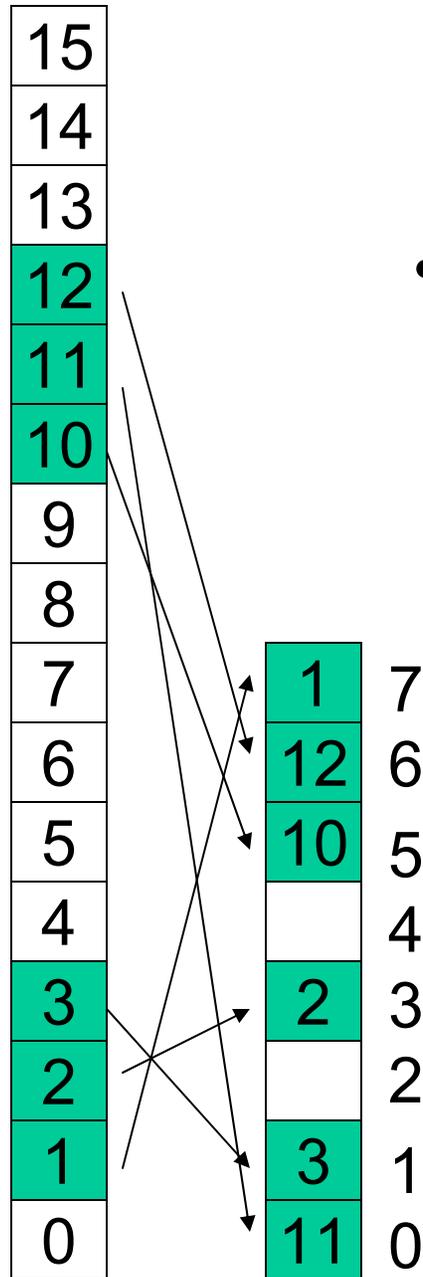


Virtual Memory



Virtual Address Space

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

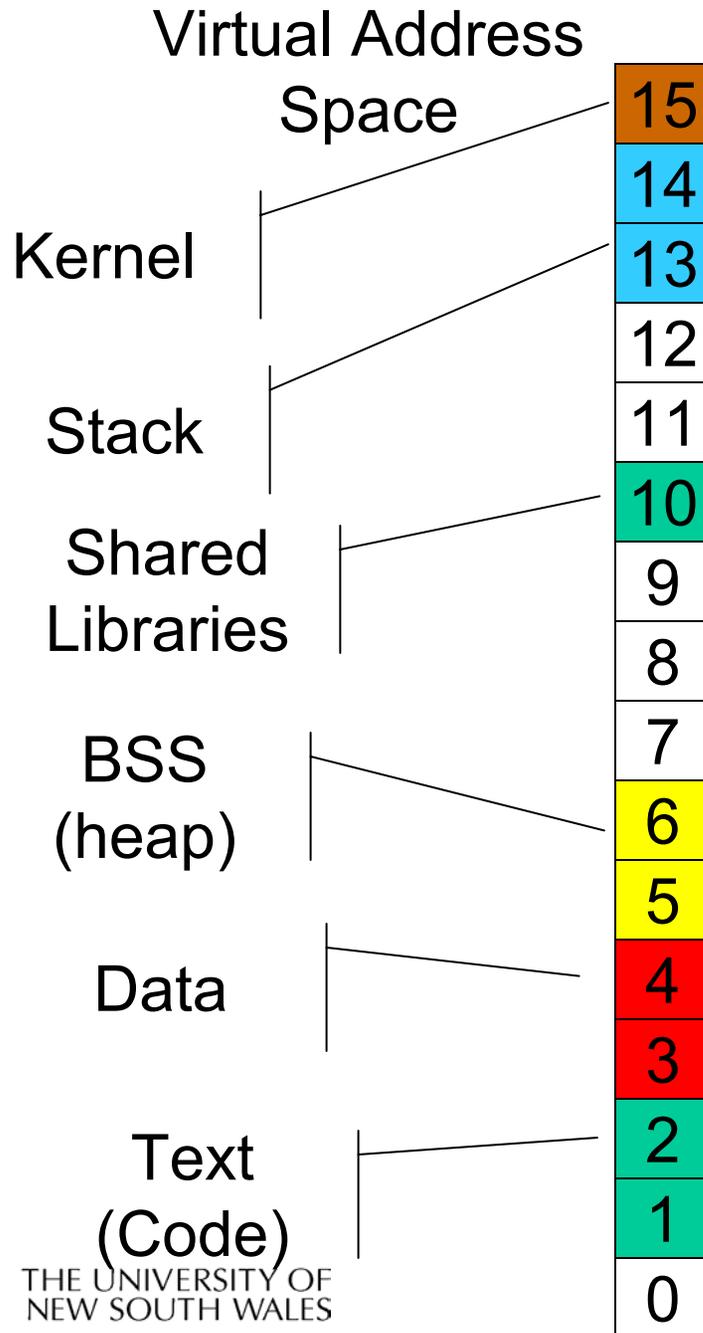


Paging

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

Physical
Address Space

Typical Address Space Layout

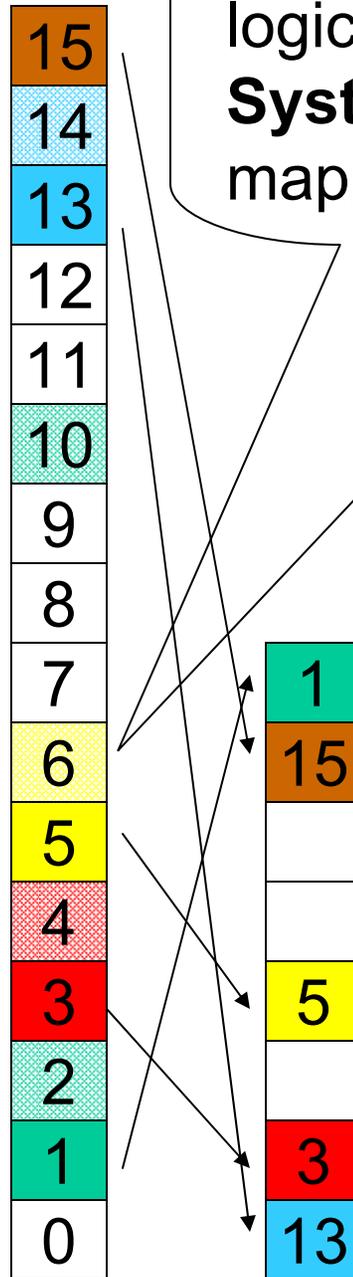


- Stack region is at top, and can grow down
- Heap has free space to grow up
- Text is typically read-only
- Kernel is in a reserved, protected, shared region
- 0-th page typically not used, why?



