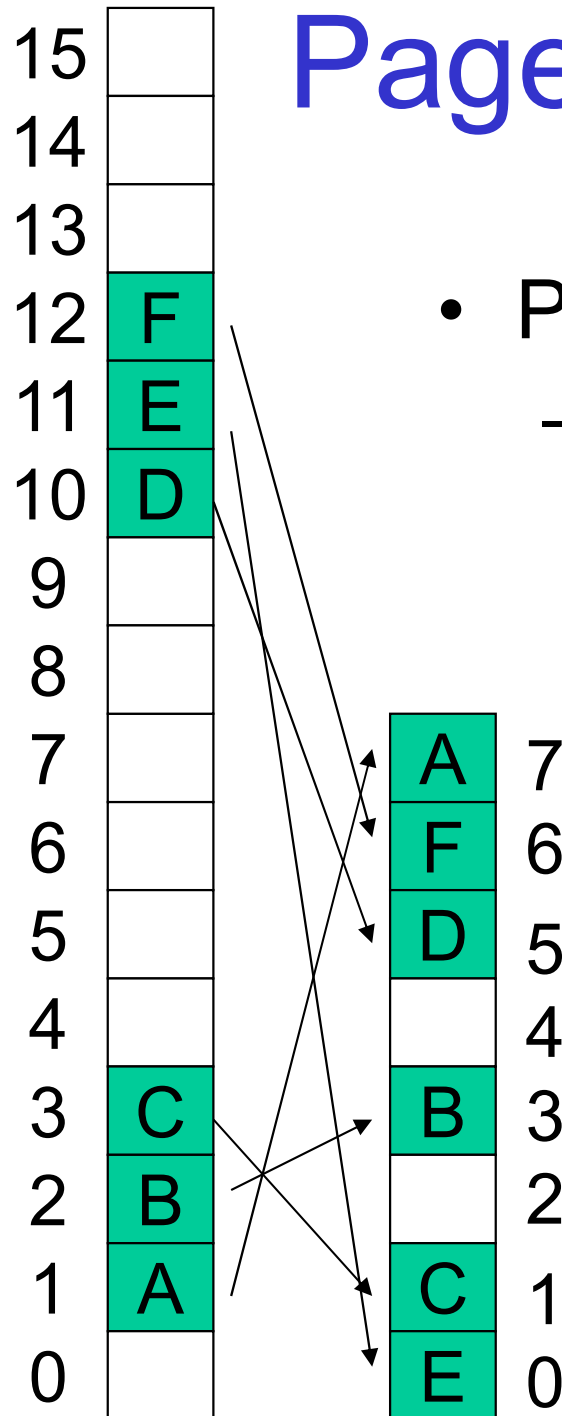
The position and function of the MMU

## Virtual Address Space

- Virtual Memory
  - Divided into equal-sized *pages*
  - A *mapping* is a translation between
    - A page and a frame
    - A page and *invalid*
  - Mappings defined at runtime
    - They can change
  - Address space can have holes
  - Process does not have to be contiguous in physical memory

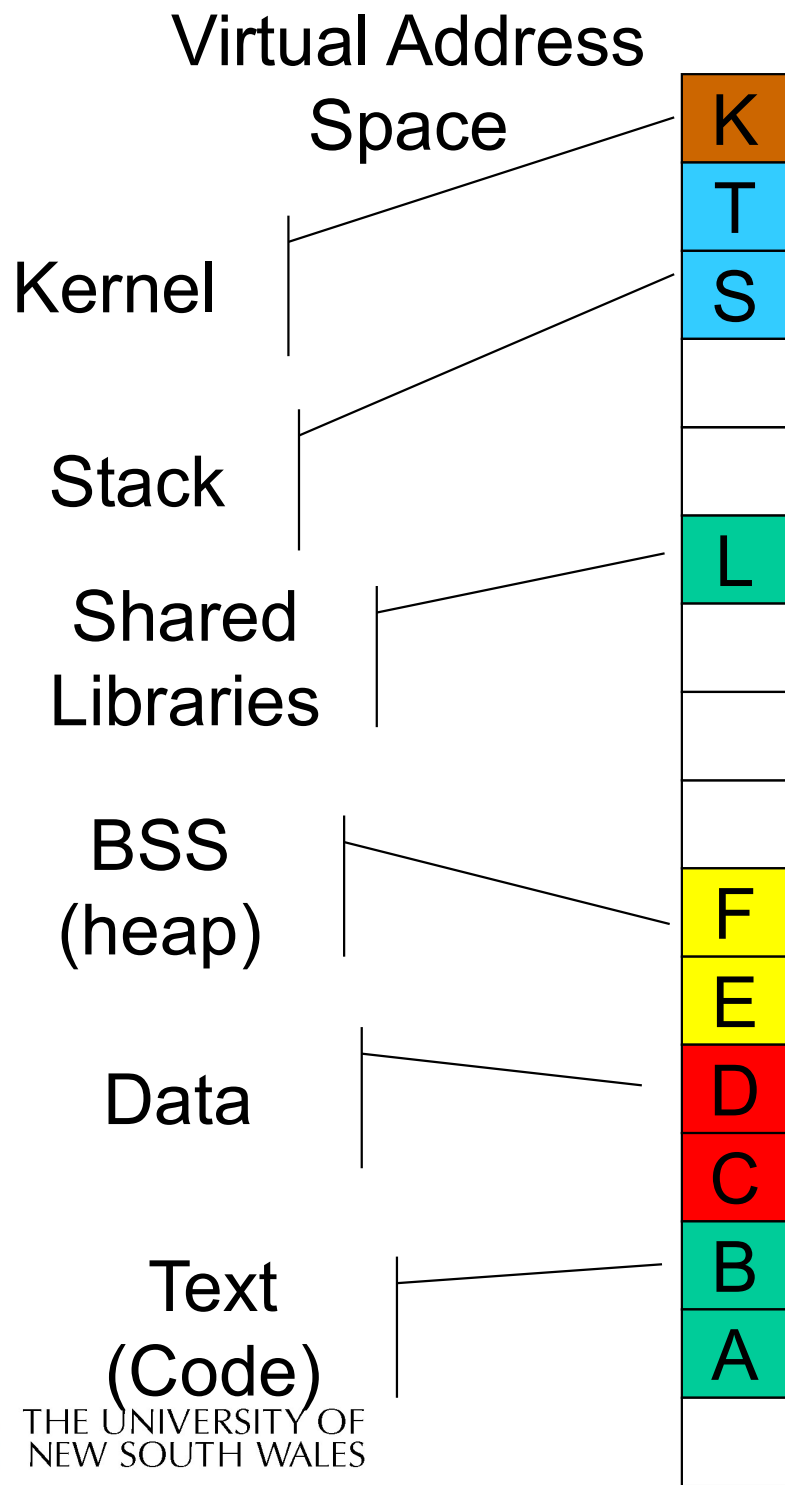
## Page-based VM

- Physical Memory
  - Divided into equal-sized *frames*



Physical Address Space 4

# Typical Address Space Layout



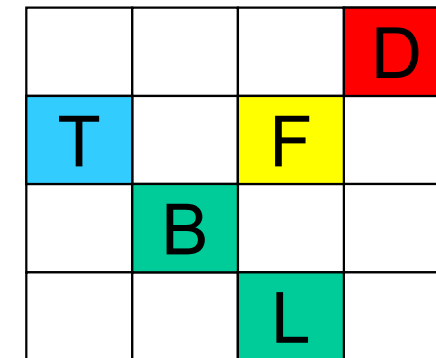
- Stack region is at top, and can grow down
- Heap has free space to grow up
- Text is typically read-only
- Kernel is in a reserved, protected, shared region
- 0-th page typically not used, why?

## Virtual Address Space

- A process may be only partially resident
  - Allows OS to store individual pages on disk
  - Saves memory for infrequently used data & code
- What happens if we access non-resident memory?



**Programmer's perspective:**  
logically present  
**System's perspective:** Not mapped, data on disk



Disk



Physical Address Space

# Proc 1 Address Space

Space

Currently running

Memory Access

# Proc 2 Address Space

Space

Physical Address Space

U			
T		Y	
	B		L
Z		J	

Disk



# Page Faults

- Referencing an invalid page triggers a page fault
  - An exception handled by the OS
- Broadly, two standard page fault types
  - Illegal Address (protection error)
    - Signal or kill the process
  - Page not resident
    - Get an empty frame
    - Load page from disk
    - Update page (translation) table (enter frame #, set valid bit, etc.)
    - Restart the faulting instruction





- Page table for resident part of address space

Virtual Address  
Space

15	K
14	T
13	S
12	
11	
10	L
9	
8	
7	
6	F
5	E
4	D
3	C
2	B
1	A
0	

Page  
Table

6	15
	14
0	13
	12
	11
	10
	9
	8
	7
	6
3	5
	4
1	3
	2
7	1
	0

Physical  
Address Space

A	7
K	6
	5
	4
E	3
	2
C	1
S	0



# Shared Pages

- Private code and data
  - Each process has own copy of code and data
  - Code and data can appear anywhere in the address space
- Shared code
  - Single copy of code shared between all processes executing it
  - Code must not be self modifying
  - Code must appear at same address in all processes



