Lecture 6, 7

Michal Vrábel, 27/11/2019

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Memory management requirements

- **Relocation**
  - not possible for the programmer to know in advance which other programs will be resident in main memory at the time of execution,
  - Need for swapping
  - Need to relocate the process to a different area of memory
  - the processor hardware and operating system software must be able to **translate the memory references found in the code** of the program into actual physical memory addresses, reflecting the current location of the program in main memory
- **Protection**
  - Process should be protected against unwanted interference by other processes
  - All memory references generated by a process must be checked at run time to ensure that they refer only to the memory space allocated to that process.
  - Processor must be able to abort such instructions at the point of execution.
- **Sharing**
  - Processes that are cooperating on some task may need to share access to the same data structure.
- **Logical Organization** – organization into modules / segments
- **Physical Organization** – information movement between primary and secondary

Reallocation

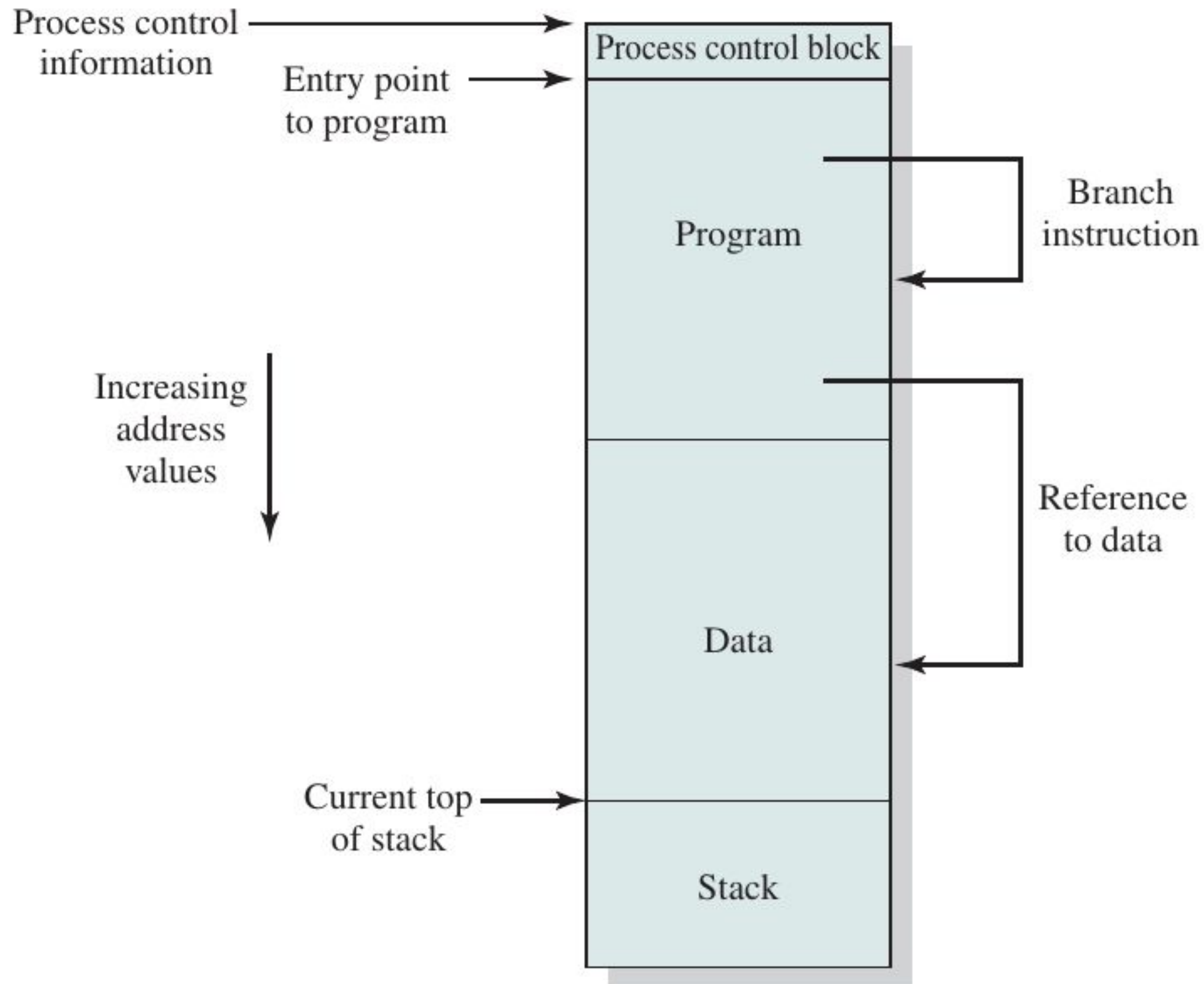
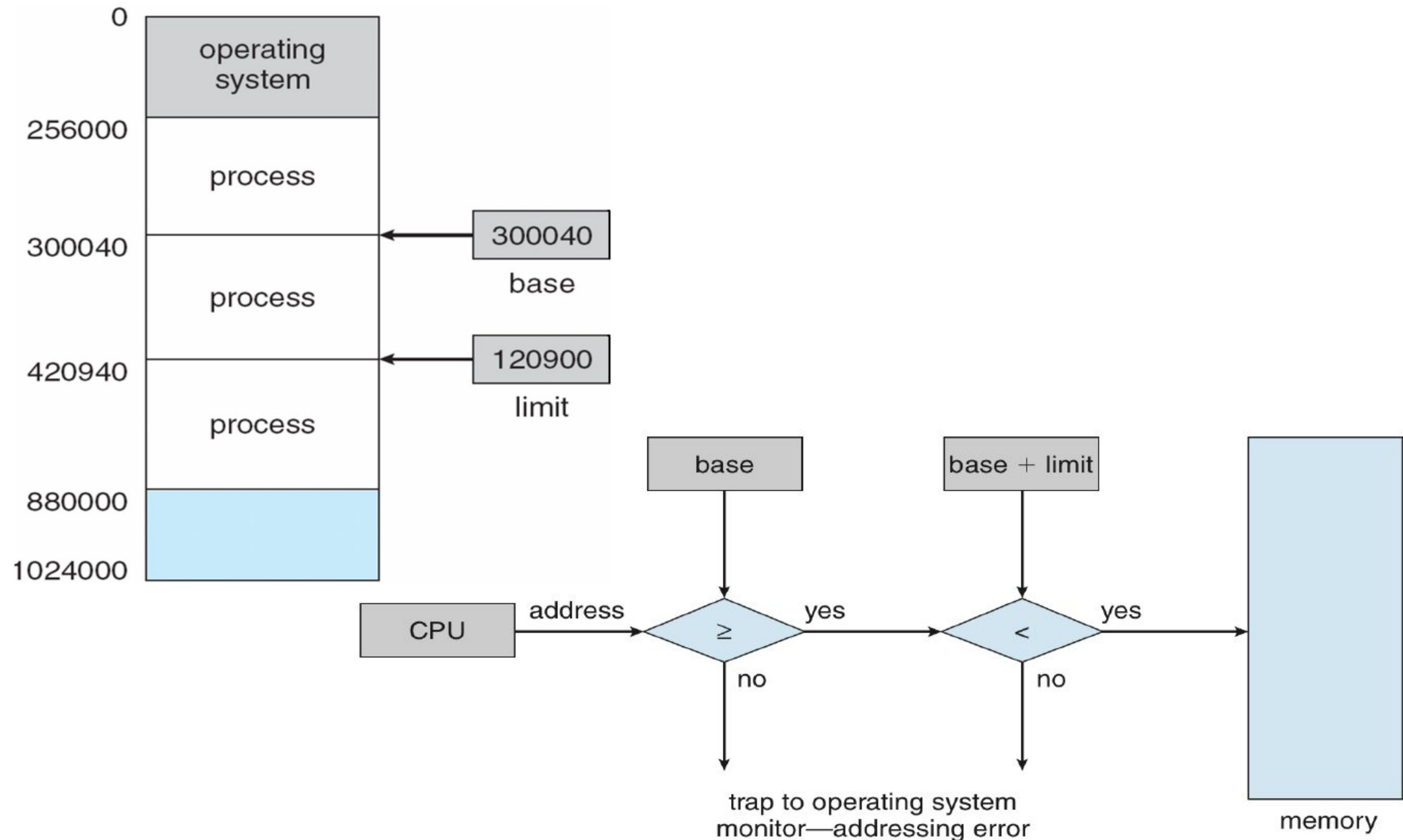


Figure 7.1 **Addressing Requirements for a Process**

Reallocation - Base and Limit Registers



Address Binding

- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

Table 7.3 Address Binding

(a) Loader

Binding Time	Function
Programming time	All actual physical addresses are directly specified by the programmer in the program itself.
Compile or assembly time	The program contains symbolic address references, and these are converted to actual physical addresses by the compiler or assembler.
Load time	The compiler or assembler produces relative addresses. The loader translates these to absolute addresses at the time of program loading.
Run time	The loaded program retains relative addresses. These are converted dynamically to absolute addresses by processor hardware.

