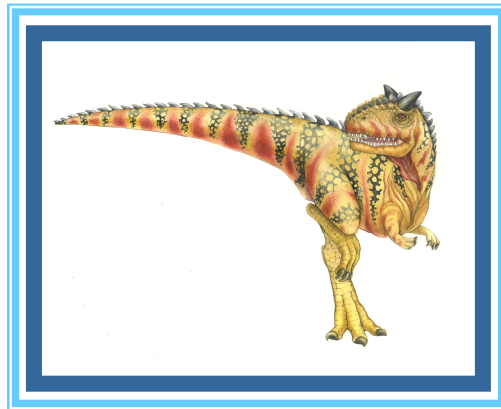


Chapter 9: Main Memory





Chapter 9: Memory Management

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





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



Human user



Non-interactive process with no human user

Human user



Human user



Process A

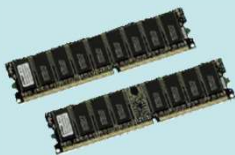
Process B

Process B

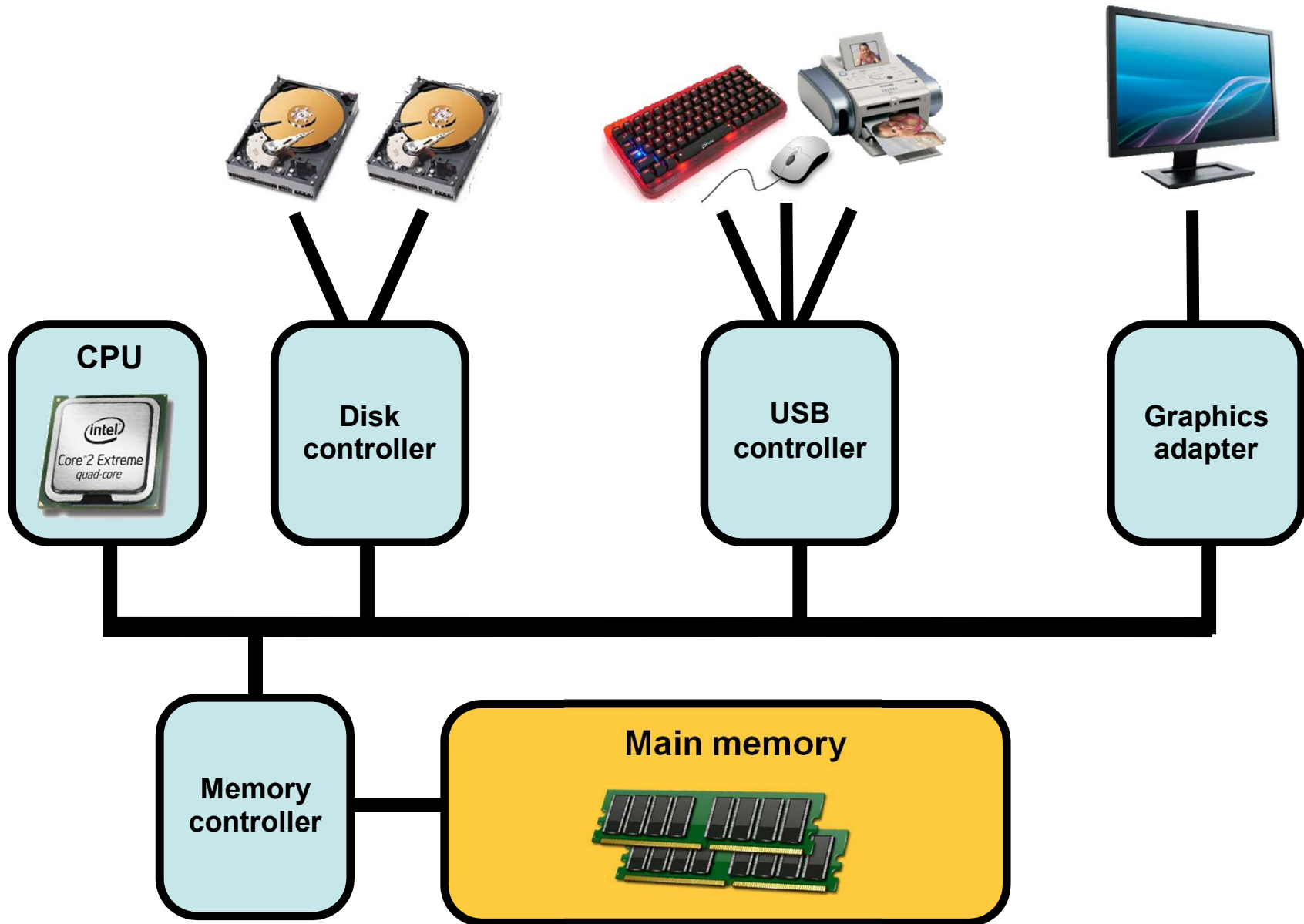
Process Z

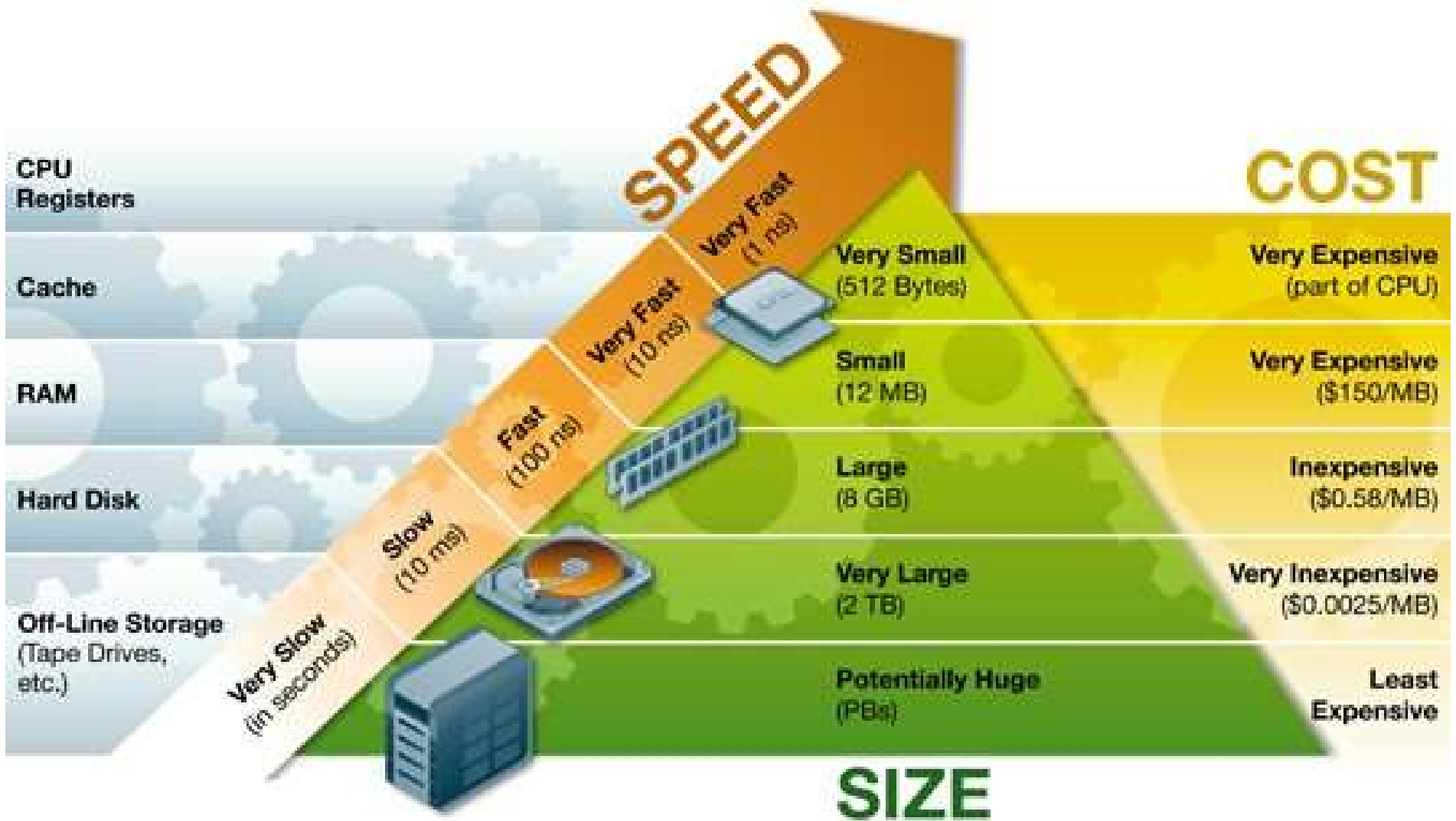
The operating systems **controls** the **hardware** and **coordinates** its use among the various application programs for the various user. An operating system provides an **environment** for the **execution** of programs.

Computer hardware



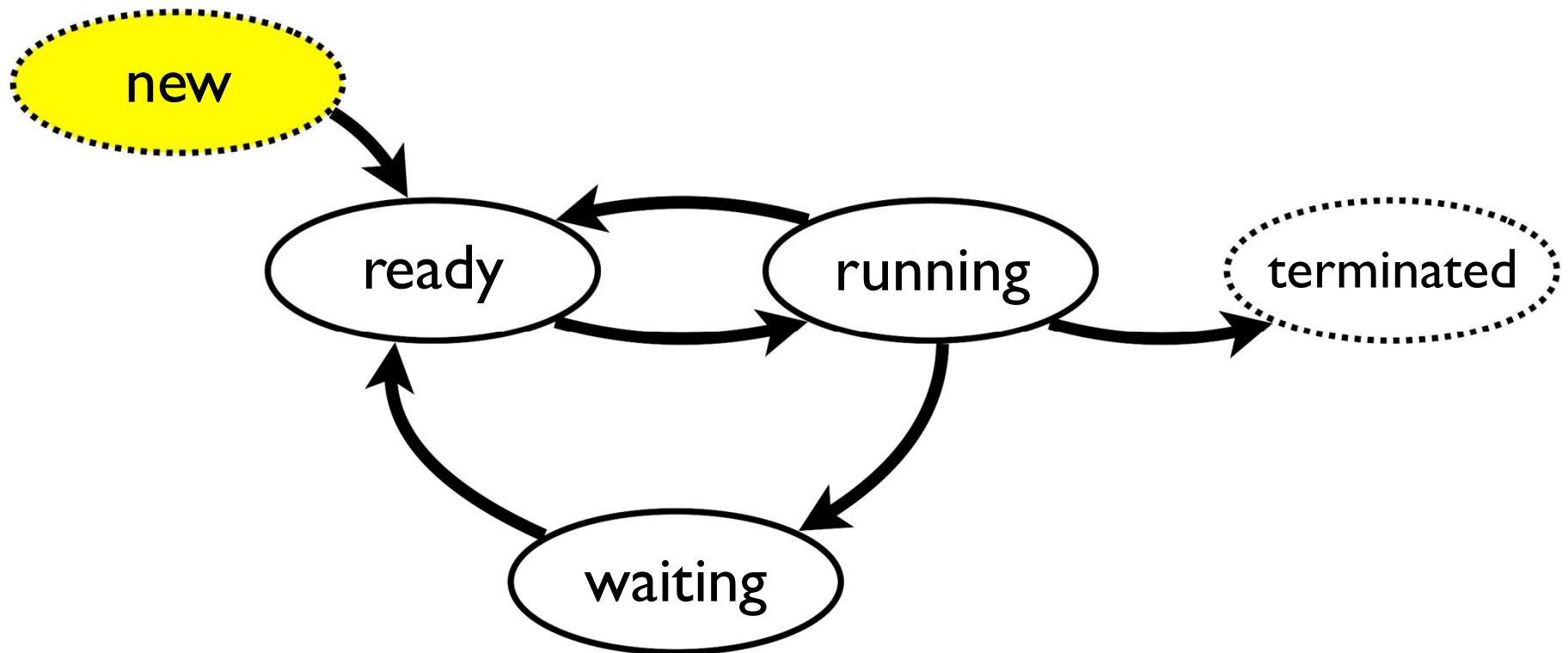
A typical computer system





Process creation

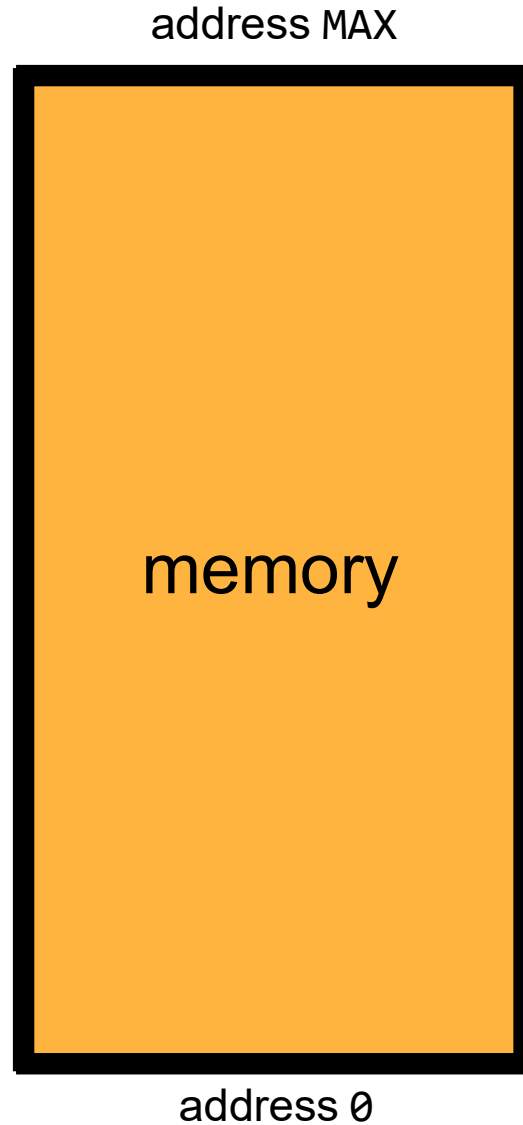
To run a program, the operating system must **fetch** the program **executable** from **disk** and place the program in a **new process** in **main memory** for it to be run.



Process memory space

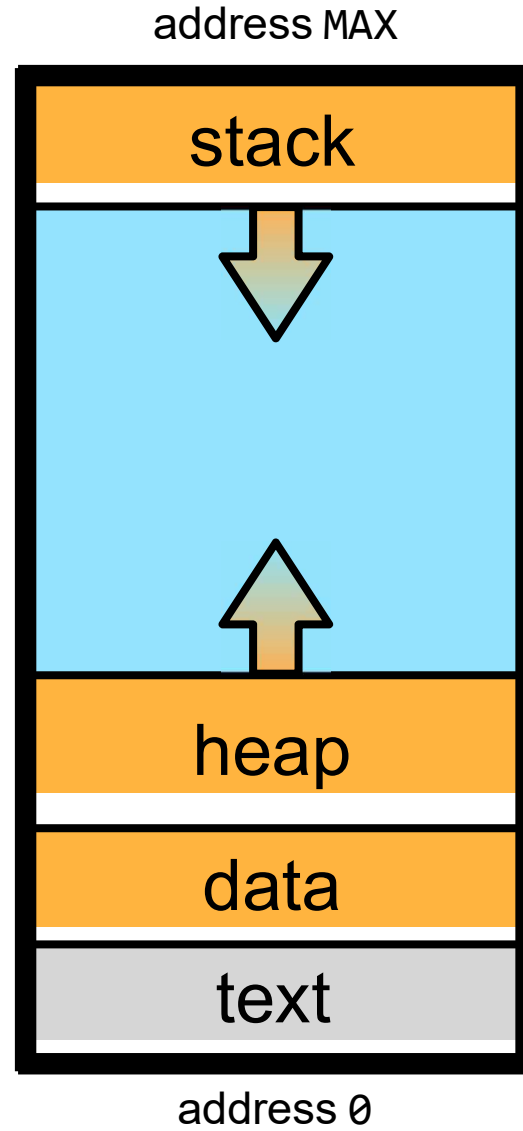
Operating system must allocate a new **memory space** for each new process. This memory space is often referred to as the process **memory image**.

Static memory allocation



The operating system must allocate a blob of memory for the new process memory image.

Static memory allocation



The allocated memory image is divided into the following segments:

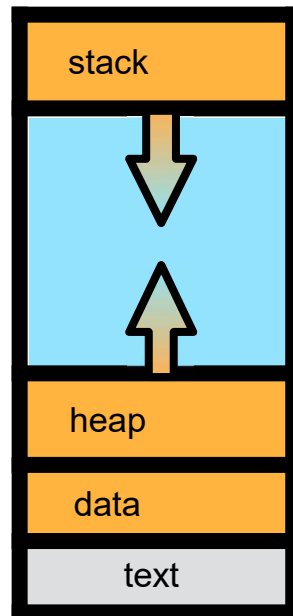
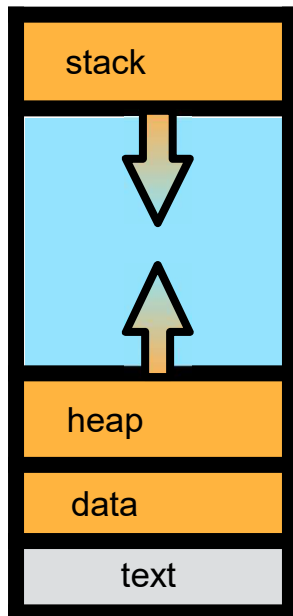
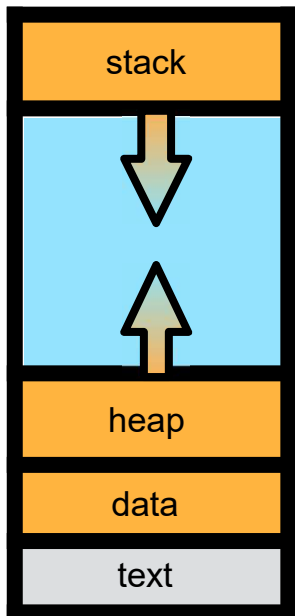
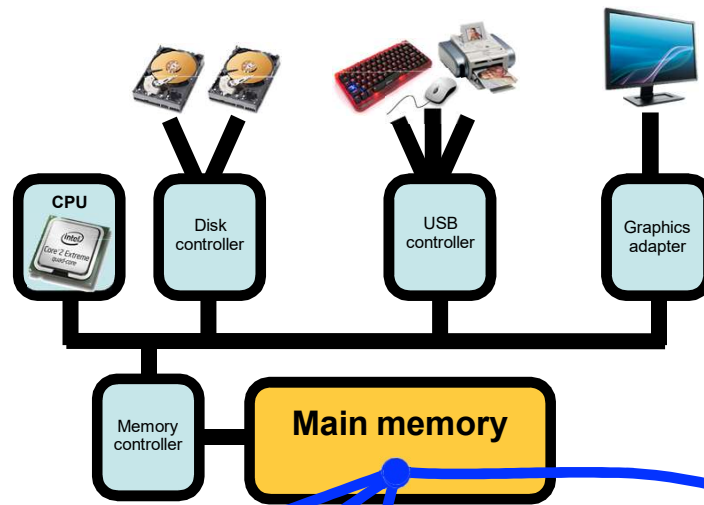
- ▶ text
- ▶ data
- ▶ heap
- ▶ stack



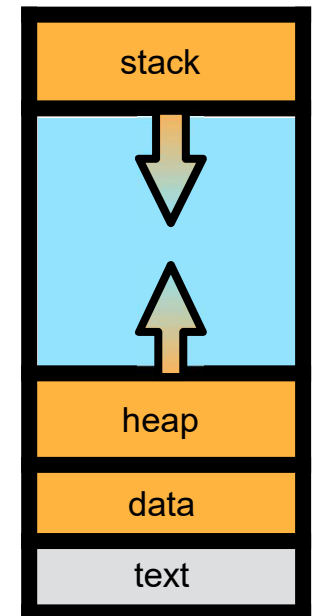
Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Memory Management
 - The task of allocating main memory to the operating system and the various user processes, in the most efficient way possible
- Main memory and registers are the only storage CPU can access directly.
- Register access in one CPU clock (or less).
- Main memory can take many cycles.
- Cache sits between main memory and CPU registers.





