Amdhal's Law

Amdahl's Law Identifies performance gains from adding additional cores to an application that has both serial and parallel components

S is serial portion

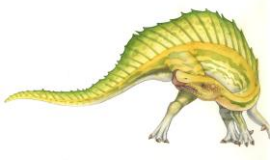
N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

If we have 2 cores ($p = 2$), the speedup would be:

$$S_{2 \text{ cores}} = \frac{1}{0.25 + \frac{0.75}{2}} = \frac{1}{0.625} \approx 1.6$$

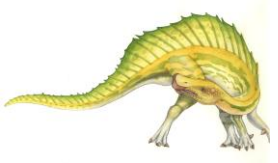


Amdhal's Law

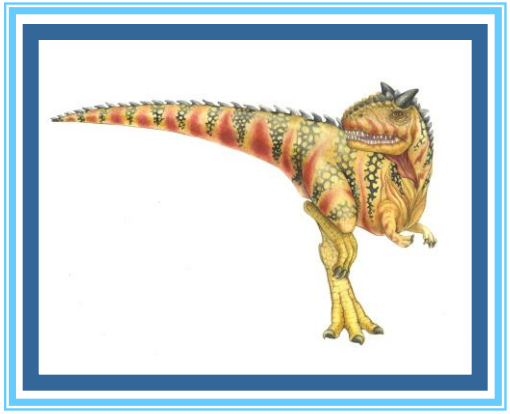
$$F_s = 0.20$$
$$F_p = 0.80$$
$$N = 5$$

$$\text{Speedup} = \frac{1}{(0.20 + \frac{0.80}{5})}$$
$$\text{Speedup} = \frac{1}{(0.20 + 0.16)}$$
$$\text{Speedup} = \frac{1}{0.36}$$
$$\text{Speedup} \approx 2.78$$

So, the potential speedup with 5 threads is approximately 2.78.

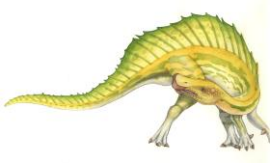


Chapter 7: Main Memory



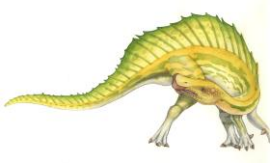
Chapter 7: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

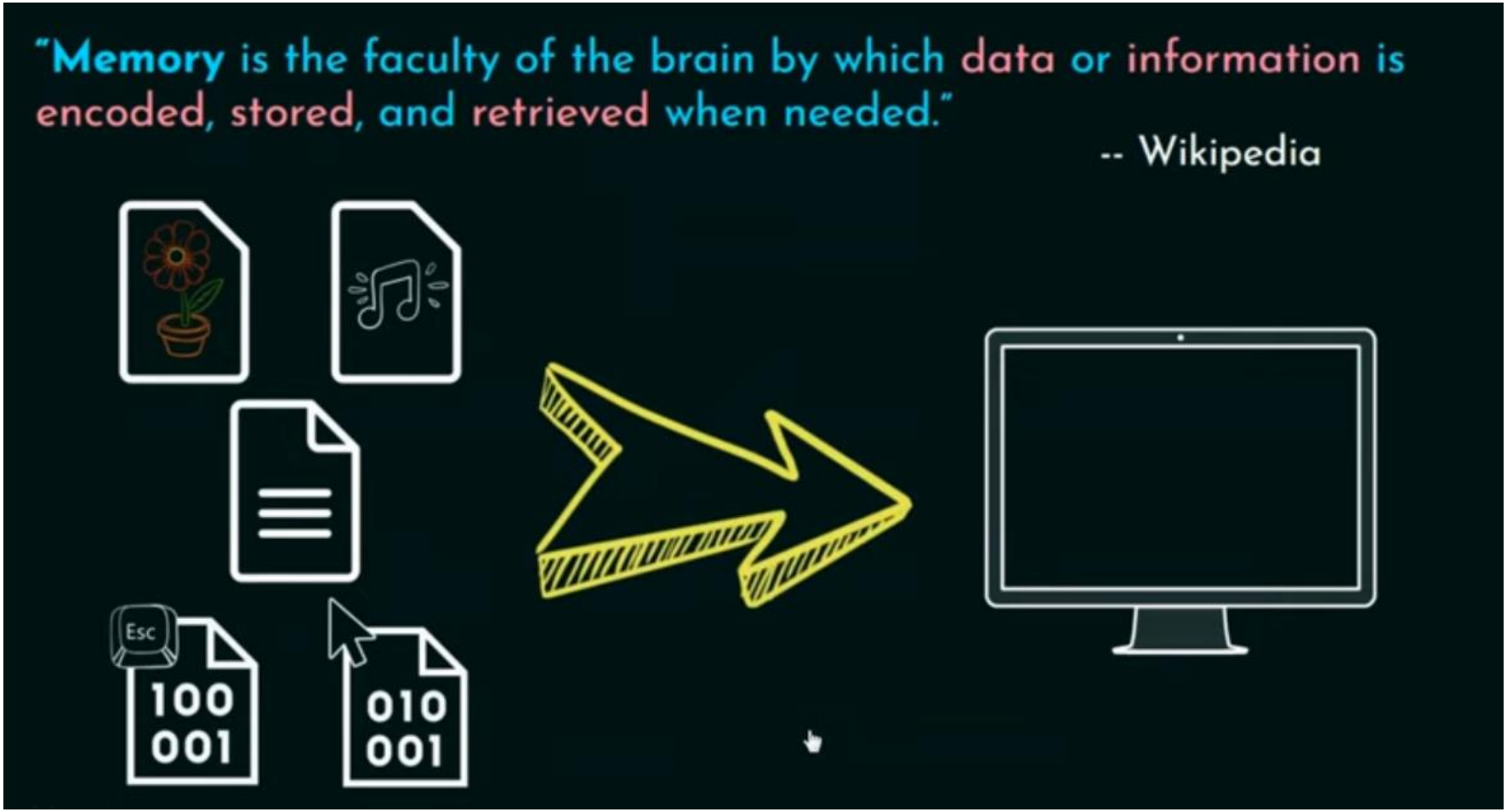


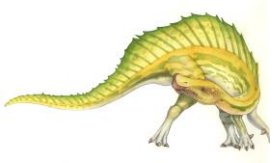
Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation



Memory Management Basics





Memory Management Basics

Memory Management Basics

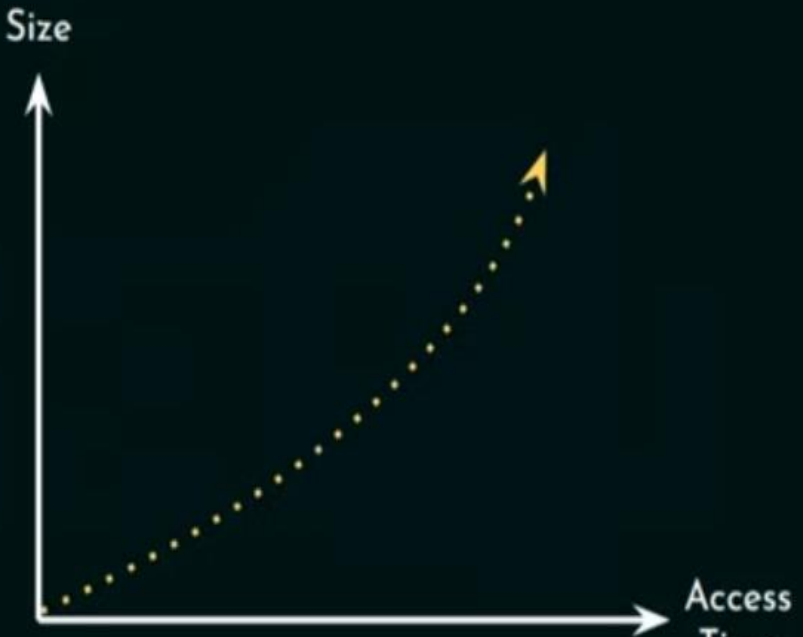
"**Memory** is the faculty of the brain by which data or information is encoded, stored, and retrieved when needed."

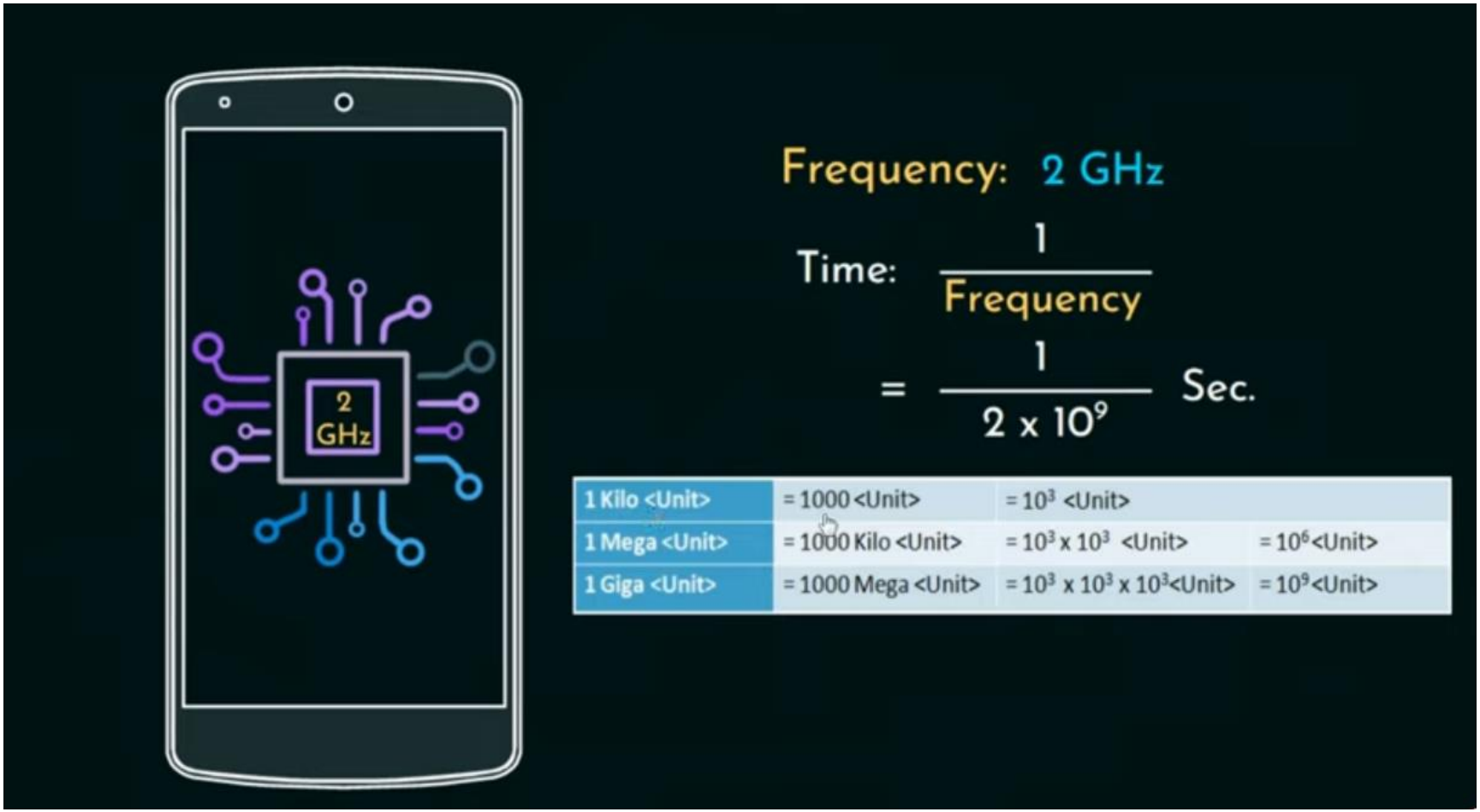
-- Wikipedia

Memory Management Basics

"**Memory** is the faculty of the brain by which data or information is encoded, stored, and retrieved when needed."

-- Wikipedia




2
GHz
Frequency: 2 GHz
Time: 1 / Frequency
= 1 / 2 x 10^9 Sec.
1 Kilo <Unit> = 1000 <Unit> = 10^3 <Unit>
1 Mega <Unit> = 1000 Kilo <Unit> = 10^3 x 10^3 <Unit> = 10^6 <Unit>
1 Giga <Unit> = 1000 Mega <Unit> = 10^3 x 10^3 x 10^3 <Unit> = 10^9 <Unit>

Frequency: 2 GHz

$$\text{Time: } \frac{1}{\text{Frequency}}$$

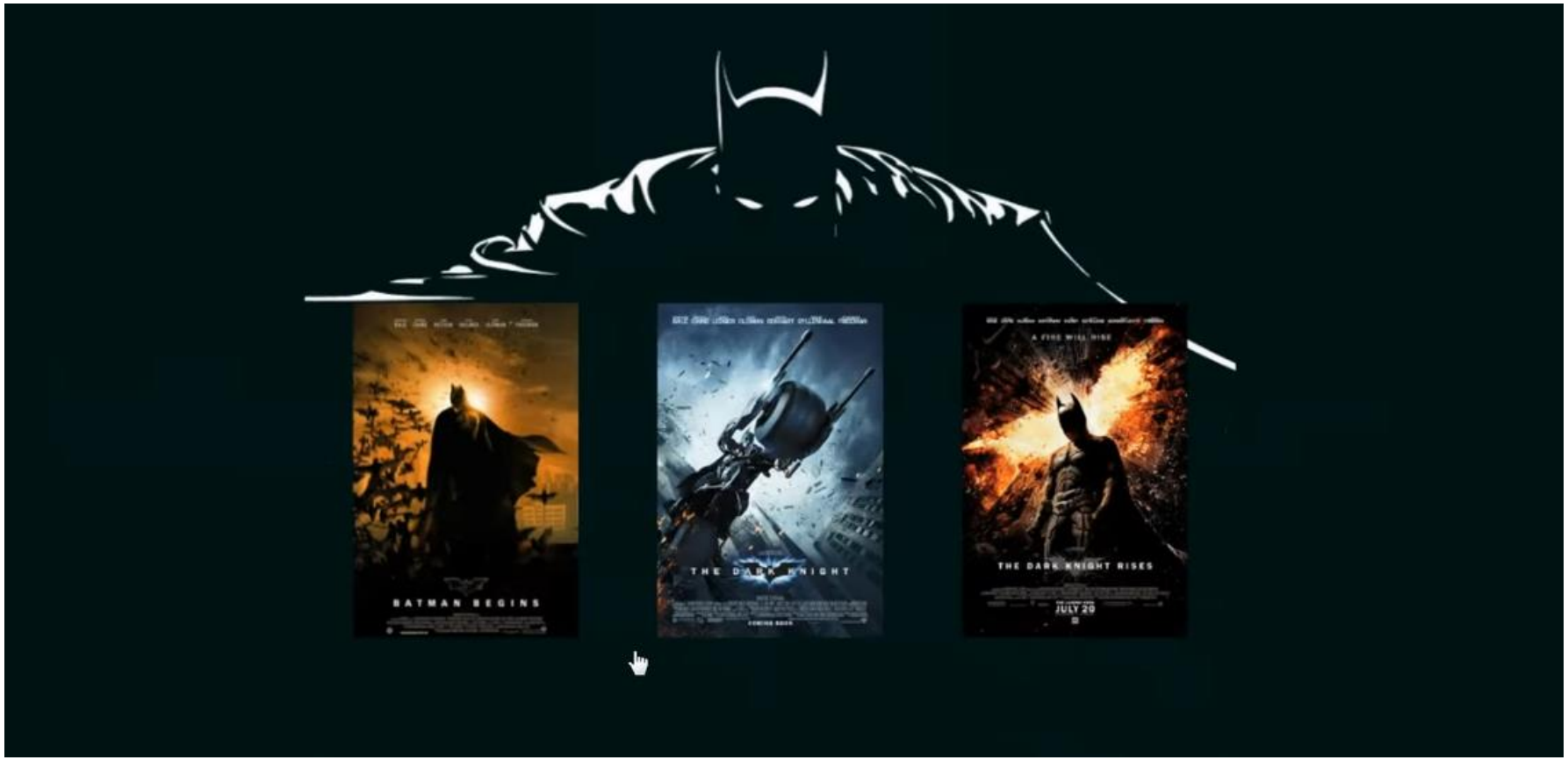
$$= \frac{1}{2 \times 10^{9}} \text{ Sec.}$$