(b) Linker

Linkage Time	Function
Programming time	No external program or data references are allowed. The programmer must place into the program the source code for all subprograms that are referenced.
Compile or assembly time	The assembler must fetch the source code of every subroutine that is referenced and assemble them as a unit.
Load module creation	All object modules have been assembled using relative addresses. These modules are linked together and all references are restated relative to the origin of the final load module.
Load time	External references are not resolved until the load module is to be loaded into main memory. At that time, referenced dynamic link modules are appended to the load module, and the entire package is loaded into main or virtual memory.
Run time	External references are not resolved until the external call is executed by the processor. At that time, the process is interrupted and the desired module is linked to the calling program.

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit

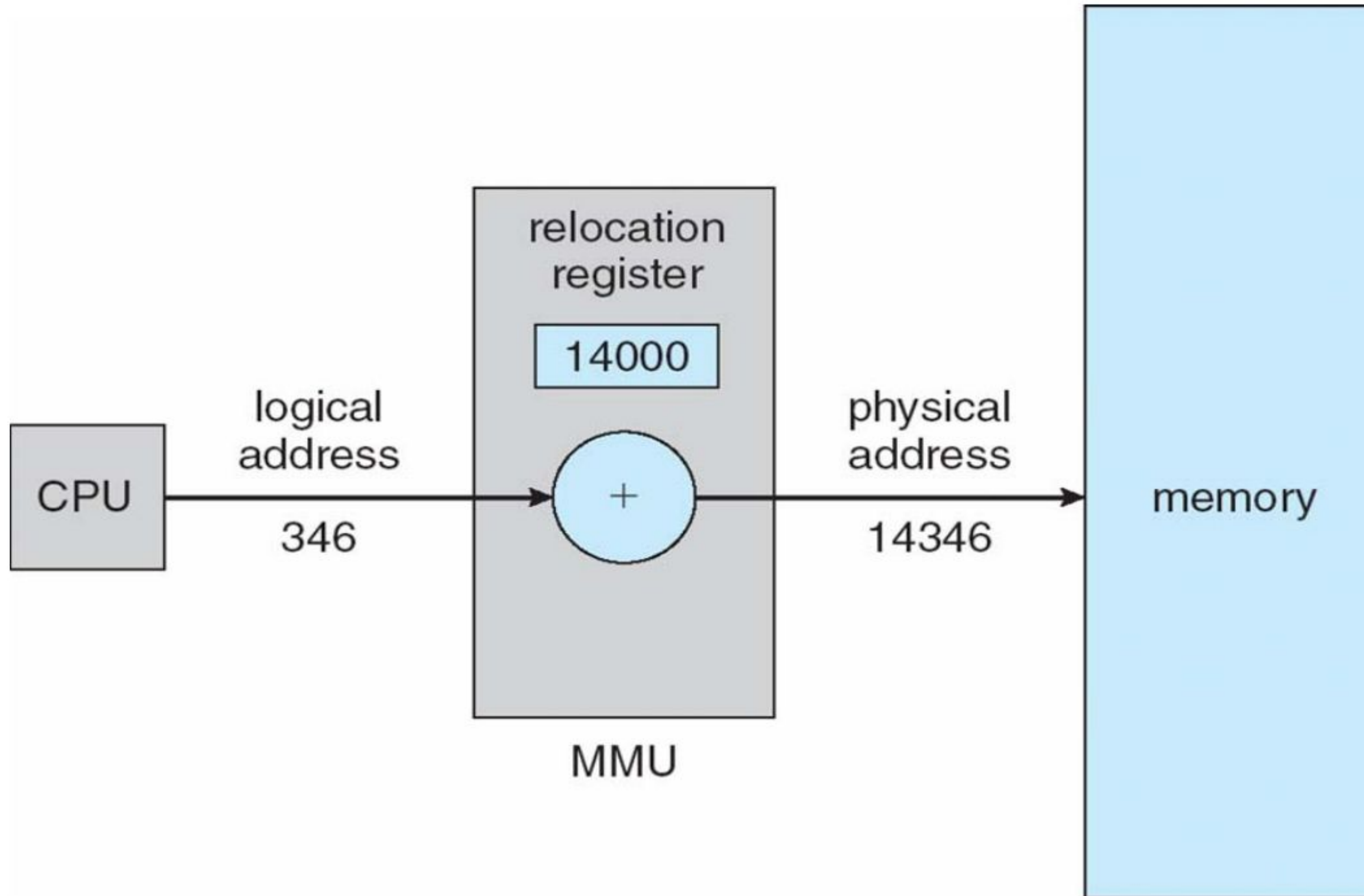
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Relative address
– particular example of a logical address
– address expressed as a location relative to some known point

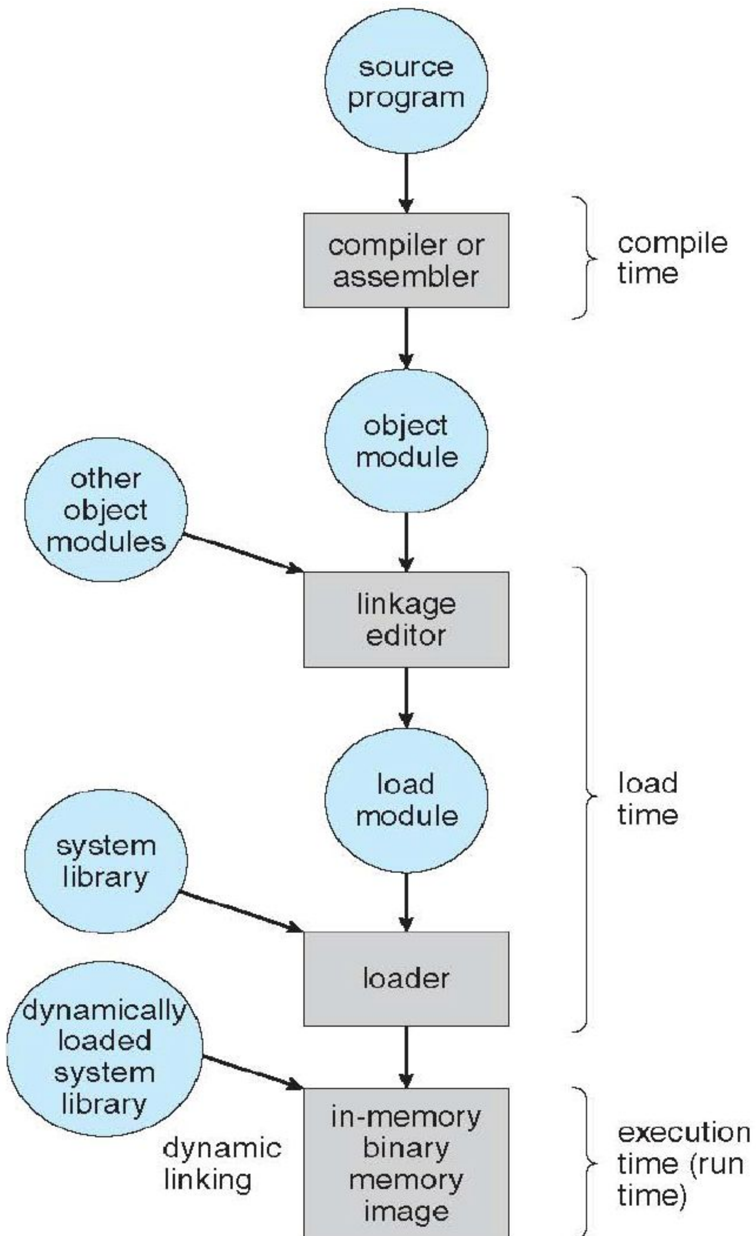
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register