How can the **available memory** be allocated by the various processes?



Memory Management



- Contiguous Allocation
 - Single Allocation
 - Fixed (static) Partitioning
 - Dynamic Partitioning
 - Buddy Algorithm
- Page Memory Management
- Segmented Memory Management

Contiguous Memory Allocation

- 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

- Fixed (static) partitioning
 - ▣ Available memory is partitioned into regions with fixed boundaries
 - ▣ Each partition may contain exactly one process
 - ▣ When a partition is free, a process is selected from the input queue and is loaded into the free partition

Single contiguous allocation

Single allocation is the simplest memory management technique. All the computer's memory, usually with the exception of a small portion reserved for the operating system, is available to the single application.

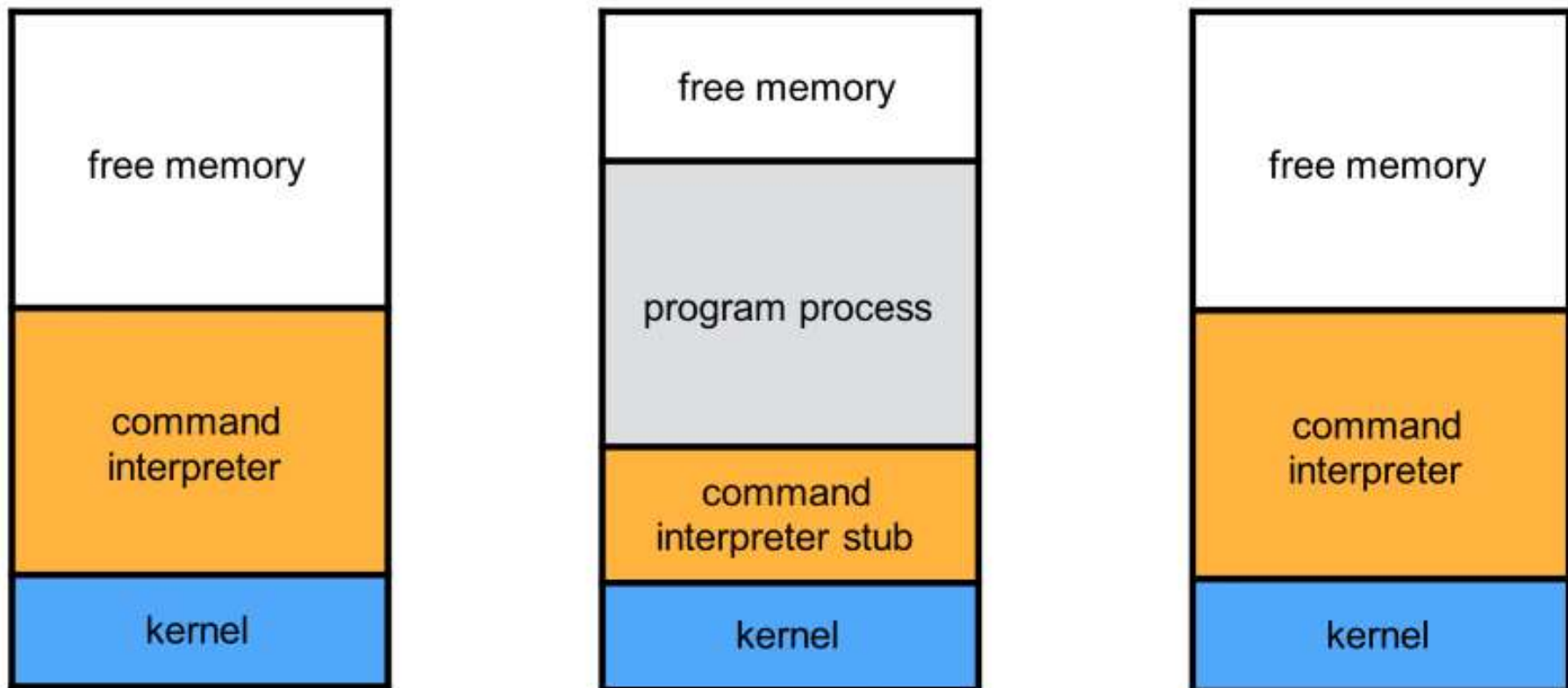
Single contiguous allocation

- ▶ **MS-DOS** (released 1981) is an example of a system which allocates memory in this way.
- ▶ An **embedded system** running a single application might also use this technique.
- ▶ A system using single contiguous allocation may still **multitask** by **swapping** the contents of memory to switch among users. Sometimes the term **single-tasking** is used instead of multi-tasking for such systems.



- ★ MS-DOS (acronym for Microsoft Disk Operating System) is a operating system for x86-based personal computers mostly developed by Microsoft and initially **released 1981**.
- ★ During its lifetime, several competing products were released for the x86 platform, and MS-DOS went through eight versions, until **development ceased in 2000**.
- ★ MS-DOS is a **single-tasking** operating system.

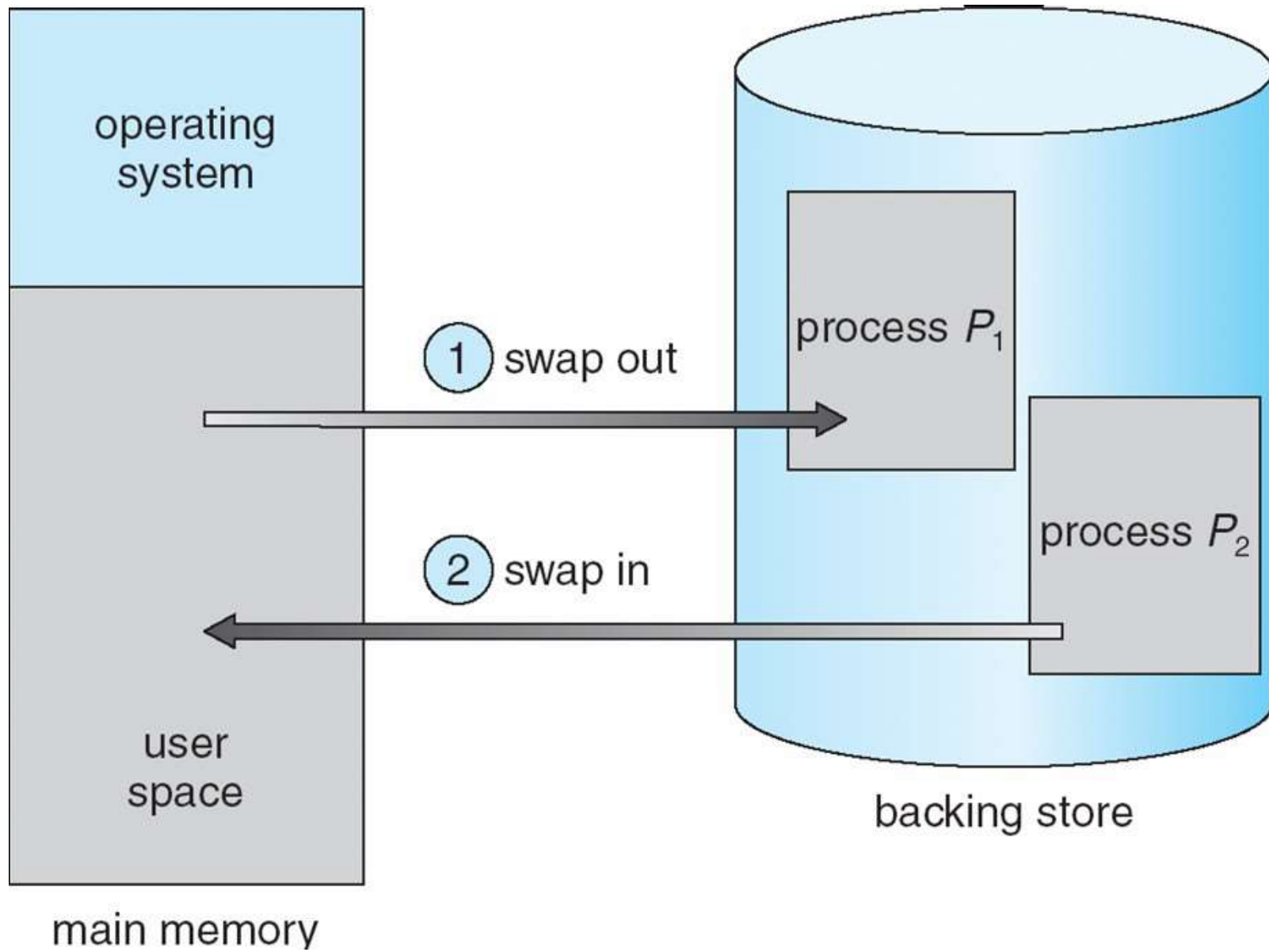
- ▶ MS-DOS is a single-tasking operating system. It has a command interpreter that is invoked when the computer is started.
- ▶ MS-DOS simply loads a program into memory, writing over most of the command interpreter to give the program as much memory as possible.
- ▶ When the program terminates, the small portion of the command interpreter that was not overwritten resumes execution. Its first task is to reload the rest of the command interpreter from disk.



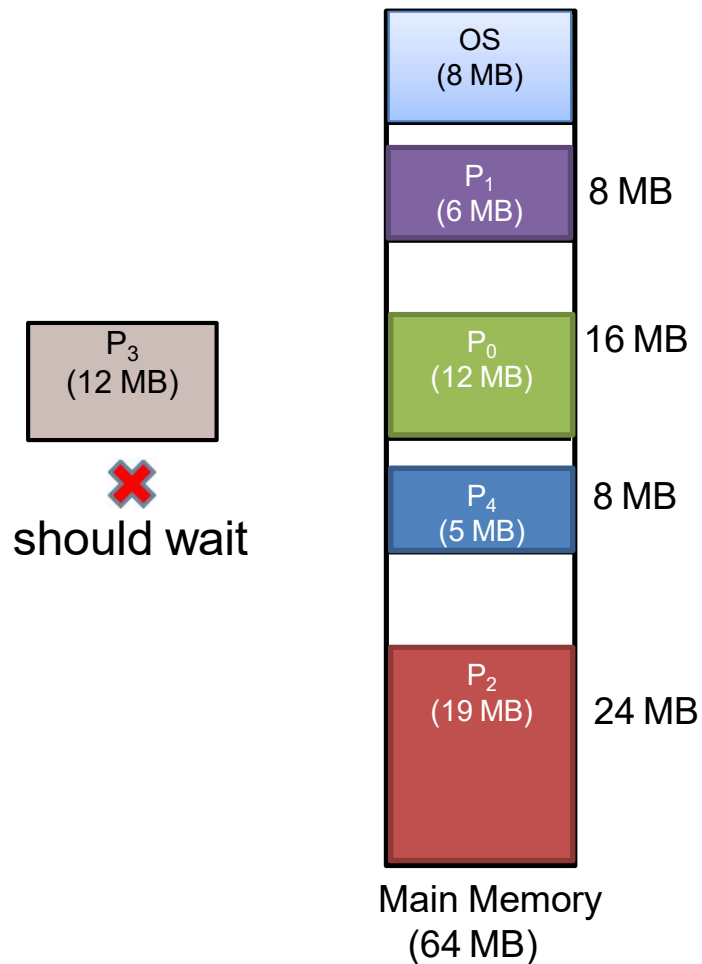
Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- ▶ **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
- ▶ Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).
- ▶ System maintains a **ready queue** of ready-to-run processes which have memory images on disk.



Fixed Partitioning

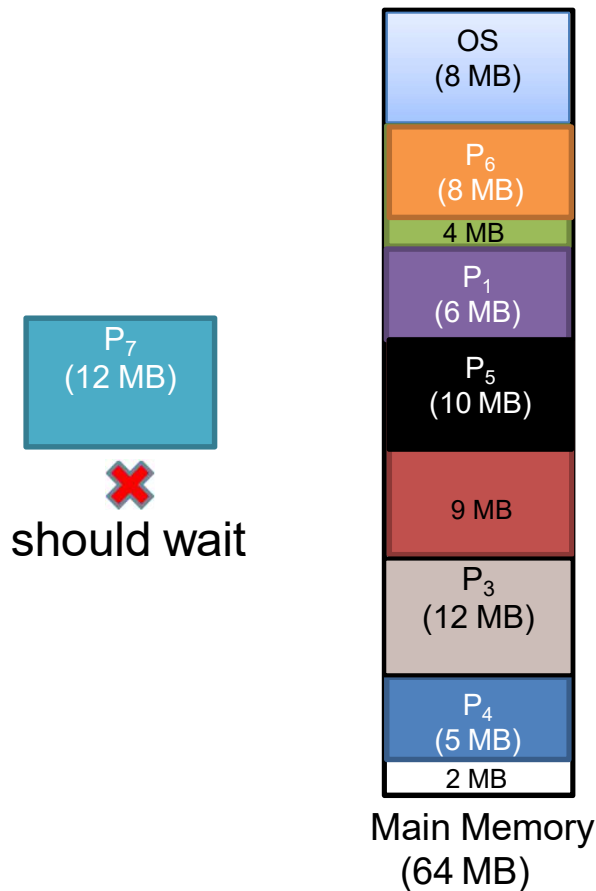


- Problems
 - The degree of multiprogramming is bound by the number of partitions
 - A process bigger than the biggest partition cannot be executed
 - **Internal Fragmentation**
 - There is enough total memory space to satisfy a request, but the available spaces do not belong to a single partition

Dynamic Partitioning

- Partitions are of variable length and number
- How it works
 - ▣ Initially, whole available memory is considered one large free partition
 - ▣ When a process arrives, we search for a free partition large enough for this process ([how?](#) – see next slide)
 - ▣ If we find one, we allocate [exactly](#) as much memory as is needed
 - ▣ When a process terminates, it releases its partition, which is then merged to its adjacent partitions (if any exists)

Dynamic Partitioning

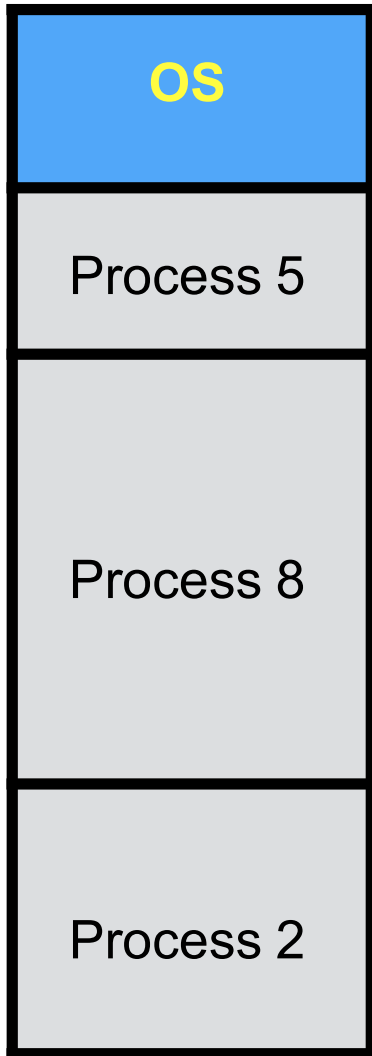


Problem

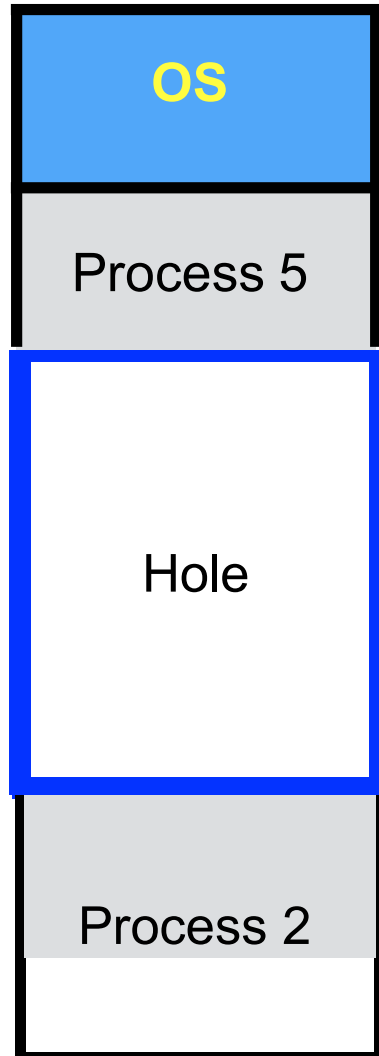
External Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces
- There is enough total memory space to satisfy a request, but the available spaces are not contiguous

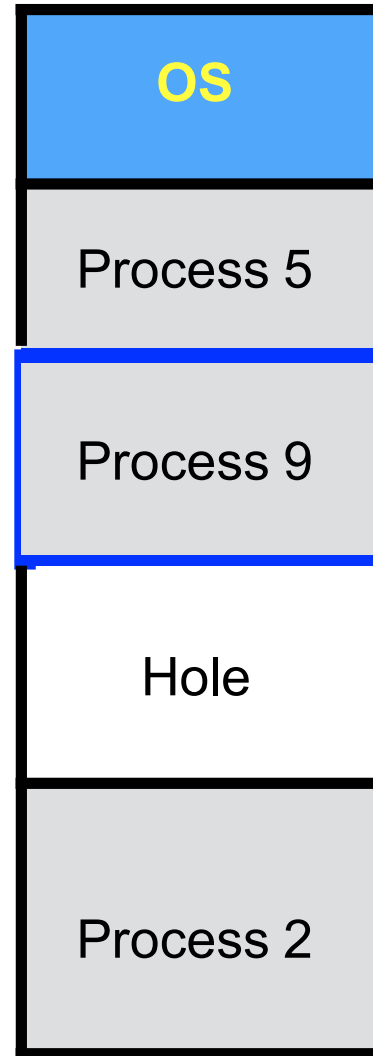
Three process occupies all available memory.



