Memory Management

Sirak Kaewjamnong

517-312 Operating Systems

Memory hierarchy

Computers have:

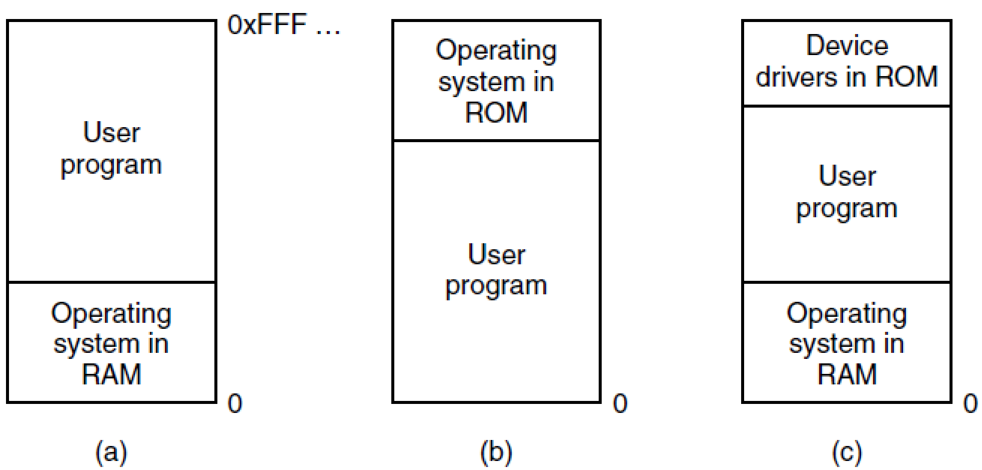
- A few megabytes of very fast, expensive, volatile cache memory
- A few gigabytes of medium-speed, medium-priced volatile main memory
- A few terabytes of slow, cheap, nonvolatile magnetic or solid state disk storage
- Some removable storages such as DVD and USB sticks

OS abstracts this hierarchy into a useful model and then manages the abstraction

No memory abstraction

- In the past (before 1980 in PCs and before 1970 and 1960 in minicomputers and mainframe) had no memory abstraction. Every program saw the physical memory
- Under this condition, it was not possible to have two running programs in memory at the same time
  - What happen if the first program wrote a value in a physical location and then another wrote a new value to the same location???

No memory abstraction



Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

Running multiple programs without a memory abstraction

- The early model of IBM360 divides memory into 2 KB blocks and each assigned a 4 bit protection key held in special registers inside the CPU
- A machine with 1 MB memory needed 512 of 4 bit registers for a total of 256 bytes key storage
- The IBM 360 hardware trapped any attempt by a running process to access memory with a protection code different from its key
- Only the OS could change the protection keys, user processes were prevented from interfering with other process's space

Drawback

- Both program refer to absolute physical memory
- Memory location in the second program(b) have to be converted
  - Static relocation was used
  - It requires extra information in all programs to indicate which would contain relocation addresses
  - Thus, it is slow and complicate

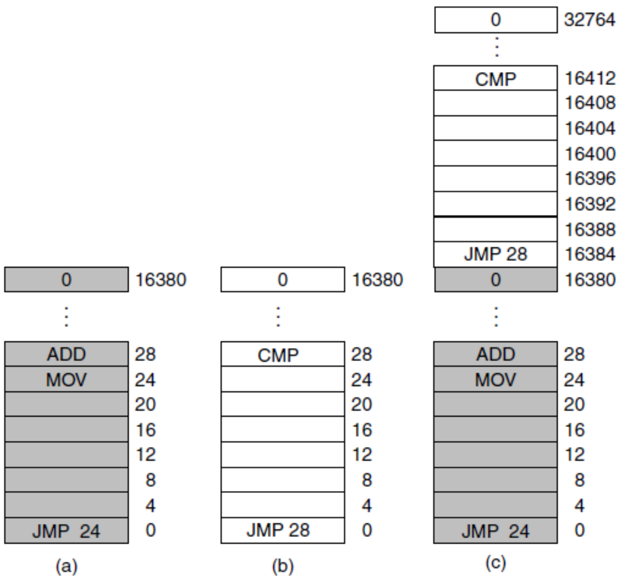


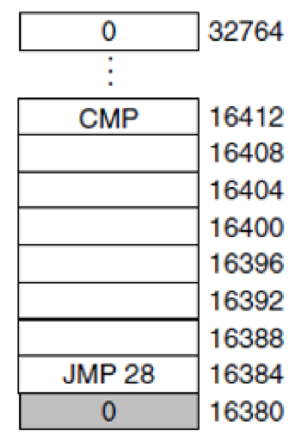
Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory

Memory abstraction: address space

- An address space is the set of addresses that a process can use to address memory
- Each process has it own address space
- It is independent from those belonging to other processes
- Address 28 in one program means a different physical location than address 28 in another program

Base and limit registers

- Dynamic relocation maps each process's address space onto a different part of physical memory
- Two special hardware registers are used, "base" and "limit"
- Every time a process references memory, the CPU hardware automatically adds the base value to the address
- It checks the address is equal to or greater then the value in the "limit" register

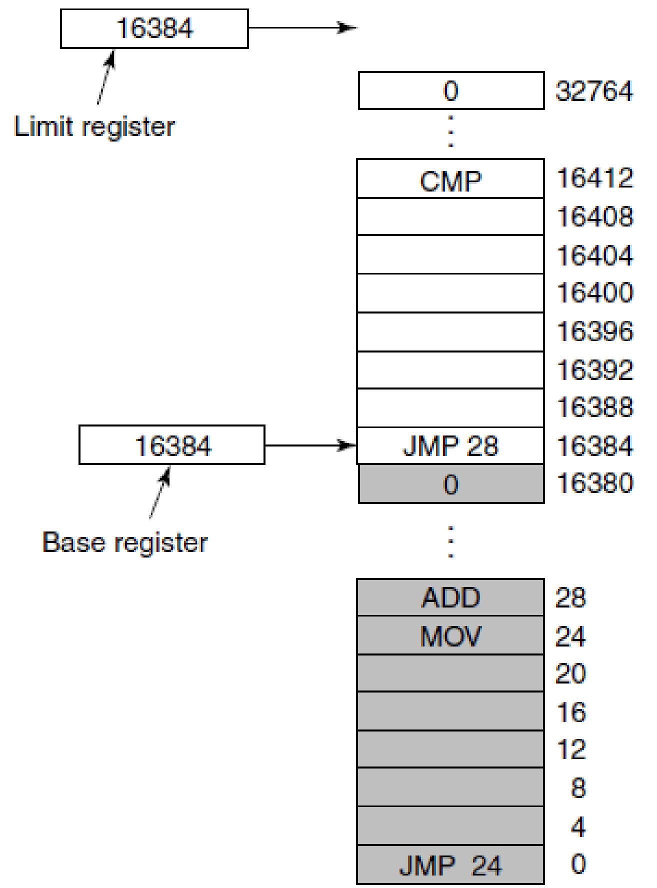


Base register contains 16384

28 is convert to 16412

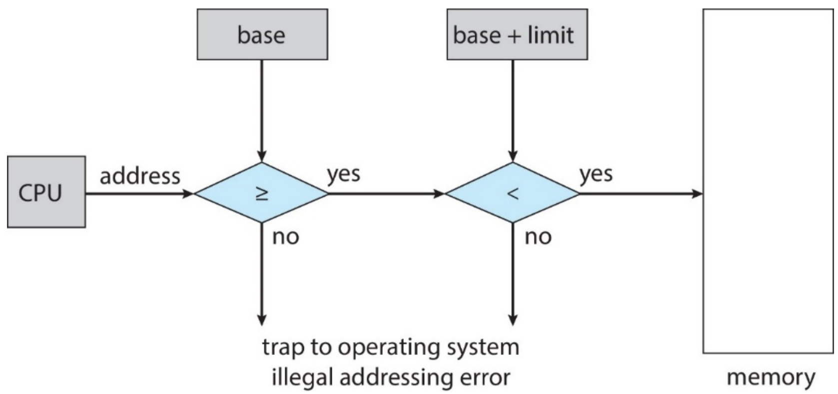
Base and limit registers

- Base and limit registers can be used to give each process a separate address space



