Operating Systems

Chapter 7 Memory Management(内存管理)

A few words

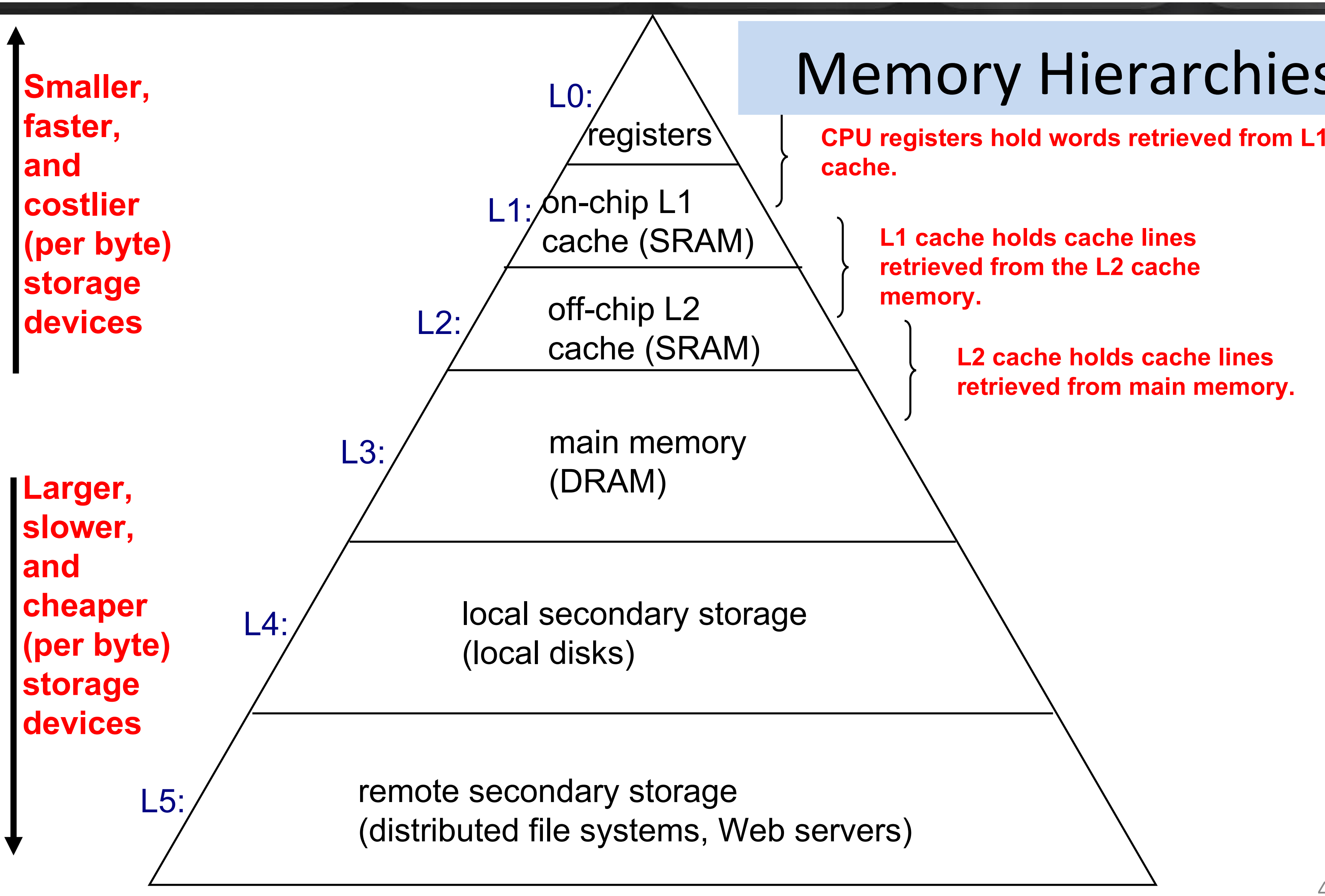
- 本章介绍了内存管理的要求，思想
- 重定位，分页和分段三个基本概念
- 但实用内存管理办法是第八章的虚拟内存

