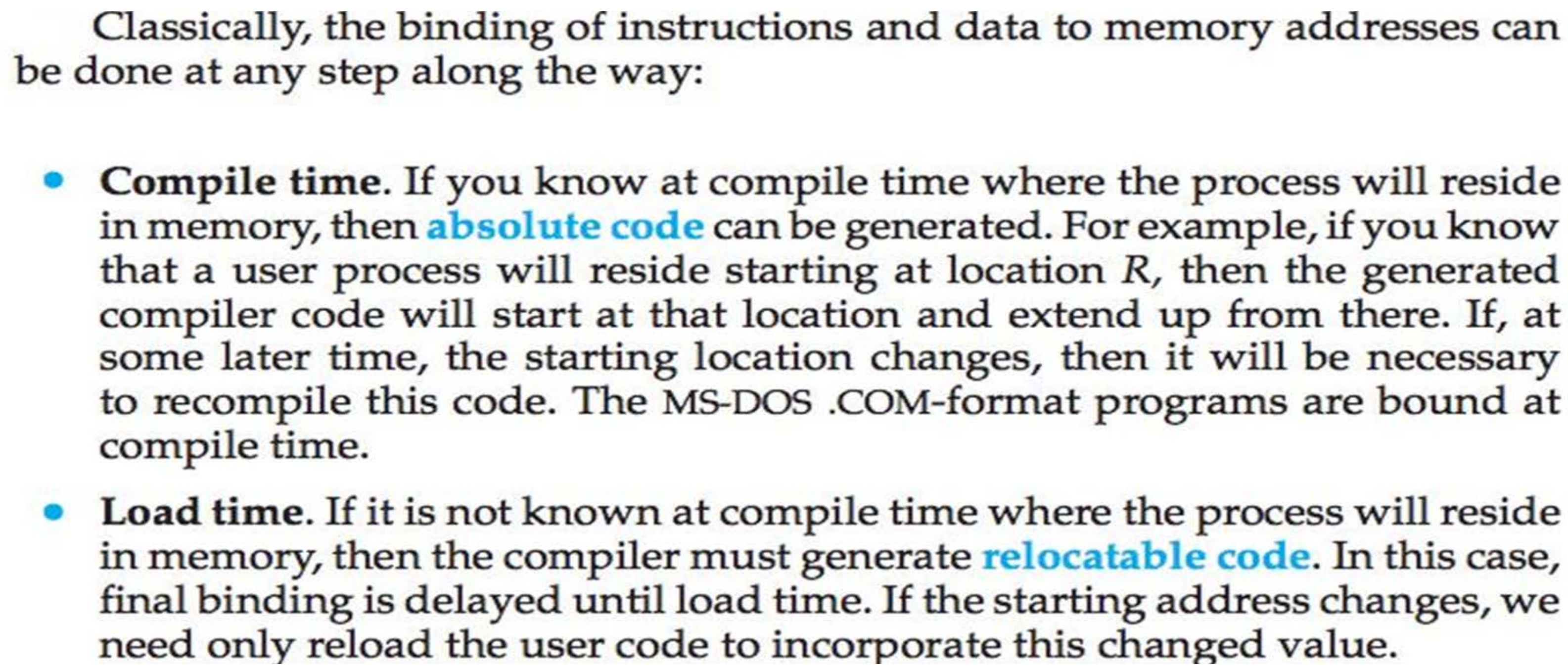
Tutorial 8

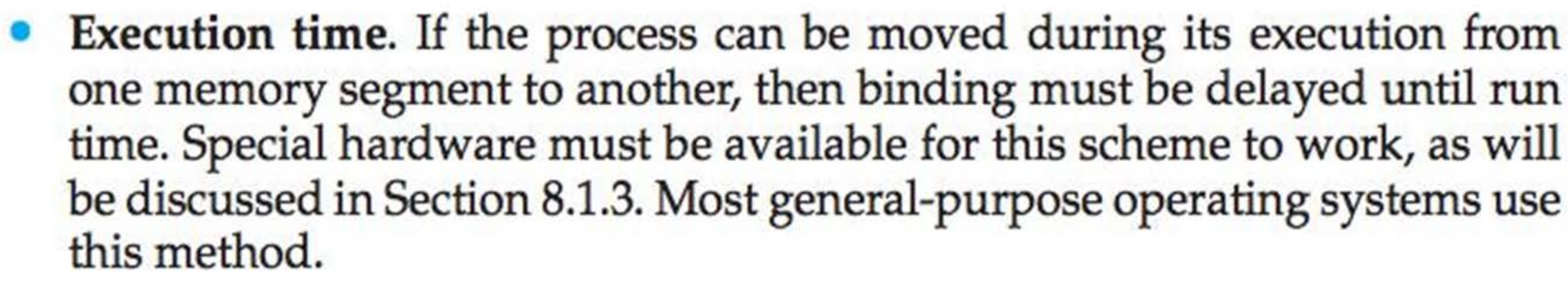
Exercises CH8

8.1 Name two differences between logical and physical addresses.



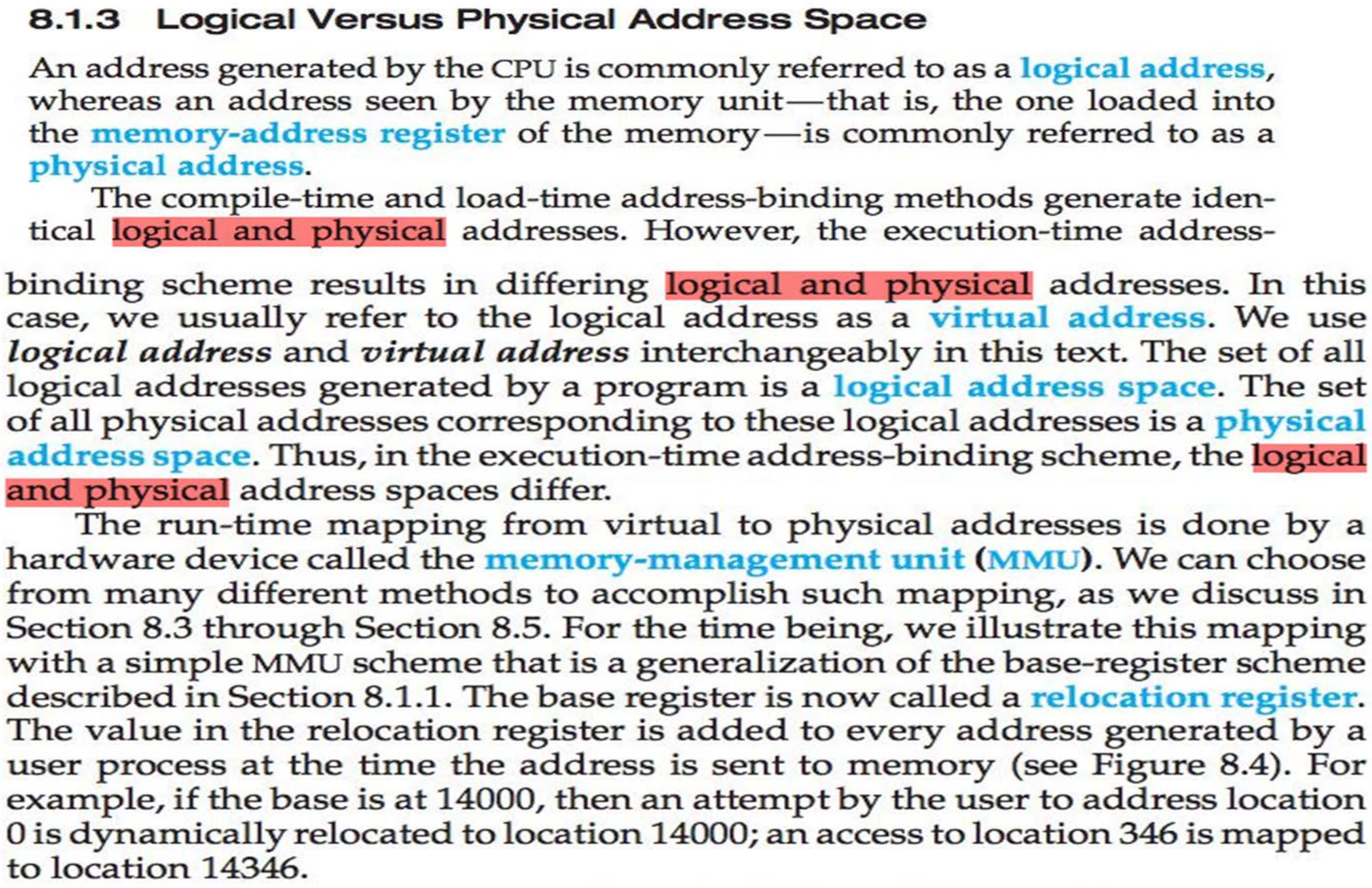
Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- **Compile time.** If you know at compile time where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.
- **Load time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.