Hardware address protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

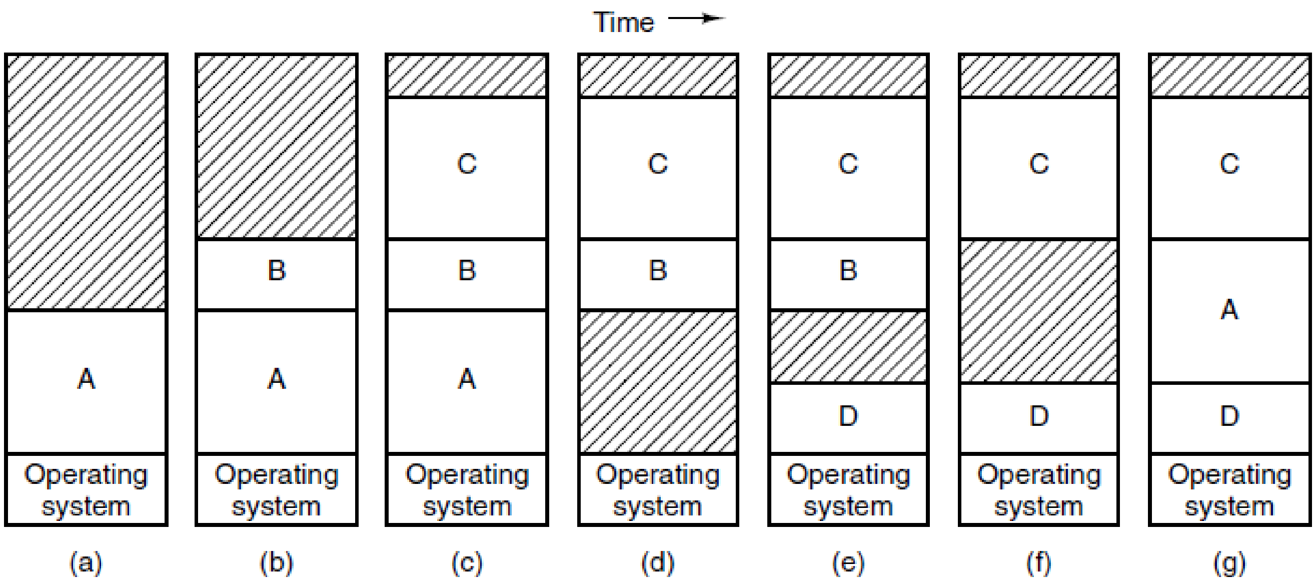


- the instructions to loading the base and limit registers are privileged by the OS

Swapping

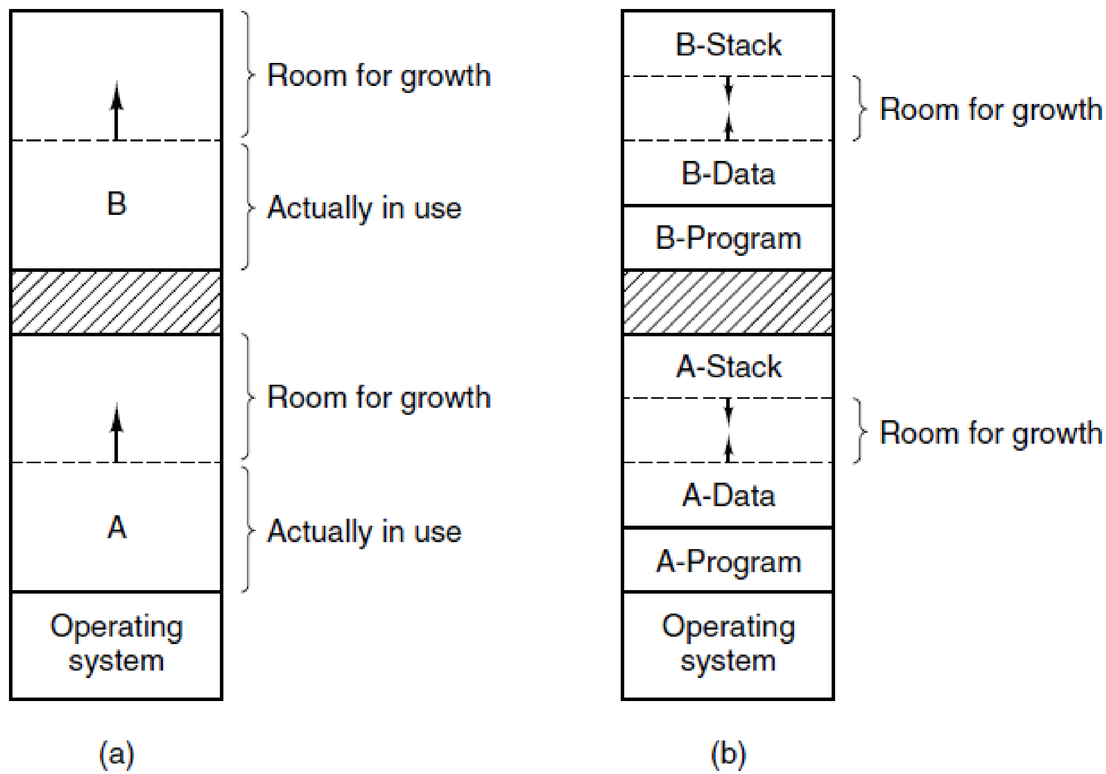
- The total amount of RAM needed by all the processes is often much more than the amount of physical memory
  - 50-100 processes may be started up as soon as the computer is booted
  - These processes may occupy 5-10 MB of memory
  - Some may require 500 MB or more
- Swapping: bringing in each process entirely in memory, running it for a while, then putting it back on the disk
- Virtual memory: allows processes to run even when they are only partially in main memory

Swapping



Memory allocation changes as processes come into memory and leave it.
The shaded regions are unused memory

