Chapter 7
Memory Management

Memory Management

- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time

Memory Management Requirements

- Relocation
  - Programmer does not know where the program will be placed in memory when it is executed
  - While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
  - Memory references must be translated in the code to actual physical memory address

We need to allow the process to shift around in main memory, so when we refer to a global variable in the process things get wonky because the address of the global variable gets mucked when we move things.

Addressing Requirement

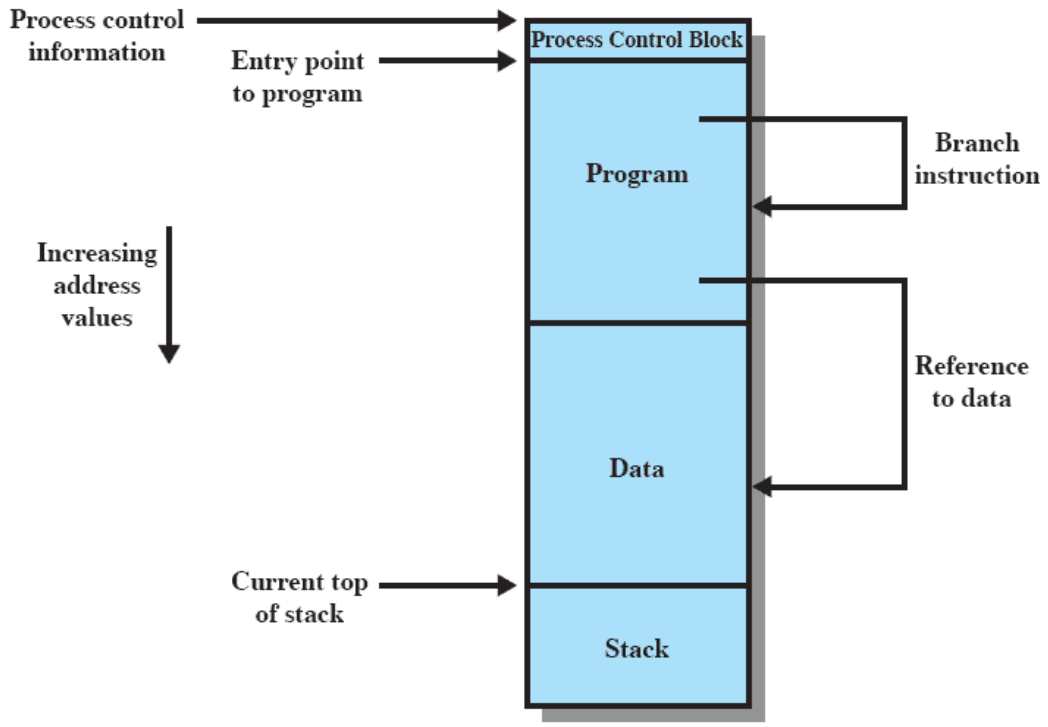


Figure 7.1 Addressing Requirements for a Process

Here is an example of references.

Memory Management Requirements

- Protection
  - Processes should not be able to reference memory locations in another process without permission
  - Impossible to check absolute addresses at compile time
    => Must be checked at run time
    => see your ‘segmentation fault’ bugs

Requirement 1: how do we implement relocation?

Requirement 2: how do we implement protection?
We want to make sure that all memory references a process makes is within its owned data and catch when it tries to access memory it doesnt own in main memory.

Memory Management Requirements

- Sharing
  - Allow several processes to access the same portion of memory
  - Better to allow each process access to the same copy of the program rather than have their own separate copy

Requirement 3: how do we implement sharing?
We need to allocate a shared peice of memory (shared by process A and B)

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

Memory Management Requirements

- Physical Organization
  - Memory available for a program plus its data may be insufficient
    - Overlaying allows various modules to be assigned the same region of memory
  - Programmer does not know how much space will be available

Requrement 5: how do we implement physical organization
How do we organize the physical space so that we can have enough for each program without knowing what they need at the start.

APPROACHES

One method to implement physical organization is to have a fixed chunk size of memory. When a process is loaded it is stored in one chunk (programs are limited by chunk size since they cannot span multiple chunks). Here the programmer doesnt need to know what the total size of memory is but is instead constrained in the size of their program. It is also inefficient because process that is smaller than the chunk has unused space resulting in fragmented memory. Internal fragmentation only occurs here when the process image is smaller than the partition size.

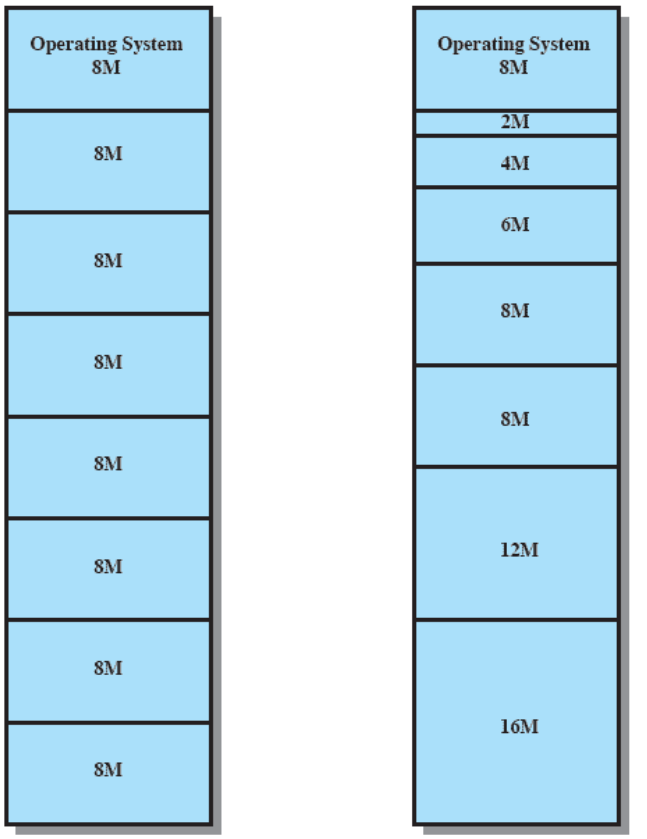
Fixed Partitioning

- Equal-size partitions
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
  - If all partitions are full, the operating system can swap a process out of a partition

Fixed Partitioning

- Equal-size partitions (problems)
  - A program may not fit in a partition. The programmer must design the program with overlays
  - Main memory use is inefficient. Any program, no matter how small, occupies an entire partition.
    - This is called internal fragmentation.