Multistep Processing of a User Program



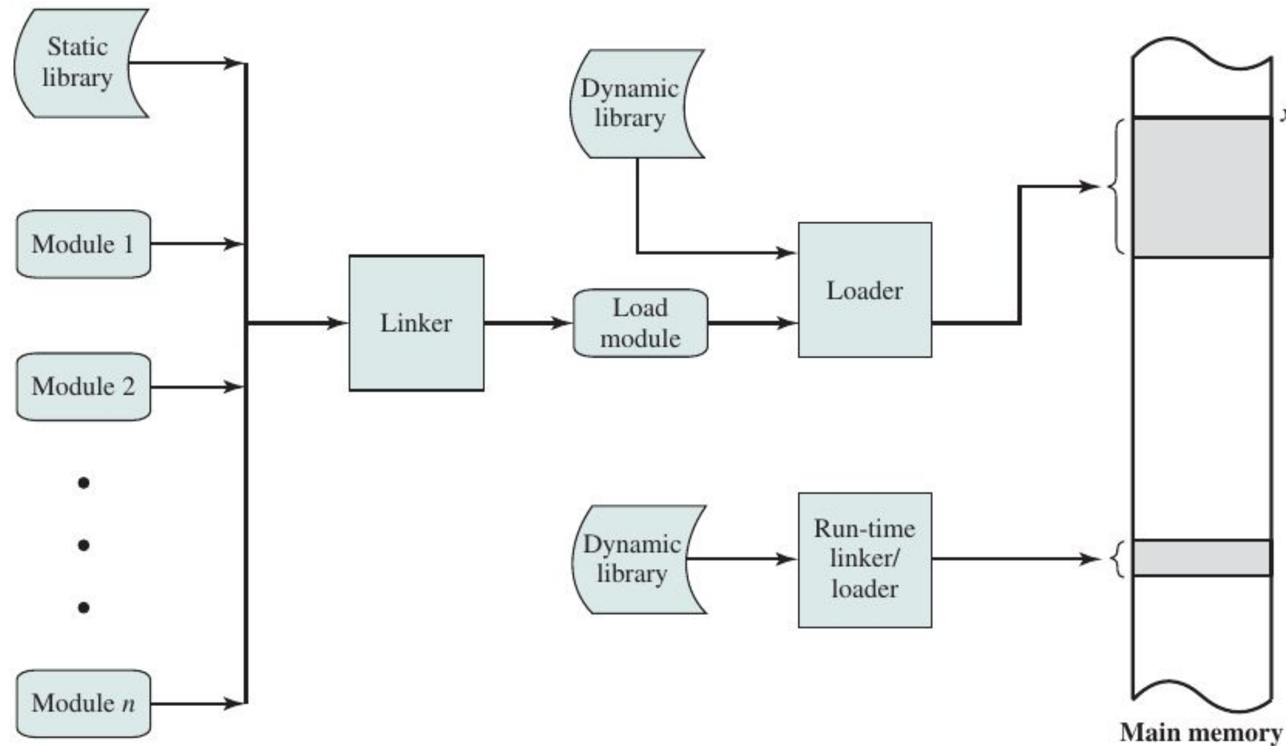
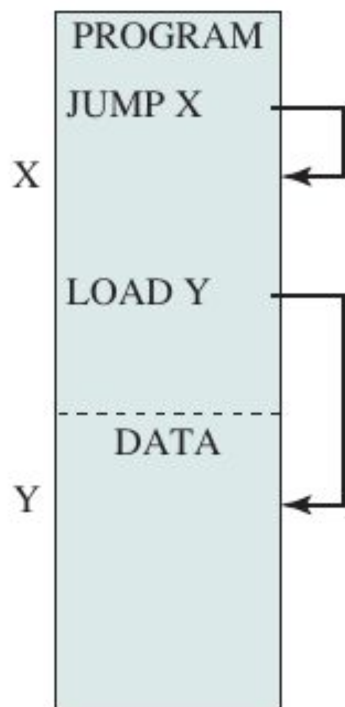


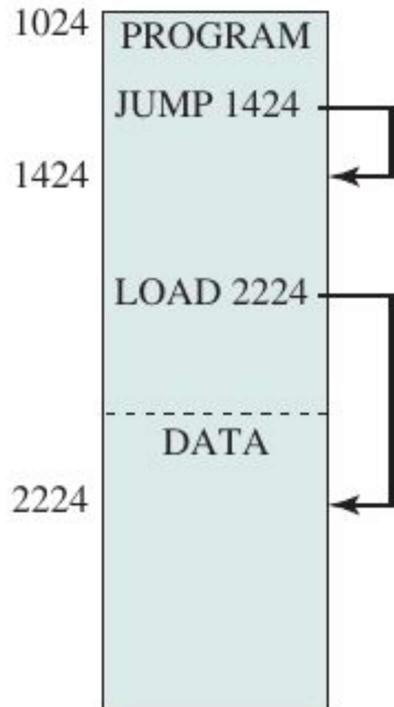
Figure 7.16 A Linking and Loading Scenario

Loading

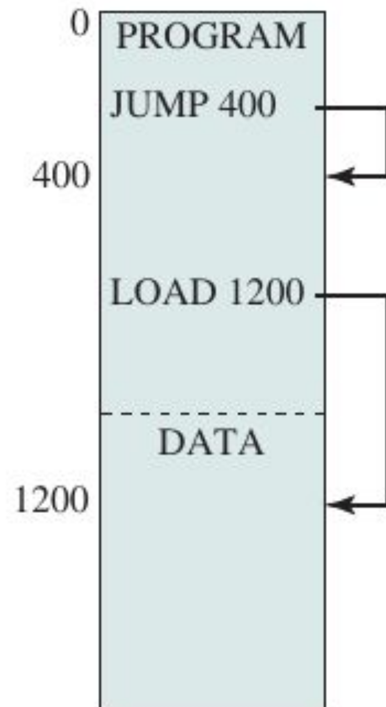
- the loader places the load module in main memory starting at location x.
- Absolute loading
  - An absolute loader requires that a given load module always be loaded into the same location in main memory
  - Assignment of specific address values to memory references within a program can be done either by the programmer or at compile or assembly time
  - reference to an instruction or item of data is initially represented by a symbol
- Relocatable loading
  - Decision at load time
  - Assembler / compiler produces relative addresses to known point (start of the program)
- Dynamic run-time loading



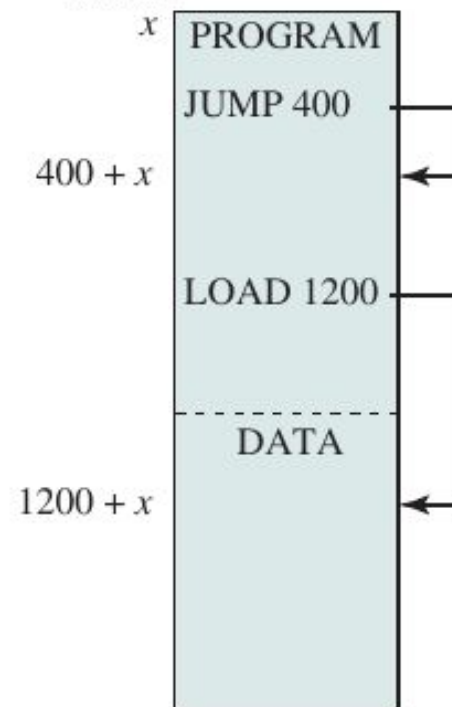
(a) Object module



(b) Absolute load module



(c) Relative load module



(d) Relative load module loaded into main memory starting at location x

Figure 7.17 Absolute and Relocatable Load Modules

Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Linking

- Linker takes as input a collection of object modules and produce a load module, consisting of an integrated set of program and data modules, to be passed to the loader.
- In each object module, there may be address references to locations in other modules.
  - Each such reference can only be expressed symbolically in an unlinked object module.
- The linker creates a single load module that is the contiguous joining of all of the object modules.
- Each intramodule reference must be changed from a symbolic address to a reference to a location within the overall load module.