Memory allocation

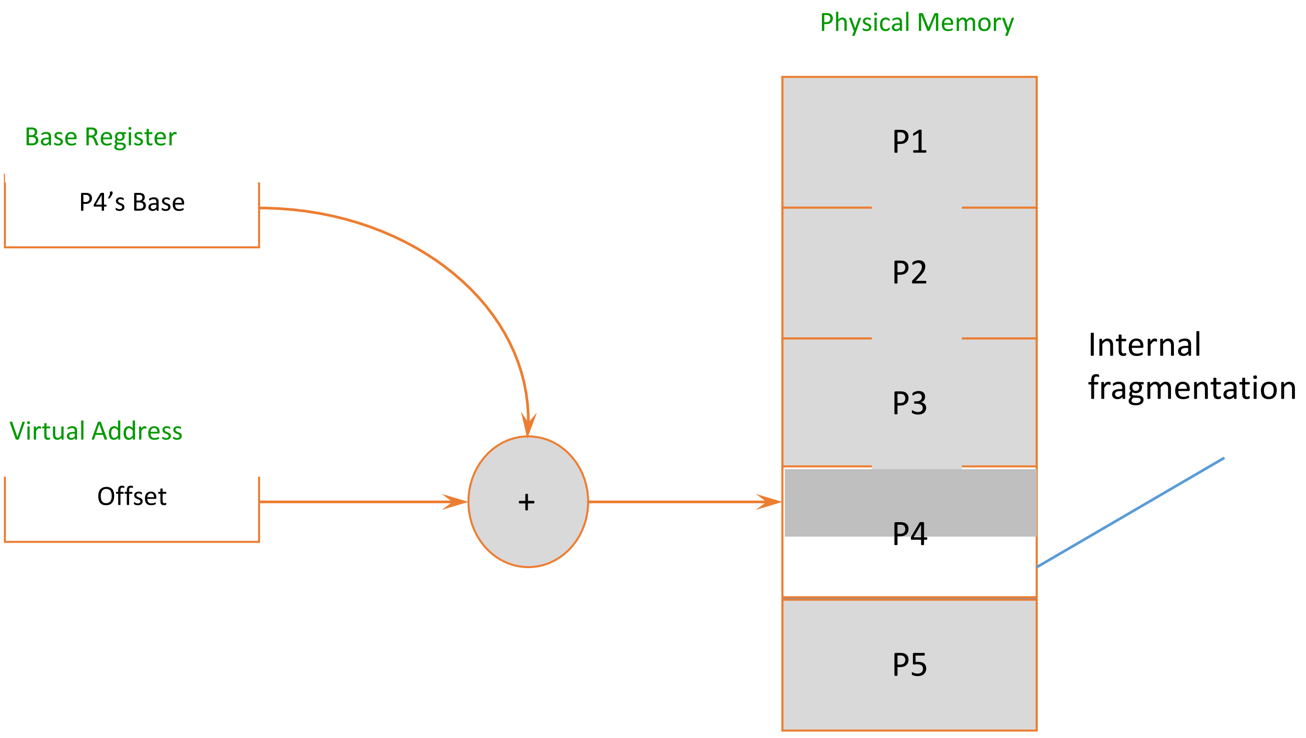


(a) Allocating space for a growing data segment.
(b) Allocating space for a growing stack and a growing data segment

Fixed partitions

- Physical memory is broken up into fixed partitions
  - Hardware requirements: base register
  - Physical address = virtual address + base register
  - Base register loaded by OS when it switches to a process
  - Size of each partition is the same and fixed
  - How do we provide protection?
- Advantages
  - Easy to implement, fast context switch
- Problems
  - Internal fragmentation: memory in a partition not used by a process is not available to other processes
  - Partition size: one size does not fit all (very large processes?)

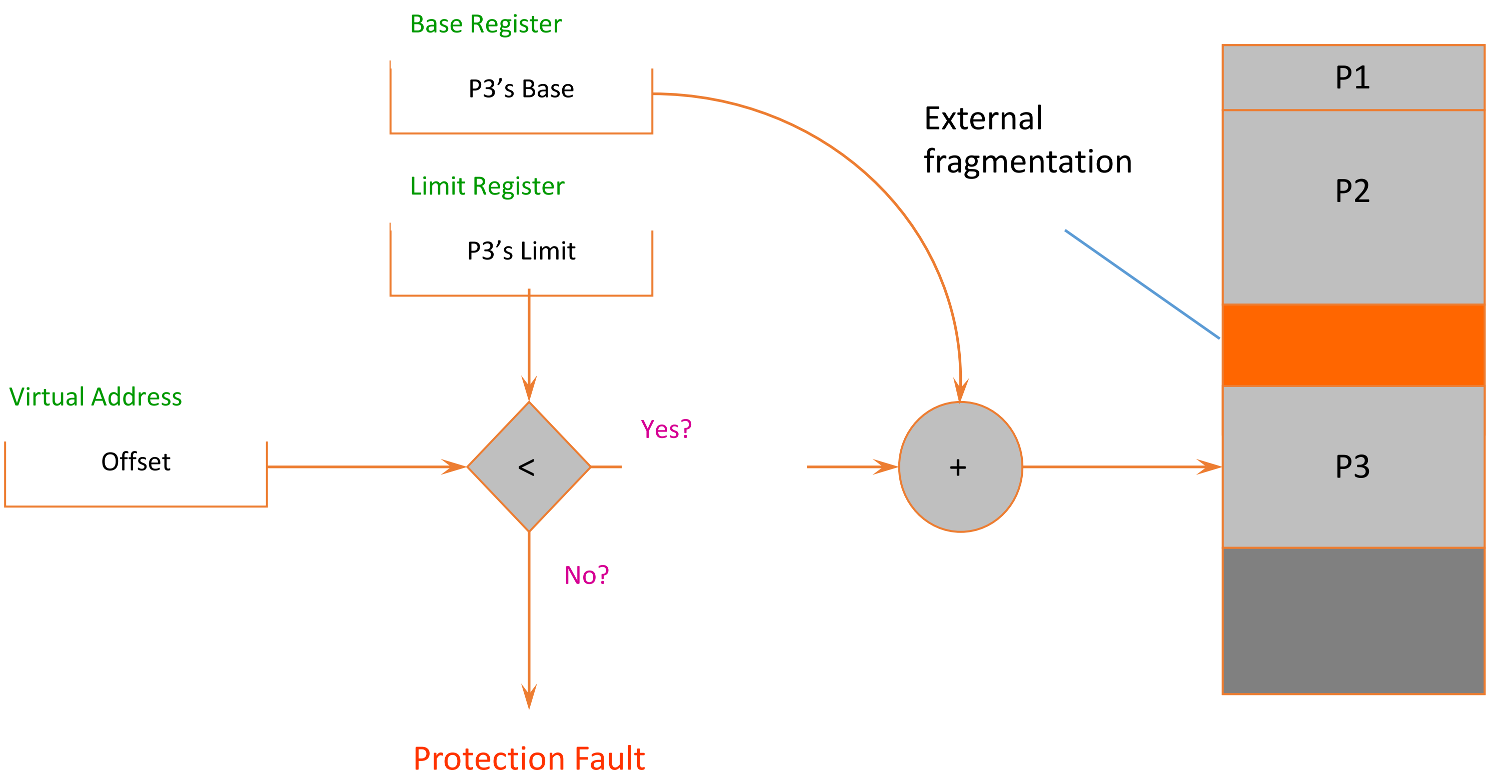
Fixed partitions



Variable partitions

- Natural extension – physical memory is broken up into variable sized partitions
  - Hardware requirements: base register and limit register
  - Physical address = virtual address + base register
  - Why do we need the limit register? Protection
    - If (physical address > base + limit) then exception fault
- Advantages
  - No internal fragmentation: allocate just enough for process
- Problems
  - External fragmentation: job loading and unloading produces empty holes scattered throughout memory

