

# Memory Management

---

# Memory Management

- **Is the task carried out by the OS and hardware to accommodate multiple processes in main memory**
- **If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle**
- **Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible**

# Memory Management

- **In most schemes, the kernel occupies some fixed portion of main memory and the rest is shared by multiple processes**

# Memory Management Requirements

## ■ Relocation

- ◆ programmer cannot know where the program will be placed in memory when it is executed
- ◆ a process may be (often) **relocated** in main memory due to swapping
- ◆ swapping enables the OS to have a larger pool of ready-to-execute processes
- ◆ memory references in code (for both instructions and data) must be translated to actual physical memory address

# Memory Management Requirements

## ■ Protection

- ◆ processes should not be able to reference memory locations in another process without permission
- ◆ impossible to check addresses at compile time in programs since the program could be relocated
- ◆ address references must be checked at run time by hardware

# Memory Management Requirements

## ■ Sharing

- ◆ must allow several processes to access a common portion of main memory without compromising protection
  - ☞ cooperating processes may need to share access to the same data structure
  - ☞ better to allow each process to access the same copy of the program rather than have their own separate copy

# Memory Management Requirements

## ■ Logical Organization

- ◆ users write programs in modules with different characteristics
  - ☞ instruction modules are execute-only
  - ☞ data modules are either read-only or read/write
  - ☞ some modules are private others are public
- ◆ To effectively deal with user programs, the OS and hardware should support a basic form of module to provide the required protection and sharing

# Memory Management Requirements

## ■ Physical Organization

- ◆ secondary memory is the long term store for programs and data while main memory holds program and data currently in use
- ◆ moving information between these two levels of memory is a major concern of memory management (OS)
  - ☞ it is highly inefficient to leave this responsibility to the application programmer

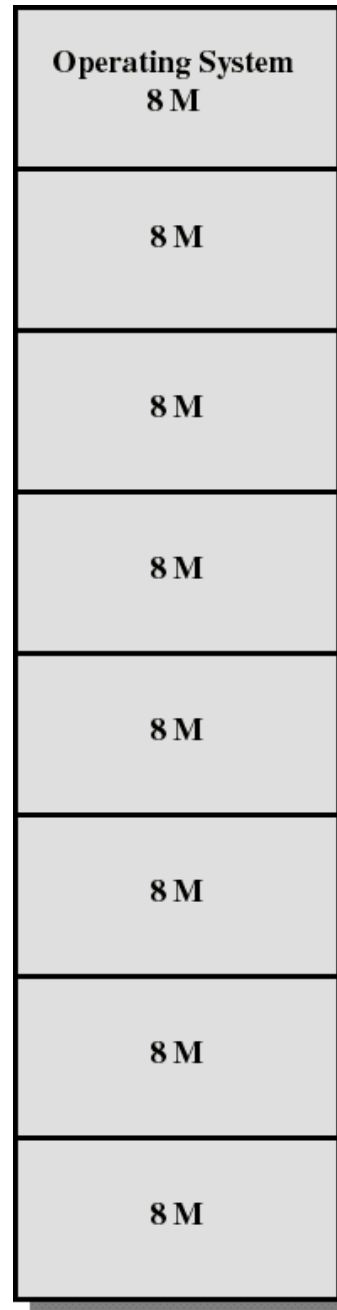


# Simple Memory Management

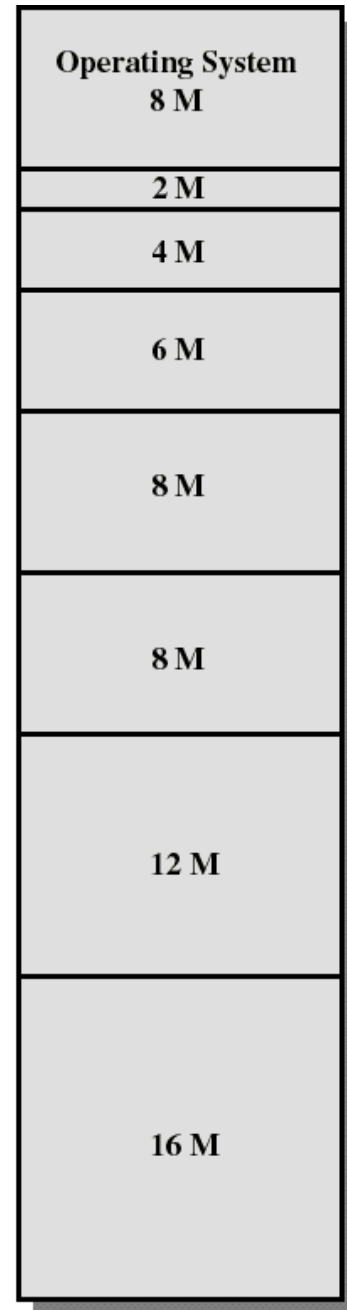
- In this chapter we study the simpler case where there is **no virtual memory**
- An executing process must be loaded entirely in main memory (if overlays are not used)
- Although the following simple memory management techniques are not used in modern OS, they lay the ground for a proper discussion of virtual memory (next chapter)
  - ◆ fixed partitioning
  - ◆ dynamic partitioning
  - ◆ simple paging
  - ◆ simple segmentation

# Fixed Partitioning

- Partition main memory into a set of non overlapping regions called **partitions**
- Partitions can be of equal or unequal sizes



Equal-size partitions



Unequal-size partitions

# Fixed Partitioning

- any process whose size is less than or equal to a partition size can be loaded into the partition
- if all partitions are occupied, the operating system can swap a process out of a partition
- a program may be too large to fit in a partition. The programmer must then design the program with overlays
  - ◆ when the module needed is not present the user program must load that module into the program's partition, overlaying whatever program or data are there

# Fixed Partitioning

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called **internal fragmentation**.
- Unequal-size partitions lessens these problems but they still remain...
- Equal-size partitions was used in early IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks)

