Week 14 – RTOS: Memory Management

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Today's Lecture

- Memory Protection using the MPU
- Introduction to virtual memory
- How does virtual memory work?
- Single-level and two-level page tables
- Translation lookaside buffers
- A quick look at ARM's memory management unit

Memory Protection

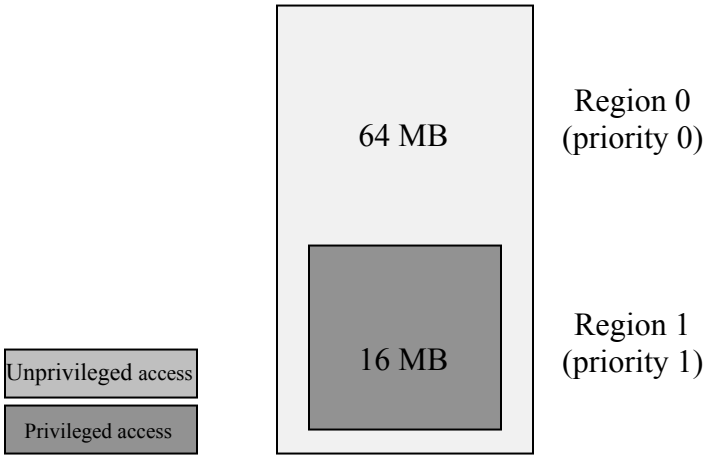
- ARM processors provide two different ways of protecting a system i.e. by providing restricted access to system's resources (memory and peripheral devices)
  - MPU -- Memory Protection Unit
  - MMU -- Memory Management Unit
- Some ARM processors only have an MPU while others only have an MMU
- ARM's Memory Protection Unit
  - Allows physical memory to be divided into 8 or fewer sections or regions
  - Can specify the starting address and size of each section
  - Maximum Size -- 4GB
  - Minimum Size -- 4KB
  - Access permissions can be specified for each section
  - Regions can have privileged mode only access, privileged mode full access and user mode read-only access, full access

Regions

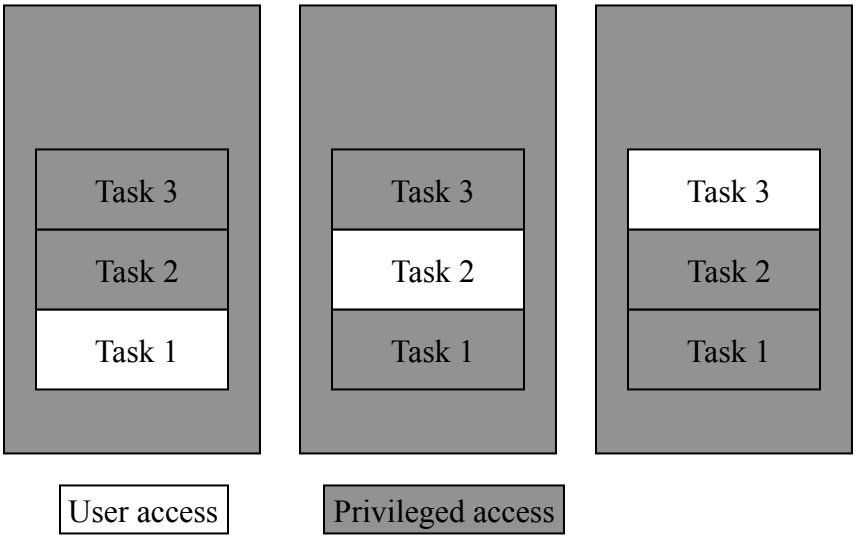
MPU: Regions

- A maximum of 8 regions can be defined
- Rules governing regions
  - Regions assigned privileges
  - Regions can overlap with other regions
  - Regions are assigned priorities independent of privilege level of the region
  - When regions overlap, the attributes of region with highest priority take precedence in the overlapped memory locations
  - A region's starting address must be a multiple of its size
  - Size of a region can be a power of two from 4KB to 4GB
  - Data or prefetch abort is generated when the processor accesses a region in a mode where it does not have access to a region

