Memory Management

Chapter 7

Memory Management

- OS + hardware: subdivide memory to accommodate multiple processes
- Memory: allocated to ensure reasonable supply of ready processes (consume available CPU time)
- Few processes in main memory?
  - Much of the time all processes will be waiting for I/O
  - CPU will be idle
- Memory needs to be allocated efficiently in order to pack as many processes into memory as possible

Memory Management

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

Memory Management Requirements

- Relocation
  - Programmer: not know where program will be in memory when executed
  - process may be 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

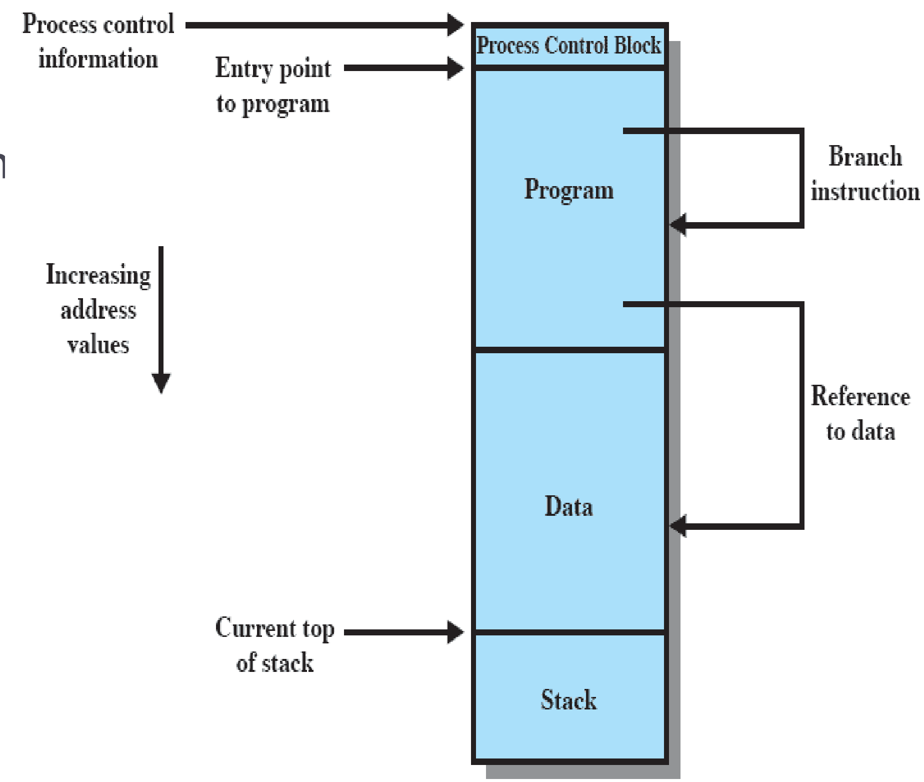


Figure 7.1 Addressing Requirements for a Process

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
    - Operating system cannot anticipate all of the memory references a program will make

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, e.g., parent and child processes after fork().

Memory Management Requirements

- Logical Organization
  - Programs are written in modules
  - Modules can be written and compiled independently
  - Different degrees of protection given to modules (read-only, execute-only)
  - Share modules among processes
  - 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
  - Memory available for a program plus its data may be insufficient. 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
    - Overlaying allows various modules to be assigned the same region of memory
  - Programmer does not know how much space will be available

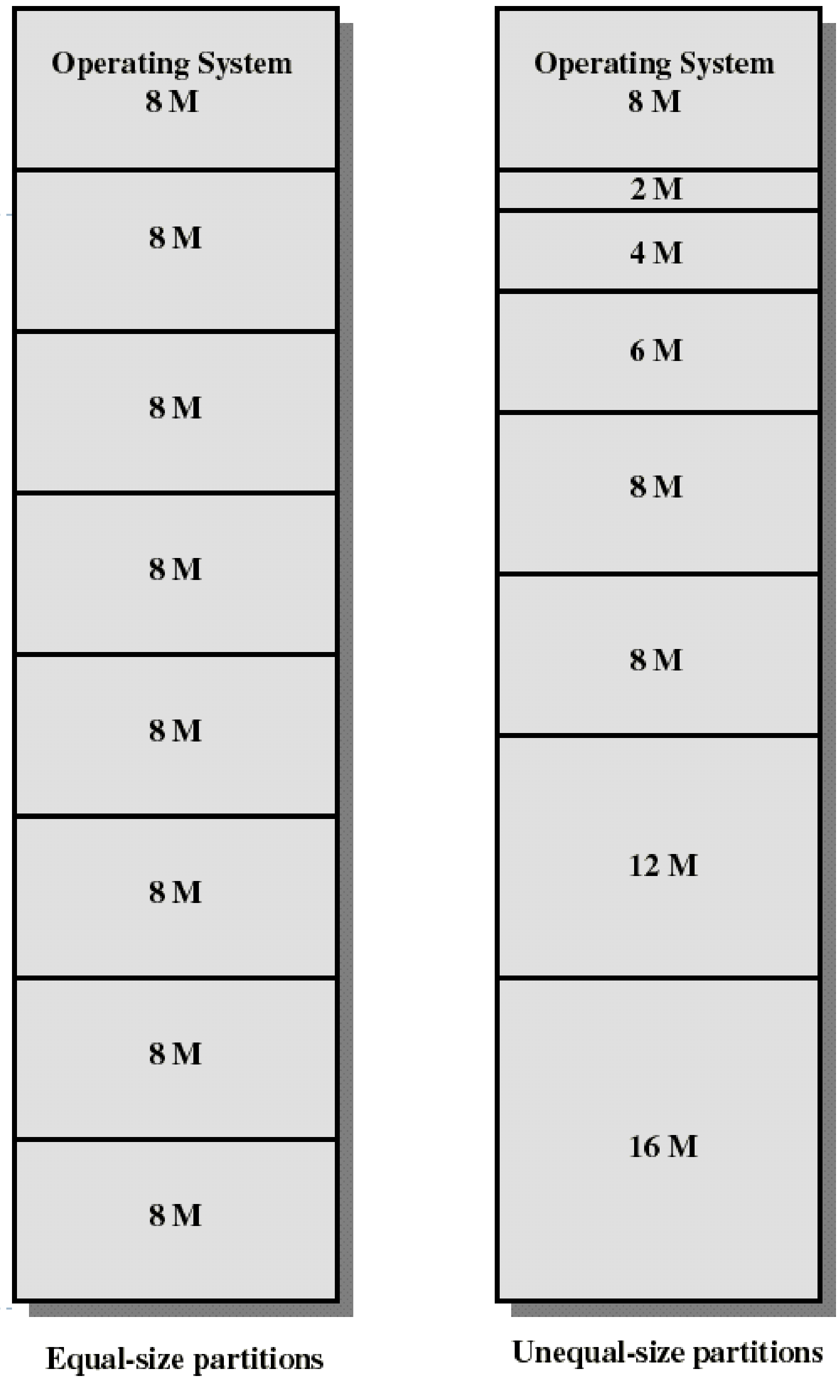
Simple Memory Management

- First 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 (to be discussed later)
  - 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