Agenda

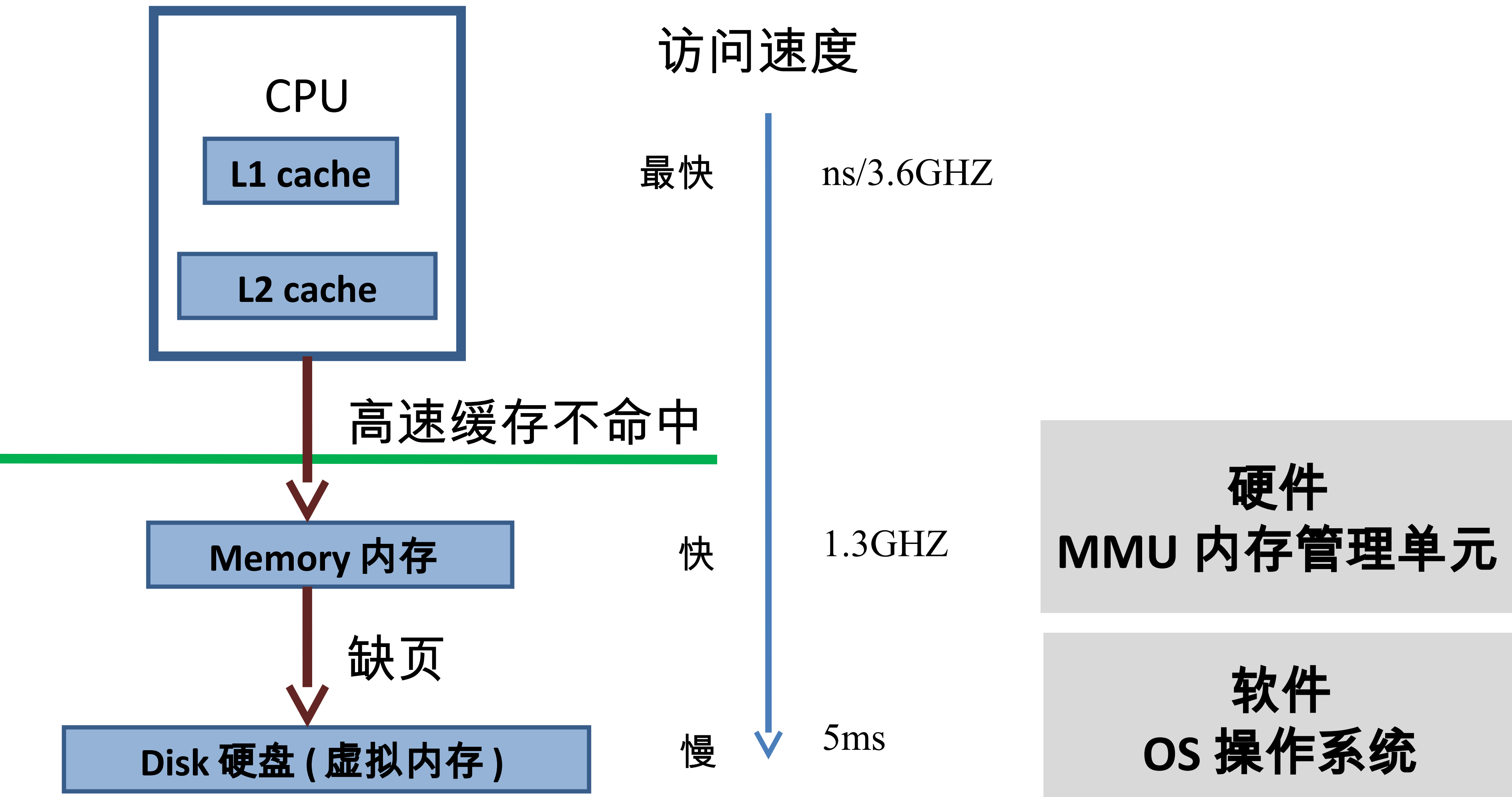
- 7.1 Memory Management Requirements
- 7.2 Memory Partitioning
- 7.3 Paging
- 7.4 Segmentation
- 7.5 Summary

7.1 Memory Management Requirements(1/12)

Memory Hierarchies



7.1 Memory Management Requirements(2/12)



7.1 Memory Management Requirements(3/12)

- Subdividing memory to accommodate multiple processes(为支持多道程序将内存进行划分)
- Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time(内存管理应确保有适当数目的就绪进程使用处理器时间)