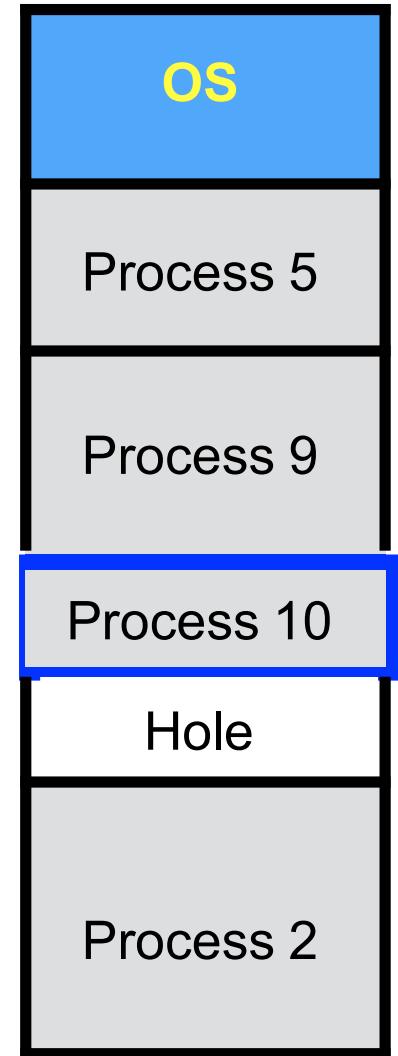
Process 8 terminates and leaves a free hole.

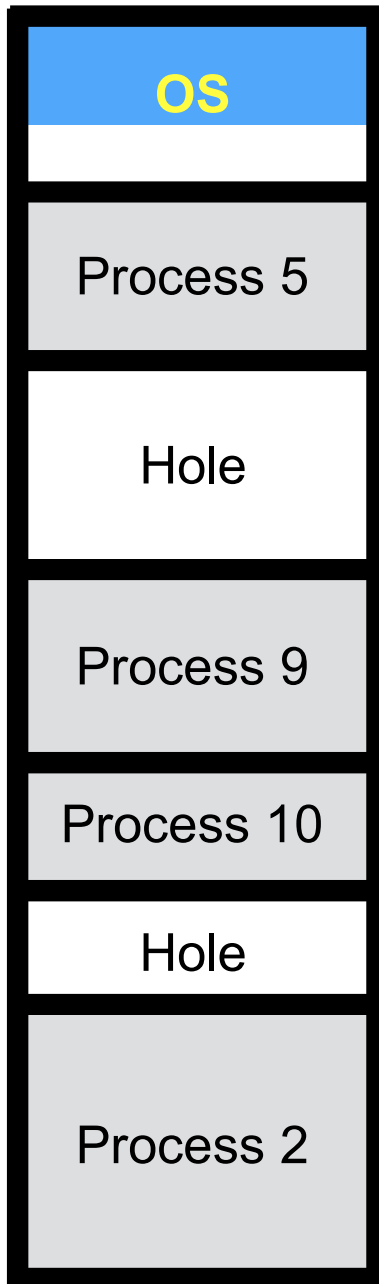


Process 9 allocates part of the hole left by process 8 and the hole of free memory shrinks.



The hole of free memory gets smaller and smaller.





Hole – block of available free 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.

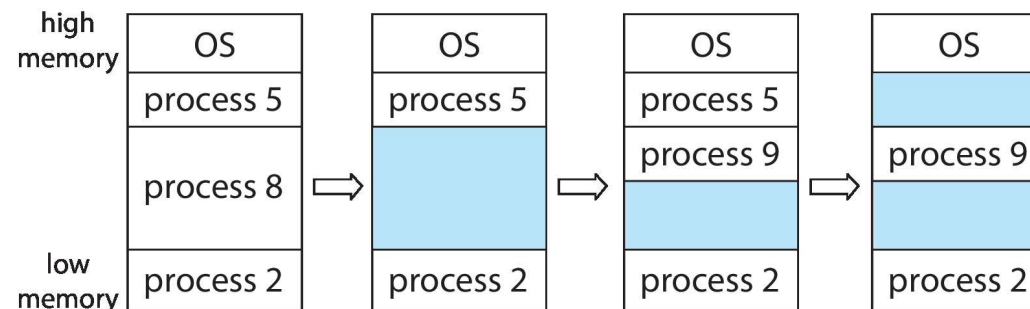
Operating system maintains information about:

- ▶ Allocated partitions.
- ▶ Free partitions (holes).



Variable Partition

- 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)





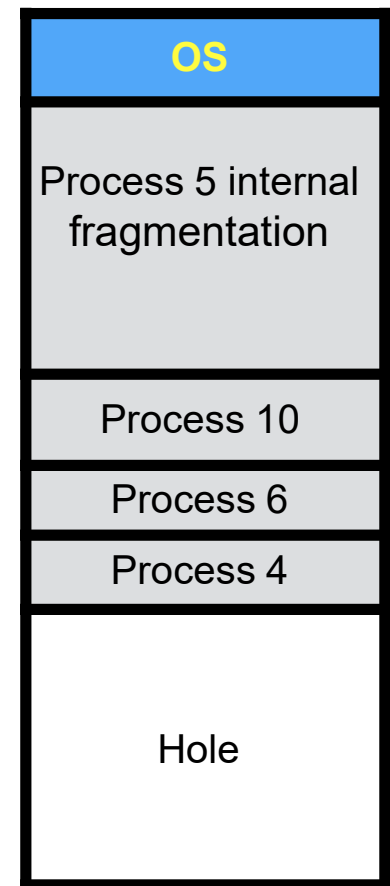
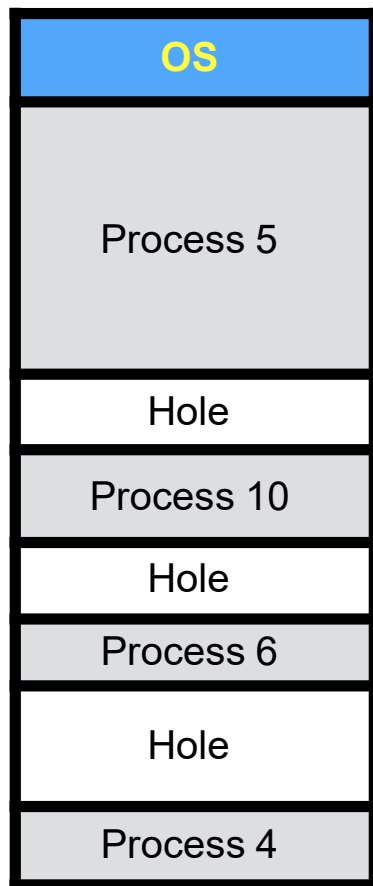
Fragmentation

- **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**



Compaction

Move all processes to one end of the address space producing one large hole at the other end of the address space.



When should compaction be done?

This method can be **expensive** (takes long time).

Compaction makes a process memory image **move around** in the physical memory during run time.

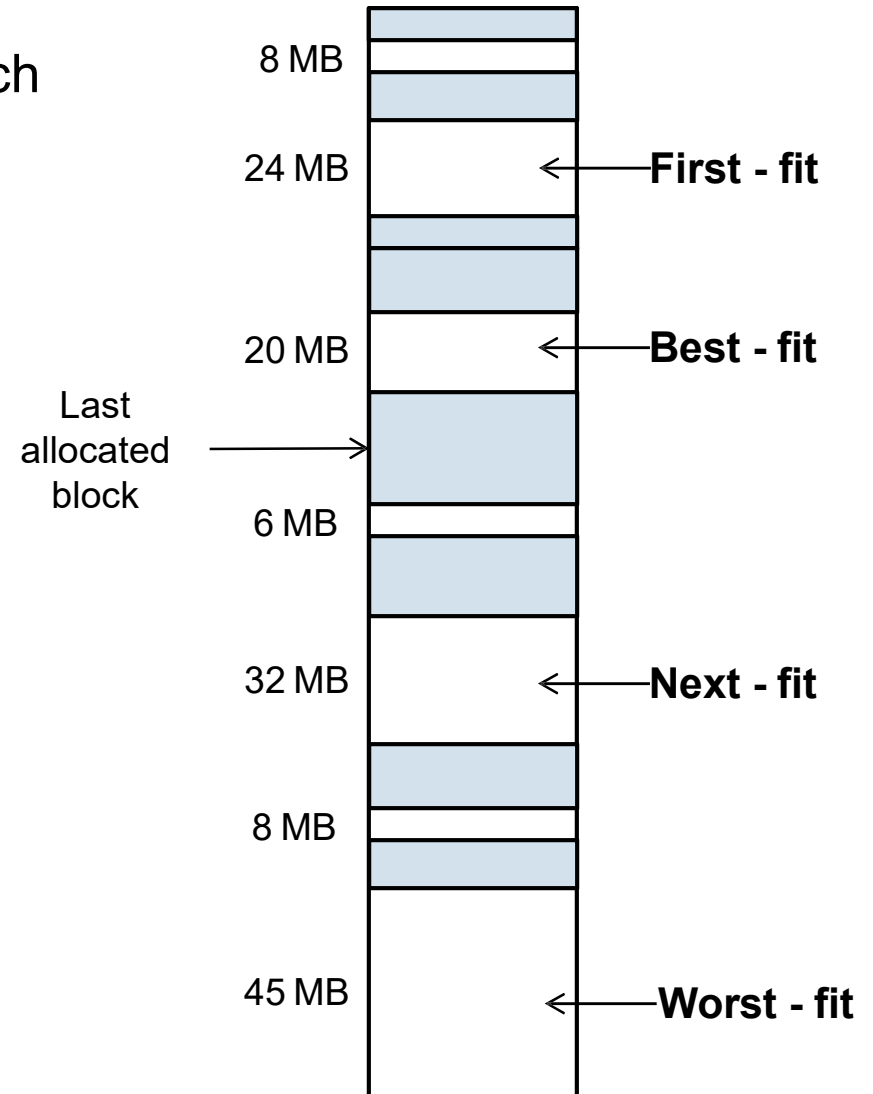
Solutions to External Fragmentation

- **Compaction**
 - From time to time, the operating system shifts the processes (if possible) so that they are contiguous and so that all of the free memory is together in one block.
 - Problem: it is a time consuming procedure and wasteful of processor time
- To permit the logical address space of the processes to be noncontiguous
 - i.e., divide the processes into smaller pieces
 - We will see **Paging** and **Segmentation** techniques

Placement Algorithms

- Operating system must decide which free block to allocate to a process
- Algorithms
 - ▣ First-fit
 - ▣ Next-fit
 - ▣ Best-fit
 - ▣ Worst-fit
- Example
 - ▣ A new process = 16 MB

Simulations show that first-fit and best-fit performs better than worst-fit in terms of speed and storage utilization.



Placement Algorithms

- First-fit
 - ▣ Scans memory from the beginning and allocate the **first** hole that is big enough
 - ▣ Is not only the simplest but usually the best and fastest as well

- Next-fit
 - ▣ Begins to scan memory from the location of the last placement and allocate the next hole that is big enough

- Best-fit
 - ▣ Allocate the **smallest** hole that is big enough
 - ▣ Worst performer overall; produces the smallest leftover hole

- Worst-fit
 - ▣ Allocate the **largest** hole

Buddy System

- The buddy system is a reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes
- Entire space available is treated as a single block of 2^U
- If a request of size s where $2^{U-1} < s \leq 2^U$
 - ▣ entire block is allocated
- Otherwise block is split into two equal buddies
 - ▣ Process continues until smallest block greater than or equal to s is generated

Buddy System

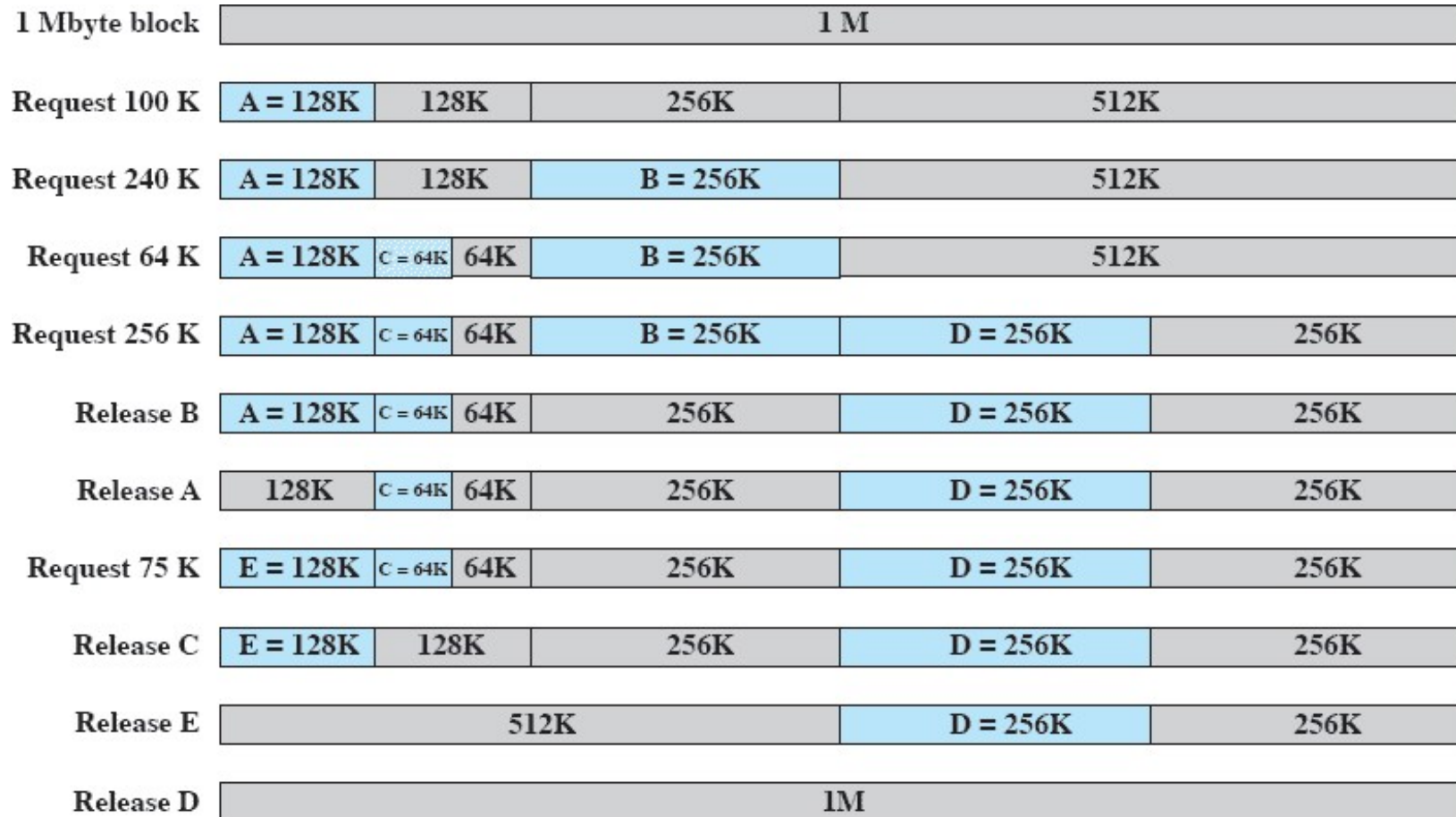


Figure 7.6 Example of Buddy System

Buddy System

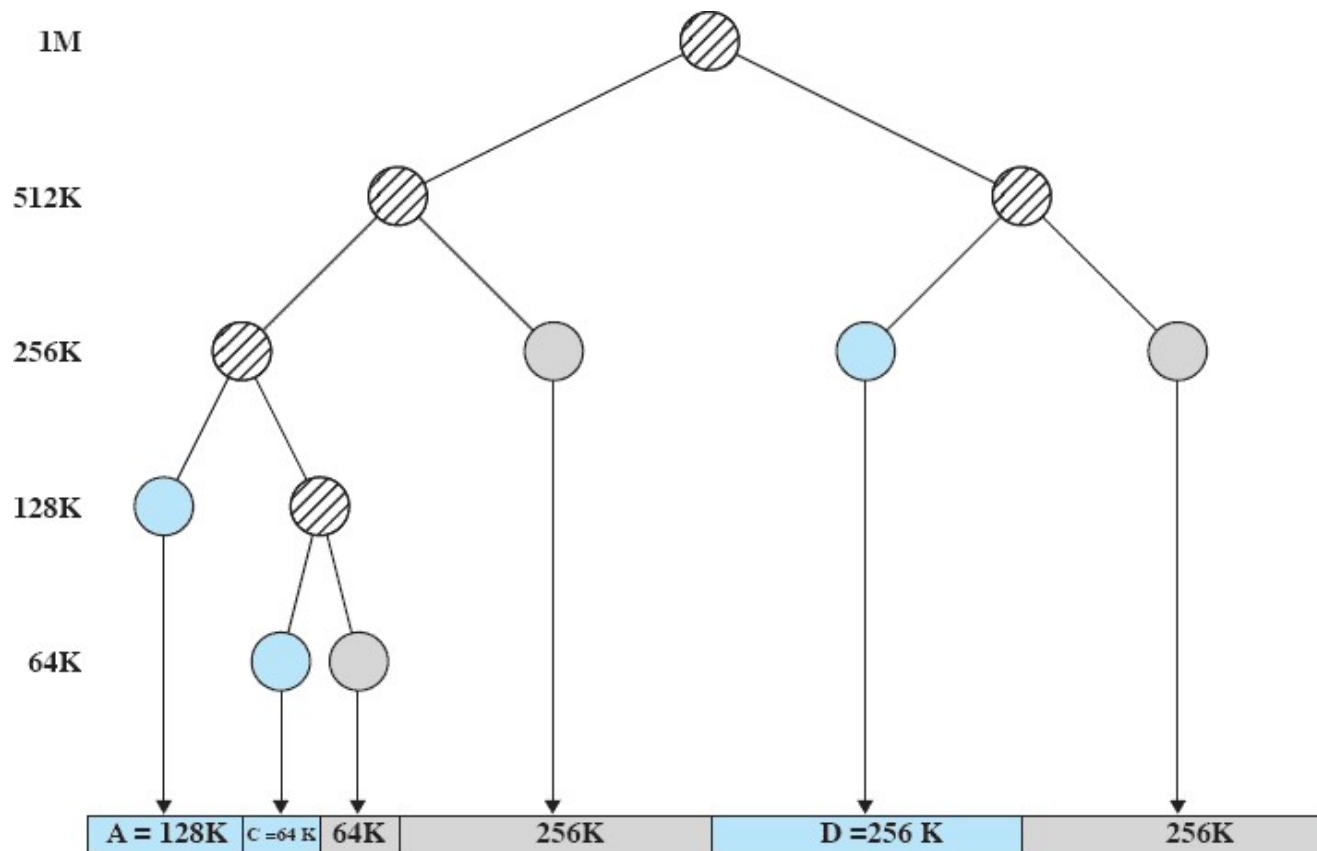


Figure 7.7 Tree Representation of Buddy System

Address Binding

- Logical Address
 - ▣ Reference to a memory location independent of the current assignment of data to memory
 - ▣ A translation must be made to a physical address before the memory access can be achieved
 - ▣ Silberschatz: address generated by the CPU
- Relative Address
 - ▣ Is an example of logical address, in which the address expressed as a location relative to some known point
- Physical or AbsoluteAddress
 - ▣ The absolute address or actual location in main memory
 - ▣ Silberschatz: address seen by the memory unit

