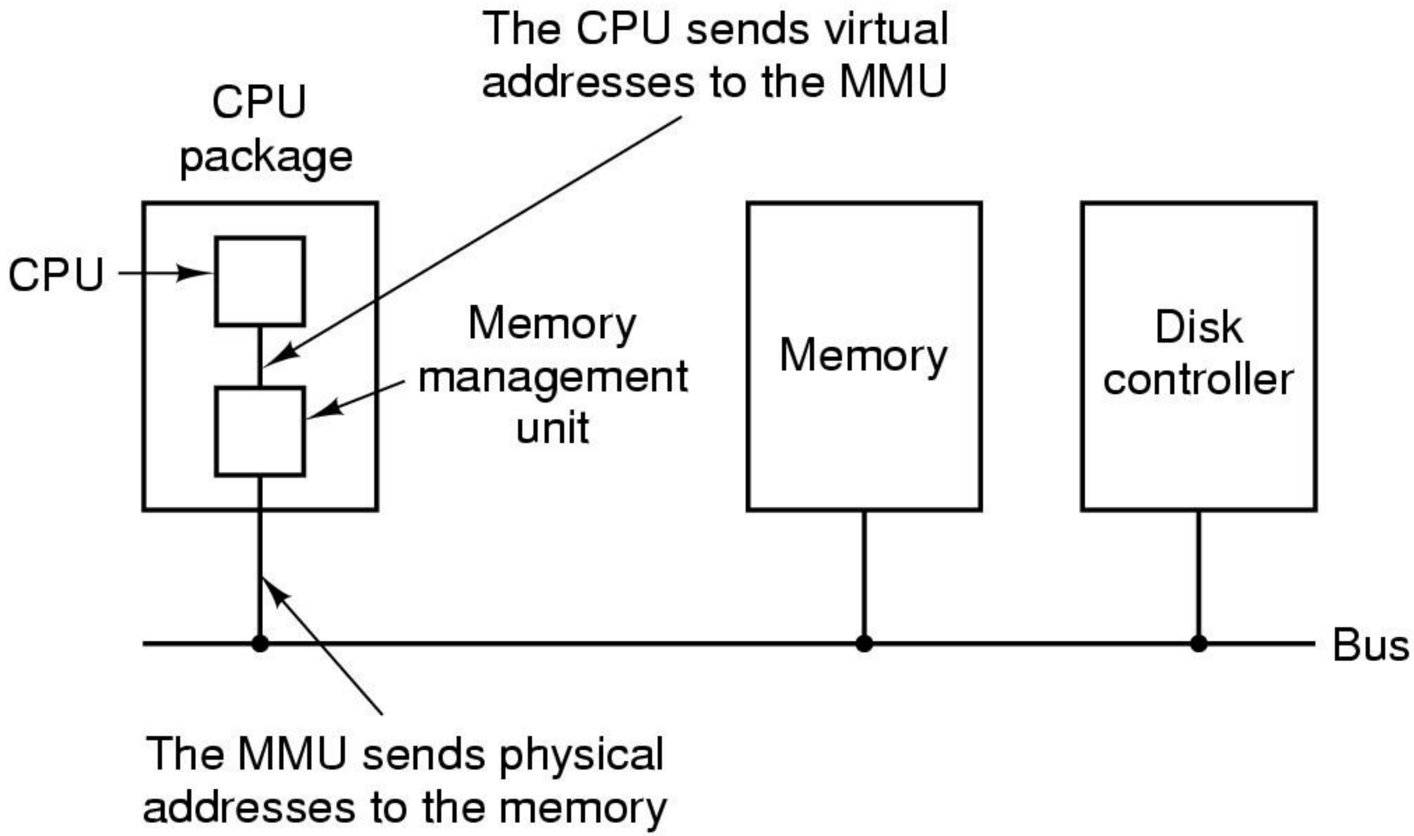
Virtual Memory

Learning Outcomes

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

Memory Management Unit (or TLB)



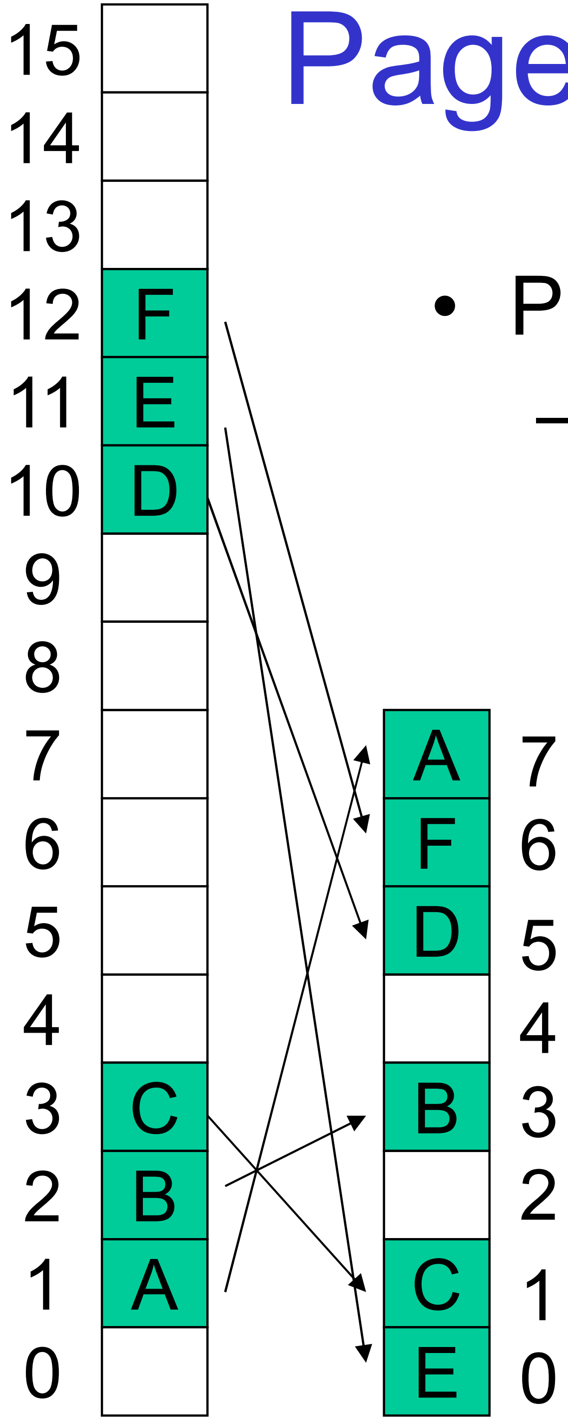
The position and function of the MMU

Page-based VM

if each page is 4k, we have a total of 64k pages

16 pages

Virtual Address Space

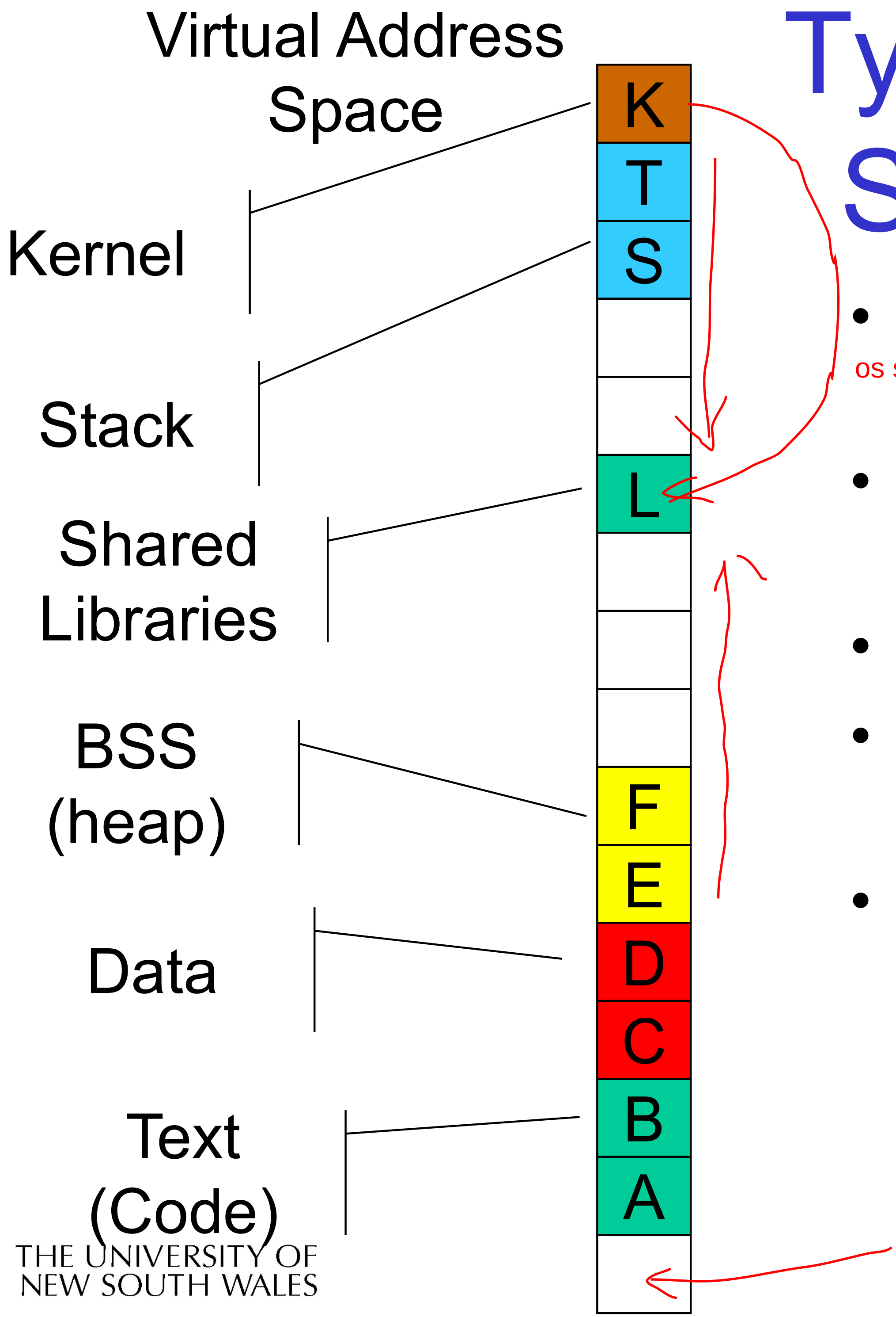


Physical Address Space

- 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

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

Typical Address Space Layout



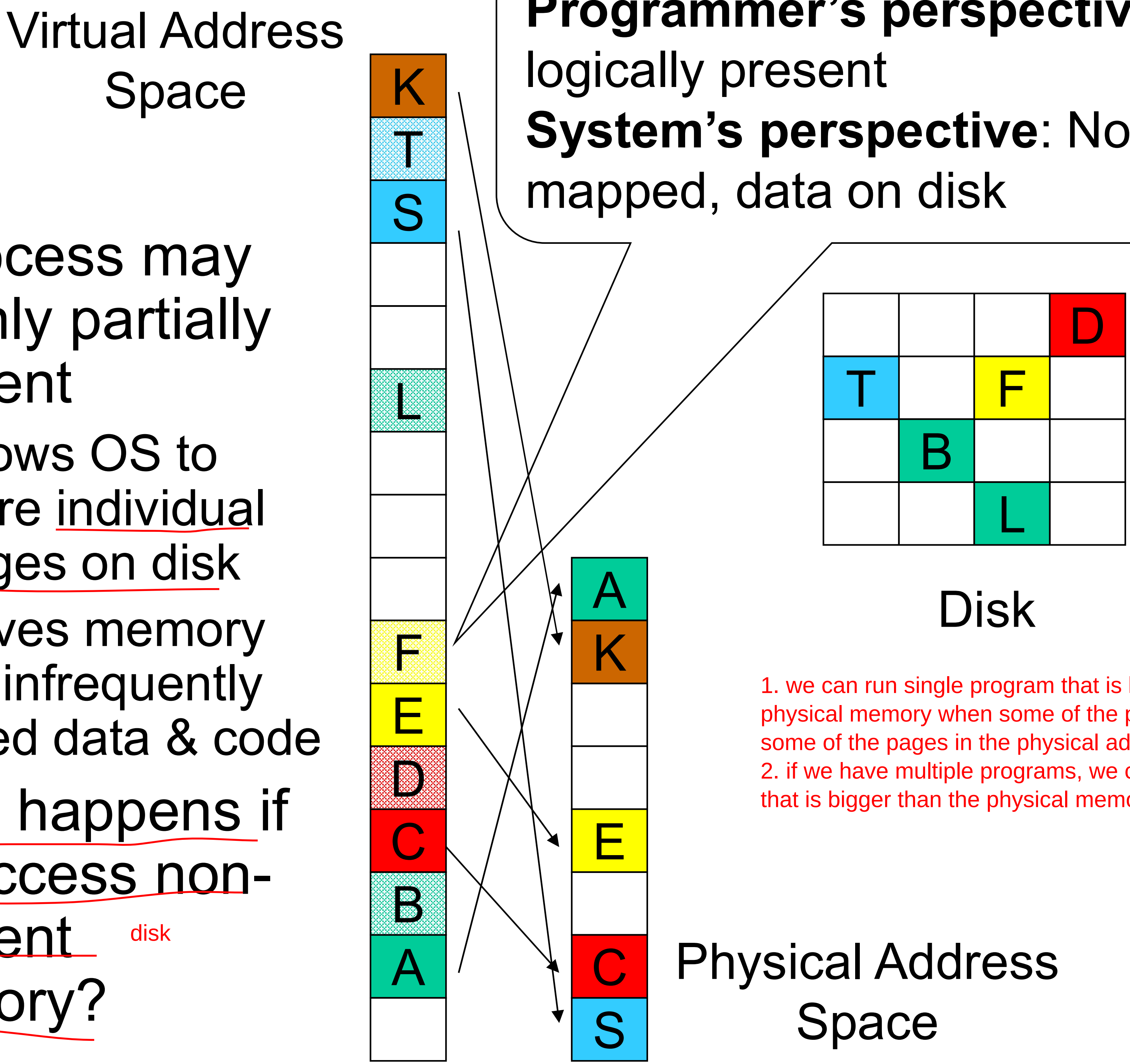
malloc

- 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?