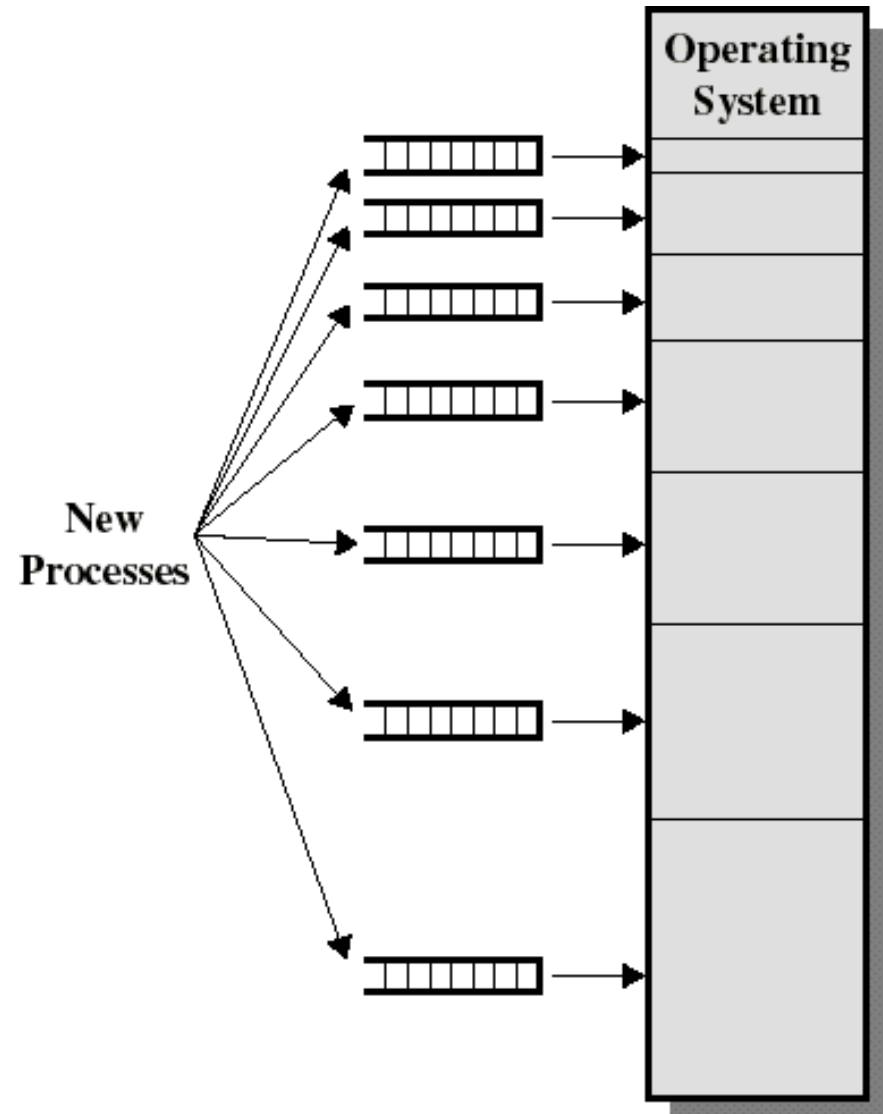
# Placement Algorithm with Partitions

- **Equal-size partitions**

- ◆ If there is an available partition, a process can be loaded into that partition
  - ☞ because all partitions are of equal size, it does not matter which partition is used
- ◆ If all partitions are occupied by blocked processes, choose one process to swap out to make room for the new process