Addresses

Logical

- Reference to a memory location independent of the current assignment of data to memory

Relative

- A particular example of logical address, in which the address is expressed as a location relative to some known point

Physical or Absolute

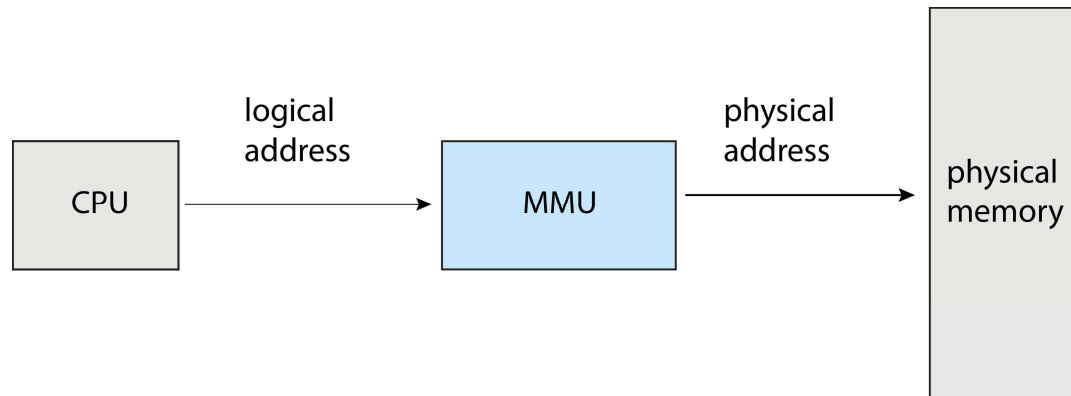
- Actual location in main memory

Address Binding

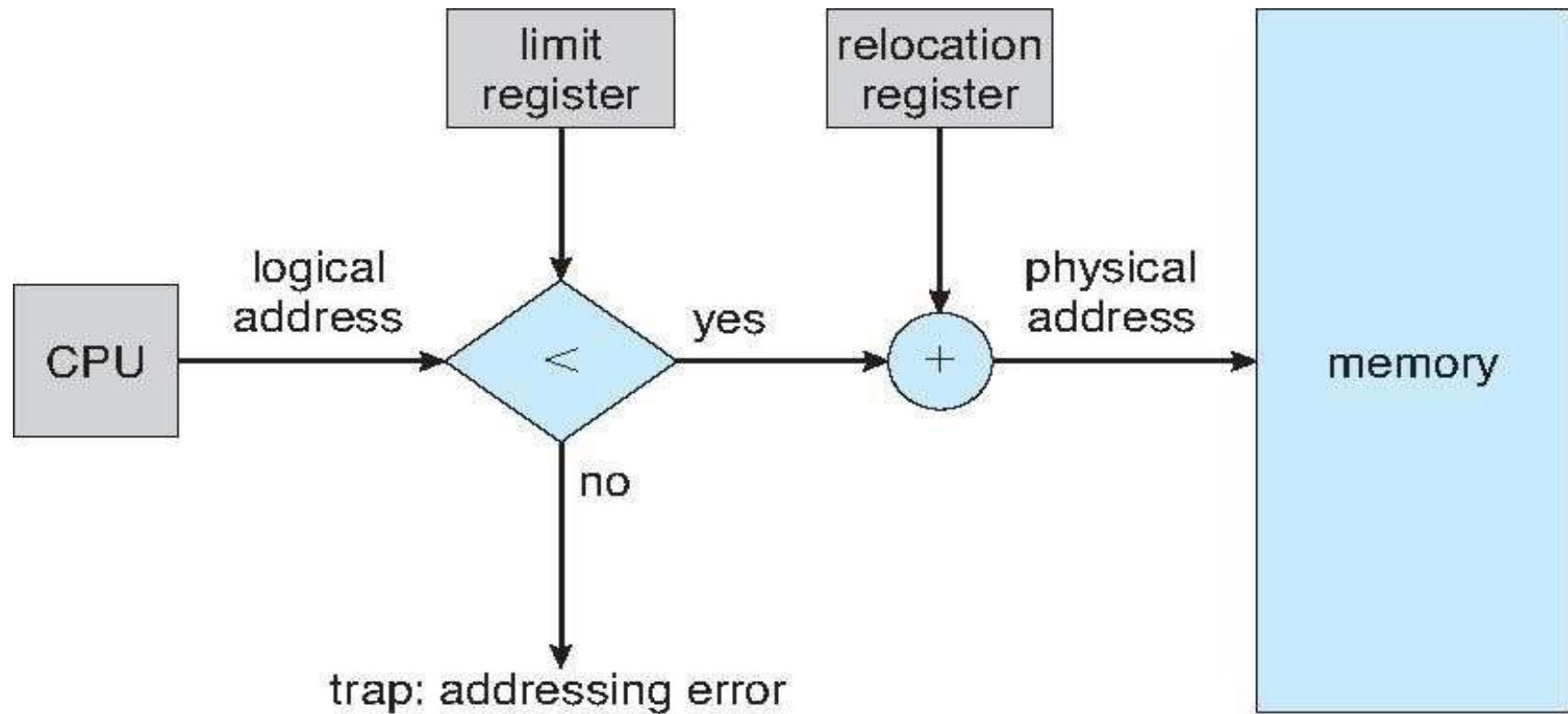
- Addresses in the source program are generally symbolic
- Address binding is the process of **mapping** the program's logical or virtual addresses to corresponding physical or main memory addresses. In other words, a given logical address is mapped by the MMU (Memory Management Unit) to a physical address.
- The binding of instructions and data to memory addresses can be done at any step along the way
 - ▣ Compile time: If you know at compile time where the process will reside in memory, then absolute code can be generated
 - ▣ Load time: Compiler generate relative addresses, final binding is delayed until load time
 - ▣ Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
 - Special hardware must be available for this scheme to work

Memory-Management Unit (MMU)

- Hardware device that at run time maps logical to physical address
- MMU in contiguous allocation uses two registers
 - ▣ Base register (or relocation register)
 - Starting address for the process
 - ▣ Bounds register (or limit register)
 - Ending location of the process
 - ▣ These values are set when the process is loaded



Address Binding



Paged memory management



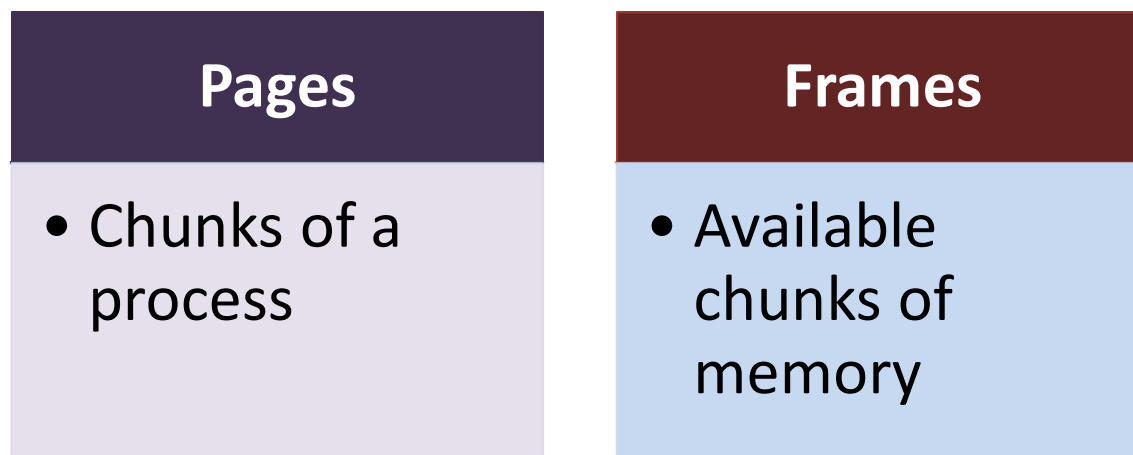
Paged allocation divides the computer's primary memory into **fixed-size** units called page **frames**, and the program's virtual address space into **pages** of the **same size**.

The hardware memory management unit **maps pages to frames**.

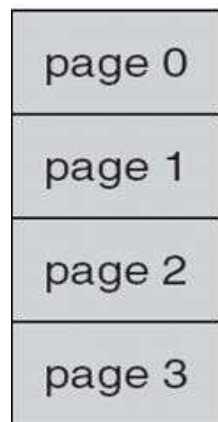
The **physical** memory can be **allocated non-contiguous** on a page basis while the logical address space appears contiguous.

Page Memory Management

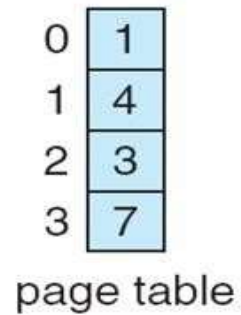
- Divide memory into small equal fixed-size chunks called **frames**
 - ▣ Size is power of 2, between 512 bytes and 16 Mbytes
- Divide each process into the same size chunks called **pages**
- To run a program of size N pages, need to find N free frames (no matter contiguous or not) and load program
- Operating system maintains a **page table** for each process
 - ▣ Contains the frame location for each page in the process



Page Memory Management

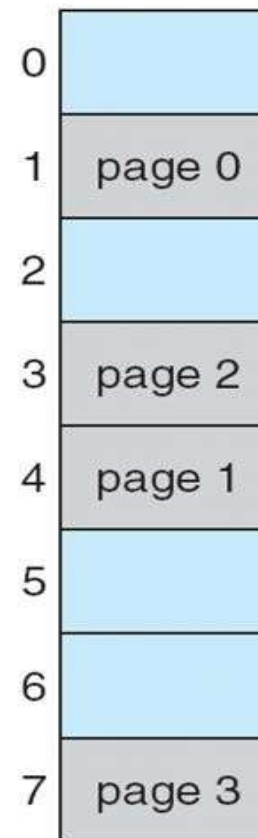


logical
memory



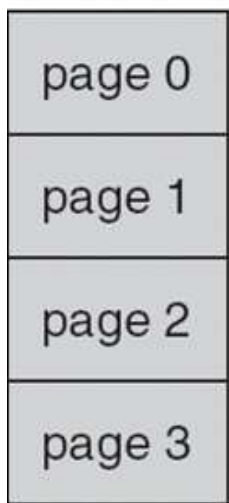
page table

frame
number



physical
memory

Currently executing process



logical memory



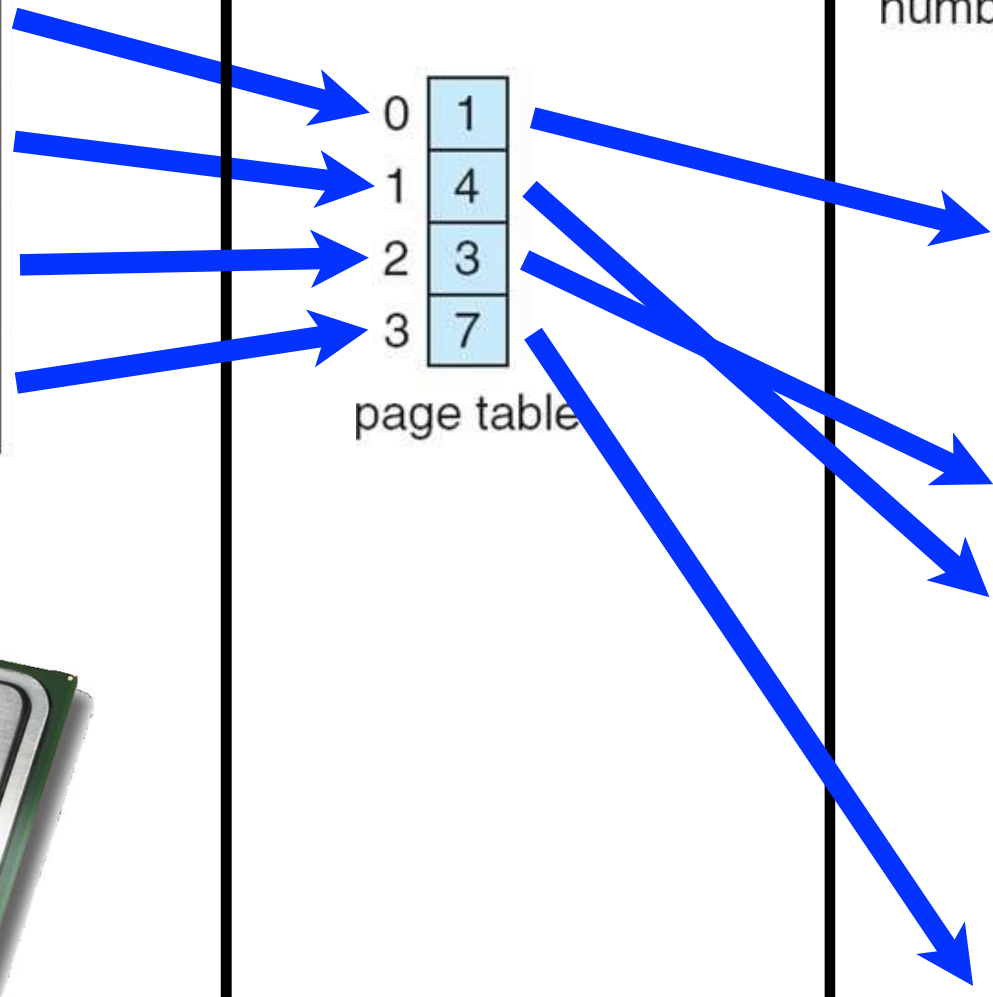
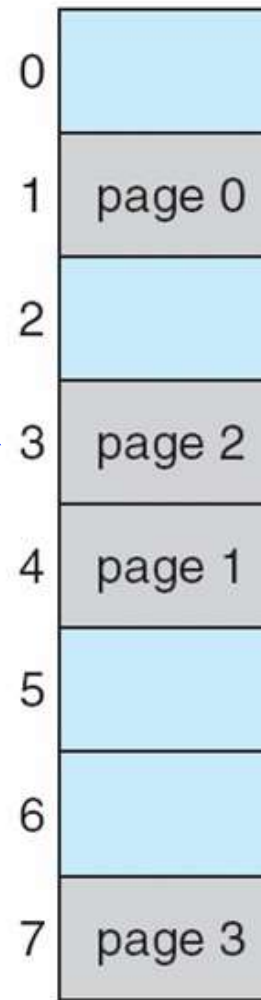
CPU

page table

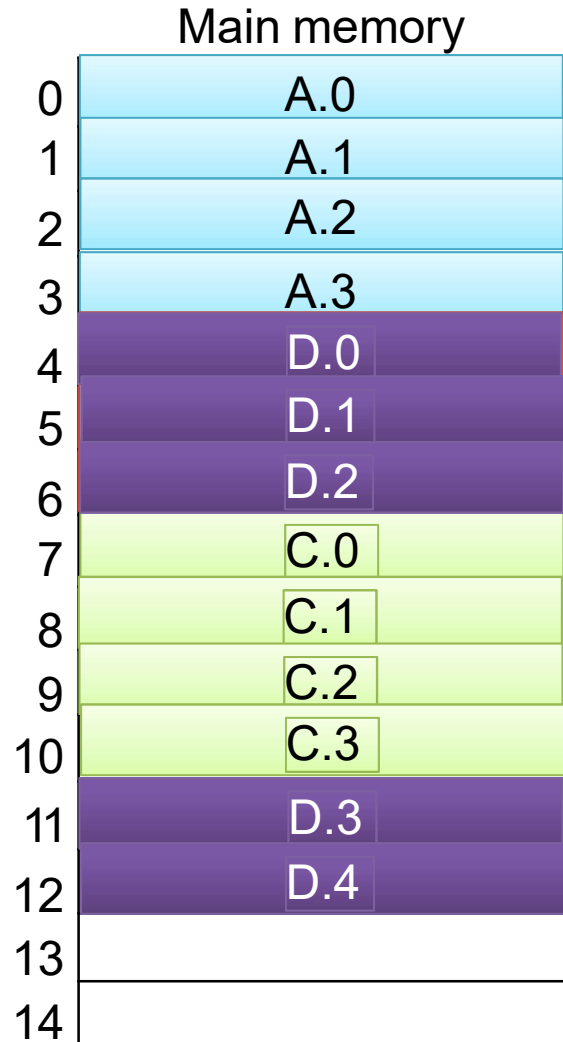
0	1
1	4
2	3
3	7

Physical memory

frame number



Page Memory Management



0	0
1	1
2	2
3	3

Process A
Page table

0	--
1	--
2	--

Process B
Page table

0	13
1	14

Free frame
list

0	7
1	8
2	9
3	10

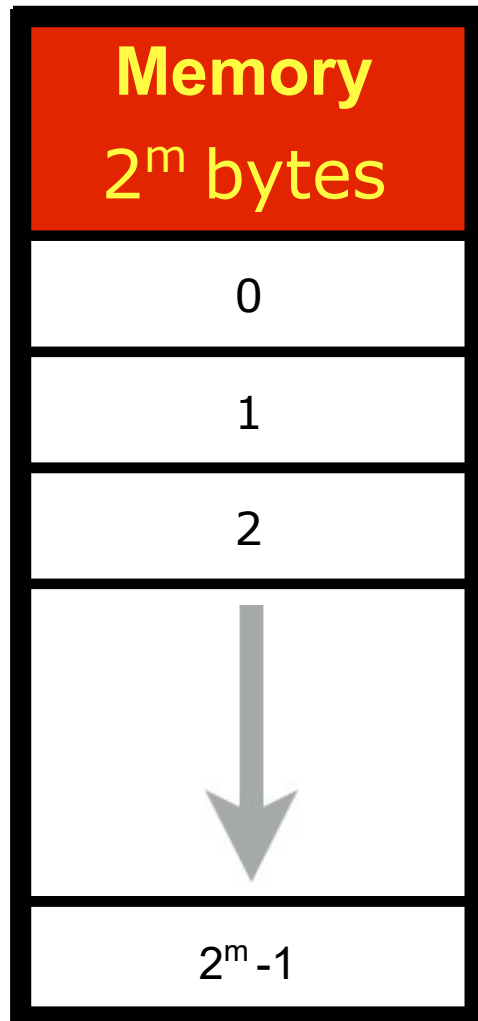
Process C
Page table

0	4
1	5
2	6
3	11
4	12

Process D
Page table

Addressing the memory

To address the memory, how many bits do we need?