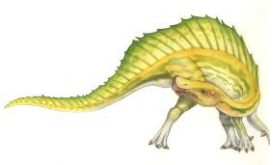
$$= \frac{1}{2} \times 10^{-9} \text{ Sec.}$$

$$= \frac{1}{2} \text{ nsec.}$$

Memory Management Basics

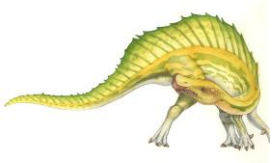
Memory Management Basics



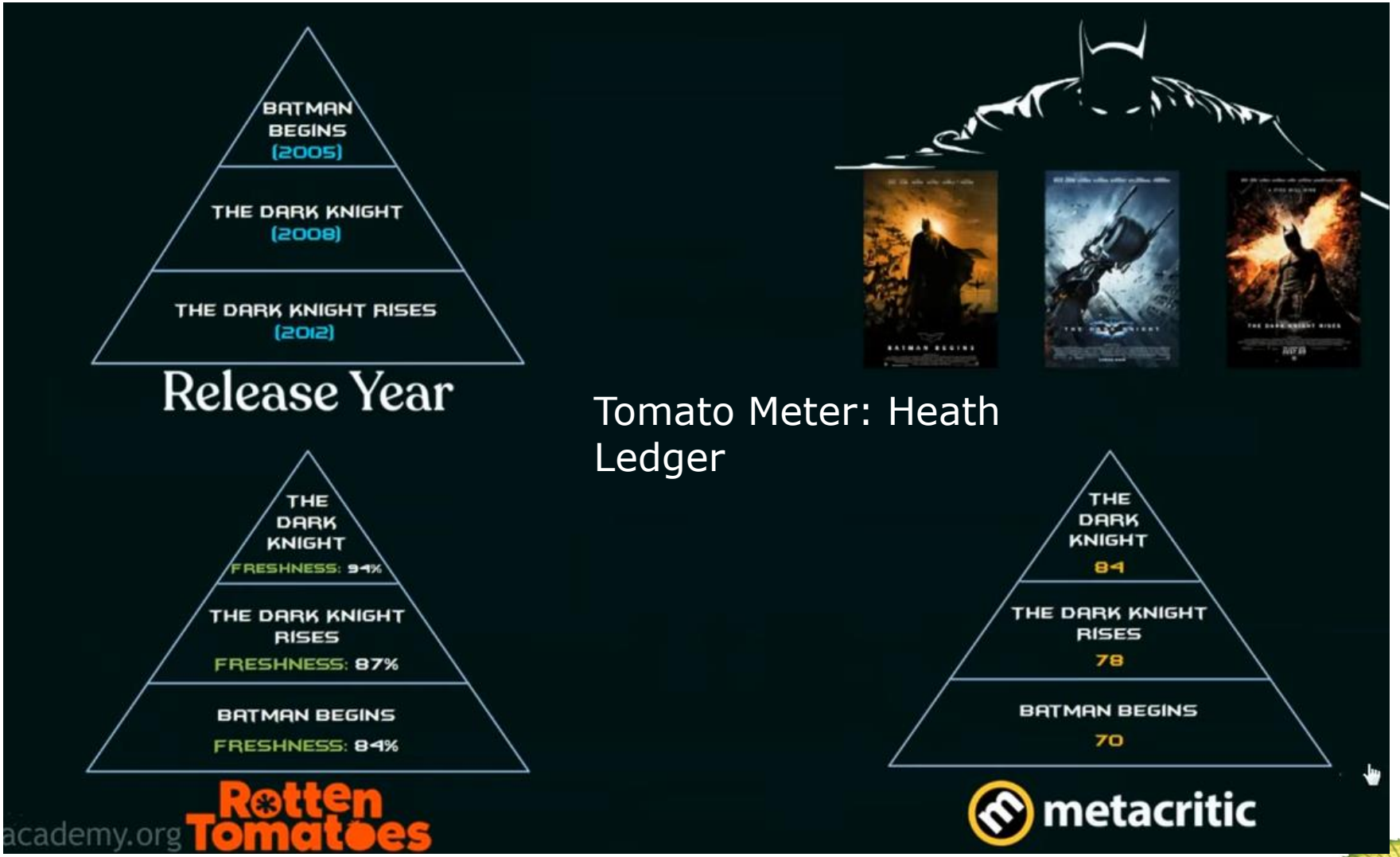


Memory Management Basics

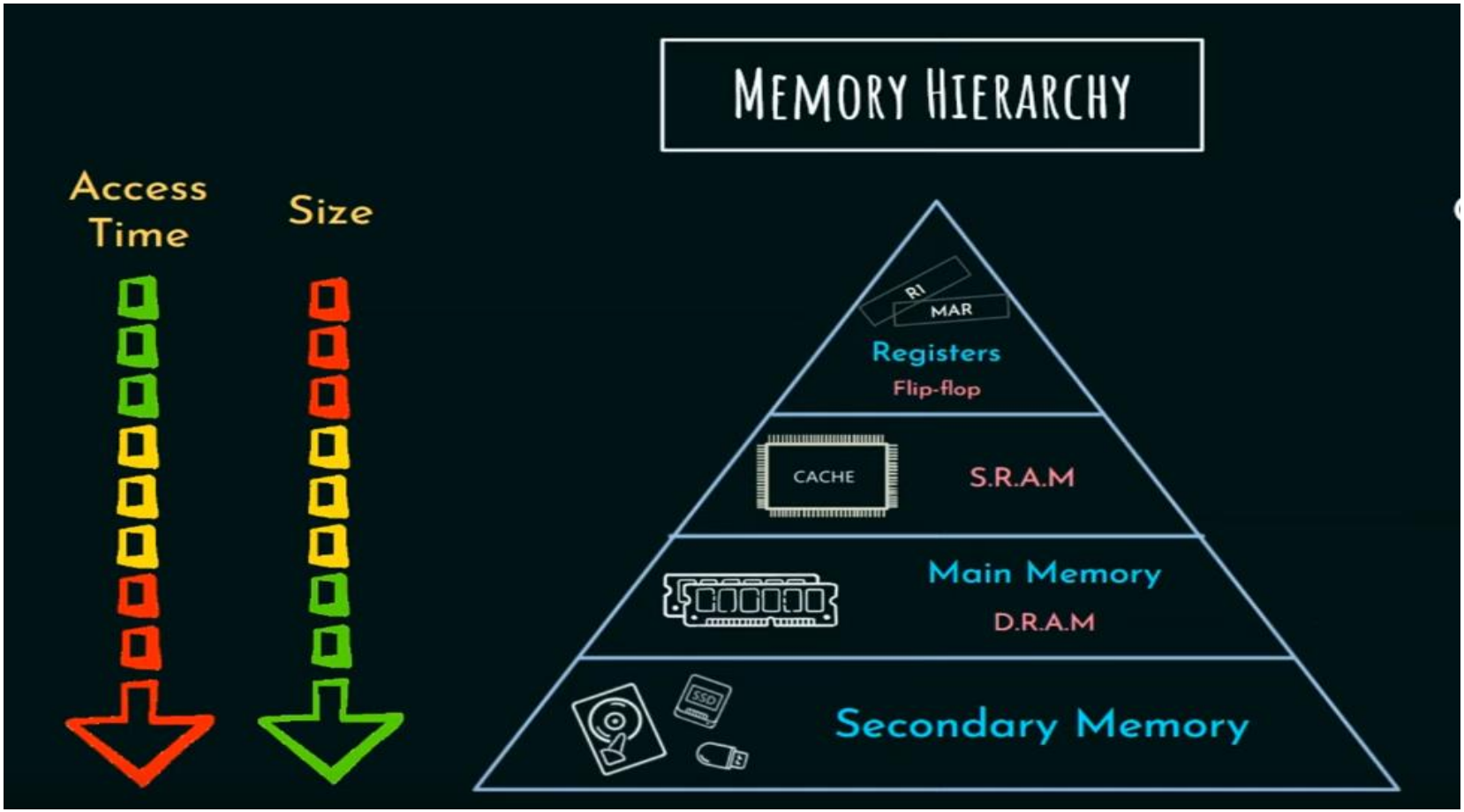
Nolan's Dark Knight Tribology: Release Date