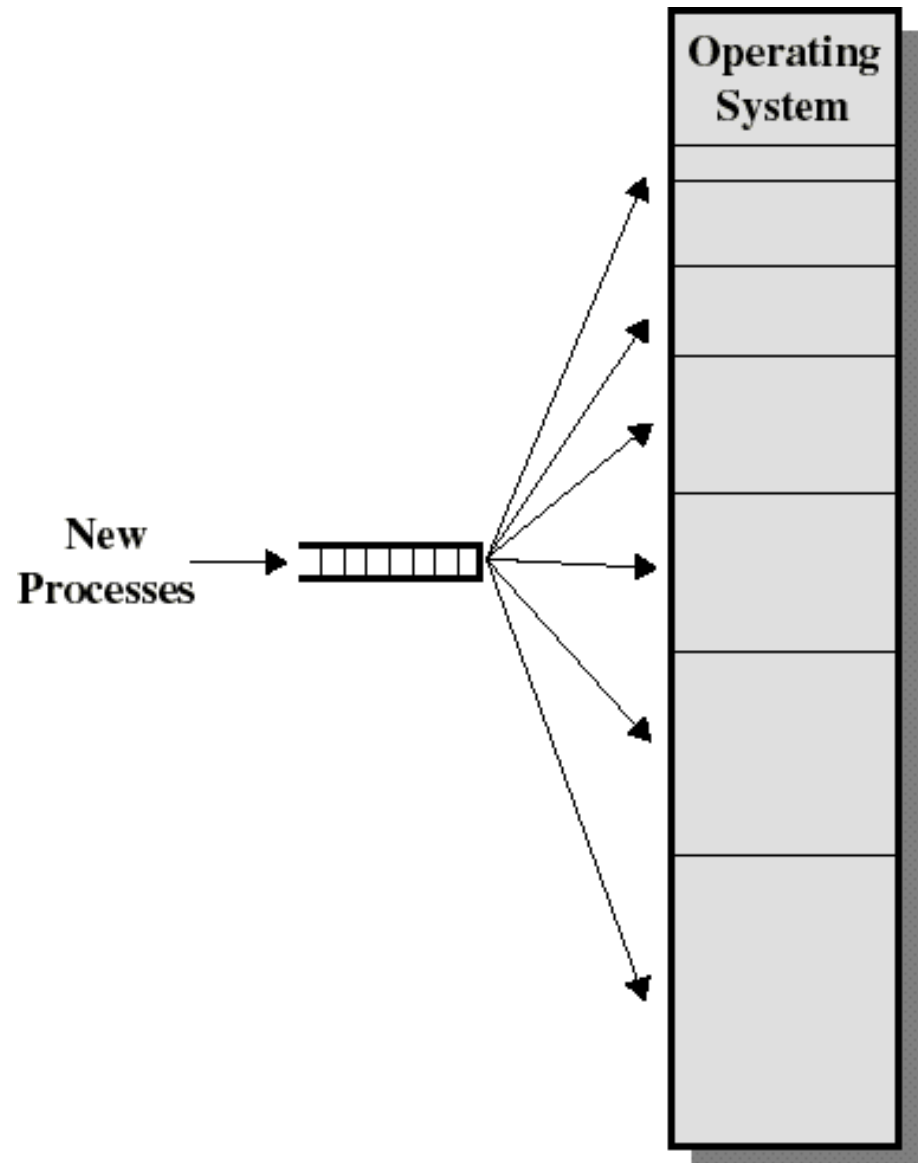
# Placement Algorithm with Partitions

- **Unequal-size partitions: use of multiple queues**
  - ◆ assign each process to the smallest partition within which it will fit
  - ◆ A queue for each partition size
  - ◆ tries to minimize internal fragmentation
  - ◆ Problem: some queues will be empty if no processes within a size range is present



# Placement Algorithm with Partitions

- **Unequal-size partitions: use of a single queue**
  - ◆ When its time to load a process into main memory the smallest available partition that will hold the process is selected
  - ◆ increases the level of multiprogramming at the expense of internal fragmentation

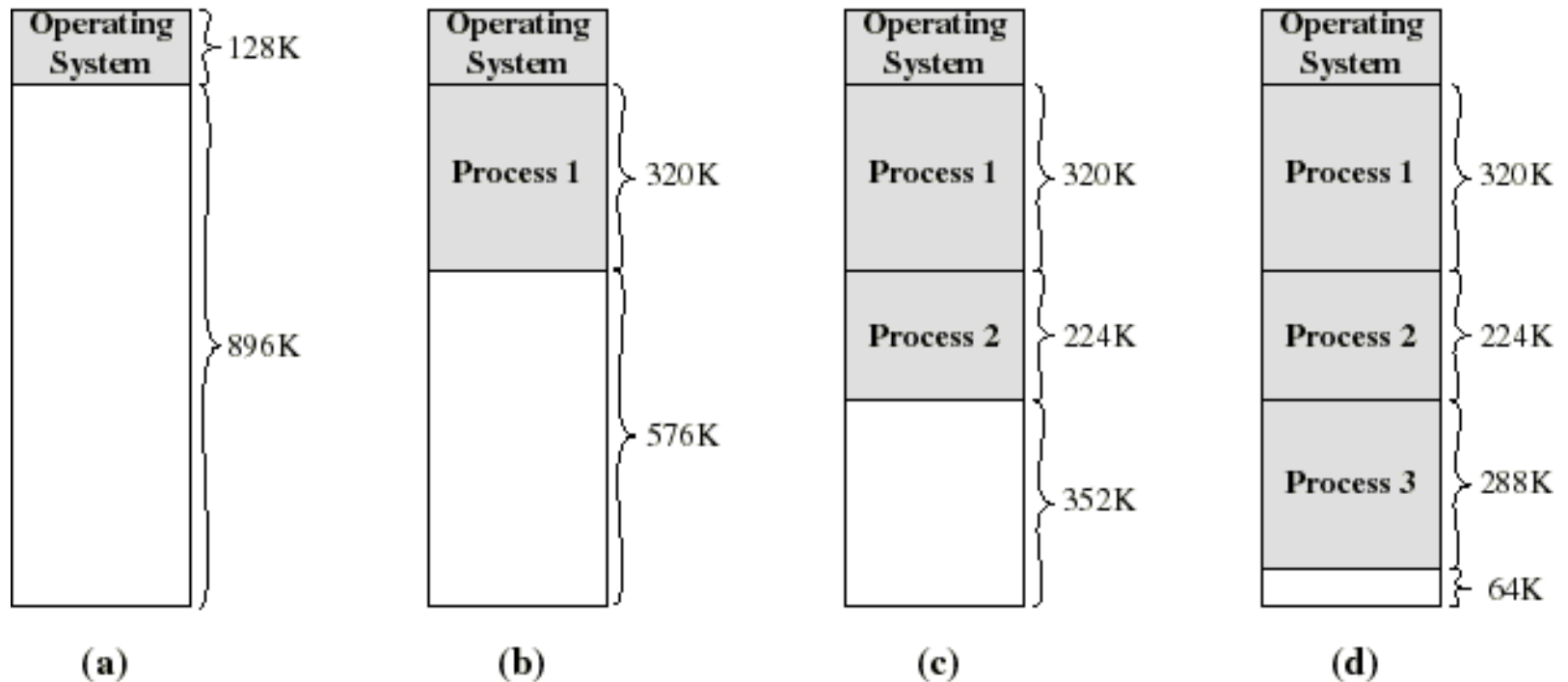


# Dynamic Partitioning

- Partitions are of variable length and number
- Each process is allocated exactly as much memory as it requires
- Eventually holes are formed in main memory. This is called **external fragmentation**
- Must use **compaction** to shift processes so they are contiguous and all free memory is in one block
- Used in IBM's OS/MVT (Multiprogramming with a Variable number of Tasks)