7.1 Memory Management Requirements(4/12)

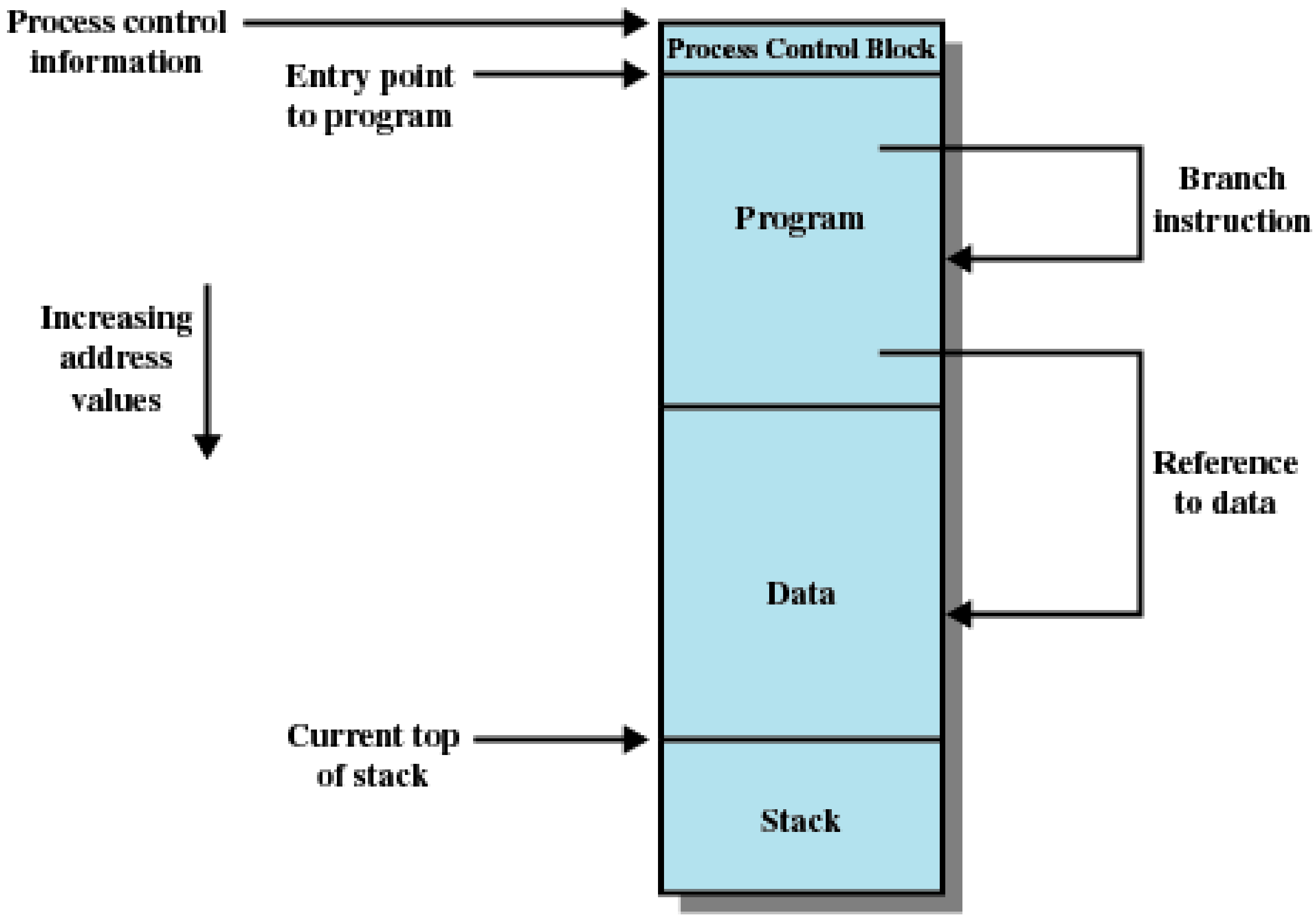


Figure 7.1 Addressing Requirements for a Process

7.1 Memory Management Requirements(5/12)