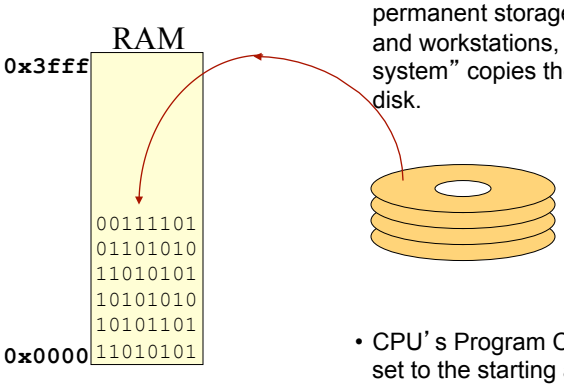
Overlapping Regions



Overlapping Regions



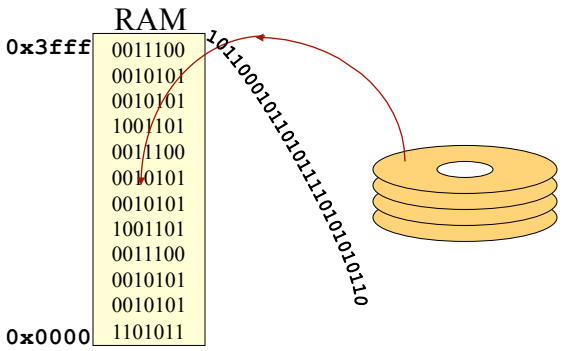
Virtual Memory Introduction – How Does a Program Start Running?



- OS (loader) copies a program from permanent storage into RAM - on PCs and workstations, the "operating system" copies the program (bits) from disk.
- CPU's Program Counter is then set to the starting address of the program and the program begins execution.

What if the program is too big?

- Some machines won't let you run the program
  - Original DOS
  - Why this limitation?

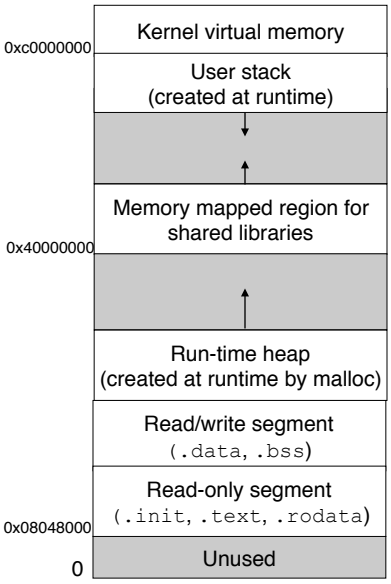


Solution: Virtual Memory

- What is **virtual memory**?
  - Technique that allows execution of a program that may not completely reside in memory (RAM)
- Allows the computer to "fake" a program into believing that its memory space is larger than physical RAM
- Key concept -- use DRAM as a cache for the disk
  - Address space of a process can exceed main memory size
  - Sum of address space of multiple processes can exceed main memory size
- Why is VM important?
  - Cheap - no longer have to buy lots of RAM
  - Removes burden of memory resource management from the programmer

But Wait There is More…

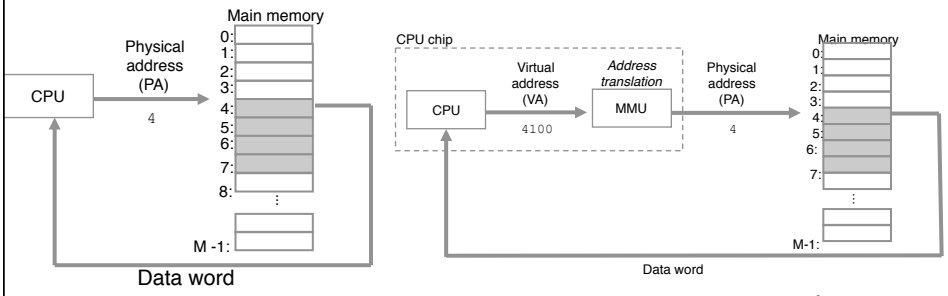
- Virtual Memory concept also
  - Simplifies memory management by providing each process with same uniform address space
  - Protects one process from corrupting data of another process or corrupting its own read only (text) sections
- In embedded systems virtual memory is typically used to provide memory protection and uniform address space for processes