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

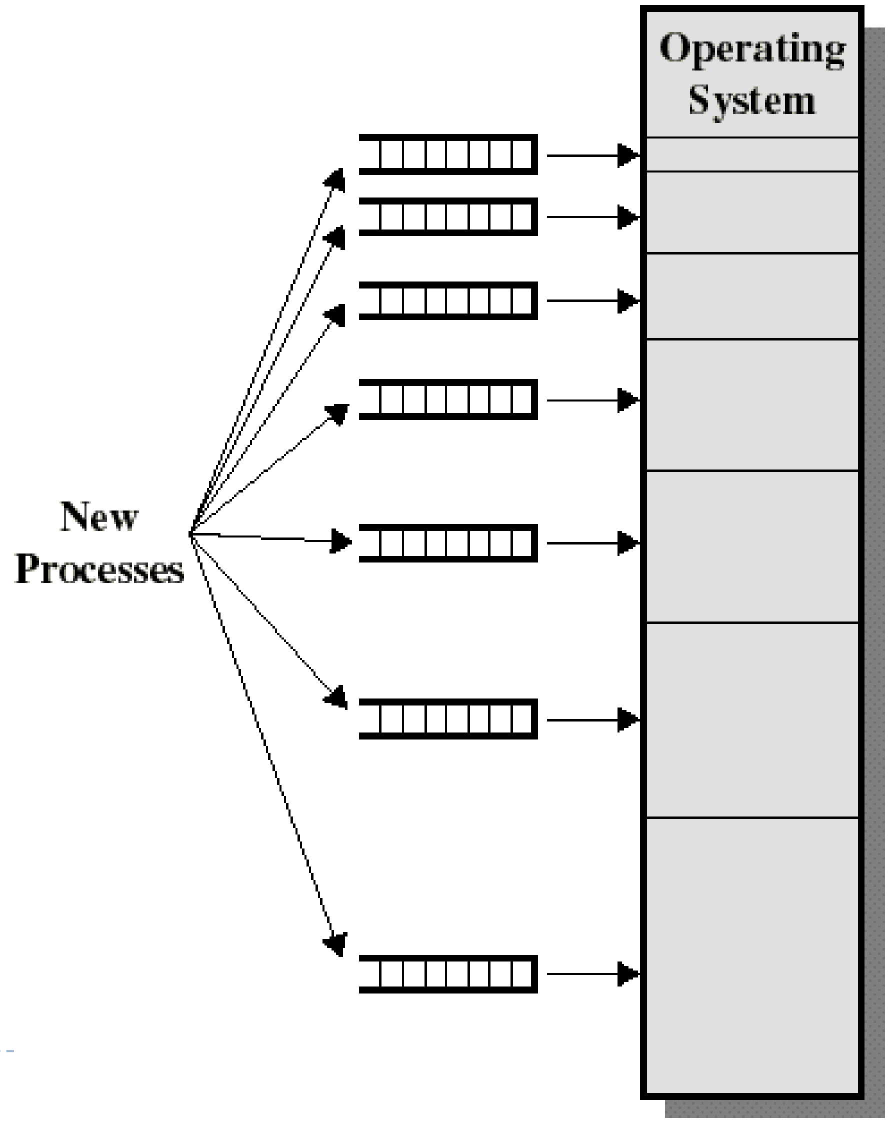
- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.
- Unequal-size partitions lessens these problems but they still remain...
- Equal-size partitions was used in early IBM's OS/MFT (Multiprogramming with a Fixed number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

