Chapter 3: Memory Management

3.1. Main Memory

GV: Nguyễn Thị Thanh Vân

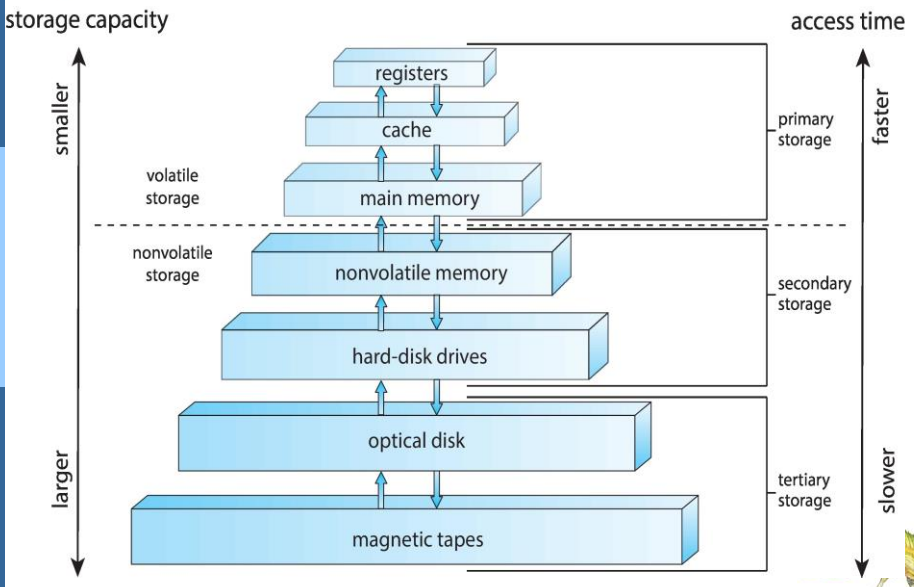
Outline

- Background
- Contiguous Memory Allocation
  - Fixed partition allocation
  - Variable Partition Allocation
- Non-contiguous Memory Allocation
  - Paging
  - Segmentation
  - Segmentation with paging
- Structure of the Page Table. Techniques:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables
- Swapping
- Example: The Intel 32 and 64-bit Architectures

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Storage-Device Hierarchy

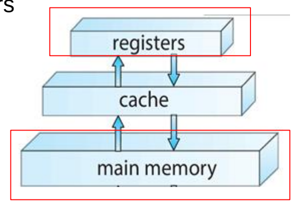


OS with memory management

- OS chế độ đa nhiệm nhằm nâng cao hiệu suất sử dụng CPU.
  - nảy sinh nhu cầu chia sẻ bộ nhớ giữa các tiến trình khác nhau .
  - Vấn đề nằm ở chỗ : « bộ nhớ thì hữu hạn và các yêu cầu bộ nhớ thì vô hạn ».
- OS chịu trách nhiệm cấp phát vùng nhớ cho các tiến trình có yêu cầu. Để thực hiện tốt nhiệm vụ này, OS cần phải xem xét nhiều khía cạnh :
  - Sự tương ứng giữa địa chỉ logic và địa chỉ vật lý (physic) :làm cách nào để chuyển đổi một địa chỉ tượng trưng (symbolic) trong chương trình thành một địa chỉ thực trong bộ nhớ chính?
  - Quản lý bộ nhớ vật lý: làm cách nào để mở rộng bộ nhớ có sẵn nhằm lưu trữ được nhiều tiến trình đồng thời?
  - Chia sẻ thông tin: làm thế nào để cho phép hai tiến trình có thể chia sẻ thông tin trong bộ nhớ?
  - Bảo vệ: làm thế nào để ngăn chặn các tiến trình xâm phạm đến vùng nhớ được cấp phát cho tiến trình khác?
- Các giải pháp quản lý bộ nhớ phụ thuộc rất nhiều vào đặc tính phần cứng và trải qua nhiều giai đoạn cải tiến để trở thành những giảp pháp khá ổn như hiện nay.

Background

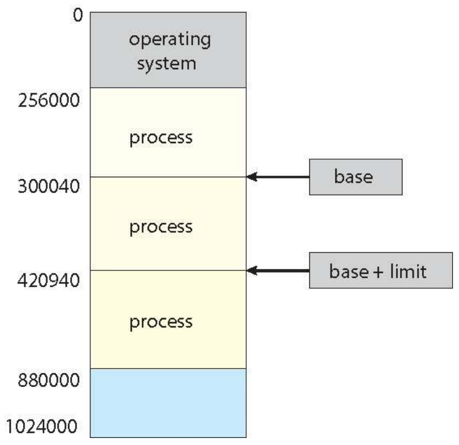
- Program is permanently kept on **backing store** (disks)
- When run: it must be brought from disk into memory and placed within a process
- CPU can access directly to main memory and registers
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests



- Register access is done in one CPU clock (or less)
- Main memory access can take many cycles, causing a **stall,** since it does not have the data required to complete the instruction that it is executing
  - Solution: add fast memory between the CPU and main memory, typically on the CPU chip for fast access
- **Cache** sits between main memory and CPU registers
- Protection of memory is required to ensure correct operation

Memory Protection

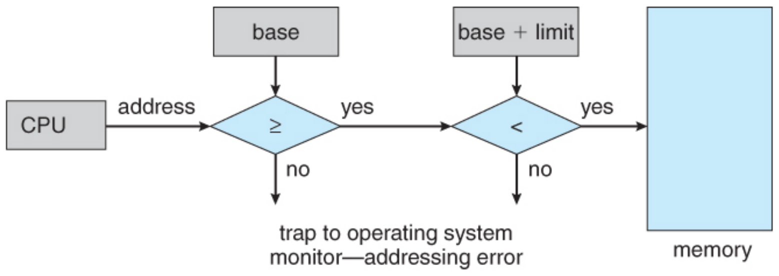
- Need to ensure that a process can access only access those addresses in it address space
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



the limit register is 120900

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
- Hardware address protection with base and limit registers:



- The OS load the base and limit registers
  - uses a special privileged instruction, executes only in kernelmode,
- OS can change the value of the registers but prevents user programs from changing the registers' contents

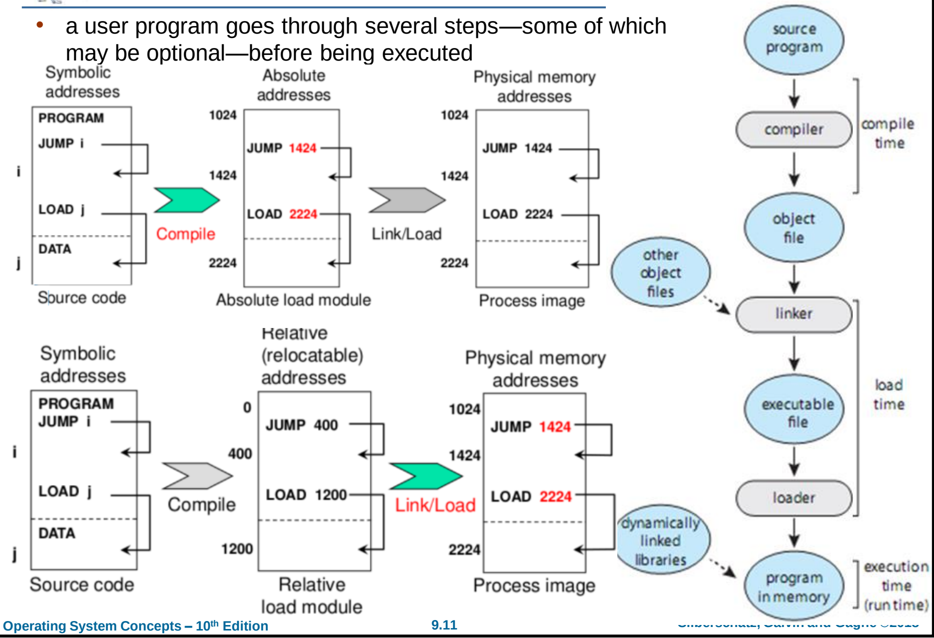
Address Binding

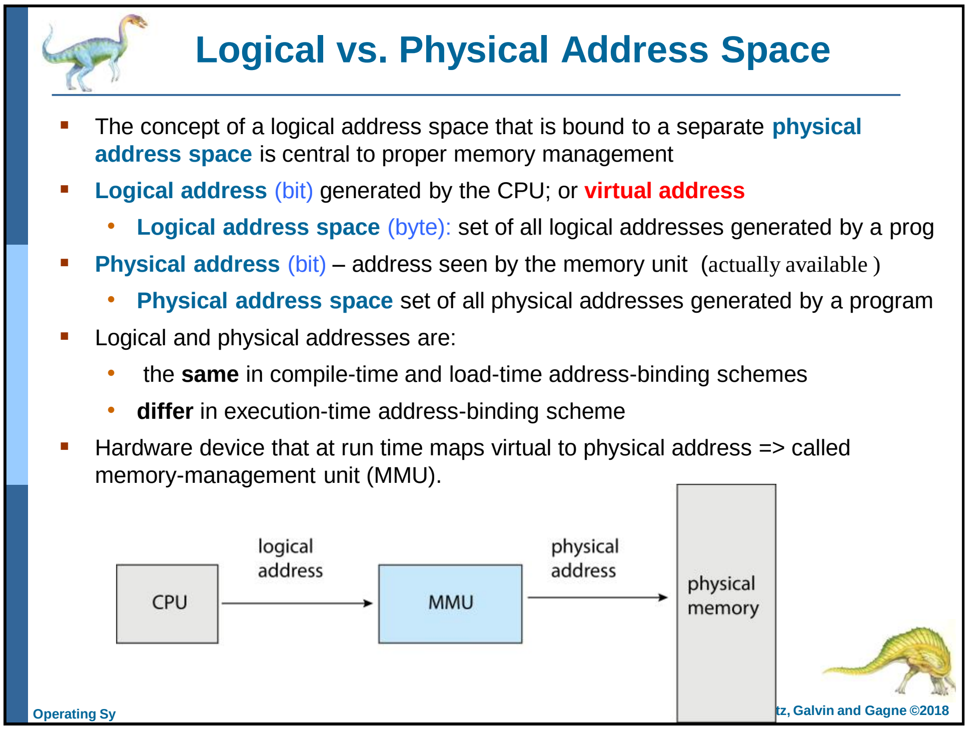
- Programs on disk, ready to be brought into memory to execute, are placed in an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?