Source: Bryant & O'Hallaron

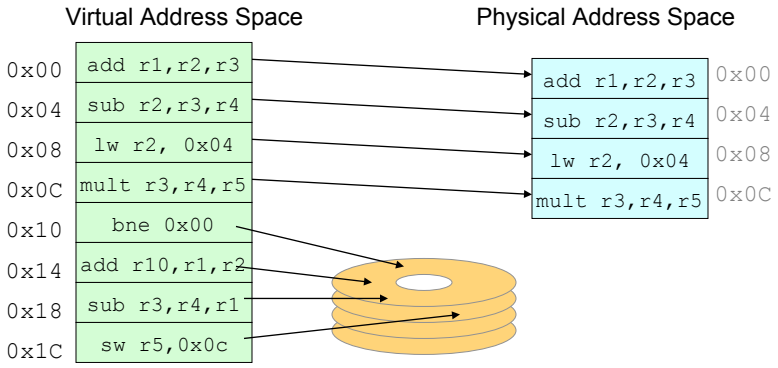
Physical versus Virtual Addressing

- Two memory "spaces"
  - **Virtual memory space** - what the program "sees"
  - **Physical memory space** - what the program runs in (size of RAM)
- Virtual memory requires
  - Dedicated hardware on CPU chip called Memory Management Unit (MMU)
  - Cooperation between CPU hardware & operating system