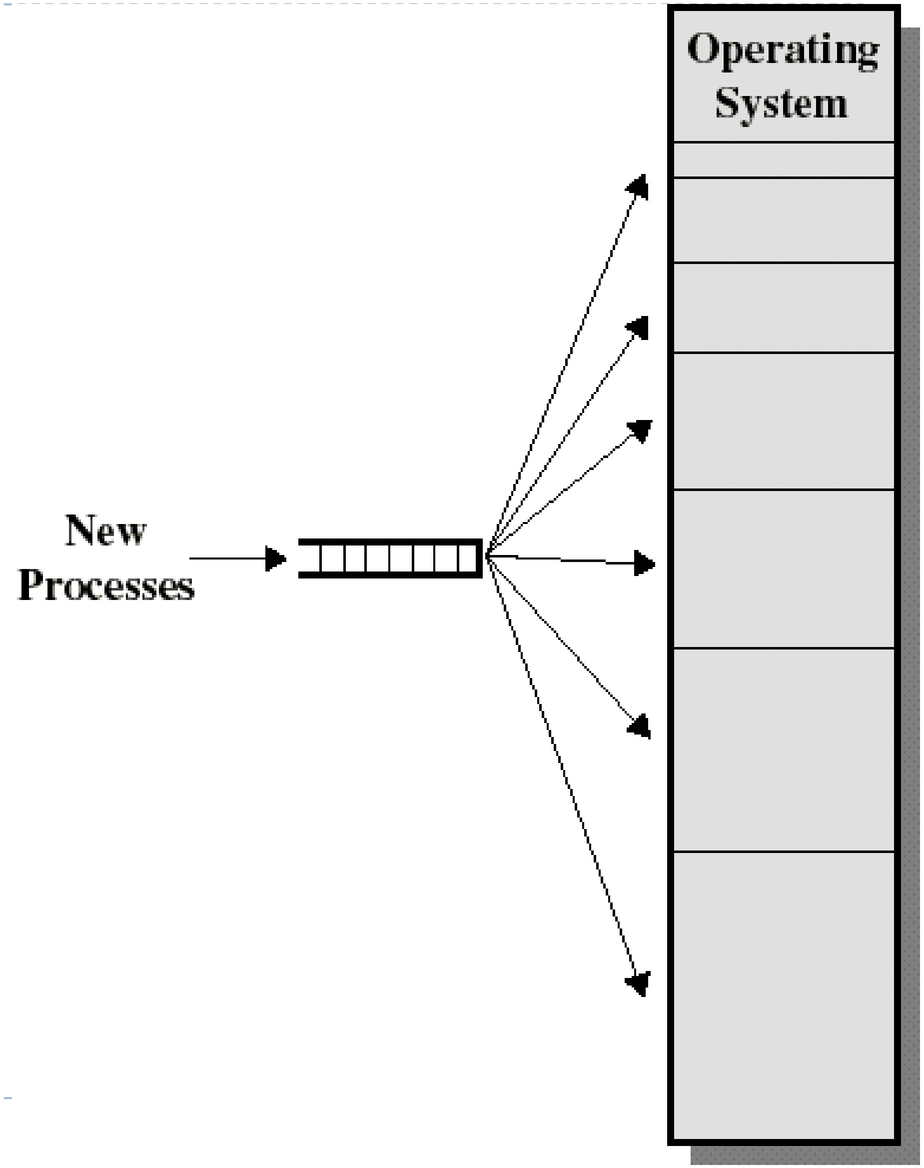
Placement Algorithm with Partitions

- 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

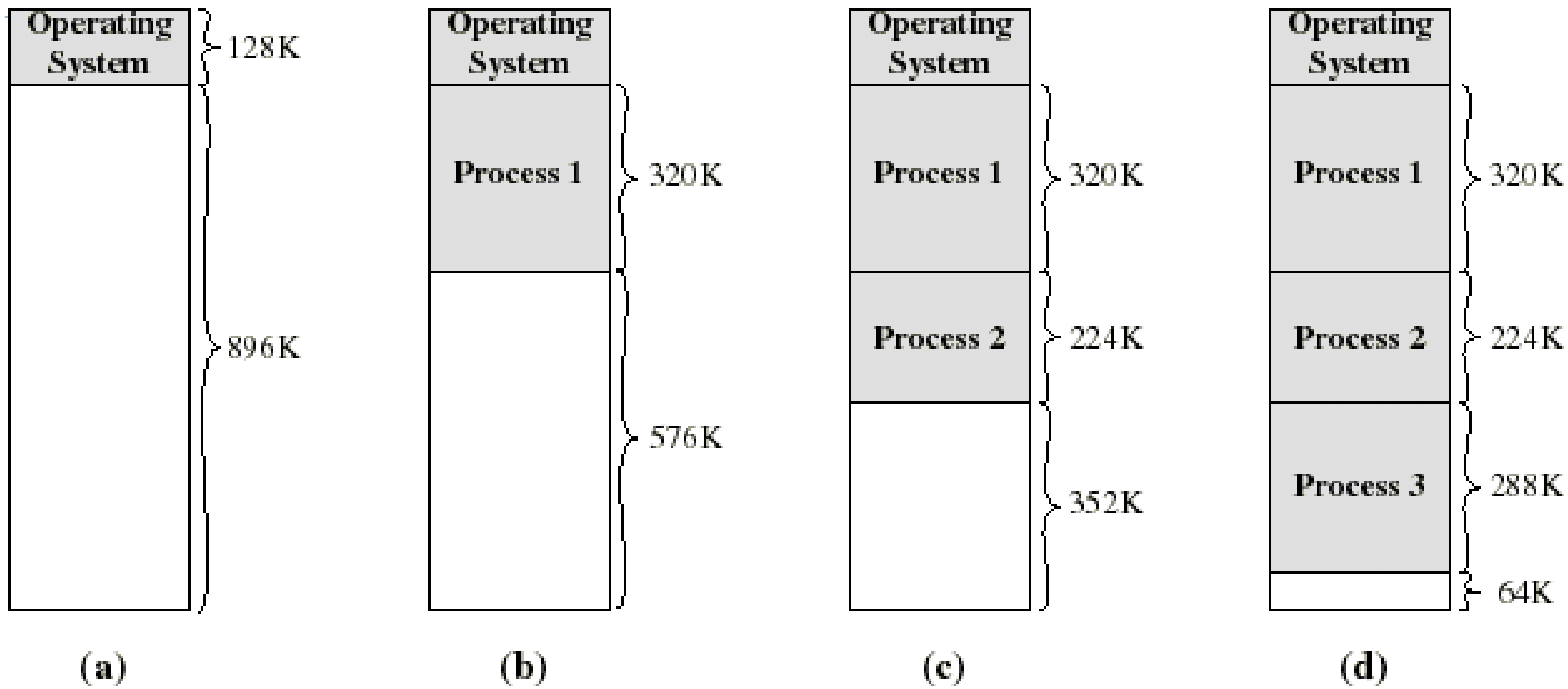
- Unequal-size partitions: use of a single queue
  - When its time to load a process into main memory the smallest available partition that will hold the process is selected
  - increases the level of multiprogramming at the expense of internal fragmentation
    - The smallest available partition may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



Dynamic Partitioning

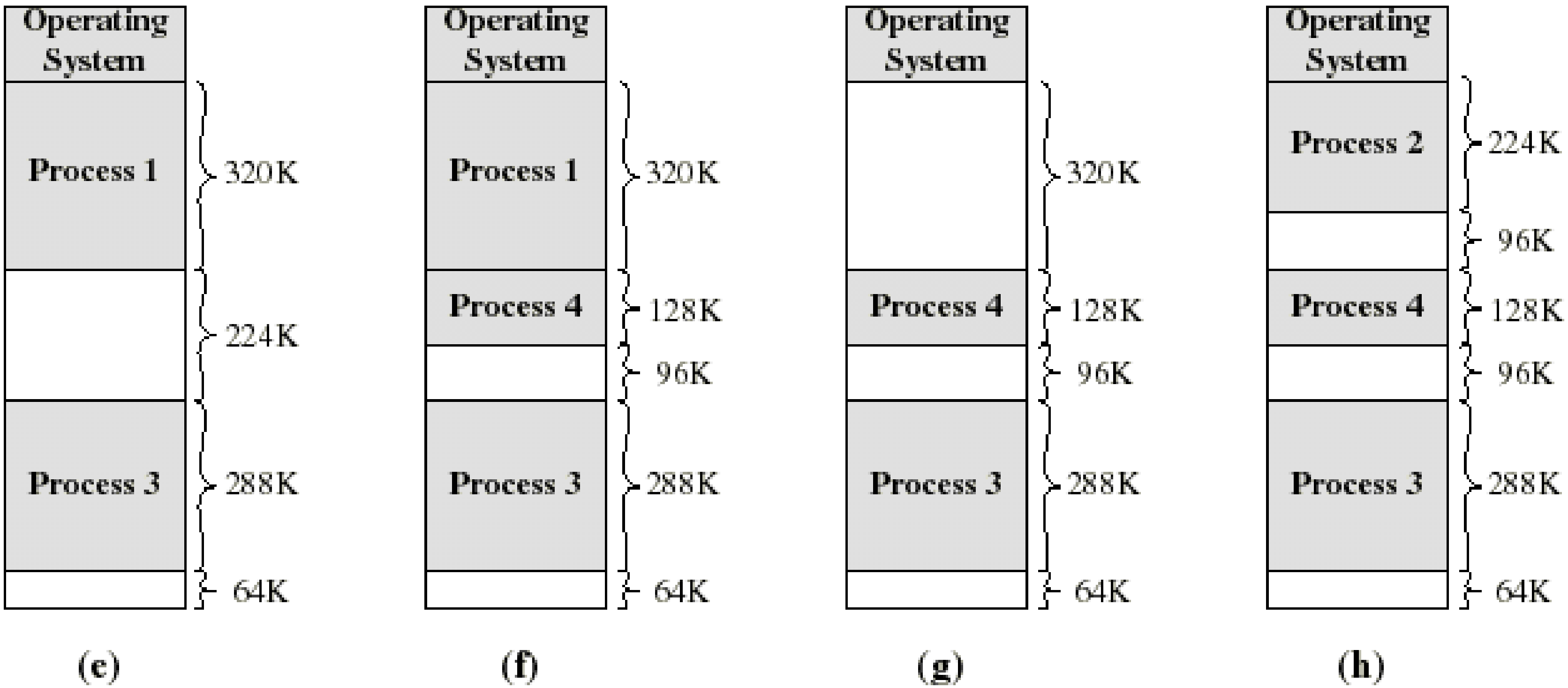
- Partitions are of variable length and number
- Each process is allocated exactly as much memory as it requires
- Eventually holes are formed in main memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block
- Used 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