  user program actually places a process in physical memory.
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses are usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another

Multistep Processing of a User Program

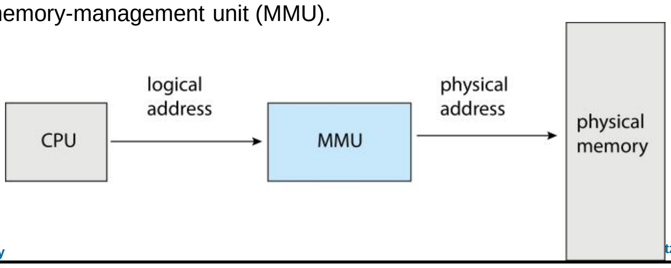
- a user program goes through several steps—some of which may be optional—before being executed

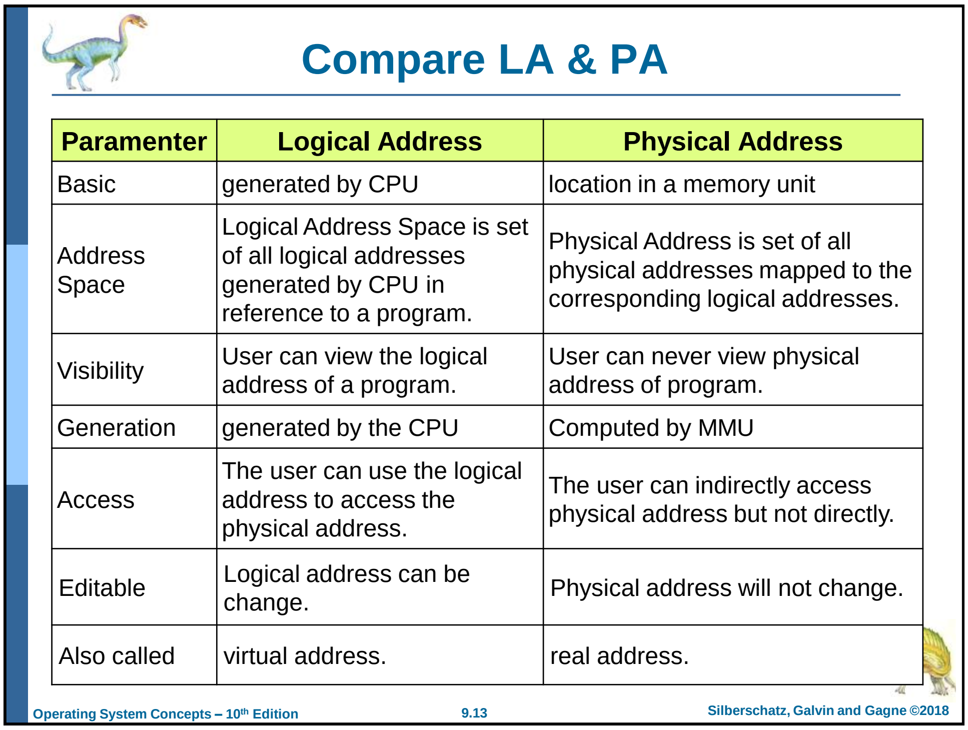




Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
- **Logical address** (bit) generated by the CPU; or **virtual address**
  - **Logical address space** (byte): set of all logical addresses generated by a prog
- **Physical address** (bit) – address seen by the memory unit (actually available)
  - **Physical address space** set of all physical addresses generated by a program
- Logical and physical addresses are:
  - the **same** in compile-time and load-time address-binding schemes
  - **differ** in execution-time address-binding scheme
- Hardware device that at run time maps virtual to physical address => called memory-management unit (MMU).



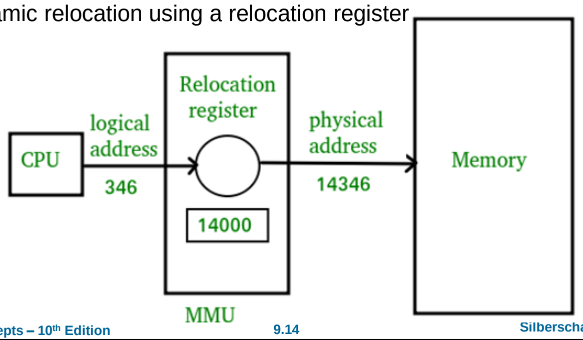


Compare LA & PA

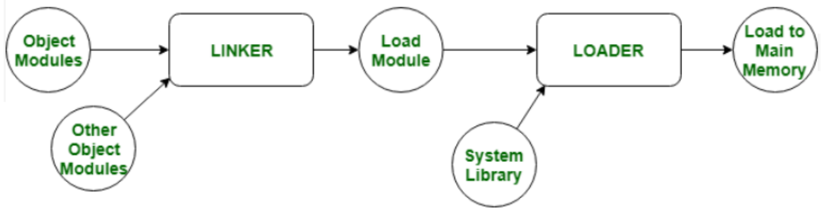
Paramenter	Logical Address	Physical Address
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.
Editable	Logical address can be change.	Physical address will not change.
Also called	virtual address.	real address.

Relocation Register

- A simple MMU scheme is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses
- Ex: Dynamic relocation using a relocation register



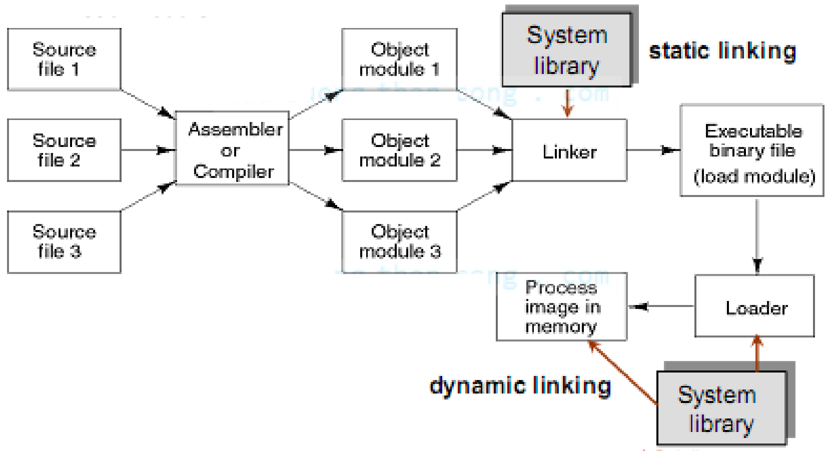
Execution of a program



- **Linking** and **Loading** are the utility programs that play a important role in the execution of a program.
  - Linking intakes the object codes generated by the assembler and combines them to generate the executable module.
  - Loading loads this executable module to the main memory for execution.

Loading

- Bộ linker:
  - Tái định vị địa chỉ tương đối và phân giải các external reference
  - Kết hợp các object module thành một file nhị phân khả thực thi gọi là **load module**



Dynamic Linking

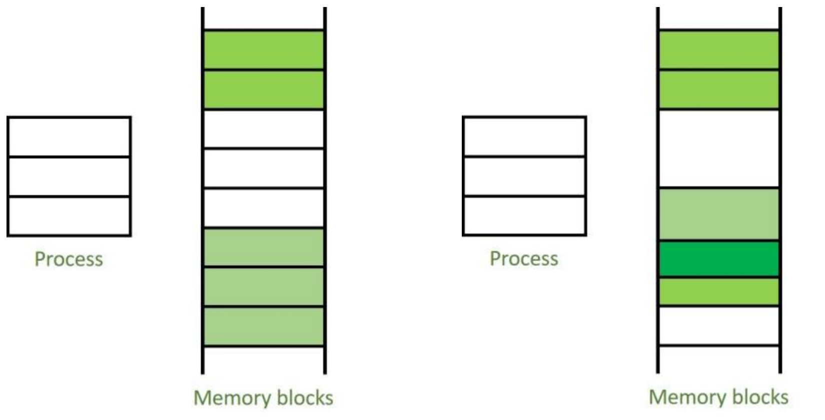
- **Static linking** – system libraries and program code combined by the loader into the binary program image.
- **Dynamic linking** – system libraries that are linked to user programs when the programs are run, is postponed until execution time
- Small piece of code, called **stub**, is used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries -
- System also known as **shared libraries**

Dynamic Loading

- The program consist of main part and a number of routines
- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- To obtain better memory-space utilization, we can use **dynamic loading.**
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Memory Allocation

- How to allocate available memory to the processes that are waiting to be brought into memory
- Two methods:



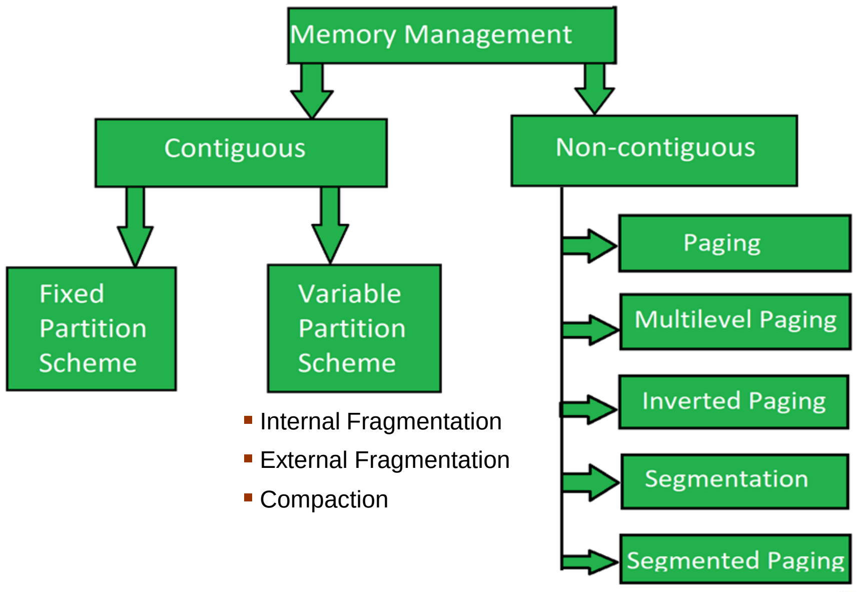
Contiguous Memory Allocation

Noncontiguous Memory Allocation

Memory Allocation

N.	Contiguous Memory Allocation	Non-Contiguous Memory Allocation
1.	Contiguous memory allocation allocates consecutive blocks of memory to a file/process.	Non-Contiguous memory allocation allocates separate blocks of memory to a file/process.
2.	Faster in Execution.	Slower in Execution.
3.	It is easier for the OS to control.	It is difficult for the OS to control.
4.	Overhead is minimum as not much address translations are there while executing a process.	More Overheads are there as there are more address translations.
5.	Internal fragmentation occurs in Contiguous memory allocation method.	External fragmentation occurs in Non-Contiguous memory allocation method.
6.	It includes single partition allocation and multi-partition allocation.	It includes paging and segmentation.
7.	Wastage of memory is there.	No memory wastage is there.
8.	In contiguous memory allocation, swapped-in processes are arranged in the originally allocated space.	In non-contiguous memory allocation, swapped-in processes can be arranged in any place in the memory.