- OpenEuler 虚拟地址空间布局

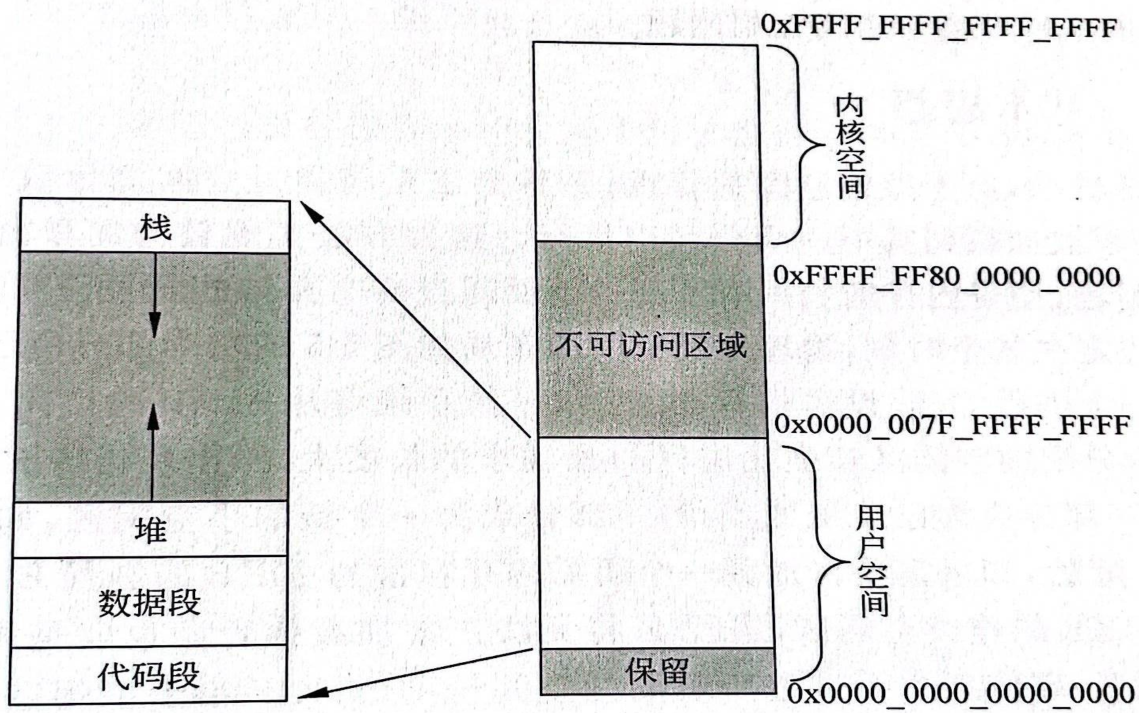
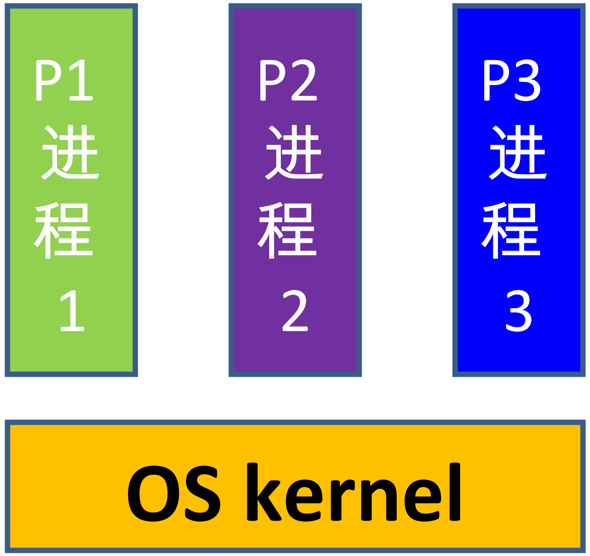
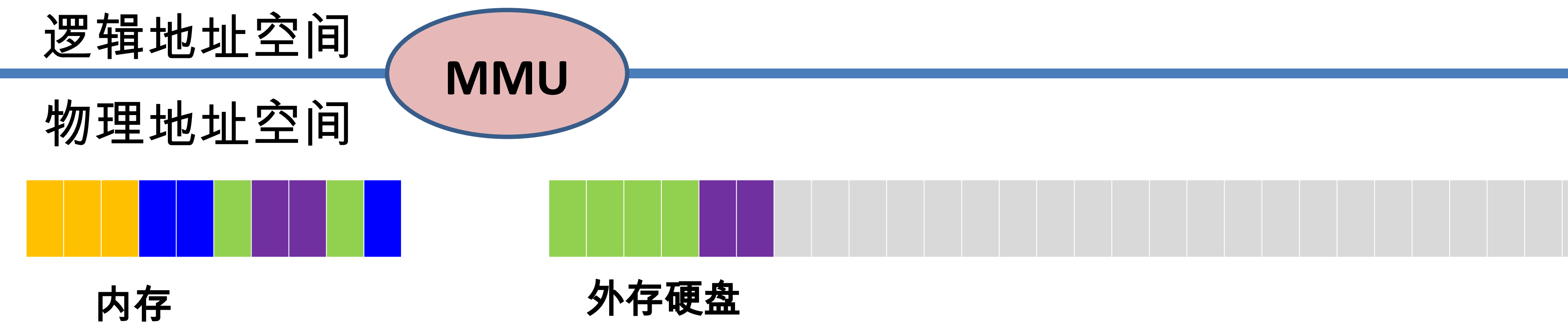