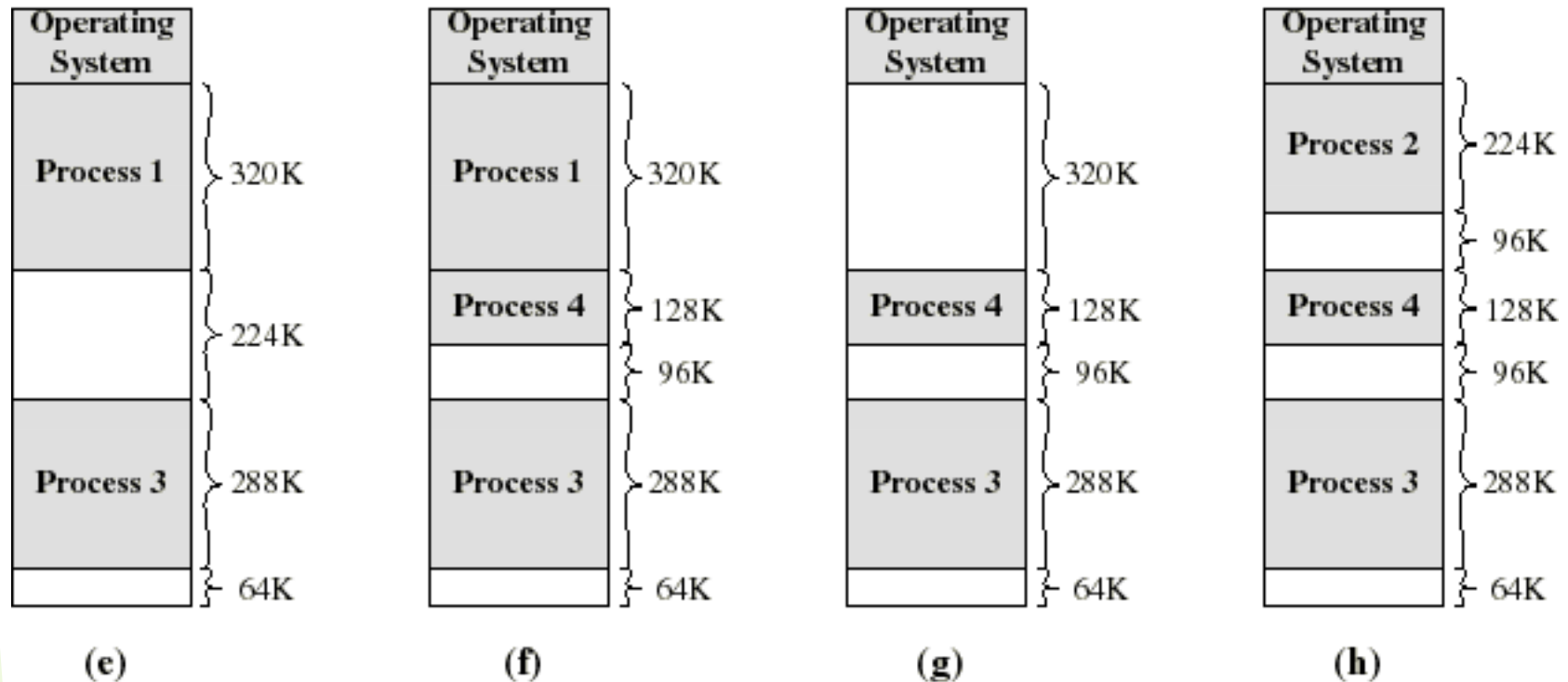


# Dynamic Partitioning: an example



- A hole of 64K is left after loading 3 processes: not enough room for another process
- Eventually each process is blocked. The OS swaps out process 2 to bring in process 4

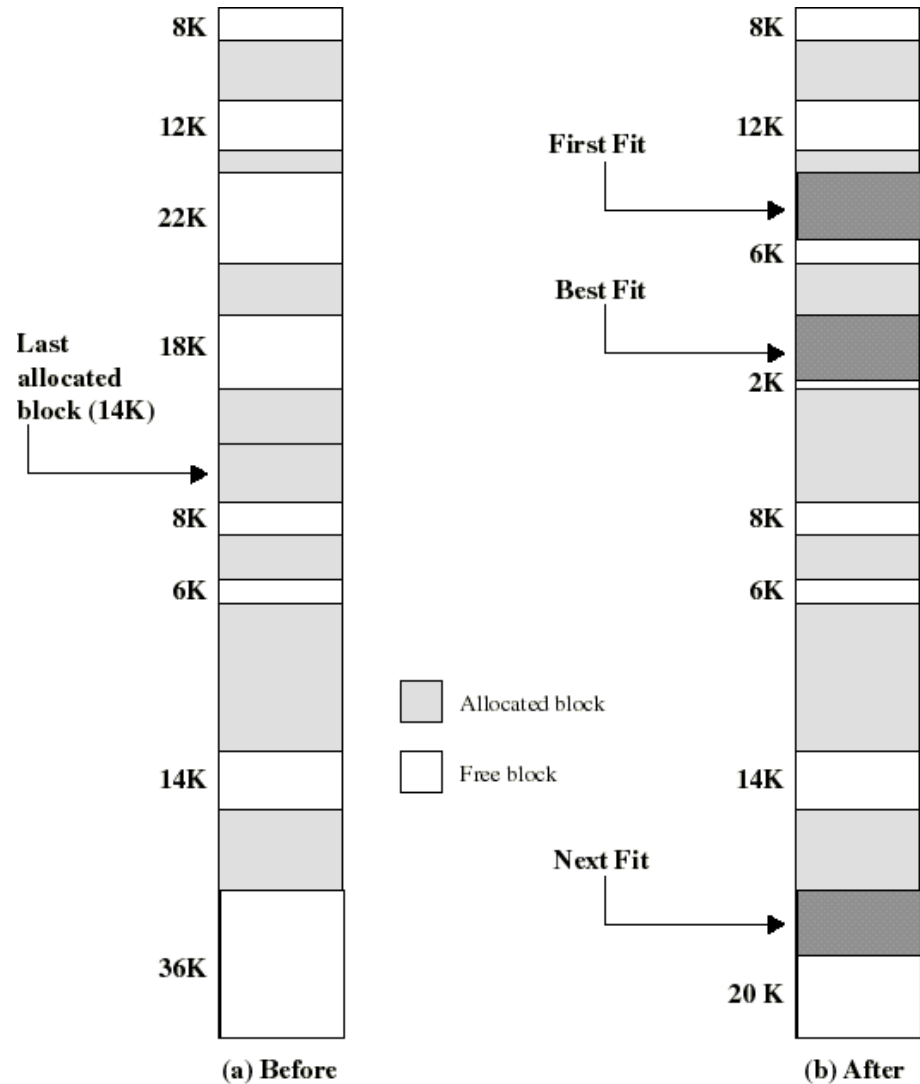
# Dynamic Partitioning: an example



- another hole of 96K is created
- Eventually each process is blocked. The OS swaps out process 1 to bring in again process 2 and another hole of 96K is created...
- Compaction would produce a single hole of 256K