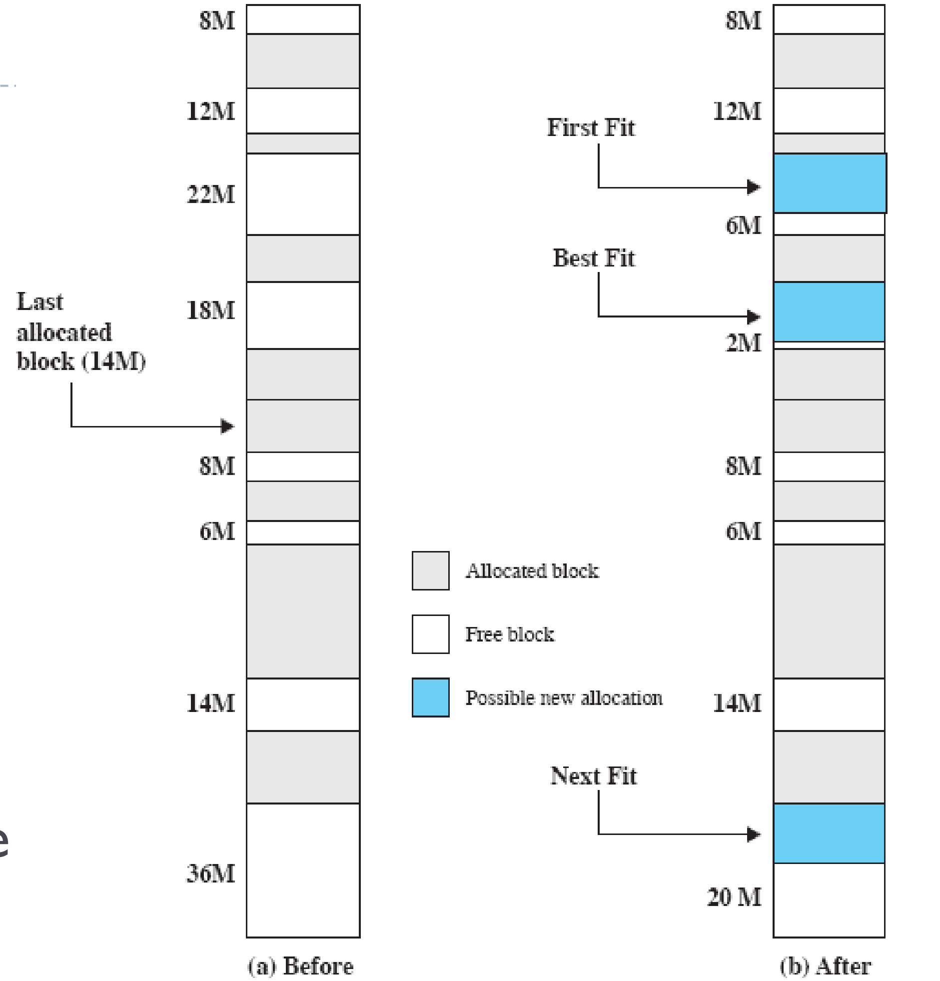


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

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 then 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

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s <= 2^U$, 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