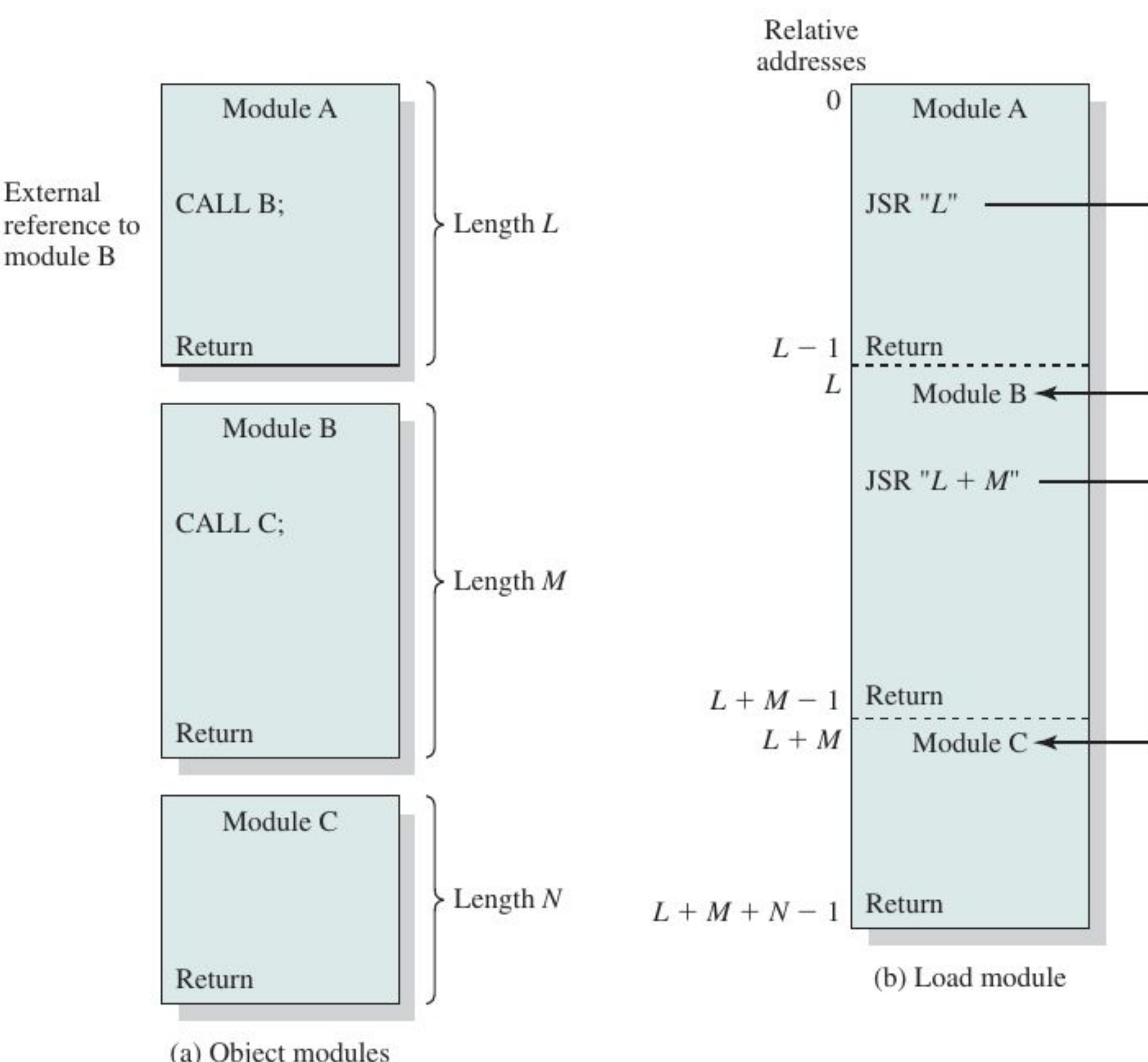


Figure 7.18 The Linking Function

Module A contains a procedure invocation of module B.
When these modules are combined in the load module, this symbolic reference to module B is changed to a specific reference to the location of the entry point of B within the load module.

Relocatable load module - Each compiled or assembled object module is created with references relative to the beginning of the object module. All of these modules are put together into a single relocatable load module with all references relative to the origin of the load module. This module can be used as input for relocatable loading or dynamic run-time loading.

A linker that produces a relocatable load module is often referred to as a **linkage editor**

Dynamic Linking

- Static linking – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Memory management techniques

Table 7.2 Memory Management Techniques

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.

- Memory management - bring processes into main memory for execution by the processor.
- In almost all modern multiprogramming systems, this involves a sophisticated scheme known as **virtual memory**.
- Virtual memory is, in turn, based on the use of one or both of two basic techniques: **segmentation** and **paging**.
- Partitioning – the simplest scheme for managing available memory

- **Fixed Partitioning**
  - Memory partitioned into regions with fixed boundaries
- Dynamic Partitioning
  - The partitions are of variable length and number

Fixed Partitioning

- Partition sizes – equal, un-equal
- Any process whose size is less than or equal to the partition size can be loaded into any available partition
- If all partitions are full and no process is in the Ready or Running state, the operating system can swap a process out of any of the partitions and load in another process
- Difficulties
  - Size of a program - If too big to fit a partition – overlays – only portion of the program at any one time
  - memory utilization is extremely inefficient – program occupies entire partition - **internal fragmentation**
- Placement algorithm
  - Equally sized – simple
  - Unequally sized – smallest/optimal part., smallest available