- **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 8.1.3. Most general-purpose operating systems use this method.

8.1 Name two differences between logical and physical addresses.



8.1.3 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.

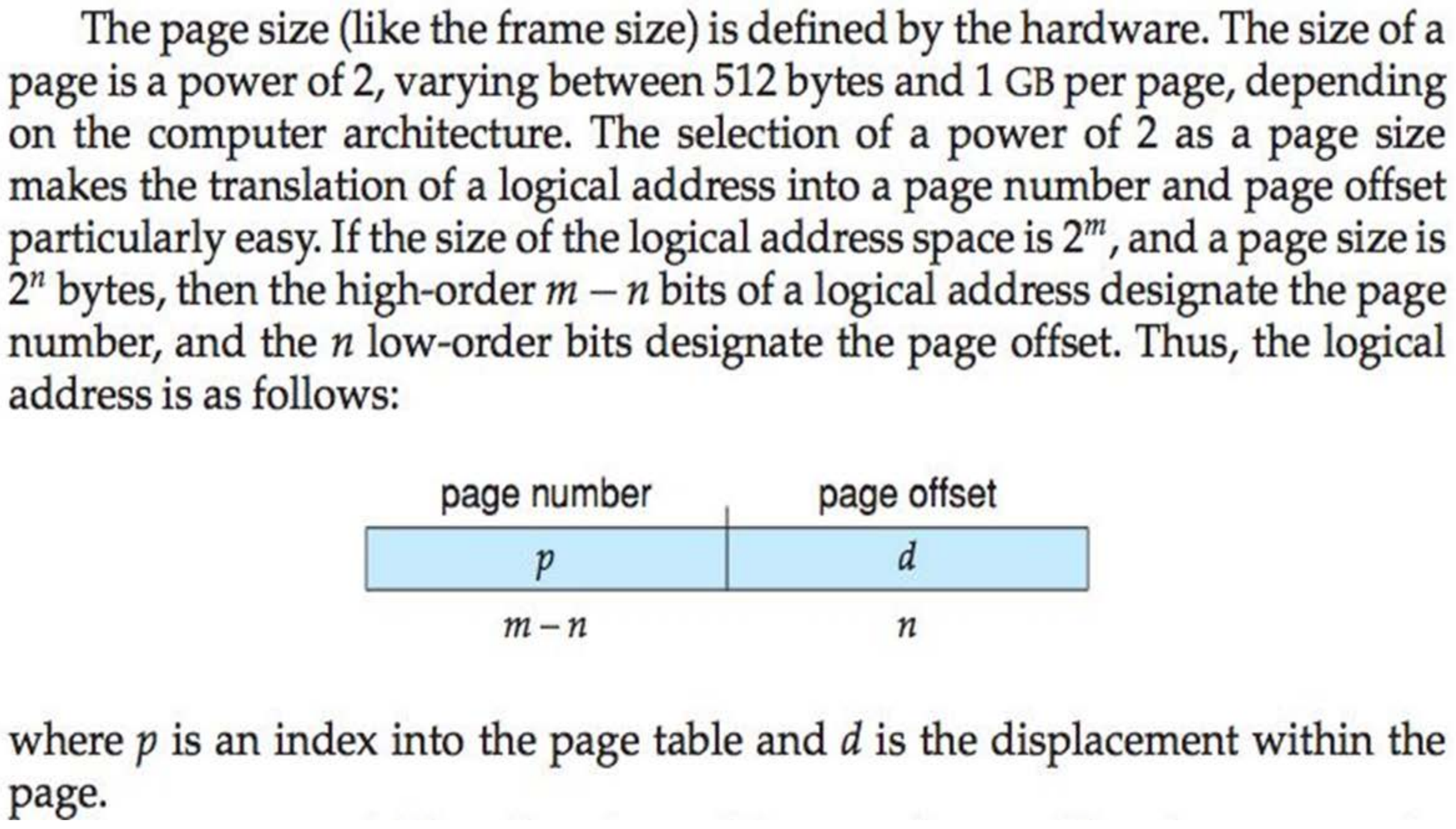
The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (**MMU**). We can choose from many different methods to accomplish such mapping, as we discuss in Section 8.3 through Section 8.5. For the time being, we illustrate this mapping with a simple MMU scheme that is a generalization of the base-register scheme described in Section 8.1.1. The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 8.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