Example of Buddy System

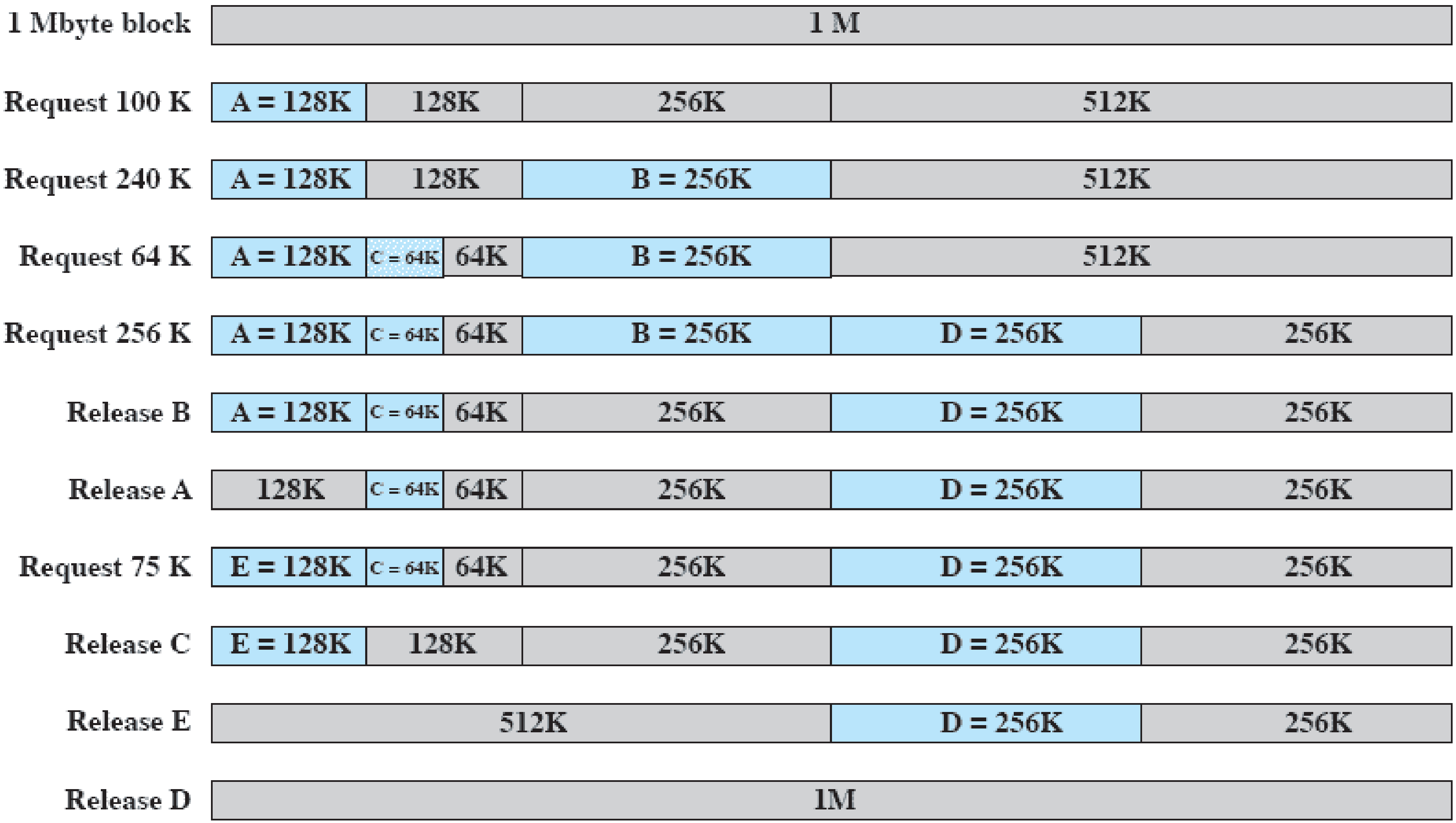


Figure 7.6 Example of Buddy System

Tree Representation of Buddy System

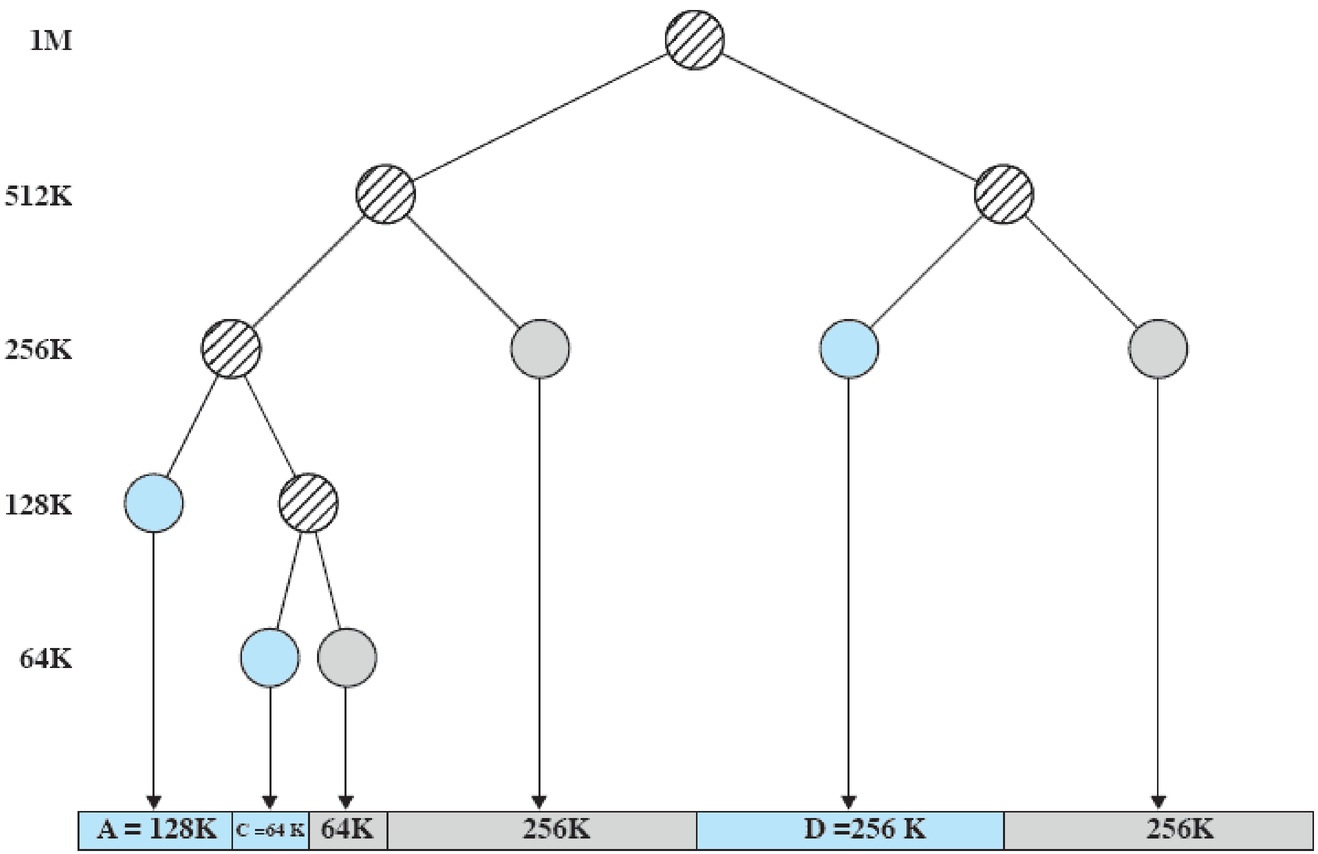


Figure 7.7 Tree Representation of Buddy System

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

Relocation

- When program is loaded into memory, the actual (absolute) memory locations are determined
- 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
- 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
- This is called dynamic run-time loading
- For adequate performance, the translation from relative to physical address must by done by hardware