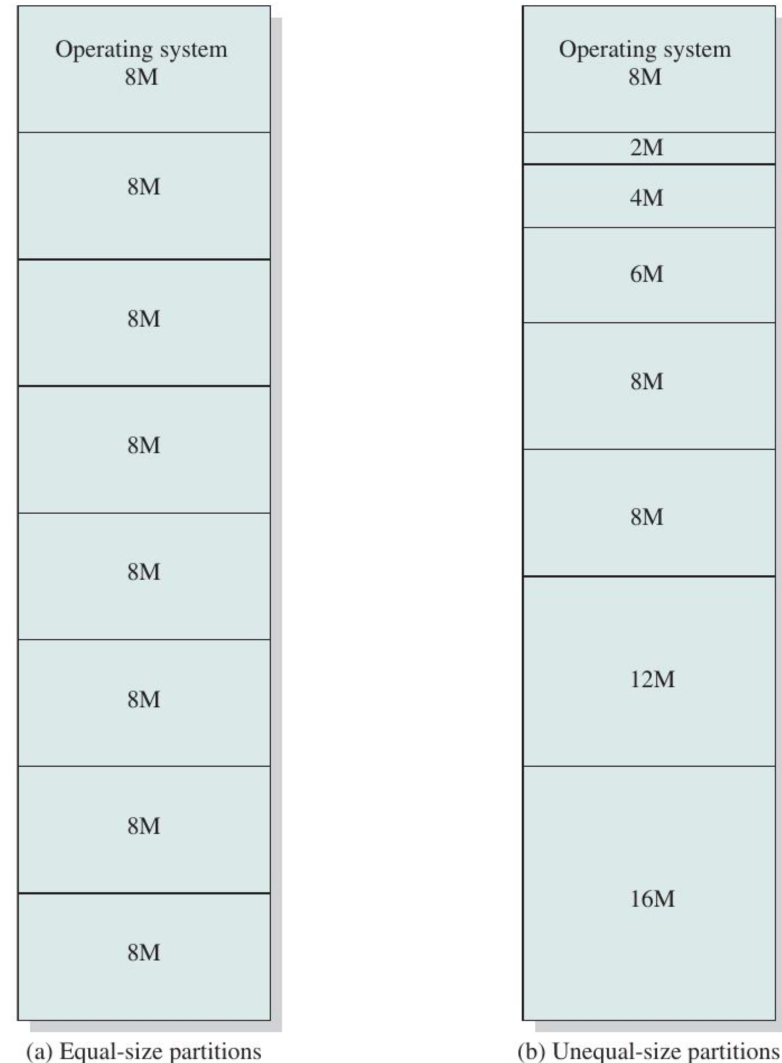


Figure 7.2 **Example of Fixed Partitioning of a 64-Mbyte Memory**

Fixed Partitioning – memory assignment

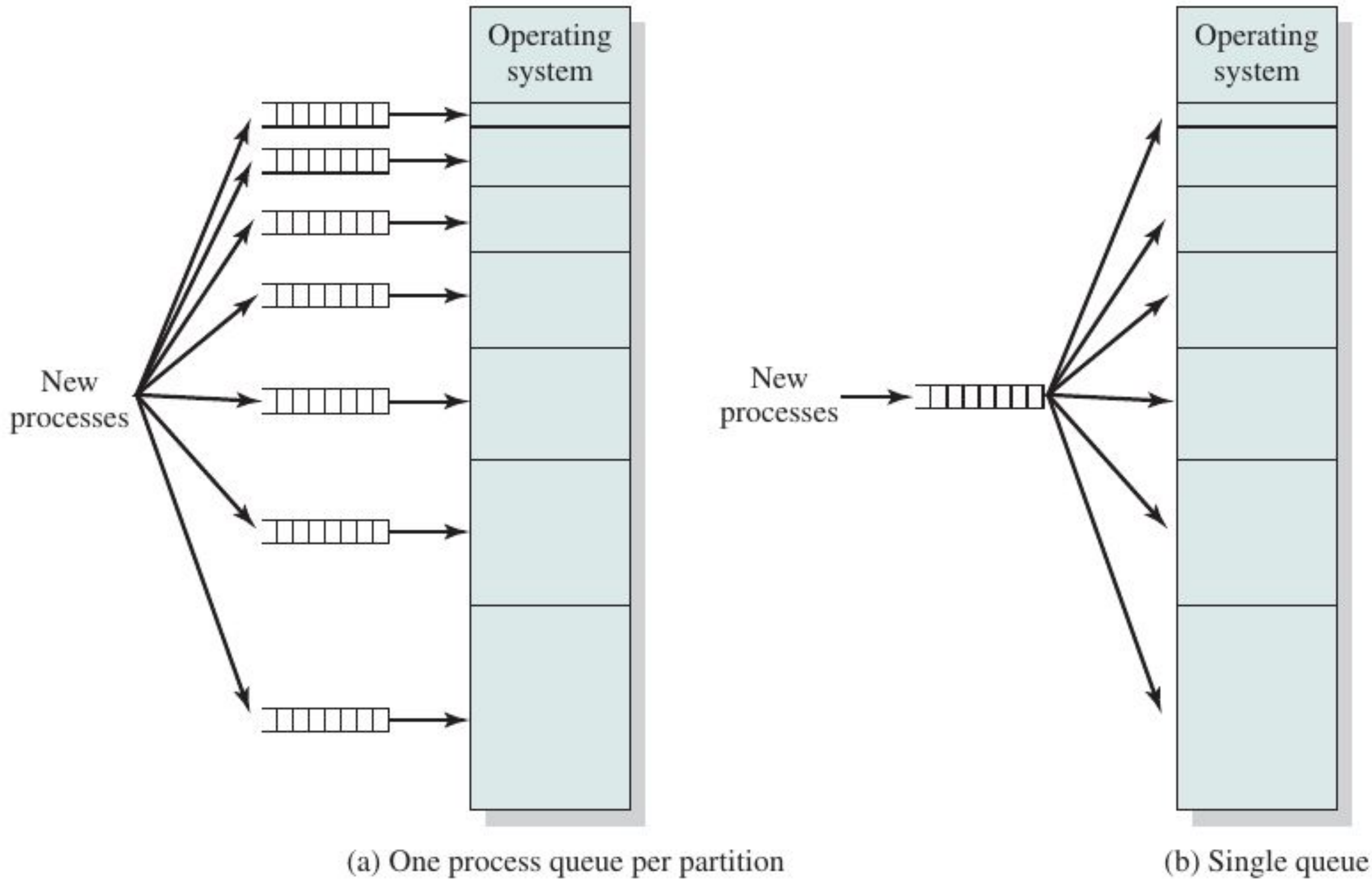


Figure 7.3 Memory Assignment for Fixed Partitioning

Dynamic partitioning

- the partitions are of variable length and number
- process is brought into main memory, it is allocated **exactly as much memory as it requires** and no more.
- As time goes on, memory becomes more and more fragmented, and memory utilization declines - **external fragmentation**
- technique for overcoming external fragmentation is **compaction**
  - time-consuming procedure and wasteful of processor time
  - requires **dynamic relocation**
    - It must be possible to move a program from one region to another in main memory without invalidating the memory references in the program
- **Placement algorithm** – best-fit, first-fit, next-fit
- **Replacement algorithm**