Variable partitions



Variable partitions and fragmentation

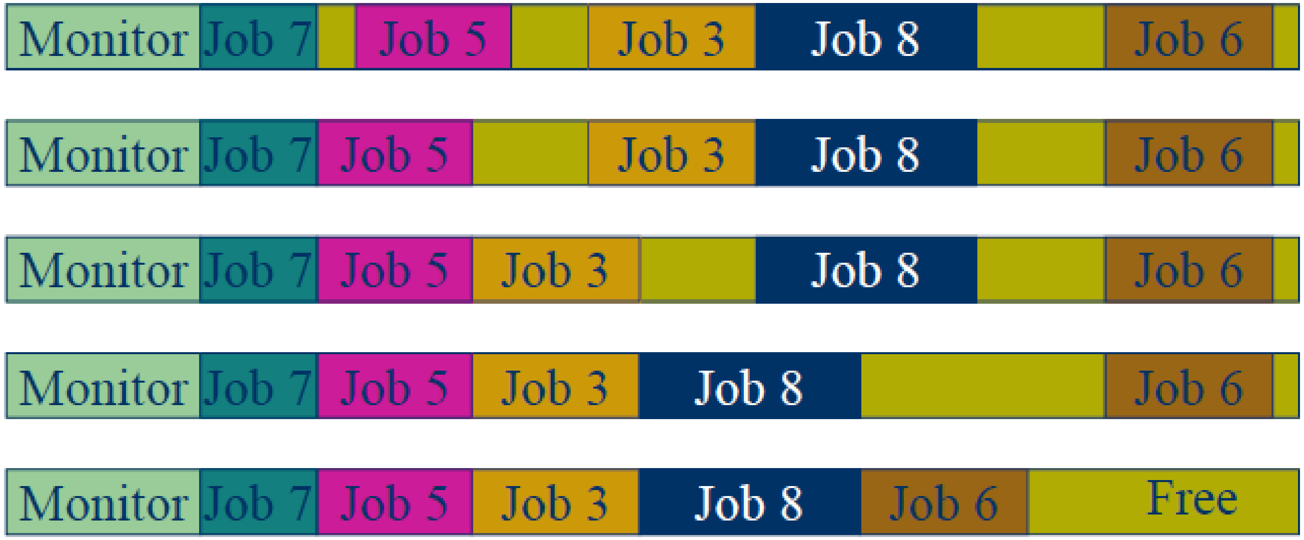
Memory wasted by External Fragmentation

Do you know about disk de-fragmentation?

It can improve your system performance!

Compaction

- Processes must be suspended during compaction
- Need be done only when fragmentation gets very bad



Virtual memory

- There is a need to run programs that are too large to fit in memory
- Solution adopted in the 1960s, split programs into little pieces, called overlays
  - Kept on the disk, swapped in and out of memory
- Virtual memory : each program has its own address space, broken up into chunks called pages
- Pages are mapped onto physical memory but not all pages have to be in physical memory at the same time

Why virtual memory?

- The abstraction that the OS will provide for managing memory is virtual memory (VM)
  - *Enables a program to execute with less than its complete data in physical memory*
    - A program can run on a machine with less memory than it "needs"
    - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
  - *Processes cannot see the memory of others*
  - OS will adjust amount of memory allocated to a process based upon its behavior
  - VM requires *hardware support* and OS management algorithms to pull it off

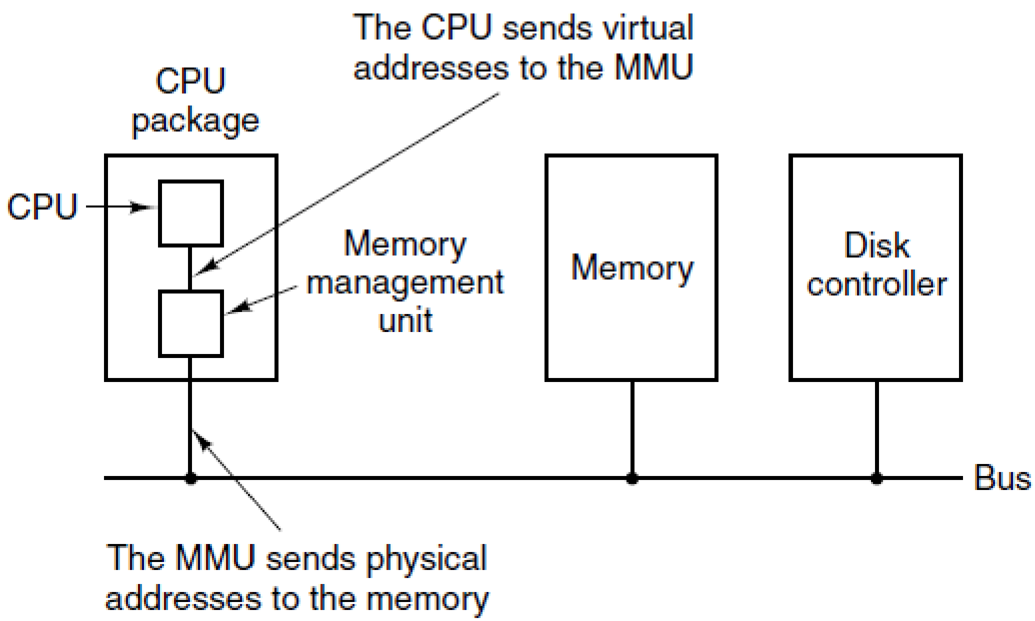
Virtual addresses or logical addresses

- Virtual addresses are independent of the actual physical location of the data referenced
- OS determines location of data in physical memory
- Instructions executed by the CPU issue virtual addresses
- Virtual addresses are translated by hardware into physical addresses (with help from OS)
- The set of virtual addresses that can be used by a process comprises its virtual address space

User/Process perspective

- Users (and processes) view memory as one contiguous address space from 0 through N
  - Virtual address space (VAS)
- In reality, pages are scattered throughout physical storage
  - Different from variable partition, where the physical memory for each process is contiguously allocated
- The mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
  - The address “0x1000” maps to different physical addresses in different processes

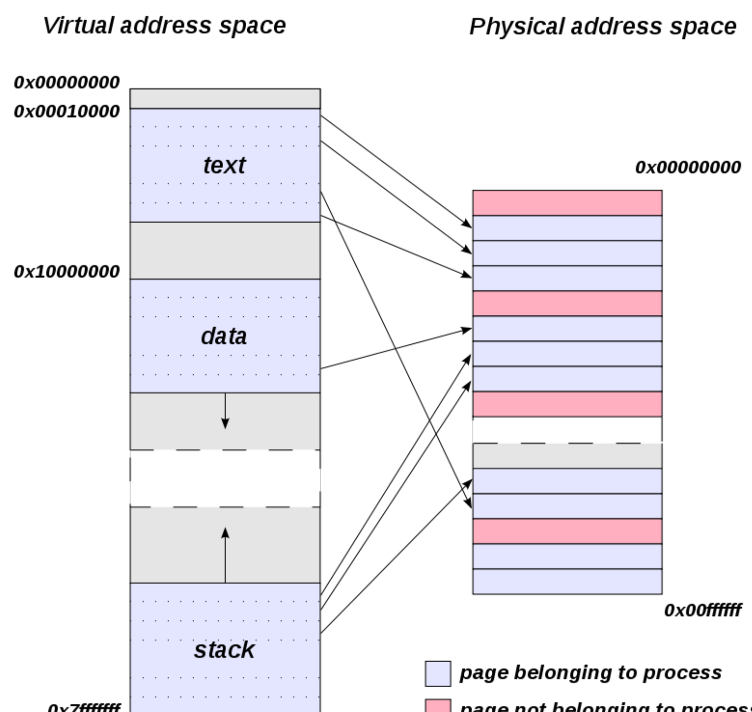
Memory management unit (MMU)