os specific

this is null, not mapped to memory
never valid in any OS

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

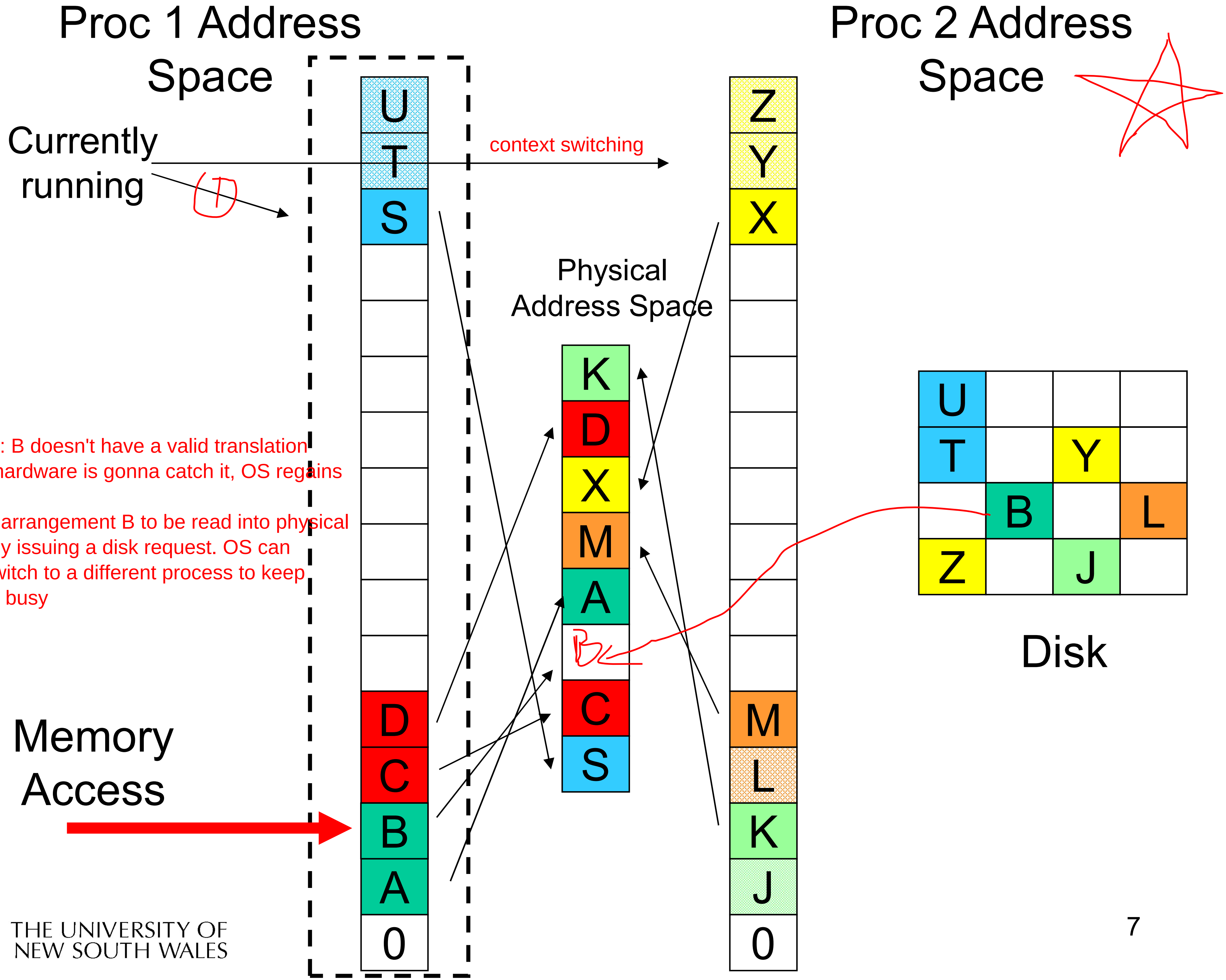


- A process may be only partially resident
  - Allows OS to store individual pages on disk
  - Saves memory for infrequently used data & code
- What happens if we access non-resident memory?

disk

1. we can run single program that is larger than the physical memory when some of the pages in the disk and some of the pages in the physical address space
2. if we have multiple programs, we can have total pages that is bigger than the physical memory

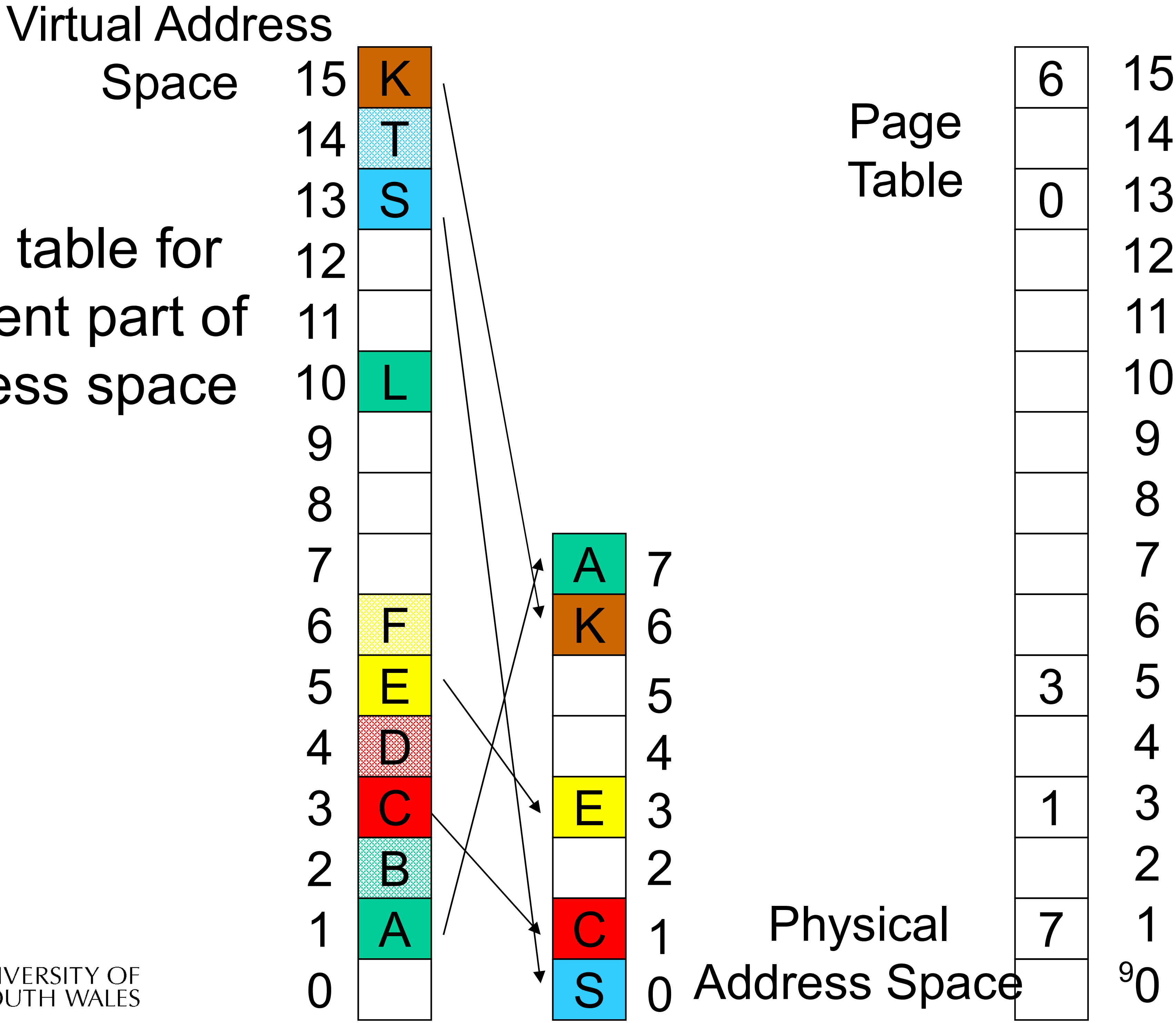
page fault: B doesn't have a valid translation now, the hardware is gonna catch it, OS regains control
OS: it will arrangement B to be read into physical memory by issuing a disk request. OS can context switch to a different process to keep processor busy



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) something like null
        - 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

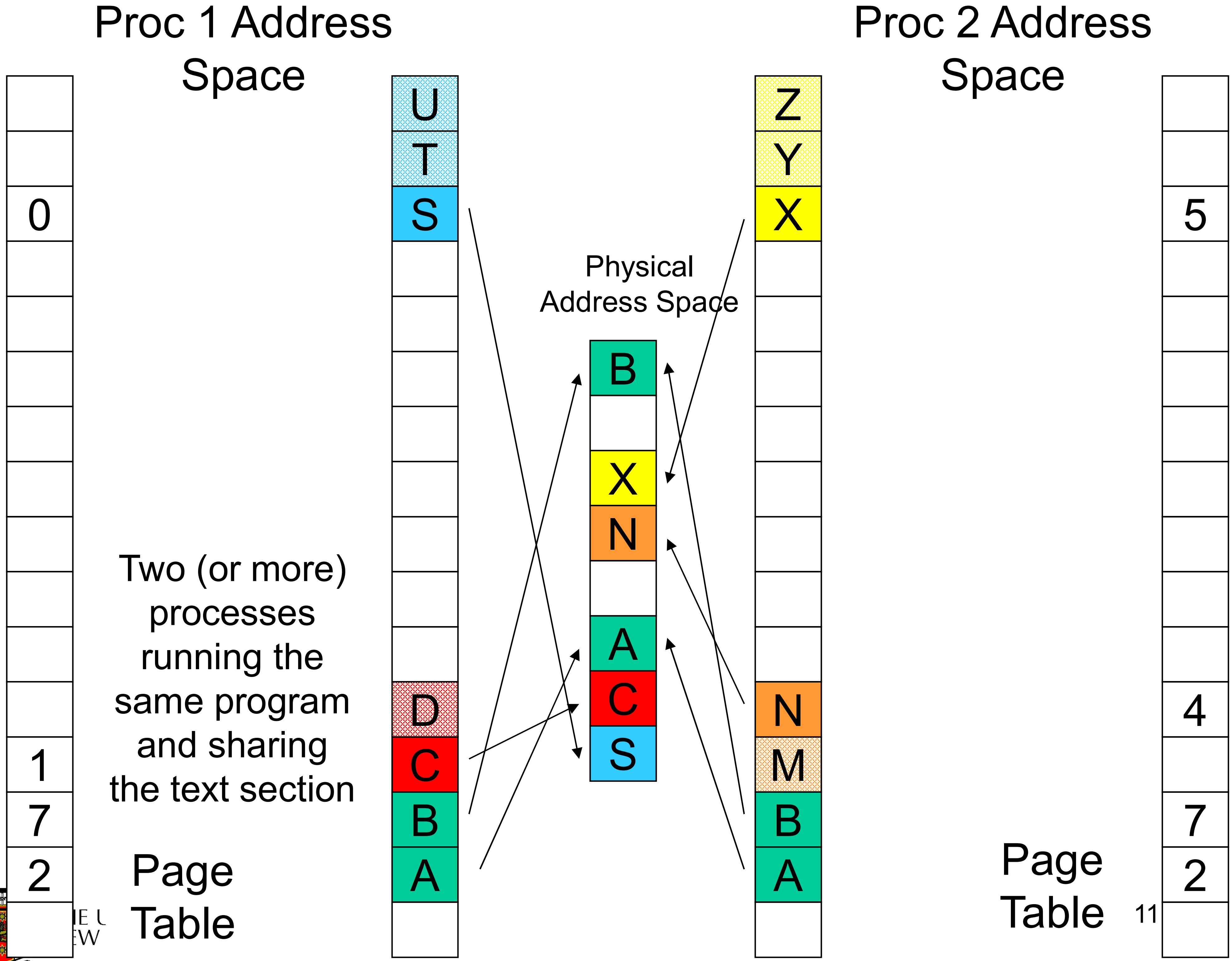
- Page table for resident part of address space