# Proc 1 Address Space

# Proc 2 Address Space

0
1
7
2

U
T
S
D
C
B
A

Z
Y
X
N
M
B
A

5
4
7
2

Physical Address Space

B
X
N
A
C
S

Two (or more) processes running the same program and sharing the text section

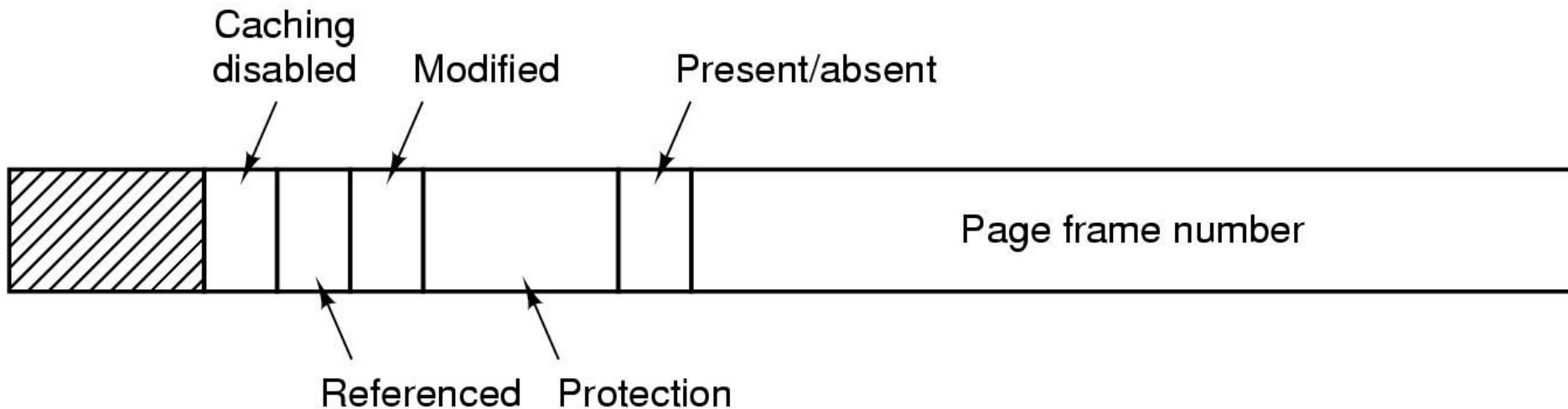
Page Table

Page Table



# Page Table Structure

- Page table is (logically) an array of frame numbers
  - Index by page number
- Each page-table entry (PTE) also has other bits



Page  
Table

12

5
4
7
2



# PTE Attributes (bits)

- Present/Absent bit
  - Also called *valid bit*, it indicates a valid mapping for the page
- Modified bit
  - Also called *dirty bit*, it indicates the page may have been modified in memory
- Reference bit
  - Indicates the page has been accessed
- Protection bits
  - Read permission, Write permission, Execute permission
  - Or combinations of the above
- Caching bit
  - Use to indicate processor should bypass the cache when accessing memory
    - Example: to access device registers or memory



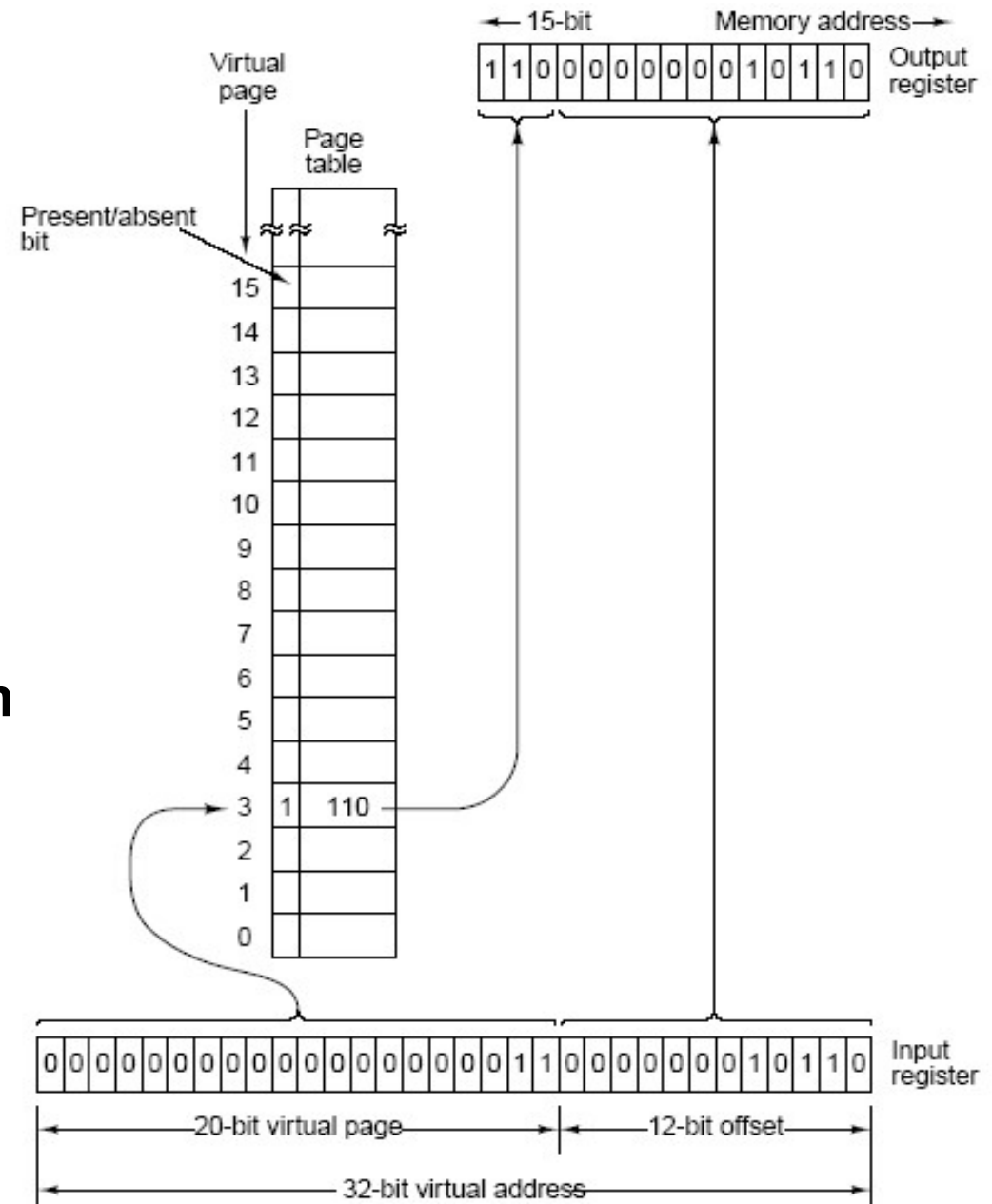
# Address Translation

- Every (virtual) memory address issued by the CPU must be translated to physical memory
  - Every *load* and every *store* instruction
  - Every instruction fetch
- Need Translation Hardware
- In paging system, translation involves replace page number with a frame number

# Virtual Memory Summary

virtual and physical mem chopped up in pages/frames

- programs **use virtual addresses**
- virtual to physical mapping by **MMU**
  - first check if page present (**present/absent bit**)
  - if yes: address in page table form MSBs in physical address
  - if no: bring in the page from disk  
→ **page fault**



# Page Tables

- Assume we have
  - 32-bit virtual address (4 Gbyte address space)
  - 4 KByte page size
  - How many page table entries do we need for one process?





# Page Tables

- Assume we have
  - 64-bit virtual address (humungous address space)
  - 4 KByte page size
  - How many page table entries do we need for one process?
- Problem:
  - Page table is very large
  - Access has to be fast, lookup for every memory reference
  - Where do we store the page table?
    - Registers?
    - Main memory?



# Page Tables

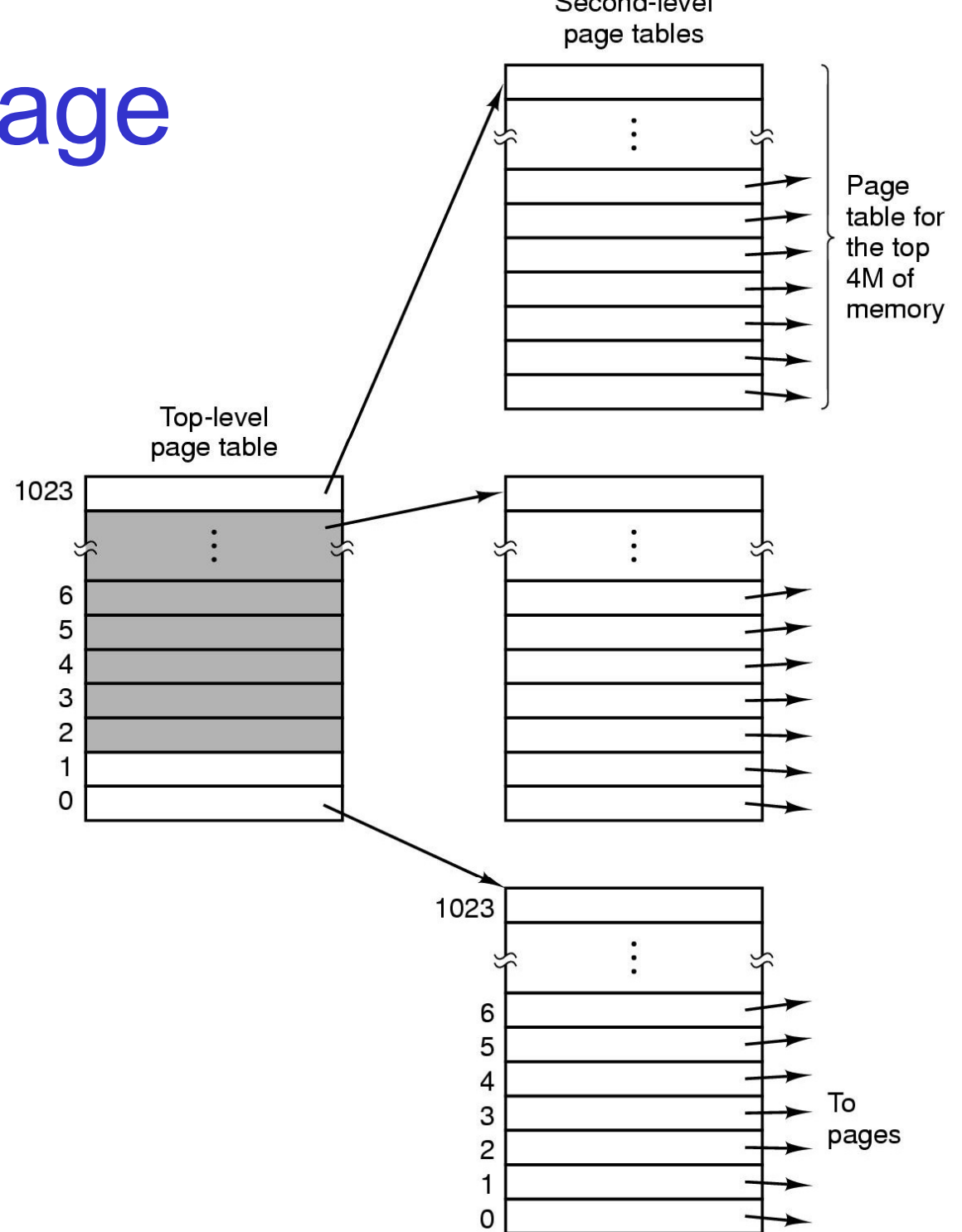
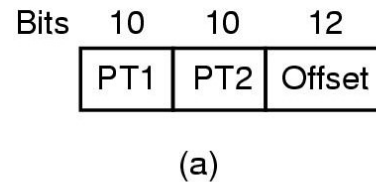
- Page tables are implemented as data structures in main memory
- Most processes do not use the full 4GB address space
  - e.g., 0.1 – 1 MB text, 0.1 – 10 MB data, 0.1 MB stack
- We need a compact representation that does not waste space
  - But is still very fast to search
- Three basic schemes
  - Use data structures that adapt to sparsity
  - Use data structures which only represent resident pages
  - Use VM techniques for page tables (details left to extended OS)