- The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays.
- However, logically it could be a separate chip and was years ago

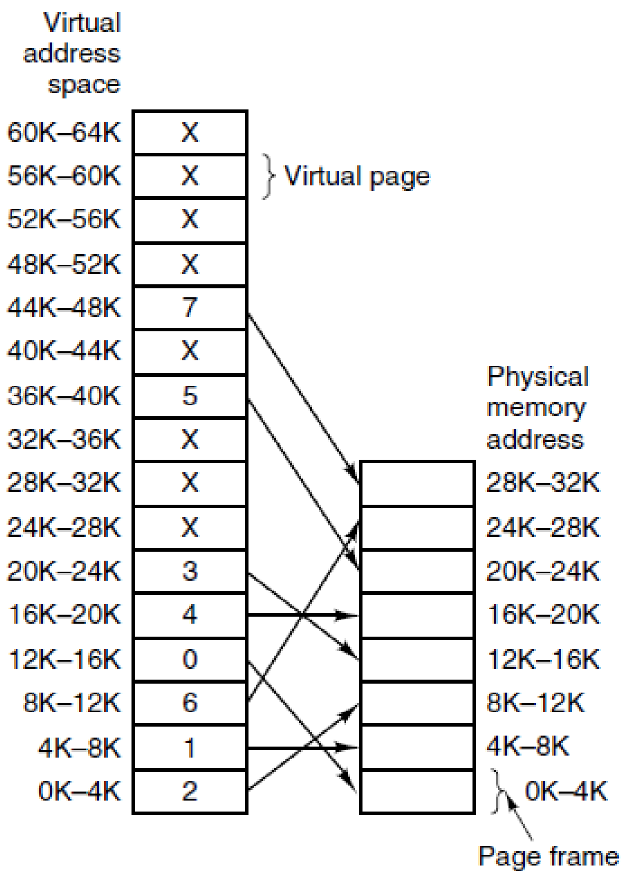
Paging

- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory



Paging

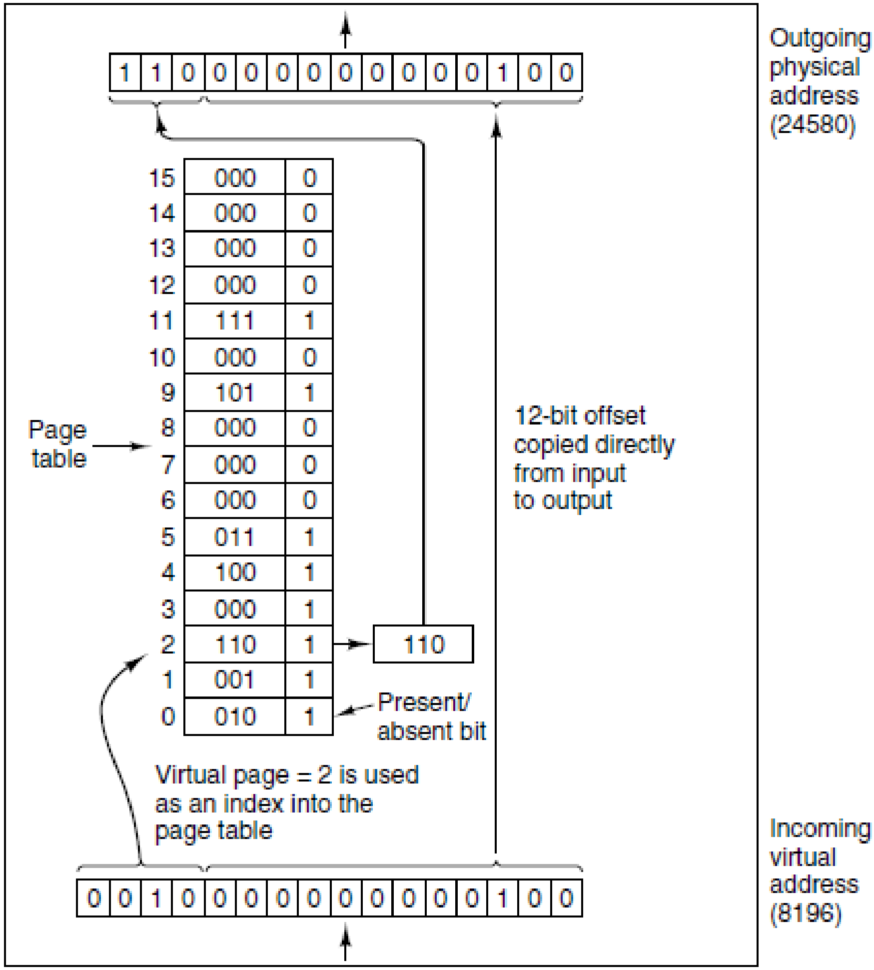
- The relation between virtual addresses and physical memory addresses is given by the page table.
- Every page begins on a multiple of 4096 and ends 4095 addresses higher,
  - so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287
  - MOV REG, 8192
    MMU transforms to
    MOV REG, 24576
- If the MMU notices that the page is unmapped (call page fault), OS picks a little used page and write it to the disk then fetches the missed page