THE UNIVERSITY OF NEW SOUTH WALES

Shared Pages

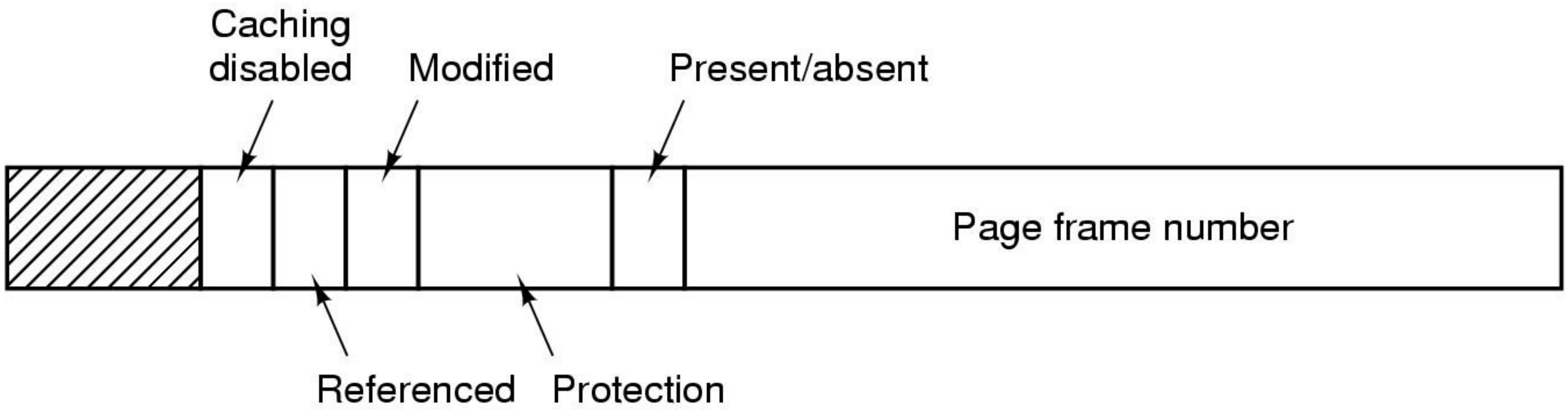
- Private code and data
  - Each process has own copy of code and data
  - Code and data can appear anywhere in the address space
- Shared code
  - Single copy of code shared between all processes executing it
  - Code must not be self modifying
  - Code must appear at same address in all processes

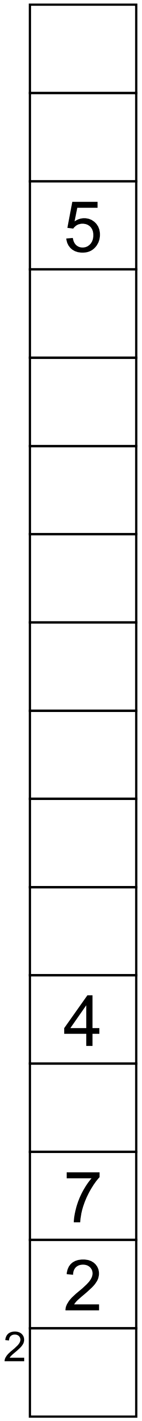


Two (or more) processes running the same program and sharing the text section

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

THE UNIVERSITY OF NEW SOUTH WALES

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

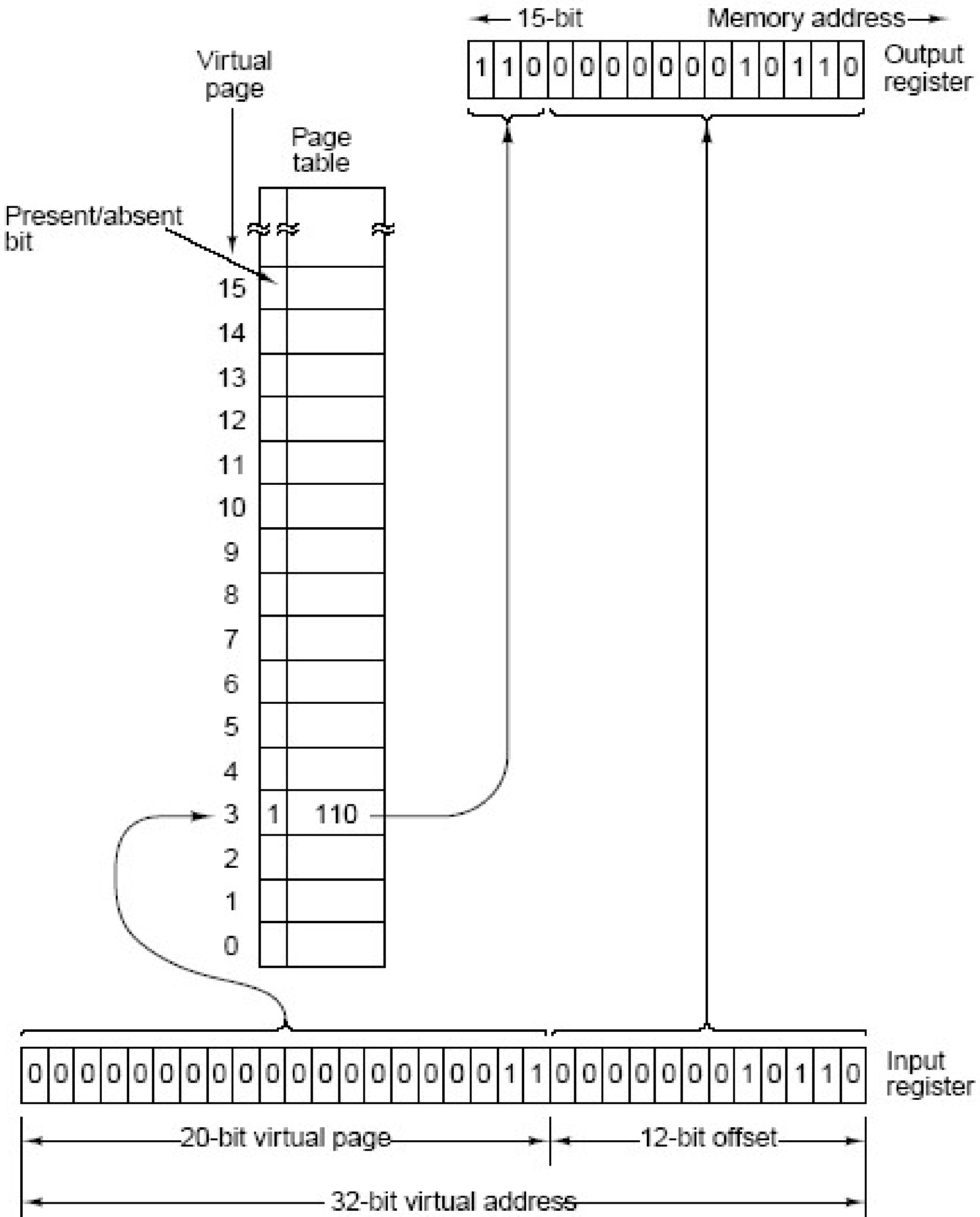
Address Translation

- 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

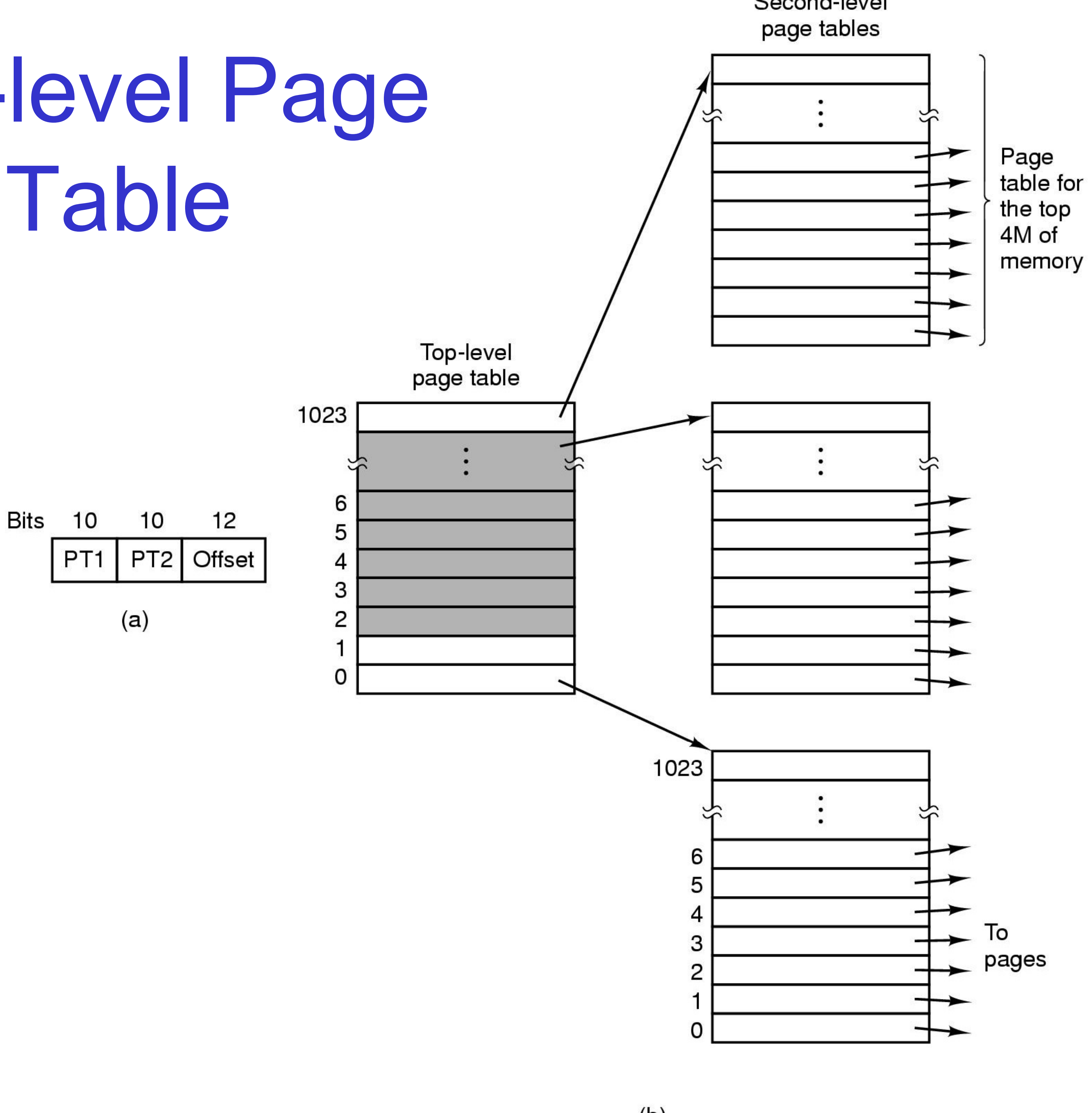
- Assume we have
  - 64-bit virtual address (humungous address space)
  - 4 KByte page size
  - How many page table entries do we need for one process?
- Problem:
  - Page table is very large
  - Access has to be fast, lookup for every memory reference
  - Where do we store the page table?
    - Registers?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

Page Tables

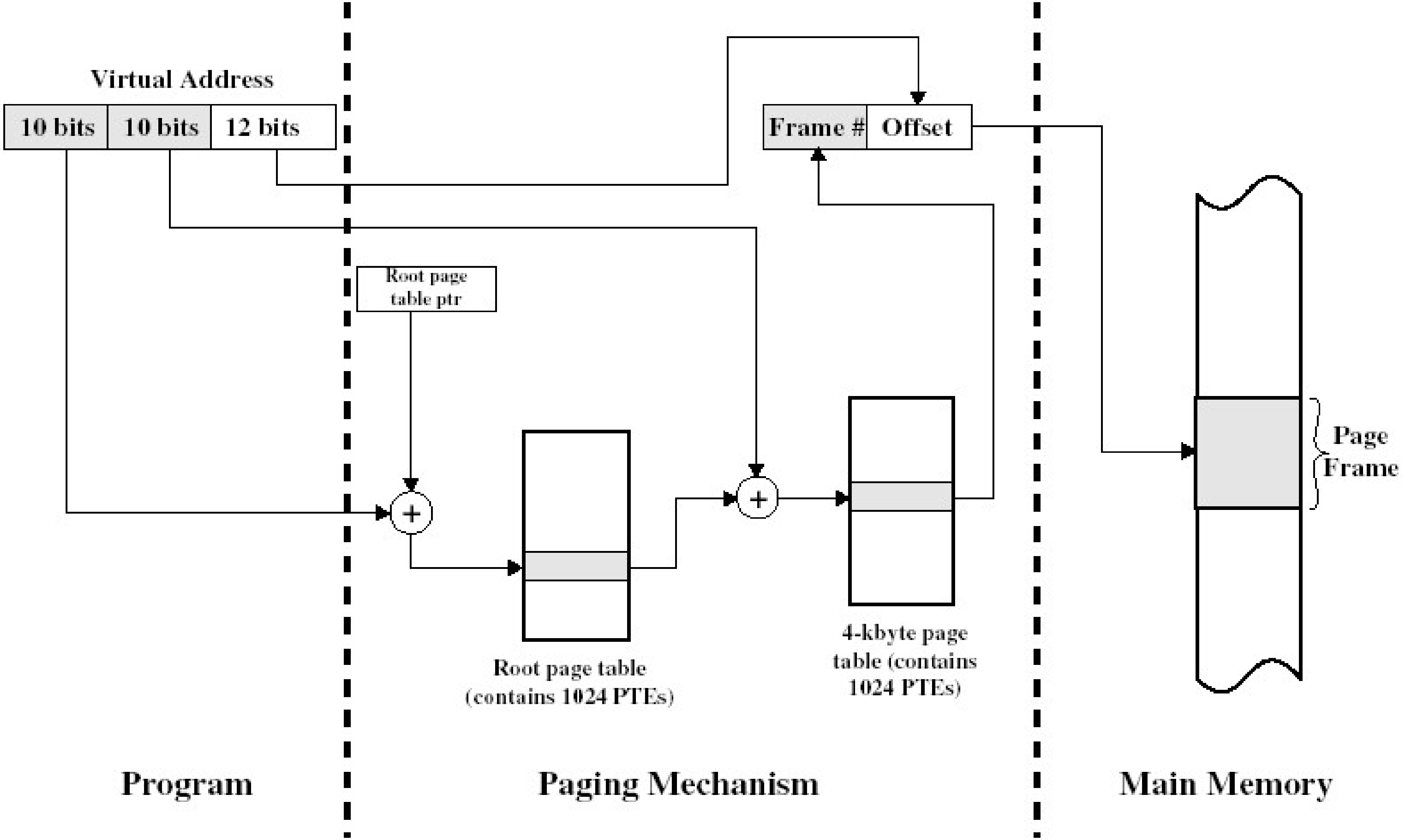
- Page tables are implemented as data structures in main memory
- Most processes do not use the full 4GB address space
  - e.g., 0.1 – 1 MB text, 0.1 – 10 MB data, 0.1 MB stack
- We need a compact representation that does not waste space
  - But is still very fast to search
- Three basic schemes
  - 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