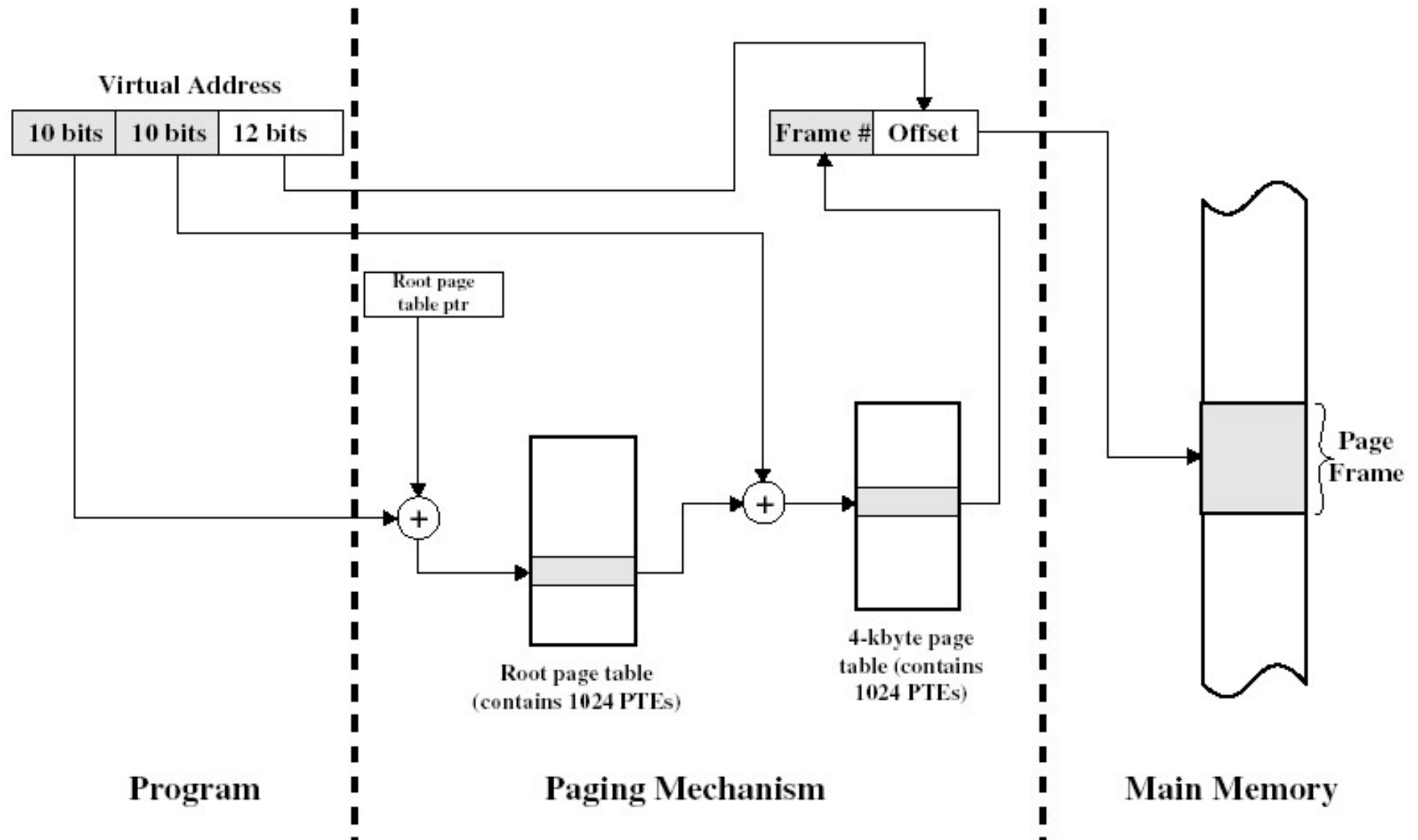
# Two-level Page Table

- 2<sup>nd</sup> -level page tables representing unmapped pages are not allocated

- Null in the top-level page table



# Two-level Translation



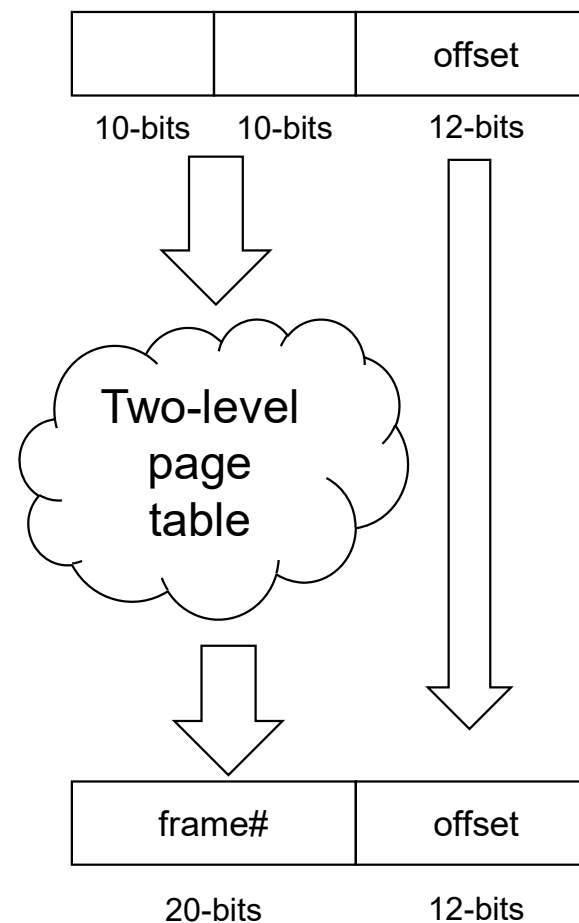
# Example Translations





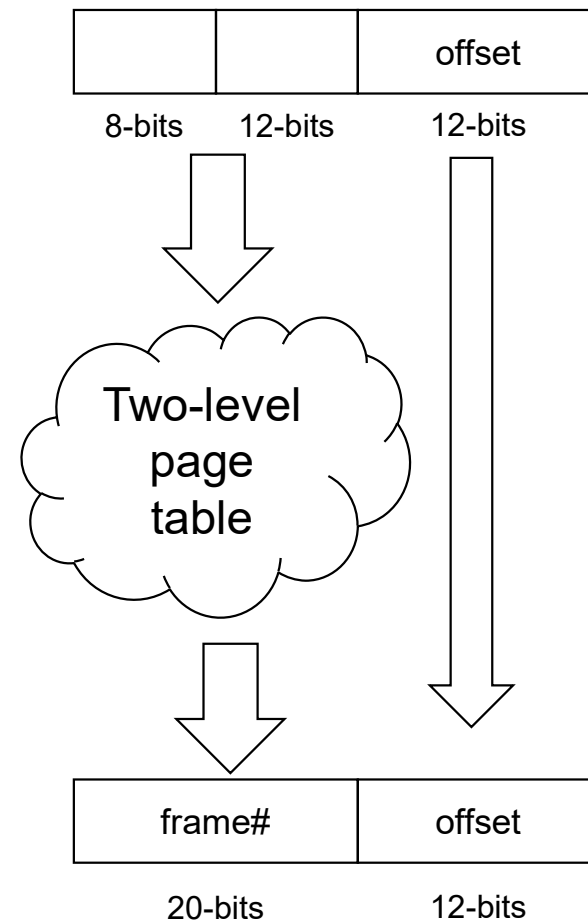
# Summarising Two-level Page Tables

- Translating a 32-bit virtual address into a 32-bit physical
- Recall:
  - the level 1 page table node has  $2^{10}$  entries
    - $2^{10} * 4 = 4$  KiB node
  - the level 2 page table node have  $2^{10}$  entries
    - $2^{10} * 4 = 4$  KiB node