Virtual Address Space

- A process may be only partially resident
 - Allows OS to swap individual pages to 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

			4
14		6	
	2		
		10	

Disk

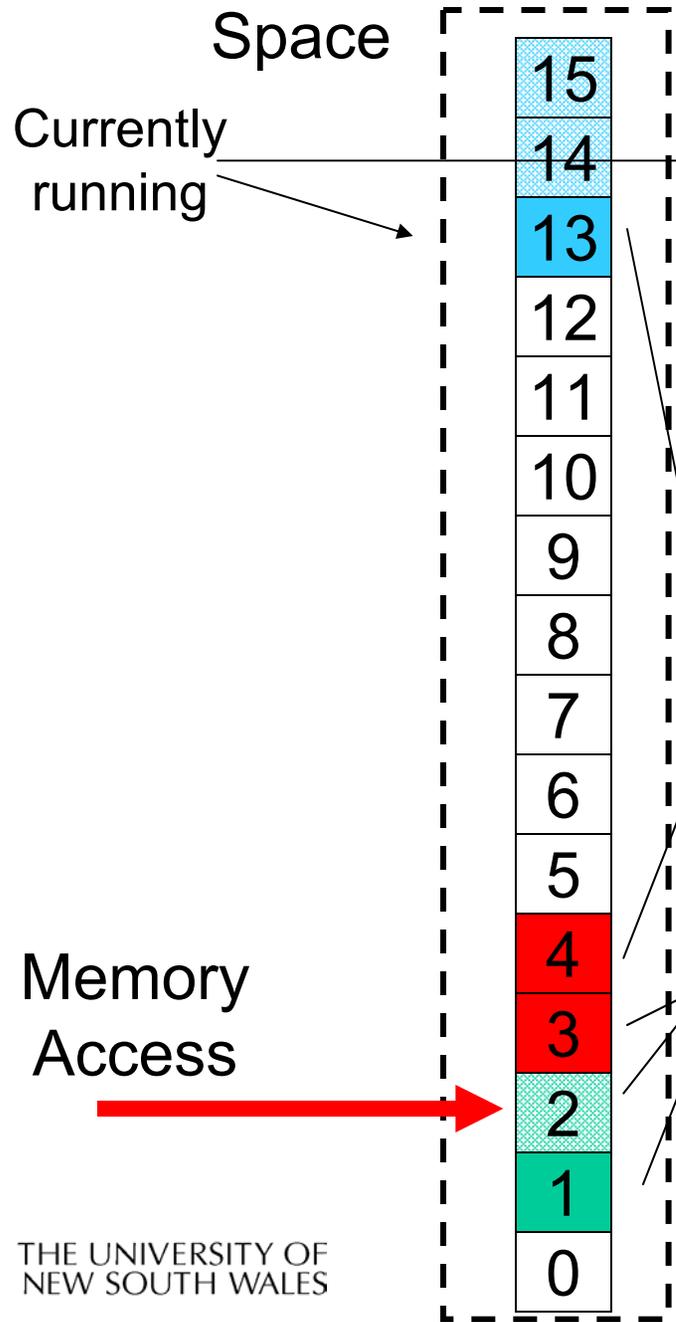
Physical Address Space

Page Faults

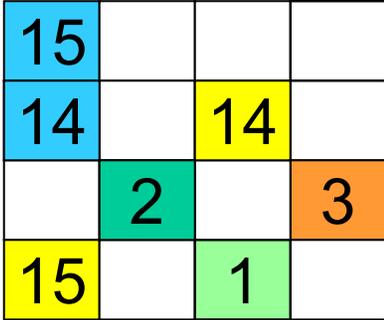
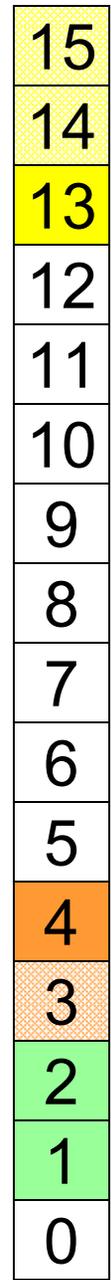
- Referencing an invalid page triggers a page fault
 - An exception handled by the OS
- Broadly, two standard page fault types
 - Illegal Address (protection error)
 - Signal or kill the process
 - Page not resident
 - Get an empty frame
 - Load page from disk
 - Update page (translation) table (enter frame #, set valid bit, etc.)
 - Restart the faulting instruction
- Note: Some implementations store disk block numbers of non-resident pages in the page table (with valid bit ***Unset***)