Fixed Partitioning



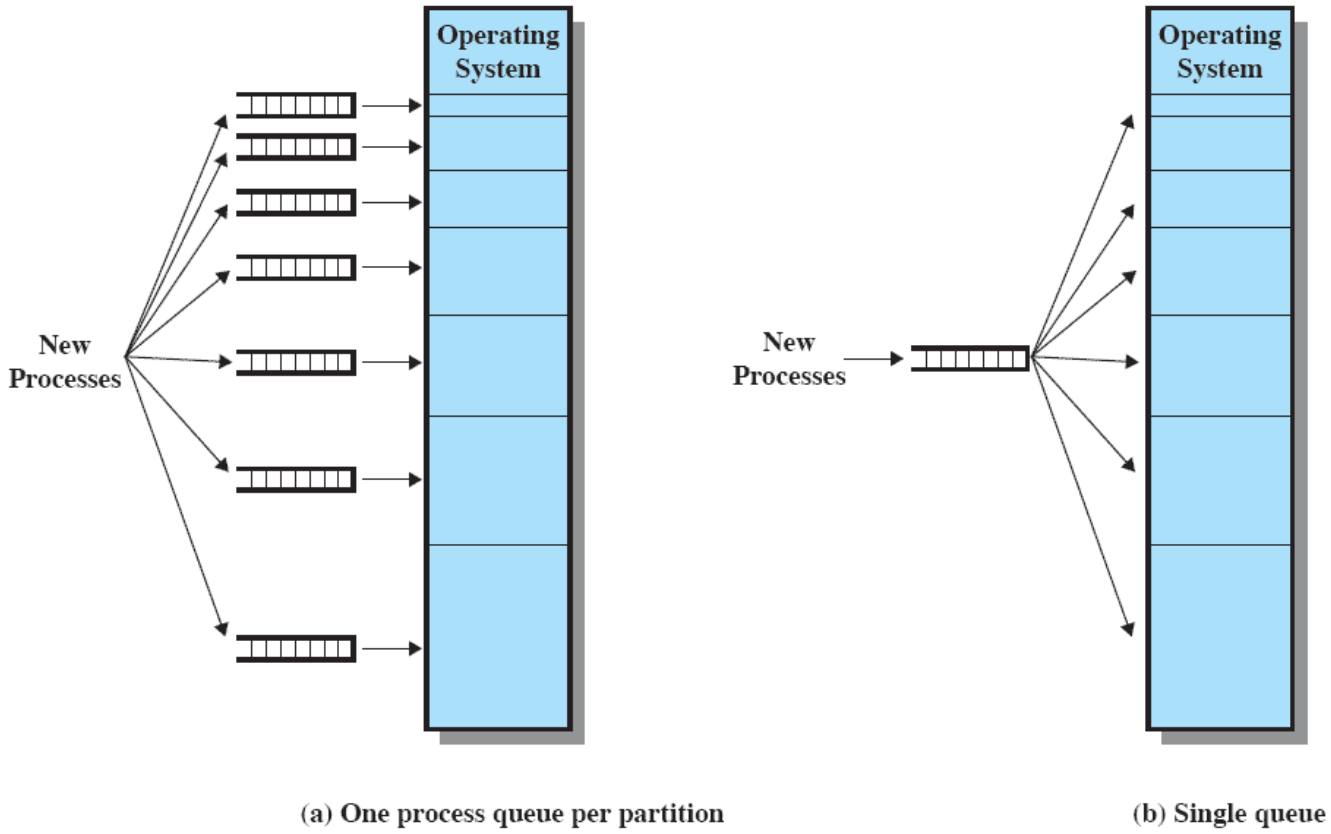
(a) Equal-size partitions (b) Unequal-size partitions

Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Placement Algorithm

- Equal-size
  - Placement it trivial
- Unequal-size
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition

Fixed Partitioning



(a) One process queue per partition

(b) Single queue

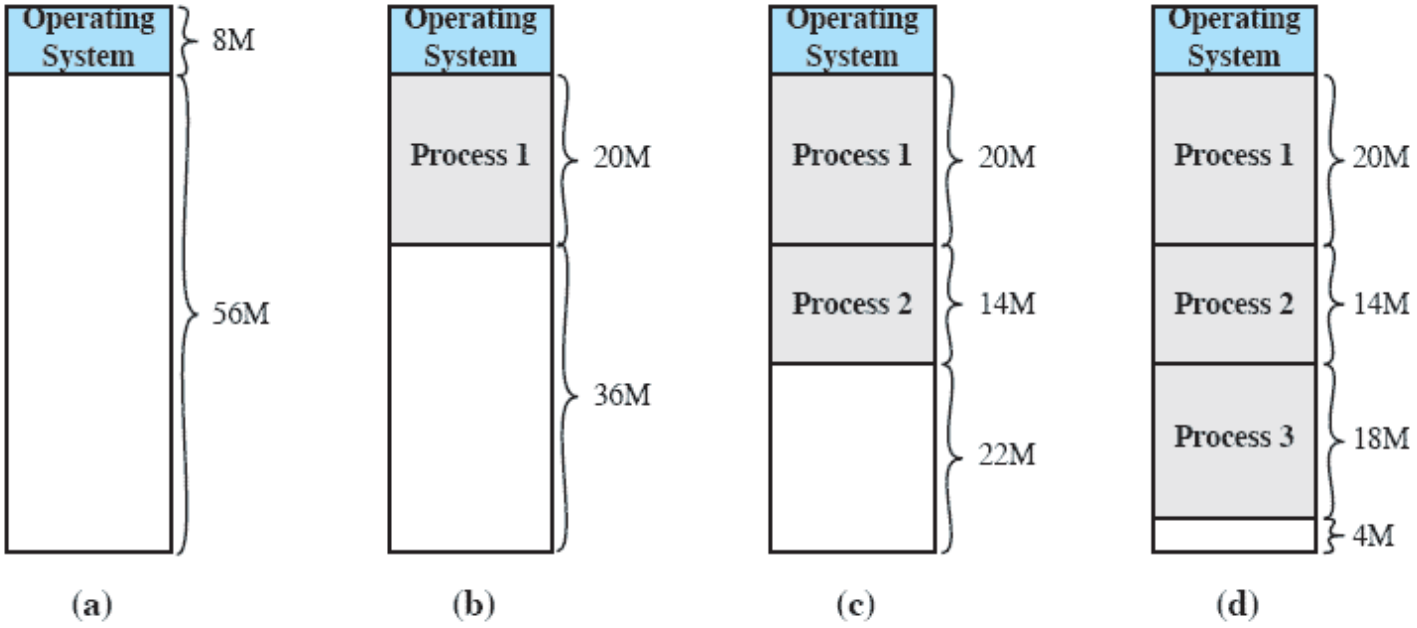
Figure 7.3 Memory Assignment for Fixed Partitioning

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually get holes in the memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block

This is one step up. Instead of fixing the size of the partion at boot up we expand the allocated size to exactly what the process needs. We can still get fragmentation, but here its called external fragmentation.