- 2nd –level page tables representing unmapped pages are not allocated
  - Null in the top-level page table



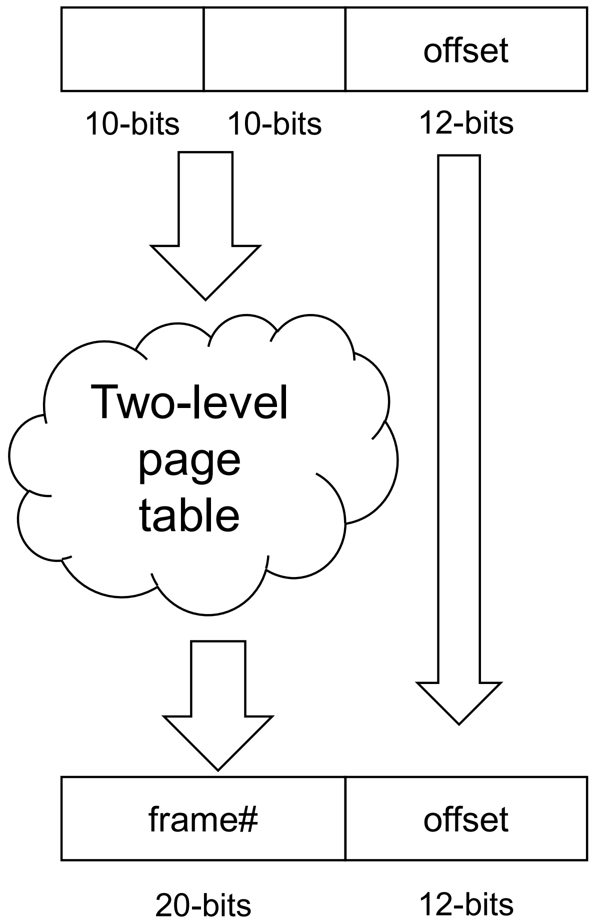
Two-level Translation



Example Translations

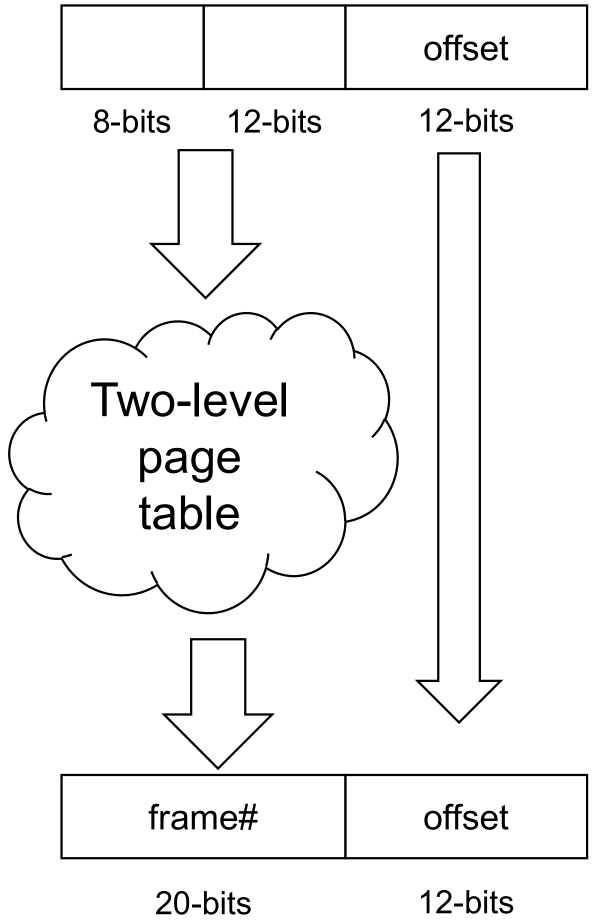
Summarising Two-level Page Tables

- 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



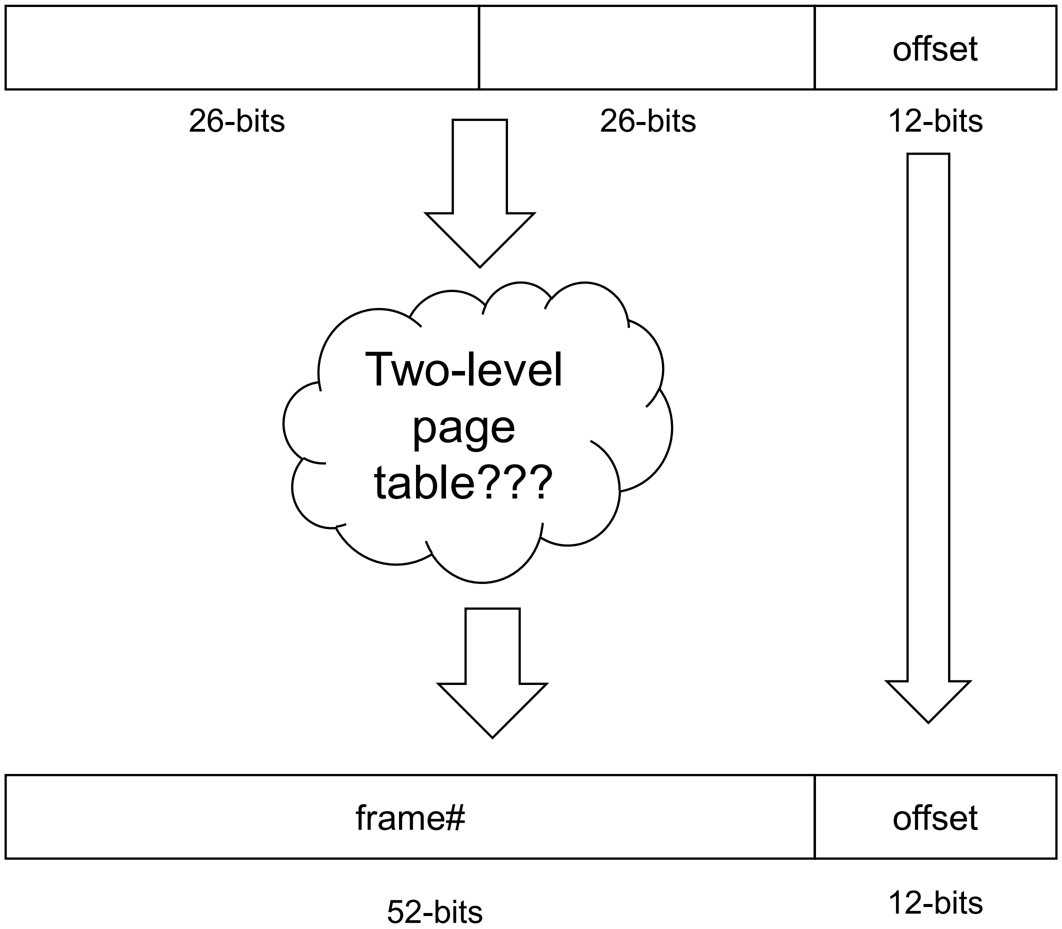
Index bits determine node sizes

- 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



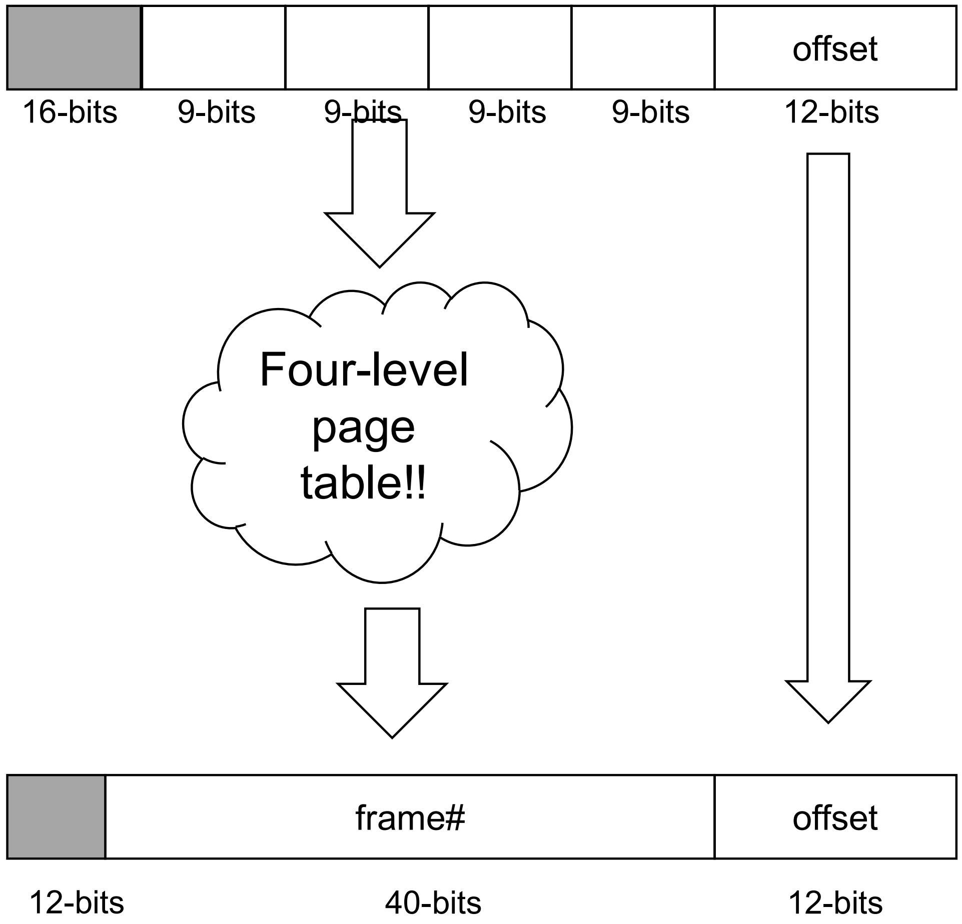
Supporting 64-bit Virtual to Physical Translation

- 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
    - $2^{26} * 8 = 512$ 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

