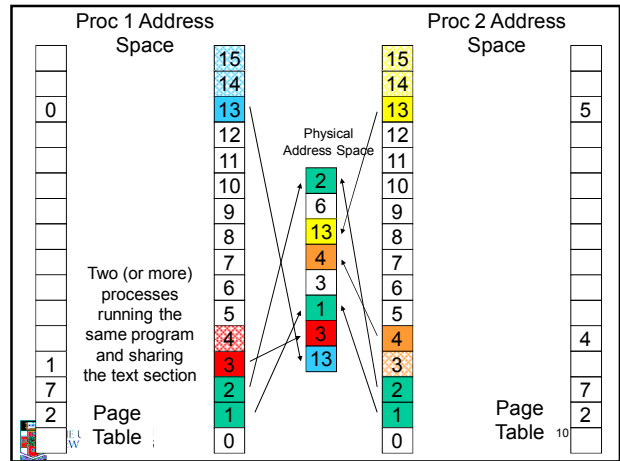


Shared Pages

- 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

9



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
 - Caching disabled
 - Modified
 - Present/absent
 - Referenced
 - Protection

Page Table

11

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

12

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

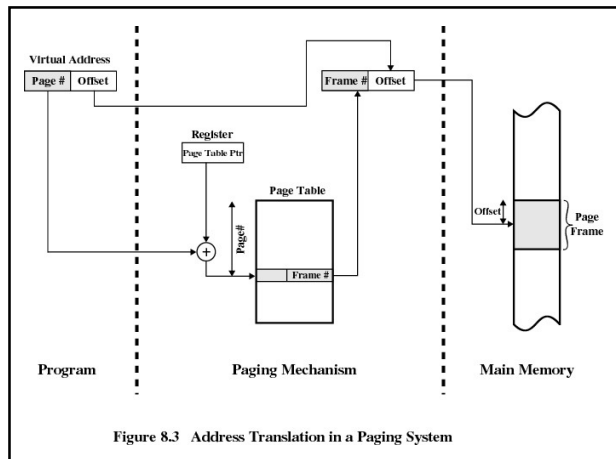


Figure 8.3 Address Translation in a Paging System

Page tables (recap)

virtual memory

virtual and physical mem chopped up in pages

- 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

The diagram illustrates the mapping process. A 'Virtual address' is split into a '20-bit virtual page' and a '12-bit offset'. The '20-bit virtual page' is used to look up a '15-bit memory address' in a 'Page table'. The '15-bit memory address' and the '12-bit offset' are then used to form the final '32-bit physical address'. The 'Page table' also includes a 'Present/absent bit'.

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

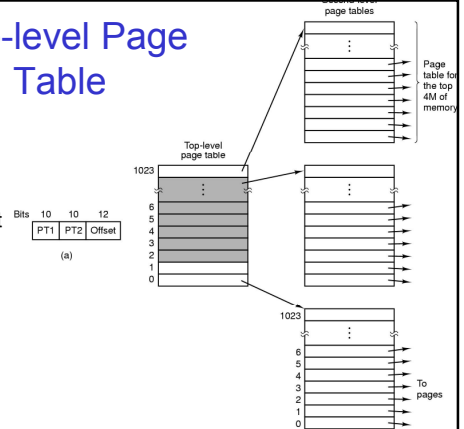
- Assume we have
 - 64-bit virtual address (humungous address space)
 - 4 KByte page size
 - How many page table entries do we need for one process?
- Problem:
 - Page table is very large
 - Access has to be fast, lookup for every memory reference
 - Where do we store the page table?
 - Registers?
 -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

Page Tables

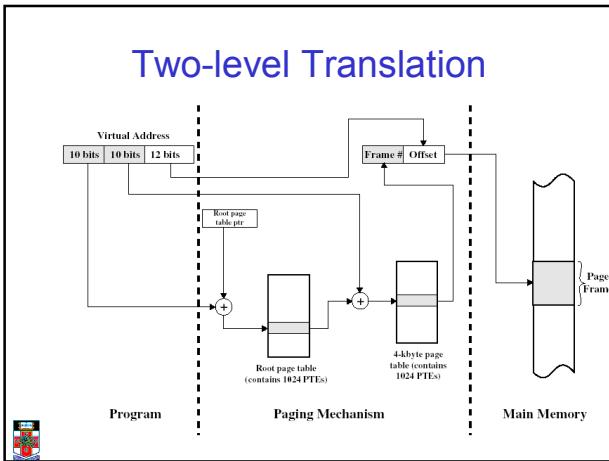
- Page tables are implemented as data structures in main memory
- Most processes do not use the full 4GB address space
 - e.g., 0.1 – 1 MB text, 0.1 – 10 MB data, 0.1 MB stack
- We need a compact representation that does not waste space
 - But is still very fast to search
- Three basic schemes
 - Use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