Dynamic Partitioning



We leave the entire chunk of memory unallocated and when a process is loaded we give it exactly the size of its program image.

Dynamic Partitioning

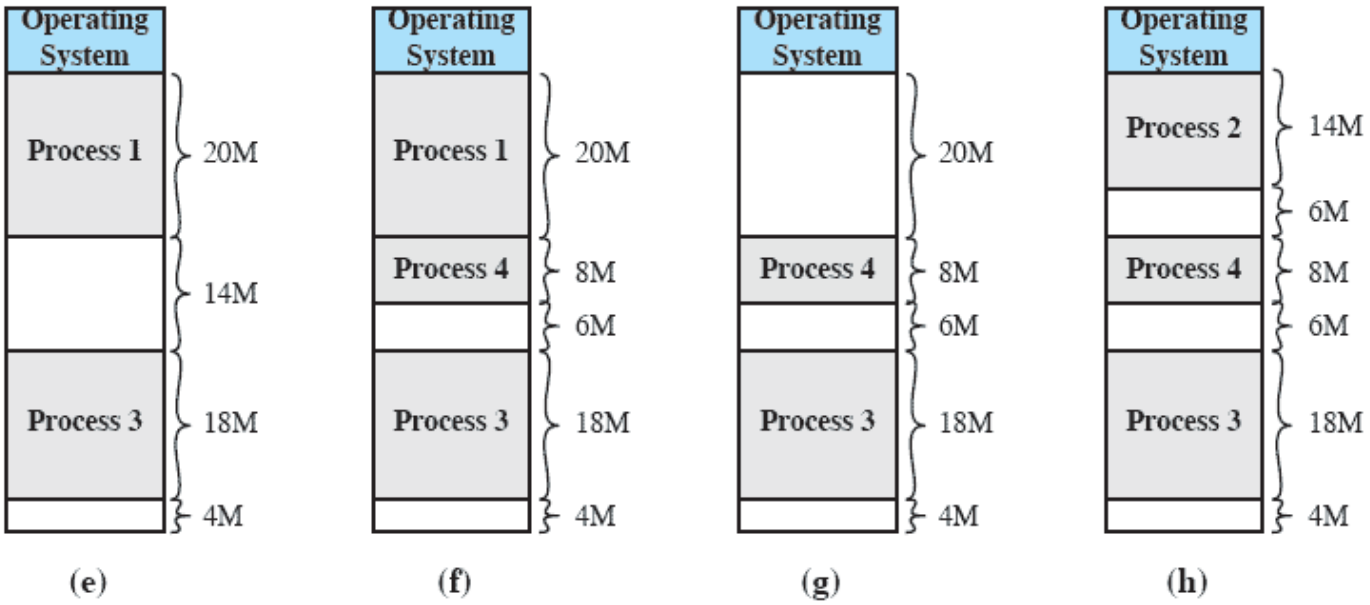


Figure 7.4 The Effect of Dynamic Partitioning

Here we see holes appearing when we swap processes in and out.

Dynamic Partitioning

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often

Look at available memory blocks and pick the block that is closest to your memory size. Over time this is the worst fragmenter of you memory.

Dynamic Partitioning

- First-fit algorithm
  - Scans memory from the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many processes loaded in the front end of memory that must be searched over when trying to find a free block

Choses the first block that is large enough for your process. This is the fastest but we get lots of processes at the front of the memory block so scanning takes longer as you go.

Dynamic Partitioning

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocates a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

Start allocating space from the last allocated space of memory (think of a linked list ish).

Allocation

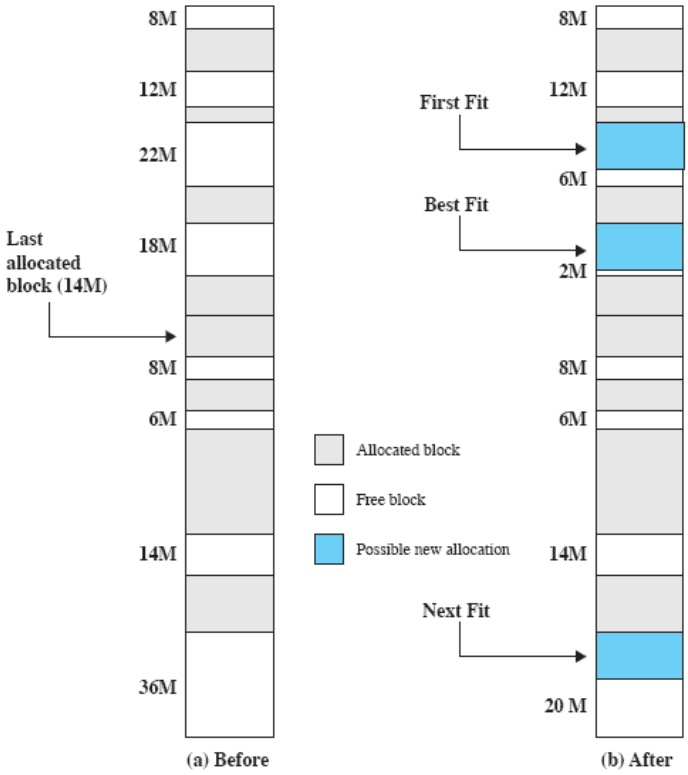


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size *s* such that
  $2^{U-1} < s <= 2^U$
  then the entire block will be allocated
  - Otherwise the block is split into two equal sized buddies
  - Process continues until smallest block greater than or equal to *s* is generated

Take all of your memory called 2^u always want the size of our process to be $2^{u-1} < s \leq 2^u$. This becomes a recursive algorithm. We keep dividing memory in half until we find a place for the program that fulfills the previous inequality. We keep a tree of allocations, for each level we store them as a list.