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

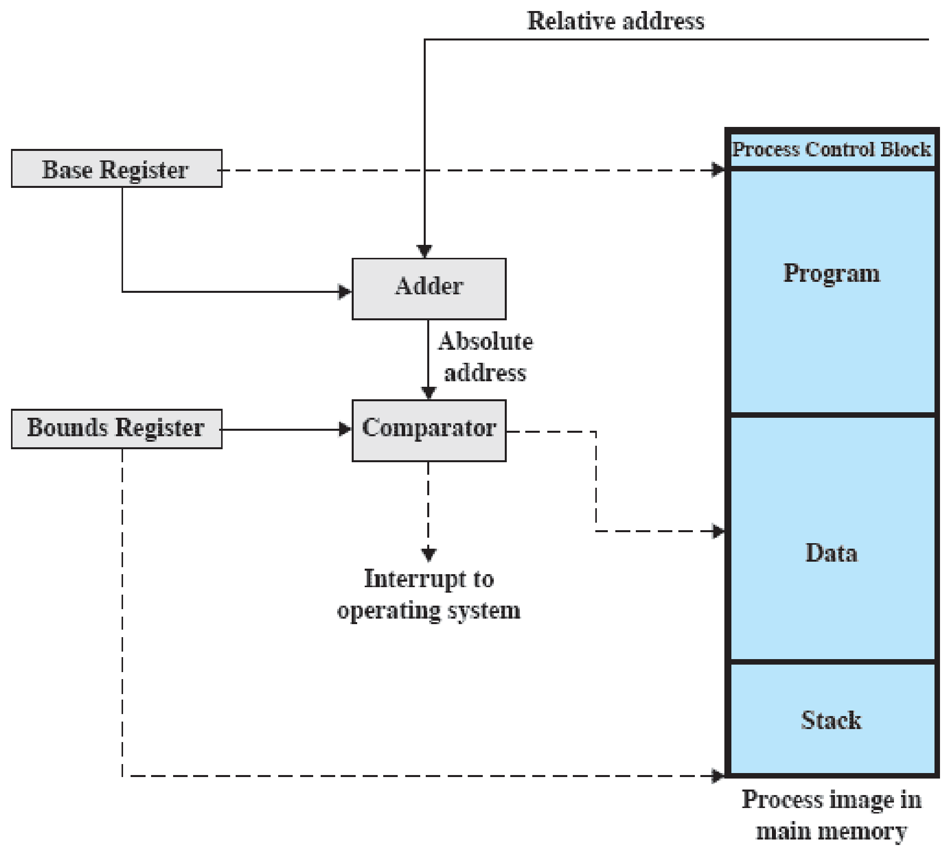


Figure 7.8 Hardware Support for Relocation

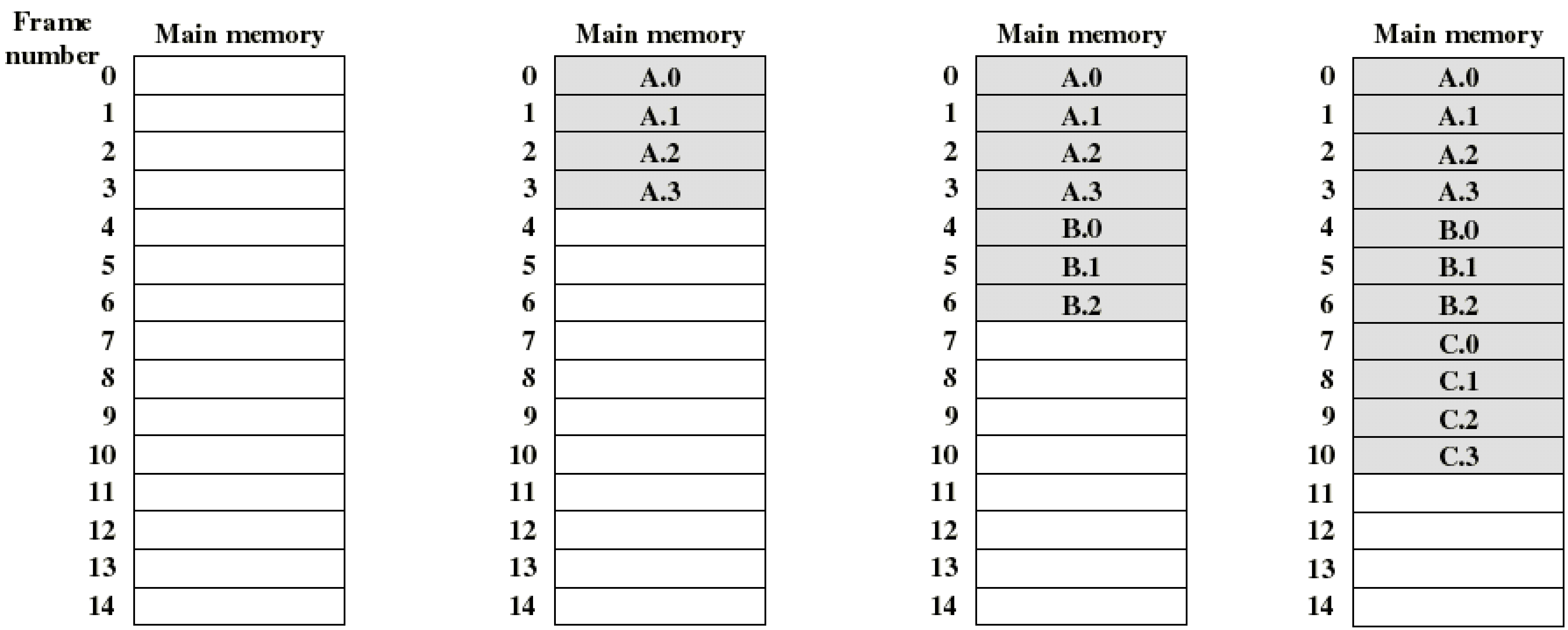
Paging

- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called pages and chunks of memory are called frames
- Consequence: a process does not need to occupy a contiguous portion of memory

Paging

- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consists of a page number and offset within the page

Example of process loading

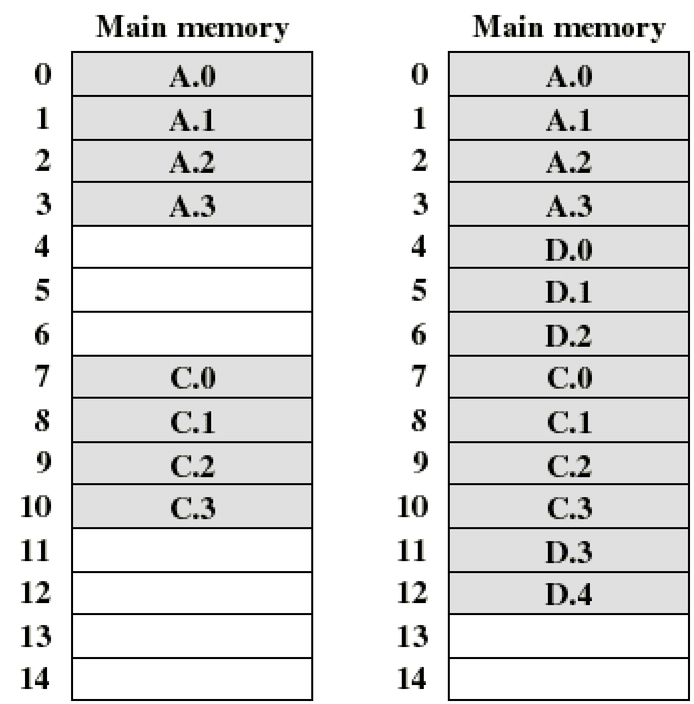


(a) Fifteen Available Pages (b) Load Process A (b) Load Process B (d) Load Process C

- Now suppose that process B is swapped out

Example of process loading (cont.)