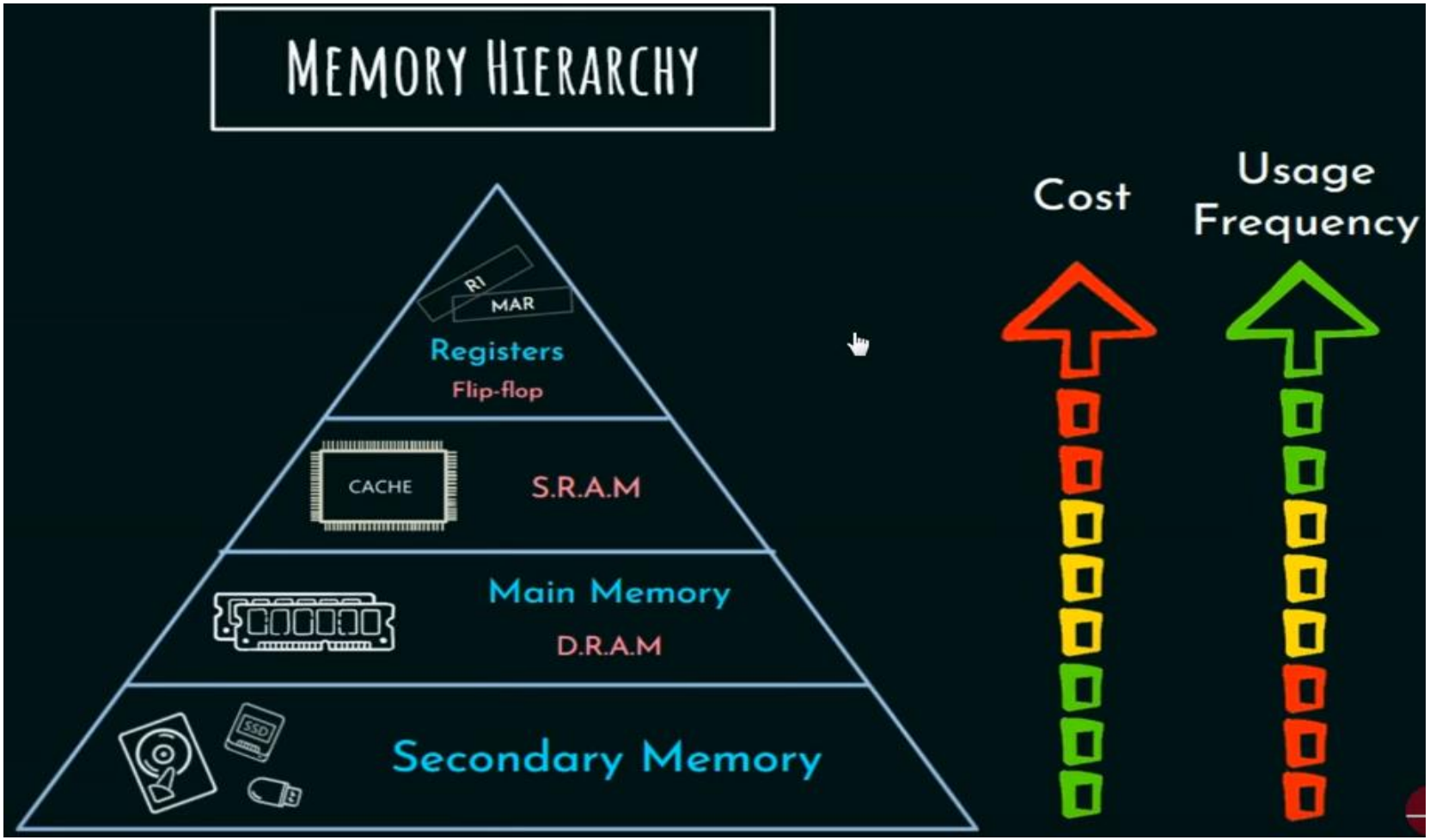
Memory Management Basics

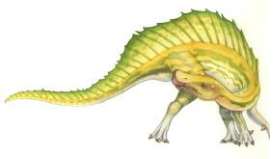


Memory Management Basics

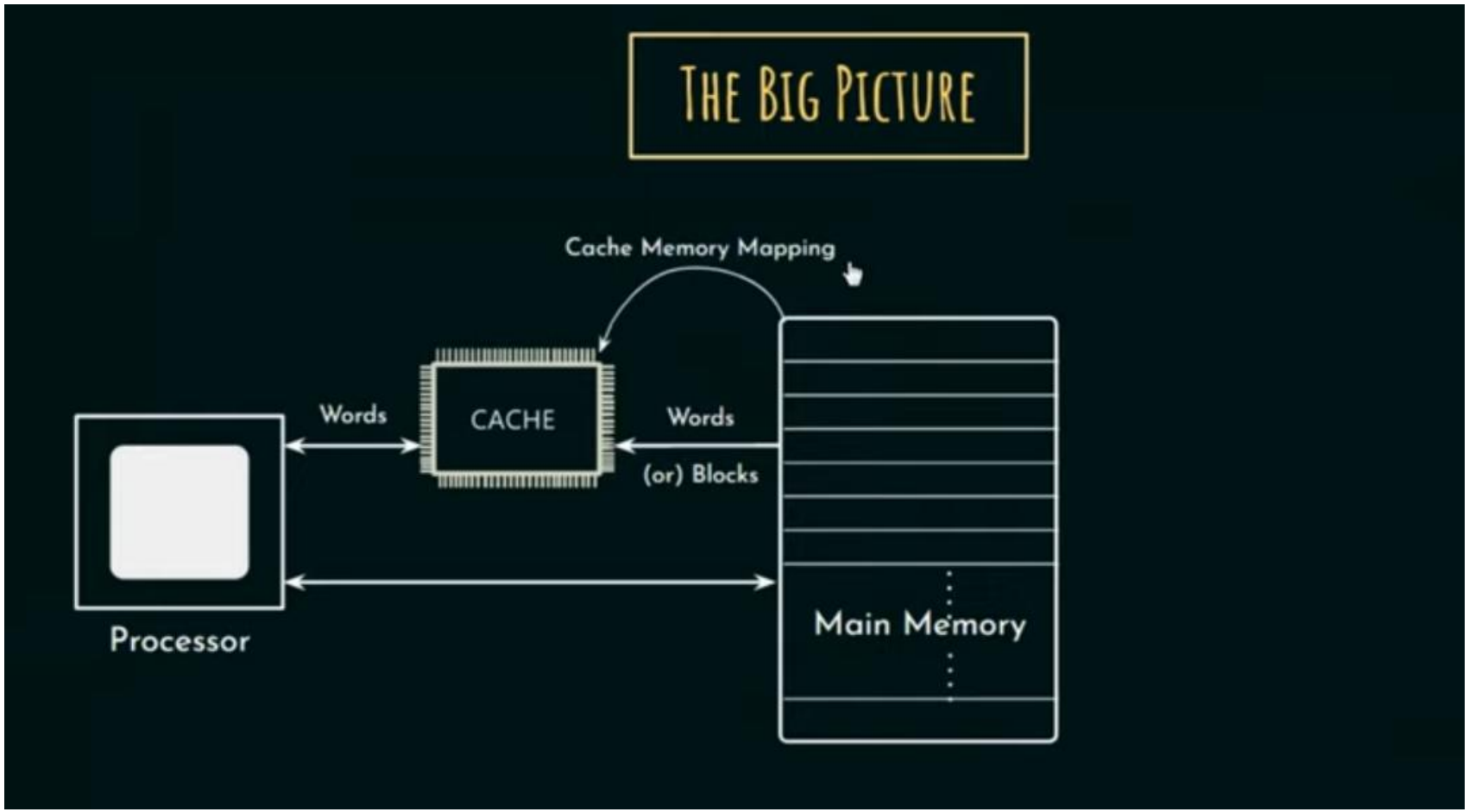


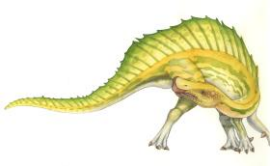
Memory Management Basics



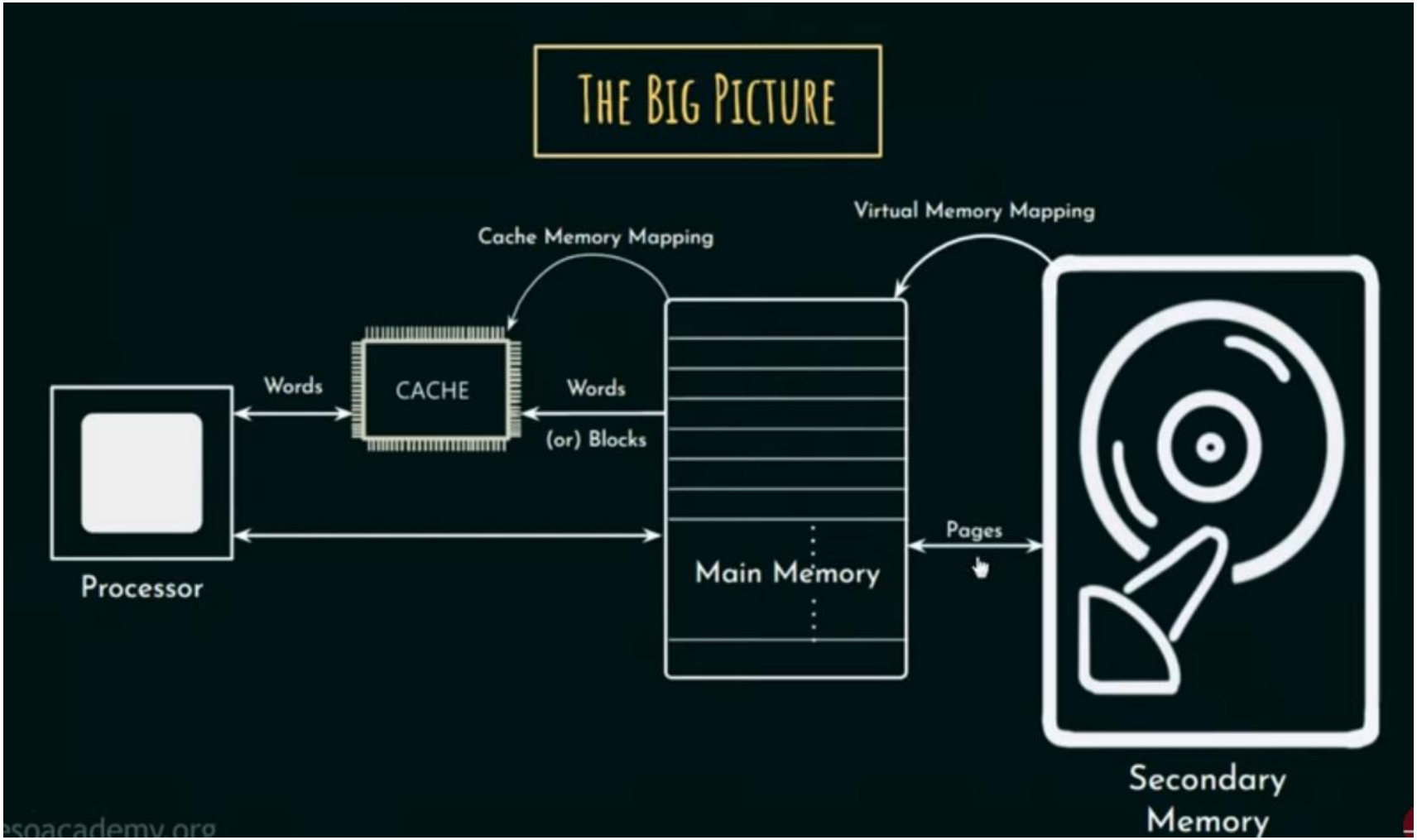


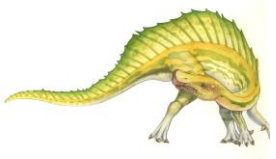
Memory Management Basics





Memory Management Basics



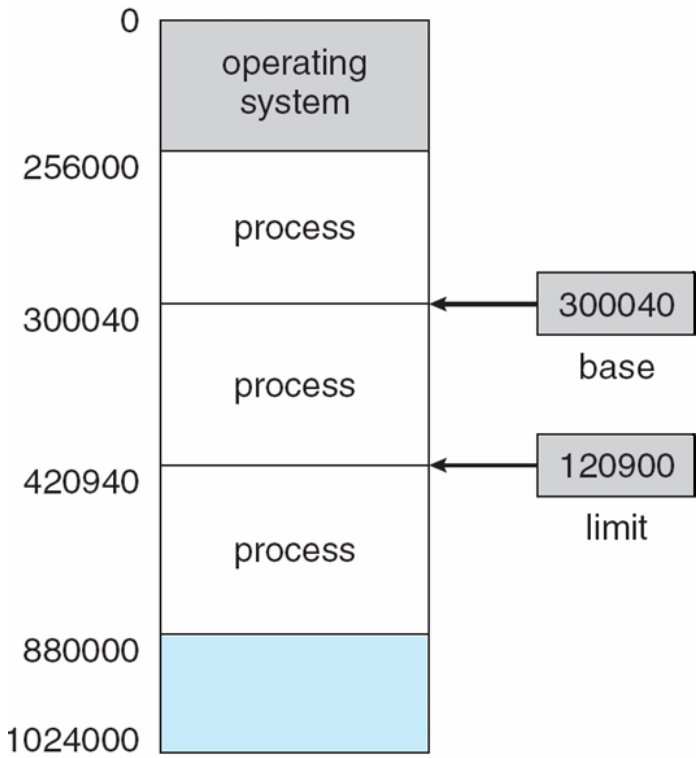


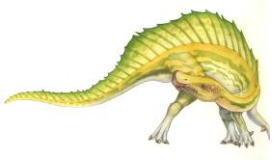
Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

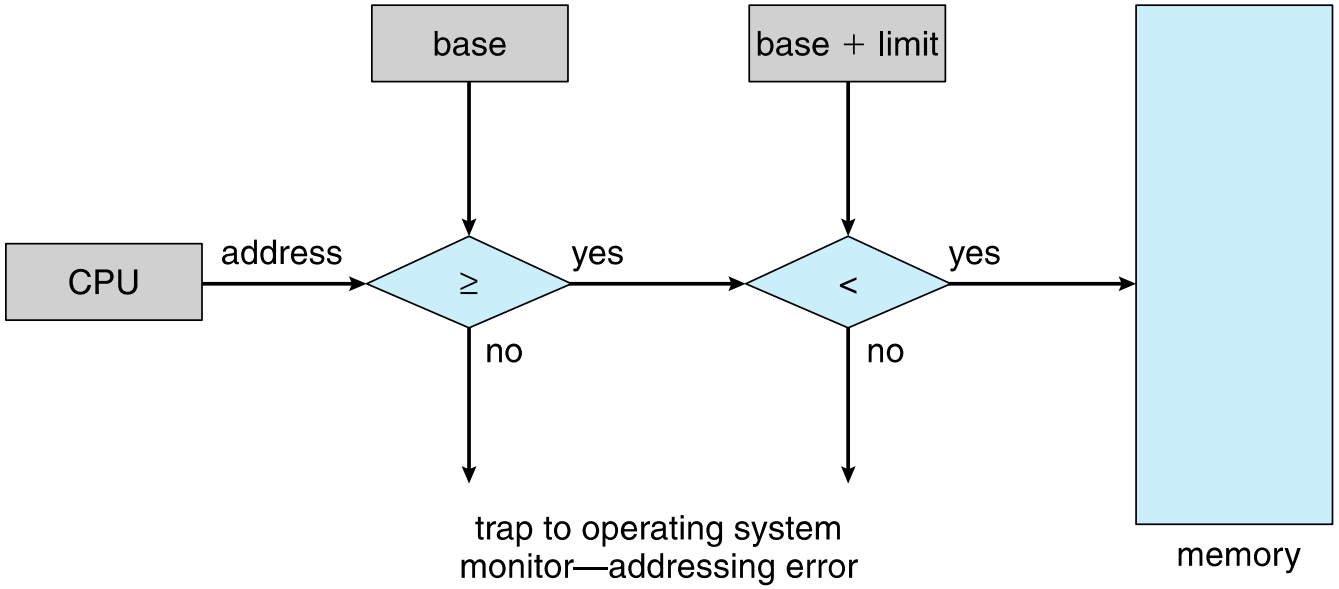
Base and Limit Registers

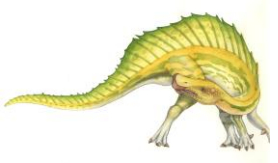
- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user





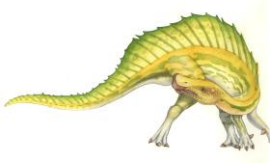
Hardware Address Protection





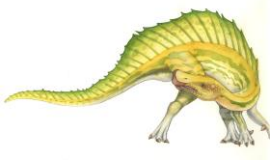
Address Binding

- Programs on disk, ready to be brought into memory to execute form 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?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses 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

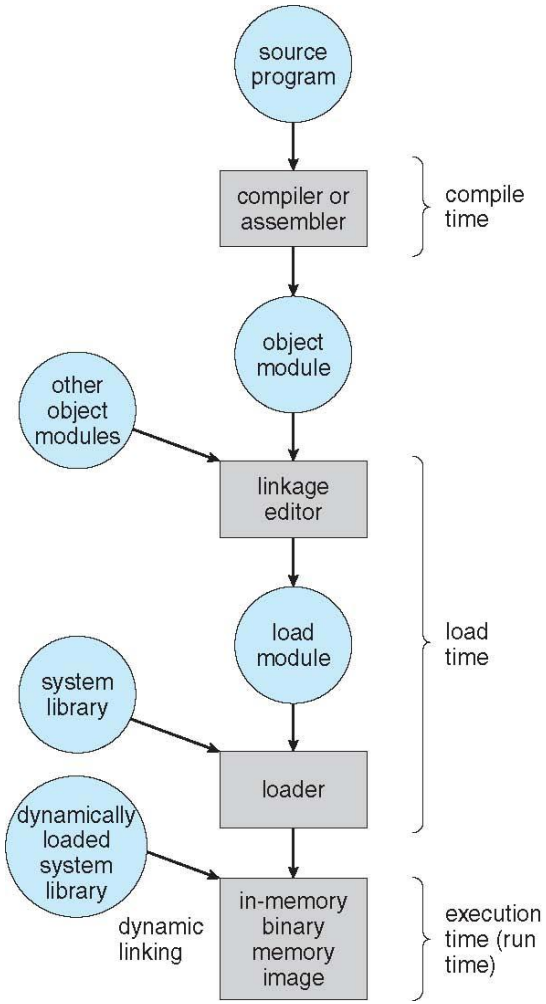


Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

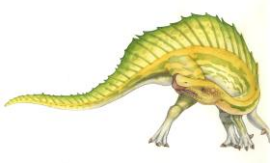


Multistep Processing of a User Program



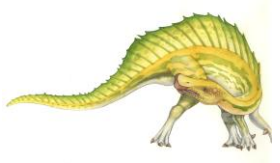
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program



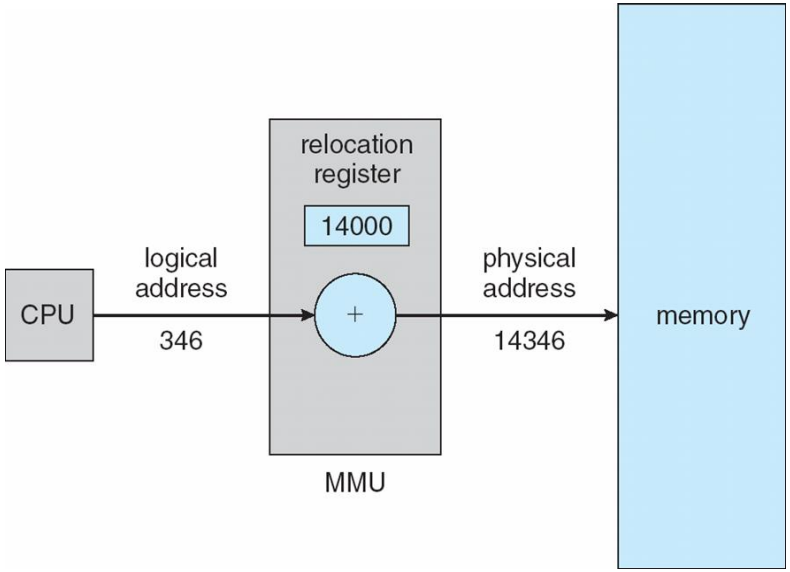
Memory-Management Unit (MMU)

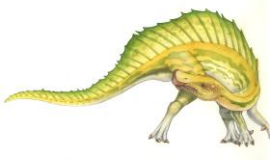
- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- 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



Dynamic relocation using a relocation register

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- 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



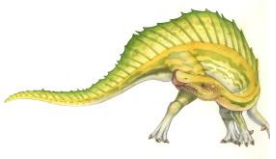


Dynamic Linking

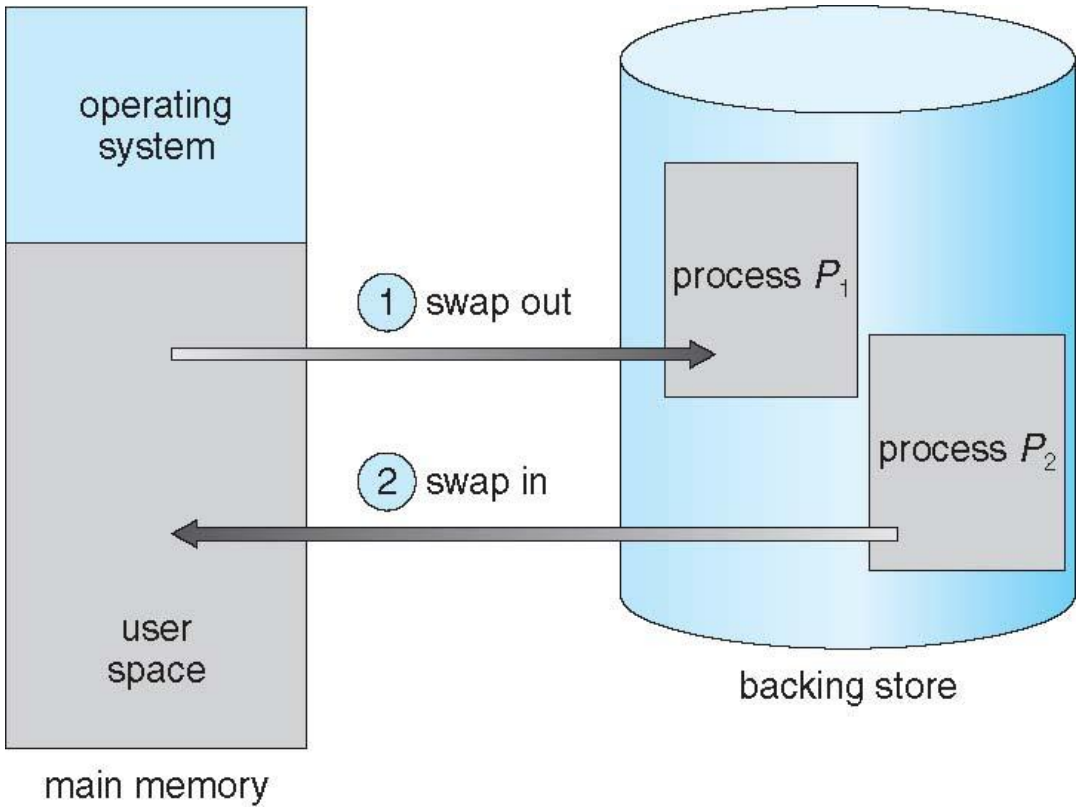
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address 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**
- Consider applicability to patching system libraries
  - Versioning may be needed