图 5-4 openEuler 虚拟地址空间布局示意

7.1 Memory Management Requirements(6/12)



relocation 重定位
Protection 保护
Sharing 共享



7.1 Memory Management Requirements(7/12)

- Relocation(重定位)
  - Programmer does not know where the program will be placed in memory when it is executed 程序员无关
  - While the program is executing, it may be swapped(交换) to disk and returned to main memory at a different location (relocated) 每次载入位置允许不同
  - Memory references must be translated in the code to actual physical memory address(逻辑地址到物理内存地址)
  - Q ： whose job ？ OS or hardware

7.1 Memory Management Requirements(8/12)

- Protection(保护)
  - Processes should not be able to reference memory locations in another process without permission or jump to instructions area of another process (进程不能在未授权的情况下访问其他进程的数据，不能跳转到其他进程的代码区域执行指令)
  - Normally, processes cannot access any portion of the OS, neither program nor data
  - Impossible to check absolute addresses at compile time, instead, absolute addresses must be checked ***at rum time. 运行时检测绝对地址***
  - Whose Job ？ Memory protection requirement must be satisfied by the processor (***hardware***) rather than the operating system
    - MMU on chip

7.1 Memory Management Requirements(9/12)

- Sharing(共享)
  - Allow several processes to access the same portion of memory
    - Share same copy of the program
    - Share data structure to cooperate on some task

7.1 Memory Management Requirements(10/12)

- Logical Organization(逻辑组织)
  - Conflicts
    - Main memory is organized in a linear address space, consisting of a sequence of bytes or words
    - 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
  - Segmentation 分段 satisfies these requirements

7.1 Memory Management Requirements(11/12)

- Physical Organization(物理组织)
  - Memory is organized into at least two levels, referred to as main memory and secondary memory(disk)
  - Memory available for a program plus its data may be insufficient(内存对程序和其数据来说可能不足)
    - Overlaying(覆盖) allows various modules to be assigned the same region of memory
  - Programmer does not know how much space will be available and where his/her program will be loaded in memory

7.1 Memory Management Requirements(12/12)

- Keys of memory management
  - Relocation
  - Segmentation
  - Paging
  - Virtual memory

Agenda

- 7.1 Memory Management Requirements
- <u>7.2 Memory Partitioning</u>
- 7.3 Paging
- 7.4 Segmentation
- 7.5 Summary

7.2 Memory Partitioning

固定分区	过时
动态分区	
简单分页	未投入实用
简单分段	
虚存分页	页大小一致
虚存分段	段大小不一致

7.2 Memory Partitioning(内存分区)

- <u>7.2.1 Fixed Partitioning</u>
- 7.2.2 Dynamic Partitioning
- 7.2.3 Buddy System
- 7.2.4 Relocation

7.2.1 Fixed Partitioning(1/9)(固定分区)

- Alternatives for Fixed Partitioning
  - Equal-size partitions(大小相等的分区)
  - Unequal-size partitions(大小不等的分区)