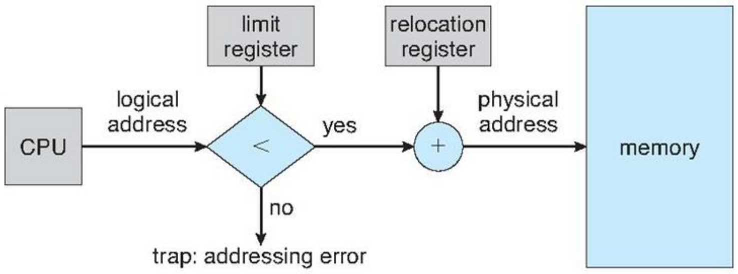
Types of memory management



- Internal Fragmentation
- External Fragmentation
- Compaction

Memory Protection

- Relocation registers used to protect user processes from each other, and from changing OS code and data
  - Relocation register contains value of **smallest** physical address
  - Limit register contains **range** of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically* by adding the value in the relocation register
  - Can then allow actions such as kernel code being **transient** and kernel changing size
- Hardware support for relocation and limit registers:



Memory Allocation - Contiguous allocation

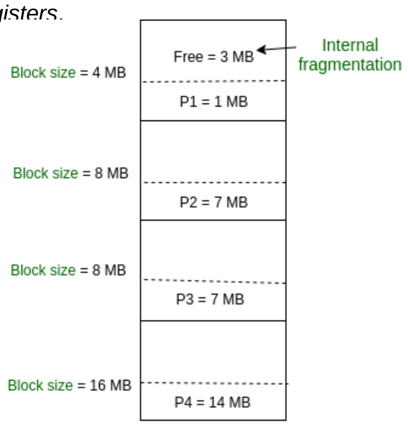
- **Contiguous allocation** is one early method
  - Main memory usually consists of two **partitions**:
    - Resident operating system, usually held in low memory with interrupt vector
    - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- 2 types:
  - Fixed partition allocation
  - Variable Partition Allocation

Fixed Partitioning Allocation

- Main memory is divided in to many partitions (same size or different size)
- The simplest technique used to put more than 1 processes in the main memory
  - In every partition only one process will be accommodated.
- Degree of multi-programming is restricted by number of partitions in the memory.
  - Maximum size of the process is restricted by maximum size of the partition.
- Every partition is associated with the *limit registers*.
- **Limit: Internal Fragmentation:**

  Any program, no matter how small, occupies an entire partition.

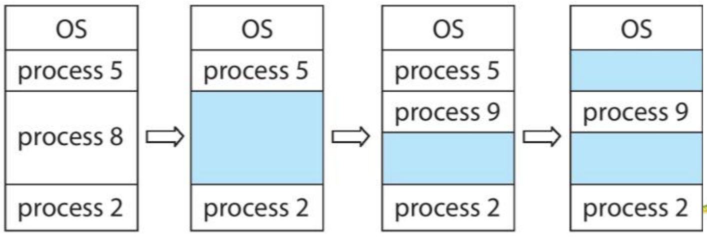
  => Main memory use is inefficient.



Sum of Internal Fragmentation in every block
(4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB

Variable Partition Allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  - (a) allocated partitions
  - (b) free partitions (holes)

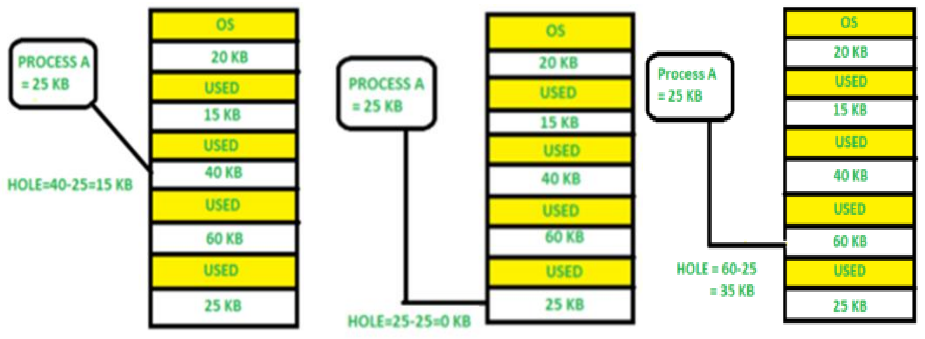


Dynamic Storage-Allocation Problem

- This problem concerns:
  - How to satisfy a request of size ***n*** from a list of free holes? …
- Solution for selecting a free hole from the set of available holes. Strategy:
  - **First-fit**: Allocate the ***first*** hole that is big enough
  - **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
    - Produces the largest leftover hole
  - **Next Fit**: similar to the first fit but it will search for the first sufficient partition from the last allocation point.
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- **Limit: External Fragmentation**

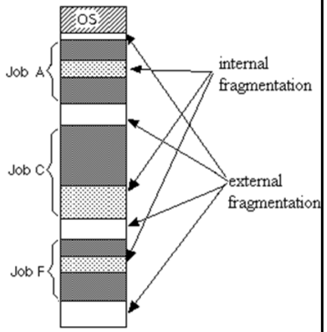
Ex

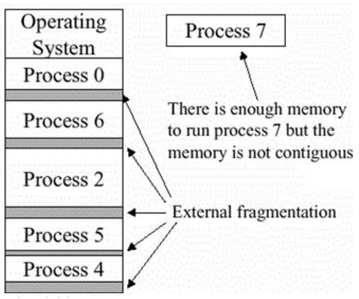
- **First-fit** **Best-fit** **Worst-fit**



Fragmentation

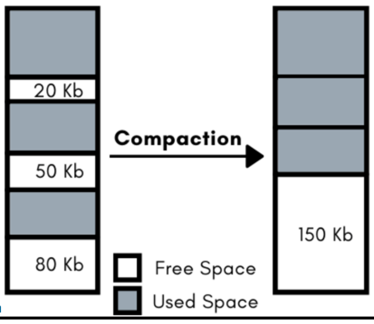
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- **External Fragmentation** – total memory space available to satisfy a request, but it is not contiguous
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation: 1/3 may be unusable -> **50-percent rule**
- Can reduce external fragmentation by **compaction**