Example of Buddy System

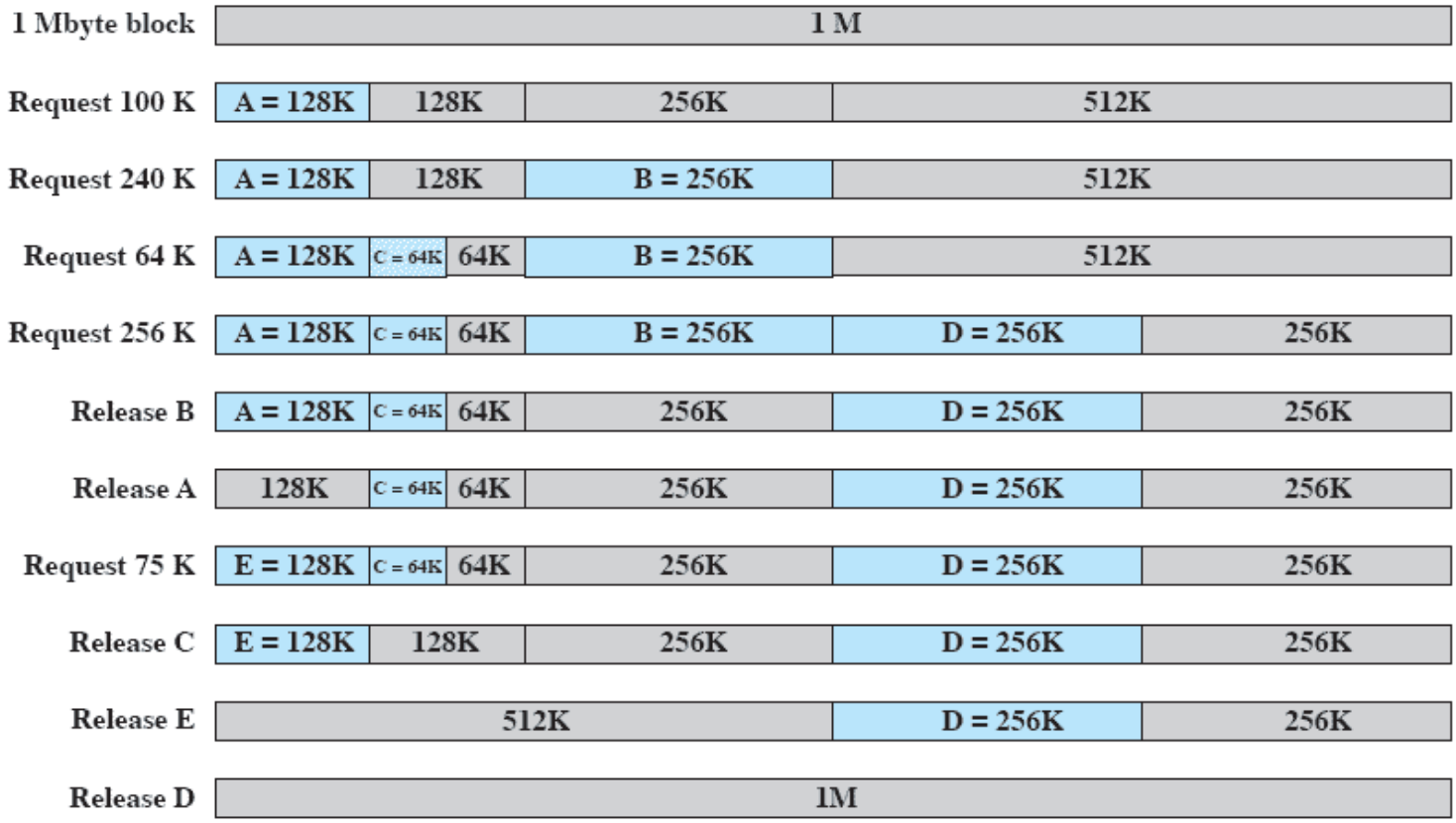


Figure 7.6 Example of Buddy System

Free Representation of Buddy System

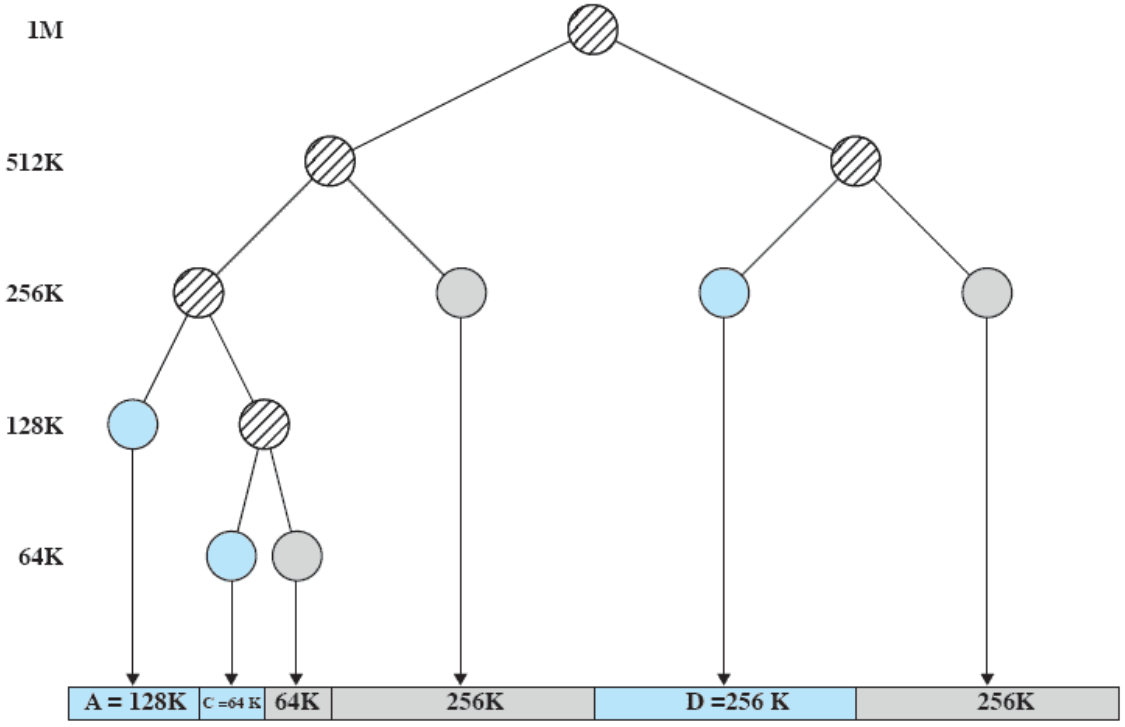


Figure 7.7 Tree Representation of Buddy System

Relocation

- When program is loaded into memory, the actual (absolute) memory locations are determined
- A process may occupy different partitions, which means different absolute memory locations during execution (from swapping)
- Compaction will also cause a program to occupy a different partition, which means different absolute memory locations

Addresses

- Logical
  - Reference to a memory location independent of the current assignment of data to memory
  - Translation must be made to the physical address
- Relative
  - Address expressed as a location relative to some known point
- Physical
  - The absolute address or actual location in main memory