# Index bits determine node sizes

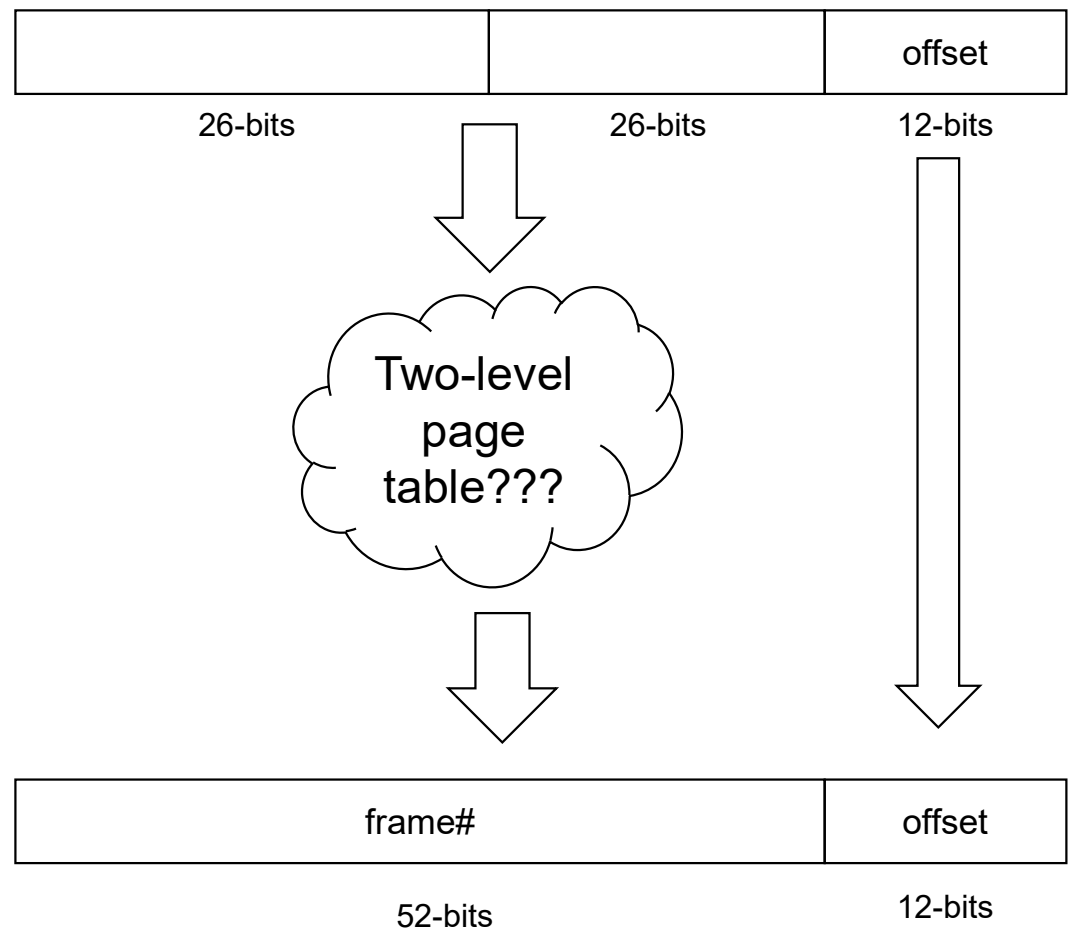
- Translating a 32-bit virtual address into a 32-bit physical
- Changing the indexing:
  - the level 1 page table node has  $2^8$  entries
    - $2^8 * 4 = 1$  KiB node
  - the level 2 page table node have  $2^{12}$  entries
    - $2^{12} * 4 = 16$  KiB node





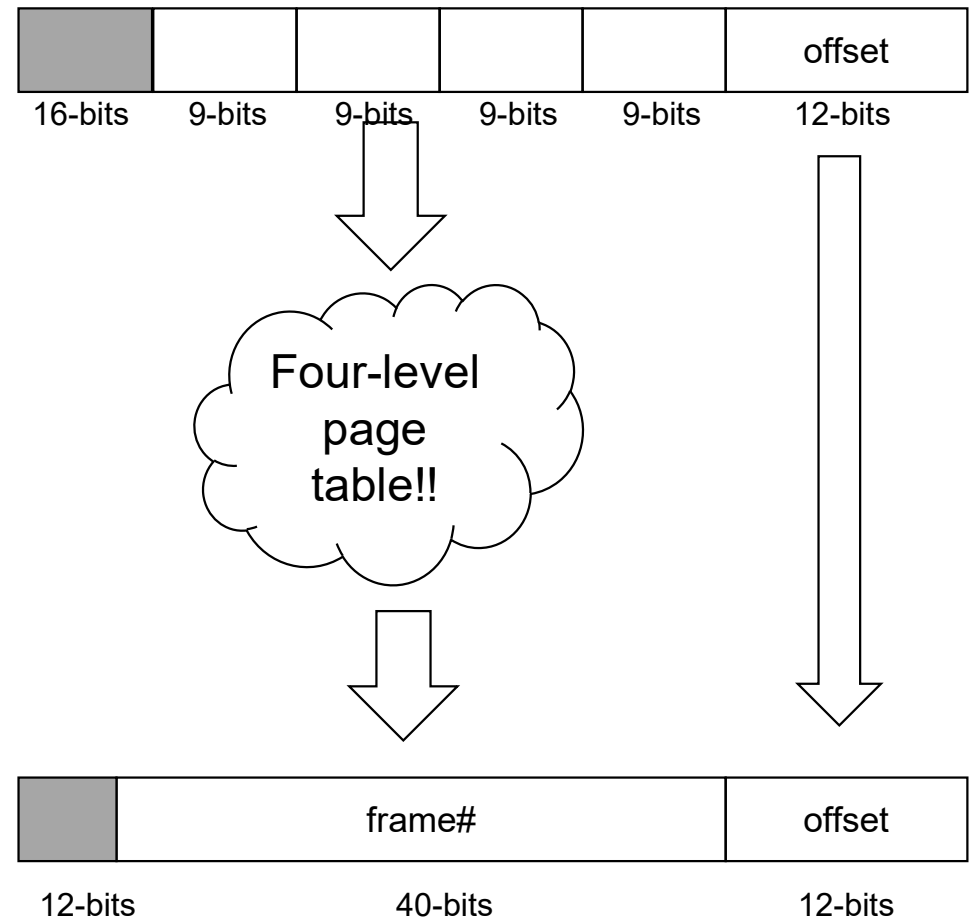
# Supporting 64-bit Virtual to Physical Translation

- Translating a 64-bit virtual address into a 64-bit physical???
- Support 64-bits?:
  - the level 1 page table node has  $2^{26}$  entries
    - $2^{26} * 8 = 512$  MiB node
  - the level 2 page table node have  $2^{12}$  entries
    - $2^{12} * 8 = 512$  MiB node



# Multi-level Page Tables

- Translating a 64-bit virtual address into a 64-bit physical (Intel/AMD pre-Ice Lake)
  - Only support 48-bit addresses
    - Top 16-bits unused
  - the level 1 page table node has  $2^9$  entries
    - $2^9 * 8 = 4$  KiB node
  - the level 2 page table node have  $2^9$  entries
    - $2^9 * 8 = 4$  KiB node
  - the level 3 page table node have  $2^9$  entries
    - $2^9 * 8 = 4$  KiB node
  - the level 4 page table node have  $2^9$  entries
    - $2^9 * 8 = 4$  KiB node