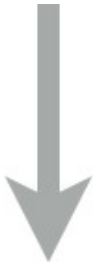


To address 2^m locations, we need a m bit address.

A = 
m bit address

Frames

Physical memory of size 2^m bytes divide into frames of size 2^n bytes each.

Memory 2^m bytes	
Frame	Size (bytes)
0	2^n
1	2^n
2	2^n
	
$[2^{(m-n)}] - 1$	2^n

We number the frames 0, 1, ...

How many frames do we get?

$$\frac{\text{Size of memory}}{\text{Size of frame}} = \frac{2^m}{2^n} = 2^{m-n}$$

We number the frames 0, 1, ..., $[2^{(m-n)}] - 1$

Pick a frame (address translation)

How can we map a logical address as seen by the CPU to a physical frame in memory?

Memory 2^m bytes	
Frame	Size (bytes)
0	2^n
1	2^n
2	2^n
↓	↓
$[2^{(m-n)}] - 1$	2^n

To address 2^m locations, we need a m bit address.

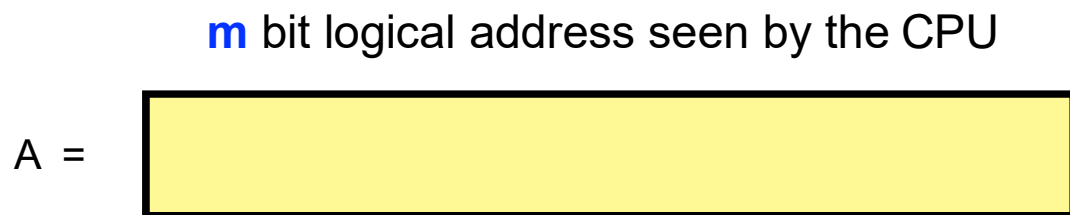
A = 
m bit address

To which of the $(m - n)$ frames should an address be mapped?

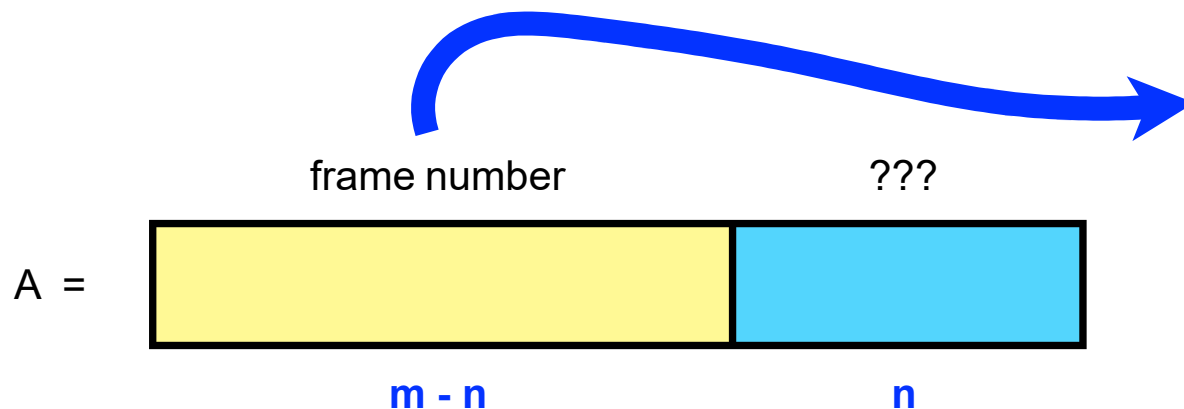
- ▶ Could map address A to frame $A \bmod (m - n)$, i.e., use the $(m - n)$ least significant bits of address A as frame number.

Pick a frame (address translation)



How can we map a logical address as seen by the CPU to a physical frame in memory?



Use $m-n$ high order bits to pick a frame.

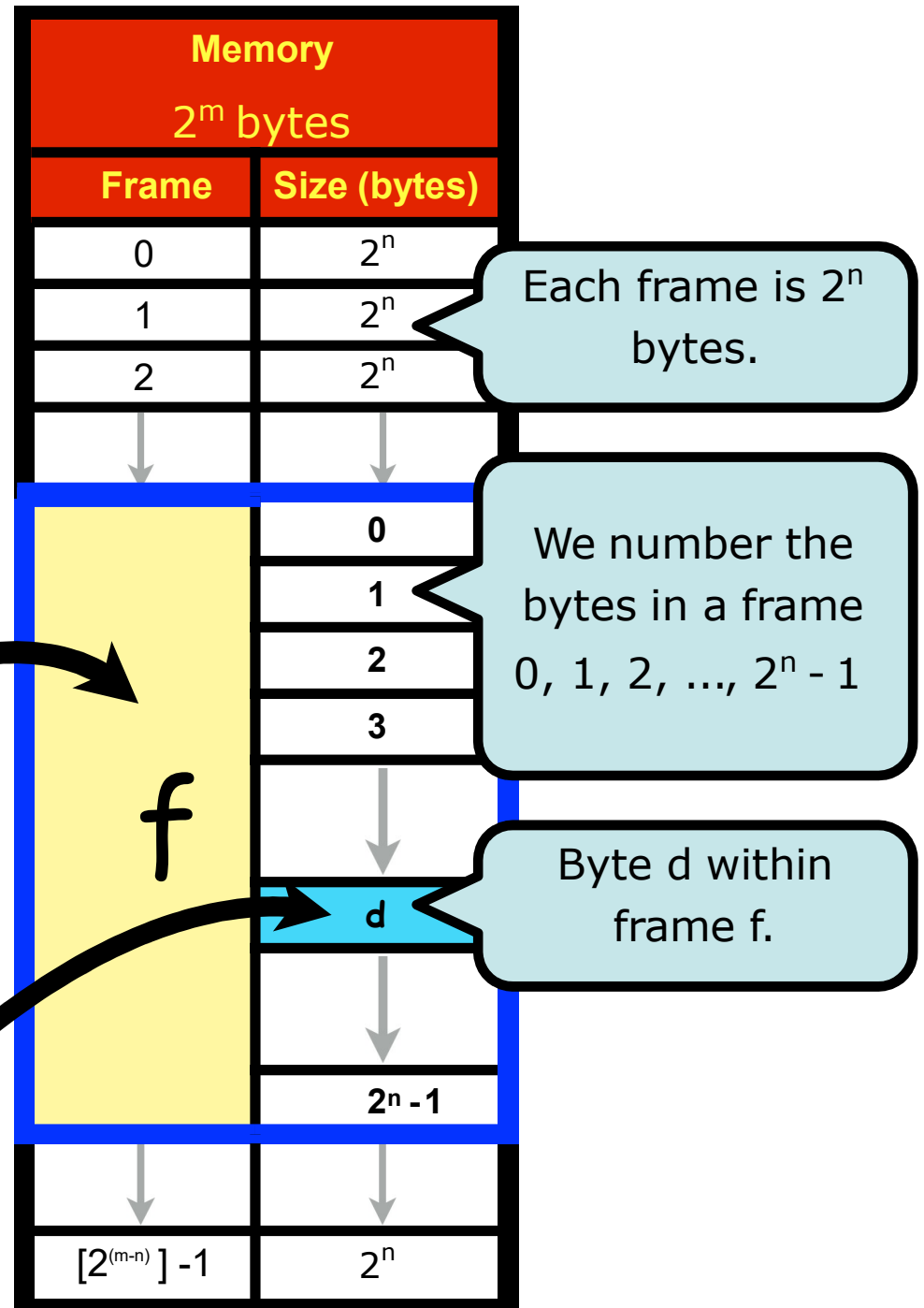
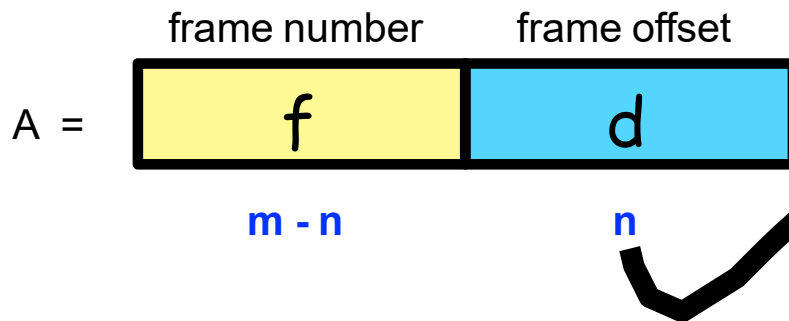


What does the n least significant bits mean?

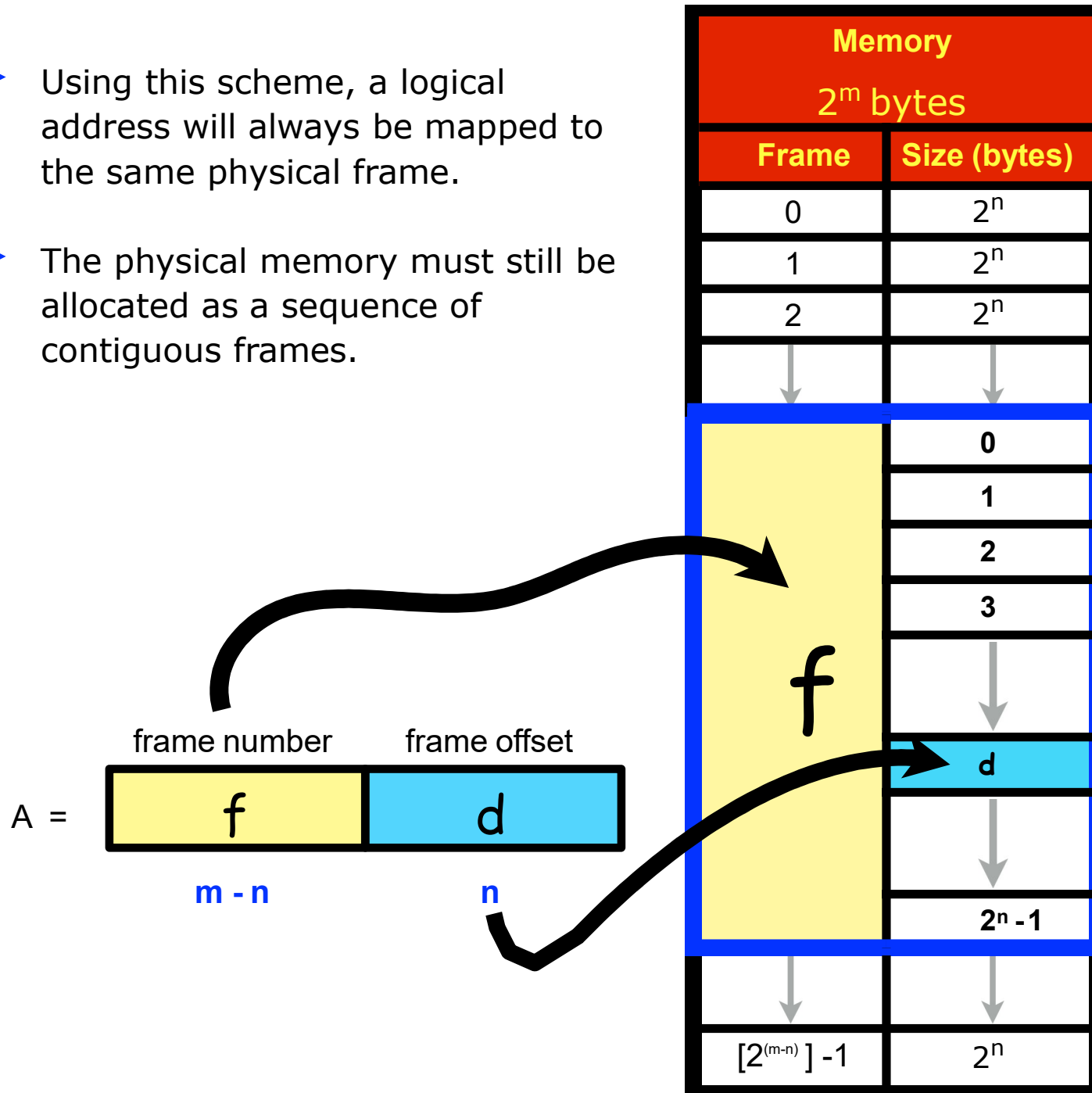
Memory 2^m bytes	
Frame	Size (bytes)
0	2^n
1	2^n
2	2^n
	
$[2^{(m-n)}] - 1$	2^n

Use $m-n$ high order bits to pick a **frame**.

The n least significant bits denotes the **byte offset** within the frame.



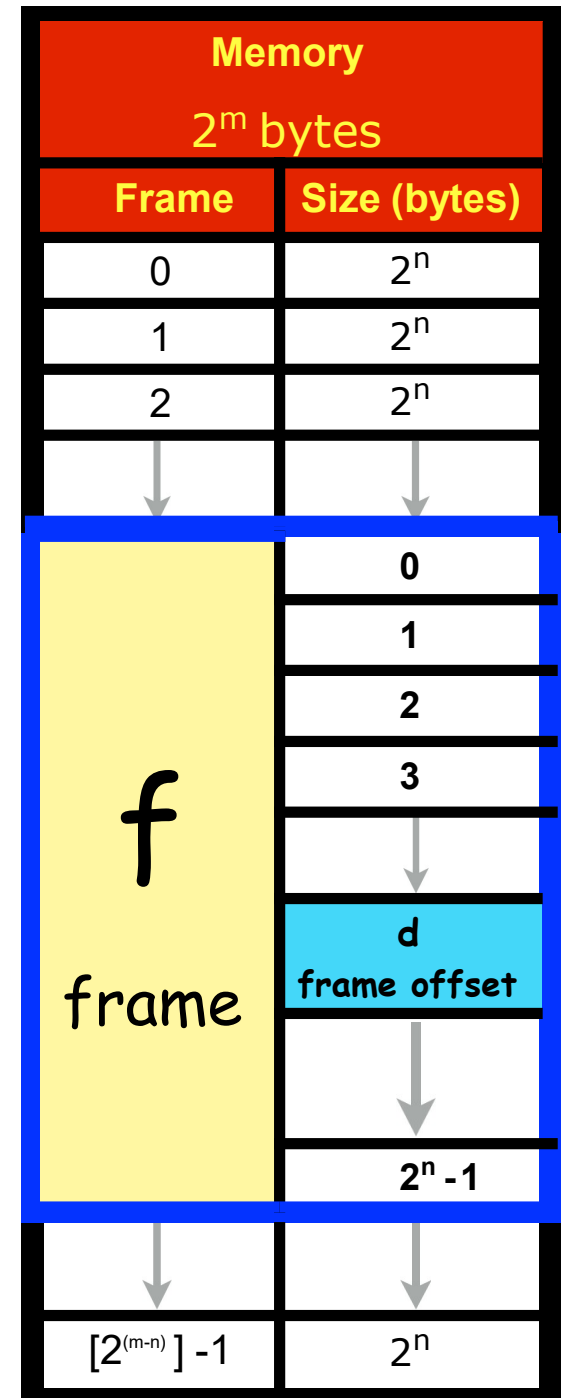
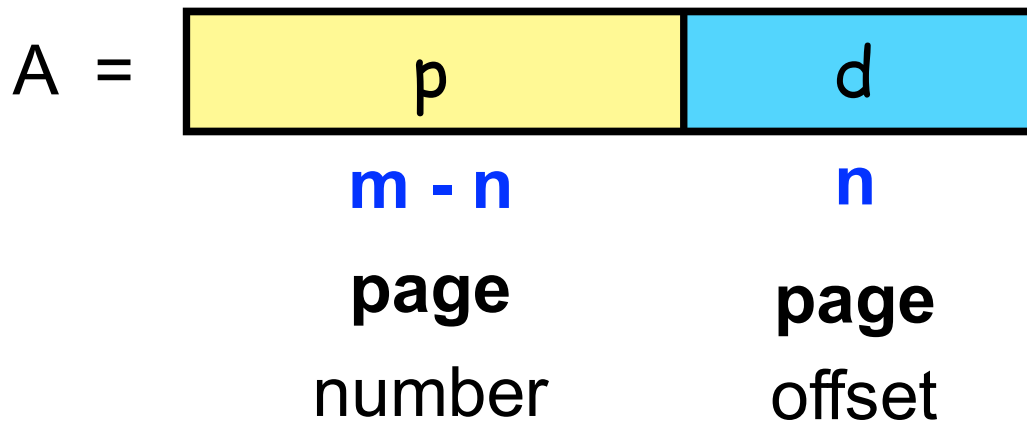
- ▶ Using this scheme, a logical address will always be mapped to the same physical frame.
- ▶ The physical memory must still be allocated as a sequence of contiguous frames.



Pages and frames

A solution that allows for non-contiguous allocation of physical frames.

- ▶ **Logical** address space divided into fixed sized **pages**.
- ▶ **Physical** memory divided into **frames** of the same fixed size as the pages.