7.2.1 Fixed Partitioning(2/9)

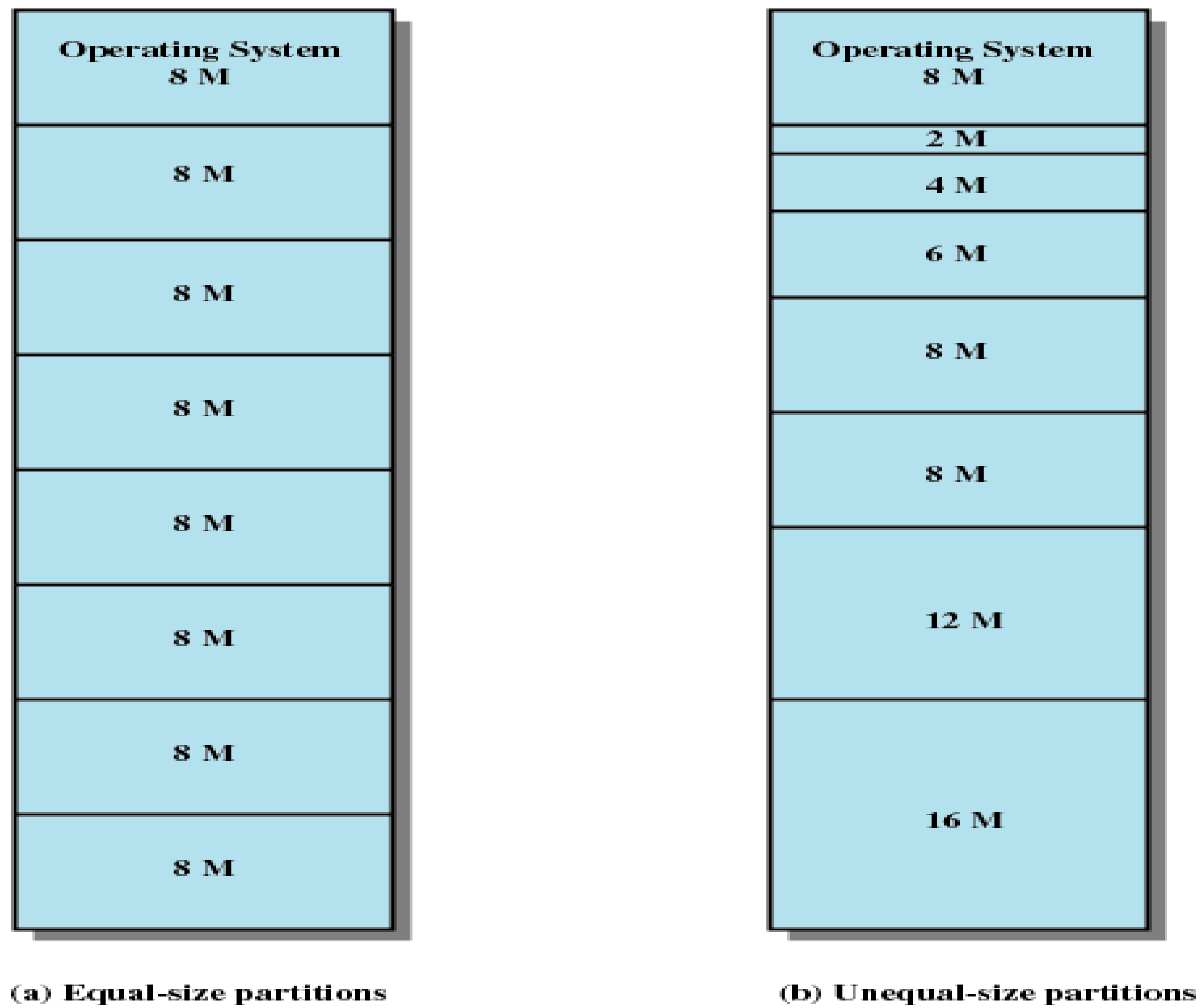


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

7.2.1 Fixed Partitioning(3/9)

- Equal-size partitions: features
  - 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