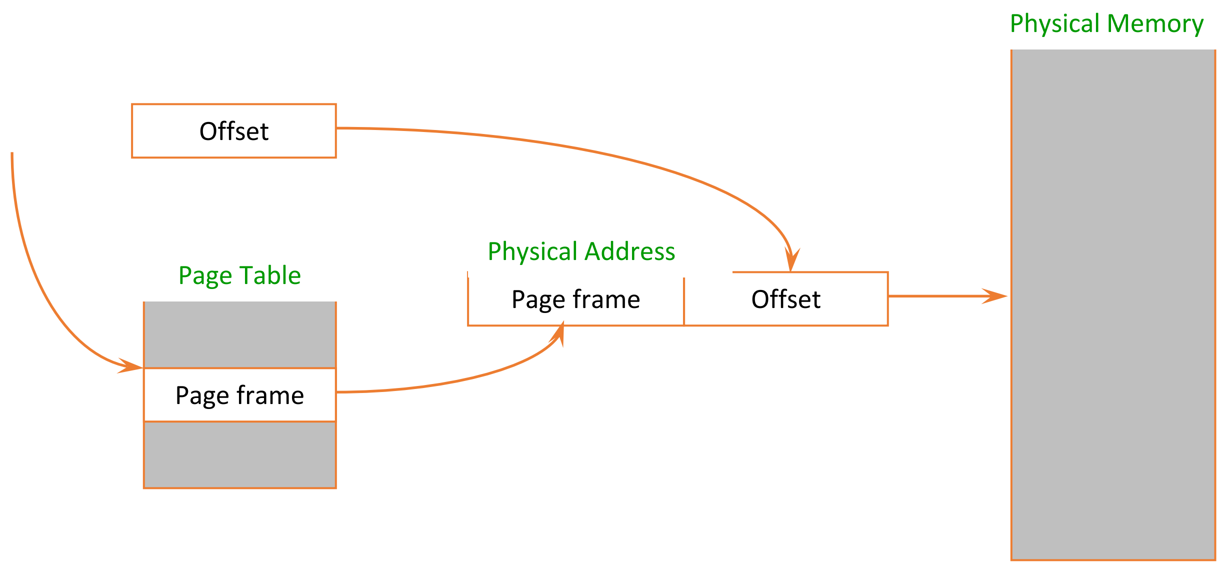
Page translation

- Translating addresses
  - Virtual address has two parts: virtual page number and offset
  - Virtual page number (VPN) is an index into a page table
  - Page table determines page frame number (PFN)
  - Physical address is PFN::offset
- Page tables
  - Map virtual page number (VPN) to page frame number (PFN)
    - VPN is the index into the table that determines PFN
  - One page table entry (PTE) per page in virtual address space
    - Or, one PTE per VPN

Page lookups

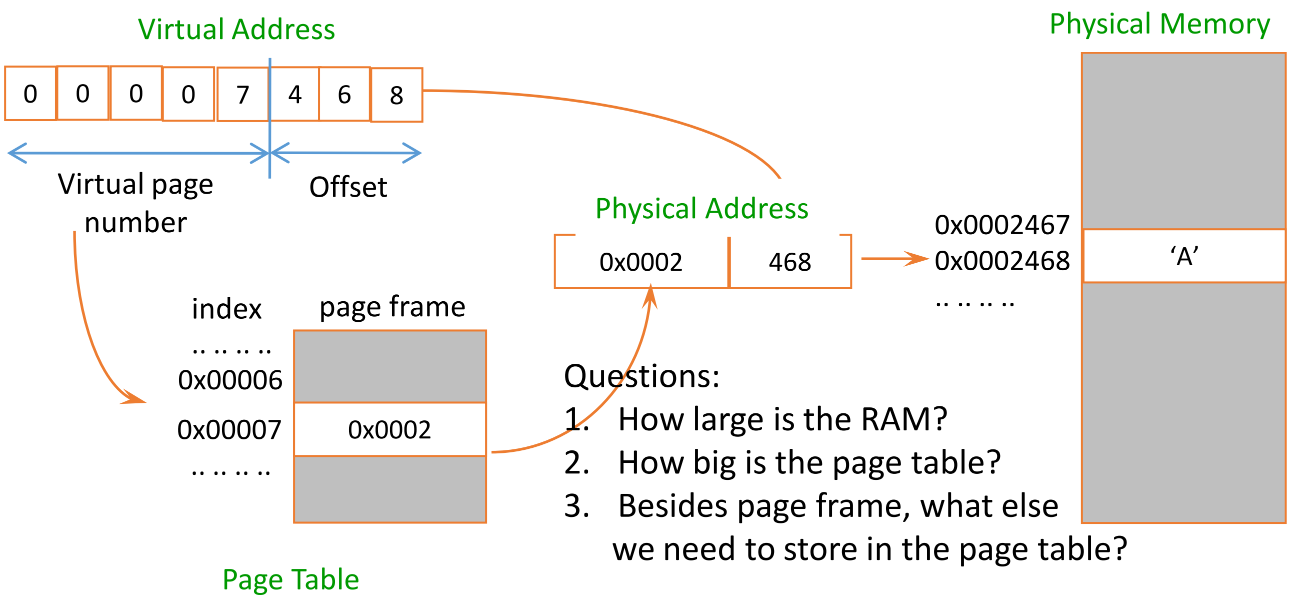


Page lookups



Page lookups example

Example: how do we ‘load 0x00007468’?

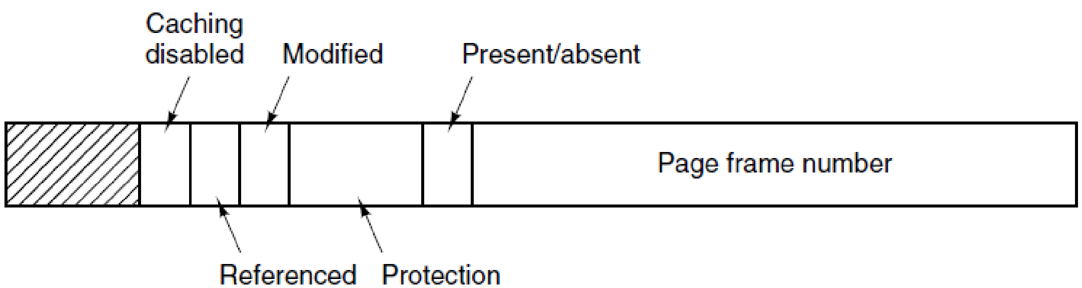


Questions:
1. How large is the RAM?
2. How big is the page table?
3. Besides page frame, what else we need to store in the page table?

Page tables

- A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses
- Operating system maps the virtual address provided by the process to the physical address of the actual memory where that data is stored
- The page table is where the operating system stores its mappings of virtual addresses to physical addresses, with each mapping also known as a *page table entry*
- Each process has it own page table