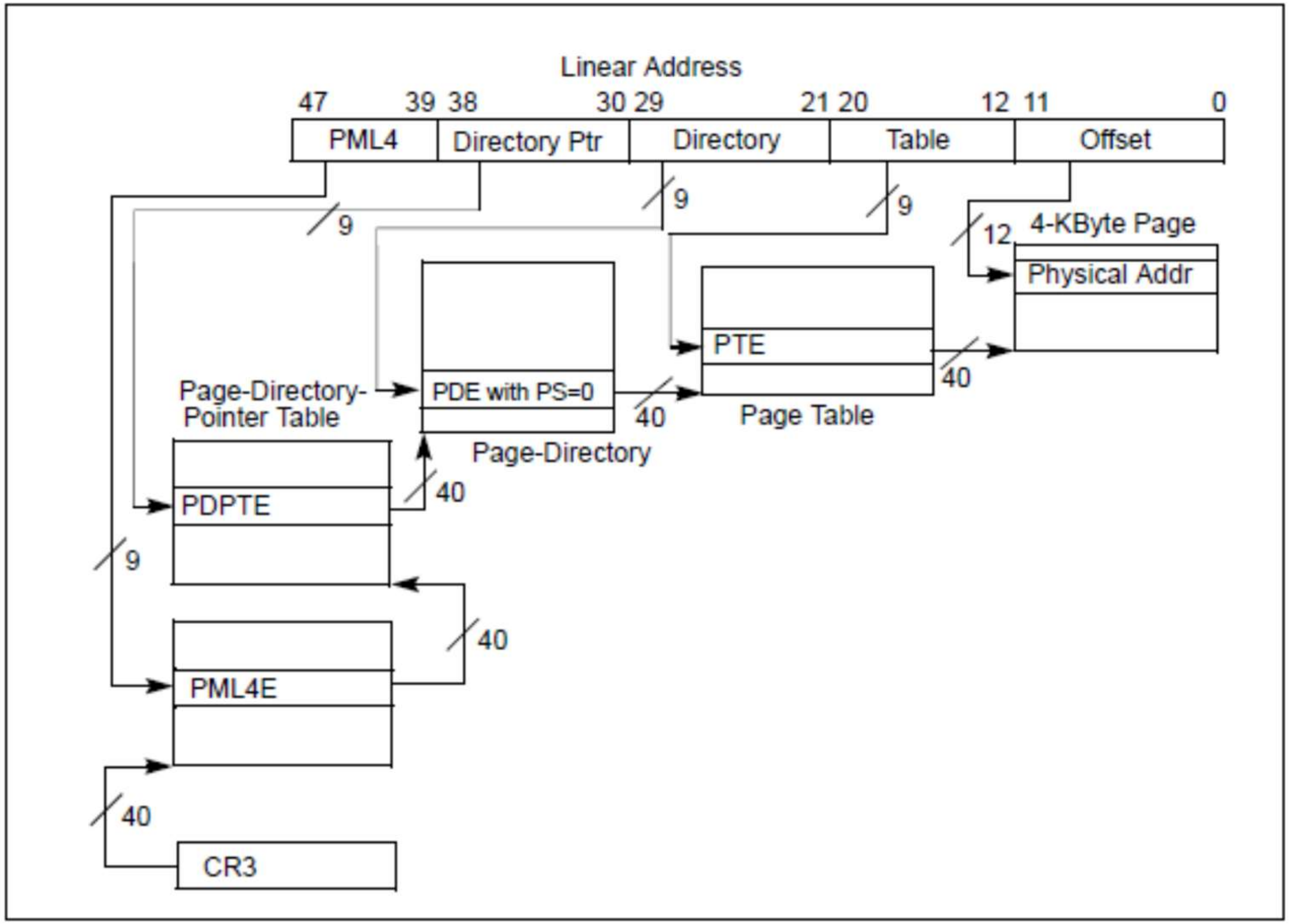
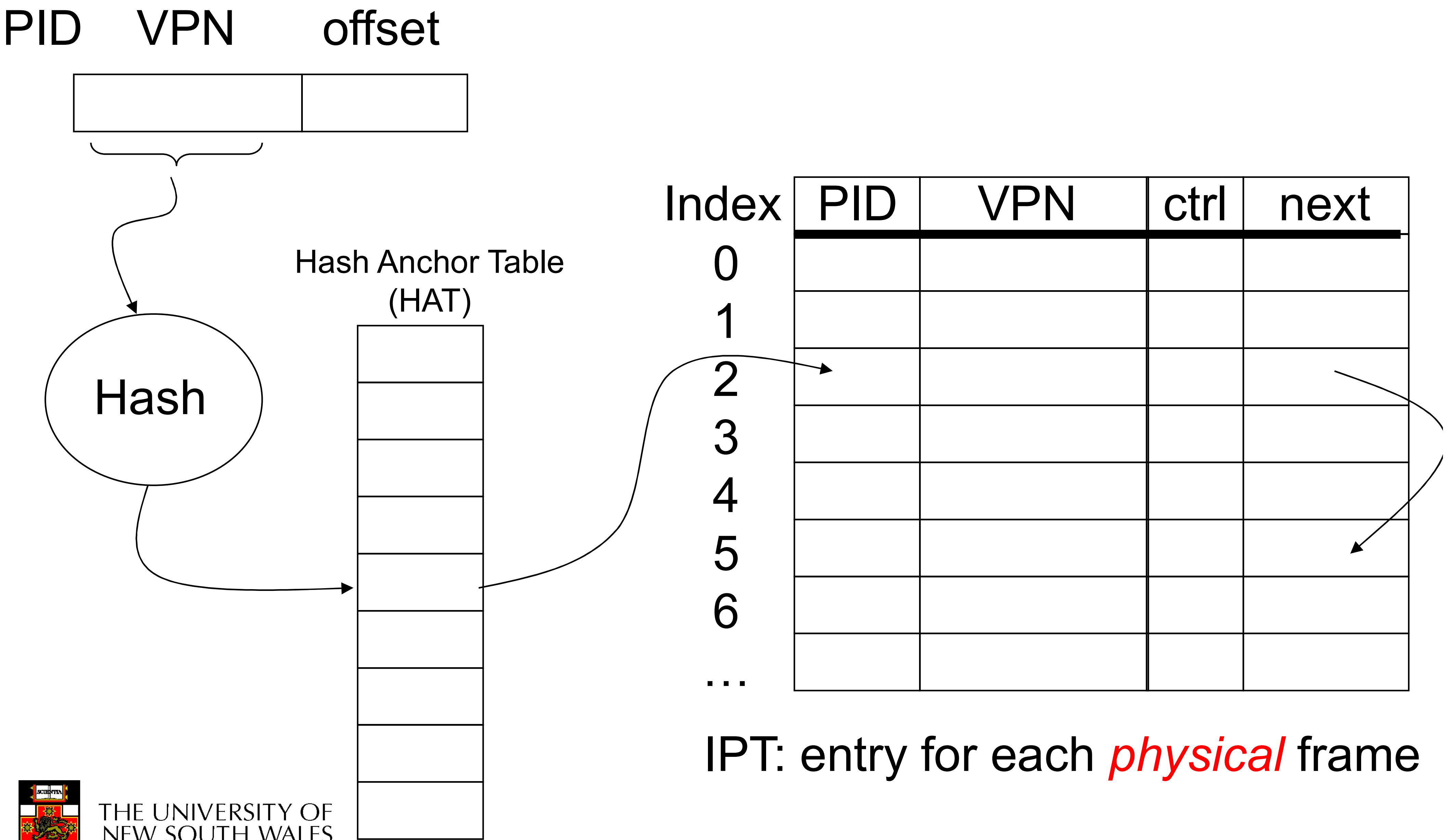


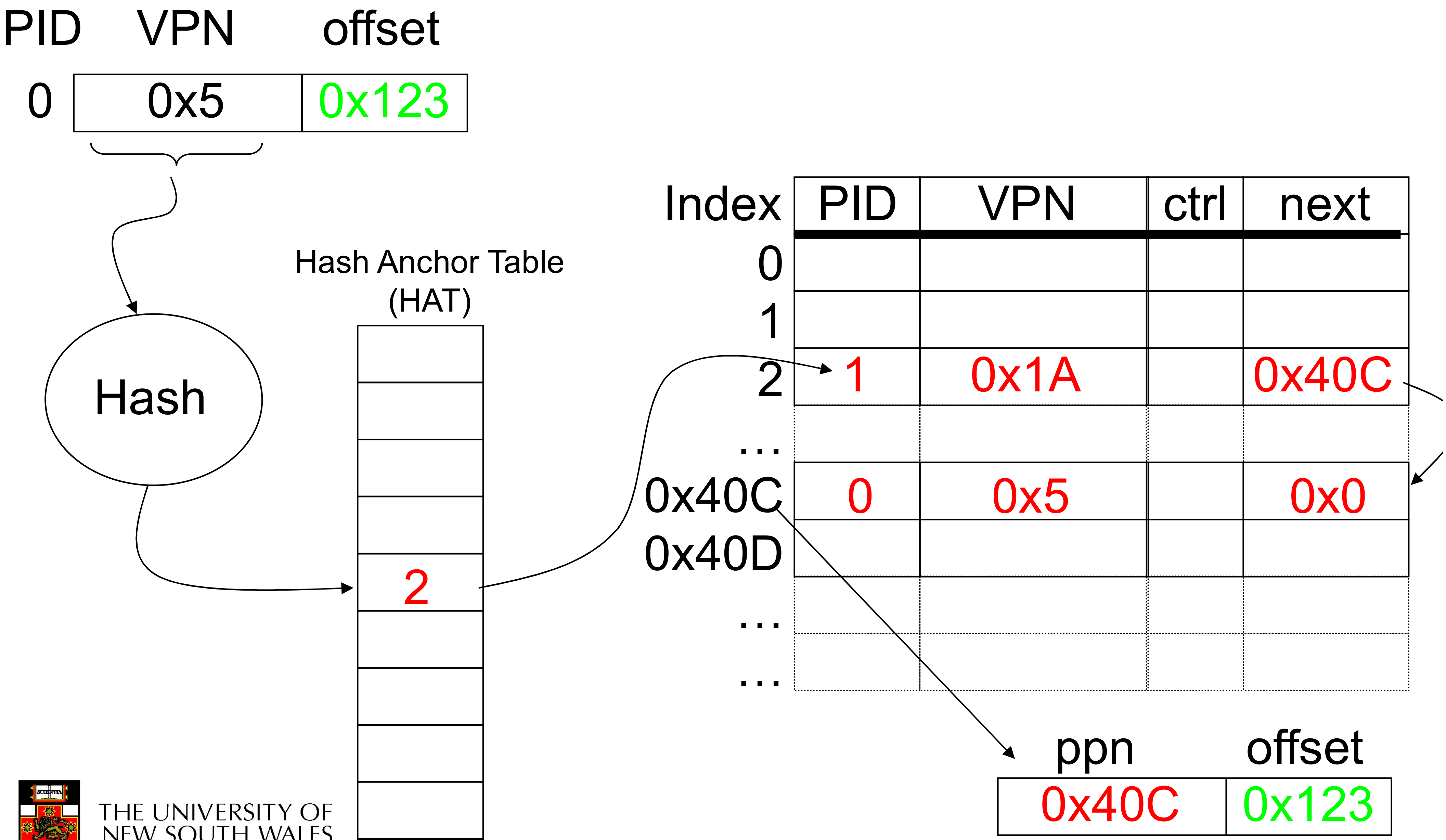
Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

Alternative: Inverted Page Table



IPT: entry for each *physical* frame

Alternative: Inverted Page Table



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?