8.1 Name two differences between logical and physical addresses.

A logical address does not refer to an actual existing address; rather, it refers to an abstract address in an abstract address space. Contrast this with a physical address that refers to an actual physical address in memory. A logical address is generated by the CPU and is translated into a physical address by the memory management unit(MMU). Therefore, physical addresses are generated by the MMU.

8.3 Why are page sizes always powers of 2?



The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

where p is an index into the page table and d is the displacement within the page.

8.3 Why are page sizes always powers of 2?

Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

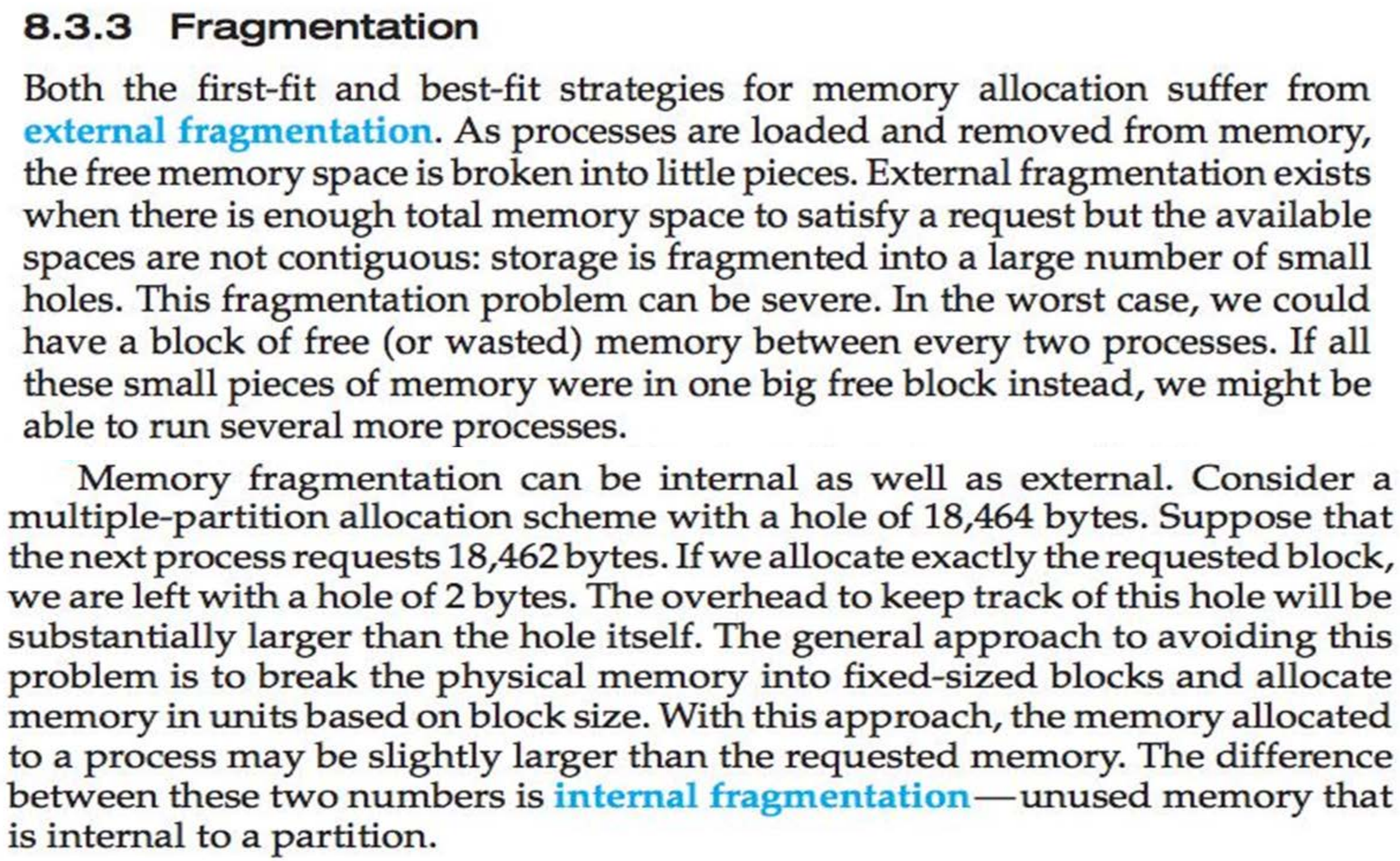
8.4 Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.