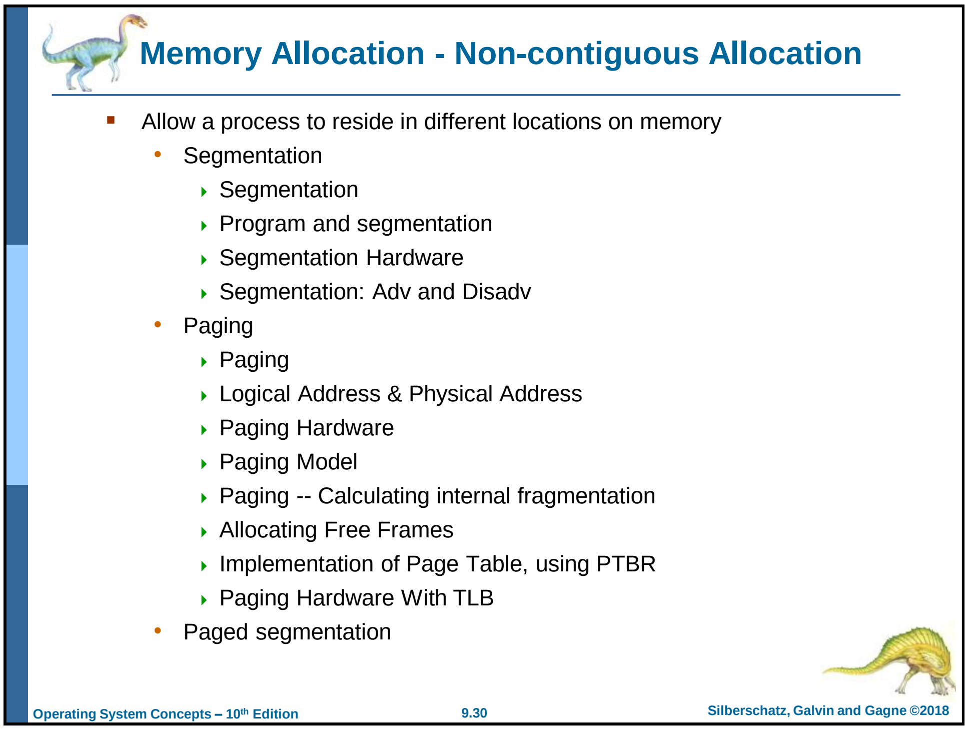




Compaction

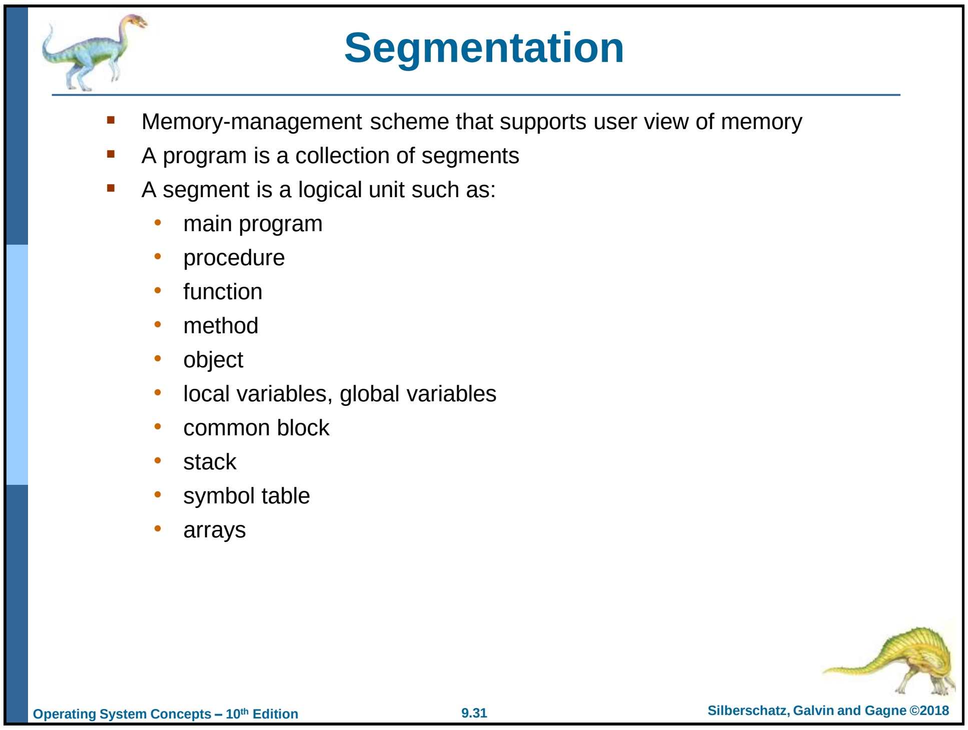
- One solution to the problem of **external fragmentation**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem. I/O done while compaction. Data arrives in wrong place.
- Solutions to I/O problem
  - Latch process in memory while it is involved in I/O
  - Do I/O only into OS buffers
- Now consider that **backing store** has same fragmentation problems but on disks





Memory Allocation - Non-contiguous Allocation

- Allow a process to reside in different locations on memory
  - Segmentation
    - Segmentation
    - Program and segmentation
    - Segmentation Hardware
    - Segmentation: Adv and Disadv
  - Paging
    - Paging
    - Logical Address & Physical Address
    - Paging Hardware
    - Paging Model
    - Paging -- Calculating internal fragmentation
    - Allocating Free Frames
    - Implementation of Page Table, using PTBR
    - Paging Hardware With TLB
  - Paged segmentation

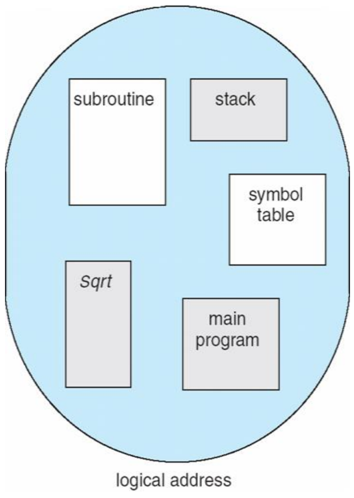


Segmentation

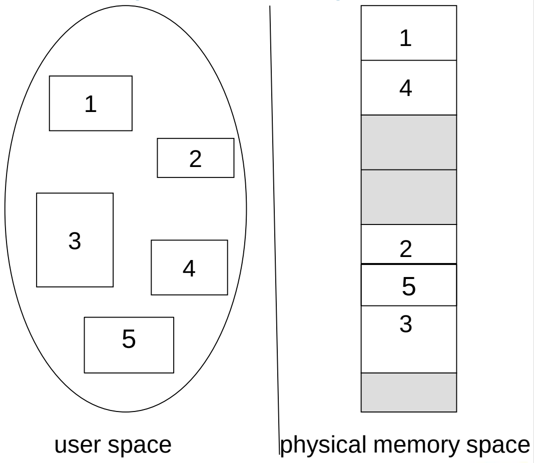
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
- A segment is a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays

Program and segmentation

User's View of a Program

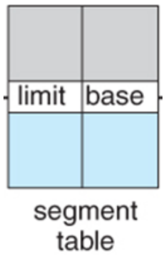


Logical View of Segmentation



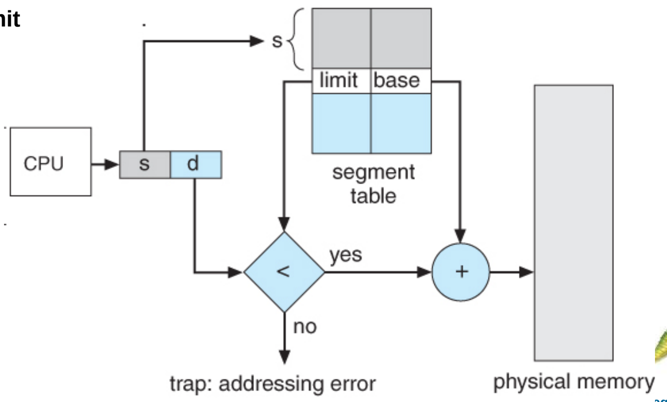
Segmentation Architecture

- **Segment table** – maps two-dimensional physical addresses; include:
  - **base** – contains the starting physical address where the segments reside in mem
  - **limit** – specifies the length of the segment
- **Segment-table base register** (**STBR**) points to the segment tables location in mem
- **Segment-table length register** (**STLR**) indicates number of segments used by a program.
- A logical address: <s,d>
  - s - segment number : index in segment table
  - d: an offset into that segment,
- Note:
  - Segment number ***s*** is legal if ***s*** < **STLR**
  - an offset is legal: **d < limit**



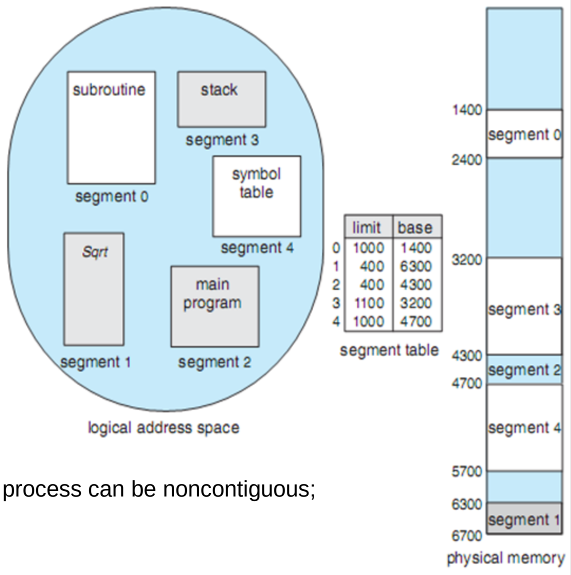
Segmentation Hardware

- A logical address consists of two parts: **<segment-number, offset>**
- an offset into that segment, d - be between 0 and the segment limit
  - If it is not, we trap to the OS
  - an offset is legal, it is added to the segment base to produce the address in physical memory: **correct offset+ base (in table segment)**
    - **base** is mapped from **s**
    - **d** < **limit**



Segmentation, ex

- Logical Addr (s,d), **check**:
  - s in seg table, d<limit
- Seg2=400bytes, at 4300
- Thus: CPU want to
  - A ref to byte 53 of seg 2 (s=2,d=53) is mapped onto location: 4300 + 53 = 4353
  - A ref to byte 852 of seg 3 (s=3, d=852) is mapped onto location: 3200 + 852 = 4052
  - A ref to byte 1222 of seg 0: would result in a trap to the OS, as this segment is only 1,000 bytes



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

- Physical address space of a process can be noncontiguous;
- Varying size memory chunks
- External fragmentation

Segmentation: Adv and Disadv

- **Advantages of Segmentation**
  - Segmentation is more close to the programmer's view of physical memory.
  - Segmentation prevents internal fragmentation.
  - Segmentation prevents the CPU overhead as the segment contain an entire module of at once.
- **Disadvantages of Segmentation**
  - The segmentation leads to **external fragmentation**.

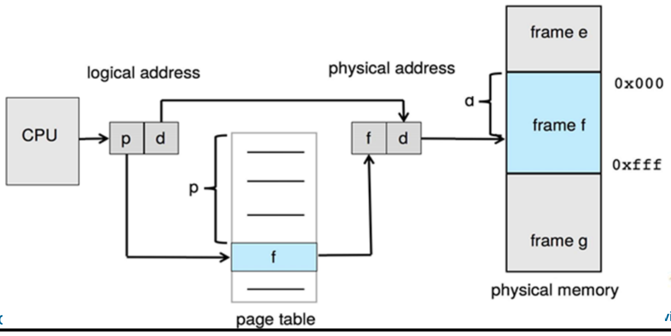
Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids **external fragmentation**
  - Avoids problem of varying sized memory chunks
- **Paging** is implemented through cooperation between the operating system and the computer hardware
- Implementing paging involves:
  - breaking physical memory into fixed-sized blocks called **frames** and
  - breaking logical memory into blocks of the same size called **pages**
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Backing store likewise split into **blocks**
- Still have **Internal fragmentation**