# Intel 4-Level Page Tables

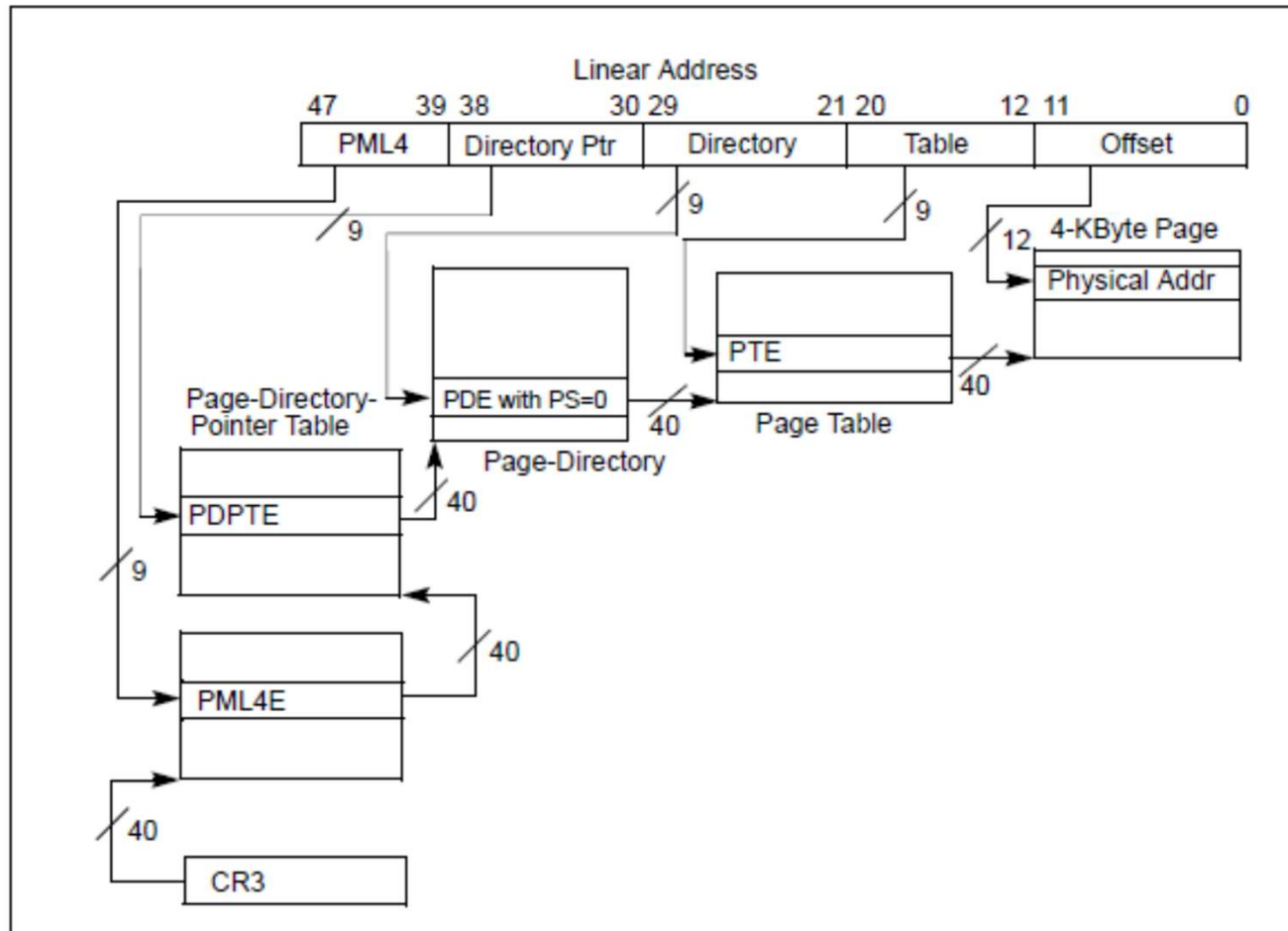
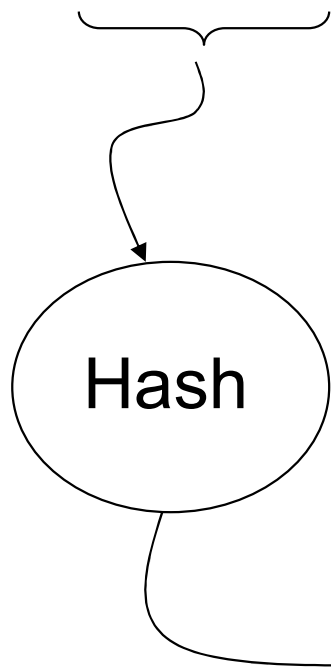


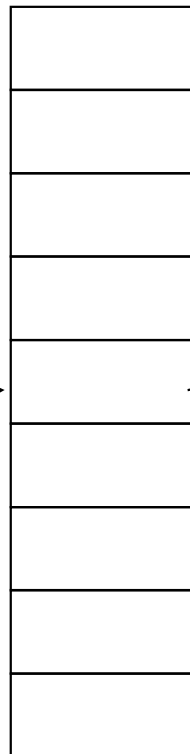
Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

# Alternative: Inverted Page Table

PID    VPN    offset



Hash Anchor Table  
(HAT)



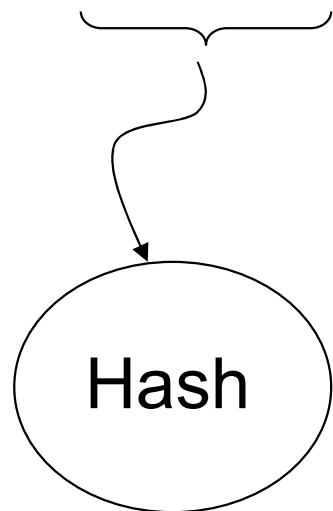
Index	PID	VPN	ctrl	next
0				
1				
2				
3				
4				
5				
6				
...				

IPT: entry for each *physical* frame



# Alternative: Inverted Page Table

PID	VPN	offset
0	0x5	0x123



Hash Anchor Table  
(HAT)

2

Index	PID	VPN	ctrl	next
0				
1				
2	1	0x1A		0x40C
...				
0x40C	0	0x5		0x0
0x40D				
...				
...				

ppn	offset
0x40C	0x123



# Inverted Page Table (IPT)

- “Inverted page table” is an array of page numbers sorted (indexed) by frame number (it’s a frame table).
- Algorithm
  - Compute hash of page number
  - Extract index from hash table
  - Use this to index into inverted page table
  - Match the PID and page number in the IPT entry
  - If match, use the index value as frame # for translation
  - If no match, get next candidate IPT entry from chain field
  - If NULL chain entry  $\Rightarrow$  page fault



# Properties of IPTs

- IPT grows with size of RAM, NOT virtual address space
- Frame table is needed anyway (for page replacement, more later)
- Need a separate data structure for non-resident pages
- Saves a vast amount of space (especially on 64-bit systems)
- Used in some IBM and HP workstations



# Given $n$ processes

- how many page tables will the system have for
  - ‘normal’ page tables
  - inverted page tables?





# Another look at sharing...



Proc 1 Address  
Space

Proc 2 Address  
Space

*'easy' for normal page tables*

*bit of a nightmare for IPT*

Two (or more)  
processes  
running the  
same program  
and sharing  
the text section



# Improving the IPT: Hashed Page Table

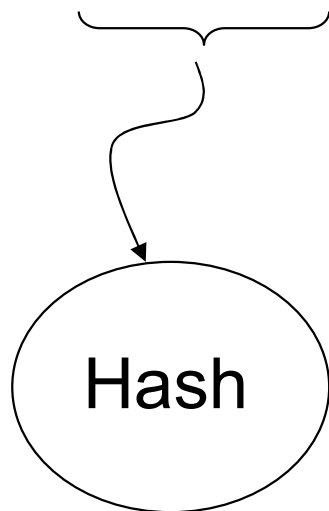
- Retain fast lookup of IPT
  - A single memory reference in best case
- Retain page table sized based on physical memory size (not virtual)
  - Enable efficient frame sharing
  - Support more than one mapping for same frame



# Hashed Page Table

PID    VPN    offset

--	--



Best-case lookup: one memory reference

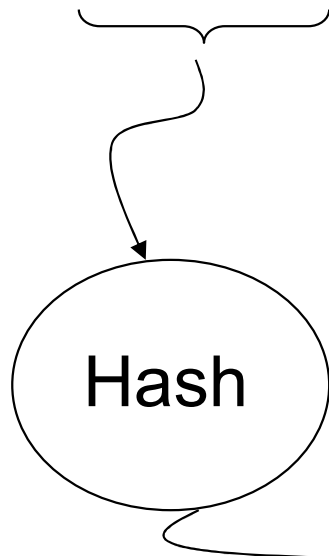
PID	VPN	offset			

HPT: Frame number stored in table



# Hashed Page Table

PID	VPN	offset
0	0x5	0x123



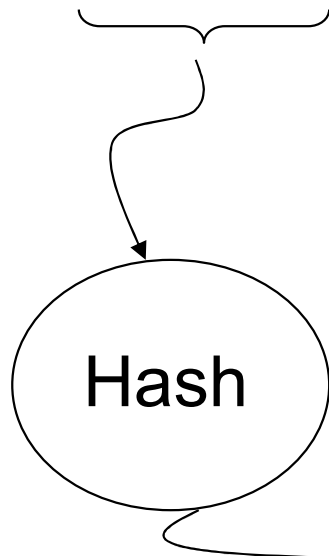
	PID	VPN	PFN	ctrl	next
1					
2					
3	0	0x5	0x42		0x0
4					
5					
6	1	0x1A	0x13		0x3
...					

ppn	offset
0x42	0x123



# Sharing Example

PID	VPN	offset
0	0x5	0x123



	PID	VPN	PFN	ctrl	next
1					
2					
3	1	0x5	0x42		0x0
4					
5					
6	0	0x5	0x42		0x3
...					

ppn	offset
0x42	0x123



# Sizing the Hashed Page Table

- HPT sized based on physical memory size
- With sharing
  - Each frame can have more than one PTE
  - More sharing increases number of slots used
    - Increases collision likelihood
- However, we can tune HPT size based on:
  - Physical memory size
  - Expected sharing
  - Hash collision avoidance.
  - HPT a power of 2 multiple of number of physical memory frame