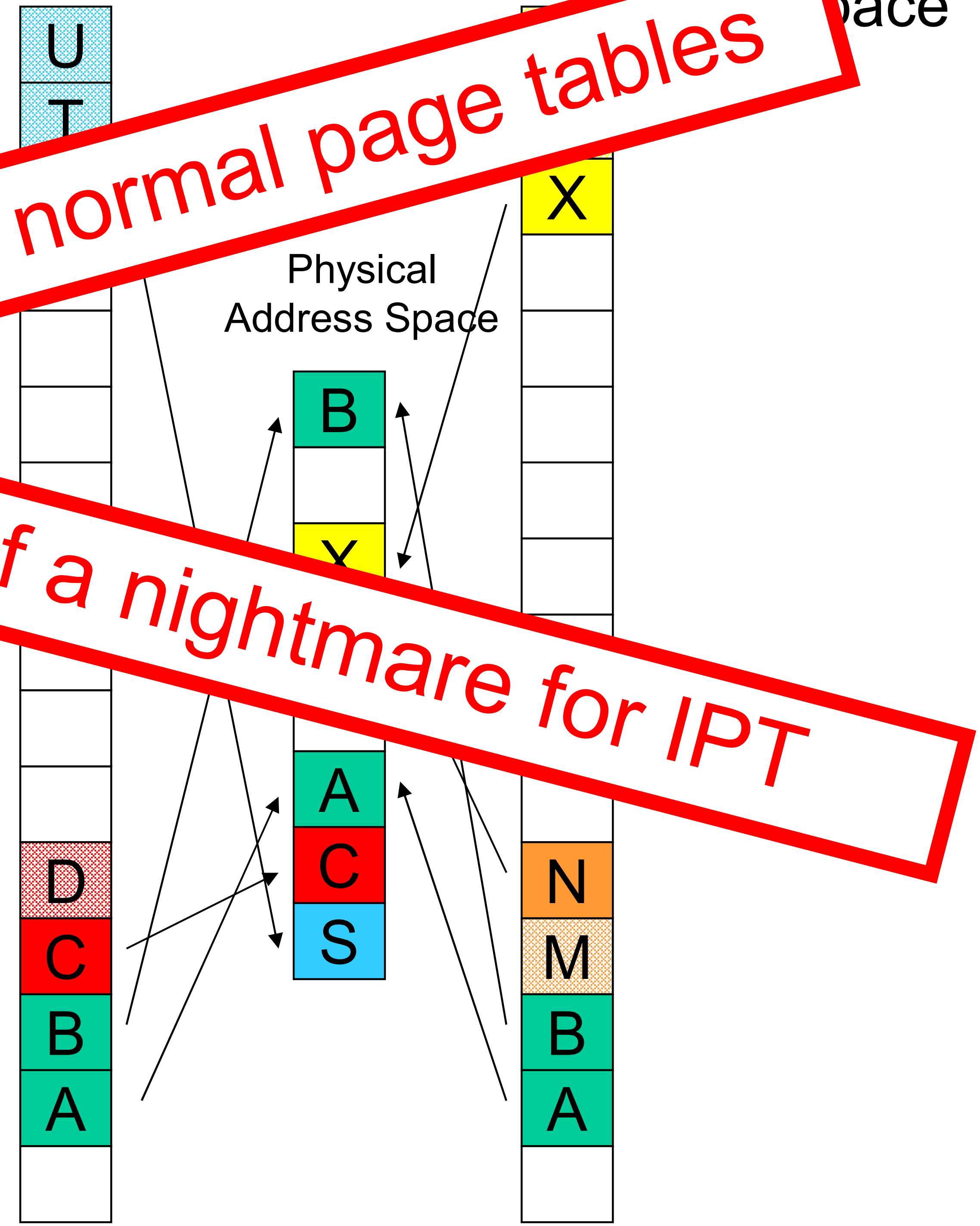
Another look at sharing…

Proc 1 Address Space

Proc 2 Address Space

‘easy’ for normal page tables

bit of a nightmare for IPT

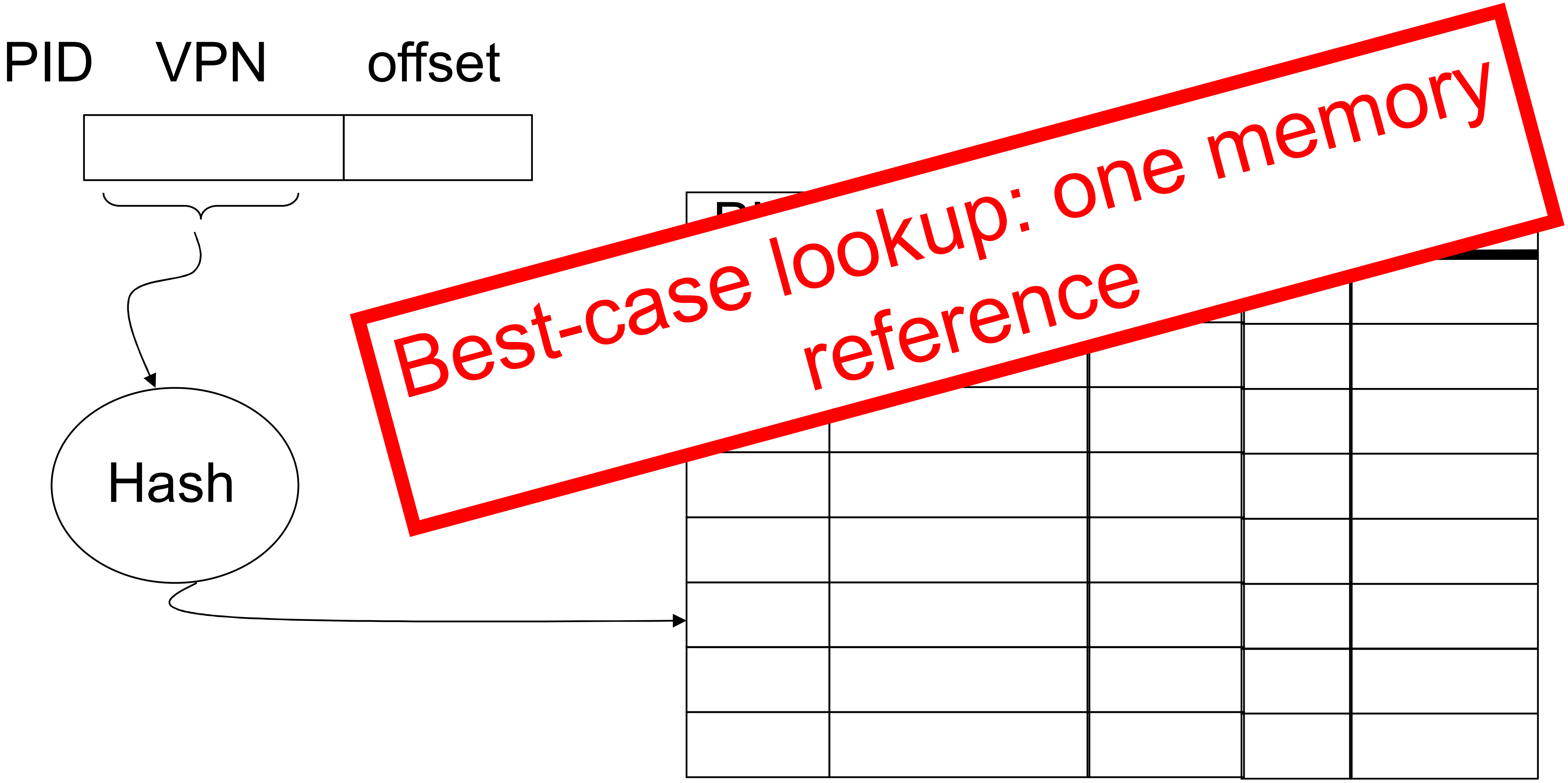


Two (or more) processes running the same program and sharing the text section

Improving the IPT: Hashed Page Table

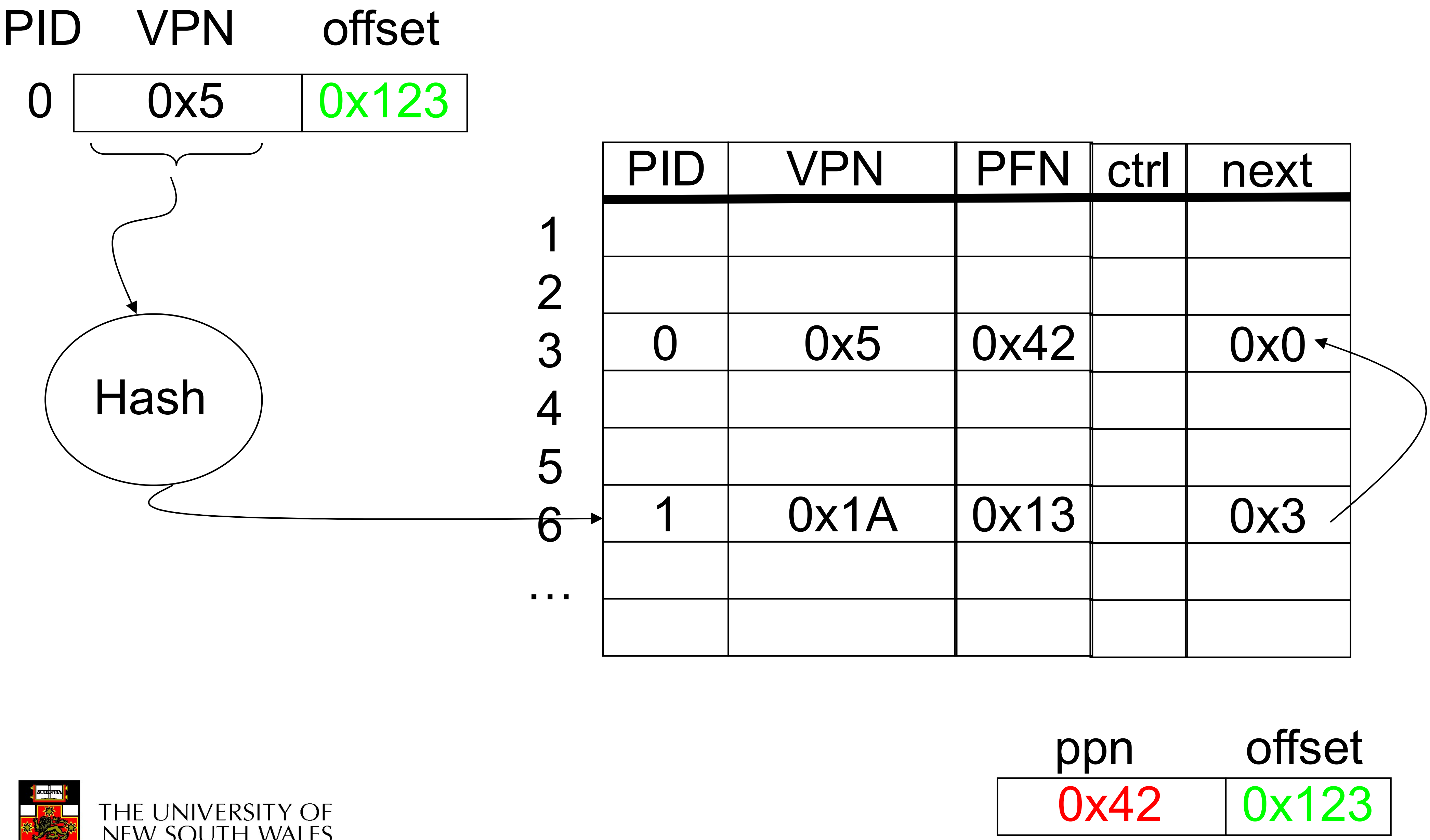
- Retain fast lookup of IPT
  - A single memory reference in best case
- Retain page table sized based on physical memory size (not virtual)
  - Enable efficient frame sharing
  - Support more than one mapping for same frame