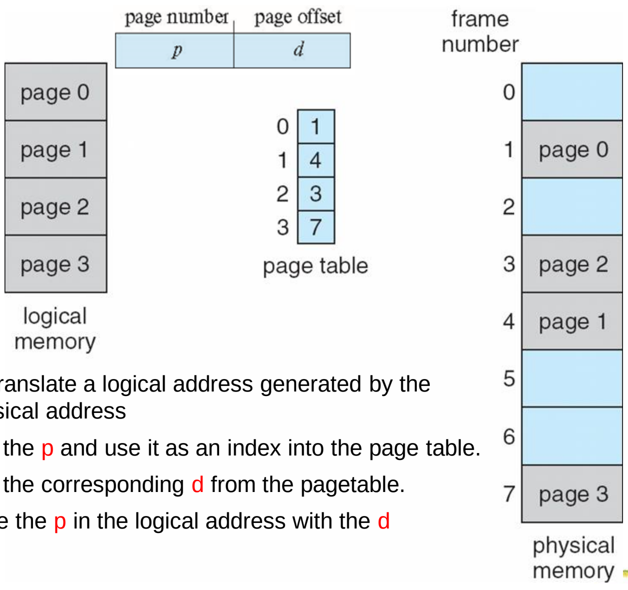
Paging Hardware

- Set up a page table to translate logical to physical addresses. Table contains:
  - The **base address** of each frame in physical memory.
  - The **offset** is the location in the frame being referenced
- Every address (logic) generated by the CPU is divided into two parts: p & d
  - **Page number** (p) – used as an index into a page table which contains base address of each page in physical memory
  - **Page offset** (d) – combined with **base address** to define

  the physical memory address that is sent to the memory unit



Paging Model of Logical and Physical Memory



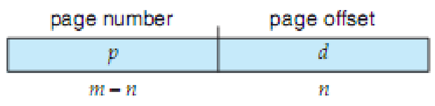
- The MMU to translate a logical address generated by the CPU to a physical address
  - 1. Extract the p and use it as an index into the page table.
  - 2. Extract the corresponding d from the pagetable.
  - 3. Replace the p in the logical address with the d

Logical Address & Physical Address

- Logical Address
  - LA Space -> blocks (pages), is divided into 2 parts p&d,
    - **Page number** (p) an index contains base address
    - **Page offset** (d) combined with **base address** => physical memory address
- Physical Address
  - PA Space -> blocks (frames), is divided into into 2 parts f&d
    - **Frame number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number.
    - **Frame offset(d):** Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame
- Ex:
  - If LA= 31bit, then LA Space = 2^{31}words = 2GB words (1G=2^{30})
  - If LA Space = 128 MB words = $2^7 * 2^{20}$ words, then LA= $\log_2 2^{27}$ = 27 bits
  - If PA = 22 bit, then PA Space = 2^{22} words = 4 MB words (1M = 2^{20})
  - If PA Space = 16 MB words = $2^4 * 2^{20}$ words, then PA = $\log_2 2^{24}$ = 24 bits

Logical address and logical address space

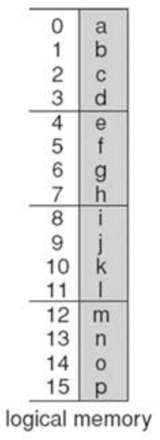
- For given logical address space size 2^m and page size 2^n bytes, we have



- Ex: In the logical address, n=2 and m=4

  => a page size: 2^n = 4 bytes (= frame sz)

  => Number of page: 2^{m-n} = 4



0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

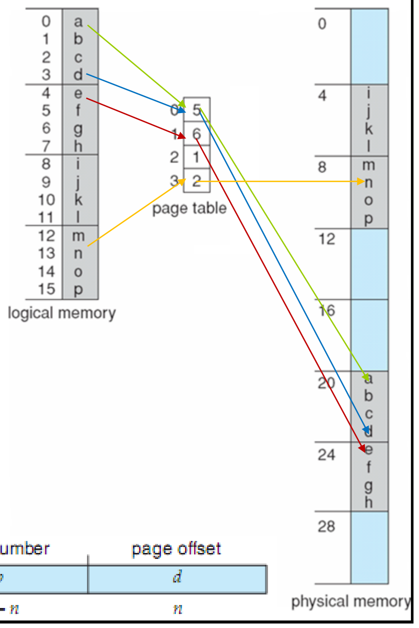
Paging Example

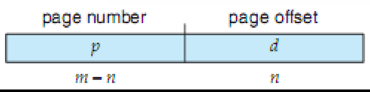
- Consider the memory: 1 page has x of size)
  - In the logical address, n=2 and m=4

  => a page size: 2^n = 4 bytes (= frame sz)

  => Number of page: 2^{m-n} = 4
  - A physical memory of 32 bytes: (sz=4)

    32/4=8 frames
- The programmer's view of memory can be mapped into physical memory:
  - LA 0 (page 0 (frame 5), offset 0) maps to PA: 20 [= (5 × 4) +0].
  - LA 3 (page 0 (frame 5), offset 3) maps to PA: 23 [= (5 × 4) +3].
  - LA 4 (page 1 (frame 6), offset 0) maps to PA: 24 [= (6 × 4) + 0].
  - LA 13 (page 3 (frame 2), offset 1) maps to PA: 9 [= (2 × 4) + 1



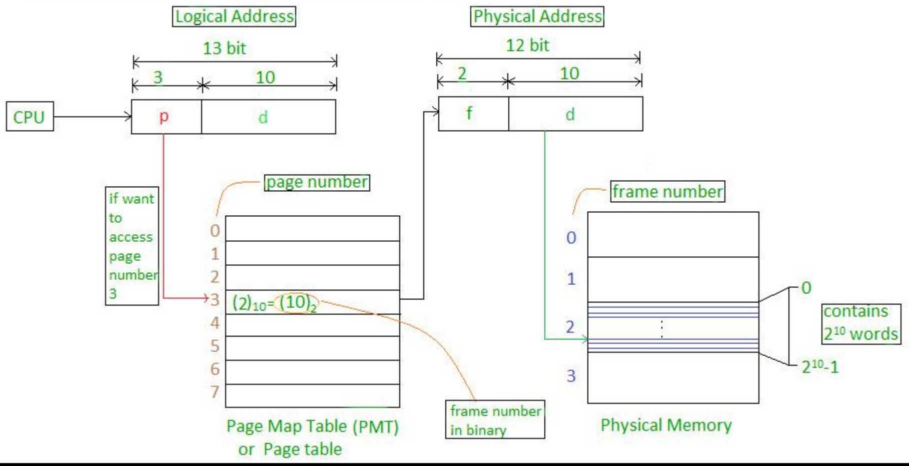


Ex

- Physical Address = 12 bits => Physical Address Space $=2^{12} = 4.2^{10}$ B =4K words
- Logical Address = 13 bits => Logical Address Space $=2^{13}=8.\ 2^{10}$ B =8 K words
- Page size = frame size = 1 K words (assumption)

Number of frames = Physical Address Space / Frame size = 4 K / 1 K = 4 = 2^2

Number of pages = Logical Address Space / Page size = 8 K / 1 K = 8 = 2^3

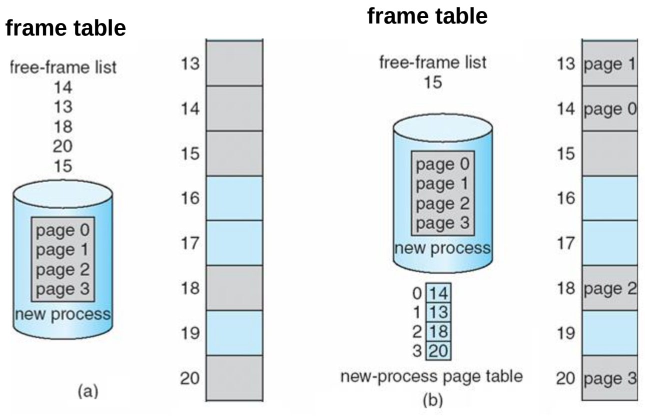


Paging -- Calculating internal fragmentation

- For example, if
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes (= 35*2048 + 1086)
- Then, will need: 35 pages + 1,086 bytes => will be allocated 36frames
- Resulting in Internal fragmentation of 2,048 - 1,086 = 962 bytes
- **Worst** case fragmentation: a process need n pages + 1byte
  - It would be allocated: n + 1 frames => internal fragmentation of almost 1 frame.
  - Ex: 35*2048 + 1 = 72765bytes
- On average fragmentation = 1 / 2 frame size
- This consideration suggests that small page sizes are desirable?
  - However, overhead is involved in each page-table entry, it reduced as the size of the pages increases
  - disk I/O is more efficient when the amount of data being transferred is larger
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two-page sizes – 8 KB and 4 MB

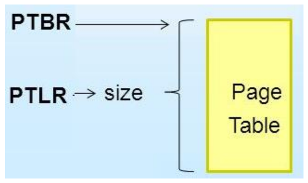
Allocating Free Frames

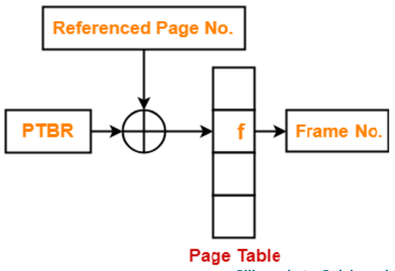
- OS is managing physical memory: Keep a list of free frames – called **frame table:**
  - has 1 entry for 1 physical page frame - indicating the latter is free or allocated
- Example of frame allocation
  - New process arrives: (a) before allocation (b) after allocation



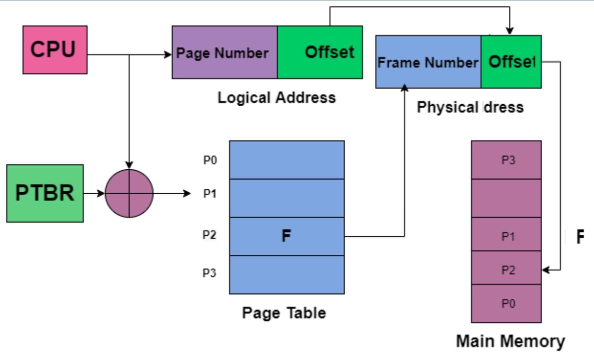
Implementation of Page Table

- The hardware implementation of the page table:
  - Use set of dedicated high-speed hardware registers (suitable **small page**)
  - makes the page-address translation very eficient.
  - However, it increases context-switch time, as each one of these registers must be exchanged during a context switch
- **Large page** table: page table is kept in main memory using
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
  - => reducing context-switch time: Changing page tables requires changing only this one register (PTBR)





Implementation of Page Table using PTBR



- In this scheme every data/instruction access requires two memory accesses
  - 1 for the page table: find the index into the page table by PTBR offset = pagenum
  - 1 for the data / instruction: the frame number & page offset = actual address