Proc 1 Address Space



Proc 2 Address Space

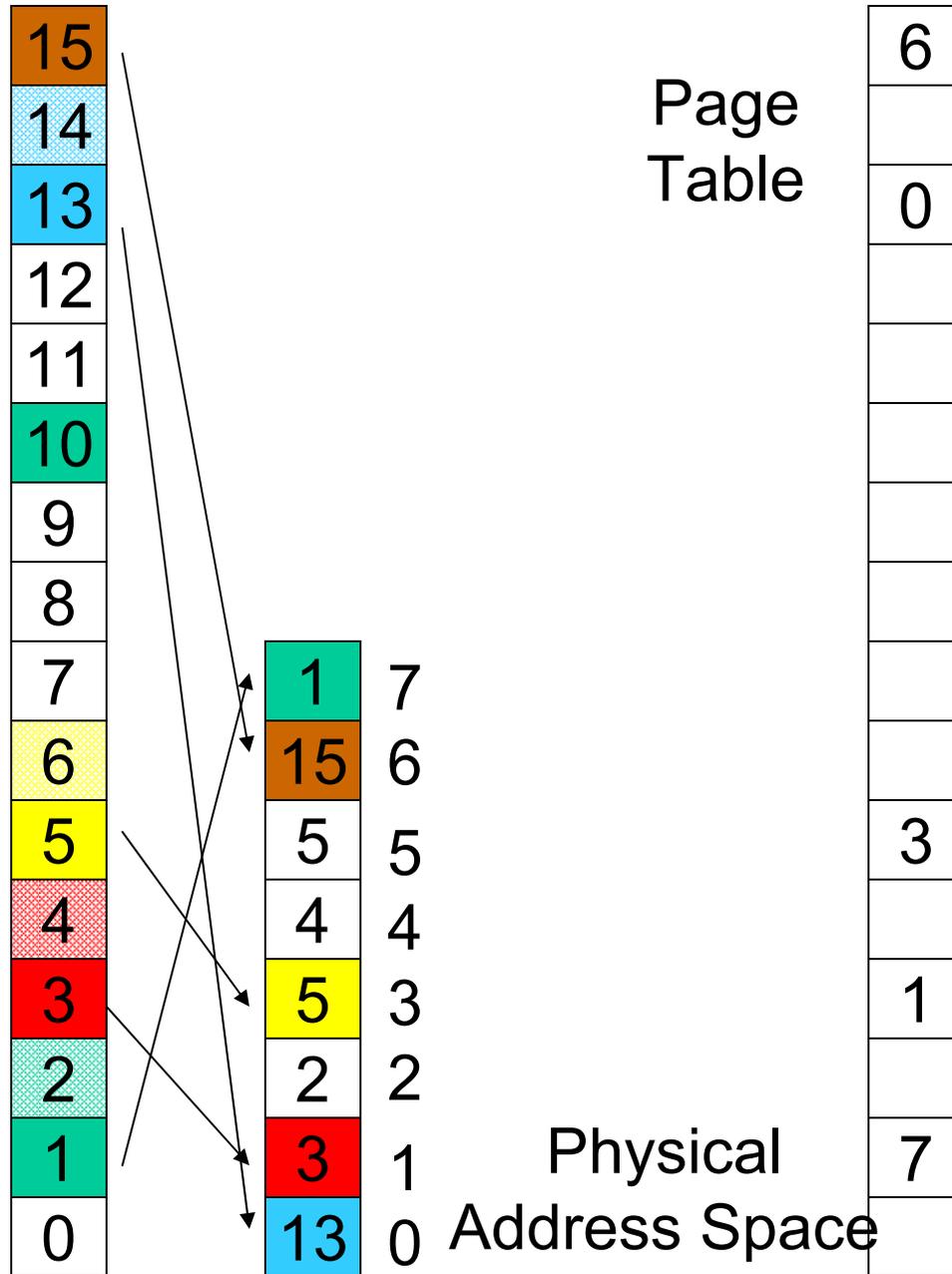


Disk



Virtual Address Space

- Page table for resident part of address space



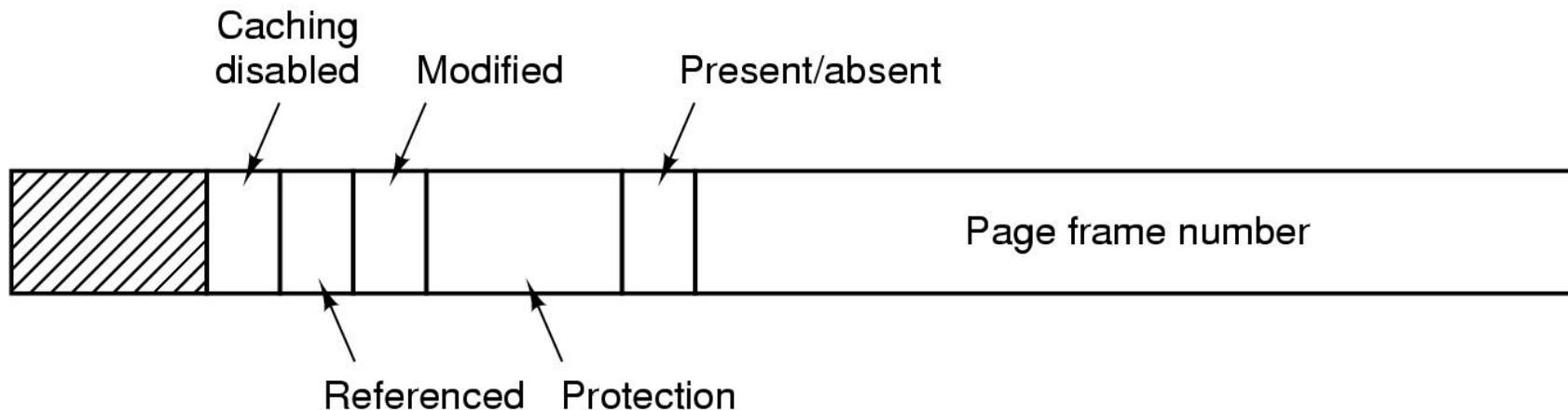
Shared Pages

- 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 be “pure” (*re-entrant*), i.e. not self modifying
 - Code must appear at same address in all processes