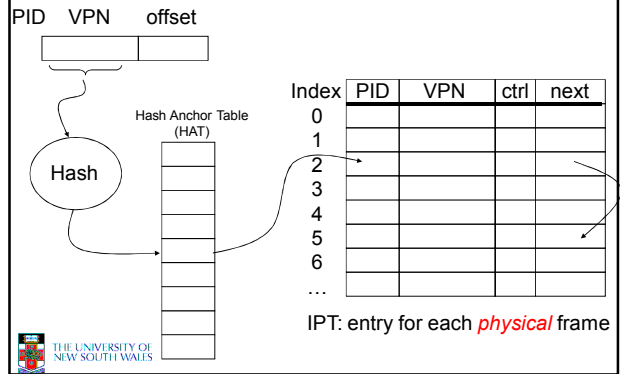
- 2nd level page tables representing unmapped pages are not allocated
 - Null in the top-level page table



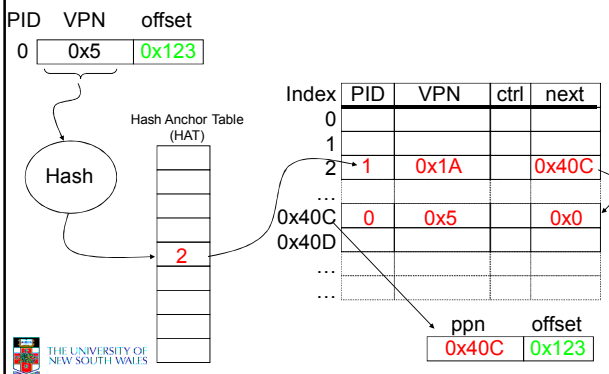
Two-level Translation



Alternative: Inverted Page Table



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 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

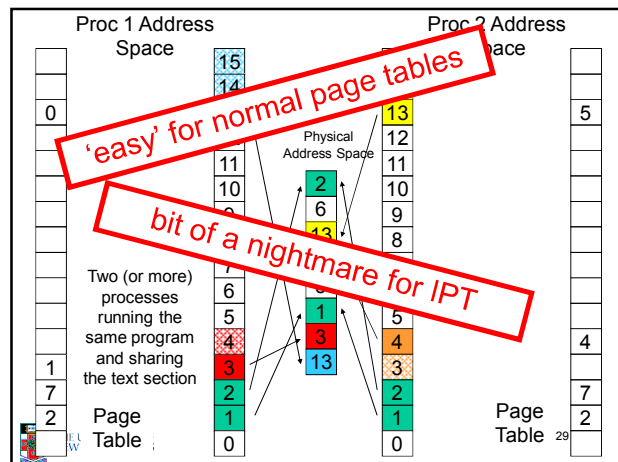
Properties of IPTs

- 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?

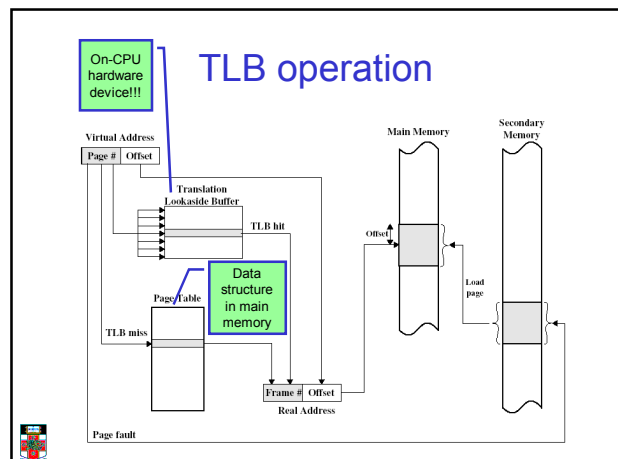
Another look at sharing...



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

page #	frame #	V	W
...
...

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: 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