(a)
Operating system 8M
56M

(b)
Operating system
Process 1 20M
36M

(c)
Operating system
Process 1 20M
Process 2 14M
22M

(d)
Operating system
Process 1 20M
Process 2 14M
Process 3 18M
4M

(e)
Operating system
Process 1 20M
14M
Process 3 18M
4M

(f)
Operating system
Process 1 20M
Process 4 8M
6M
Process 3 18M
4M

(g)
Operating system
20M
Process 4 8M
6M
Process 3 18M
4M

(h)
Operating system
Process 2 14M
6M
Process 4 8M
6M
Process 3 18M
4M

Figure 7.4 The Effect of Dynamic Partitioning

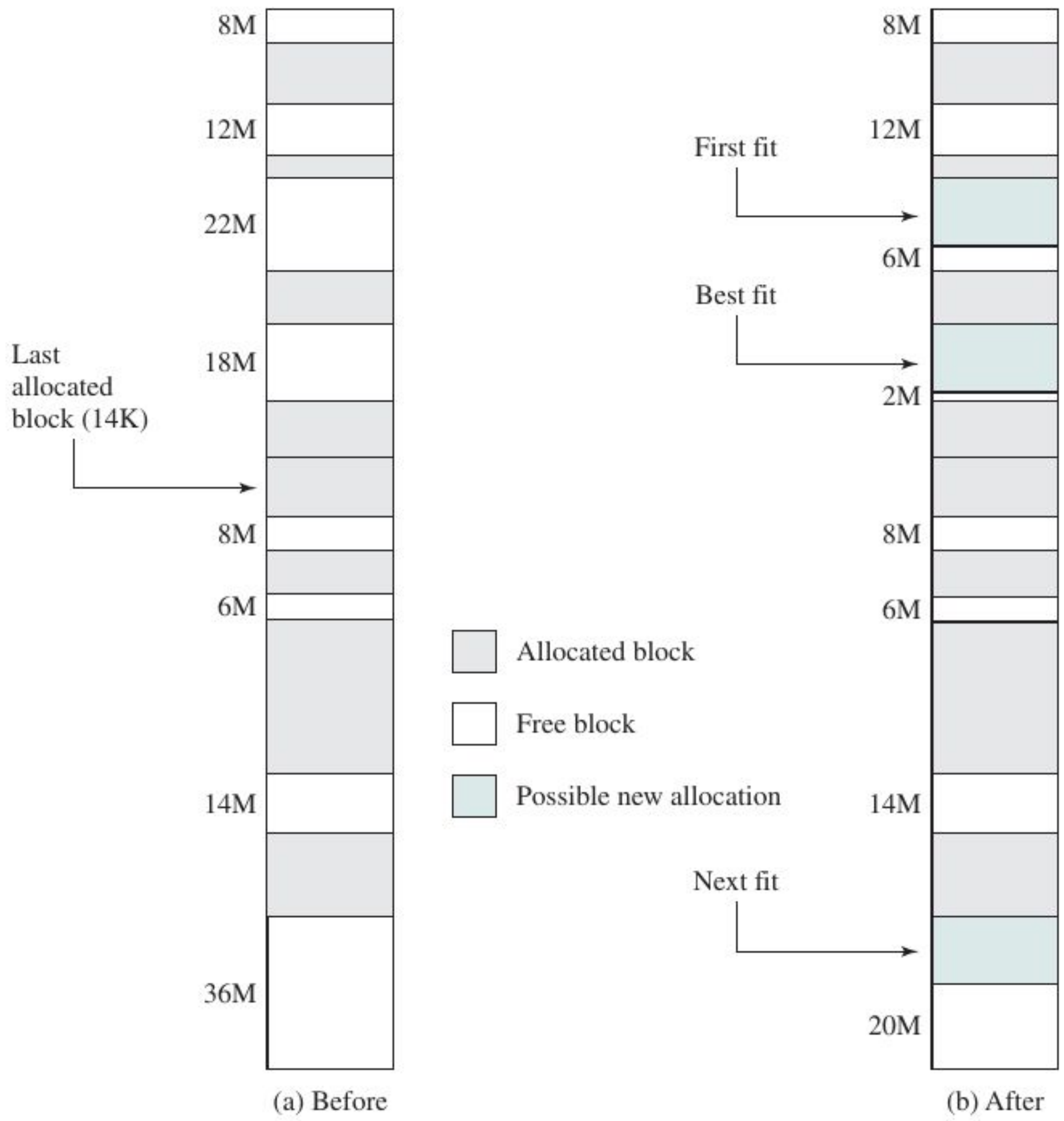


Figure 7.5 **Example Memory Configuration before and after Allocation of 16-Mbyte Block**

Partitioning – buddy system

- Compromise between dynamic and fixed partitioning

In a buddy system, memory blocks are available of size 2^K words, $L \leqslant K \leqslant U$, where

2^L = smallest size block that is allocated

2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation

- Starts with a partition spanning whole memory, then split until size is sufficient
- Maintains list of holes
  - Hole removed by splitting into two *buddies*
  - Unallocated buddies are removed and coalesced