Hashed Page Table

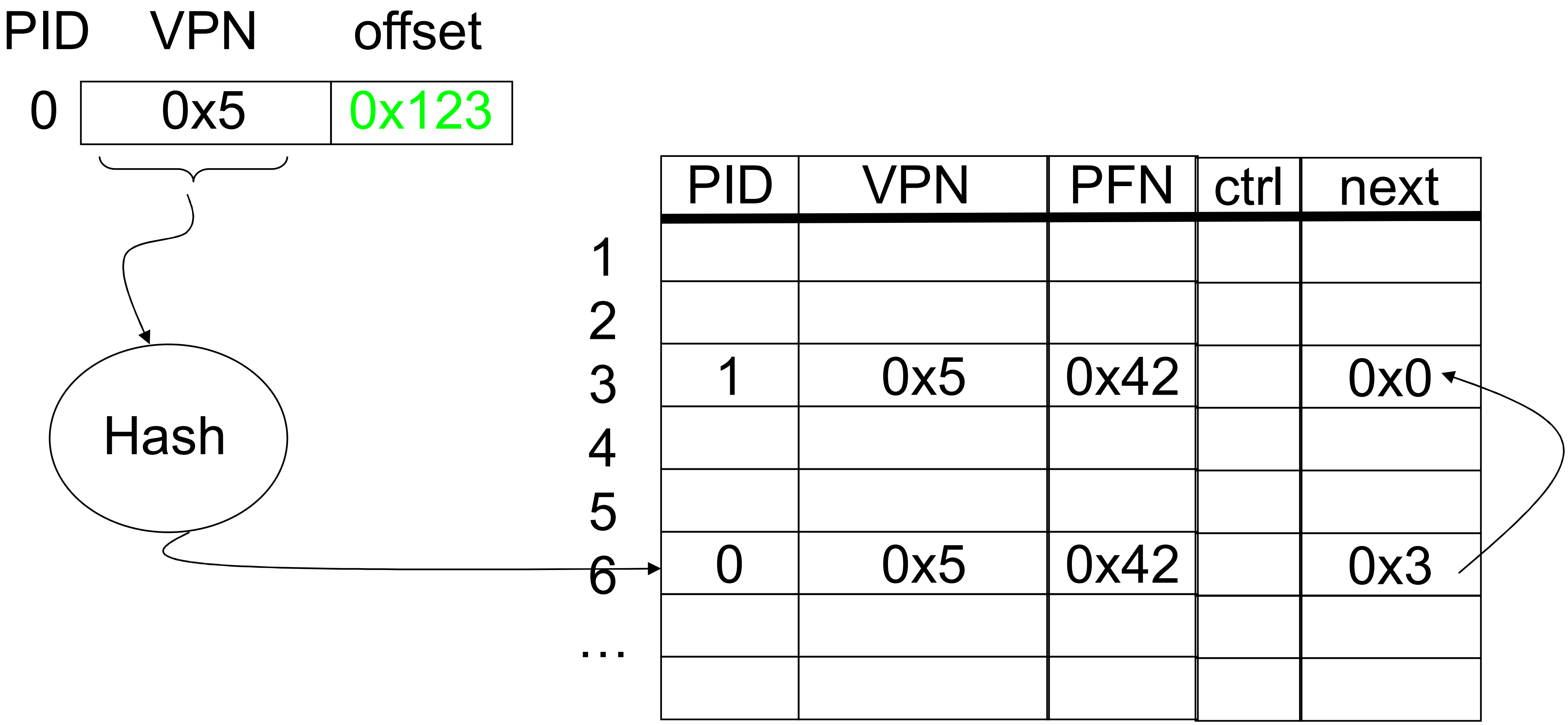


HPT: Frame number stored in table

Hashed Page Table



Sharing Example



ppn	offset
0x42	0x123

Sizing the Hashed Page Table

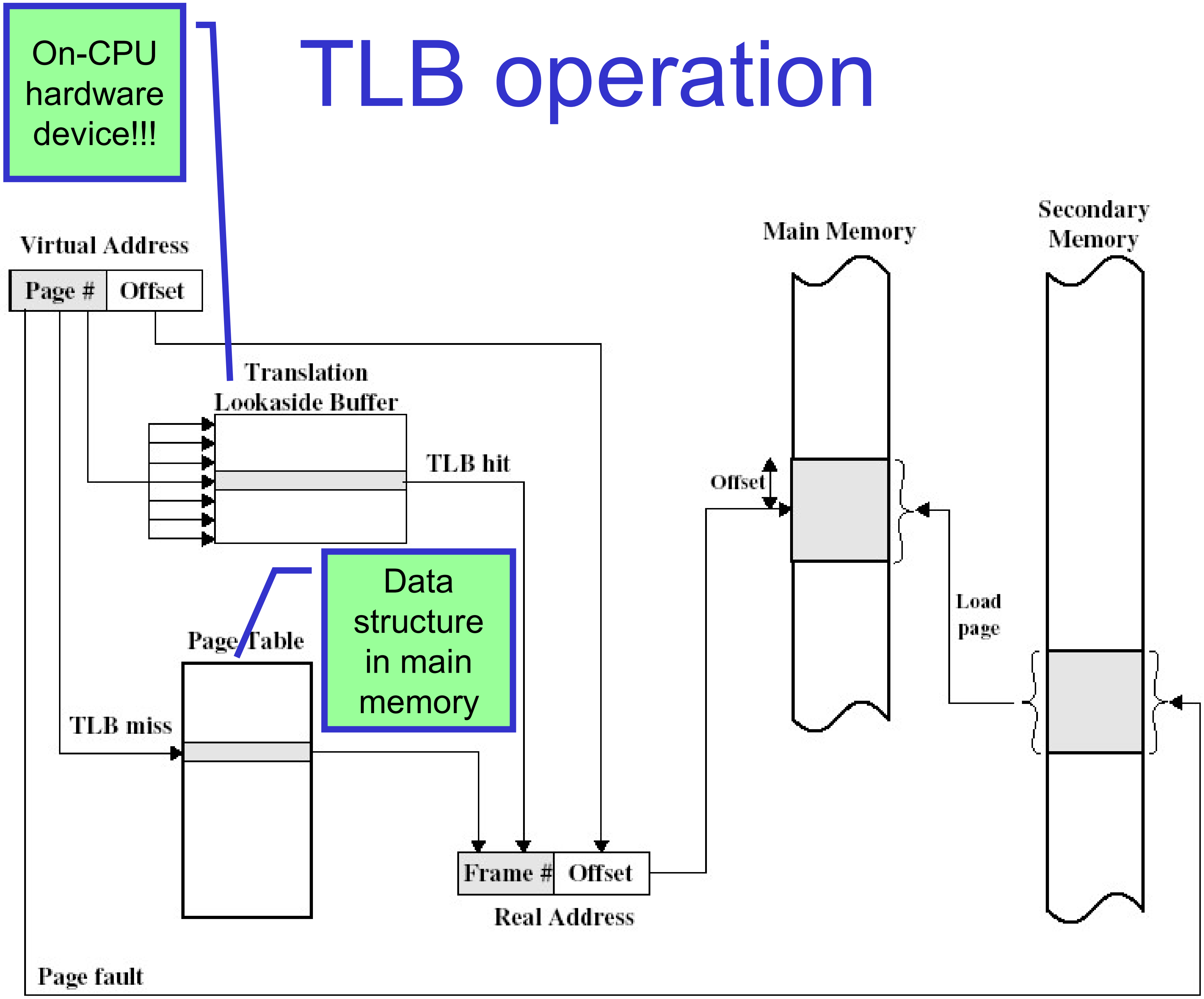
- HPT sized based on physical memory size
- With sharing
  - Each frame can have more than one PTE
  - More sharing increases number of slots used
    - Increases collision likelihood
- However, we can tune HPT size based on:
    - Physical memory size
    - Expected sharing
    - Hash collision avoidance.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

    $\Rightarrow$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)

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

$page\ \#$	$frame\ \#$	V	W
$\cdots$	$\cdots$	$\cdot$	$\cdot$
$\cdots$	$\cdots$	$\cdot$	$\cdot$

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

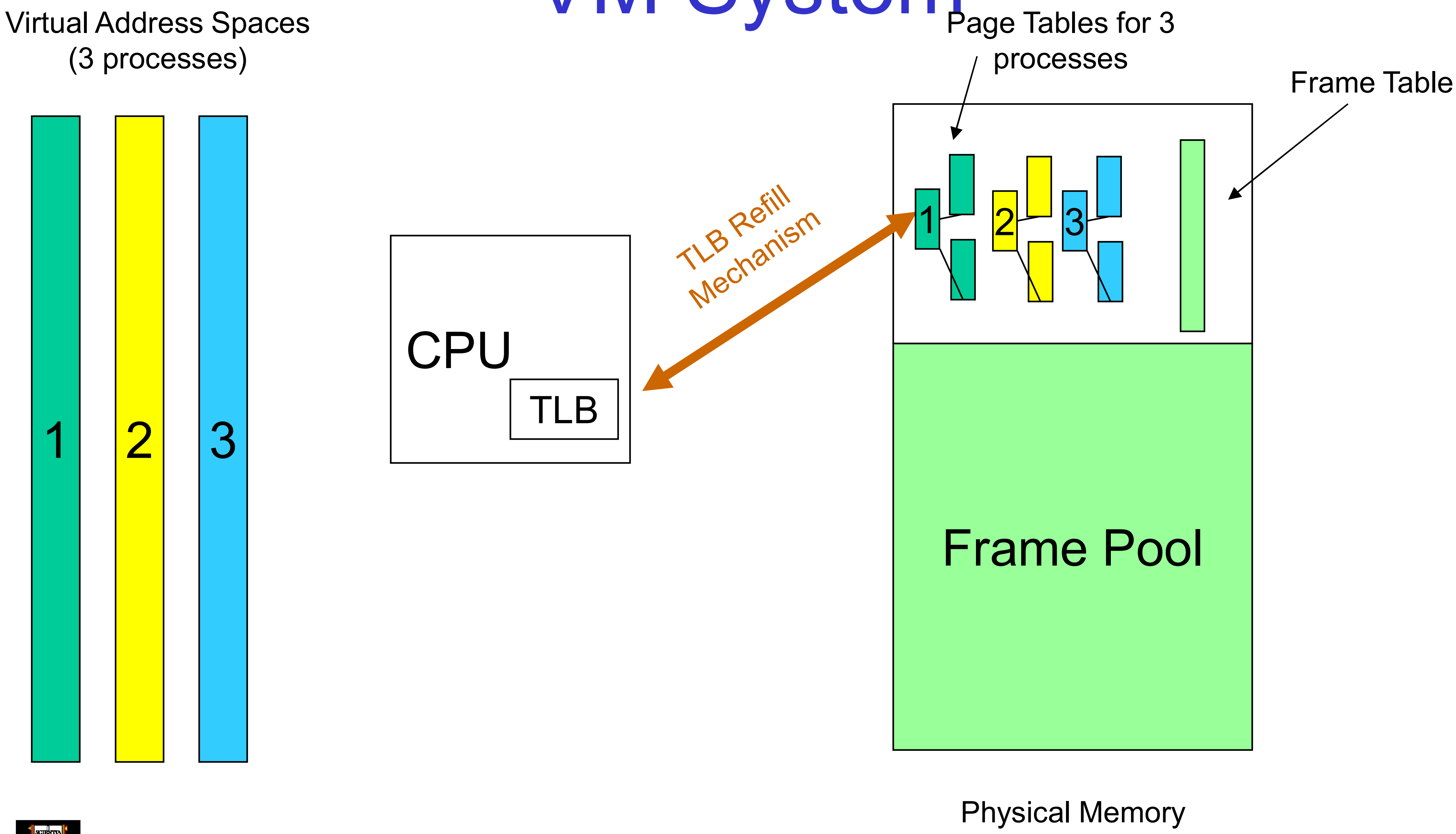
- Without TLB
  - Average number of physical memory references per virtual reference

    = 2
- With TLB (assume 99% hit ratio)
  - Average number of physical memory references per virtual reference

    = .99 * 1 + 0.01 * 2

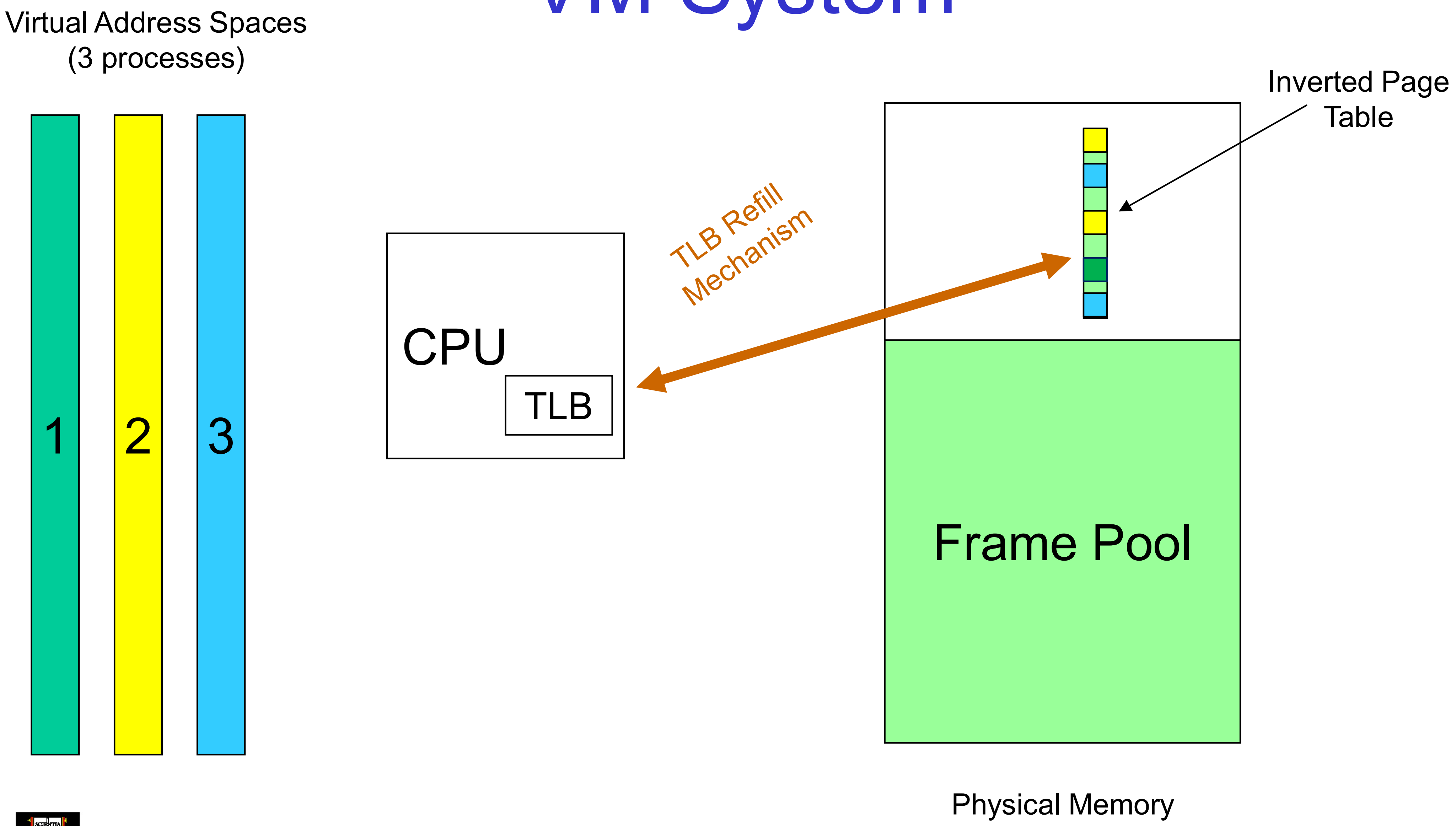
    = 1.01

Recap - Simplified Components of VM System

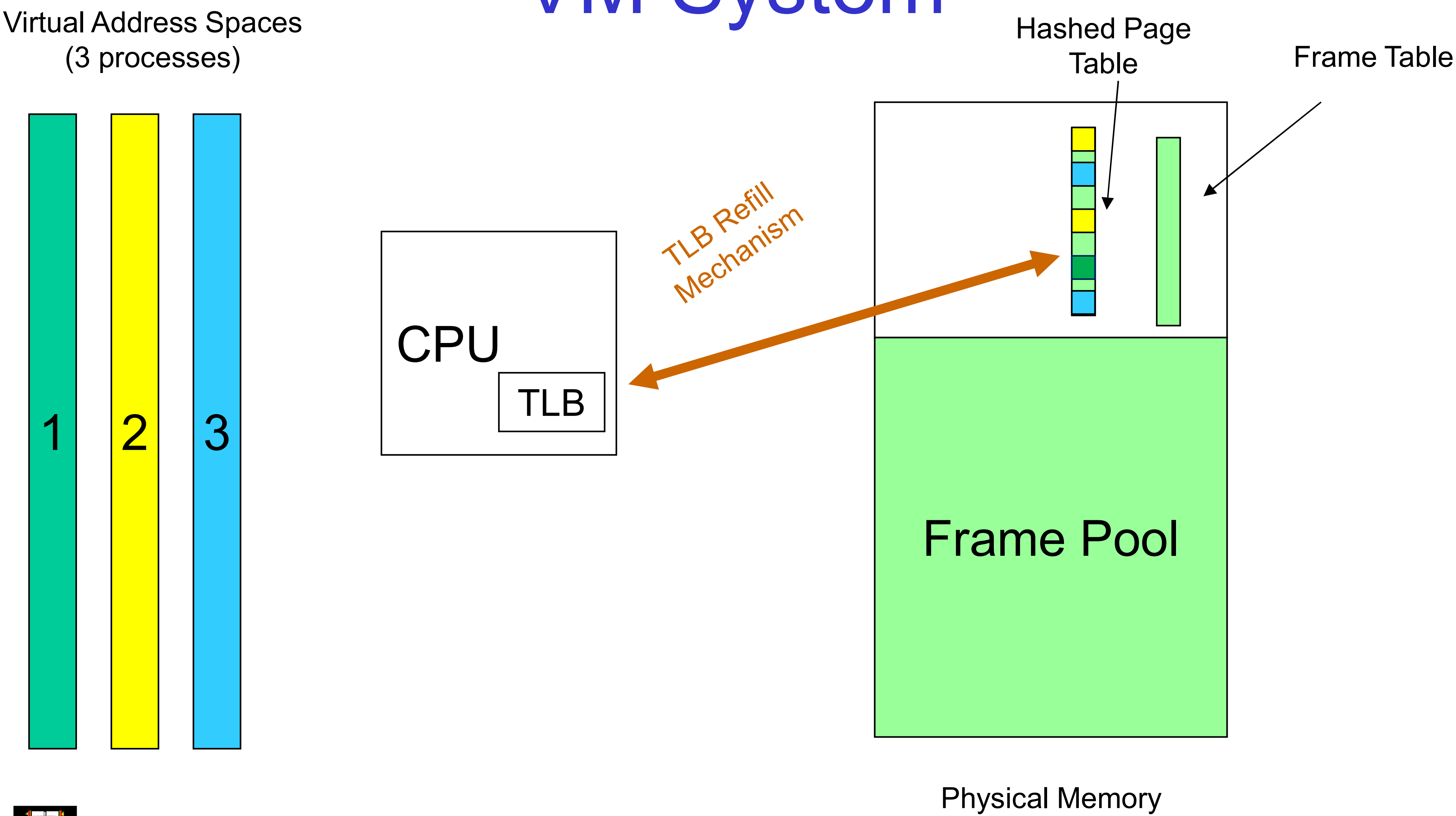


THE UNIVERSITY OF
NEW SOUTH WALES

Recap - Simplified Components of VM System



Recap - Simplified Components of VM System



MIPS R3000 TLB

31 12	11 6	5 0
VPN	ASID	0

EntryHi Register (TLB key fields)