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

5
4
7
2

10



PTE 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



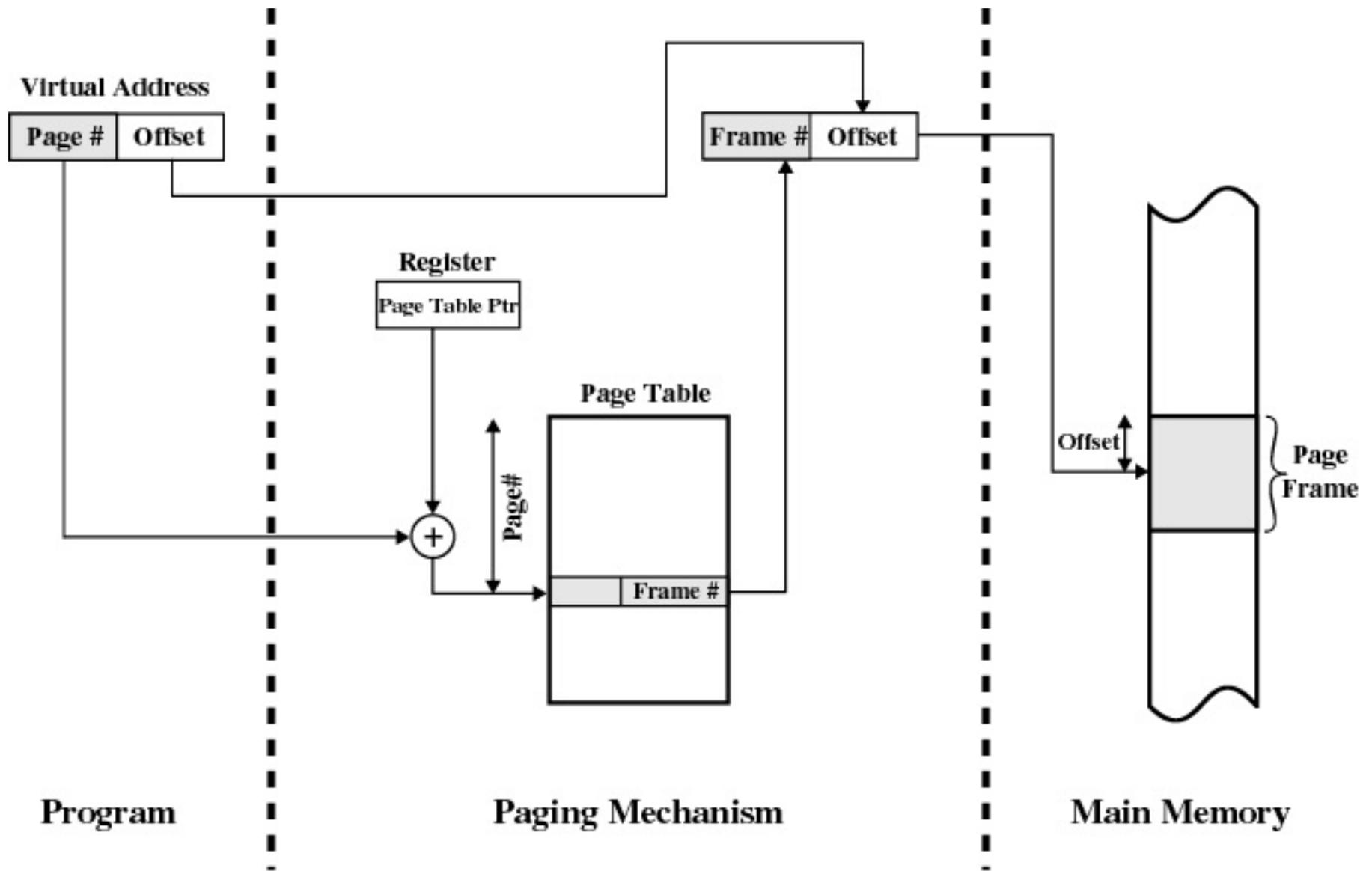


Figure 8.3 Address Translation in a Paging System

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?
- 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?



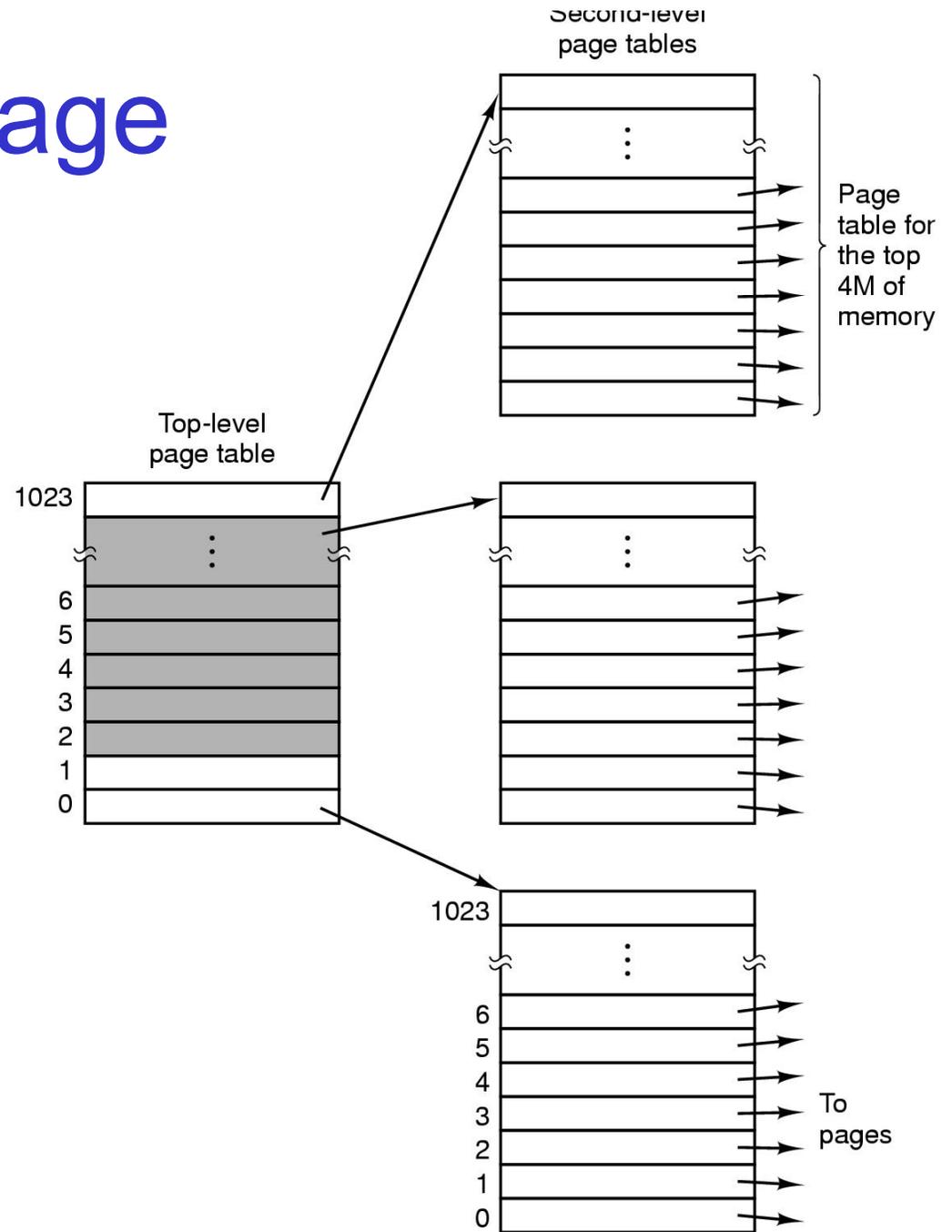
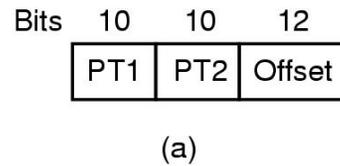
Page Tables

- 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)