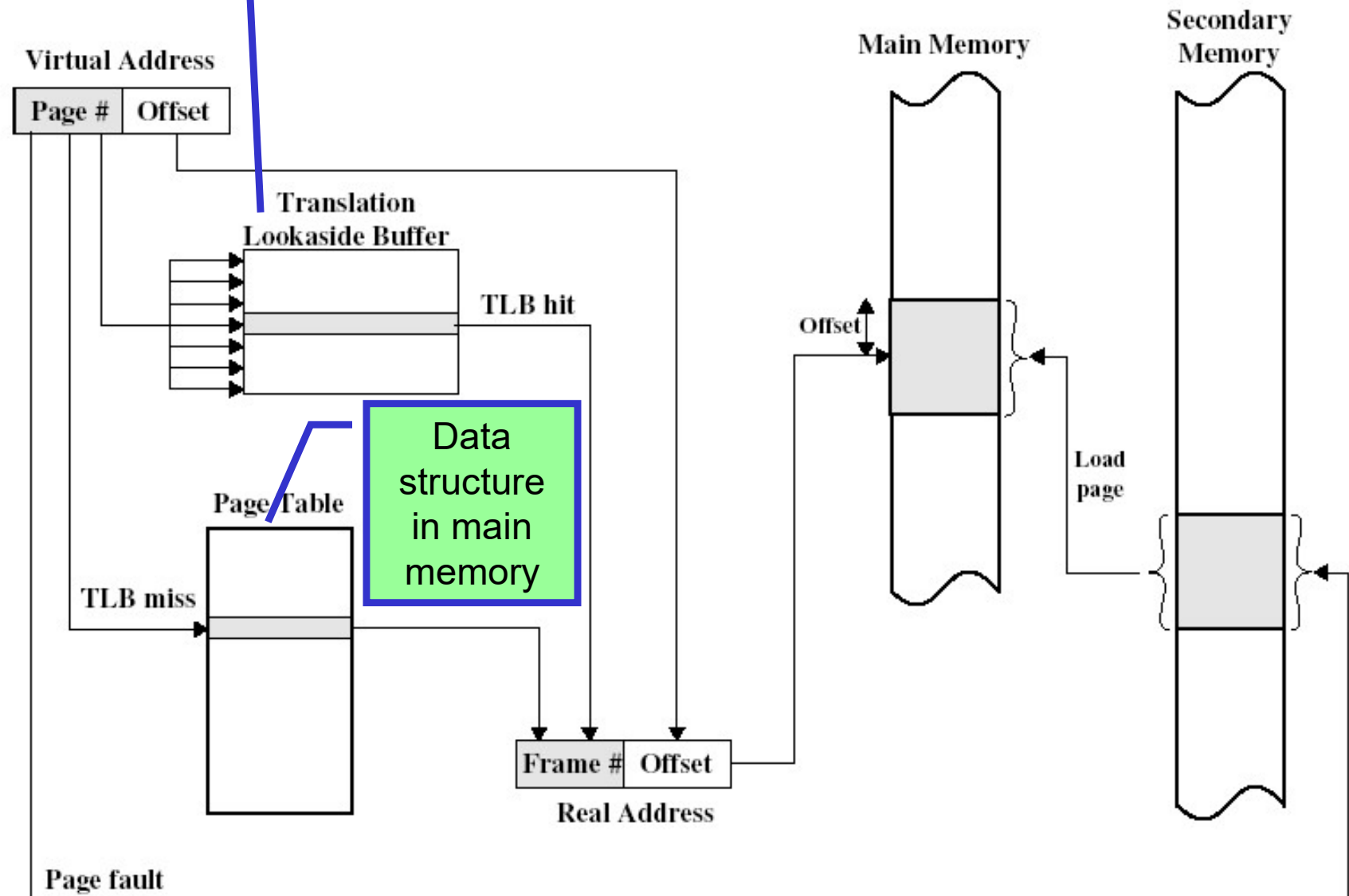
# VM Implementation Issue

- Performance?
  - Each virtual memory reference can cause two physical memory accesses
    - One to fetch the page table entry
    - One to fetch/store the data
  - ⇒ Intolerable performance impact!!
- Solution:
  - High-speed cache for page table entries (PTEs)
    - Called a *translation look-aside buffer* (TLB)
    - Contains recently used page table entries
    - Associative, high-speed memory, similar to cache memory
    - May be under OS control (unlike memory cache)



On-CPU  
hardware  
device!!!

# TLB operation



# Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If matching PTE found (*TLB hit*), the address is translated
- Otherwise (*TLB miss*), the page number is used to index the process's page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise, (page fault) check if page is on disk
    - If on disk, swap it in
    - Otherwise, allocate a new page or raise an exception



# TLB properties

- Page table is (logically) an array of frame numbers
- TLB holds a (recently used) subset of PT entries
  - Each TLB entry must be identified (tagged) with the page # it translates
  - Access is by associative lookup:
    - All TLB entries' tags are concurrently compared to the page #
    - TLB is associative (or content-addressable) memory

<i>page #</i>	<i>frame #</i>	<i>V</i>	<i>W</i>
...	...	.	.
...	...	.	.

# TLB properties

- TLB may or may not be under direct OS control
  - Hardware-loaded TLB
    - On miss, hardware performs PT lookup and reloads TLB
    - Example: x86, ARM
  - Software-loaded TLB
    - On miss, hardware generates a TLB miss exception, and exception handler reloads TLB
    - Example: MIPS, Itanium (optionally)
- TLB size: typically 64-128 entries
- Can have separate TLBs for instruction fetch and data access
- TLBs can also be used with inverted page tables (and others)



# TLB and context switching

- TLB is a shared piece of hardware
- Normal page tables are per-process (address space)
- TLB entries are *process-specific*
  - On context switch need to *flush* the TLB (invalidate all entries)
    - high context-switching overhead (Intel x86)
  - **or** tag entries with *address-space ID* (ASID)
    - called a *tagged TLB*
    - used (in some form) on all modern architectures
    - TLB entry: ASID, page #, frame #, valid and write-protect bits

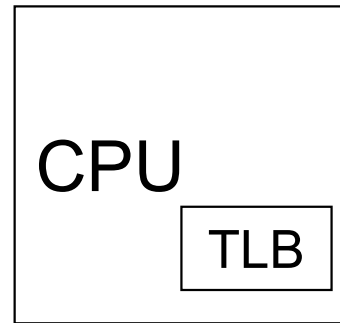
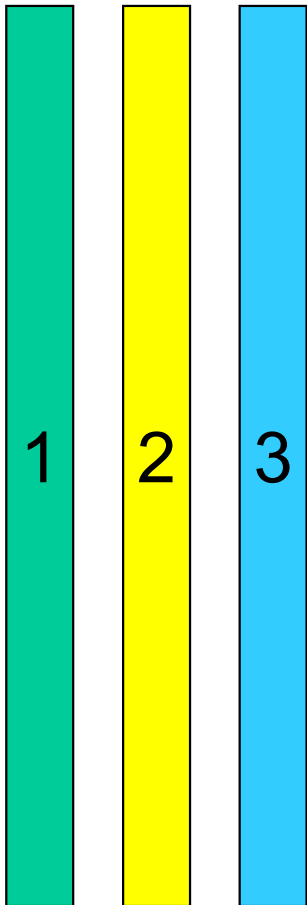


# TLB effect

- Without TLB
  - Average number of physical memory references per virtual reference  
 $= 2$
- With TLB (assume 99% hit ratio)
  - Average number of physical memory references per virtual reference  
 $= .99 * 1 + 0.01 * 2$   
 $= 1.01$

# Recap - Simplified Components of VM System

Virtual Address Spaces  
(3 processes)