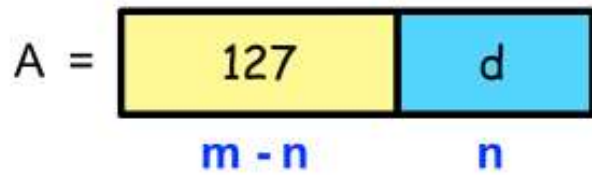


Page table

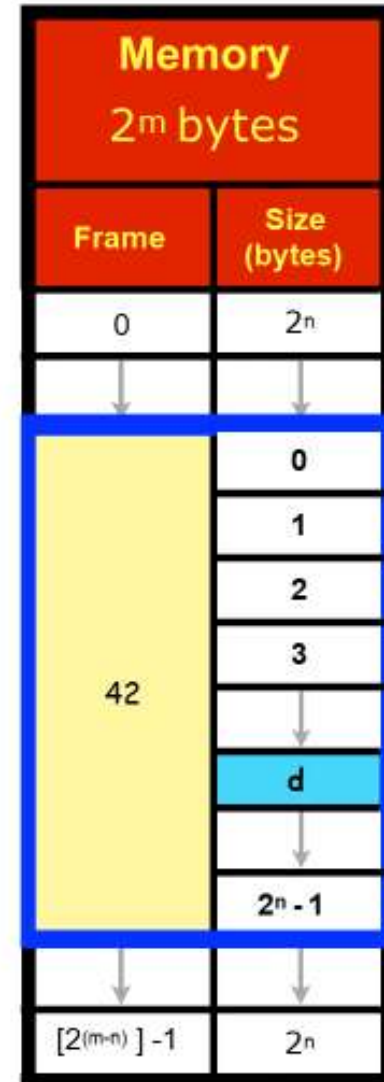
A page lookup table maps logical pages to physical frames.

Page table	
Page	Frame
0	33
1	7111
↓	↓
127	42
↓	↓
$[2^{(m-n)}] - 1$	5666

page number page offset

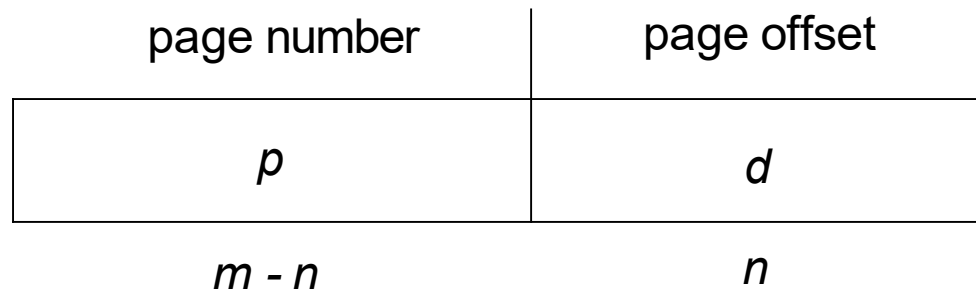


Page table	
Page	Frame
0	33
1	7111
↓	↓
127	42
↓	↓
$[2^{(m-n)}] - 1$	5666

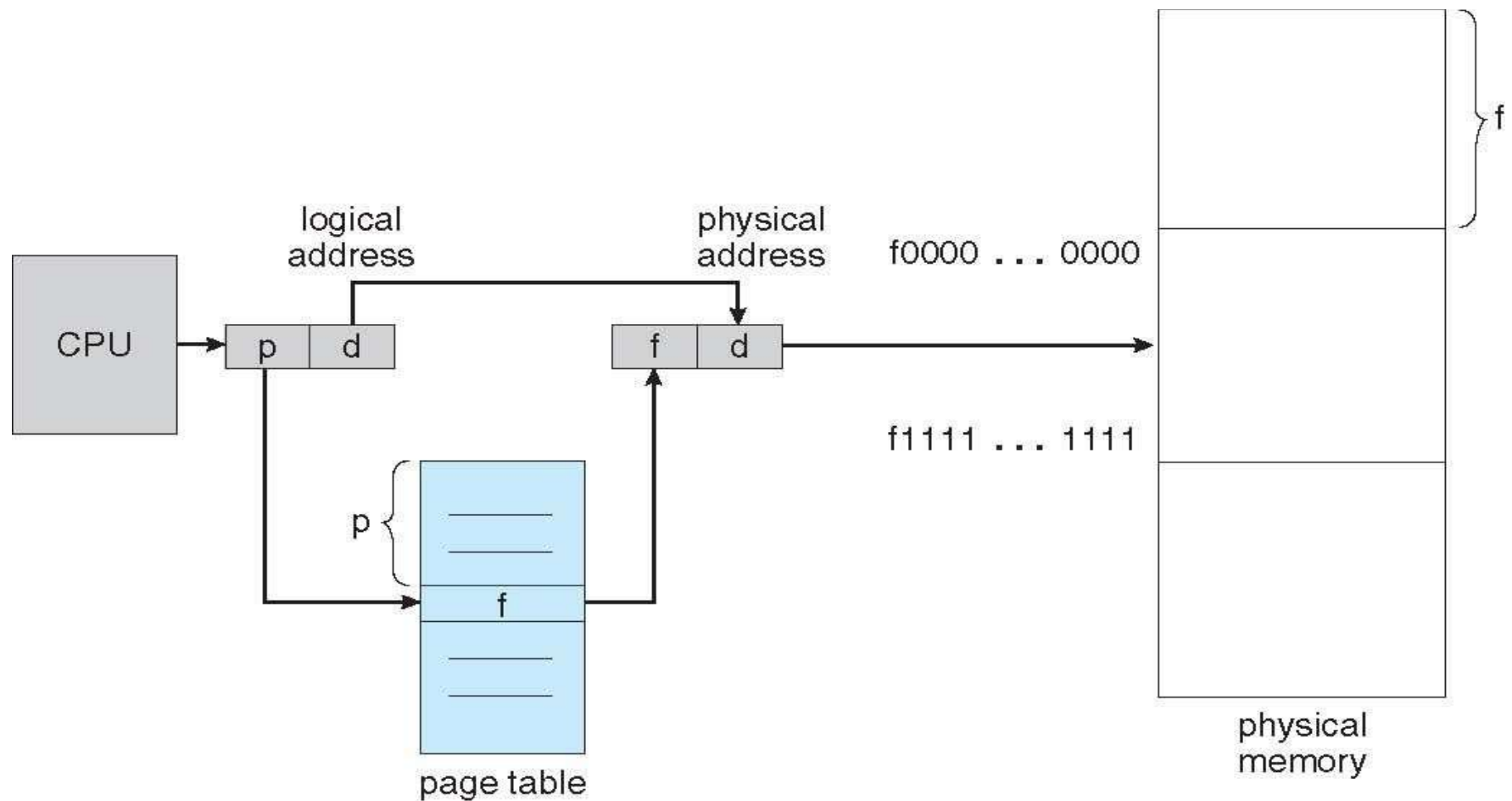


Address Translation Scheme

- Logical address generated by CPU is divided into:
 - Page number (p) – used as an index into the page table to obtain the frame number
 - Page offset (d) – is the displacement within the page. combined with the frame number to define the physical memory address
- If the size of logical address space is 2^m , and page size is 2^n , then the logical



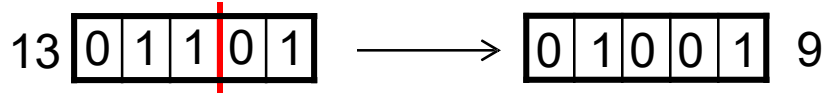
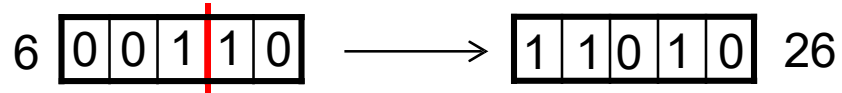
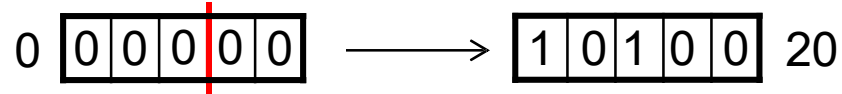
Address Translation Scheme



Address Translation Example

Logical address

Physical address



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

0	5
1	6
2	1
3	2

page table

Physical Memory = 32 Byte (m=5)

Page size = 4 Byte (n=2)

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

Paging Issues



- There is some internal fragmentation
 - ▣ The last frame allocated may not be completely full
- Page table becomes very large in a large logical address space (2^{32} to 2^{64})
- Memory access is slowed
 - ▣ Each memory access require an extra access to the page table (which is also kept in main memory)

Internal Fragmentation

- The last frame allocated may not be completely full
 - ▣ Page size = 2,048 bytes
 - ▣ Process size = 72,766 bytes = 35 pages + 1,086 bytes
 - ▣ Internal fragmentation of 2,048 - 1,086 = 962 bytes
- On average fragmentation is $\frac{1}{2}$ frame size
- Smaller page size, less amount of internal fragmentation
- But Smaller page size, more pages required per process
 - ▣ More pages per process means larger page tables
- Pages typically are between 4 KB and 8 KB in size
 - ▣ Solaris supports two page sizes – 8 KB and 4 MB

Implementation of Page Table

- The page table can be implemented as a set of dedicated registers
 - Example: DEC PDP-11, page table consists of 8 entries
 - Can be used if the page table is reasonably small
- In modern computers page tables are very large (e.g., 1 million entries) and are kept in main memory
- **Page-table base register (PTBR)** points to the page table
 - This register is stored in the PCB of the process

Implementation of Page Table

- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data/instruction
 - Thus, memory access is slowed by a factor of 2
- Solution: to use a special, small, fast lookup hardware cache, called a [translation look-aside buffer \(TLB\)](#)

Translation Look-aside Buffer (TLB)



- The TLB is associative, high-speed memory – parallel search
 - ▣ Each entry in the TLB consists of two parts: a key (or tag) and a value
 - ▣ When an item is given to TLB, the item is compared with all keys **simultaneously**. If the item is found, the corresponding value field is returned.
- The hardware is expensive, so TLBs are typically small (64 to 1,024 entries)

Translation Look-aside Buffer (TLB)

- The TLB contains only a few of the page-table entries

Page #	Frame #

- To translate a logical address, its page number is presented to the TLB
 - ▣ If page number is found (TLB **hit**), the frame number is retrieved
 - ▣ Otherwise (TLB **miss**), frame number is retrieved from the page table, and it is also added to the TLB (replacement policies can be used, if the TLB is full)

Translation Look-aside Buffer (TLB)

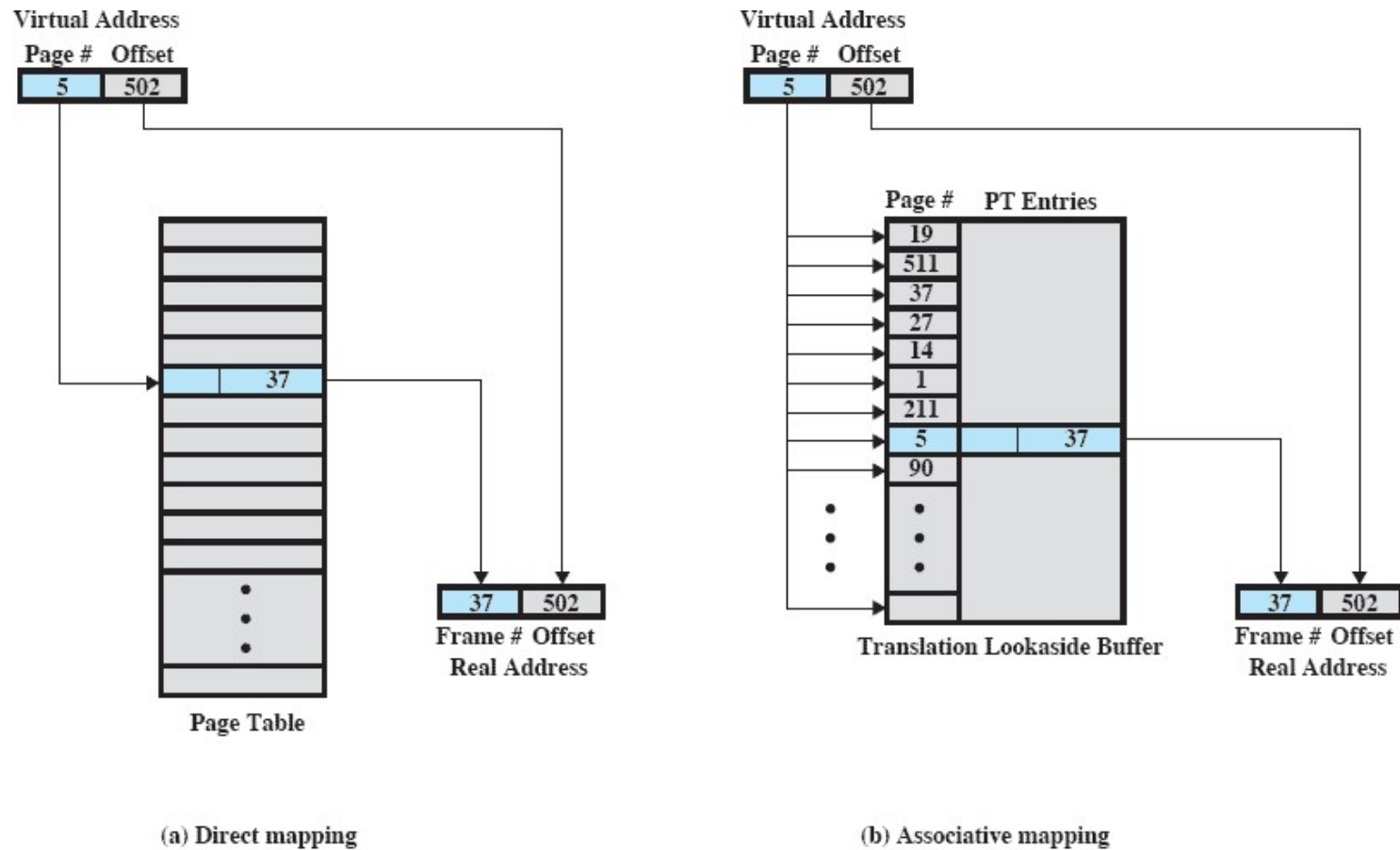
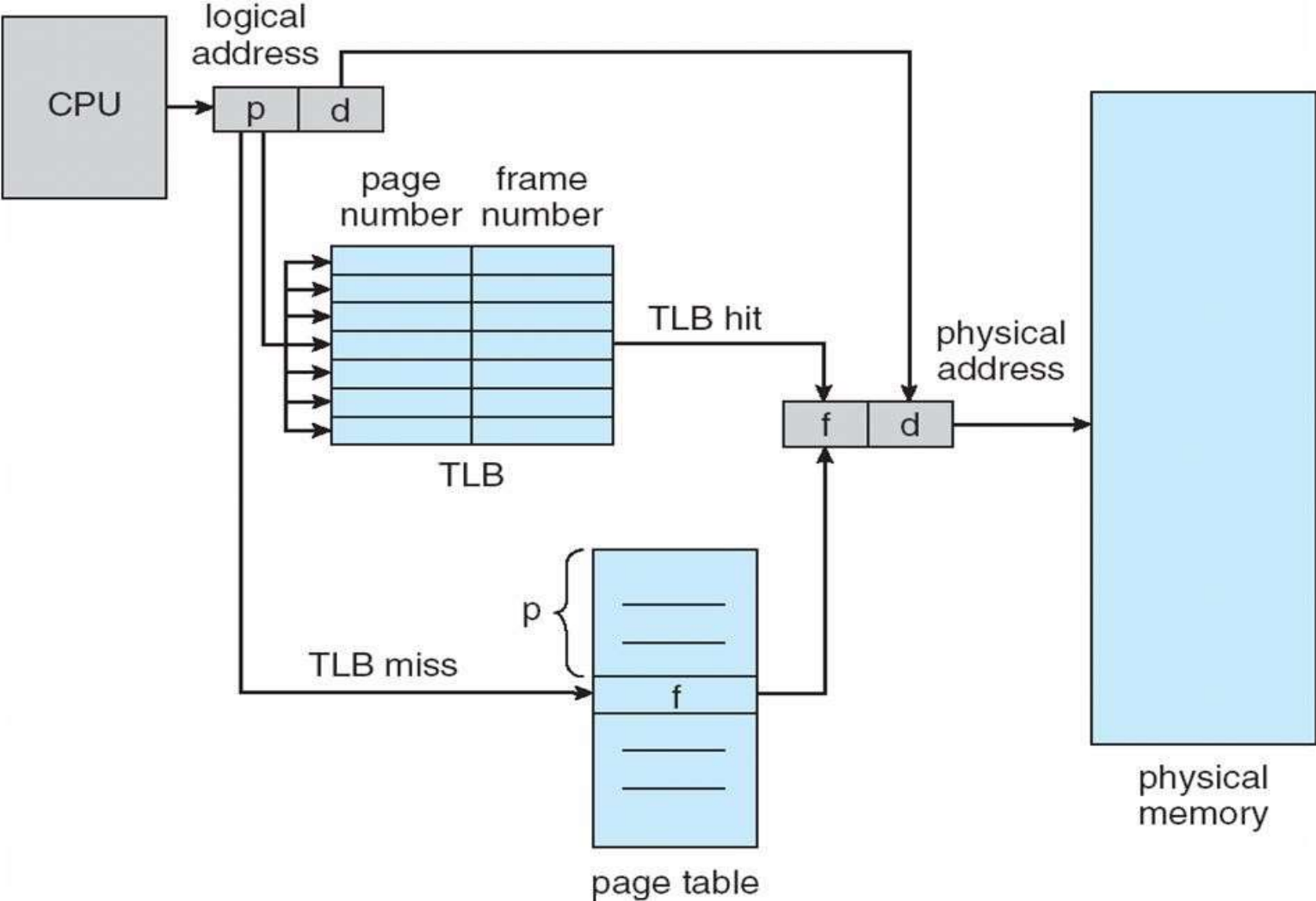


Figure 8.9 Direct Versus Associative Lookup for Page Table Entries

Translation Look-aside Buffer (TLB)



Effective Access Time using TLB

- Hit ratio
 - Percentage of times that a page number is found in the associative registers
- Example
 - hit ratio = 80%, TLB search = 20 ns, memory access = 100 ns
$$\text{EAT} = 0.8 \times (20 + 100) + 0.2 \times (20 + 100 + 100) = 140 \text{ ns}$$

40% slowdown in memory access time
 - If hit ratio increases to 98%
$$\text{EAT} = 0.98 \times (20 + 100) + 0.02 \times (20 + 100 + 100) = 122 \text{ ns}$$