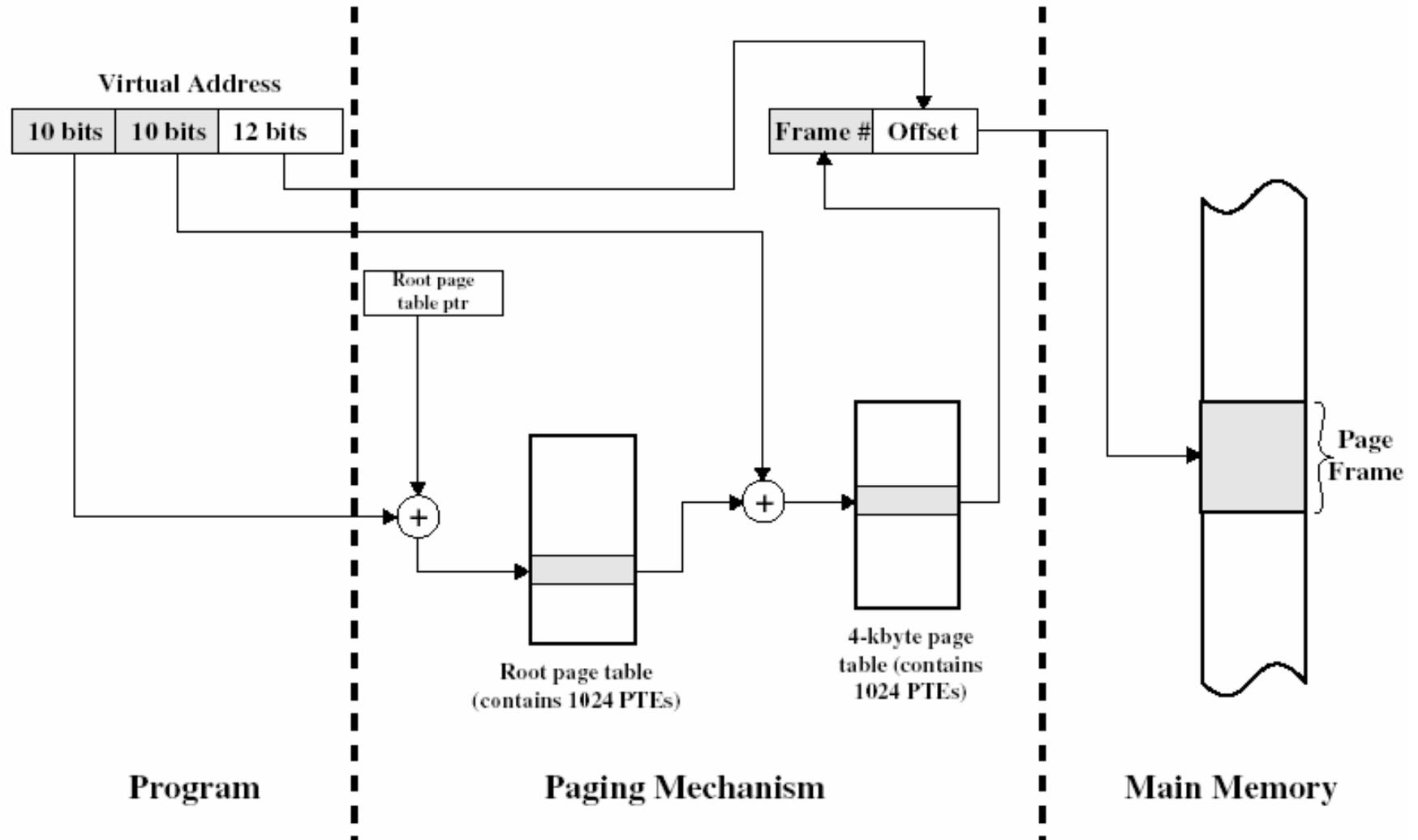


Two-level Page Table

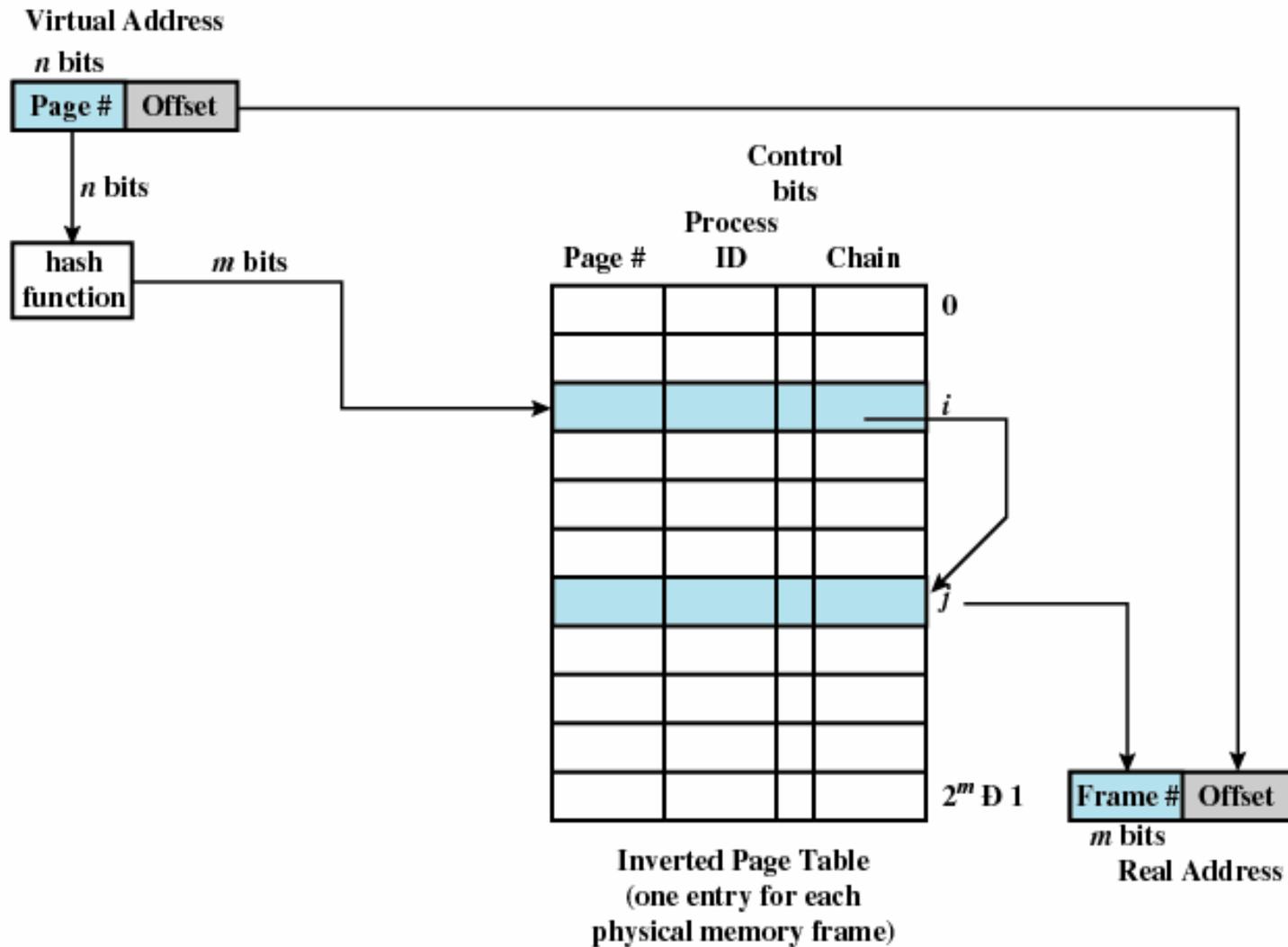
- 2nd -level page tables representing unmapped pages are not allocated
 - Null in the top-level page table