Partitioning – buddy system

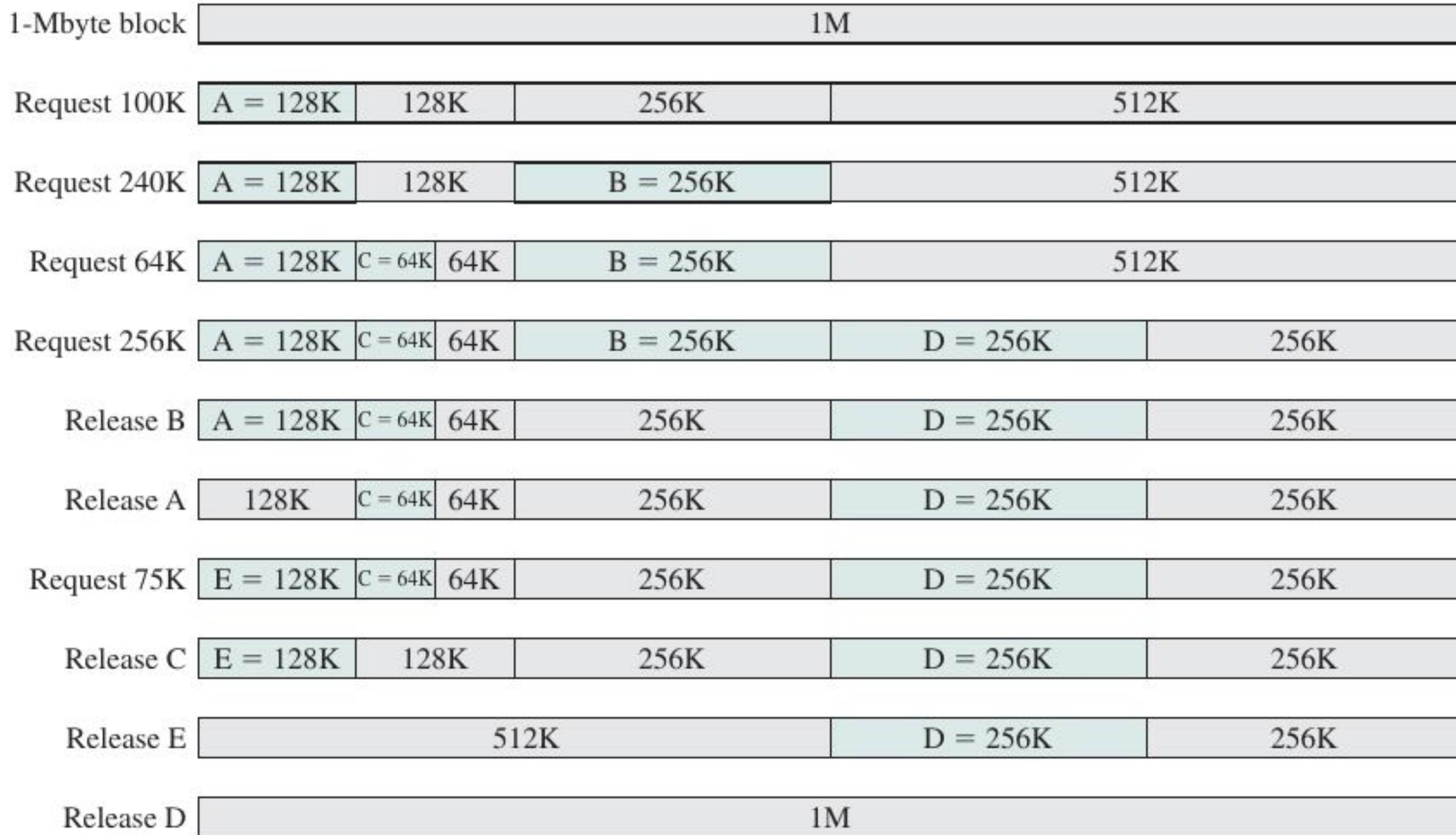


Figure 7.6 Example of Buddy System

Partitioning – buddy system

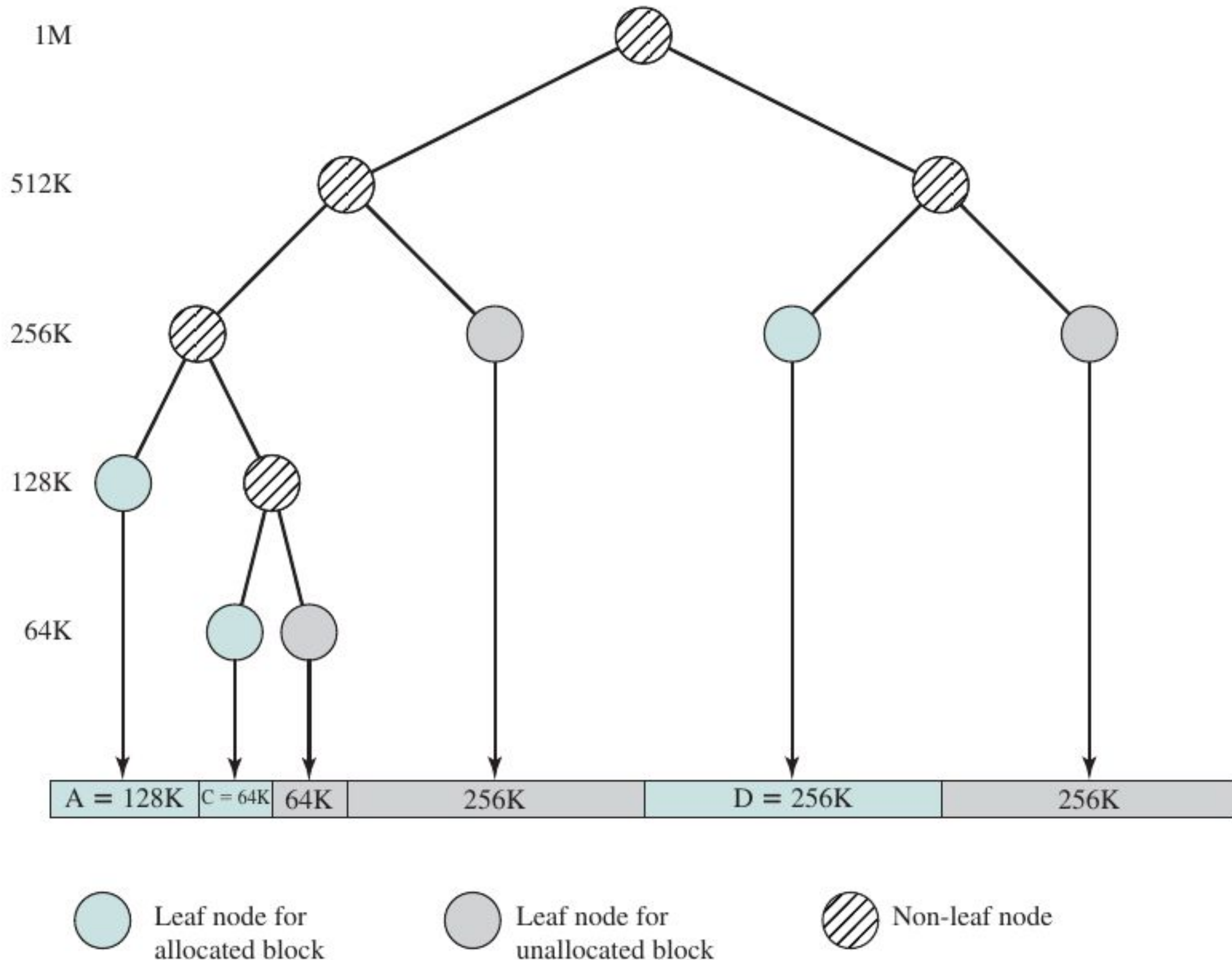


Figure 7.7 **Tree Representation of Buddy System**

Relocation

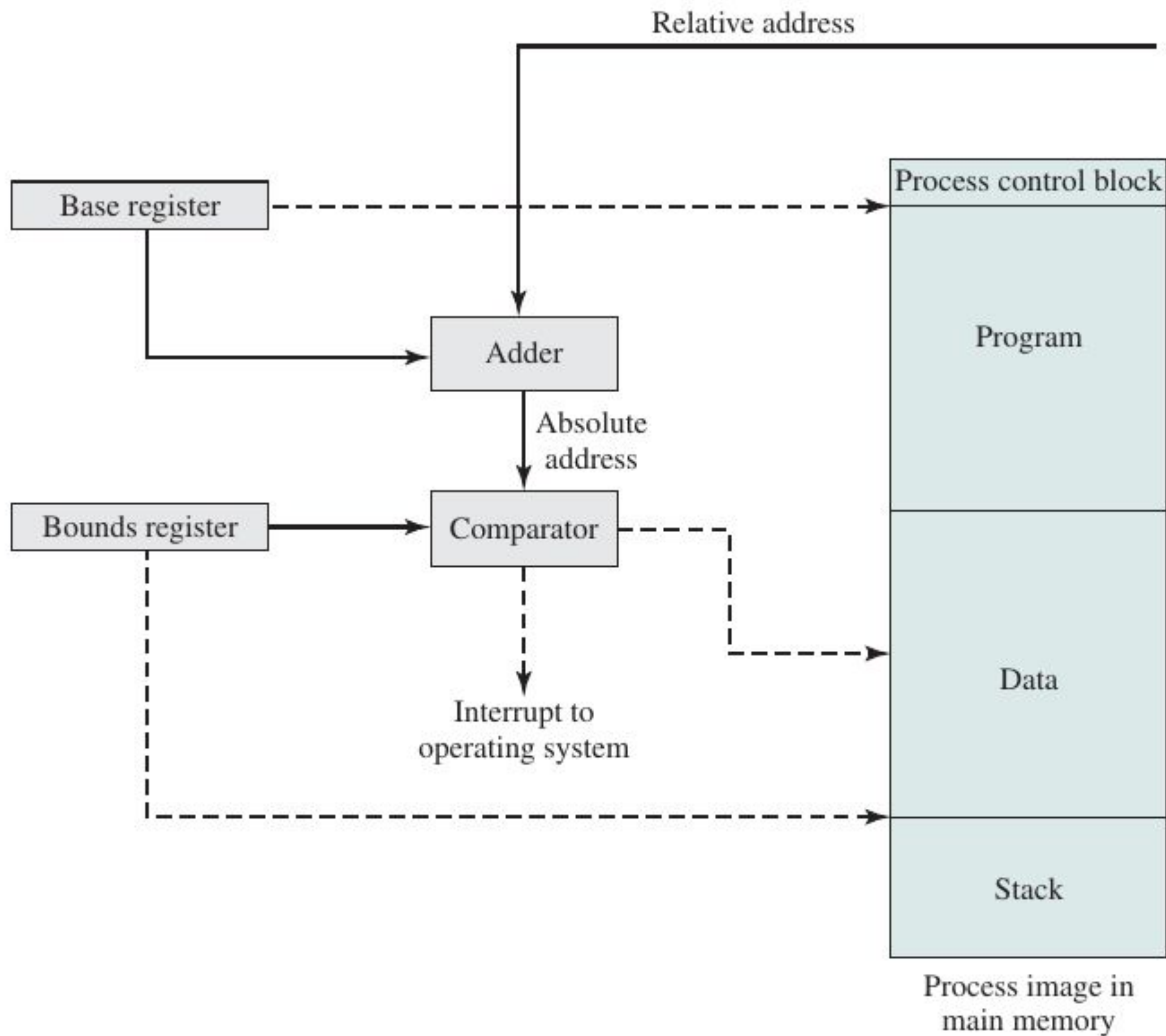


Figure 7.8 **Hardware Support for Relocation**

Paging

- Process’s address space divided into fixed-size chunks – pages
- Frames – available chunks of memory
- No external fragmentation
- Small internal fragmentation – the last page
- List of free frames is mantained by OS
- **Page table** maintained by OS for each process – base address register is not sufficient
- Logical-to-physical translation is done by processor hardware – it must access page table of the current process
- Transparent to a programmer

Paging

Page	Process A page table
0	0
1	1
2	2
3	3

Page	Process B page table
0	—
1	—
2	—

Page	Process C page table
0	7
1	8
2	9
3	10

Page	Process D page table
0	4
1	5
2	6
3	11
4	12

Free frame list
13
14

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

Frame number	(a) Fifteen available frames	(b) Load process A	(c) Load process B	(d) Load process C	(e) Swap out B	(f) Load process D
0		A.0	A.0	A.0	A.0	A.0
1		A.1	A.1	A.1	A.1	A.1
2		A.2	A.2	A.2	A.2	A.2
3		A.3	A.3	A.3	A.3	A.3
4			B.0	B.0		D.0
5			B.1	B.1		D.1
6			B.2	B.2		D.2
7				C.0	C.0	C.0
8				C.1	C.1	C.1
9				C.2	C.2	C.2
10				C.3	C.3	C.3
11						D.3
12						D.4
13						
14						