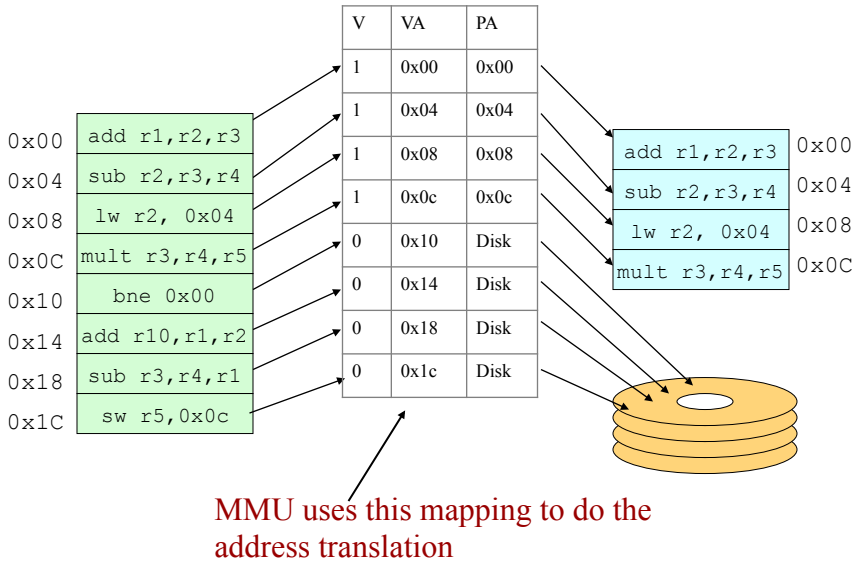


Source: Bryant & O'Hallaron

Example: Virtual and Physical Address Spaces

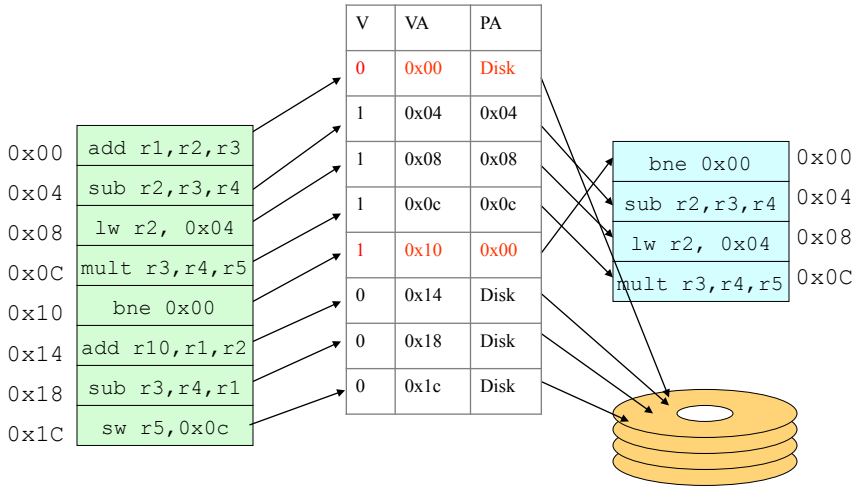


Example (con'td): Need VA-to-PA mappings

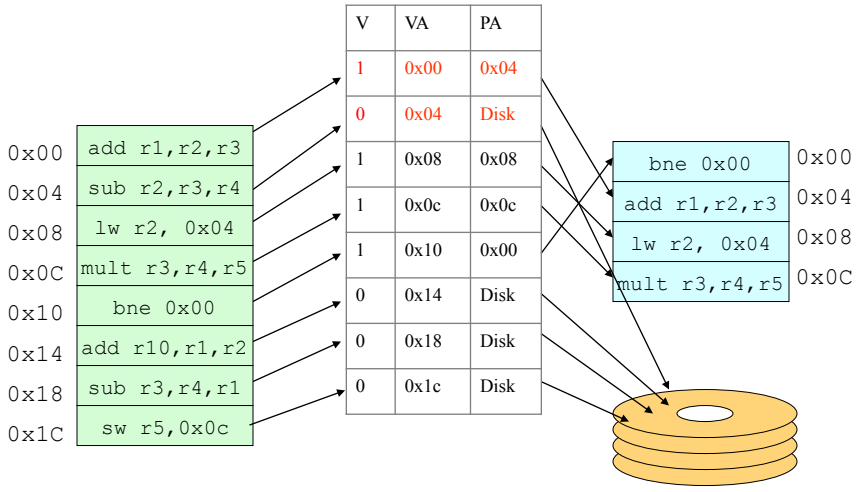


MMU uses this mapping to do the address translation

Example (cont'd): **After** handling a *page fault*

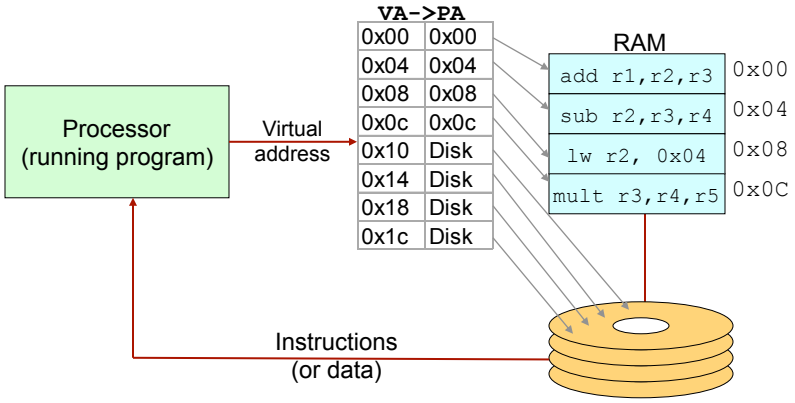


Example (con'td): After a second page fault



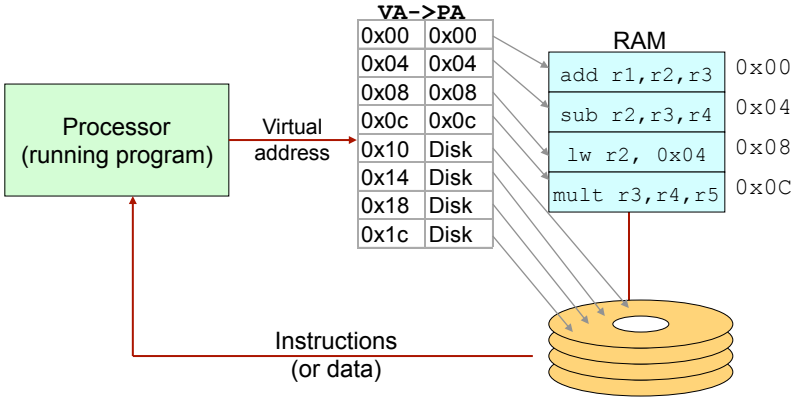
Basic VM Algorithm

- Program asks for virtual address
- MMU translates **virtual address** (VA) to **physical address** (PA)
- Computer reads PA from RAM, returning it to program



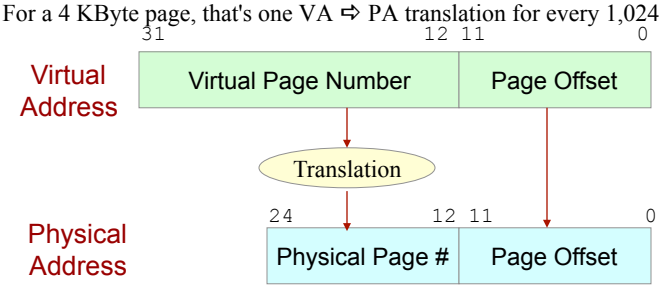
Page Tables

- Table which holds VA ⇨ PA translations is called the **page table**
- In our current scheme, each word is translated from a virtual address to a physical address
  - Where is the page table located?
  - How big is the page table (for our scheme assuming 32-bit addresses)?