Structure of a Page Table Entry

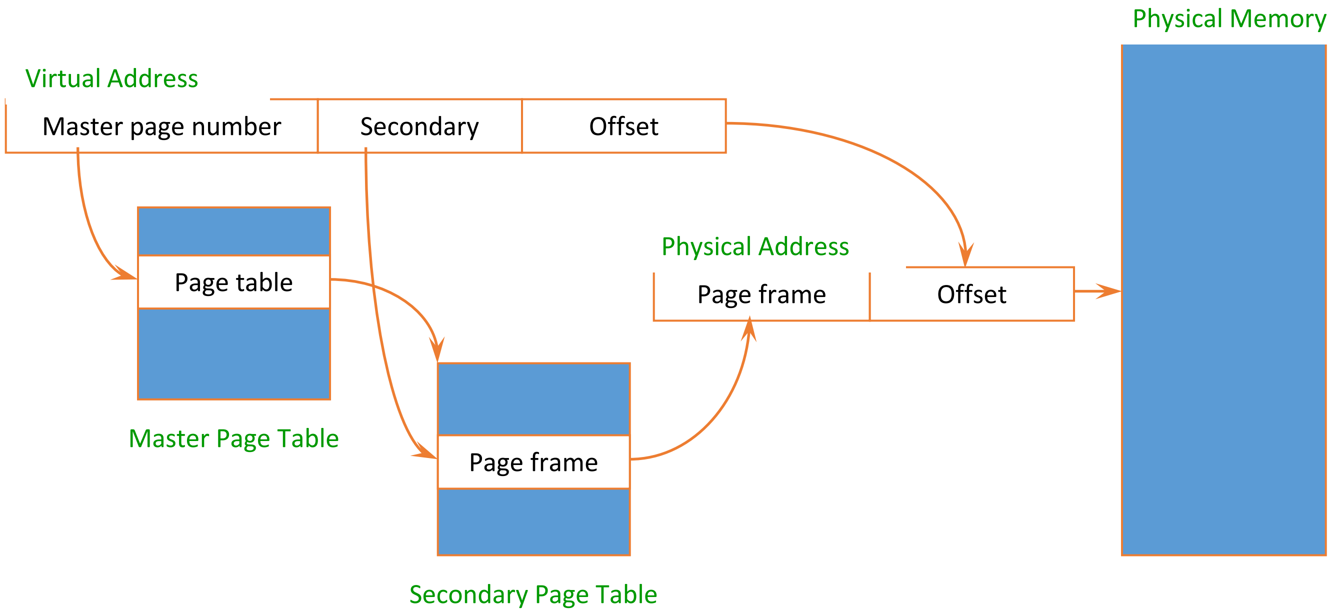


- The Reference bit says whether the page has been accessed
  - It is set when a read or write to the page occurs
  - The value is used to help OS choose a page to evict when a page fault occurs, pages that are not being used are better candidates to be replaced
- The Modify bit says whether or not the page has been written
  - It is set when a write to the page occurs
- The Protection bits say what operations are allowed on page
  - 0 for Read/write and 1 for read only
  - Can be 3 bits for enabling read/write/execution
- The Present/absent bit says whether or not the PTE can be used
  - It is checked each time the virtual address is used
  - 0 virtual page is not currently in memory, 1 the entry valid and can be used
- The page frame number (PFN) determines physical page

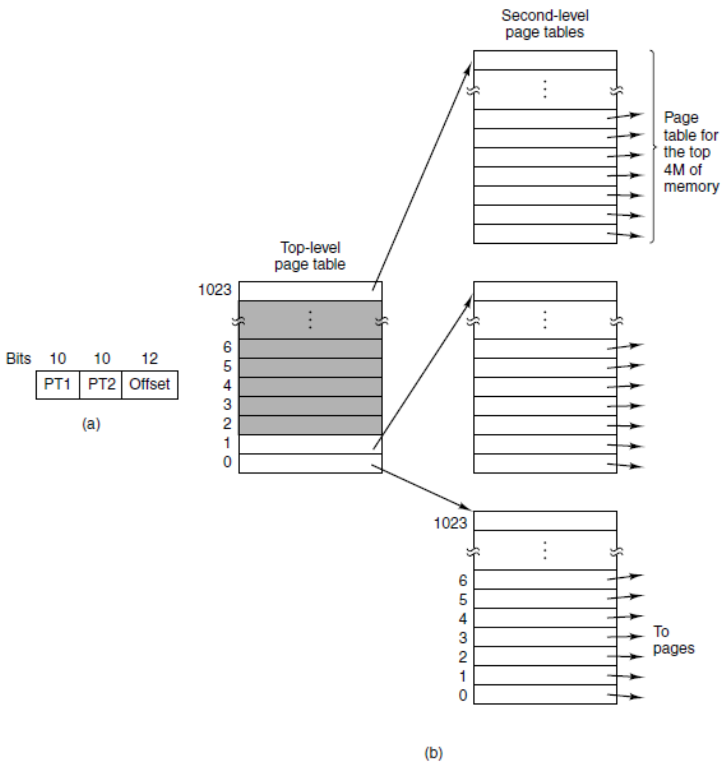
Multi-level page table

- Single level page table size is too large
  - 4KB page, 32 bit virtual address, 1M entries per page table!
- Two-level page tables
  - Virtual addresses (VAs) have three parts:
    - Master page number, secondary page number, and offset
  - Master page table maps VAs to secondary page table
  - Secondary page table maps page number to physical page
  - Offset indicates where in physical page address is located
- Example
  - 4K pages, 4 bytes/PTE
  - How many bits in offset? 4K = 12 bits
  - Want master page table in one page: 4K/4 bytes = 1K entries
  - Hence, 1K secondary page tables. How many bits?
  - Master (1K) = 10, offset = 12, inner = 32 – 10 – 12 = 10 bits

Two-Level page tables



Two-Level page tables



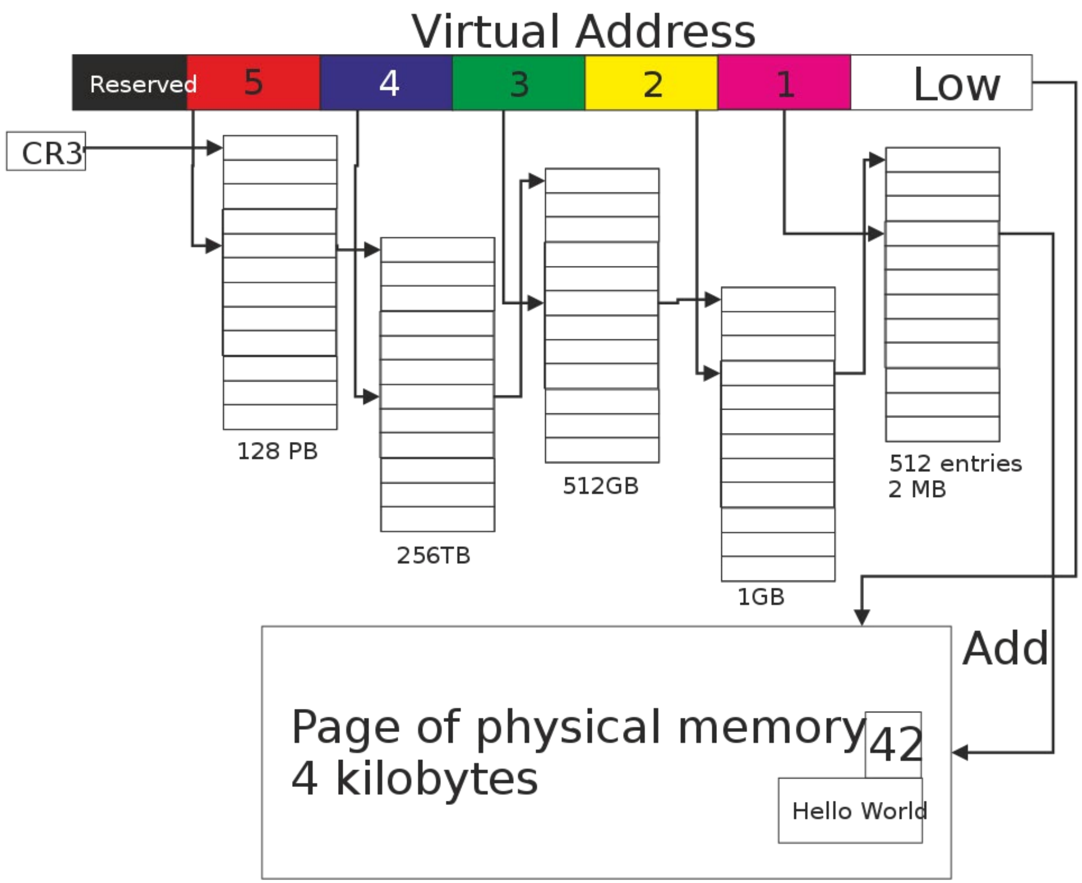
(a) A 32-bit address with two page table fields

(b) Two-level page tables

What is the problem with 2-level page table?