Relocation

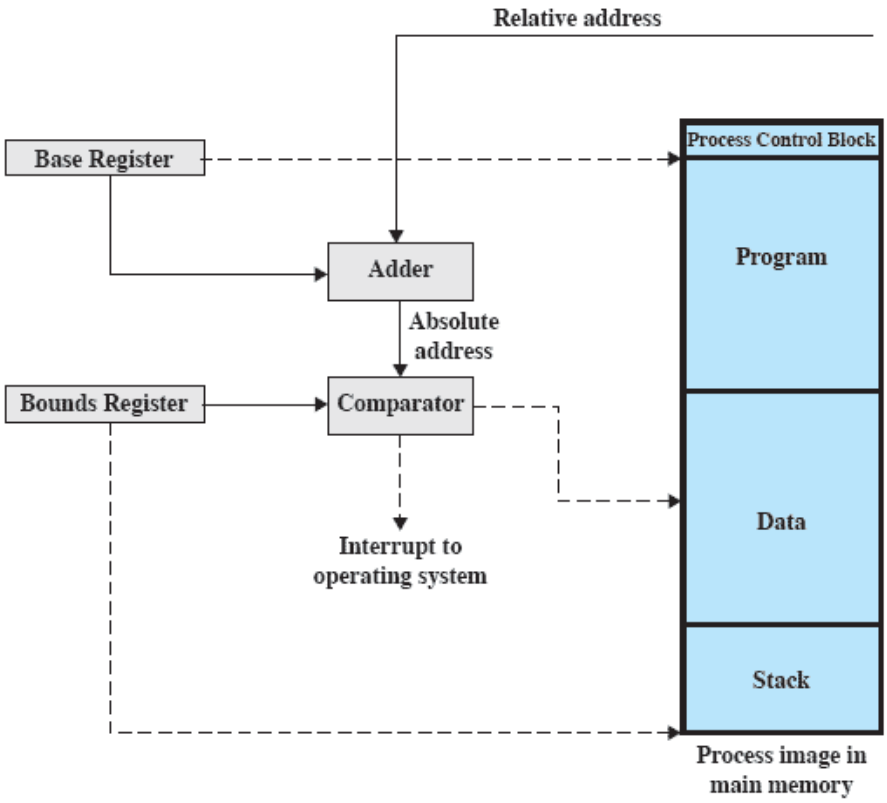


Figure 7.8 Hardware Support for Relocation

When we map out the addresses we have some references in the program image which is local address and some absolute address in local space where the memory resides whichch we need to implement. We need to add a base register where it was loaded and a bounds register which marks its limits. The adder and comparter are turning the local addresses into valid addresses at run time.

Registers Used during Execution

- Base register
  - Starting address for the process
- Bounds register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in

The registers used for house keeping (where the process is and the relative addresses used) are stored in the process control block

Registers Used during Execution

- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system

This is a good concept to remember to know how it works on a high level. The hardware implementation is much more sophistocated. Relocation in the context of fragmetation gets much more complicated.

SO WHAT TO DO ABOUT EXTERNAL FRAGMENTATION?

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
- Base address pointer no longer sufficient

Break your process image into equal sized chunks called pages. Load each page into a frame (chunk of ram) and updates the pointer to it.

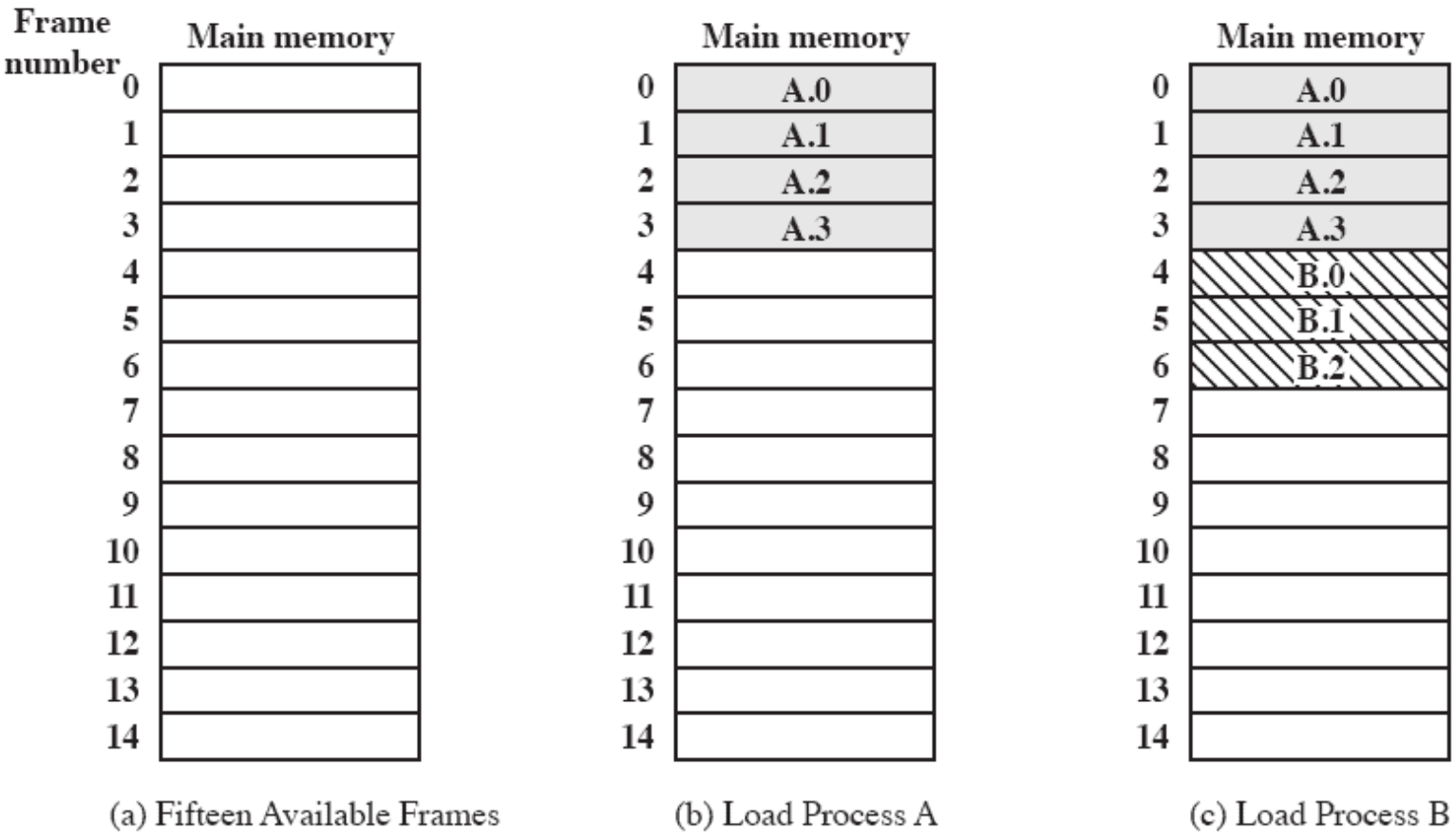
Paging

- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page
- Logical-to-physical is still done by hardware
  - Logical address: (page, offset)
  - Physical address: (frame, offset)

We keep track of the mapping of pages to main memory. We translate the offset of a value in the program image to a address in the associated frame.