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

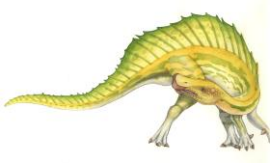


Schematic View of Swapping



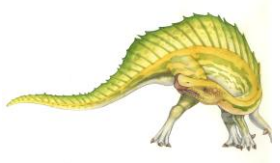
Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`



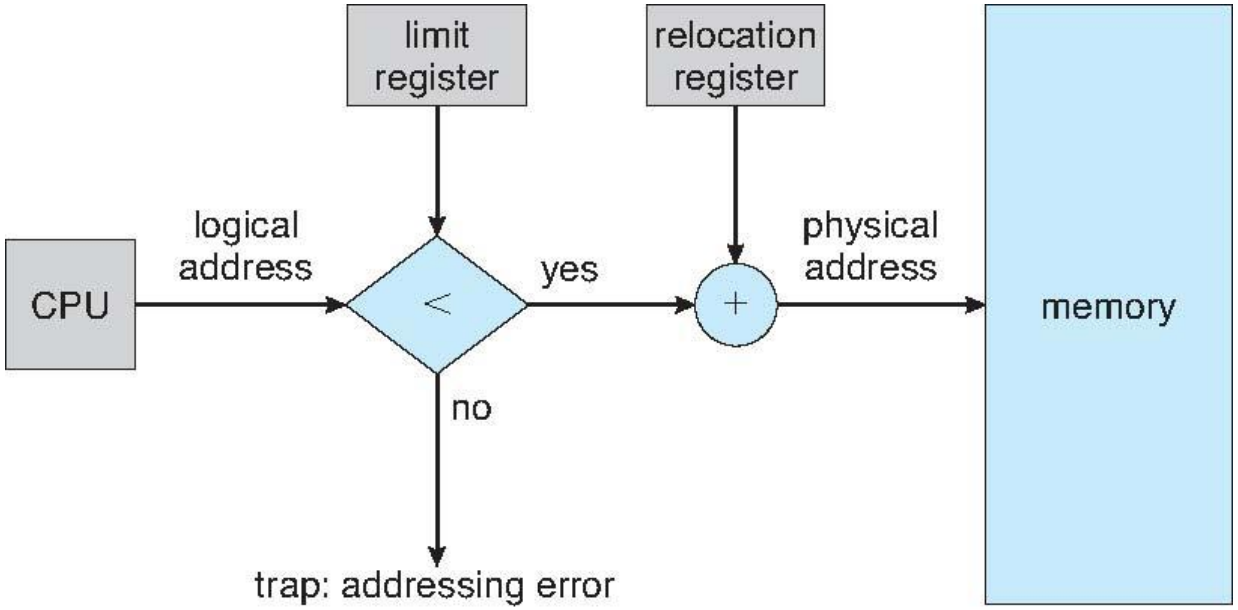
Contiguous Allocation

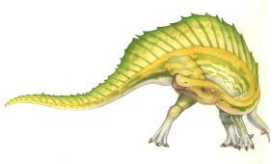
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base 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*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

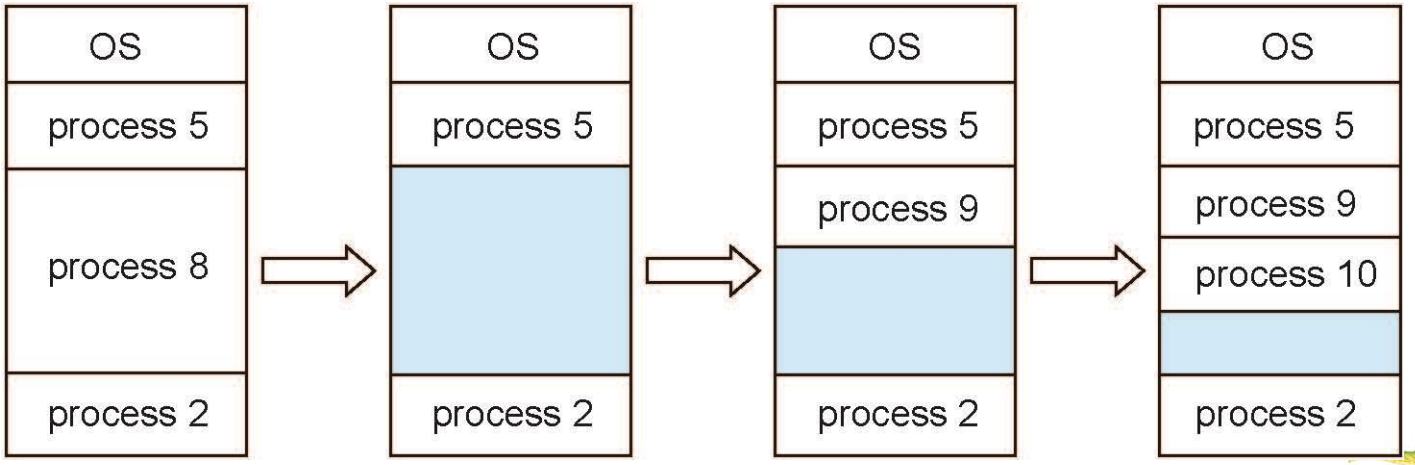
Hardware Support for Relocation and Limit Registers





Multiple-partition allocation

- Multiple-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 (hole)

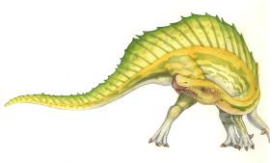


Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **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

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

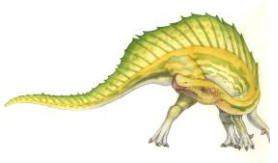


Fragmentation

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

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block



Fragmentation

Fragmented memory before compaction

Memory after compaction

Paging

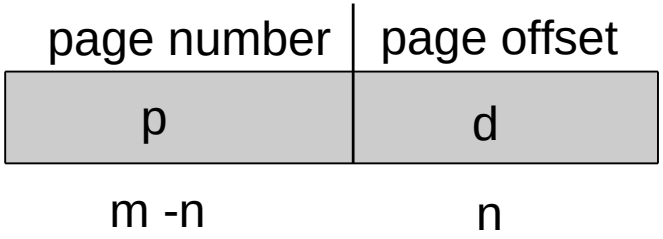
- With paging, physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 1 Gbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

64bits: Several TB of RAM: 40bits:TB, 50:PB, 60:HB,

1KB: 2 Power10
1MB: 2 Power20
1GB: 2 Power30
2GB: 2 Power31
4GB: 2 Power32

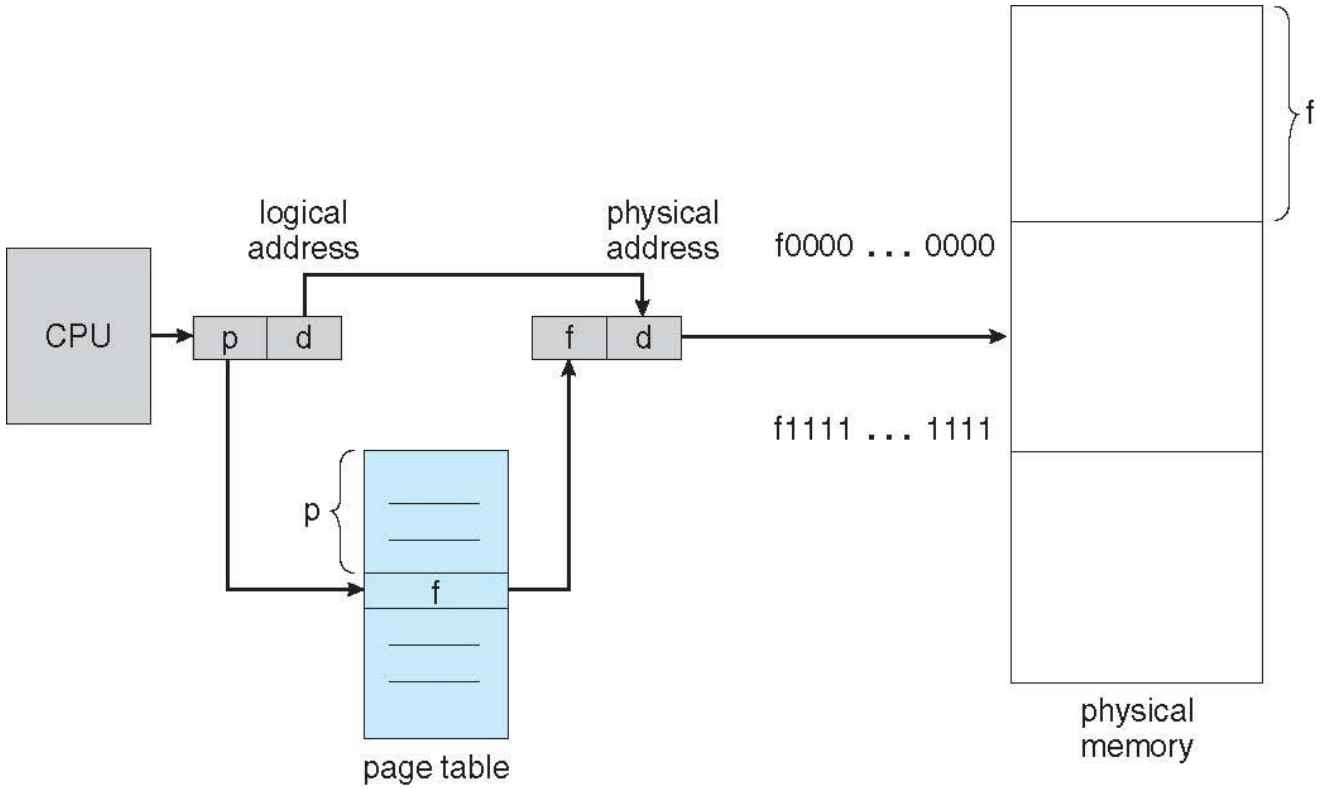
Address Translation Scheme

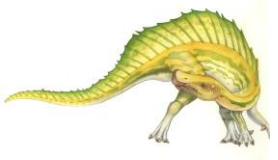
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



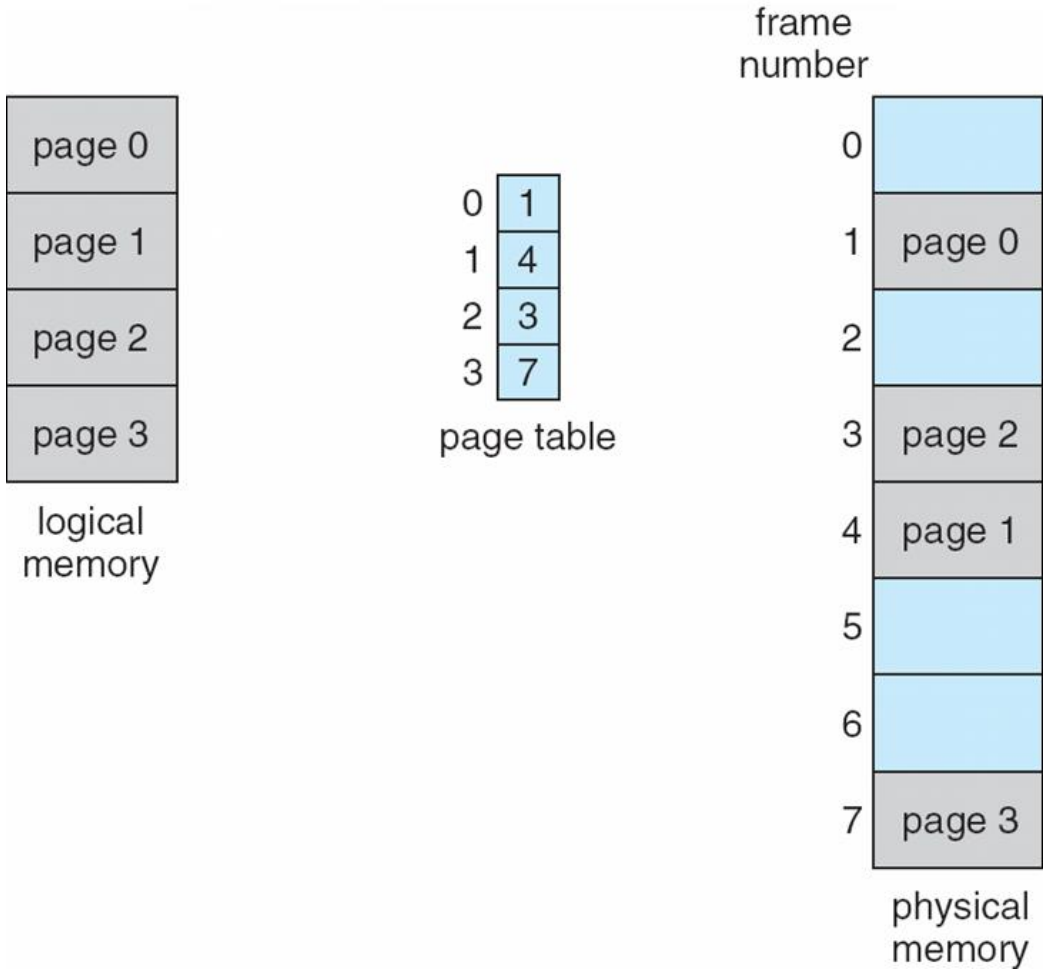
  - For given logical address space 2^m and page size 2^n

Paging Hardware

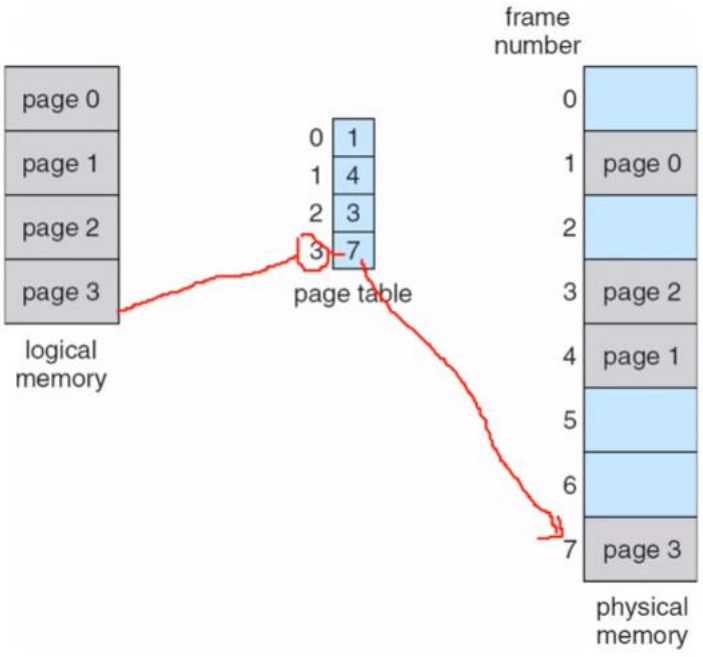




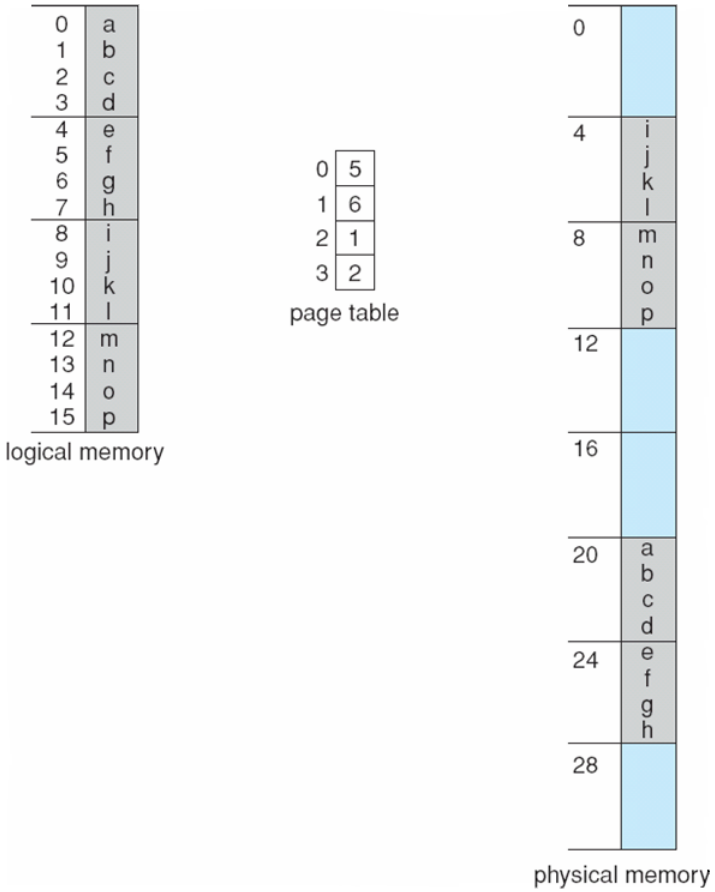
Paging Model of Logical and Physical Memory