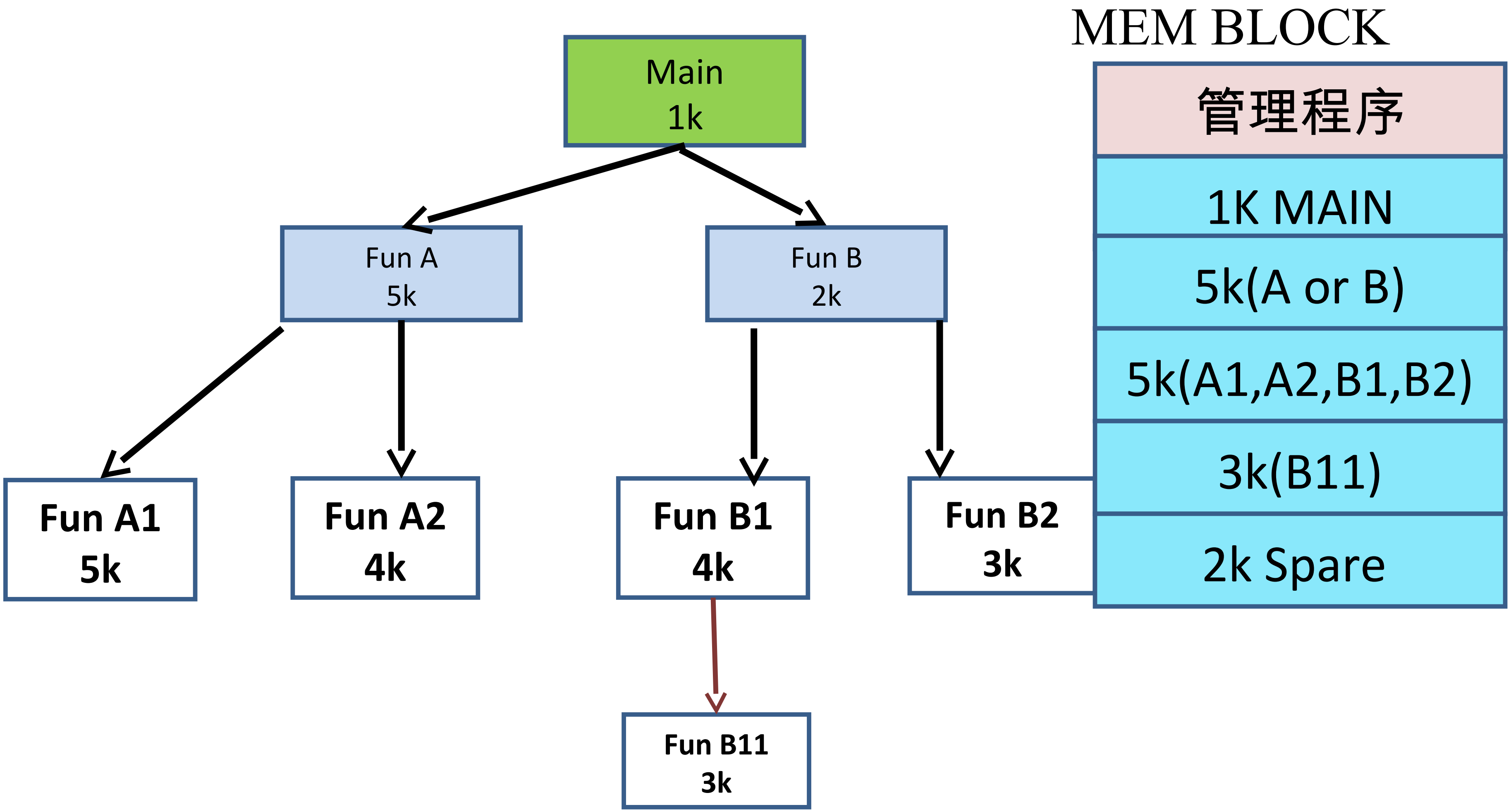
7.2.1 Fixed Partitioning(4/9)

- Equal-size partitions: difficulties
  - A program may not fit in a partition. The programmer must design the program with **overlays 覆盖**
  - Main memory use is inefficient. Any program, no matter how small, occupies an entire partition.
    - **internal fragmentation(内部碎片 / 内零头)**.

- Equal-size partitions: benefits
  - Because all partitions are of equal size, it does not matter which partition is used or replaced

7.2.1 Fixed Partitioning(5/9)

- **overlays 覆盖** by programmer



7.2.1 Fixed Partitioning(6/9)

- Unequal-size partitions(大小不等的分区)
  - Both of these problems of equal-size partitions can be lessened, though not solved, by using unequal-size partitions.

7.2.1 Fixed Partitioning(7/9)

- Unequal-size partitions
  - Policy: 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

7.2.1 Fixed Partitioning(8/9)

Placement Algorithm(放置算法) with Fixed Partitions ?