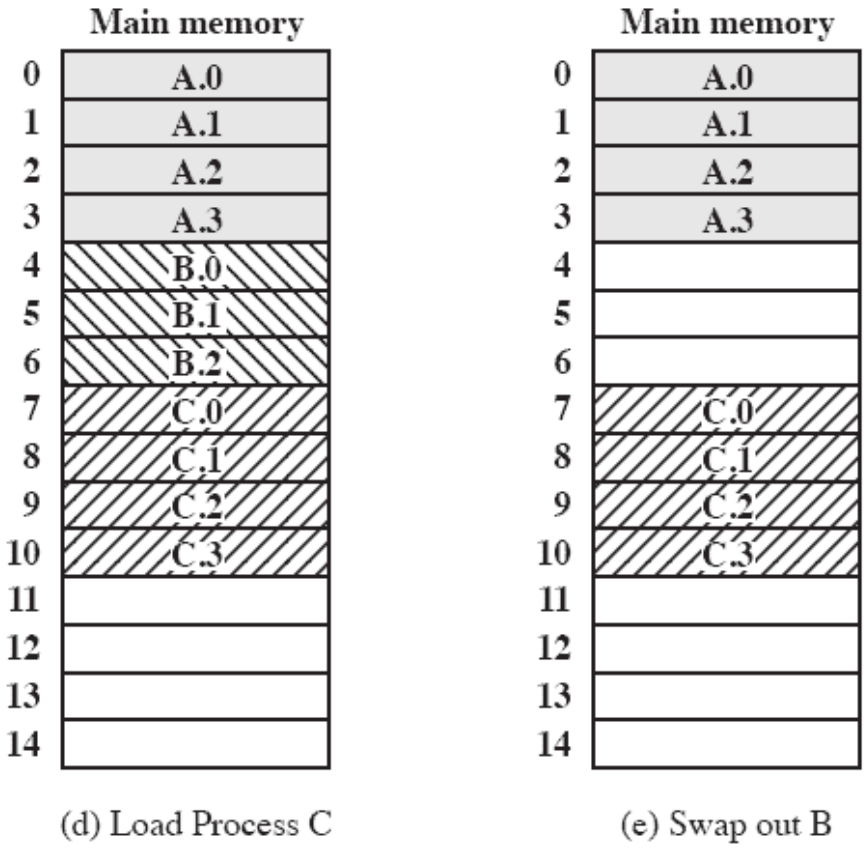
We reserve a chunk of real memory that stores your memory table (mapping). Every time we switch processes we get the page address from the PCB and load that into memory for use. We need to make sure that the memory table doesn't get erased or changed. We can nest paging tables.

Process and Frames



(a) Fifteen Available Frames (b) Load Process A (c) Load Process B

Process and Frames



Can we now allocate 5 frames for Process D?

Figure 7.9 Assignment of Process Pages to Free Frames

Here we can allocate 5 frames since the memory doesnt have to be continuous.

Page Table

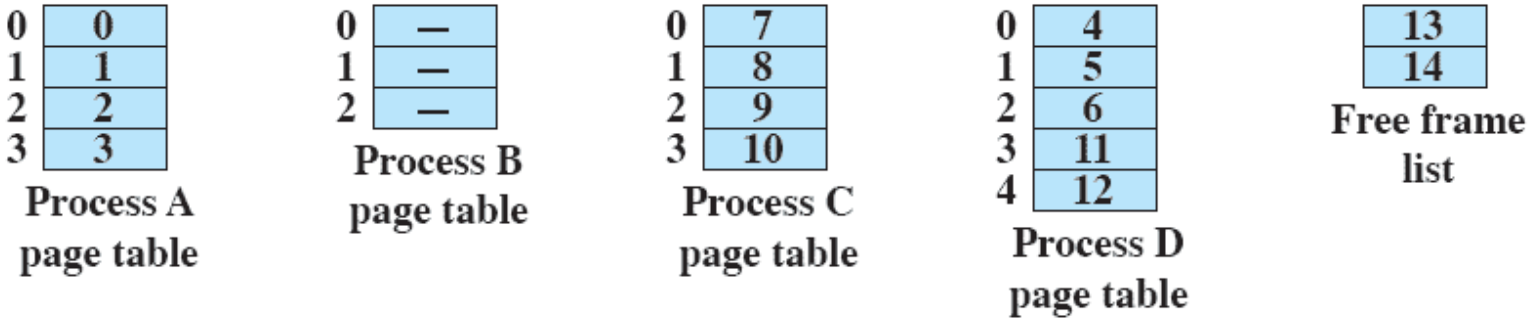


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

Here we see the page mapping where process D is not continuous.

Segmentation

- Program is divided into segments
- All segments of all programs do not have to be of the same length
- There is a maximum segment length
- Addressing consist of two parts: (segment number, offset)
- Since segments are not equal, segmentation is similar to dynamic partitioning
  - But program can have more segments (but only one partition)

Instead of dividing memory into equal size blocks have the block size be variable. We refer to a segment number and an offset in that segment.