Paging Model of Logical and Physical Memory

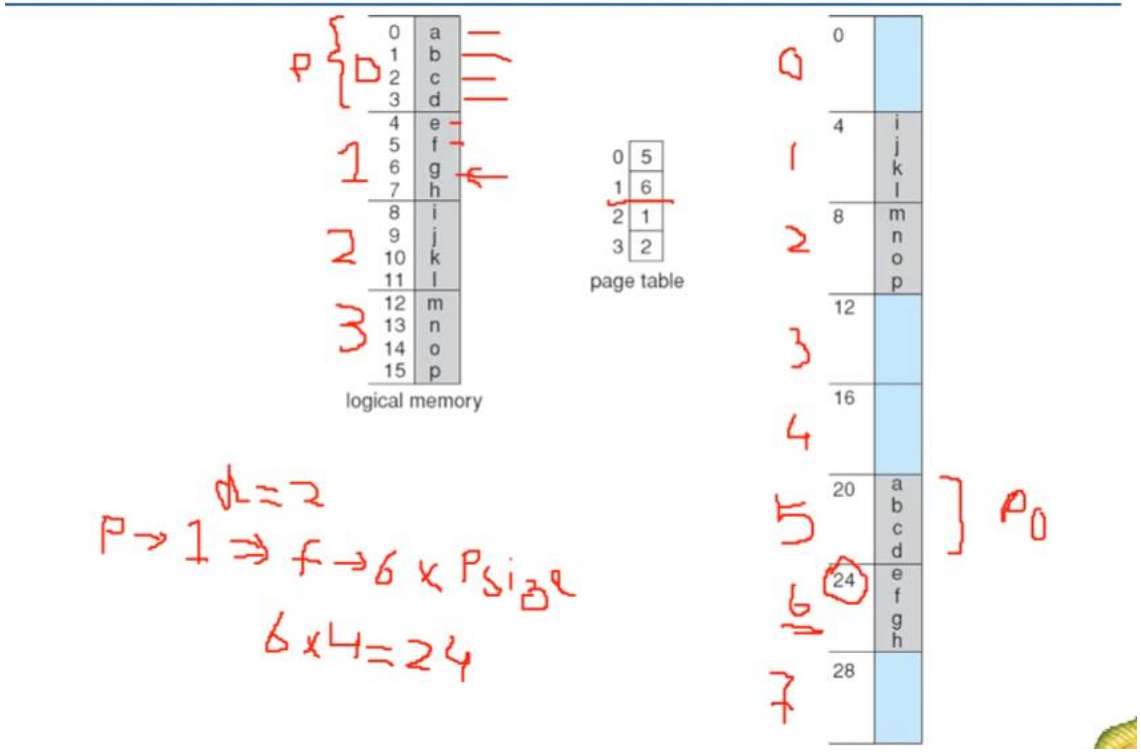


Paging Example



n=2 and *m*=4 32-byte memory and 4-byte pages

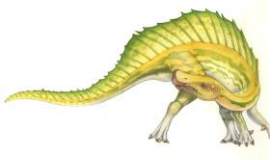
Paging Example



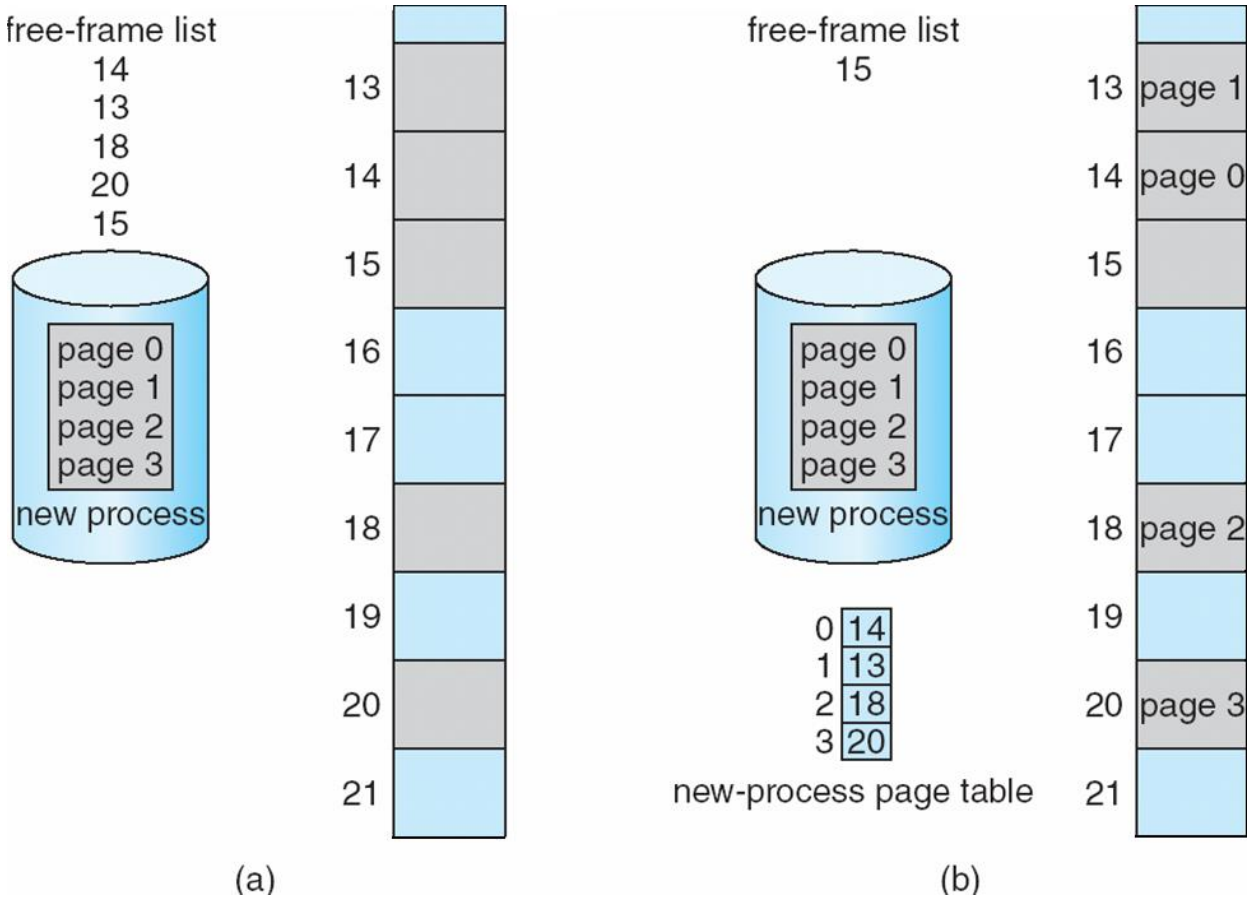
Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Page Size:2KB
71,680: 35Pages
2049B:Eg2



Free Frames



Before allocation

After allocation

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

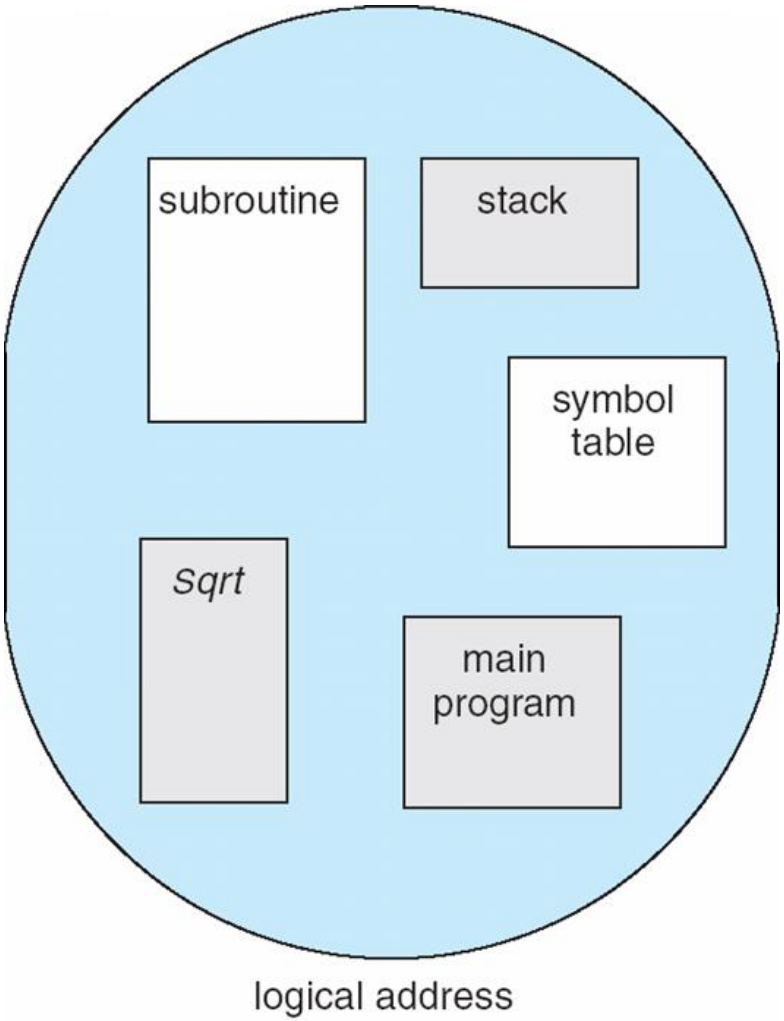
    common block

    stack

    symbol table

    arrays

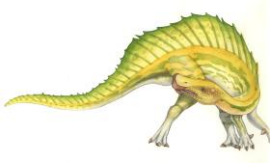
User's View of a Program



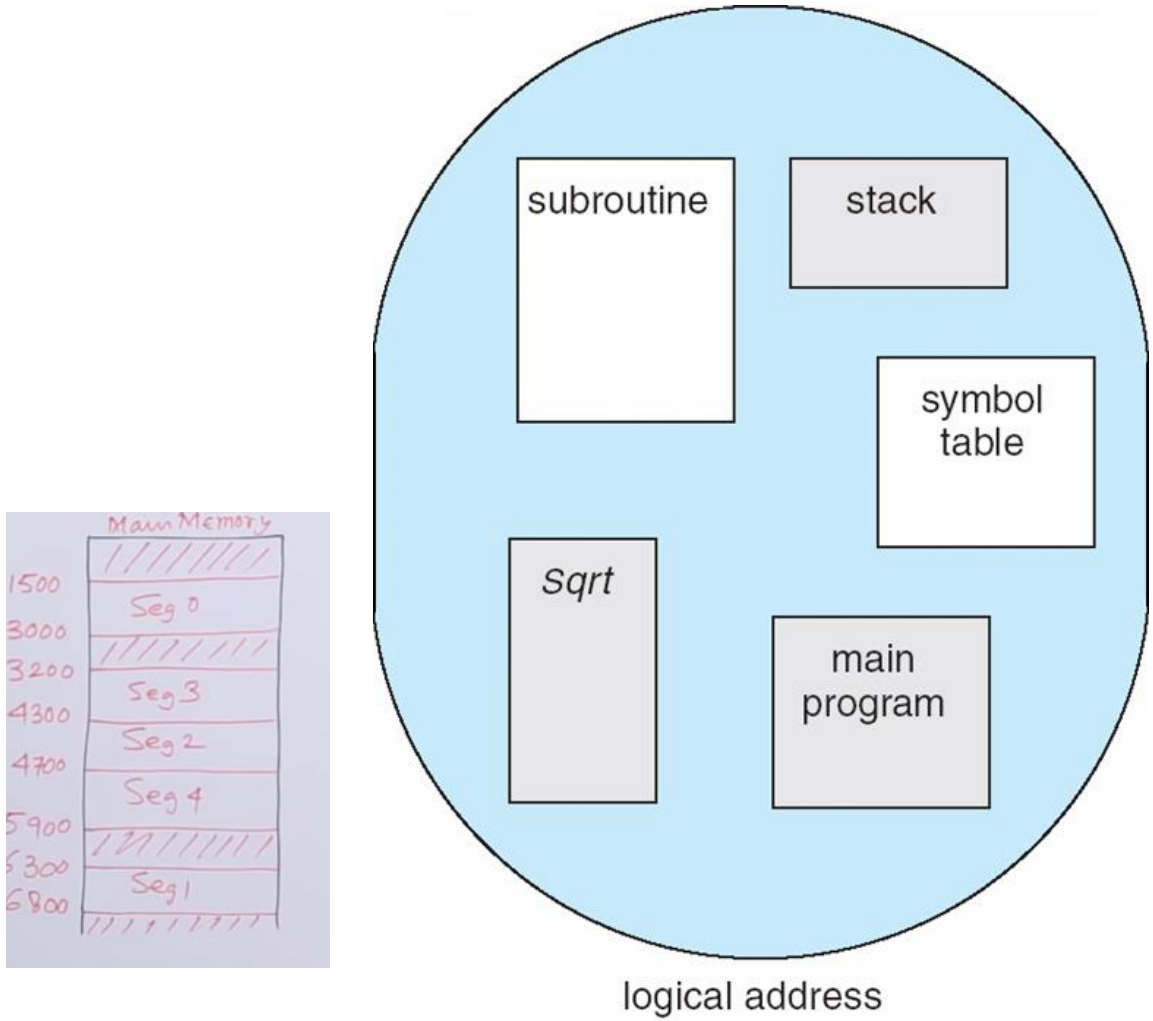
2048B x 2
= 4096B
4090 B

2048B P1
2048B P2
952B
1096B

P2 = 3000B.
2048B
952B

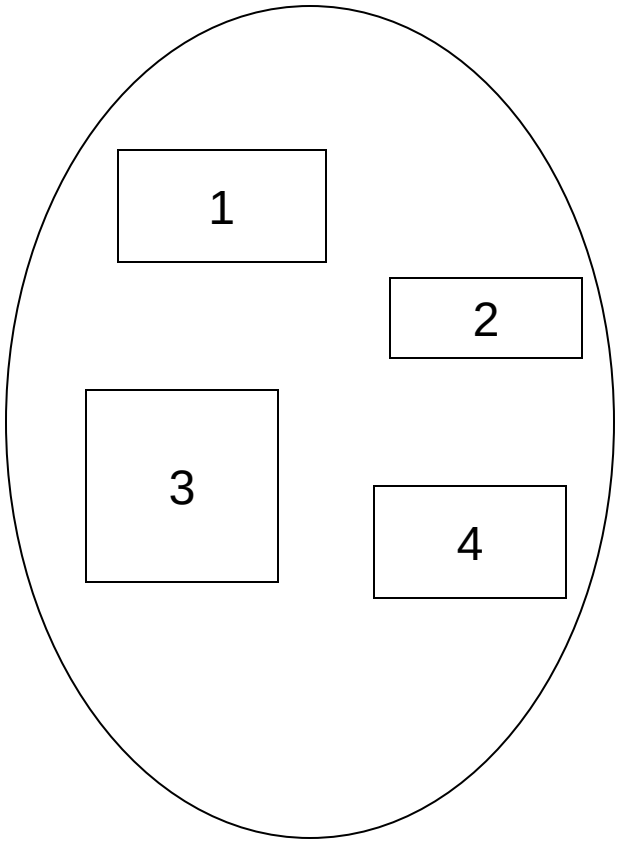


User's View of a Program



	Limit	Base
0	1500	1500
1	500	6300
2	400	4300
3	1100	3200
4	1200	4700

Logical View of Segmentation



user space

physical memory space

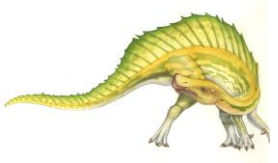
Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