Two-level Translation



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
 - Use this to index into inverted page table
 - Match the 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

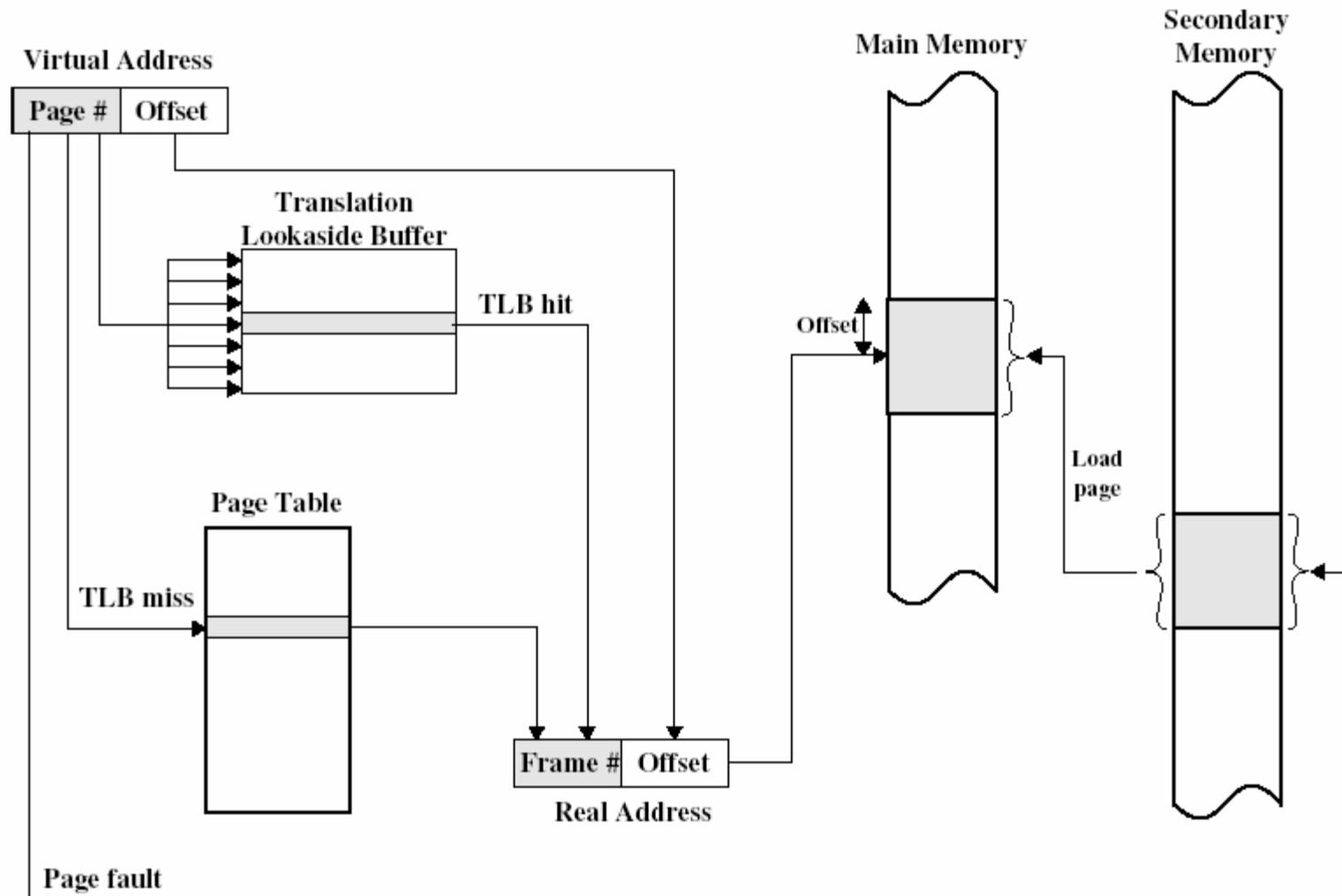


VM Implementation Issue

- Problem:
 - 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)



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 OS control
 - Hardware-loaded TLB
 - On miss, hardware performs PT lookup and reloads TLB
 - Example: Pentium
 - Software-loaded TLB
 - On miss, hardware generates a TLB miss exception, and exception handler reloads TLB
 - Example: MIPS
- 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
- 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



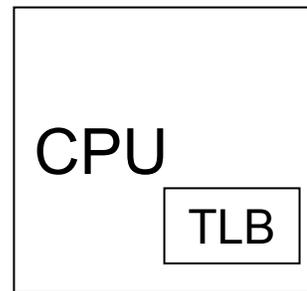
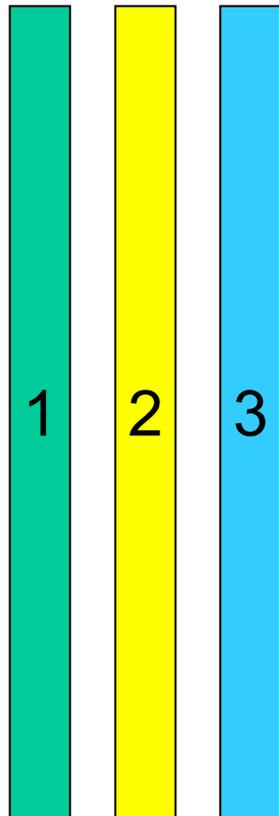
TLB effect