31 12	11	10	9	8	7 0
PFN	N	D	V	G	0

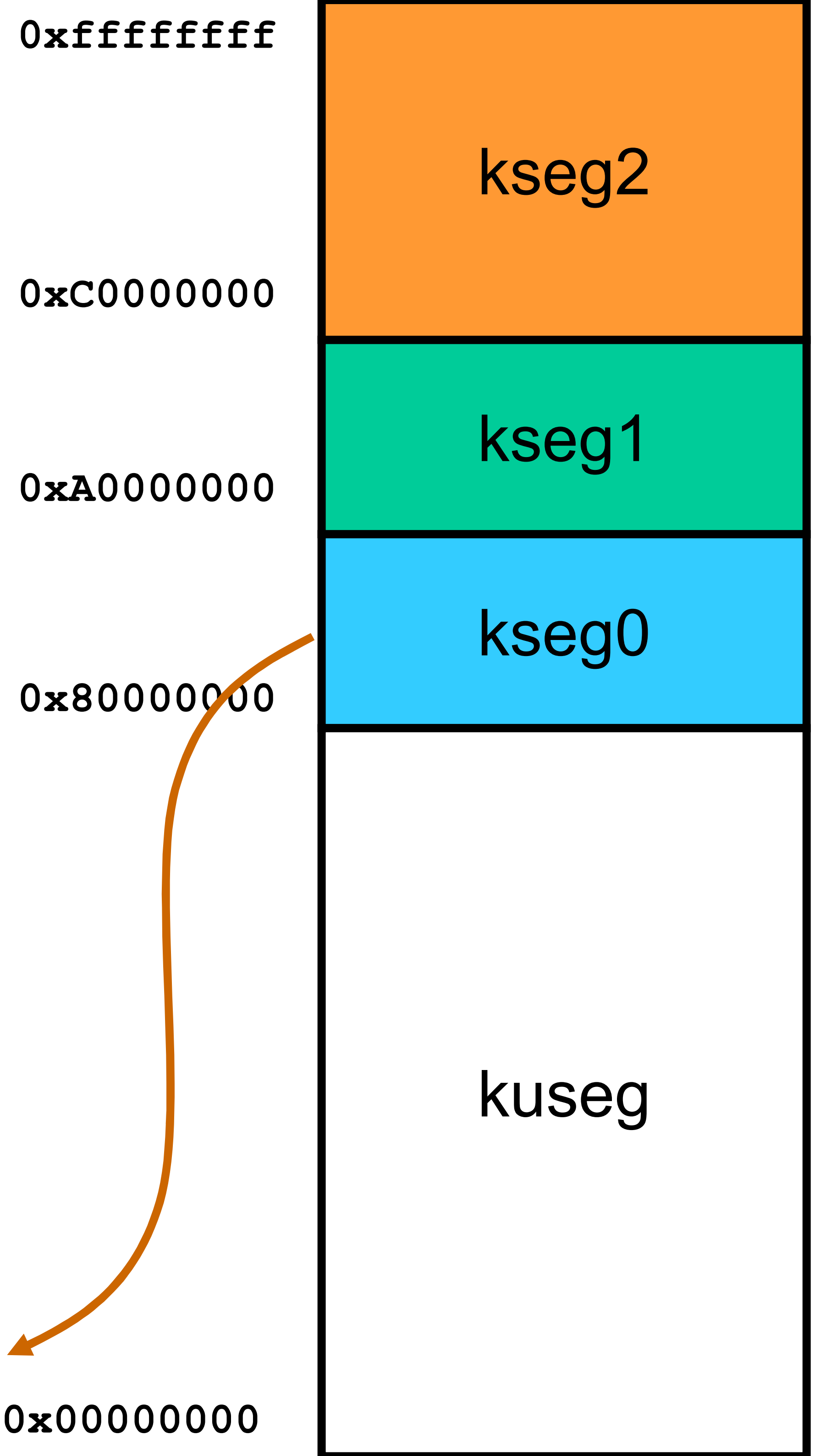
EntryLo Register (TLB data fields)

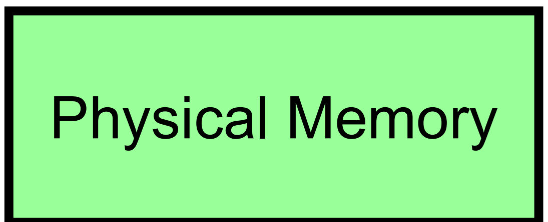
- N = Not cacheable
- D = Dirty = Write protect
- G = Global (ignore ASID in lookup)

- V = valid bit
- 64 TLB entries
- Accessed via software through Cooprocessor 0 registers
  - EntryHi and EntryLo

R3000 Address Space Layout

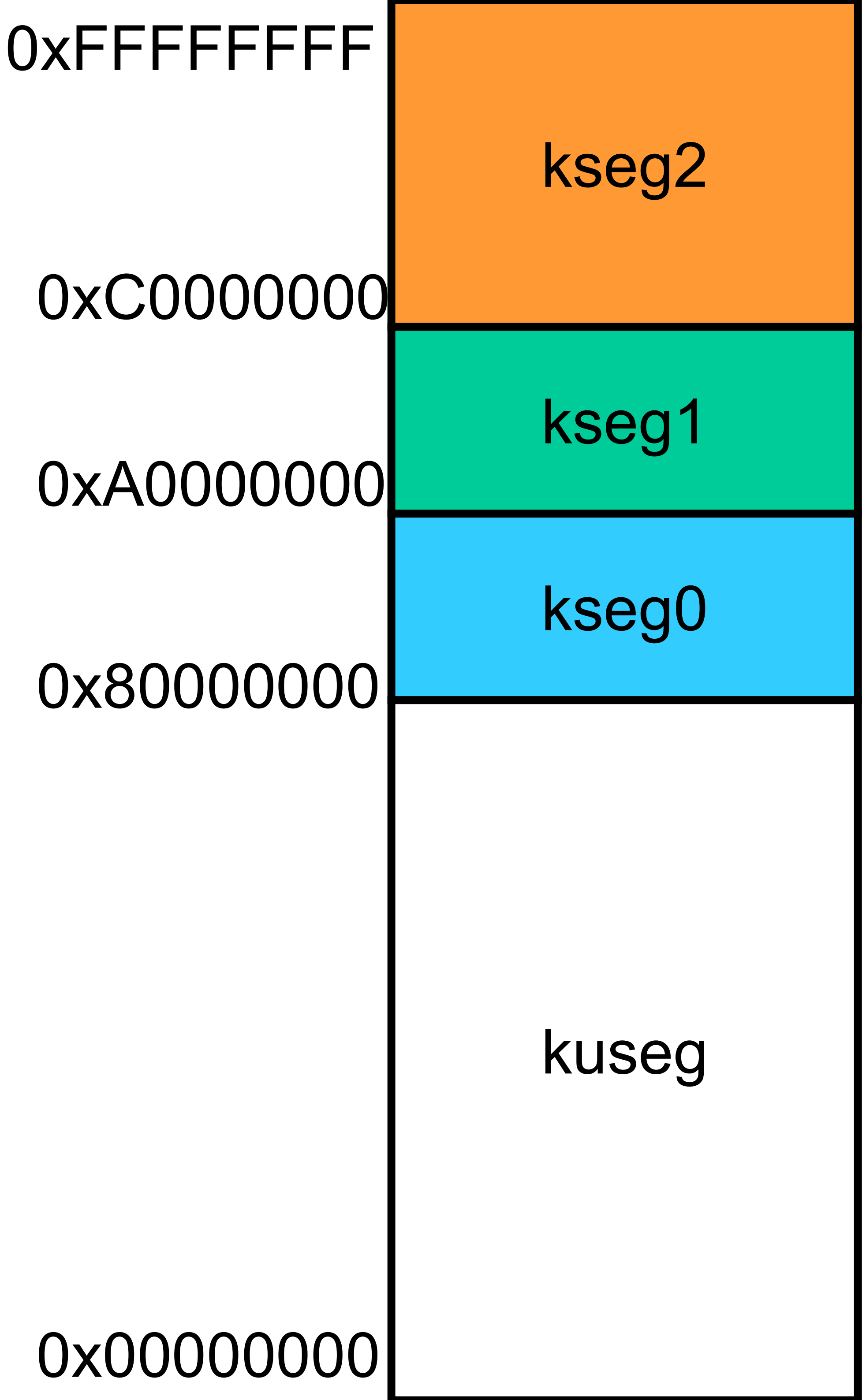
- kseg0:
  - 512 megabytes
  - Fixed translation window to physical memory
    - 0x80000000 - 0x9fffffff virtual = 0x00000000 - 0x1fffffff physical
    - TLB not used
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code and data is placed





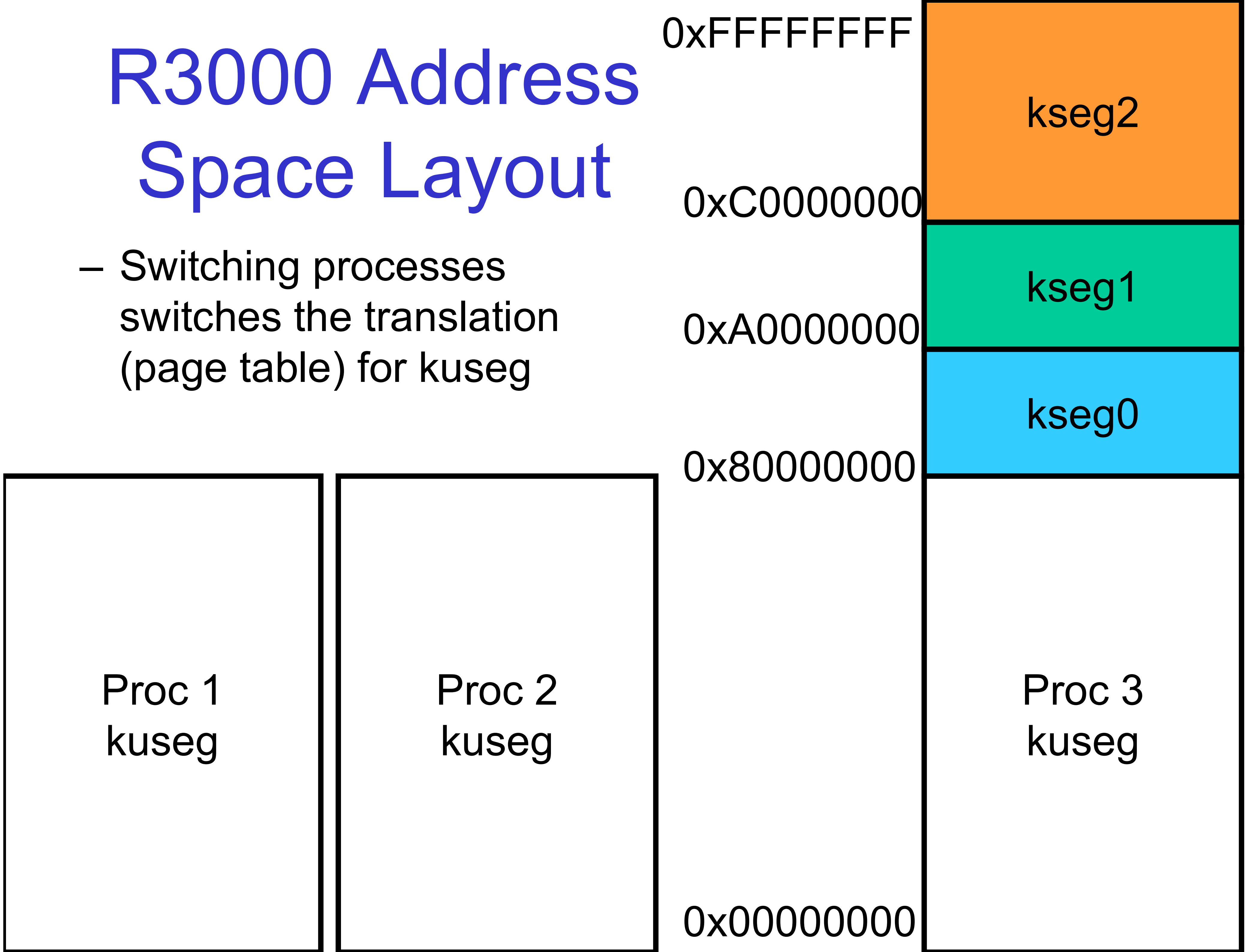
R3000 Address Space Layout

- 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)