Logical Addresses

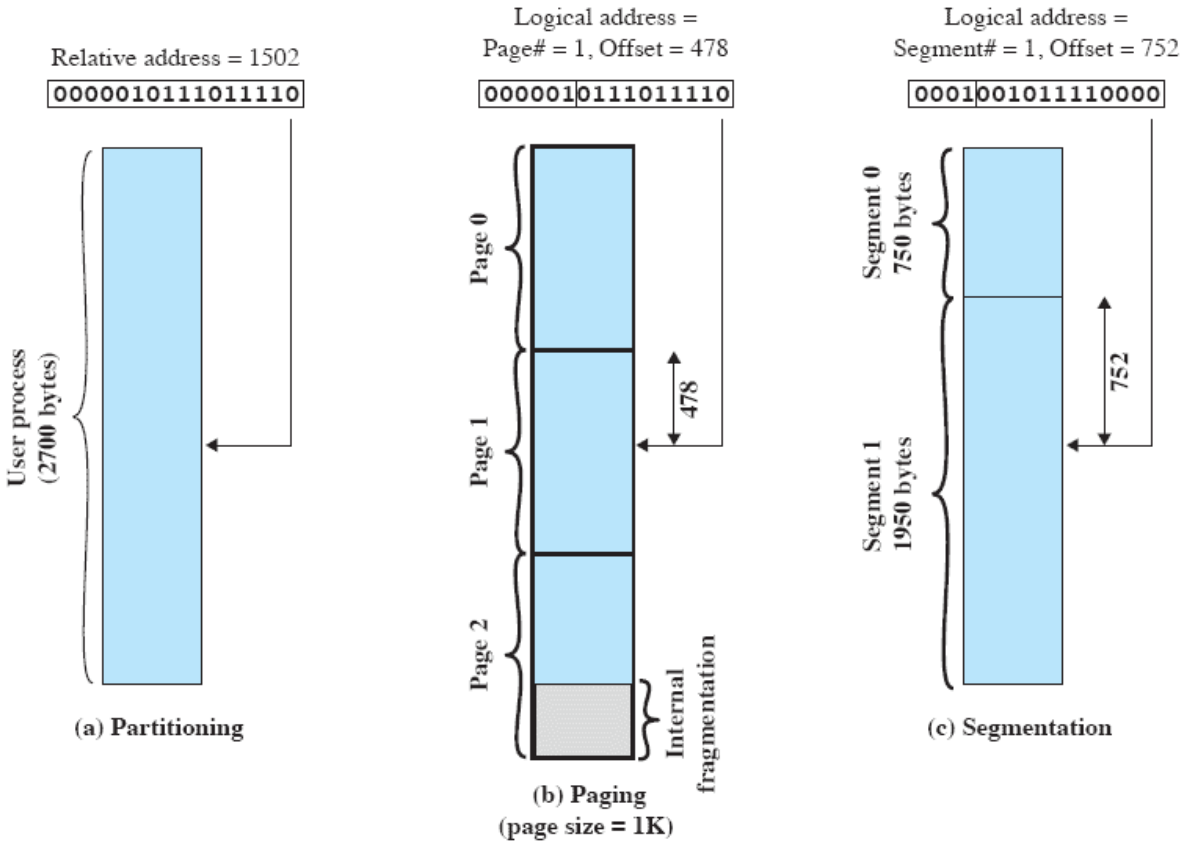
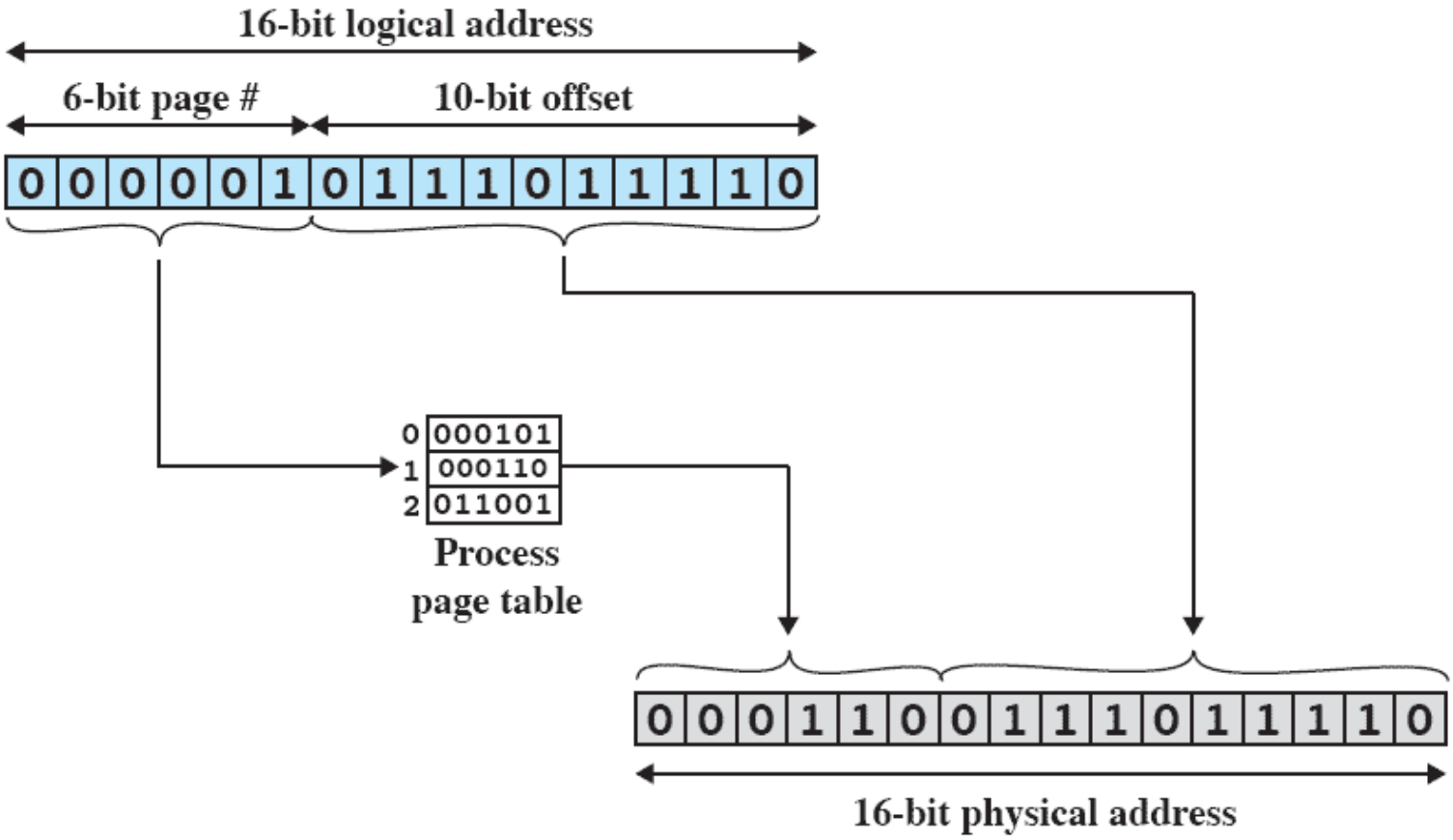


Figure 7.11 Logical Addresses

Paging



(a) Paging

Calculate the most significate bits of the offset and switch them out with the frame. We then copy over the ten bit offset to make a new physical address in the correct segment with the correct offset. The size of the offset determines the size of your page and the page number tells you how many pages you have.