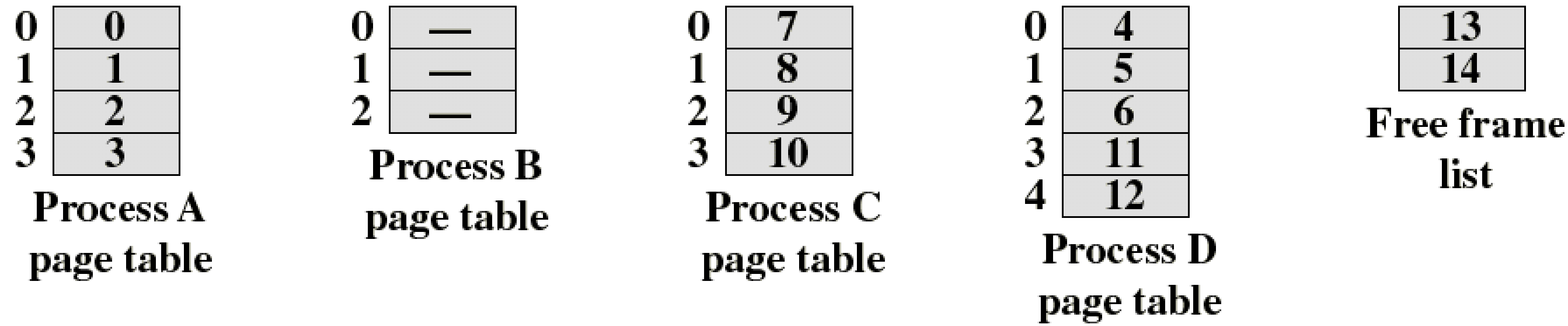
- When process A and C are blocked, the pager loads a new process D consisting of 5 pages
- Process D does not occupied a contiguous portion of memory
- There is no external fragmentation
- Internal fragmentation consist only of the last page of each process



(e) Swap out B

(f) Load Process D

Page Tables



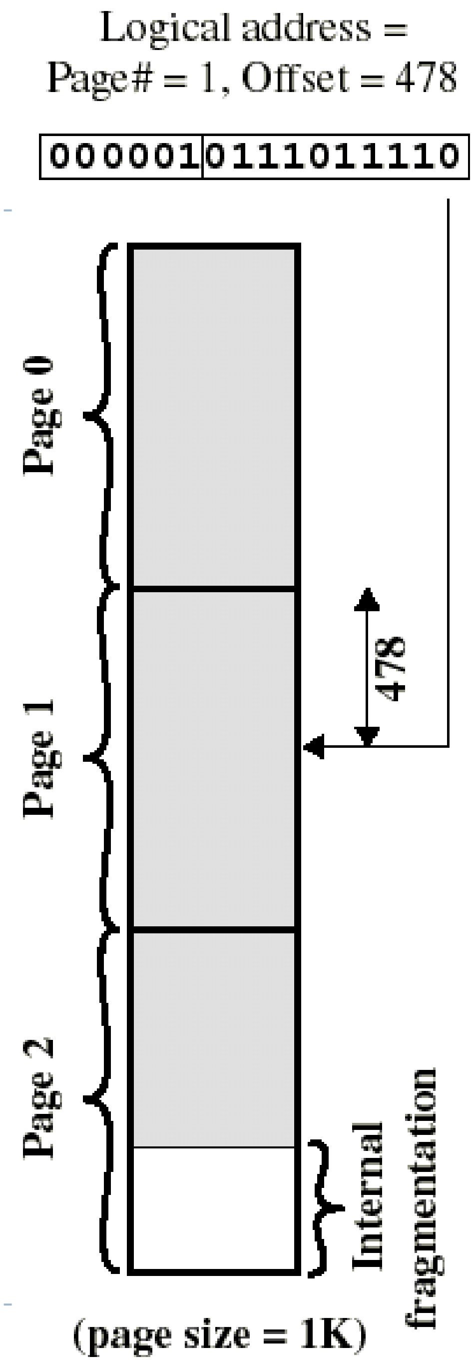
- The OS now needs to maintain (in main memory) a page table for each process
- Each entry of a page table consist of the frame number where the corresponding page is physically located
- The page table is indexed by the page number to obtain the frame number
- A free frame list, available for pages,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