  => Run at half a speed!
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

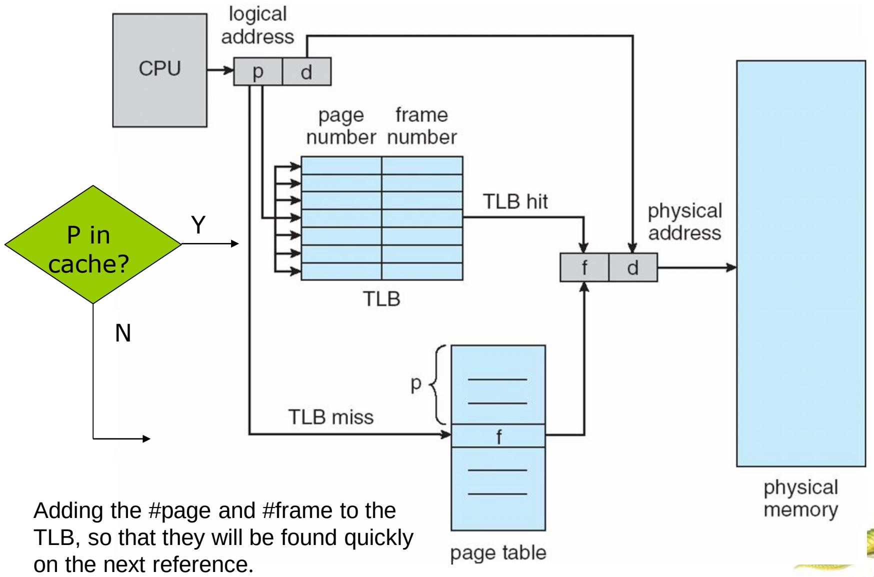
Associative Memory Hardware

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- Associative memory – parallel search: the item is compared with all keys
  - If the item is found, the corresponding value field is returned
- TLB:

Page #	Frame #

- In paging, TLBs are used to store the most recently accessed memory pages.
  - When the CPU generates an address, the page number of the address is compared with the elements in the TLBs,
  - Ex, Address translation (p, d)
    - If p is in associative register, get frame # out
    - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Adding the #page and #frame to the TLB, so that they will be found quickly on the next reference.

Translation Look-Aside Buffer - TLB

- The TLB contains only a few of the page-table entries, typically 32 -1,024 entries
- When a logical address is generated by the CPU,
  - the MMU checks if its page number is present in the TLB.
  - If the page number is found, its frame number is immediately available and is used to access memory (pipeline within the CPU)
  - If the page number is not in the TLB (**TLB miss**), addr translation proceeds
- **TLB miss**: **update TLB**: value is loaded into the TLB for faster access next time
  - Replacement policies must be considered when TLB is already full of entries:
    - an existing entry must be selected for replacement.
    - Range from least recently used (LRU) through round-robin to random.
  - Some entries can be **wired down** for permanent fast access
    - Means they cannot be removed from the TLB.
    - Typically, TLB entries for key kernel code are wired down.
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry:
  - identifies each process to provide address-space protection for that process
  - Otherwise need to flush the TLB at every context switch

Effective Access Time

- If the page is found in the TLB

  TLB_hit_time := TLB_search_time + memory_access_time
- If the page is not found in the TLB

  TLB_miss_time := TLB_search_time + memory_access_time (get p&f) + memory_access_time (get data)
- An average measure of the TLB performance: the Effective Access Time
  - **EAT := TLB_miss_time * (1- hit_ratio) + TLB_hit_time * hit_ratio.**

  ⇔ EAT := (TLB_search_time + 2*memory_access_time) * (1- hit_ratio) + (TLB_search_time + memory_access_time)* hit_ratio.
  - **Hit ratio** – percentage of times that a page number is found in the TLB
    - Ex: An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Ex: Suppose that it takes 10 nanoseconds to access memory. 80% hit ratio
  - If we find the desired page in TLB then a mapped-memory access take 10ns
  - Otherwise, we need two memory access, so it is 20ns
  - EAT = 0.80 x 10 + 0.20 x 20 = 12 ns => 20% slowdown in access time

Memory Protection

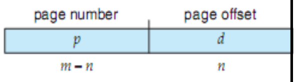
- Memory protection implemented by associating protection bit with each frame to indicate if access is allowed
- **Valid-invalid** bit attached to each entry in the page :
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page => allow access to the page
  - "invalid" indicates that the page is not in the process' logical address space => disallow access to the page
  - Or use **page-table length register** (**PTLR**)
- 1 entry:

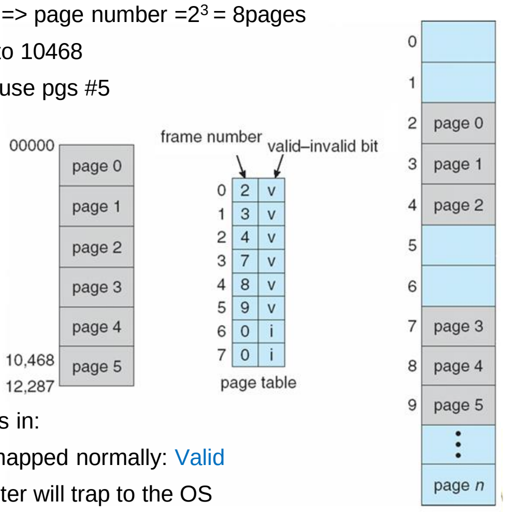
Page	Bit valid/invalid

- Any violations result in a trap to the kernel
- Can also add more bits to indicate if read-only, read-write, execute-only is allowed.

Valid (v) or Invalid (i) Bit In A Page Table

- a system with a 14-bit address space: 2^{14} = 16383 (m=14)
- Given a page size of 2KB=2*1024=2^{11}byte => d=11
- => p = Page number space = 14-11=3 => page number =2^3 = 8pages
- Ex, a program need only addresses 0 to 10468
  - 10468 > 2048*5=10240 => need use pgs #5





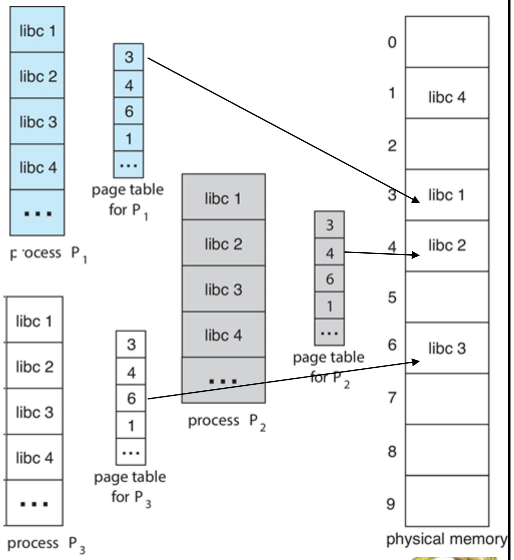
- Any attempt to generate an address in:
  - pages 0, 1, 2, 3, 4, and 5 are mapped normally: Valid
  - pages 6 or 7: invalid => computer will trap to the OS

Shared Pages

- An advantage of paging is the possibility of sharing common code
- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

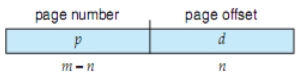
Shared Pages Example

- P1,P2,P3 sharing the pages for libc
- Each process has its own copy of registers and data storage
- Only 1 copy of the standard C library need be kept in physical memory,
- The page table for each user process maps onto the same physical copy of libc
- ➔ Saving!!!



Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Modern computer systems support a large 32-bit logical address space
    - 2^{32} physical page frames (m=32)



  - Page size of 4 KB (2^{12}Byte), d=12
  - => Page table would have 1 million entries: $2^{20} = 2^{32} / 2^{12}$ (p=32-12=20)
  - If each entry is 4 bytes ➔ each process requires: 2^{20} * 4bytes= 4MB of physical address space for the page table alone – **high cost**
    - Do not want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units.
    - We can accomplish this division in several ways

- The most common techniques for structuring the page table
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

The size of the page table

- The size of the page table depends upon the number of entries in the table and the bytes stored in one entry.
  - the number of entries are numbers of pages, ex 2^{20}

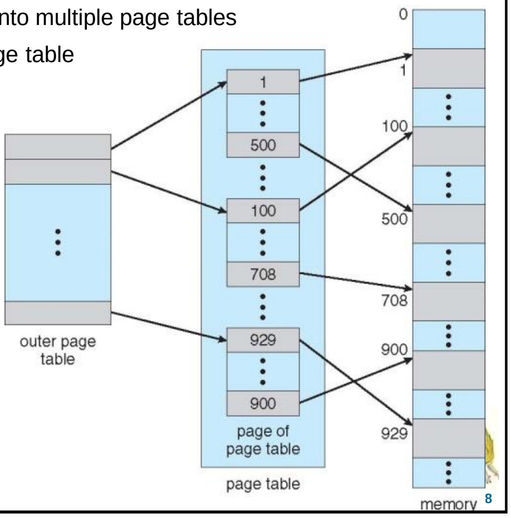
  (Number of pages is calculated by logical address space and Page size)
  - Give size of a page table entry, ex: 4 Byte

  Many fields in each page table entry
    - Page frame number (virtual addresses)
    - Page number (physical or real address)
    - Present/absent bit (is this page frame currently loaded?)
    - Protection bit (read/write vs. read only)
    - Dirty bit (has the data been modified?)
    - Referenced bit (has this page been recently referenced?)
    - Caching disabled bit (Can this data be cached?
  - Therefore, the size of the page table: 2^{20} * 4bytes= 4MB

  => each process requires 4MB of physical address space to store the page table alone (each process use 1 page table for mapping and store it on main mem)

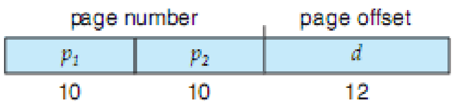
Hierarchical Page Tables

- Most modern computer systems support a large logical address space (2^{32} -> 2^{64}).
  - =>the page table itself becomes excessively large
- Noncontiguous => divide the page table into **smaller pieces**
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