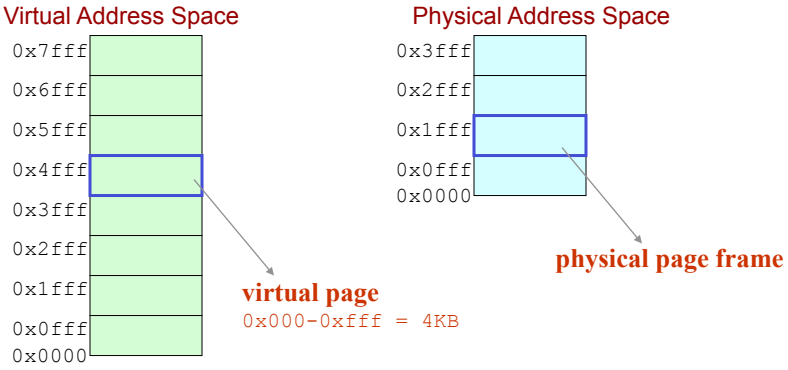


Real Page Tables

- Instead of the fine-grained VM where any virtual word can map to any RAM word location, partition memory into chunks called **pages**
  - Typical page sizes are 1, 4, 8, 64 Kbytes or 1 Mbyte
  - With 32-bit addresses and assuming 4 KByte page size, the 20 MSBs determine page number
- Map each virtual page to a physical page
  - ***Within*** a page, the virtual offset == physical offset
- This reduces the number of VA ⇨PA translation entries
  - Only one translation per page
  - For a 4 KByte page, that's one VA ⇨ PA translation for every 1,024 words



Virtual Pages & Physical Page Frames



Every address within a virtual page maps to the same location within a physical page frame

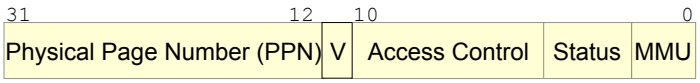
• In other words, bottom **log2**(page size in bytes) bits are **not** translated

Page Table

- Who fills/sets up and manages the page table?
- How does the MMU know where the page table is located in memory?
  - A register (typically called page-table base register (PTBR)) is used to store the starting address of the table
- Virtual page number can be used as an index into this table to determine the corresponding physical page number
- The entries in this table are called page table entries (PTE) and store the mapping to physical page number
- Each PTE is **typically** 32 bits (could also be less or more than 32 bits)
  - Assuming 4 KByte page size
    - need at most 20 bits of the PTE for storing the physical page number
  - What about the remaining 12 bits?
  - Can use these remaining bits to store other information e.g. access permissions (whether this physical page stores code/data etc.)

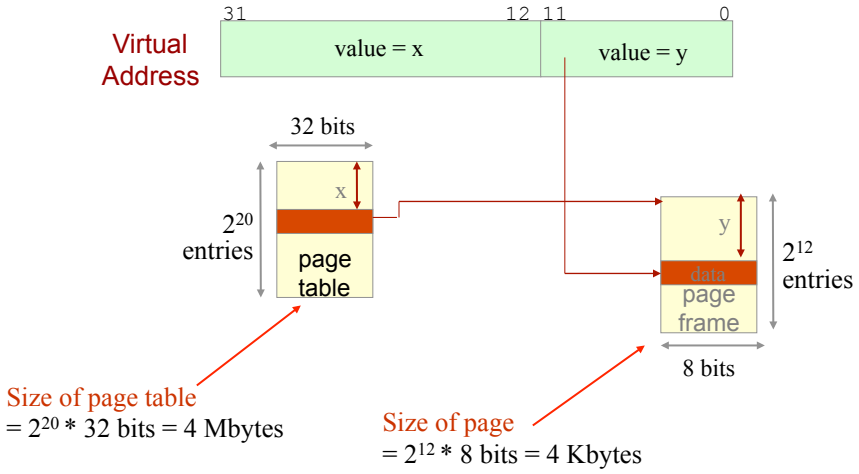
Page Table Entries

- A real **page table entry** (PTE) contains a number of fields
  - Physical page number
  - Valid/Invalid bit
  - Access control bits (e.g., writeable bit)
  - Status bits (e.g., accessed and dirty bits)
  - MMU control bits (e.g., cacheable bits)