a. How many bits are there in the logical address?

b. How many bits are there in the physical address?

Addressing within a 1024-word page requires 10 bits because $1024 = 2^{10}$. Since the logical address space consists of $64 = 2^6$ pages, the logical addresses must be $10+6 = 16$ bits. Similarly, since there are $32 = 2^5$ physical frames, physical addresses are $5 + 10 = 15$ bits long.

8.9 Explain the difference between internal and external fragmentation.



8.3.3 Fragmentation

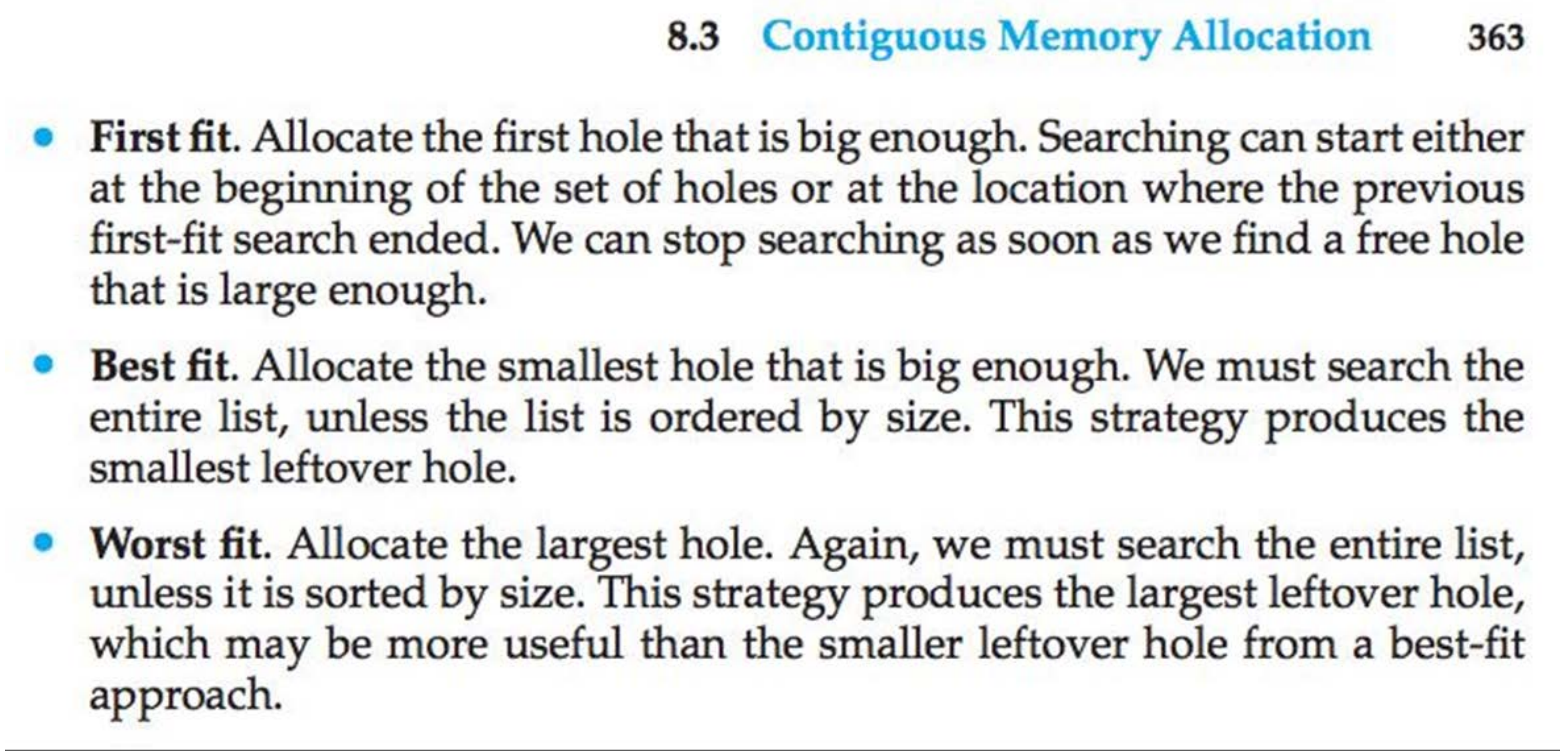
Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.