# Placement Algorithm

- Used to decide which free block to allocate to a process
- Goal: to reduce usage of compaction (time consuming)
- Possible algorithms:
  - ◆ **Best-fit**: choose smallest hole
  - ◆ **First-fit**: choose first hole from beginning
  - ◆ **Next-fit**: choose first hole from last placement



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

# Placement Algorithm: comments

- **Next-fit often leads to allocation of the largest block at the end of memory**
- **First-fit favors allocation near the beginning: tends to create less fragmentation than Next-fit**
- **Best-fit searches for smallest block: the fragment left behind is small as possible**
  - ◆ main memory quickly forms holes too small to hold any process: compaction generally needs to be done more often

# Replacement Algorithm

- **When all processes in main memory are blocked, the OS must choose which process to replace**
  - ◆ A process must be swapped out (to a Blocked-Suspend state) and be replaced by a new process or a process from the Ready-Suspend queue
  - ◆ We will discuss later such algorithms for memory management schemes using virtual memory

# Buddy System

- A reasonable compromise to overcome disadvantages of both fixed and variable partitioning schemes
- A modified form is used in Unix SVR4 for kernel memory allocation
- Memory blocks are available in size of  $2^{\{K\}}$  where  $L \leq K \leq U$  and where
  - ◆  $2^{\{L\}}$  = smallest size of block allocatable
  - ◆  $2^{\{U\}}$  = largest size of block allocatable (generally, the entire memory available)

# Buddy System

- We start with the entire block of size  $2^{\{U\}}$
- When a request of size  $S$  is made:
  - ◆ If  $2^{\{U-1\}} < S \leq 2^{\{U\}}$  then allocate the entire block of size  $2^{\{U\}}$
  - ◆ Else, split this block into two buddies, each of size  $2^{\{U-1\}}$
  - ◆ If  $2^{\{U-2\}} < S \leq 2^{\{U-1\}}$  then allocate one of the 2 buddies
  - ◆ Otherwise one of the 2 buddies is split again
- This process is repeated until the smallest block greater or equal to  $S$  is generated
- Two buddies are coalesced whenever both of them become unallocated

# Buddy System

- **The OS maintains several lists of holes**
  - ◆ the  $i$ -list is the list of holes of size  $2^{\{i\}}$
  - ◆ whenever a pair of buddies in the  $i$ -list occur, they are removed from that list and coalesced into a single hole in the  $(i+1)$ -list
- **Presented with a request for an allocation of size  $k$  such that  $2^{\{i-1\}} < k \leq 2^{\{i\}}$ :**
  - ◆ the  $i$ -list is first examined
  - ◆ if the  $i$ -list is empty, the  $(i+1)$ -list is then examined...



# Example of Buddy System

1 Mbyte block	1 M					
Request 100 K	A = 128 K	128 K	256 K	512 K		
Request 240 K	A = 128 K	128 K	B = 256 K	512 K		
Request 64 K	A = 128 K	C = 64 K	64 K	B = 256 K	512 K	
Request 256 K	A = 128 K	C = 64 K	64 K	B = 256 K	D = 256 K	256 K
Release B	A = 128 K	C = 64 K	64 K	256 K	D = 256 K	256 K
Release A	128 K	C = 64 K	64 K	256 K	D = 256 K	256 K
Request 75 K	E = 128 K	C = 64 K	64 K	256 K	D = 256 K	256 K
Release C	E = 128 K	128 K	256 K	D = 256 K	256 K	
Release E	512 K			D = 256 K	256 K	
Release D	1 M					

# Buddy Systems: remarks

- **On average, internal fragmentation is 25%**
  - ◆ each memory block is at least 50% occupied
- **Programs are not moved in memory**
  - ◆ simplifies memory management
- **Mostly efficient when the size  $M$  of memory used by the Buddy System is a power of 2**
  - ◆  $M = 2^{\{U\}}$  “bytes” where  $U$  is an integer
  - ◆ then the size of each block is a power of 2
  - ◆ the smallest block is of size 1
  - ◆ Ex: if  $M = 10$ , then the smallest block would be of size 5

# Relocation

- Because of swapping and compaction, a process may occupy different main memory locations during its lifetime
- Hence physical memory references by a process cannot be fixed
- This problem is solved by distinguishing between **logical address** and **physical address**

# Address Types

- A **physical address** (absolute address) is a physical location in main memory
- A **logical address** is a reference to a memory location independent of the physical structure/organization of memory
- Compilers produce code in which all memory references are logical addresses
- A **relative address** is an example of logical address in which the address is expressed as a location relative to some known point in the program (ex: the beginning)