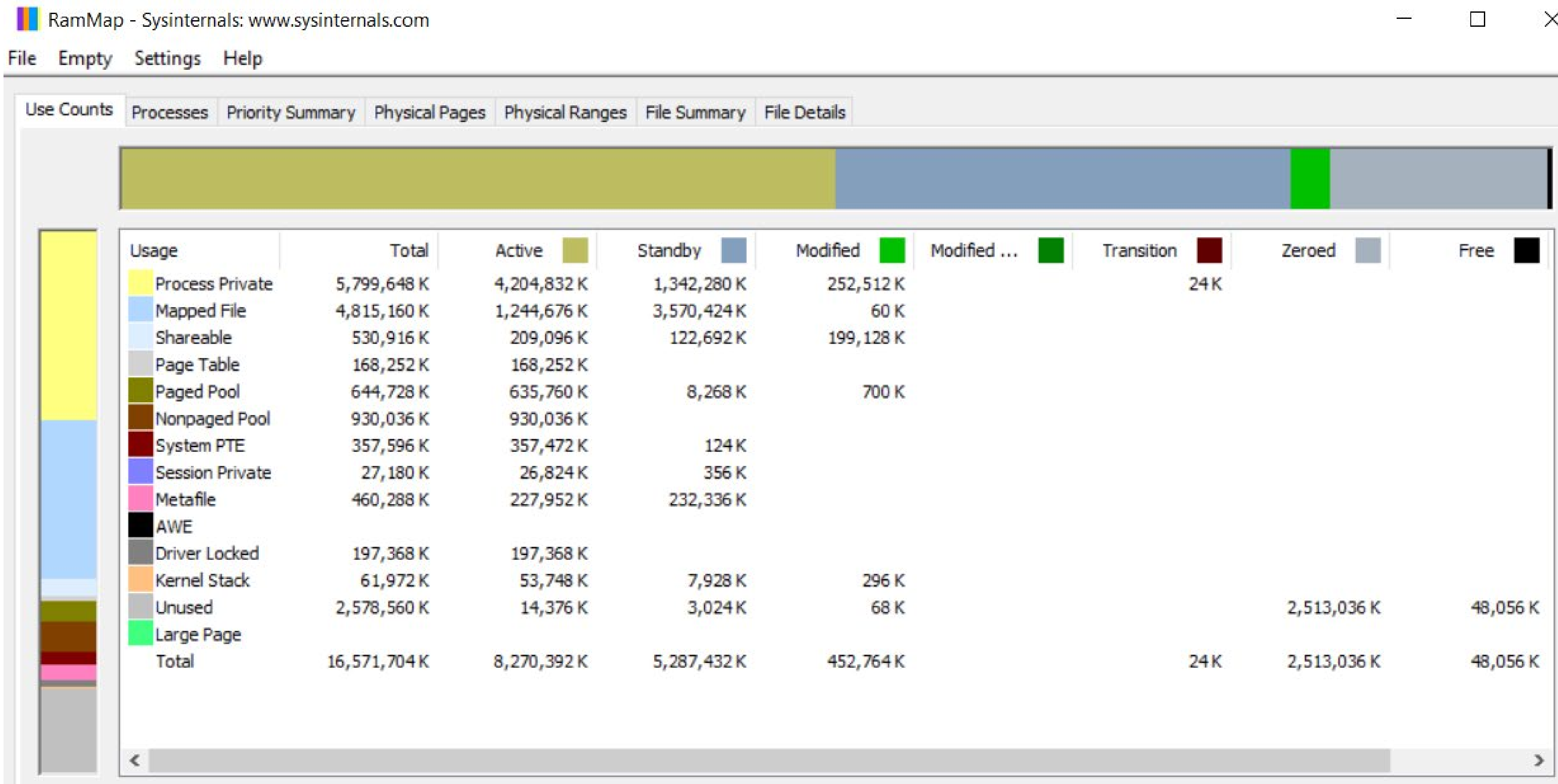
- Hints:
  - Programs only know virtual addresses
  - Each virtual address must be translated
    - Each program memory access requires several actual memory accesses

Intel 5-level Paging in X86-64 Processor



https://en.wikipedia.org/wiki/Intel_5-level_paging#/media/File:Page_Tables_(5_levels).svg

RamMap



Usage	Total	Active	Standby	Modified	Modified ...	Transition	Zeroed	Free
Process Private	5,799,648 K	4,204,832 K	1,342,280 K	252,512 K		24 K		
Mapped File	4,815,160 K	1,244,676 K	3,570,424 K	60 K				
Shareable	530,916 K	209,096 K	122,692 K	199,128 K				
Page Table	168,252 K	168,252 K						
Paged Pool	644,728 K	635,760 K	8,268 K	700 K				
Nonpaged Pool	930,036 K	930,036 K						
System PTE	357,596 K	357,472 K	124 K					
Session Private	27,180 K	26,824 K	356 K					
Metafile	460,288 K	227,952 K	232,336 K					
AWE								
Driver Locked	197,368 K	197,368 K						
Kernel Stack	61,972 K	53,748 K	7,928 K	296 K				
Unused	2,578,560 K	14,376 K	3,024 K	68 K			2,513,036 K	48,056 K
Large Page								
Total	16,571,704 K	8,270,392 K	5,287,432 K	452,764 K		24 K	2,513,036 K	48,056 K

Paging advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
- Easy to swap out chunks of a program
  - All chunks are the same size
  - Use valid bit to detect references to swapped pages
  - Pages are a convenient multiple of the disk block size

Paging limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of a page
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
    - Even more for two-level page tables!
  - Solution – use a hardware cache of lookups
- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
    - Remember: each process has its own page table!
  - Solution – 2-level page tables

What if a process requires more memory than physical memory?

- Swapping
  - Move one/several/all pages of a process to disk
    - Free up physical memory
    - “Page” is the unit of swapping
  - The freed physical memory can be mapped to other pages
  - Processes that use large memory can be swapped out (and later back in)

A variation of paging: Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
  - Virtual addresses become <segment #, offset>
  - Units of memory from user's perspective
- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table

Segment table

- Extensions
  - Can have one segment table per process
    - Segment #s are then process-relative
  - Can easily share memory
    - Put same translation into base/limit pair
    - Can share with different protections (same base/limit, diff prot)
- Problems
  - Large segment tables
    - Keep in main memory, use hardware cache for speed
  - Large segments
    - Internal fragmentation, paging to/from disk is expensive

Segmentation and Paging

- Can combine segmentation and paging
  - The x86 supports segments and paging
- Use segments to manage logically related units
  - Module, procedure, stack, file, data, etc.
  - Segments vary in size, but usually large (multiple pages)
- Use pages to partition segments into fixed size chunks
  - Makes segments easier to manage within physical memory
    - Segments become “pageable” – rather than moving segments into and out of memory, just move page portions of segment
  - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated
- Tends to be complex…

Summary

- Virtual memory
  - Processes use virtual addresses
  - OS + hardware translates virtual address into physical addresses
- Various techniques
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user's perspective
  - Combine paging and segmentation to get benefits of both

Sources and References

- Andrew S. Tanenbaum, Herbert Bos, Modern Operating 4th edition, Pearson, 2015
- Presentations by Ding Yuan, ECE Dept., University of Toronto
- Operating System Concepts 10th book official slides by Abraham Silberschatz, Greg Gagne, Peter B. Galvin