- 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

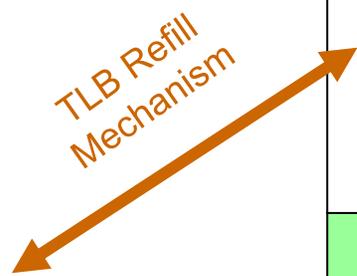


Simplified Components of VM System

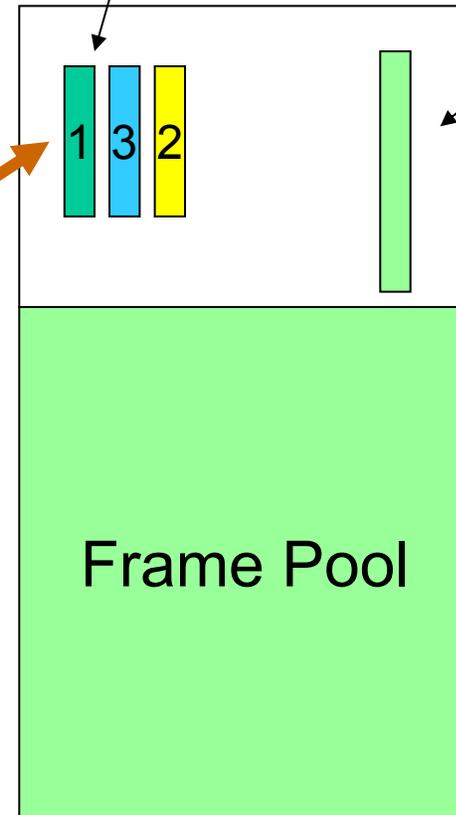
Virtual Address Spaces (3 processes)