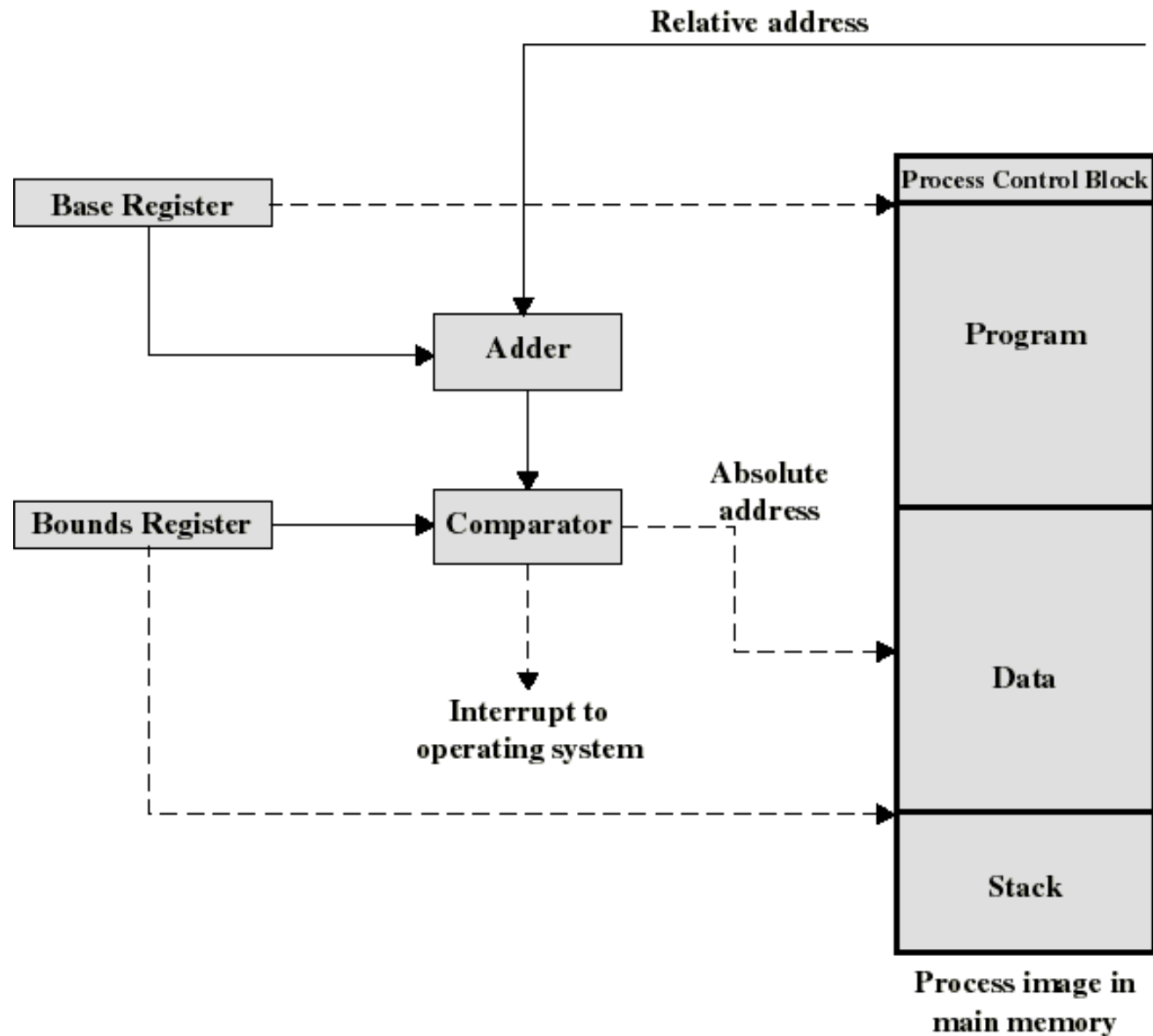
# Address Translation

- **Relative address is the most frequent type of logical address used in pgm modules (ie: executable files)**
- **Such modules are loaded in main memory with all memory references in relative form**
- **Physical addresses are calculated “on the fly” as the instructions are executed**
- **For adequate performance, the translation from relative to physical address must be done by hardware**

# Simple example of hardware translation of addresses

- When a process is assigned to the running state, a base register (in CPU) gets loaded with the starting physical address of the process
- A bound register gets loaded with the process's ending physical address
- When a relative addresses is encountered, it is added with the content of the base register to obtain the physical address which is compared with the content of the bound register
- This provides hardware protection: each process can only access memory within its process image

# Example Hardware for Address Translation



# Simple Paging

- Main memory is partitioned into equal fixed-sized chunks (of relatively small size)
- Trick: each process is also divided into chunks of the same size called **pages**
- The process pages can thus be assigned to the available chunks in main memory called **frames** (or page frames)
- Consequence: a process does not need to occupy a contiguous portion of memory



# Example of process loading

Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(a) Fifteen Available Pages

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(b) Load Process A

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	
8	
9	
10	
11	
12	
13	
14	

(b) Load Process B

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(d) Load Process C

- Now suppose that process B is swapped out

# Example of process loading (cont.)

- When process A and C are blocked, the pager loads a new process D consisting of 5 pages
- Process D does not occupied a contiguous portion of memory
- There is no external fragmentation
- Internal fragmentation consist only of the last page of each process

Main memory	
0	A.0
1	A.1
2	A.2
3	A.3
4	
5	
6	
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

(e) Swap out B

Main memory	
0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

(f) Load Process D

# Page Tables

0	0
1	1
2	2
3	3

Process A  
page table

0	—
1	—
2	—

Process B  
page table

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

13
14

Free frame  
list

- The OS now needs to maintain (in main memory) a **page table** for each process
- Each entry of a page table consist of the frame number where the corresponding page is physically located
- The page table is indexed by the page number to obtain the frame number
- A free frame list, available for pages, is maintained