Sample page table entry

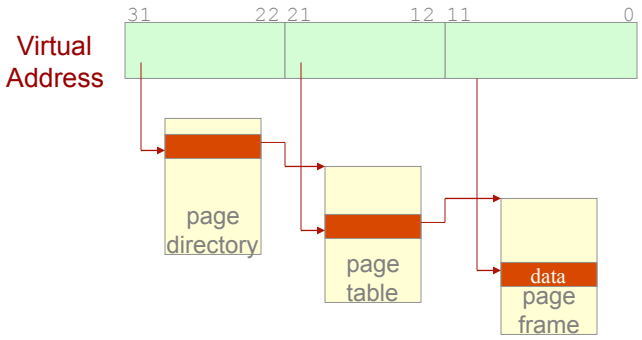
Example: Single-Level Page Table



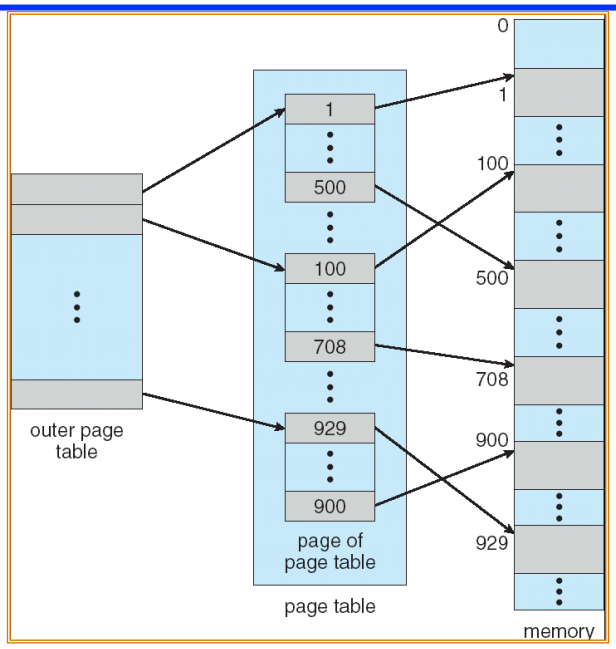
Single-Level Page Table

- Assumptions
  - 32-bit virtual addresses
  - 4 Kbyte page size = 2^{12} bytes
  - 32-bit address space
- How many virtual page numbers?
  - $2^{32} / 2^{12} = 2^{20}$ = 1,048,576 virtual page numbers = number of entries in the page table
- If each page table entry occupies 4 bytes, how much memory is needed to store the page table?
  - 2^{20} entries * 4 bytes = 2^{22} bytes = 4 Mbytes
- How do we reduce the amount of main memory to store the page table?