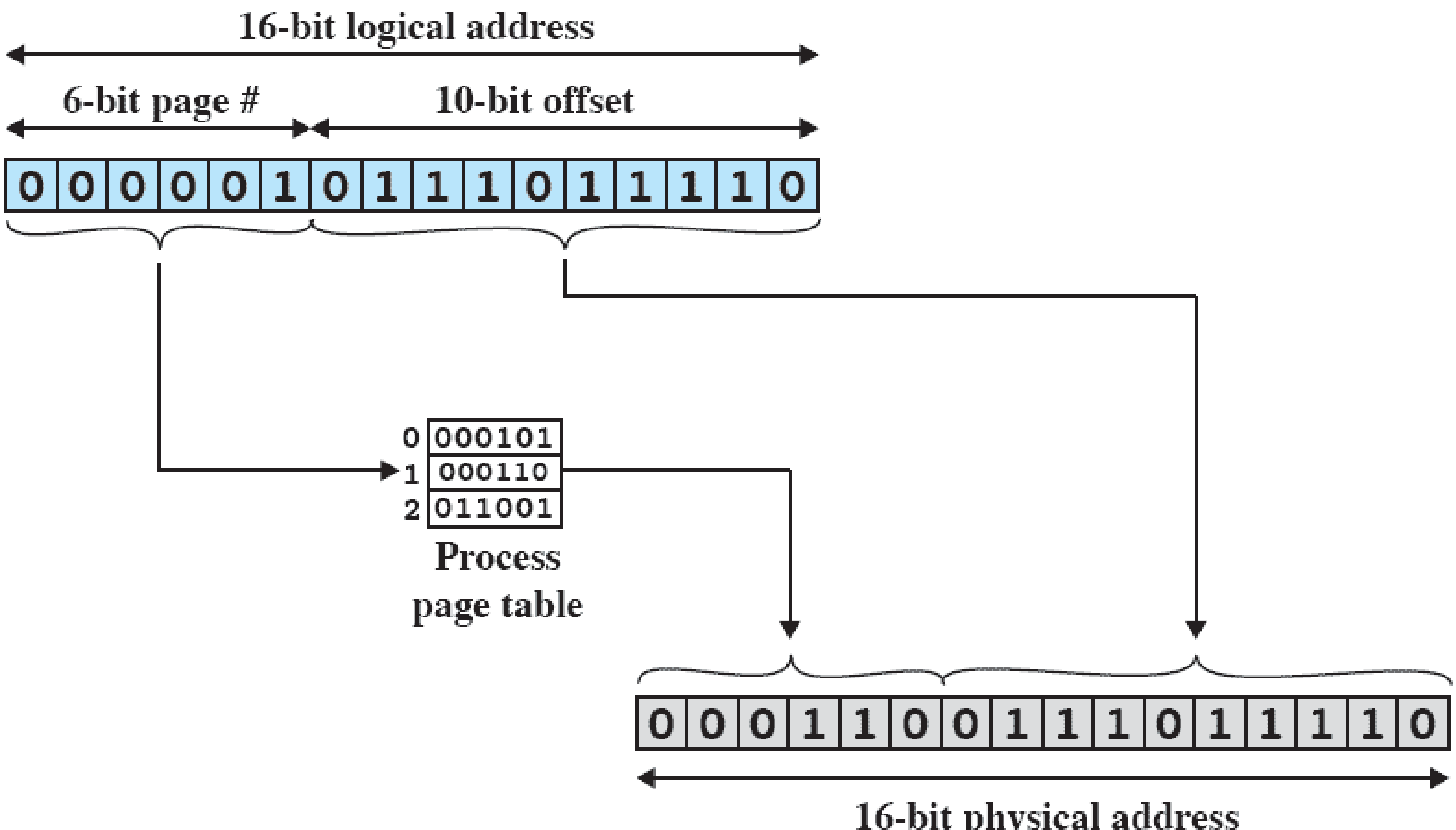
- The logical address becomes a relative address when the page size is a power of 2
- Ex: if 16 bits addresses are used and page size = 1K, we need 10 bits for offset and have 6 bits available for page number
- Then the 16 bit address obtained with the 10 least significant bit as offset and 6 most significant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



Logical address 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



(a) Paging

Simple Segmentation

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

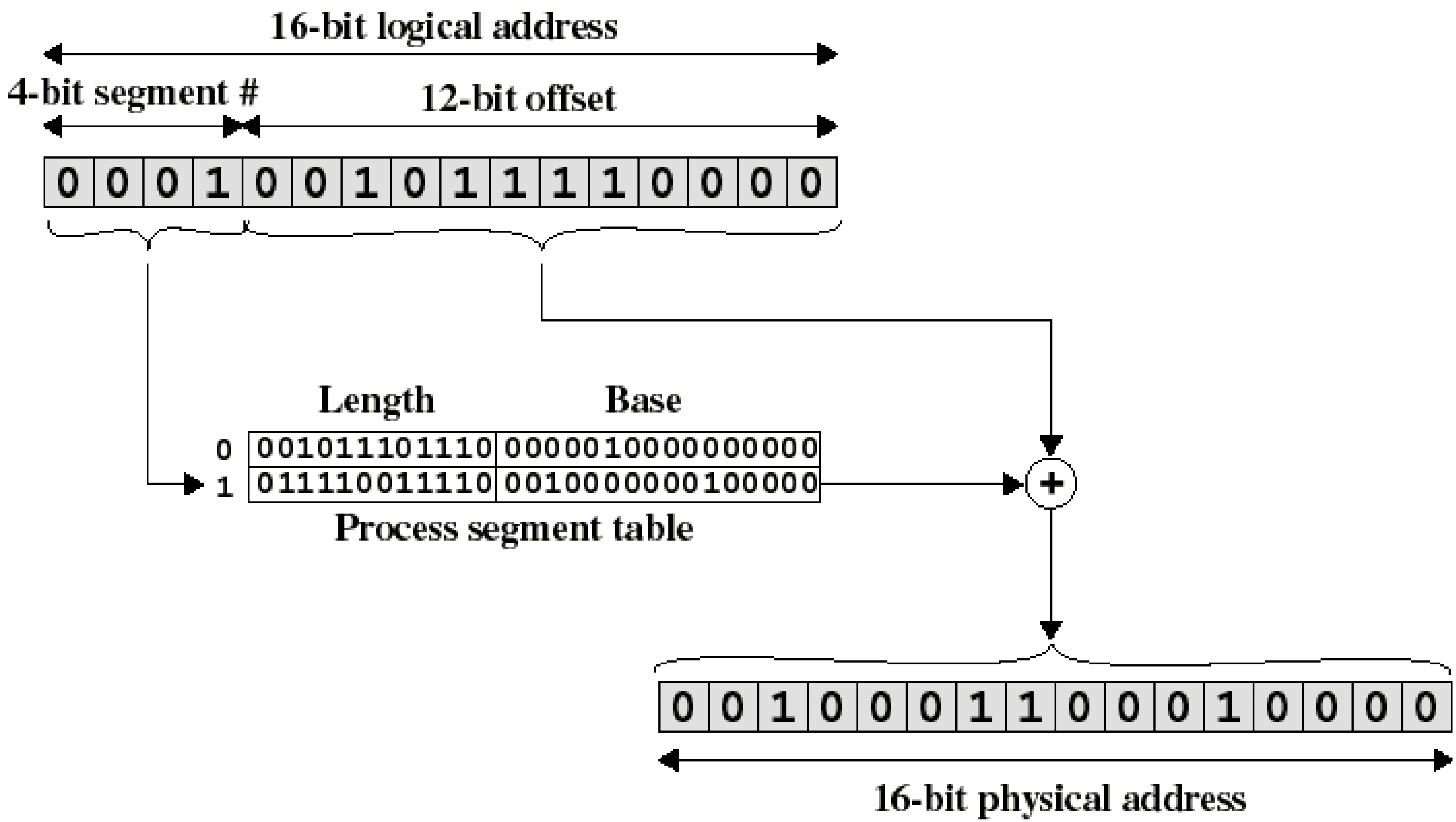
Simple Segmentation

- 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
  - we cannot directly obtain the logical address from the physical address (in contrast with paging)

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/compiler 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