# Logical address used in paging

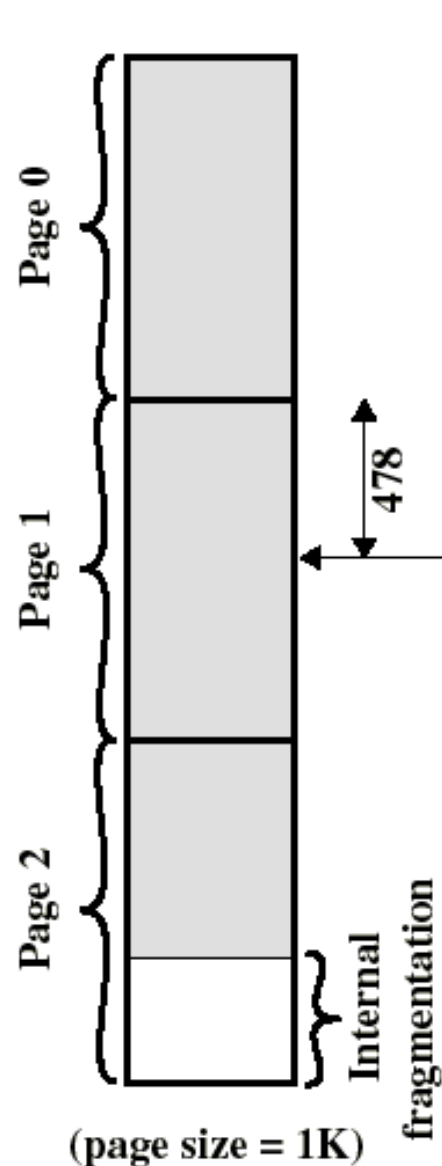
- Within each program, each logical address must consist of a **page number** and an **offset** within the page
- A CPU register always holds the starting physical address of the page table of the currently running process
- Presented with the logical address (page number, offset) the processor accesses the page table to obtain the physical address (frame number, offset)

# Logical address in paging

- The logical address becomes a relative address when the page size is a power of 2
- Ex: if 16 bits addresses are used and page size = 1K, we need 10 bits for offset and have 6 bits available for page number
- Then the 16 bit address obtained with the 10 least significant bit as offset and 6 most significant bit as page number is a location relative to the beginning of the process

Logical address =  
Page# = 1, Offset = 478

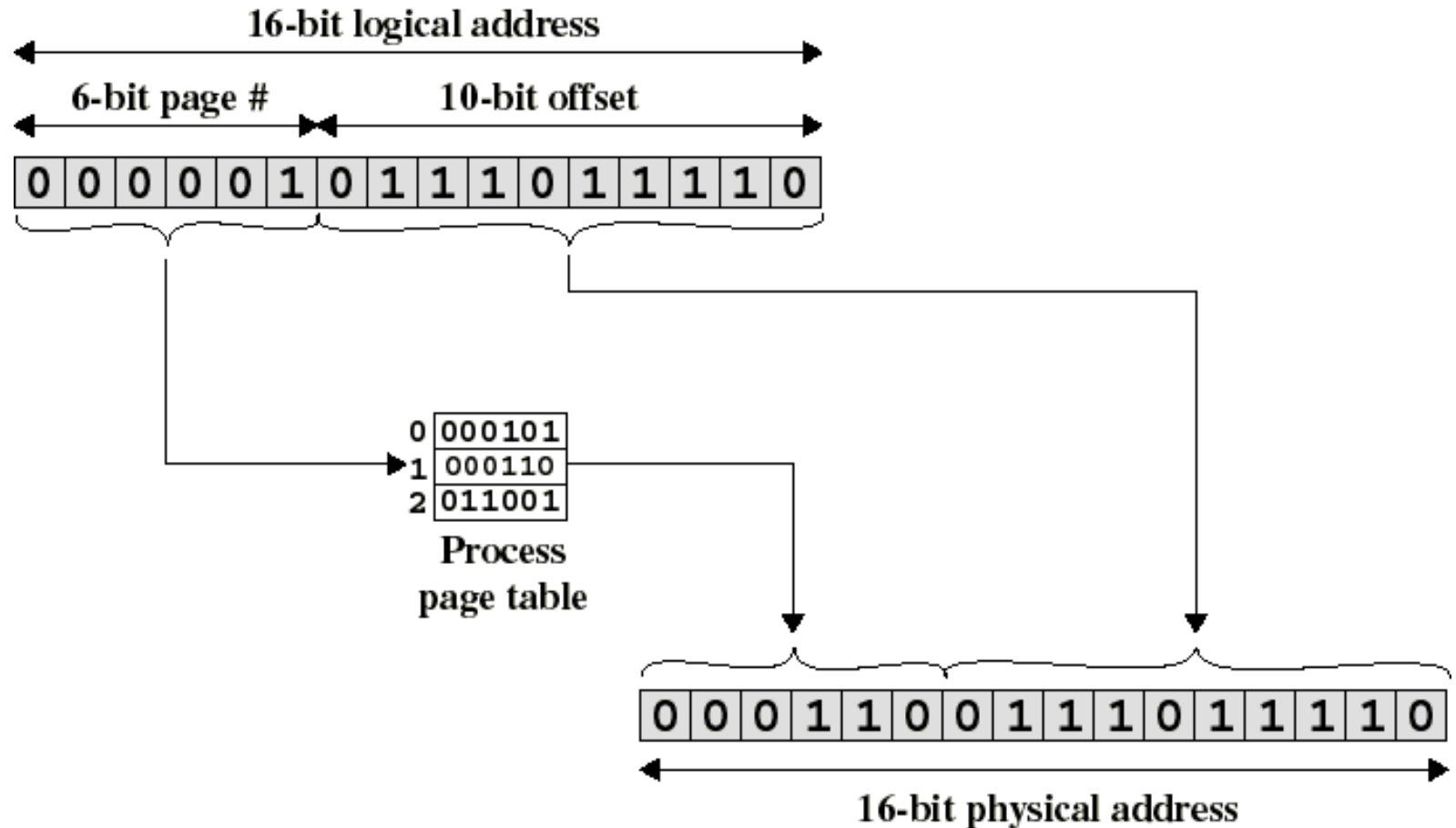
0000010111011110



# Logical address in paging

- **By using a page size of a power of 2, the pages are invisible to the programmer, compiler/assembler, and the linker**
- **Address translation at run-time is then easy to implement in hardware**
  - ◆ logical address  $(n,m)$  gets translated to physical address  $(k,m)$  by indexing the page table and appending the same offset  $m$  to the frame number  $k$