(Column headings in each diagram: Main memory)

Figure 7.9 Assignment of Process to Free Frames

Paging and segmentation – logical addres

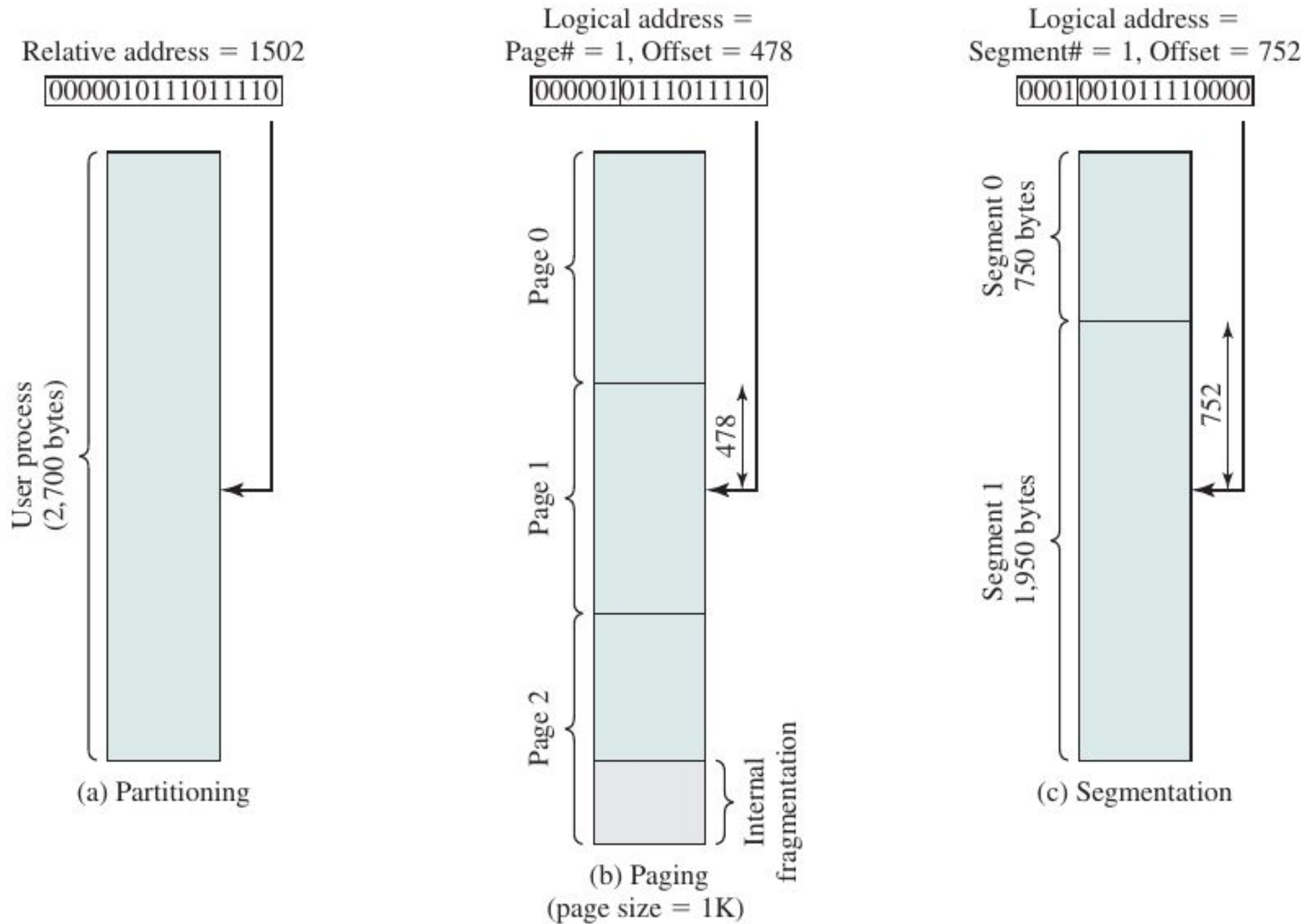


Figure 7.11 **Logical Addresses**

Paging and segmentation

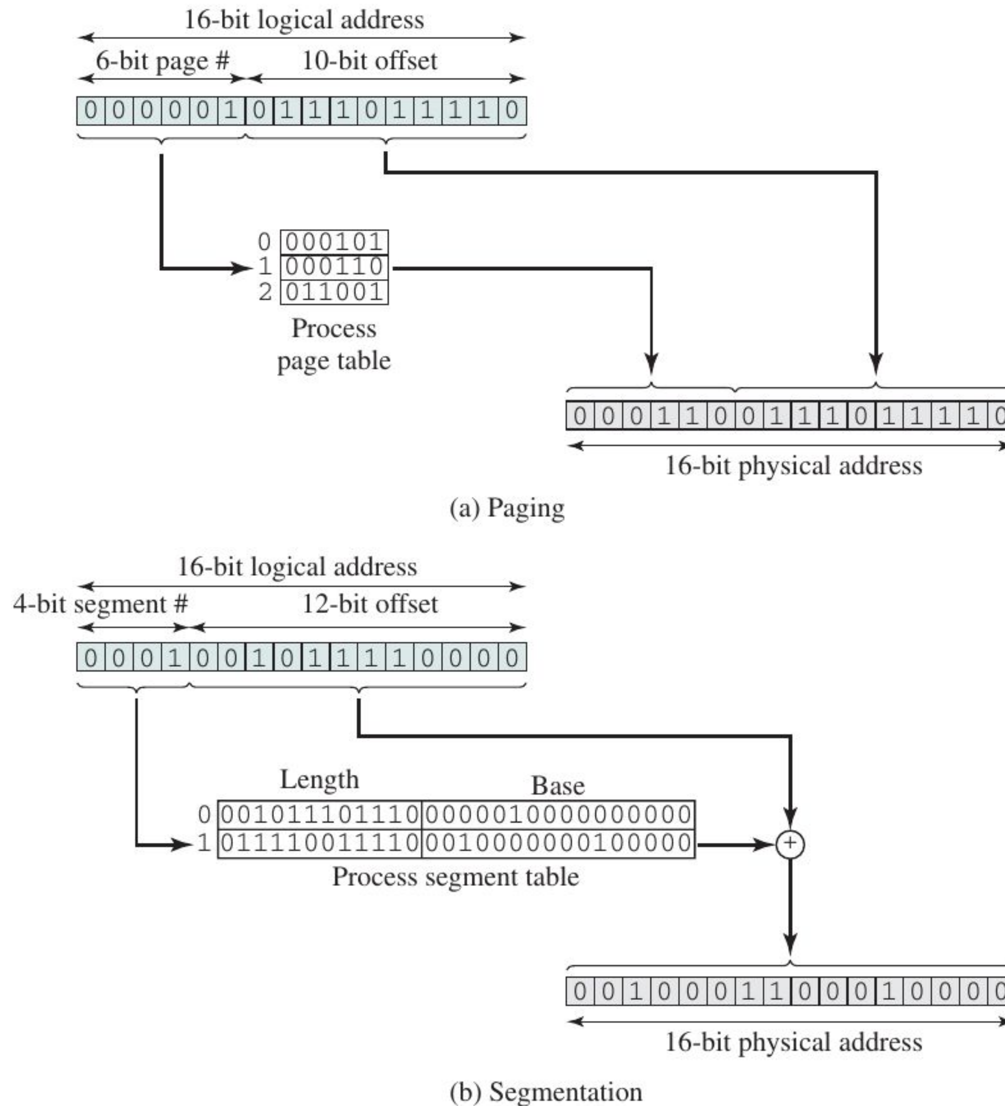


Figure 7.12 Examples of Logical-to-Physical Address Translation