Two-level Page Tables

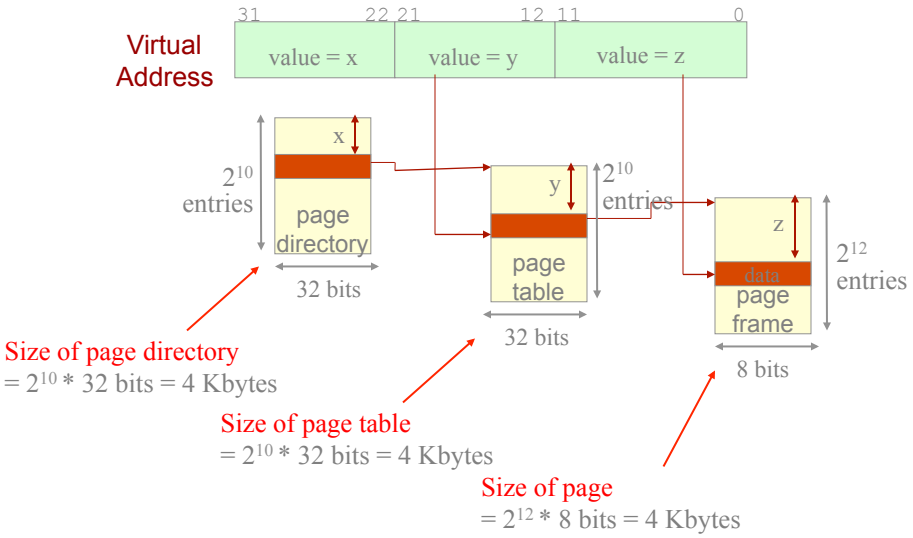


Two-Level Page Table Scheme



Source:Silberschatz,Galvin & Gagne

Example: Two-level Page Table

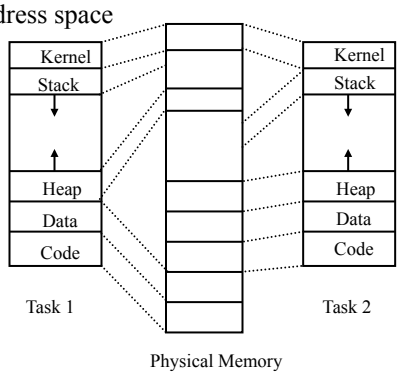


Two-Level Page Table

- Assumptions
  - 2^{10} entries in page directory (= max number of page tables)
  - 2^{10} entries in page table
  - 32 bits allocated for each page directory entry
  - 32 bits allocated for each page table entry
- How much memory is needed?
  - Page table size = 2^{10} entries * 32 bits = 2^{12} bytes = 4 Kbytes
  - Page directory size = 2^{10} entries * 32 bits = 2^{12} bytes = 4 Kbytes

So Where Are We?

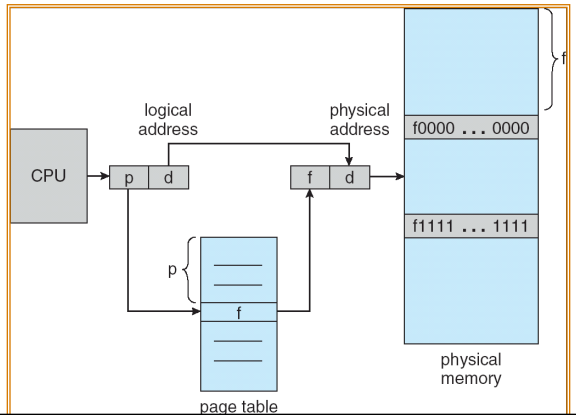
- Virtual Memory simplifies memory management by providing each process with same uniform address space



- Protects one process from corrupting data of another process or corrupting its own read only (text) sections
  - The OS can ensure that the VA generated by a process map only to the PA of that process only (unless it explicitly shares code or data)
  - Page Table Entries have access bits that can be used to disallow writes to text or read-only data sections or kernel code/data or shared library

Address translation

- The page table is typically stored in memory
- Steps involved in accessing a page in memory (assuming single-level page table)
  - MMU obtains the address from the CPU & partitions the address into virtual page number VPN and the virtual page offset VPO
  - MMU generates the address of the page table entry (PTE) and requests the contents from memory
  - Memory returns PTE to MMU
  - MMU constructs the physical address and sends it to memory
  - Memory returns the requested data word to CPU

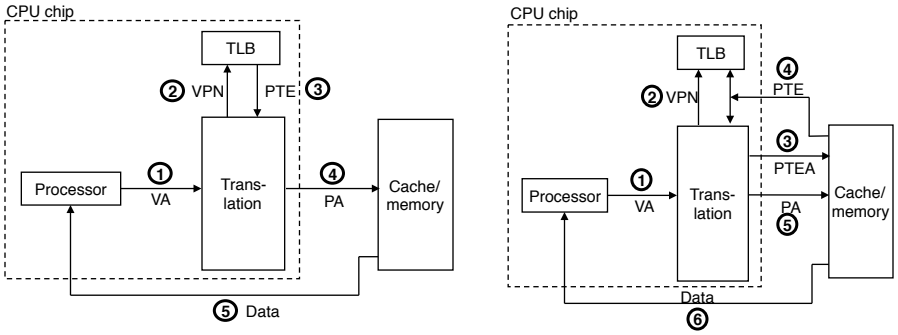


Source: Silberschatz, Galvin & Gagne

Speeding Up Address Translation with TLB

- Where is the page table stored?
- Requires two memory accesses to access any memory location
- Use "Translation Lookaside Buffer" (TLB) to speed up address translation
  - TLB is a small, fast-lookup hardware cache in the MMU
  - TLB contains a few page-table entries
  - When virtual address is generated, it is first presented to TLB
  - If page number is found in TLB (known as **TLB hit**), its frame number is immediately available and is used to access main memory
  - If page number is not found in TLB (known as **TLB miss**), a memory reference to the page table is made