# Logical-to-Physical Address Translation in Paging



# Simple Segmentation

- Each program is subdivided into blocks of non-equal size called **segments**
- When a process gets loaded into main memory, its different segments can be located anywhere
- Each segment is fully packed with instructions/data: no internal fragmentation
- There is external fragmentation; it is reduced when using small segments



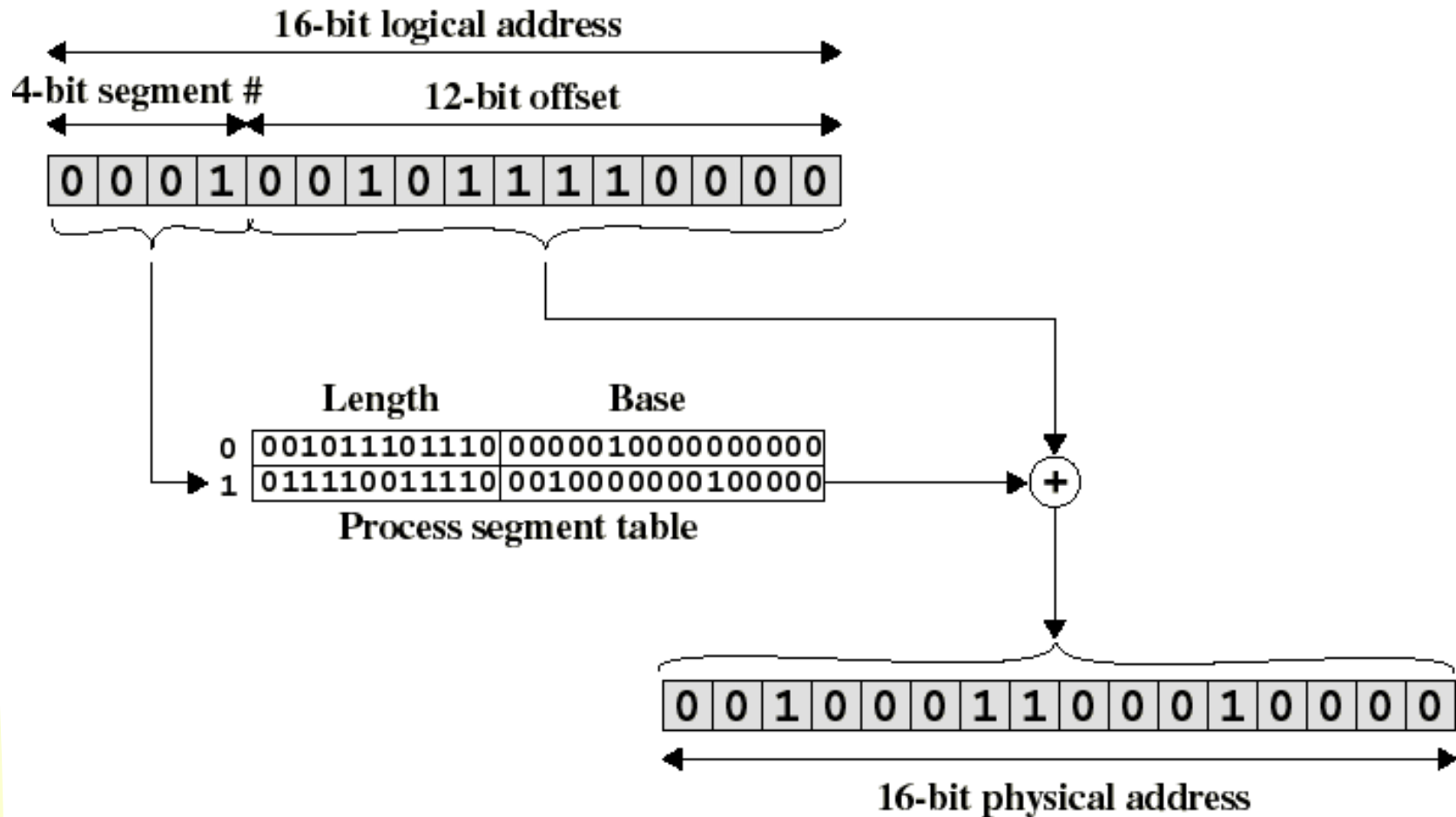
# Simple Segmentation

- **In contrast with paging, segmentation is visible to the programmer**
  - ◆ provided as a convenience to organize logically programs (ex: data in one segment, code in another segment)
  - ◆ must be aware of segment size limit
- **The OS maintains a **segment table** for each process. Each entry contains:**
  - ◆ the starting physical addresses of that segment.
  - ◆ the length of that segment (for protection)

# Logical address used in segmentation

- When a process enters the Running state, a CPU register gets loaded with the starting address of the process's segment table.
- Presented with a **logical address (segment number, offset) = (n,m)**, the CPU indexes (with n) the segment table to obtain the starting physical address k and the length l of that segment
- The physical address is obtained by **adding** m to k (in contrast with paging)
  - ◆ the hardware also compares the offset m with the length l of that segment to determine if the address is valid

# Logical-to-Physical Address Translation in segmentation



# Simple segmentation and paging comparison

- **Segmentation requires more complicated hardware for address translation**
- **Segmentation suffers from external fragmentation**
- **Paging only yield a small internal fragmentation**
- **Segmentation is visible to the programmer whereas paging is transparent**
- **Segmentation can be viewed as commodity offered to the programmer to organize logically a program into segments and using different kinds of protection (ex: execute-only for code but read-write for data)**
  - ◆ **for this we need to use protection bits in segment table entries**