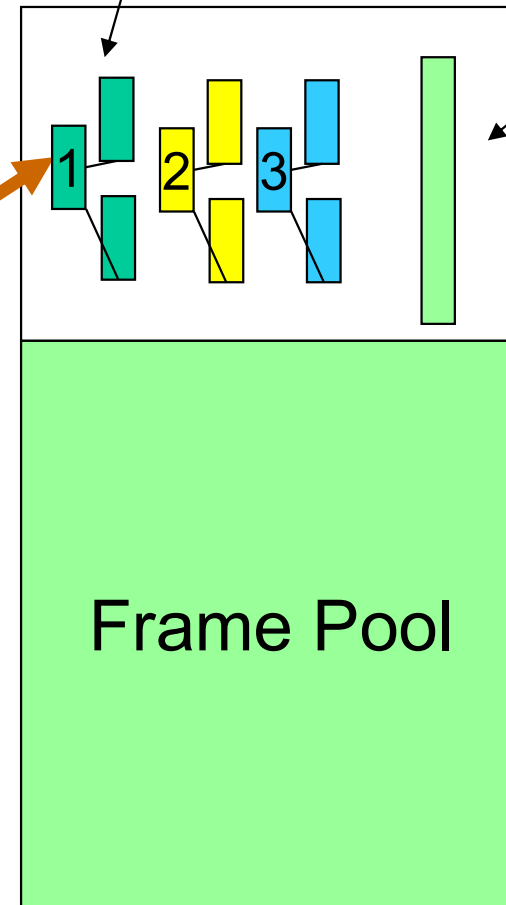


TLB Refill  
Mechanism



Page Tables for 3  
processes

Frame Table

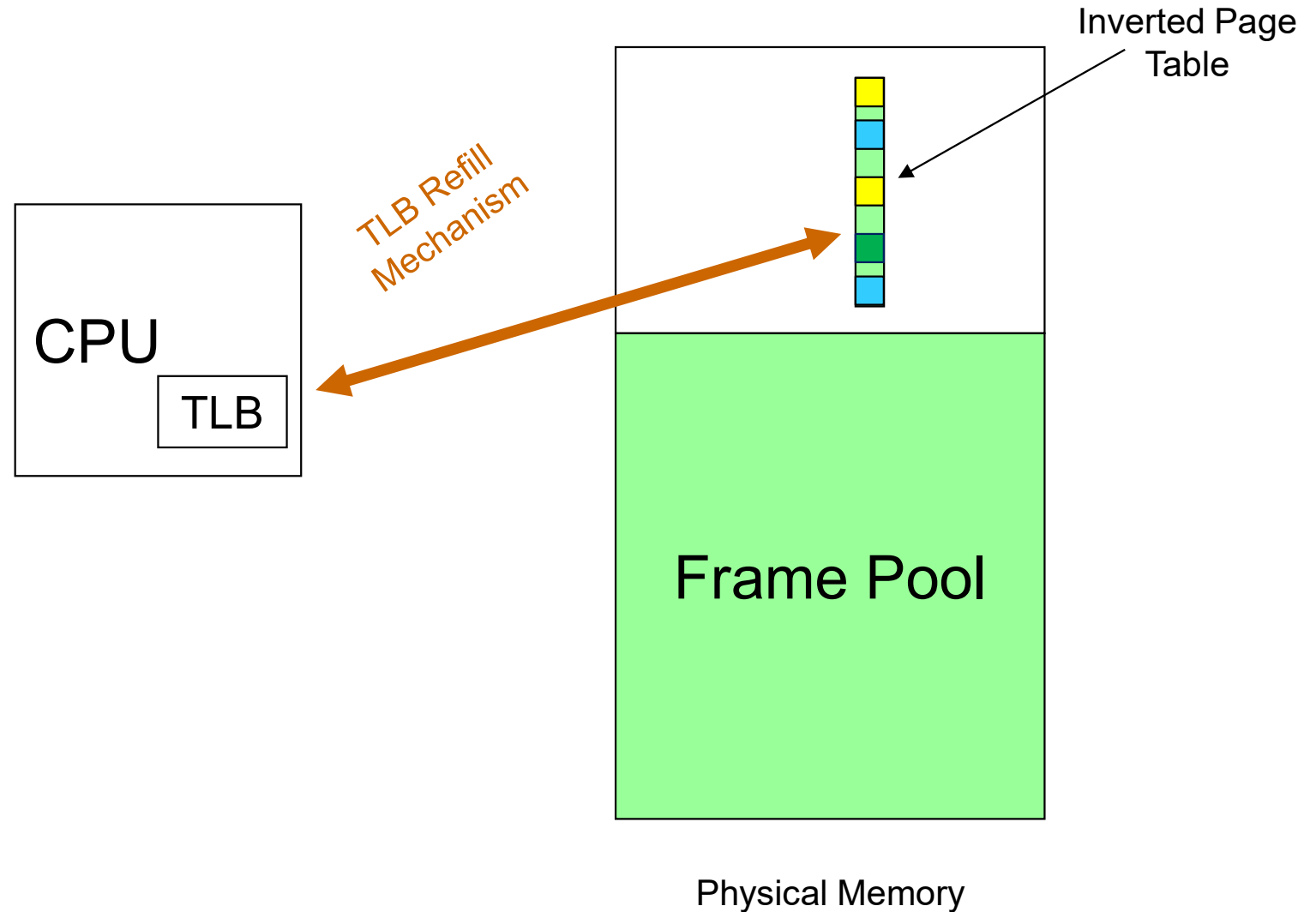


Frame Pool

Physical Memory



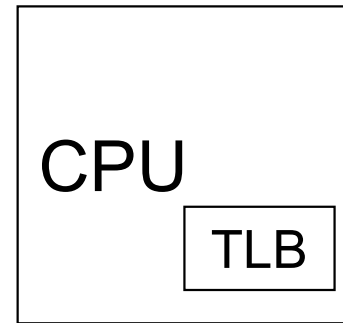
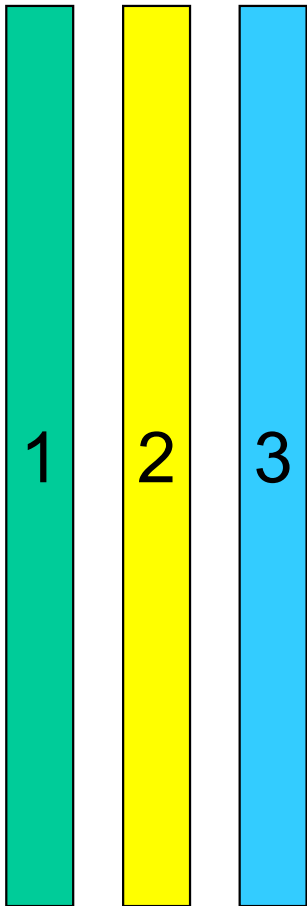
## Virtual Address Spaces (3 processes)





# Recap - Simplified Components of VM System

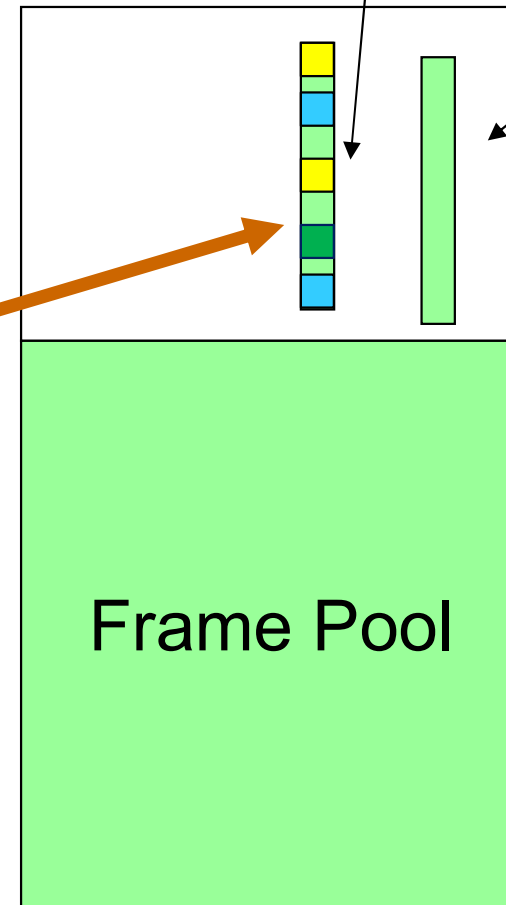
Virtual Address Spaces  
(3 processes)



TLB Refill  
Mechanism

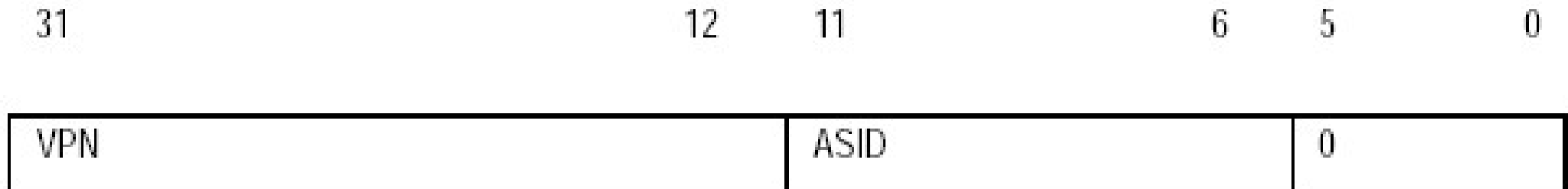
Hashed Page  
Table

Frame Table

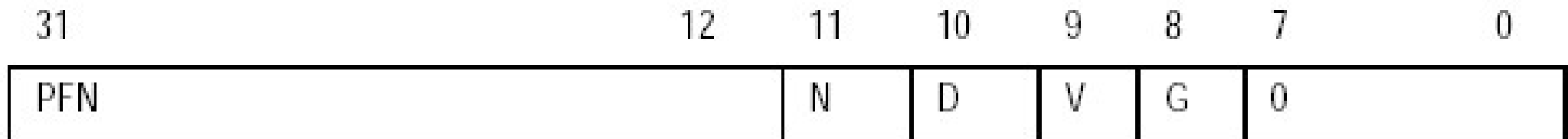


Physical Memory

# MIPS R3000 TLB



### EntryHi Register (TLB key fields)

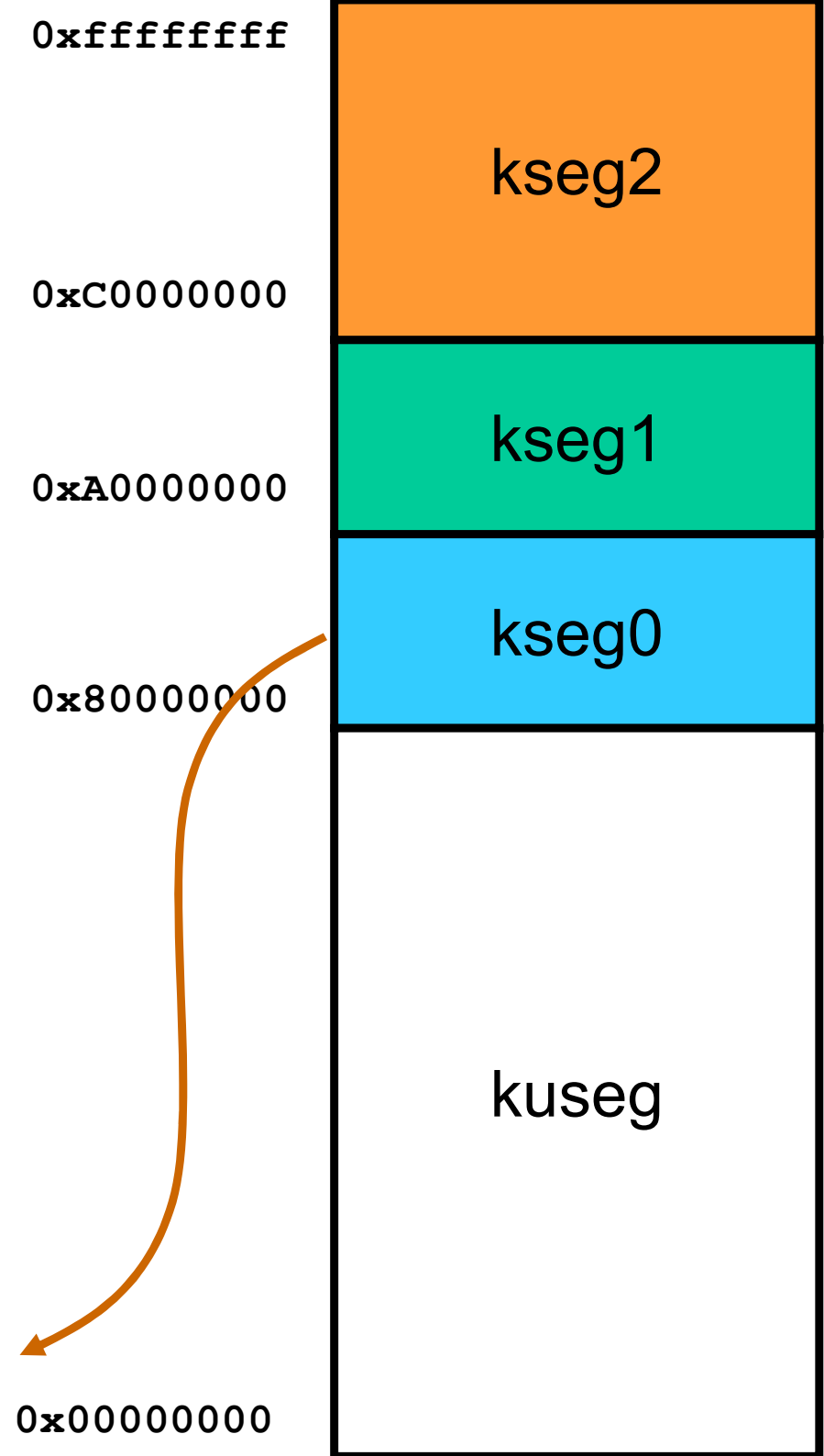


### EntryLo Register (TLB data fields)

- N = Not cacheable
- D = Dirty = Write protect
- G = Global (ignore ASID in lookup)
- V = valid bit
- 64 TLB entries
- Accessed via software through Coprocessor 0 registers
  - EntryHi and EntryLo