Segmentation

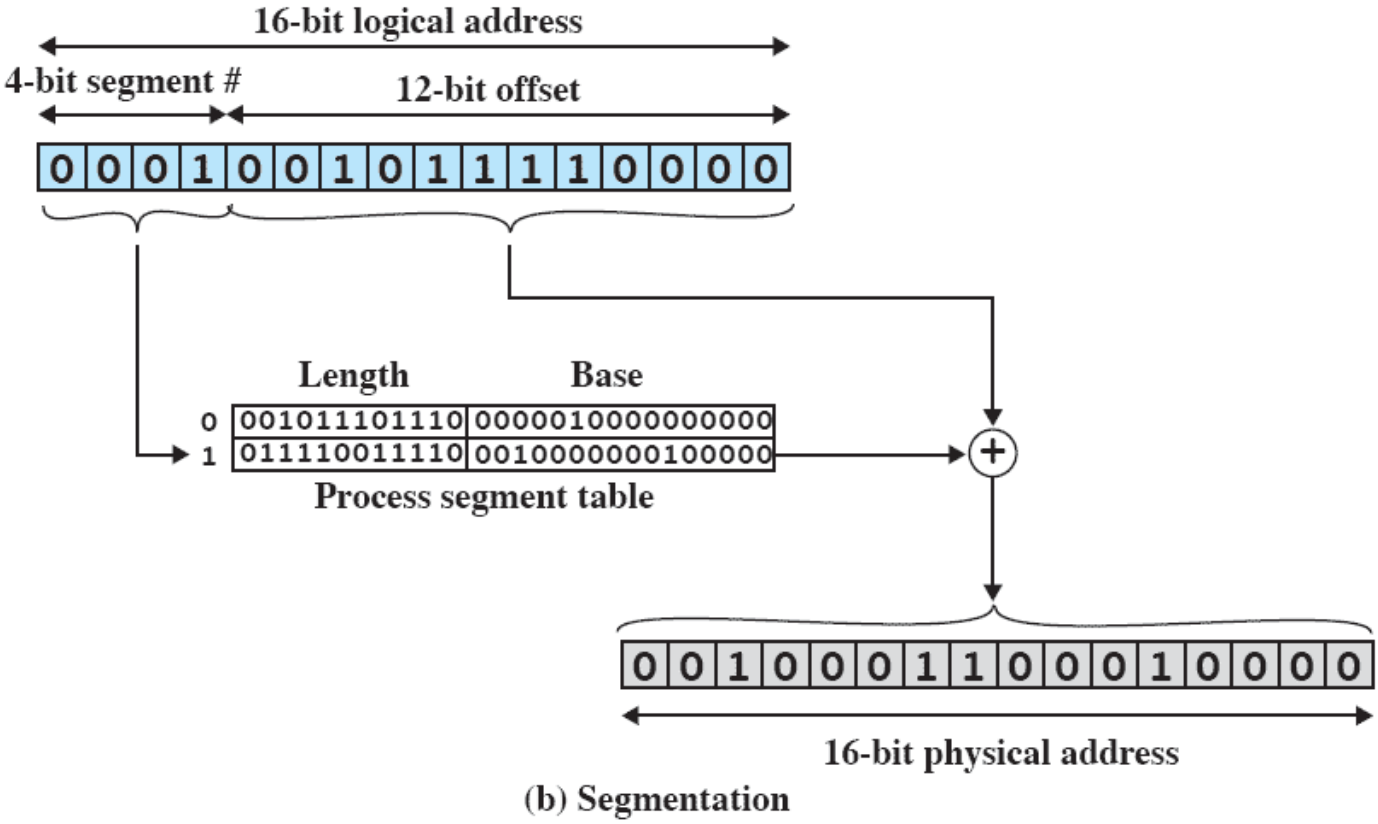
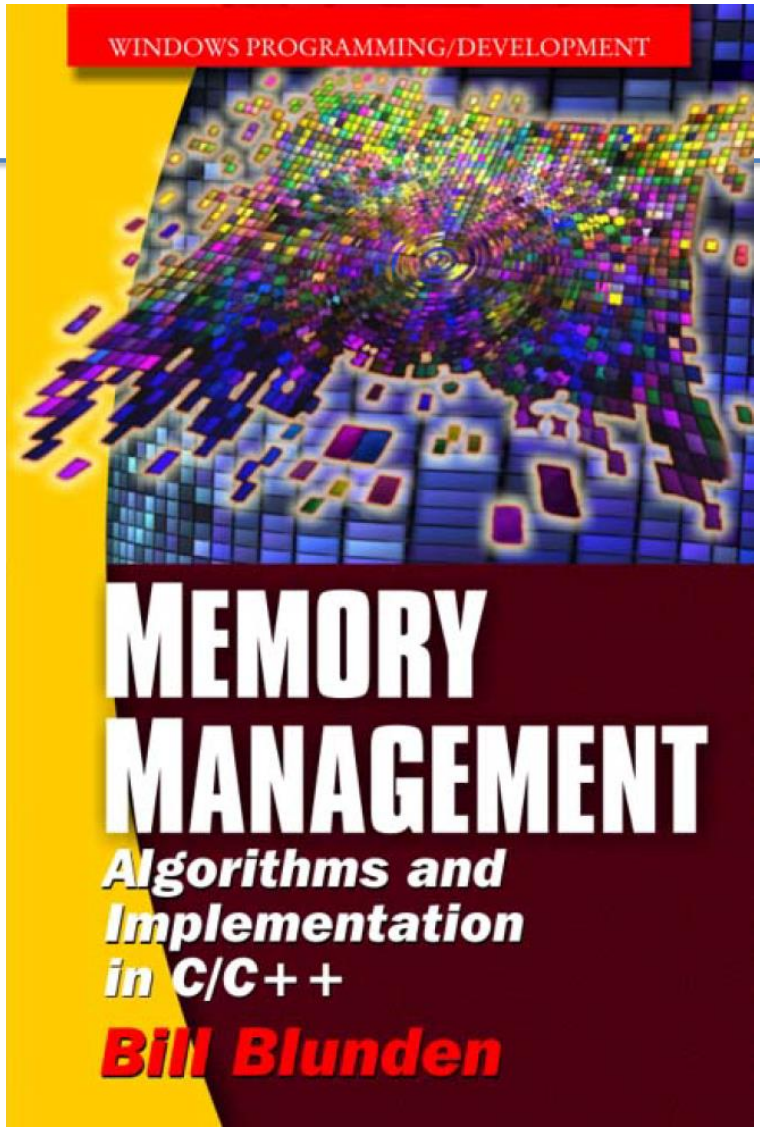


Figure 7.12 Examples of Logical-to-Physical Address Translation

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.


WINDOWS PROGRAMMING/DEVELOPMENT
MEMORY MANAGEMENT
Algorithms and Implementation in C/C++
Bill Blunden