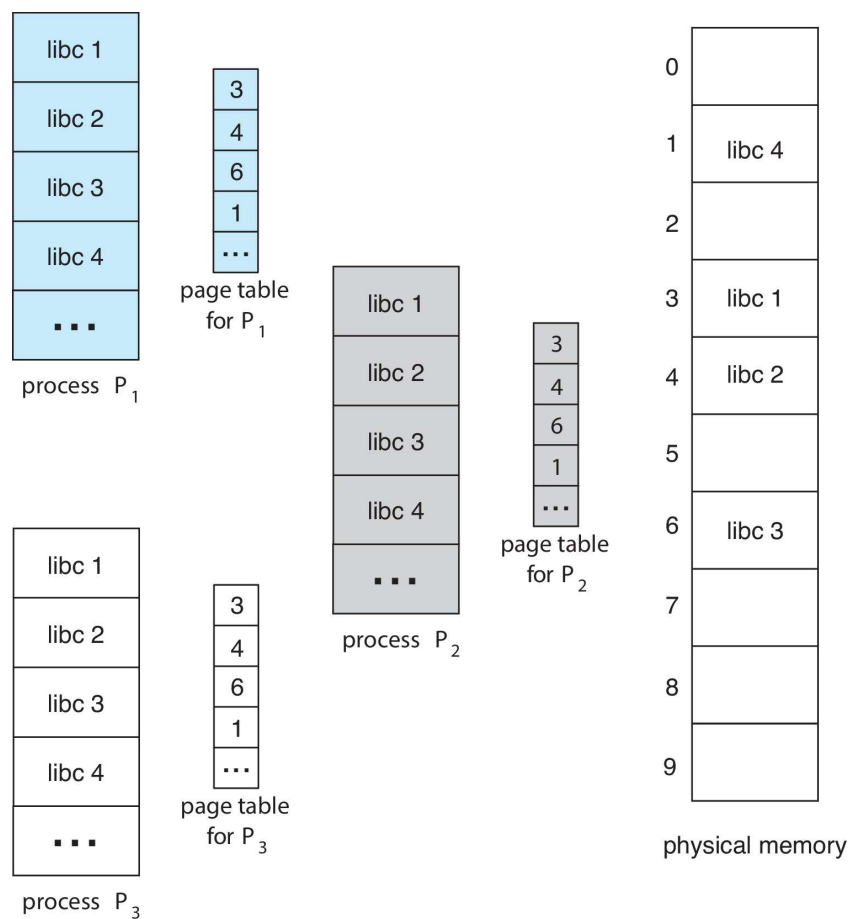
Shared Pages

- **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

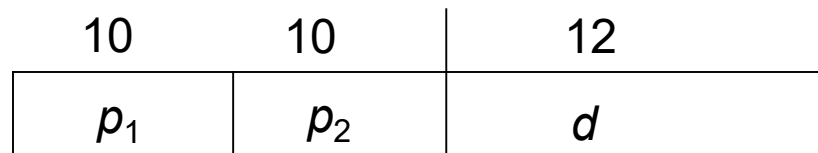


Structure of the Page Table

- Most modern computer systems support a large logical address space (2^{32} to 2^{64})
- In these environments, the page table becomes excessively large
 - Consider a 32-bit logical address space, Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12} = 2^{20}$)
 - If each entry is 4 bytes, then 4 MB of memory for page table alone
- Solutions
 - Hierarchical Paging
 - Hashed Page Tables
 - Inverted Page Tables

Hierarchical Page Tables

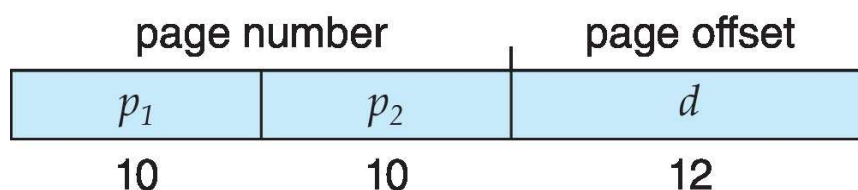
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
 - ▣ The page table itself is also paged
- Consider a 32-bit machine, with a page size of 4 KB
 - ▣ Logical address = 20 bits page number + 12 bits offset
 - ▣ Because we page the page table, the page number is further divided and a 10-bit page offset





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 10-bit page number
 - 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 within the page of the inner page table
- Known as **forward-mapped page table**

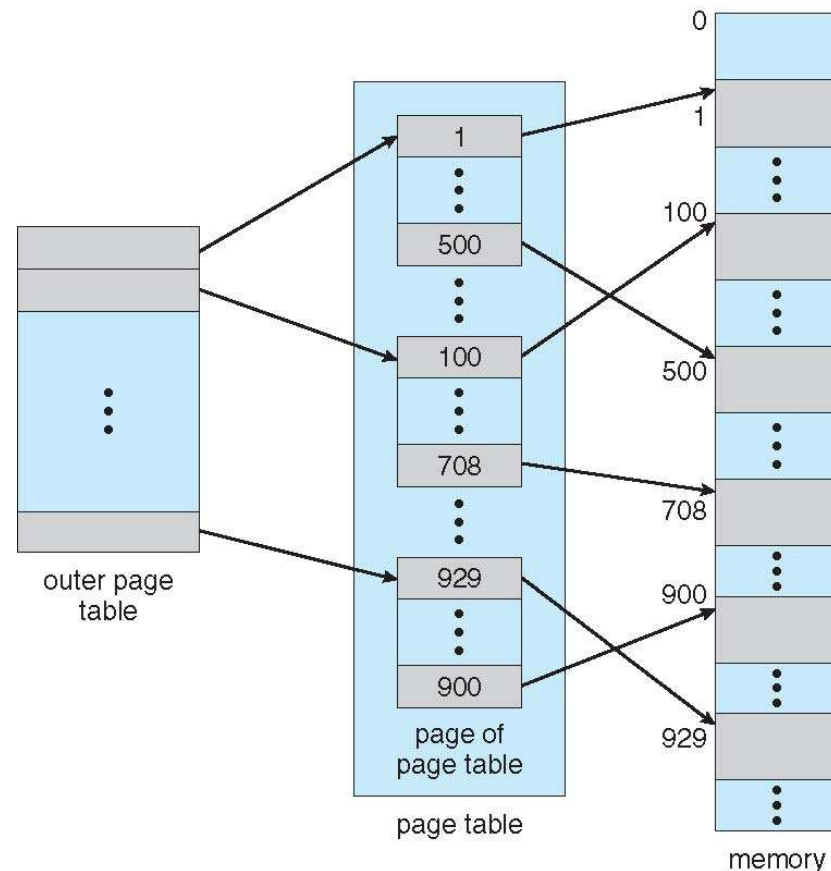




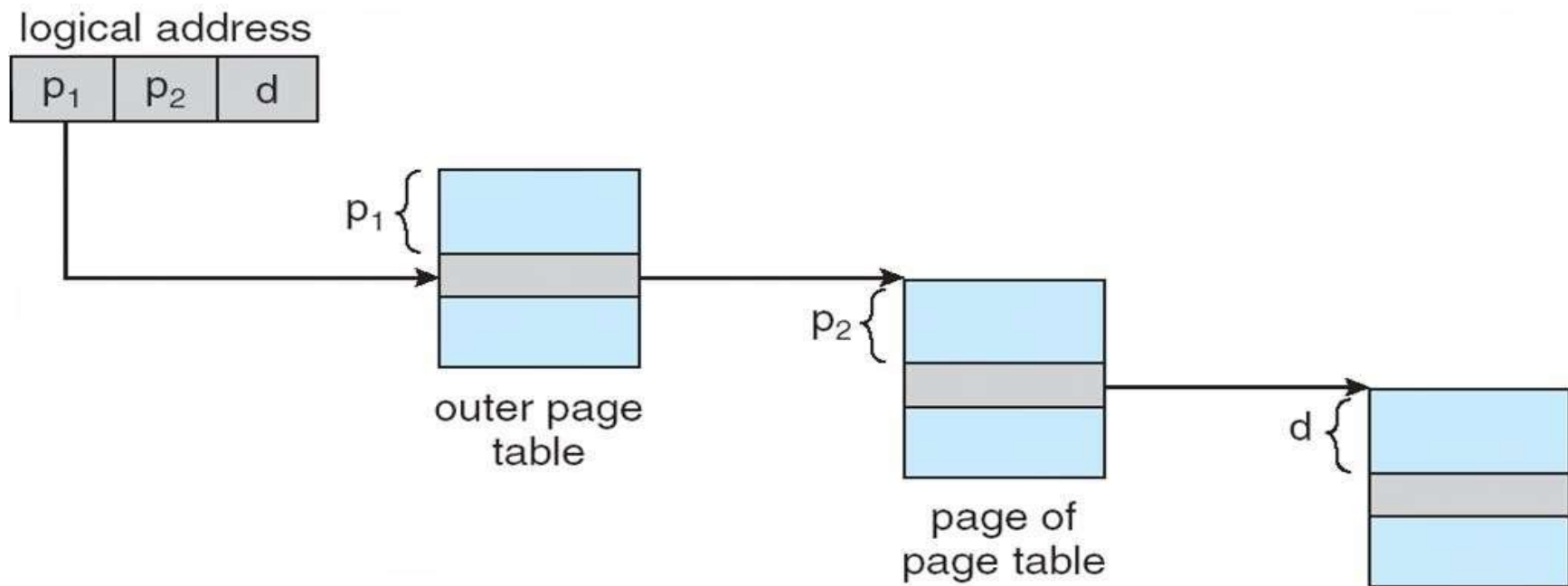
Hierarchical Page Tables

- 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

p_1	p_2	d



Hierarchical Page Tables



Important: it is not necessary to keep all the pages of page table in main memory (we will see this in the next chapter – virtual memory)

Hierarchical Page Tables

- For a 64-bit logical address space, even two-level paging scheme not sufficient
 - ▣ If page size is 4 KB, then page table has 2^{52} entries
 - ▣ With two level scheme
 - Inner page tables could be 2^{10} 4-byte entries
 - Outer page table has 2^{42} entries or 2^{44} bytes
 - ▣ Three-level (32-bit SPARC architecture) or four-level (32-bit Motorola 68030) paging schemes are required
- For 64-bit architectures, hierarchical page tables are generally considered inappropriate



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	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 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



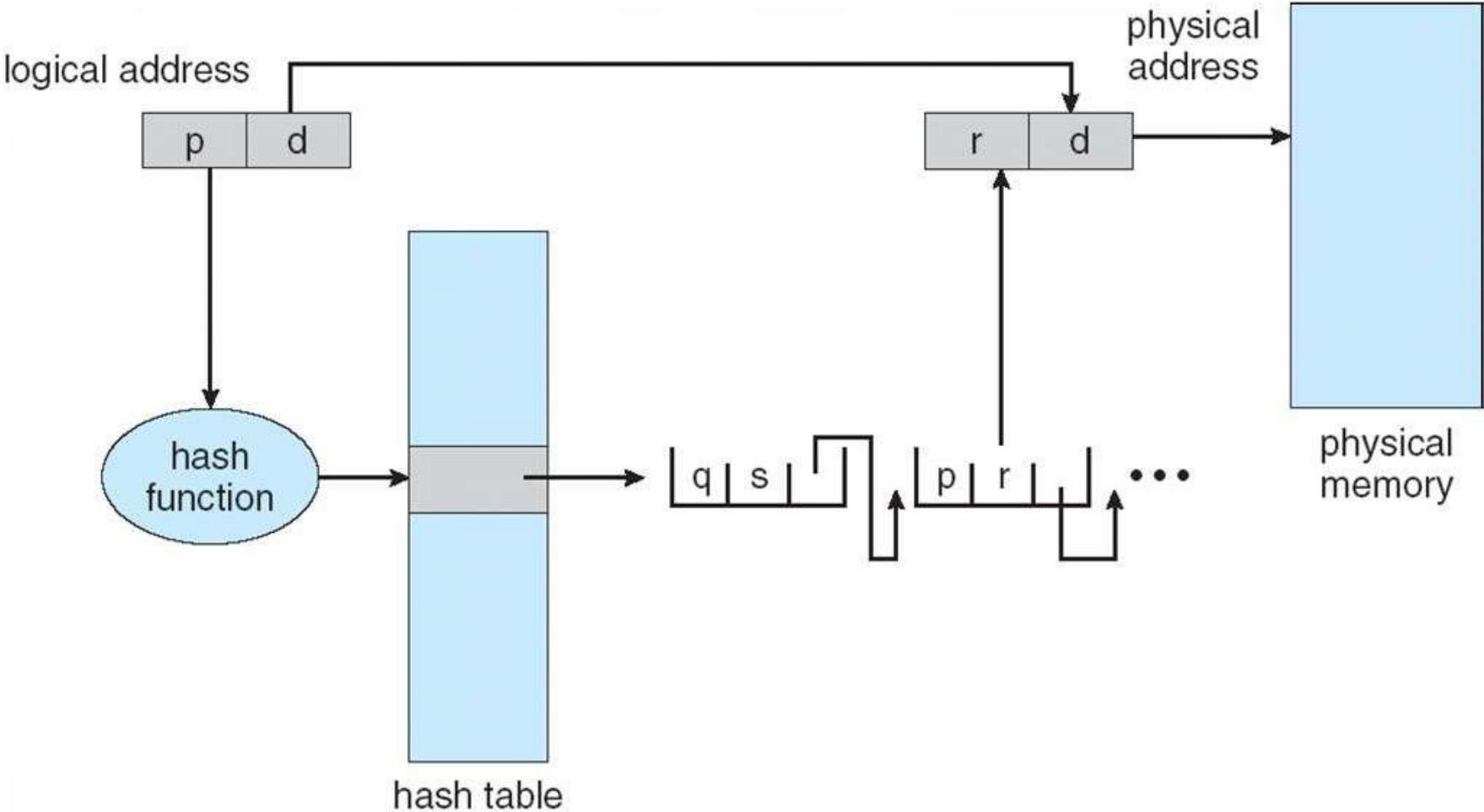


Hashed Page Tables

- 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)



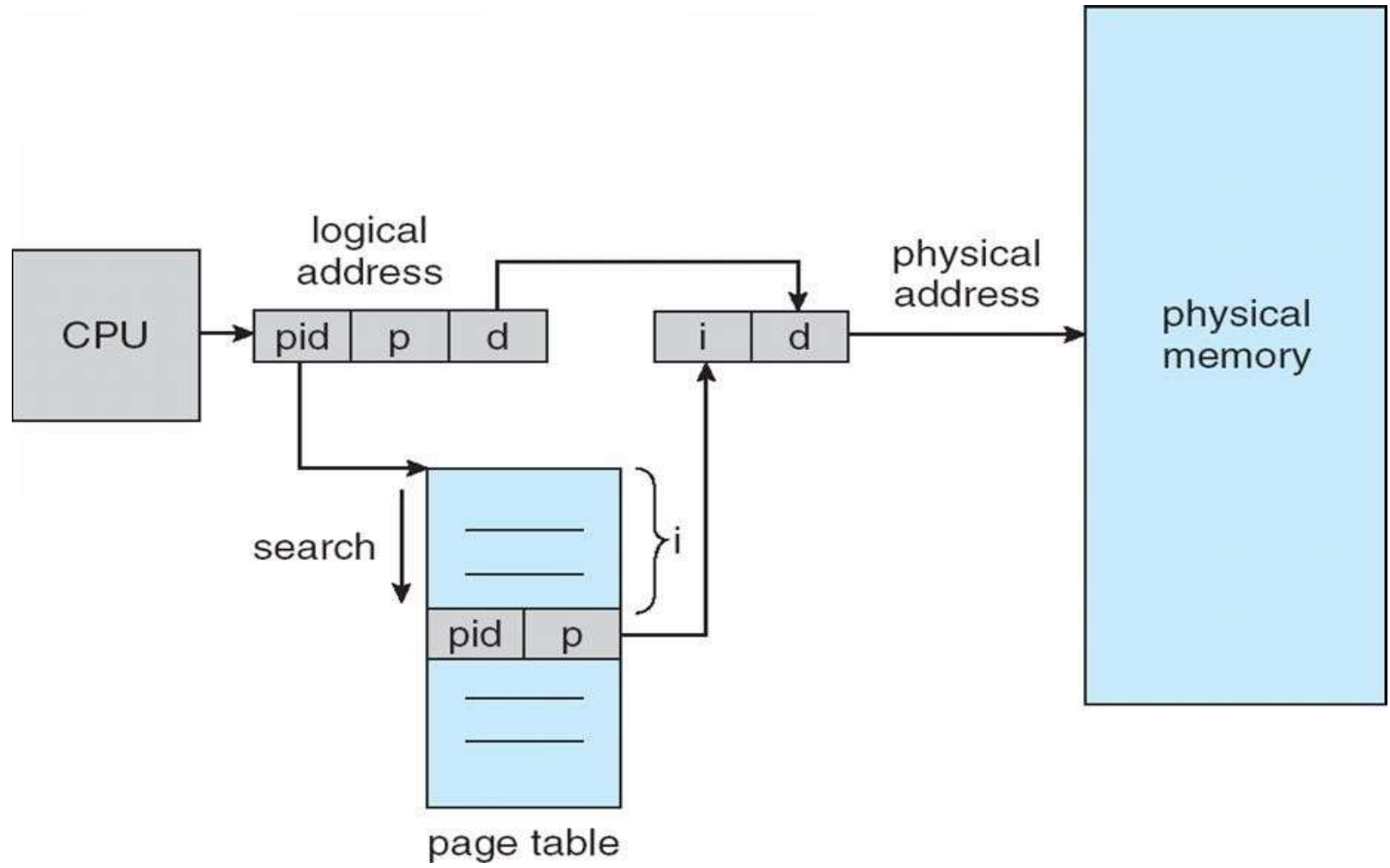
Hashed Page Tables



Inverted Page Table

- Traditional page tables
 - ▣ Each process has an associated page table
 - ▣ The page table has one entry for each logical page that the process is using
 - ▣ Drawback: each page table may consist of millions of entries, consume large amount of memory
- Inverted page tables
 - ▣ Only one page table is in the system
 - ▣ Has one entry for each real page (or frame) of memory
 - ▣ Each entry consists of <process id, page number>
 - ▣ It is called **inverted** because it indexes page table entries by frame number rather than by virtual page number.