Two-Level Paging Example

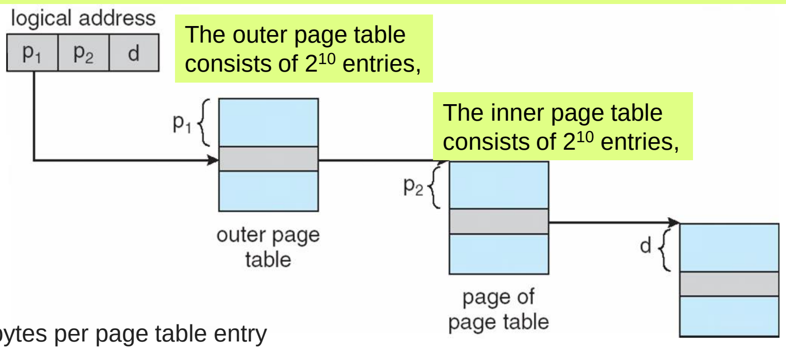
- A logical address (on 32-bit machine with 4KB page size) is divided into:
  - A page offset consisting of 12 bits (4KB=2^{12}byte => d=12)
  - A page number consisting of 20 bits (p=32-12)
- Since the page table is paged, the page number is further divided into:
  - A 10-bit page number (outer page)
  - A 12-bit page offset
- Thus, a logical address is as follows:



- where p_1 is an index into the outer page table, and p_2 is the displacement (moving) within the page of the inner page table
- Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**

Address-Translation Scheme

Address translation for a two-level 32-bit paging architecture with 4KB page size



The outer page table consists of 2^{10} entries,

The inner page table consists of 2^{10} entries,

4 bytes per page table entry

The size of first level (outer) page table = **2^{10} × 4 B = 4KB**

The size of each second level (inner) page table: **2^{10} × 4B= 4KB**

Each second level page table addresses 2^{10} entries and so it requires:

2^{10} second-level page tables

If a process size is 1GB (2^{30}B) need $2^{30}/2^{12}=2^{18}$pages use a two-level:

total memory requirement: **4KB + 4KB*** 2^{18}/**2^{10} = 260KB << 4MB -> save cost**

64-bit Logical Address Space

- A system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate.
- To illustrate, If page size is 4 KB (2^{12}), means d=12bit space for page_sz
  - Then page table has 2^{52} entries (p=64-12=52bit for space of #page)
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

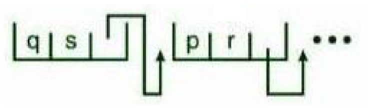
outer page	inner page	offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes (16GB)
    - And possibly 4 memory access to get to one physical memory location
  - Three-level Paging Scheme

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

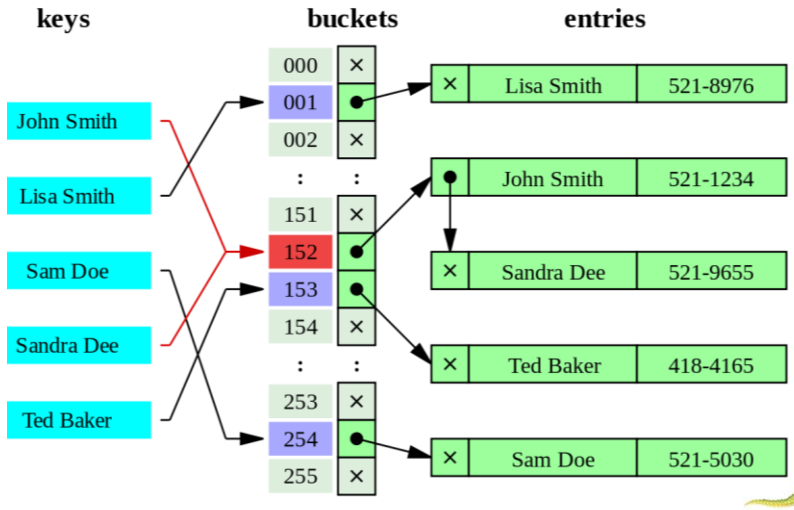
- Used in architecture with address spaces > 32 bits
- The virtual page number is **hashed into a page table**
  - Each entry in the hash table has a **linked list** of elements hashed to the same location (to avoid collisions – as we can get the same value of a hash function for different page numbers).
  - The hash value is virtual page number - is all the bits that are not a part of the page offset
- Each element contains
  1. The virtual page number
  2. The value of the mapped page frame
  3. A pointer to the next element



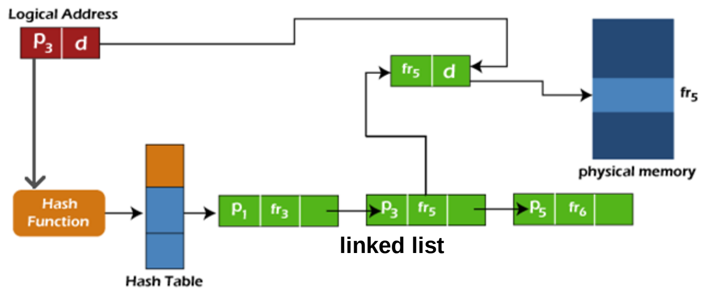
- Virtual page numbers are compared in this linked list searching for a match
  - If a match is found, the corresponding physical frame is extracted
  - Otherwise, subsequent entries in the linked list are checked until the virtual page number matches.

Expl: Hash Table Chaining

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value.



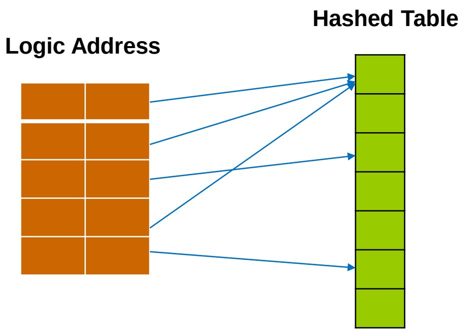
Hashed Page Table Hardware



- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number (p3) is compared with field 1 (p1) in the **first** element in the linked list
  - does not match the first element of the link list
  - move ahead and check the next element (P3): match
  - check the frame entry of the element, which is fr5.
  - append the offset provided (d) in the logical address to this frame number to reach the page's physical address.

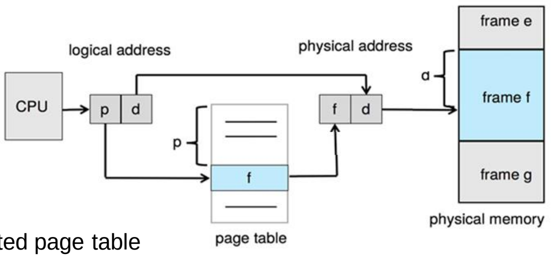
Clustered Hashed Page Tables

- Variation for 64-bit addresses uses clustered page tables - similar to hashed:
  - Hashed table: each entry refers to a single page-table entry
  - Clustered Hashed: each entry refers to **several** pages (such as 16), can store the mappings for **multiple physical-page frames**.
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)



Inverted Page Table

- Normally



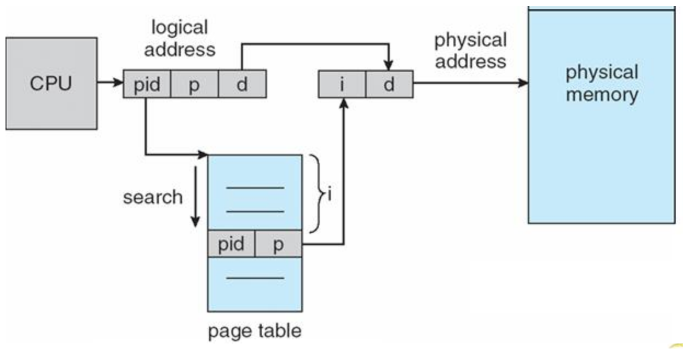
  - each process has an associated page table
  - each page table may consist of millions of entries.

  => consume large mounts of physical memory just to keep track of how other physical memory is being used

- Solution: Rather than having each process keep a page table and track of all possible logical pages, <u>track all physical pages</u> => **Inverted Page Table:**
  - Use only 1 IPT for all processes – 1 entry for each real page of memory
  - Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
  - Each inverted page-table entry is a pair **<process-id, page-number>**
    - process-id: role of the address-space identifier.

Inverted Page Table

- When a memory reference occurs
  - part of the virtual address, consisting of <**process-id, page-number**>, is presented to the memory subsystem.
  - The inverted page table is then searched for a match.
    - If a match is found at entry **i** then the physical address <**i, offset**> is generated.
    - If no match is found, then an illegal address access has been attempted.

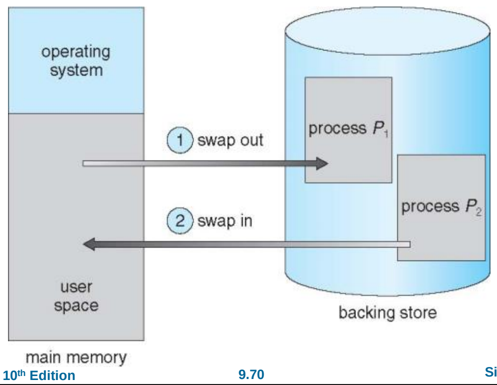


Inverted Page Table (Cont.)

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one (or at most a few) page-table entries
  - TLB can accelerate access (TLB is searched before the hash table is consulted)
- Implement shared memory in inverted page tables?
  - One mapping of a virtual address to the shared physical address
  - A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address

Swapping

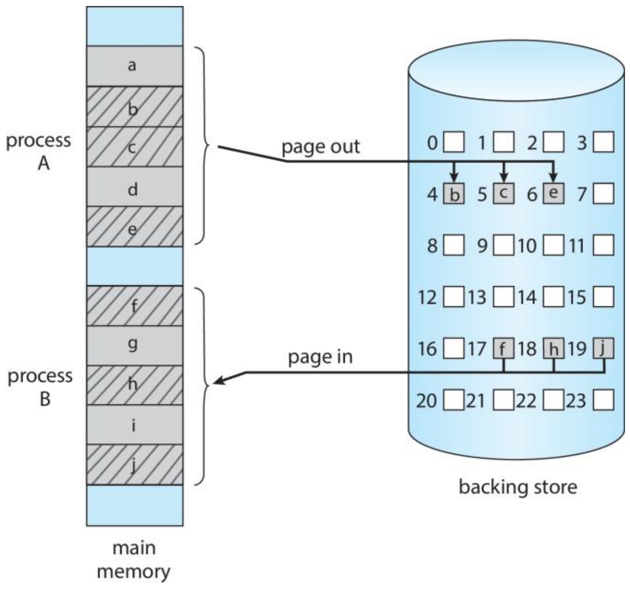
- Process instructions and their data must be in memory to be executed.
- However, a process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - increasing the degree of multiprogramming in a system
- **Backing store** is commonly fast secondary storage.
  - It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images.