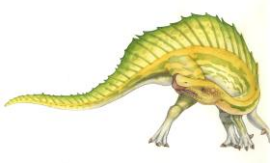
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

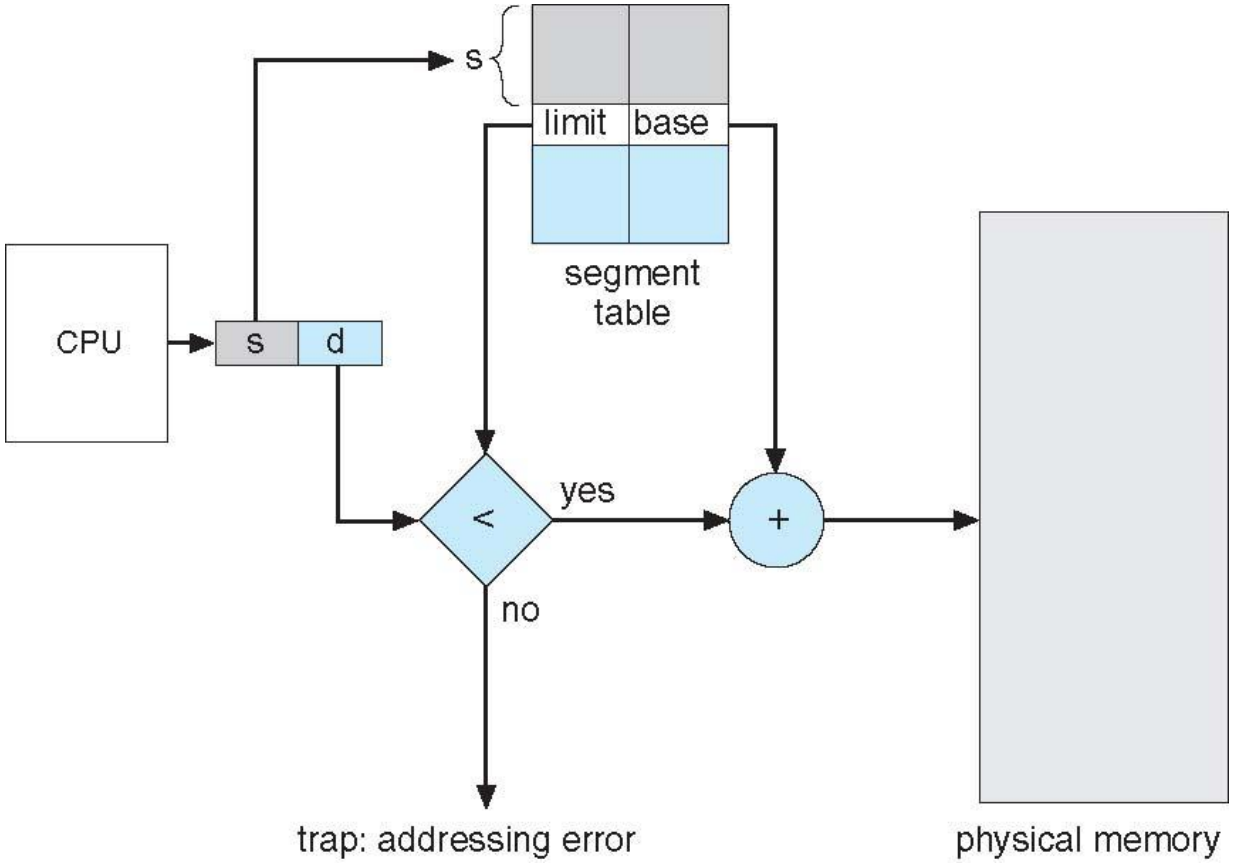


Segmentation Architecture (Cont.)

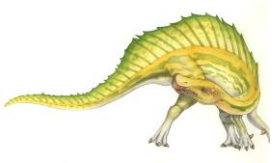
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram



Segmentation Hardware



400
d→(0 to 399)



Segmentation Hardware

Consider following segment table:

Segment No.	Base	Length
0	1219	700
1	2300	14
2	90	100
3	1327	580
4	1952	96

1. Physical address = 1219 + 430
2. " " = 2300 + 11
3. Trap addressing error
4. " " = 1327 + 425
5. " " = 1952 + 95

Which following logical address Will produce Trap addressing error

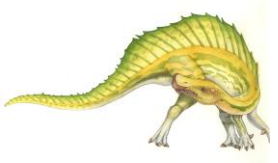
1. 0, 430 2. 1, 11 3. 2, 100
4. 3, 425 5. 4, 95

Ans: A) 1 B) 2 C) 3 D) 5

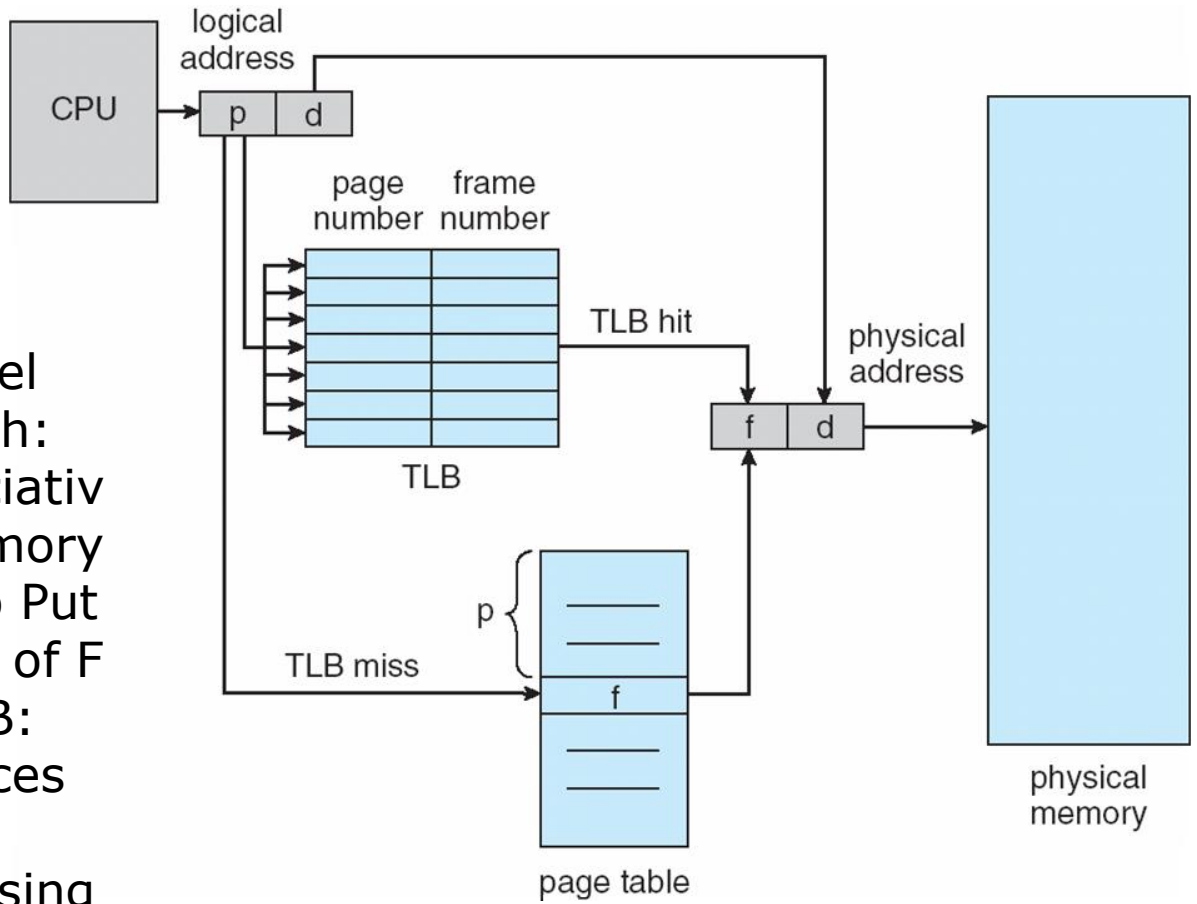
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table: Extra Protection: Can generate Trap.
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)
- **Cache Analogy**

Location of Page table Pointed by



Paging Hardware With TLB



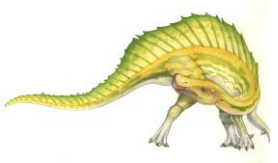
1. Parallel Search: Associative Memory
2. Try to Put Value of F in TLB: Chances of accessing same page again.

Associative Memory

- Associative memory – parallel search

Page #	Frame #

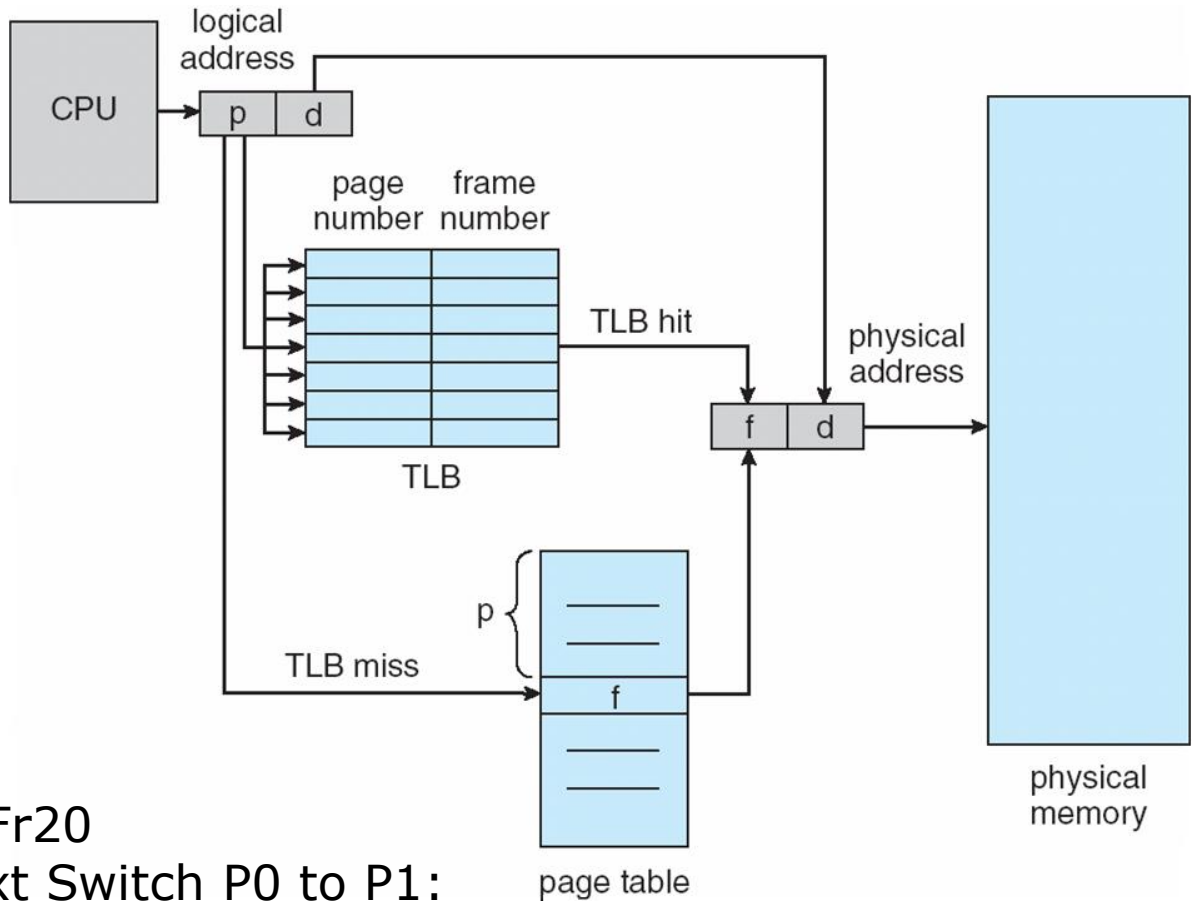
- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory



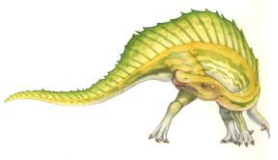
Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

Paging Hardware With TLB



- Pg15-Fr20
- Context Switch P0 to P1:
- Pg15-Fr20 in actual PT
- Stale Entry in TLB
- P0-Pg15-F2r0
- P0-Pg15-Fr20



Effective Access Time

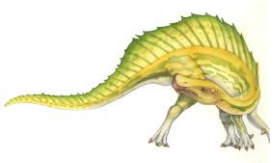
1. TLB Miss Happens
2. Example to Find Access time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)
  - 80% find entry in 1 Access TLB
  - 20% find entry 2Acces

Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access

- EAT = 0.80 x 100 + 0.20 x 200 = 120ns

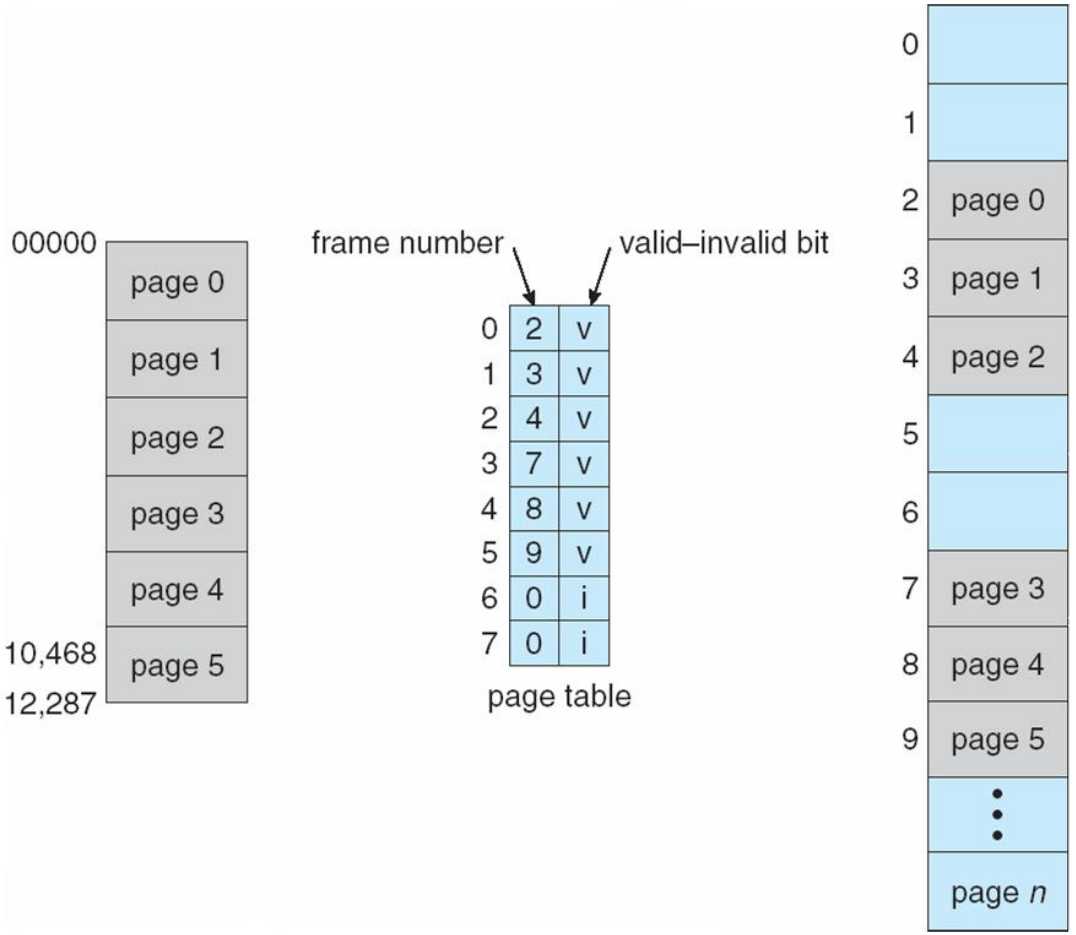
- Consider more realistic hit ratio : all imp pages get loaded in to TLB -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns: Very close to actual MA: instead of paging overhead