Inverted Page Table



Inverted Page Table

- Problem
 - Search is time consuming
- Solution
 - Combine with hash table

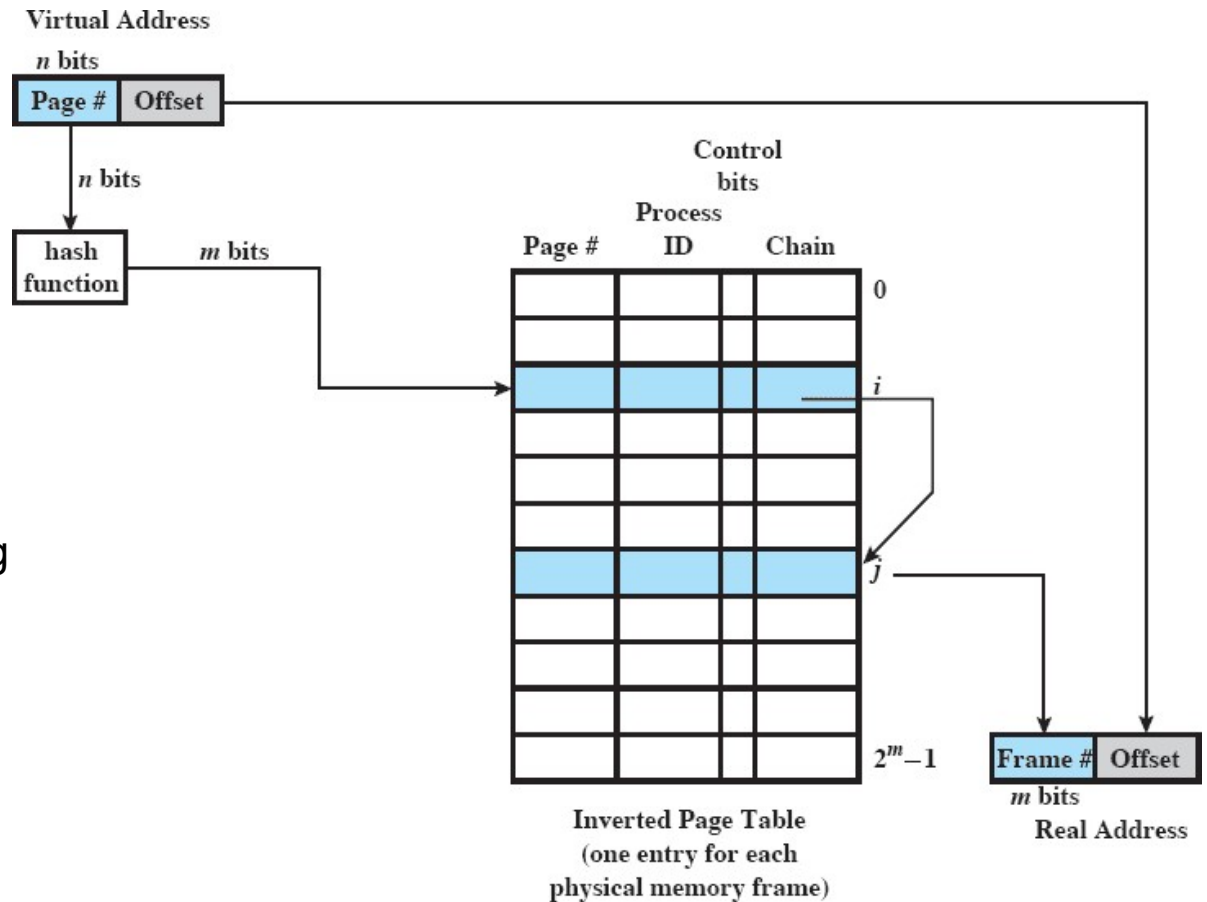
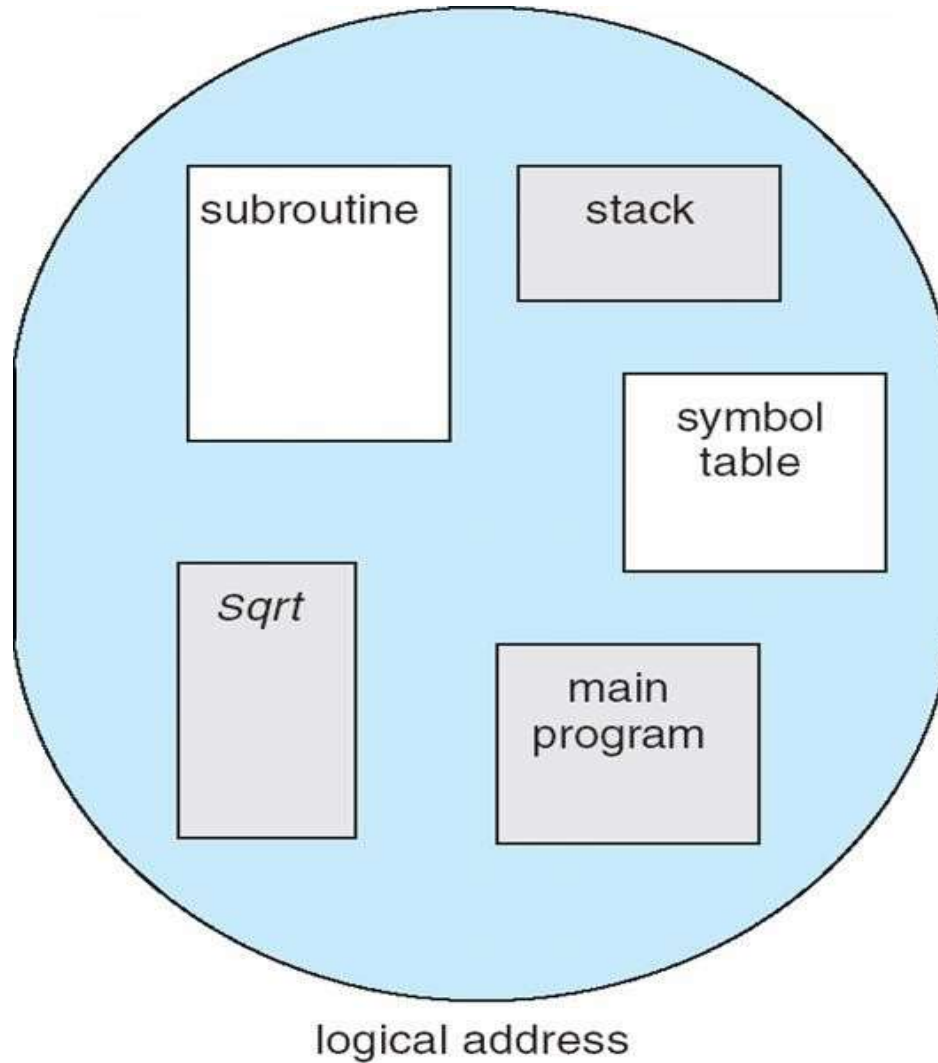


Figure 8.6 Inverted Page Table Structure

Segmented Memory Management

- An important property of paging is **separation** of the user's view of memory and the actual physical memory
- Segmentation supports user view of memory
 - ▣ Users prefer to view memory as a collection of variable-sized segments
 - ▣ A segment is a logical unit such as:
 - main program, procedure, function, method
 - Data structures like objects, arrays stacks, variables,
...

Segmented Memory Management



Segmented Memory Management

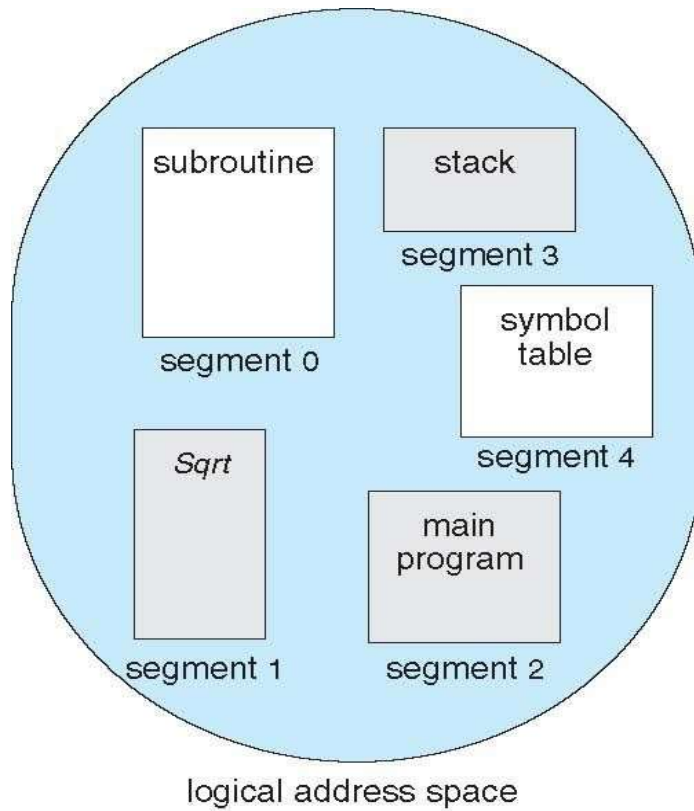


- A logical address space is a collection of segments
- Each segment has a name and a length
- The user specifies each address by two quantities: a segment name and an offset
 - ▣ For simplicity, segments are numbered
 - ▣ logical address = <segment number, offset>
- For example, a C compiler might create separate segments for the following:
 - ▣ The code, global variables, the heap, the stacks used by each thread, and the standard C library

Address Translation Scheme

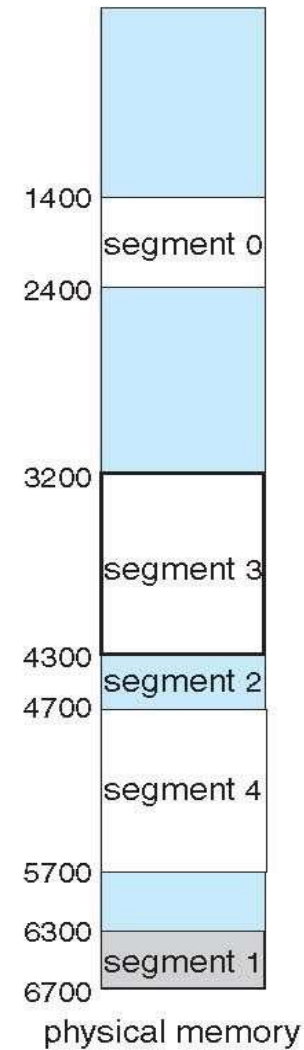
- Each process has a **segment table**
- Each entry in segment table has
 - Segment **base**
 - The starting physical address where the segment resides in memory
 - Segment **limit**
 - Specifies the length of the segment
- To translate a logical address $\langle s, d \rangle$
 - **s** is used as an index to the segment table
 - if **d** is legal, it is added to the segment base to produce physical address

Address Translation Scheme

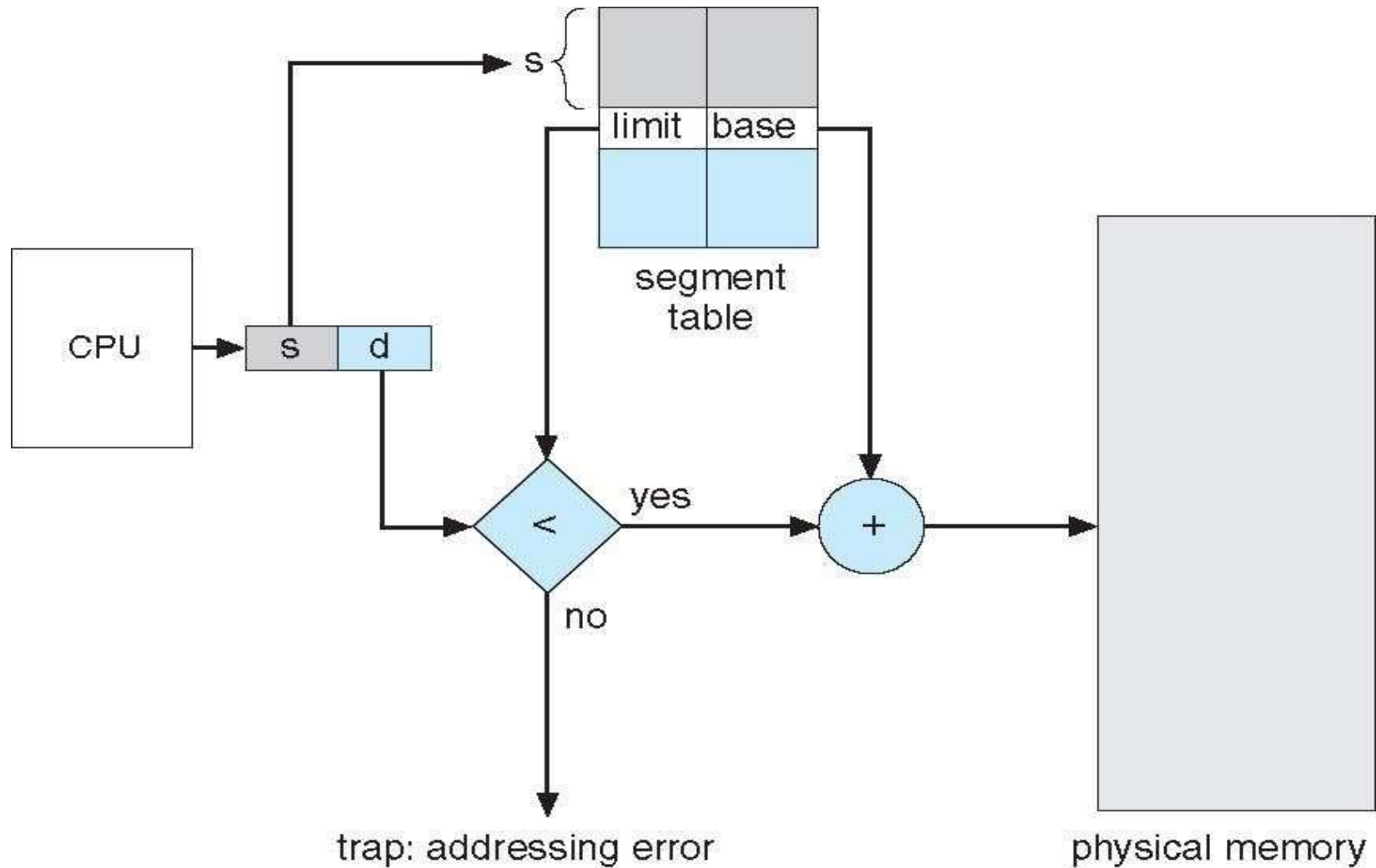


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



Address Translation Scheme



Combined Paging and Segmentation

- Both paging and segmentation have their strengths
 - Paging is transparent to the programmer, while Segmentation is visible to the programmer
- In a combined paging/segmentation system
 - A user's address space is broken up into a number of segments, at the discretion of the programmer.
 - Each segment is, in turn, broken up into a number of fixed-size pages
- Example: The Intel Pentium

Combined Paging and Segmentation

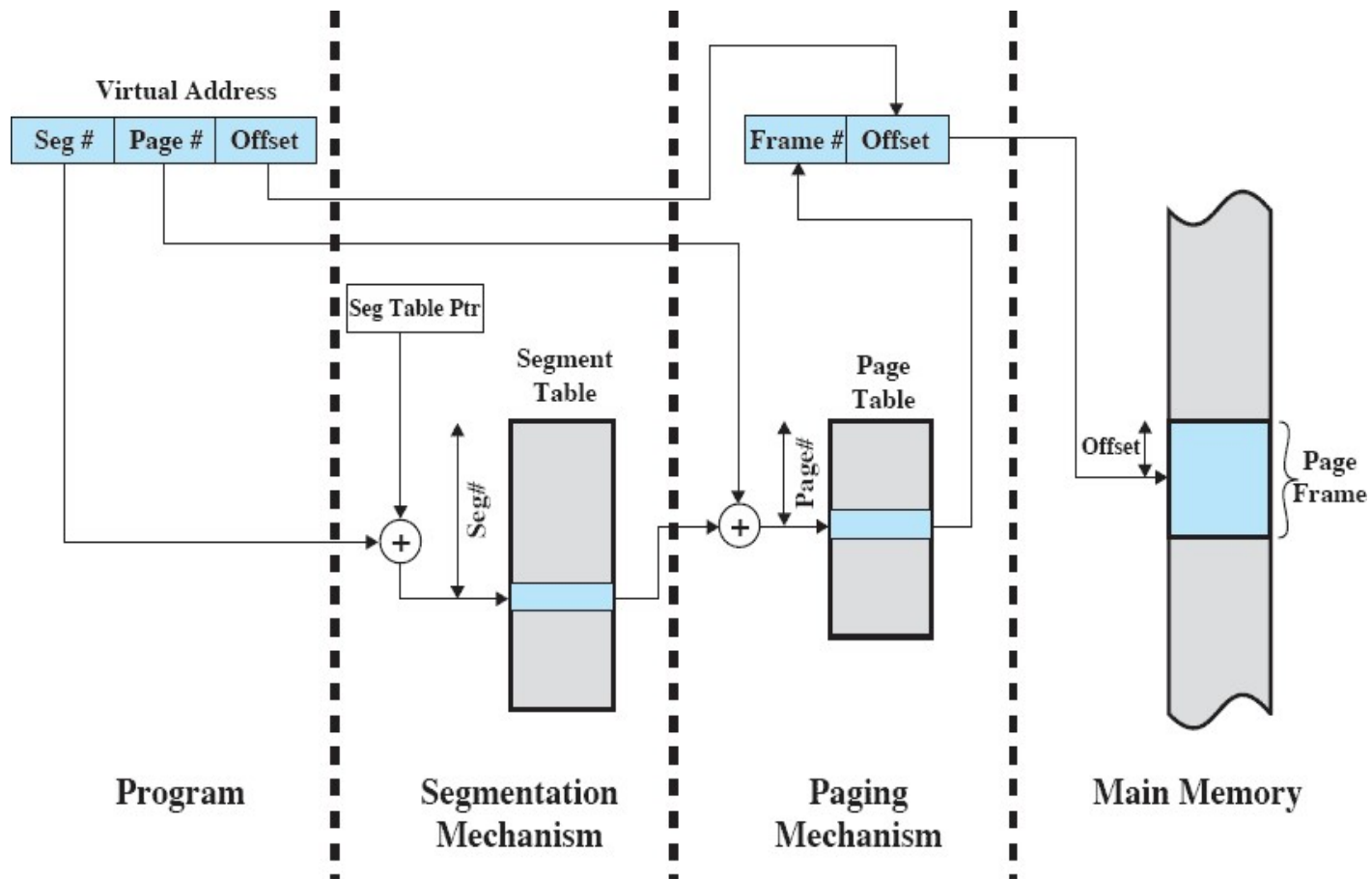


Figure 8.13 Address Translation in a Segmentation/Paging System

End of Chapter 9