Swapping

- Standard swapping is generally no longer used in contemporary OSs,
  - because the amount of time required to move entire processes between memory and the backing store is prohibitive.
  - Context switch time can then be very high
- a variation of swapping in which pages of a process can be swapped (no: an entire process)
  - still allows physicalmemory to be oversubscribed, but does not incur the cost of swapping entire processes, (only a small number of pageswill be involved in swapping.
  - A **page out** operation moves a page from memory to the backing store; the reverse process is known as a **page in**.
  - swapping with paging works well in conjunction with virtual memory.

Swapping with Paging

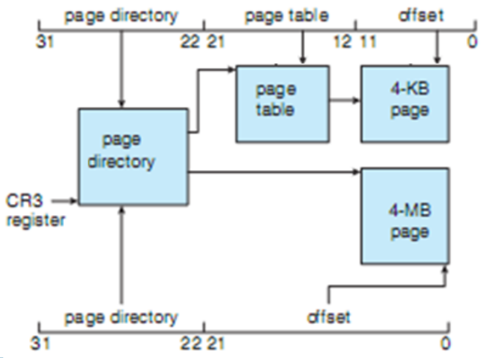


Swapping on Mobile Systems

- Not typically supported -- Flash memory based
  - Small amount of space
  - Limited number of write cycles
  - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

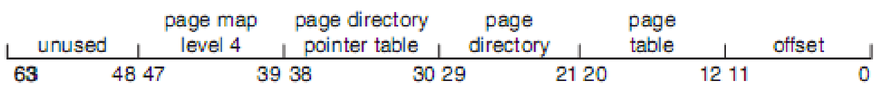
Example: Intel 32- and 64-bit Architectures

- Paging in the IA-32 architecture.
  - Address translation two-level (4KB)
  - One entry in the page directory is the Page Size flag, if set—indicates that the size of the page frame is 4 MB and not the standard 4 KB
    - the page directory points directly to the 4-MB page frame, bypassing the inner page table; and the 22 low-order bits in the linear address refer to the offset in the 4-MB page frame.



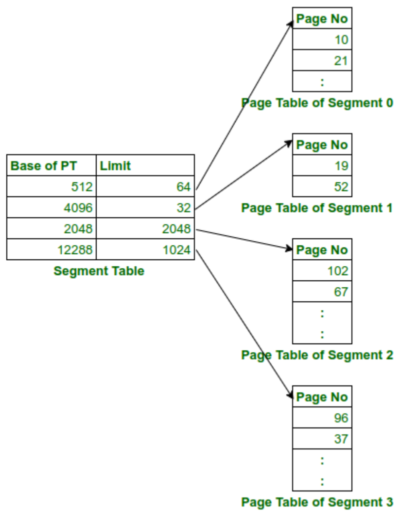
Example: Intel 32- and 64-bit Architectures

- Intel 64-bit Architectures based on the AMD's x86-64 architecture.
  - The x86-64 supported much larger logical and physical address spaces, as well as several other architectural advances.
  - Support for a 64-bit address space = 2^{64} bytes of addressable memory—a number greater than 16 quintillion (or 16 exabytes).
  - In practice far fewer than 64 bits are used for address representation in current designs.
  - The x86-64 architecture currently provides a 48-bit virtual address with support for page sizes of 4 KB, 2 MB, or 1 GB using 4 levels of paging hierarchy.
- x86-64 linear address.



Segmented Paging

- use segmentation along with paging to reduce the size of page table.
  - creating a page table for each segment.
  - hardware support is required.
  - The address provided by CPU will now be partitioned into segment no., page no. and offset.
- MMU) will use the segment table which will contain the address of page table(base) and limit.
- The page table will point to the page frames of the segments in main memory.



Base of PT	Limit
512	64
4096	32
2048	2048
12288	1024

Segment Table

Page No
10
21
:

Page Table of Segment 0

Page No
19
52

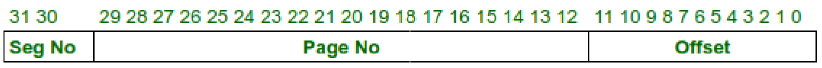
Page Table of Segment 1

Page No
102
67
:
:

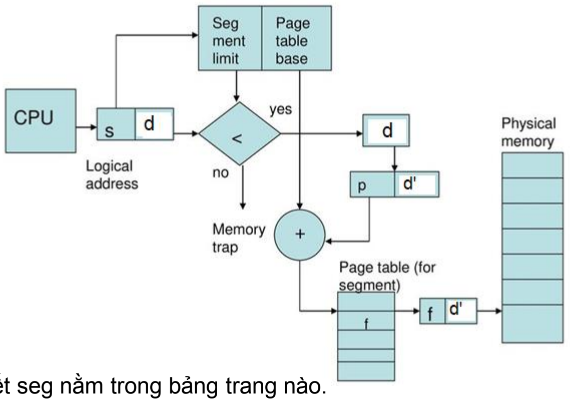
Page Table of Segment 2

Page No
96
37
:
:

Page Table of Segment 3



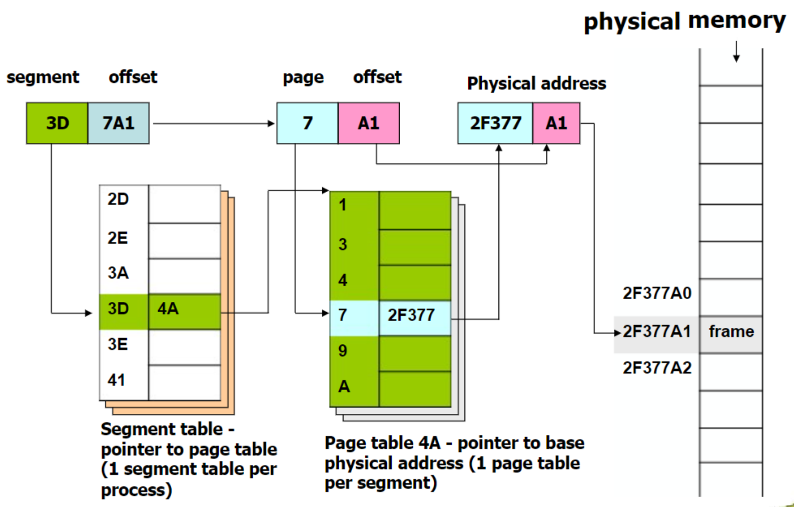
Segmented Paging Scheme



s: segment number
d: logical offset in segment
p: page
d’: offset in page
f: frame

- Xét đ/chi logic: (s,d)
  - tra **s** trong bảng seg để biết seg nằm trong bảng trang nào.
  - Offset **d** cho biết:
    - Số trang **p** trong bảng trang (đã đc xđinh từ s trong bảng seg);
    - Offset **d’** tương ứng với trang **p** trong bảng trang
  - Kết hợp offset **d’** (tách từ d), và frame **f** tại trang **p** trong bảng trang
  - => đ/chi vật lý (**f**,**d’**)

Segmented Paging, ex

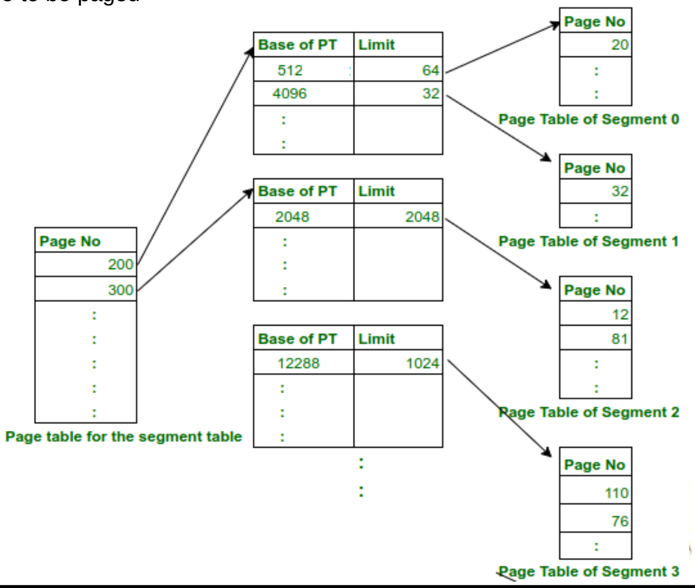

physical memory
segment
offset
3D
7A1
page
offset
7
A1
Physical address
2F377
A1
2D
2E
3A
3D
4A
3E
41
1
3
4
7
2F377
9
A
2F377A0
2F377A1
frame
2F377A2
Segment table - pointer to page table (1 segment table per process)
Page table 4A - pointer to base physical address (1 page table per segment)

Segmented Paging

- Advantages
  - Reduces external fragmentation
  - The page table size is reduced as pages are present only for data of segments, hence reducing the memory requirements.
  - the swapping out into virtual memory becomes easier .
- Disadvantages
  - Internal fragmentation still exists in pages.
  - Extra hardware is required
  - increasing the memory access time.
  - External fragmentation occurs

Paged Segmentation

- the segment table to be paged
- logical address
  - page no #1,
  - segment no,
  - page no #2
  - offset



Paged Segmentation

- Advantages of Paged Segmentation
  - No external fragmentation
  - Reduced memory requirements as no. of pages limited to segment size.
  - Page table size is smaller just like segmented paging,
  - Similar to segmented paging, the entire segment need not be swapped out.
- Disadvantages of Paged Segmentation
  - Internal fragmentation remains a problem.
  - Hardware is complexer than segmented paging.
  - Extra level of paging at first stage adds to the delay in memory access.

End of Chapter 3.1