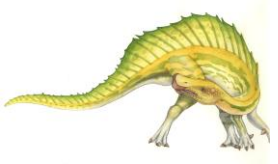


How Memory Protection: Paging

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

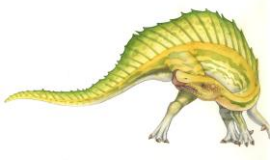
Valid (v) or Invalid (i) Bit In A Page Table



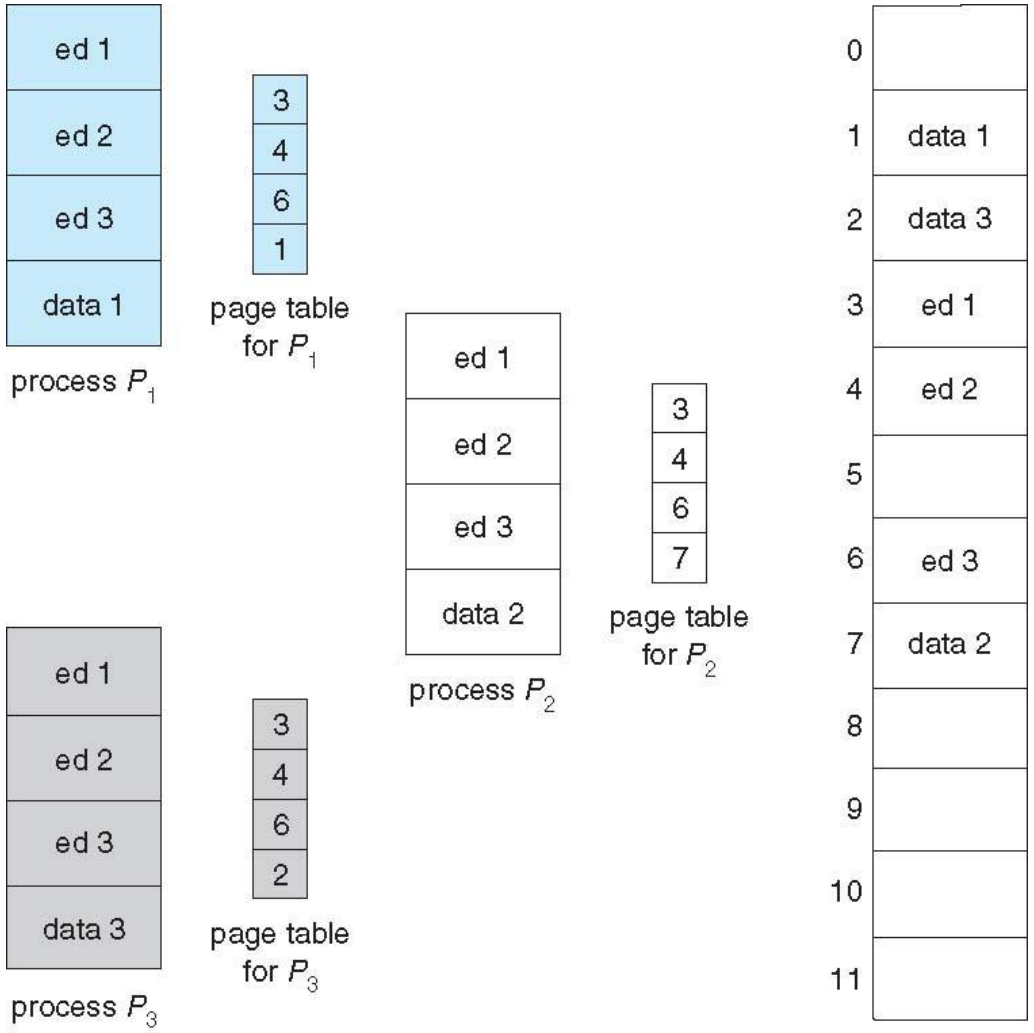


Shared Pages

- **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



Structure of the Page Table

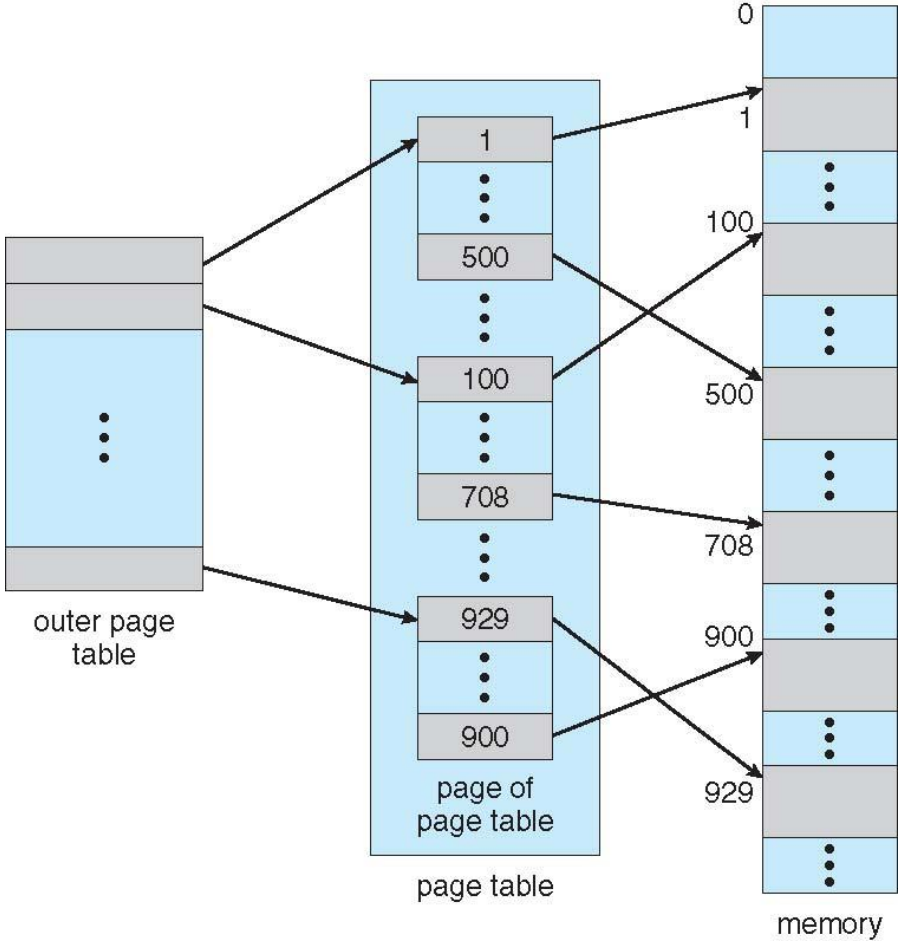
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12}): offset is 12bits
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes* 2^{20} -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

- Process not used 1G memory at a time
- Use only subset of space: grow need more p
- No need to allocate all pages:
- make valid bit set only to few

Hierarchical Page Tables

- Idea: Don’t have one layer of page table
- 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
- Land line number example: 080 28603569

Two-Level Page-Table Scheme

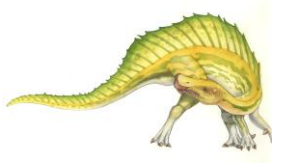


Two-Level Paging Example

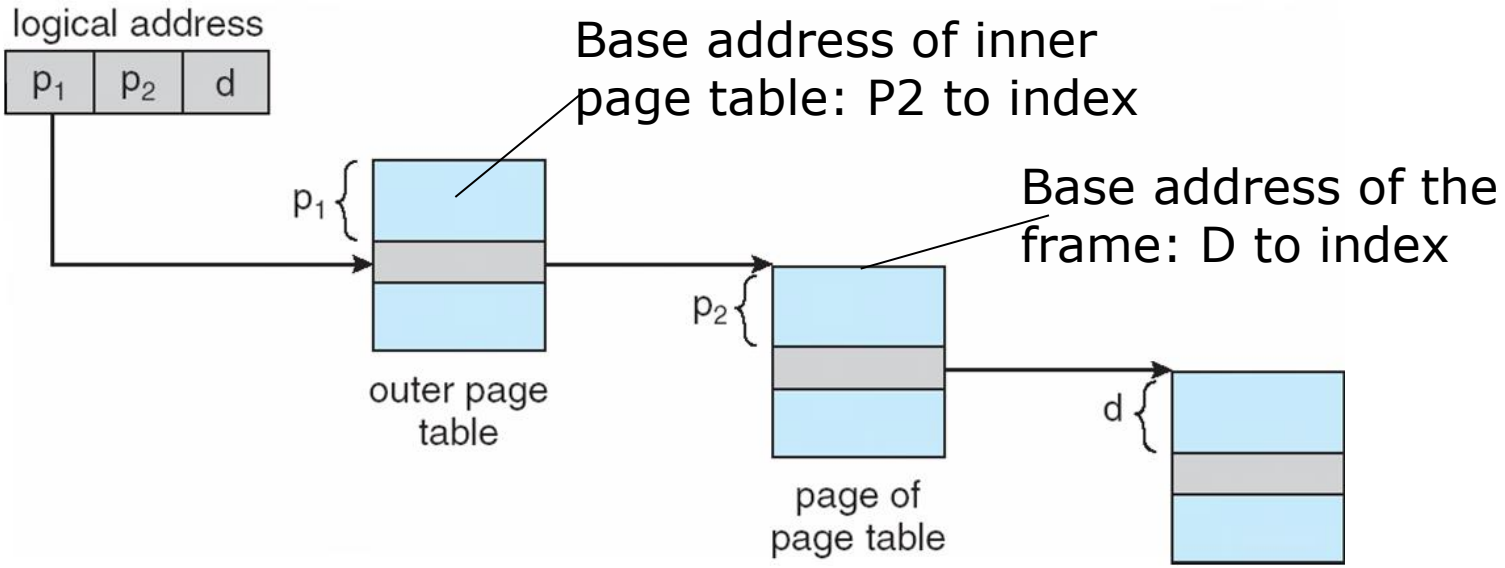
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

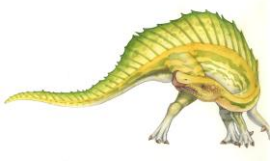
page number		page offset
p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 to index into inner page table and offset to find actual content
- Known as **forward-mapped page table**



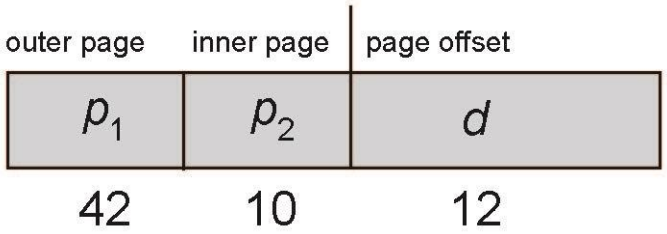
Address-Translation Scheme





64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



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

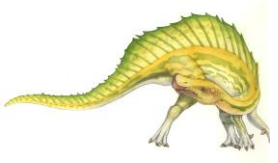
Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

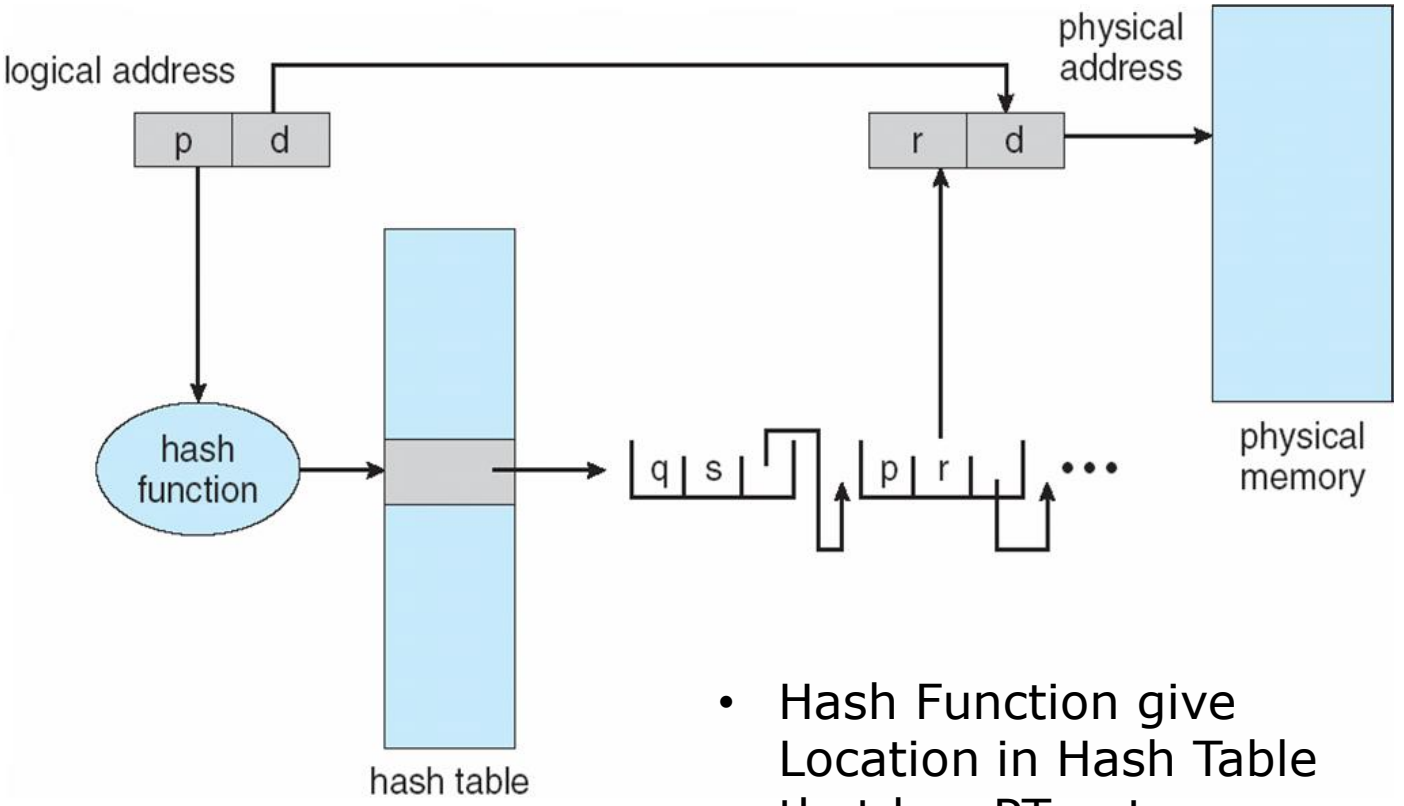
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