Translation Lookaside Buffer



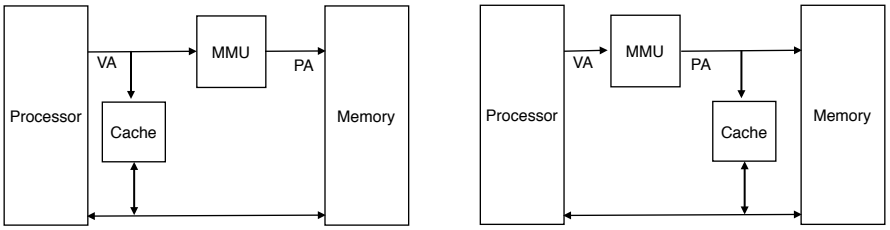
(a) TLB Hit

(b) TLB Miss

Source: Bryant & O'Hallaron

Interaction With Caches

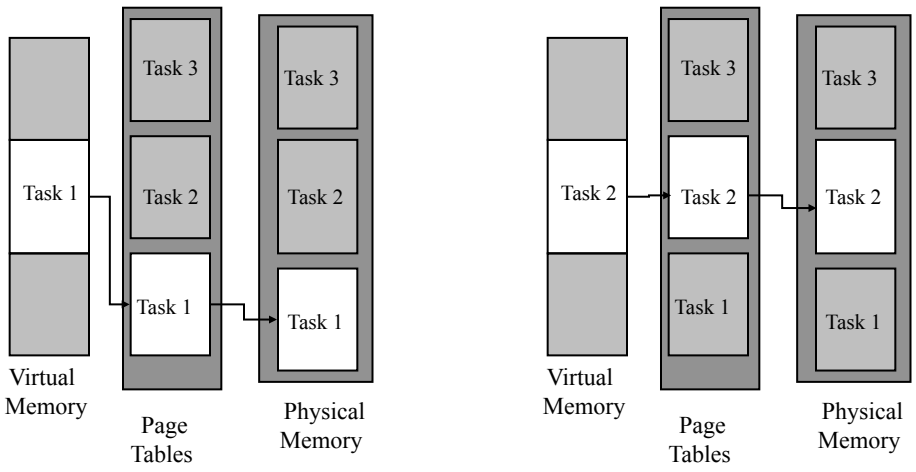
- Two possibilities for instruction and data cache (not the TLB) locations
- Logical/Virtual Cache: operates on virtual addresses
  - Advantages/disadvantages?
  - Example: Instruction/Data caches on the XScale processor
- Physical Cache: operates on physical addresses
  - Advantages/disadvantages?
  - Example: Instruction/Data caches on the Intel Pentium processor



Logical cache Physical cache

Context Switching With Virtual Memory

- Each task in the system has its own page table
  - PTBR needs to be part of the context



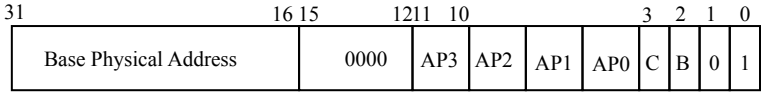
Task 1 running Task 2 running

More On Context Switching

- TLB entries need to be invalidated
  - Since the new process has the same virtual address space (but different physical address space), TLB entries are no longer valid
- Need to clean virtual cache (if any)
  - Cleaning virtual cache may involve more than invalidating the cache item
  - Cache memory on write could follow one of the two policies
    - Writethrough caches: on a cache hit for a write operation, both the cache as well as the main memory is updated
    - Writeback caches: on a cache hit for a write operation, only the cache is updated
      - Memory is updated only when the cache item is evicted
  - For a writeback logical cache, cleaning the cache would involve writing back all the modified cache locations

ARM MMU

- Presents 4 GB virtual address space (why?)
- Memory granularity: 4 options supported
  - 1MB sections
  - Large pages (64 KBytes) - access control within a large page on 16 KBytes
  - Small pages (4 KBytes) - access control within a large page on 1 Kbytes
  - Tiny pages (1KBytes)



Large page table entry

ARM MMU (contd.)

- Puts processor in **Abort Mode** when virtual address not mapped or permission check fails
- A page table base register (called the **translation table base register**, in ARM jargon) stores the base address of the page table
  - Useful for context switching of processes
- XScale processor (used on the XBoard) has
  - 32 entry instruction and data TLB
  - 32 Kbyte instruction and data cache

Summary of Lecture

- Virtual Memory (VM)
  - What is VM?
  - How does VM work?
  - Virtual address to physical address mappings
  - Page faults
  - VM Schemes
    - page tables
    - virtual pages and physical page frames
    - page table entries
  - TLBs
  - Interaction with caches
- References
  - R. E. Bryant & D. R. O'Hallaron, *"Computer Systems: A Programmer's Perspective"* Chapter 10
  - Silberschatz, Galvin and Gagne, "*Operating System Concept*" Chapters 8 & 9
  - A. S. Tannenbaum, "*Modern Operating Systems*", Chapter 4