8.9 Explain the difference between internal and external fragmentation.

Internal fragmentation is unused allocated area, whereas external fragmentation is un allocated area and unused

8.11 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?



8.3 Contiguous Memory Allocation 363

- **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

8.11 Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

- M1=300 KB M2=600 KB M3=350 KB M4=200 KB M5=750 KB M6=125 KB
- P1=115 KB P2=500 KB P3=358 KB P4=200 KB P5=375 KB

First-fit:

- P1 = 115 M1=**300** M2=600 M3=350 M4=200 M5=750 M6=125
- P2 = 500 M1=185 M2=**600** M3=350 M4=200 M5=750 M6=125
- P3 = 358 M1=185 M2=100 M3=350 M4=200 M5=**750** M6=125
- P4 = 200 M1=185 M2=100 M3=**350** M4=200 M5=392 M6=125
- P5 = 375 M1=185 M2=100 M3=150 M4=200 M5=**392** M6=125

Best-fit

- P1 = 115 M1=300 M2=600 M3=350 M4=200 M5=750 M6=**125**
- P2 = 500 M1=300 M2=**600** M3=350 M4=200 M5=750 M6=10
- P3 = 358 M1=300 M2=100 M3=350 M4=200 M5=**750** M6=10
- P4 = 200 M1=300 M2=100 M3=350 M4=**200** M5=392 M6=10
- P5 = 375 M1=300 M2=100 M3=350 M4=000 M5=**392** M6=10

8.20 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

Answer:

Page size $=2^n = 1024$ B$= 2^{10}$ B

# of bits in offset part (n) =10

Solution steps :

1. Convert logical address: Decimal to Binary
2. Split binary address to 2 parts (page # , Offset), offset : n digits
3. Convert offset & page# : Binary to Decimal

8.20 Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

Logical address (decimal)	Logical address (binary)	Page # (22 bits) (binary)	Offset (10 bits) (binary)	Page # decimal	Offset decimal
3085	00000000000000000000110000001101	0000000000000000000011	**0000001101**	3	13
42095	00000000000000001010010001101111	0000000000000000101001	**0001101111**	41	111
215201	00000000000000110100100010100001	0000000000000011010010	**0010100001**	210	161
650000	00000000000010011110101100010000	0000000000001001111010	**1100010000**	634	784
2000001	00000000000010000000000000000001	0000000000001000000000	**0000000001**	512	1

8.28 Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430

b. 1,10

c. 2,500

d. 3,400

e. 4,112

0,430	219+430=649
1,10	2300+10=2310
2,500	Invalid
3,400	1327+400=1727
4,112	Invalid