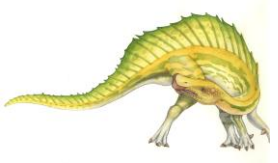
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)



Hashed Page Table

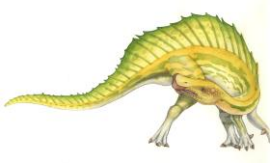


- Hash Function give Location in Hash Table that has PT entry – Chained Entry
- Match entry with page number

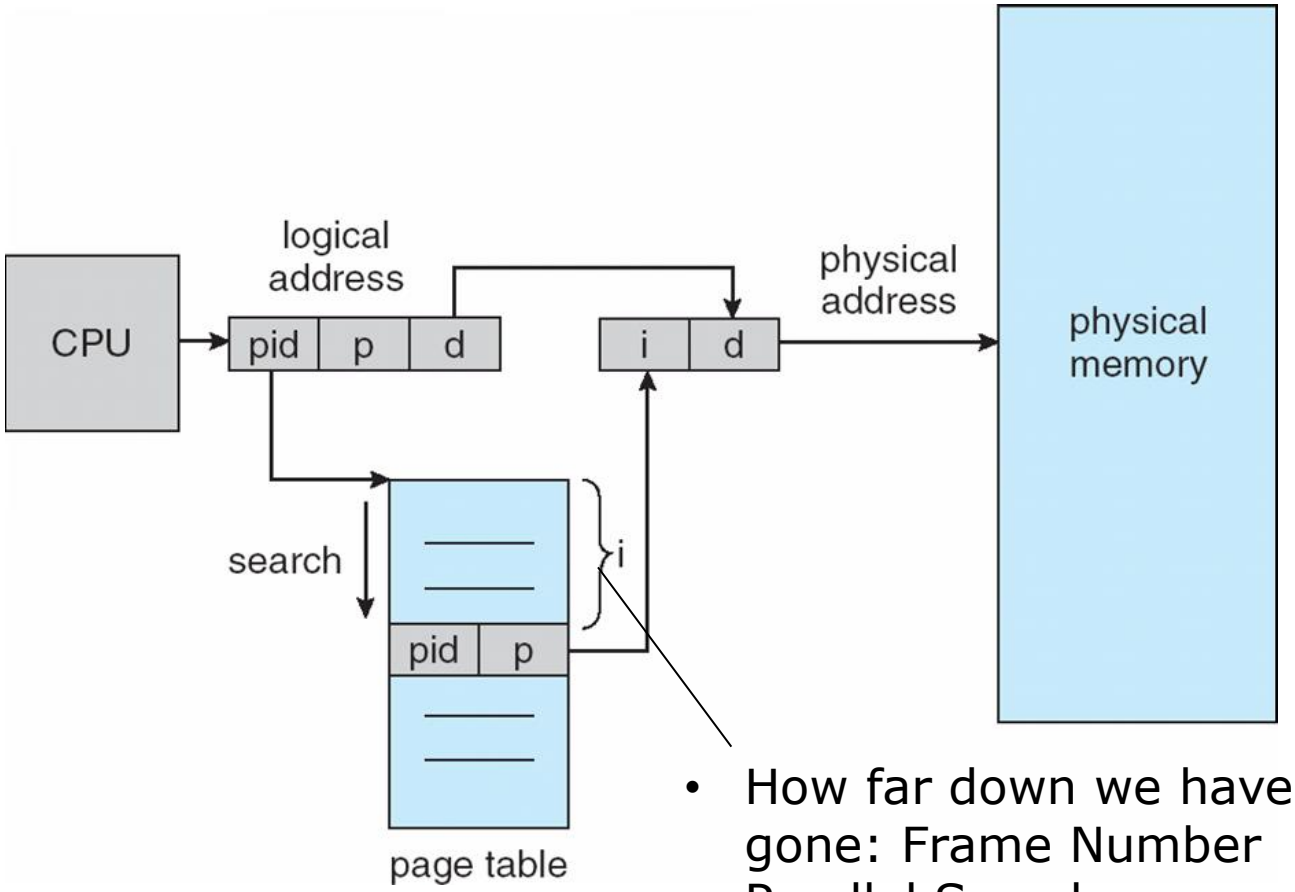


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One 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
- 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
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address



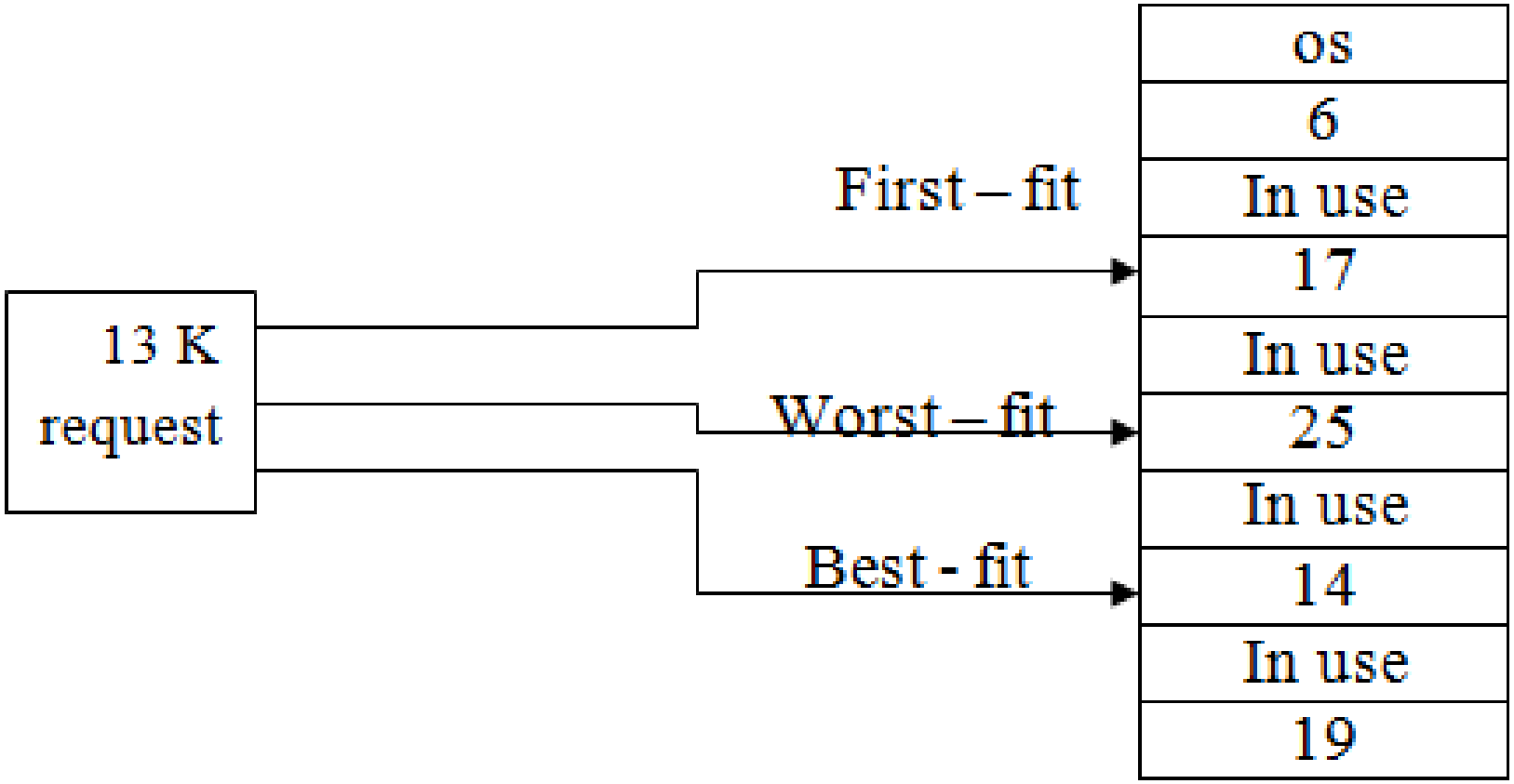
Inverted Page Table Architecture

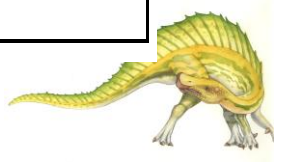


- How far down we have gone: Frame Number
- Parallel Search

Numerical

Suppose that we have free segments with sizes: 6, 17, 25, 14, and 19. Place a program with size 13kB in the free segment using first-fit, best-fit and worst fit?





Numerical

Consider a user program of logical address of size **6** pages and page size is **4** bytes.
The physical address contains **300** frames. The user program consists of **22** instructions **a, b, c, . . . u, v .**
Each instruction takes **1 byte**. Assume at that time the free frames are **7, 26, 52, 20, 55, 6, 18, 21, 70,** and **90**.
Find the following? (10 degrees)

A) Draw the logical and physical maps and page tables?
B) Allocate each page in the corresponding frame?
C) Find the physical addresses for the instructions **m, d, v, r**?
D) Calculate the fragmentation if exist?

Numerical

Answer Q6

0	. a . b . c . d
1	. e . f . g . h
2	. i . j . k . l
3	. m . n . o . p
4	. q . r . s . t
5	. u . v . .

Logical map

Page number	Frame number
0	7
1	26
2	52
3	20
4	55
5	6

page table

	Contents
6	. u . v . .
7	. a . b . c . d
20	. m . n . o . p
26	. e . f . g . h
52	. i . j . k . l
55	. q . r . s . t

Physical map

The physical address = page size * frame number + offset

The physical address of m = 4*20 +0 = 80

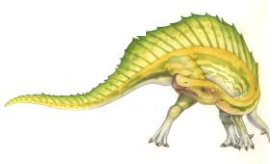
The physical address of d = 4*7+3 = 31

The physical address of v = 4* 6 +1 = 25

The physical address of r = 4*55+ 1= 221

The external fragmentation = 0

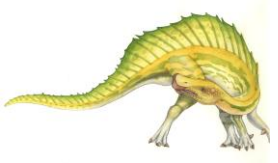
The internal fragmentation = 2



Numerical

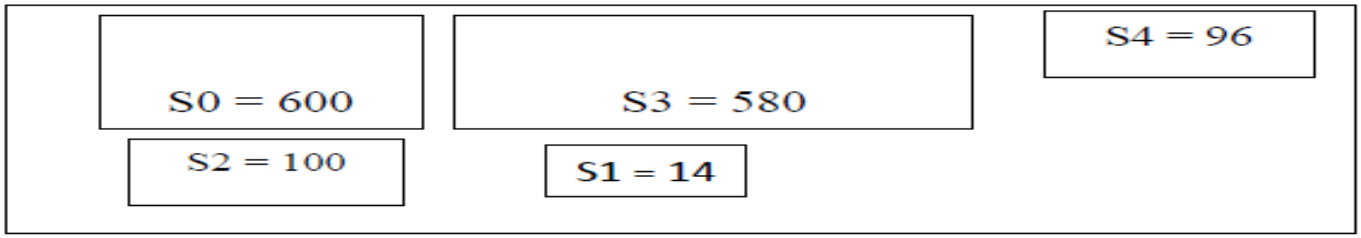
Consider a program consists of five segments: S0 = 600, S1 = 14 KB, S2= 100 KB, S3 =580 KB, and S4 = 96 KB. Assume at that time, the available free space partitions of memory are 1200–1805, 50 – 150, 220-234, and 2500-3180. Find the following:

1. Draw logical to physical maps and segment table?
2. Allocate space for each segment in memory?
3. What are the addresses in physical memory for the following logical addresses: 0.580, (b) 1.17 (c) 2.66 (d) 3.82 (e) 4.20?

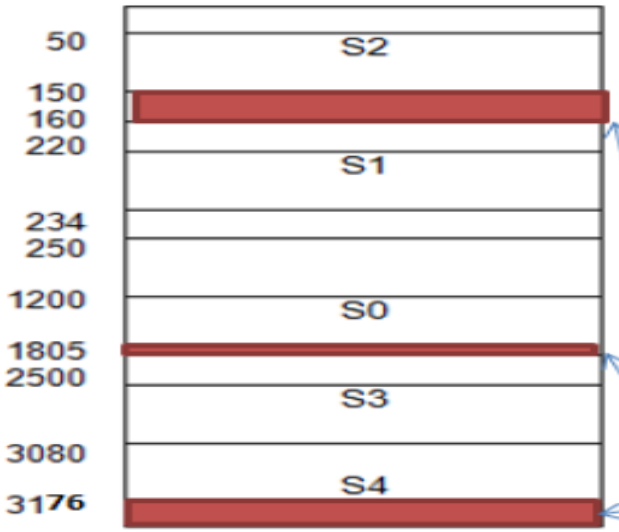


Numerical

The logical map is:



Logical map



Physical memory

Segment	Limit	Base
S0	600	1200
S1	14	220
S2	100	50
S3	580	2500
S4	96	3080

Segment table

Internal fragmentation

Numerical

3. The physical addresses are

a) 0.580→the physical address of 0.580 = 1200+580 = 1780.

b) 1.17 → because d > limit of S1, the address is **wrong**.

c) 2.66 → the physical address of 2.66= 50 + 66 = 116

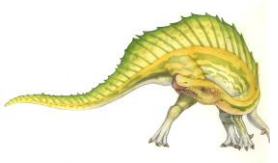
d) 3.82 → the physical address is of 3.82 is = 2500 + 82 = 2582

e) 4.20 → the physical address 4.20 = 3080+20 = 3100

Numerical

Consider a paging system with the page table stored in memory.

If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?
If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there)



Numerical

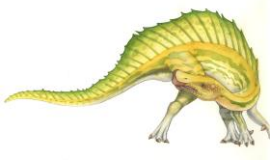
400 nanoseconds; 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

Effective access time= 0.75 * (200 nanoseconds) + 0.25 * (400 nanoseconds) = 250 nanoseconds

Numerical

Given memory partitions of 100K, 500K, 200K, 300K and 600K (in order), how would each of the First-fit, Best-fit and Worst-fit algorithms place processes of 212K, 417K, 112K and 426K (in order) ? Which algorithm makes the most efficient use of memory?
Answer:

First-fit
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition(new partition 288K=500K-212K)
426K must wait
Best-fit
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition
Worst-fit
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait



Numerical

Consider a system has 4K pages of 512 bytes in size in the logical address space . Find the number of bits in logical address space.

Ans: Number of Pages: 4K

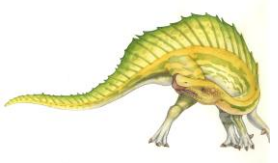
Page Size: 512B=2^9 Bytes

Number of Pages= Logical Address Space*Page Size

=4K * 2^9 Bytes

= 2^12 * 2^9 Bytes

=2^21



Numerical

Consider a system has 4K pages of 512 bytes in size in the logical address space . Find the number of bits in logical address space.

Ans: Number of Pages: 4K

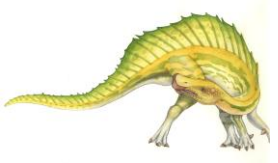
Page Size: 512B=2^9 Bytes

Number of Pages= Logical Address Space*Page Size

=4K * 2^9 Bytes

= 2^12 * 2^9 Bytes

=2^21



Numerical

If a process occupies a logical address space of 39KB in a paging system with a page size of 4KB, determine the internal fragmentation in kilobytes

Ans: No. of Pages required: 20
internal fragmentation in 1 kilobytes

Numerical

Consider a system with byte-addressable memory, 32-bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. The size of the page table in the system in megabytes is _____.

Ans:

Number of entries in page table = 2^32 / 4Kbyte

= 2^32 / 2^12 = 220

Size of page table = (No. page table entries)*(Size of an entry)

= 220 * 4 bytes = 2^22 = 4 Megabytes

End of Chapter 7