TLB Refill Mechanism



Page Tables for 3 processes



Frame Table

Physical Memory



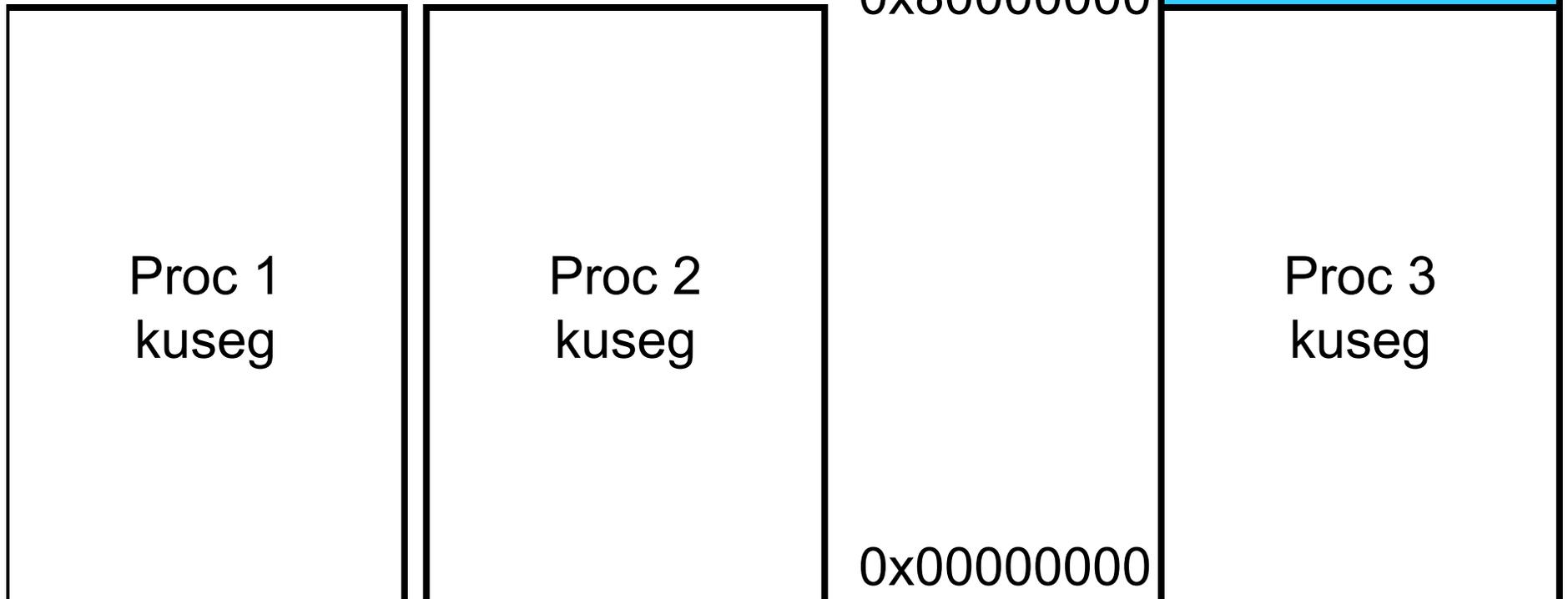
R3000 Address Space Layout

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



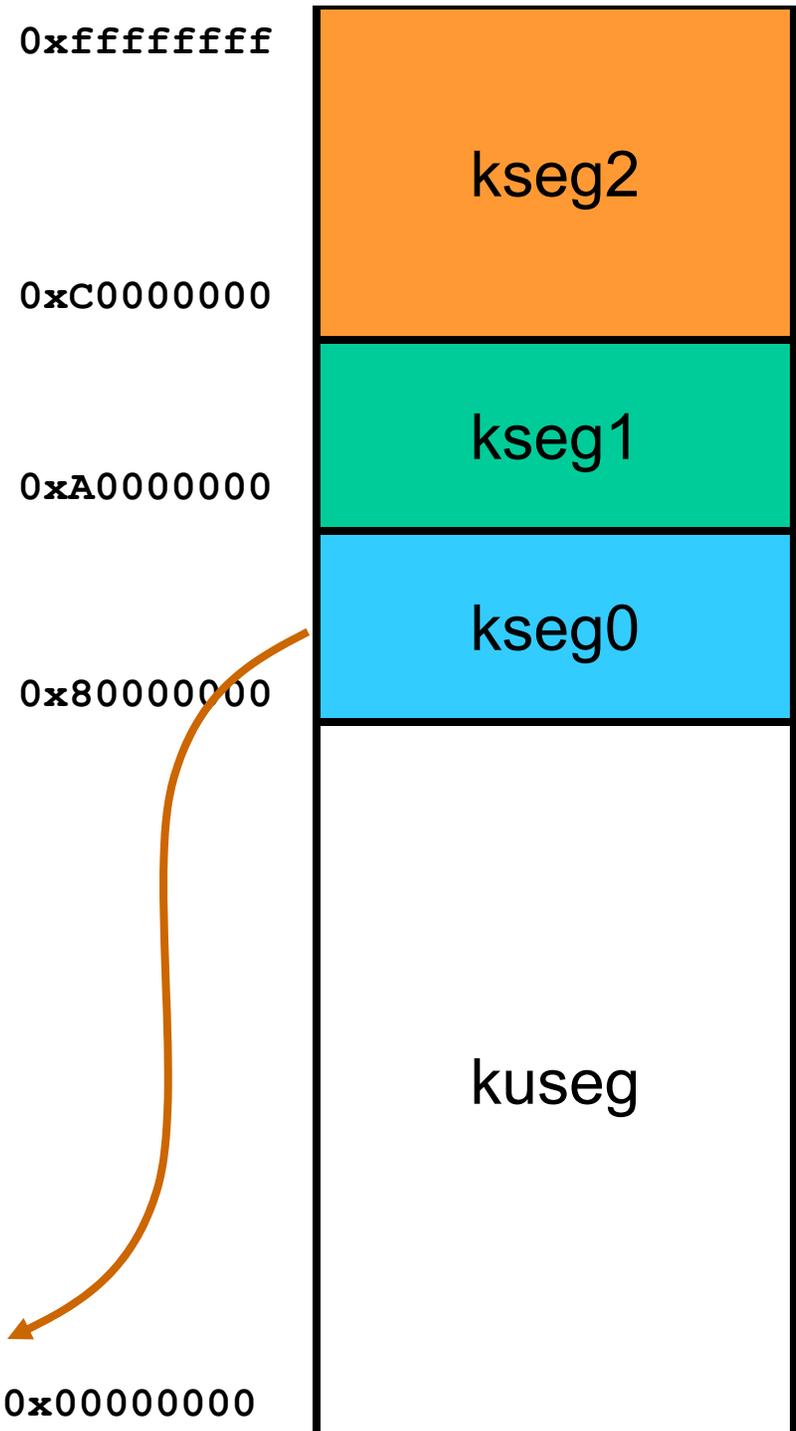
R3000 Address Space Layout

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



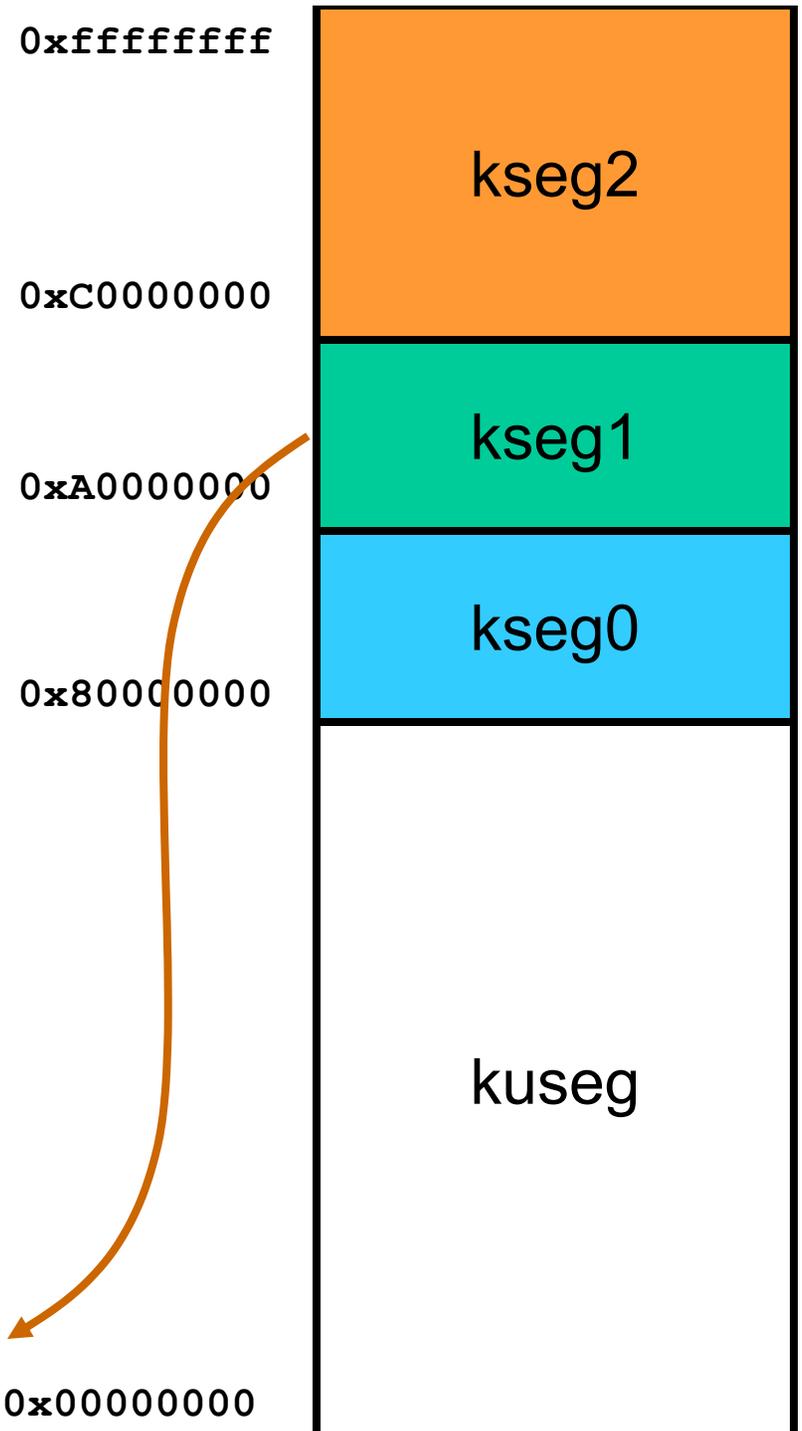
R3000 Address Space Layout

- 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 is placed



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)



R3000 Address Space Layout

- kseg2:
 - 1024 megabytes
 - TLB translated (mapped)
 - Cacheable
 - Depending on the 'N'-bit
 - Only kernel-mode accessible
 - Can be used to store the virtual linear array page table