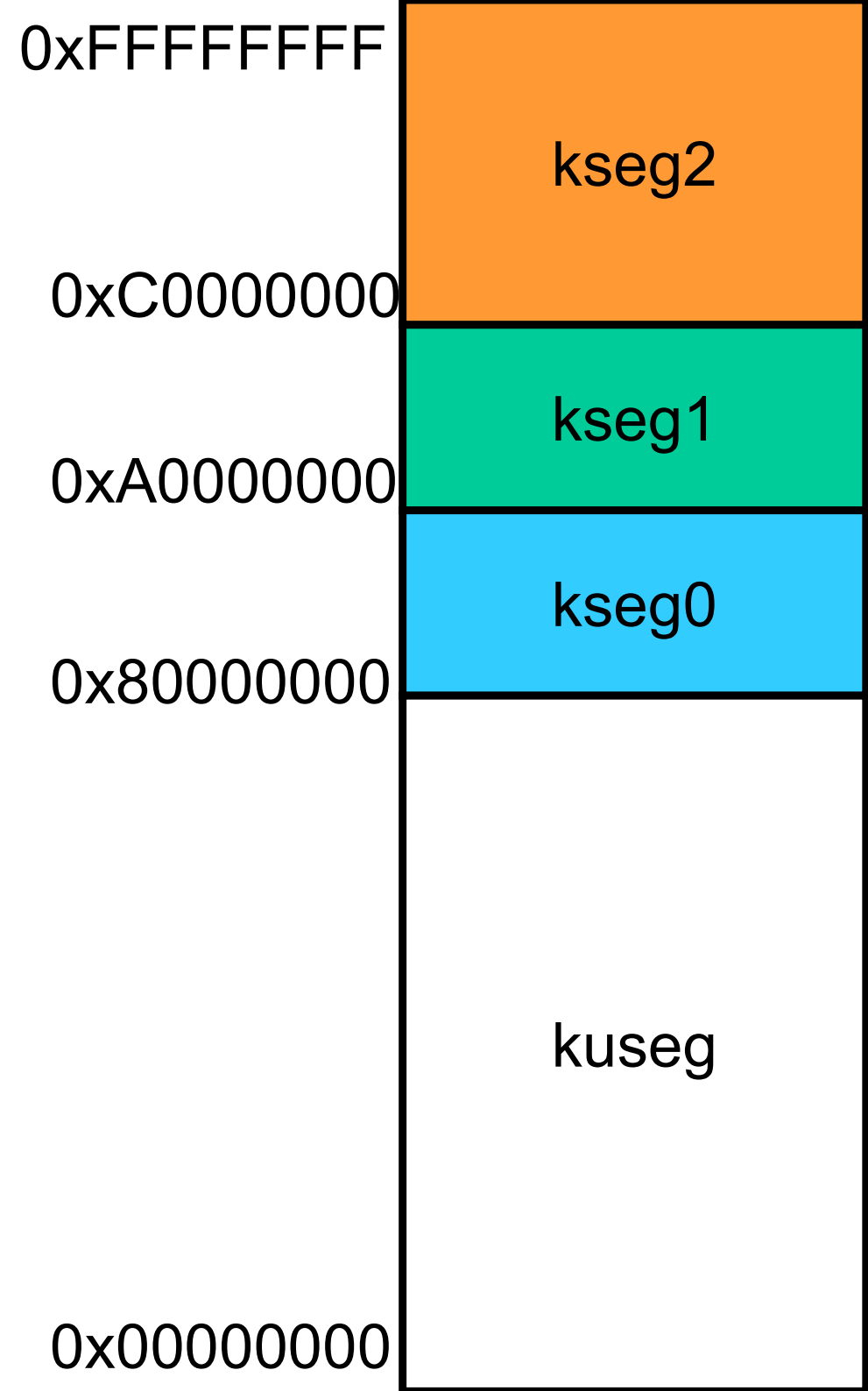
# R3000 Address Space Layout

- kseg0:
  - 512 megabytes
  - Fixed translation window to physical memory
    - 0x80000000 - 0x9fffffff virtual = 0x00000000 - 0x1fffffff physical
    - TLB not used
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code and data is placed



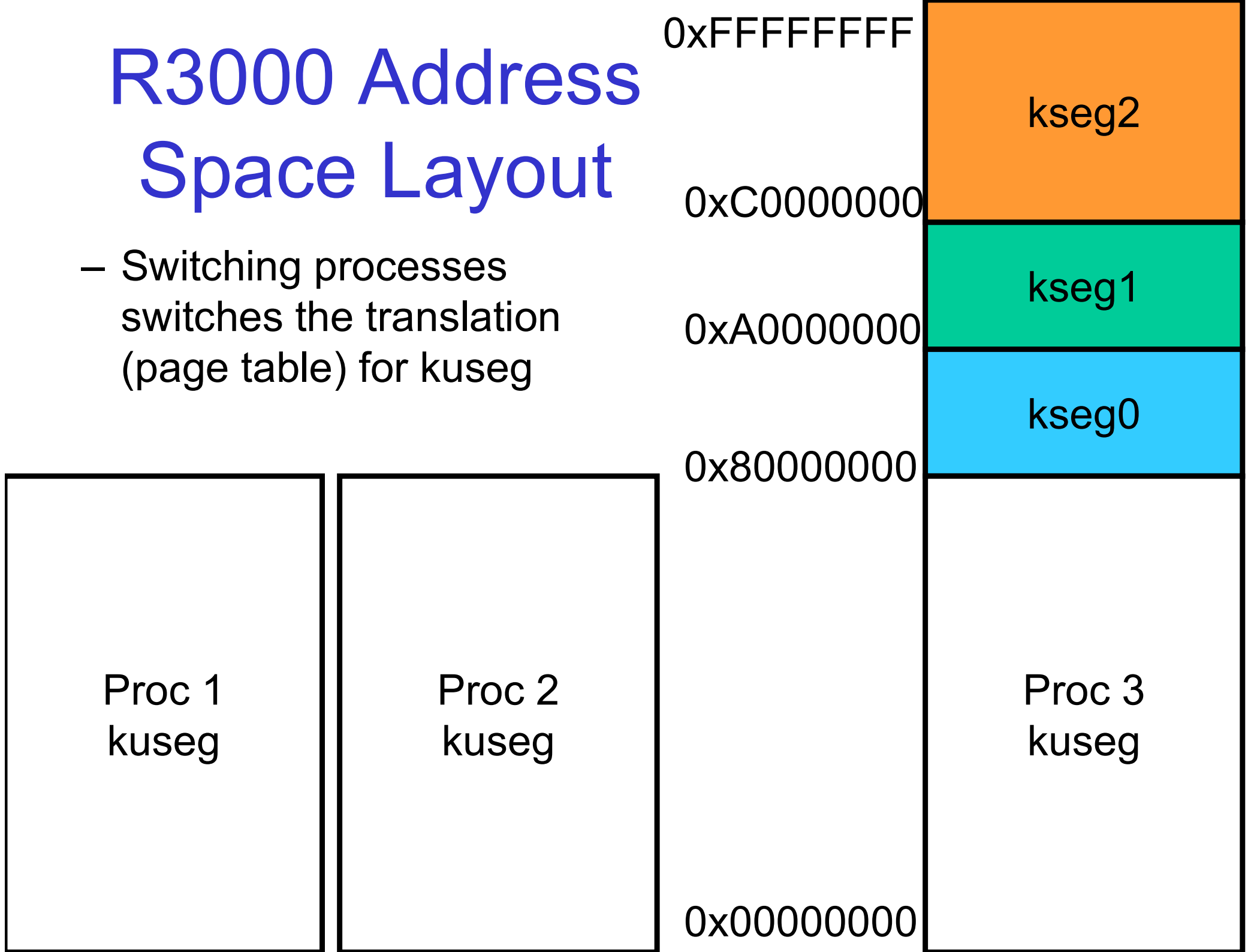
# R3000 Address Space Layout

- kuseg:
  - 2 gigabytes
  - TLB translated (mapped)
  - Cacheable (depending on 'N' bit)
  - user-mode and kernel mode accessible
  - Page size is 4K



# R3000 Address Space Layout

- Switching processes switches the translation (page table) for kuseg



# R3000 Address Space Layout

- kseg1:
  - 512 megabytes
  - Fixed translation window to physical memory
    - 0xa0000000 - 0xbfffffff virtual = 0x00000000 - 0x1fffffff physical
    - TLB not used
  - **NOT** cacheable
  - Only kernel-mode accessible
  - Where devices are accessed (and boot ROM)

0xffffffff

0xC0000000

0xA0000000

0x80000000

0x00000000

kseg2

kseg1

kseg0

kuseg

Physical Memory