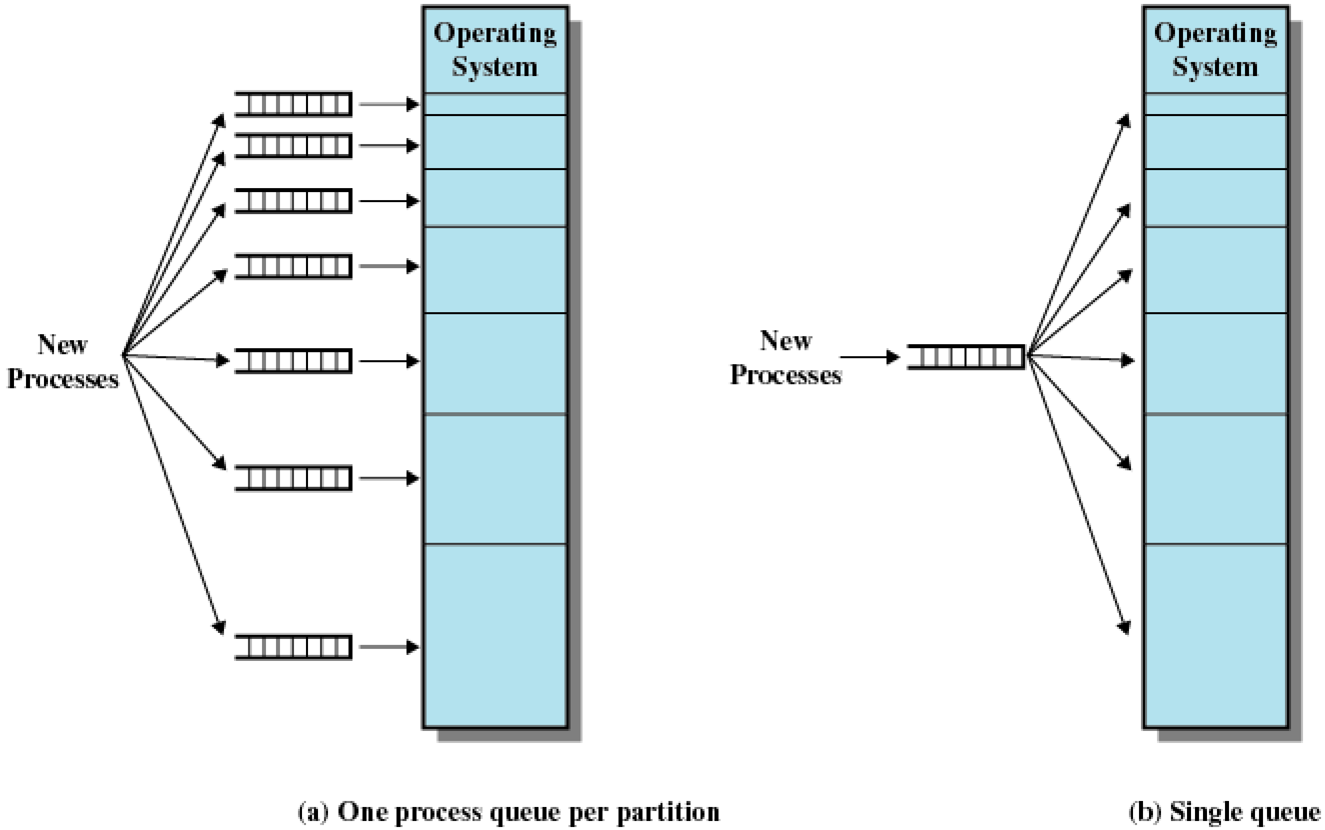


Figure 7.3 Memory Assignment for Fixed Partitioning

7.2.1 Fixed Partitioning(9/9)

- Disadvantages of Fixed Partitions
  - The number of active processes in the system is limited by the number of partitions
  - Small jobs will not utilize partition efficiently

7.2 Memory Partitioning

- 7.2.1 Fixed Partitioning
- <u>7.2.2 Dynamic Partitioning</u>
- 7.2.3 Buddy System
- 7.2.4 Relocation

7.2.2 Dynamic Partitioning(1/6)(动态分区)

- 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(外部碎片 / 外零头)**
- Must use compaction(压缩) to shift(移动) processes so they are contiguous and all free memory is in one block

7.2.2 Dynamic Partitioning(2/6)

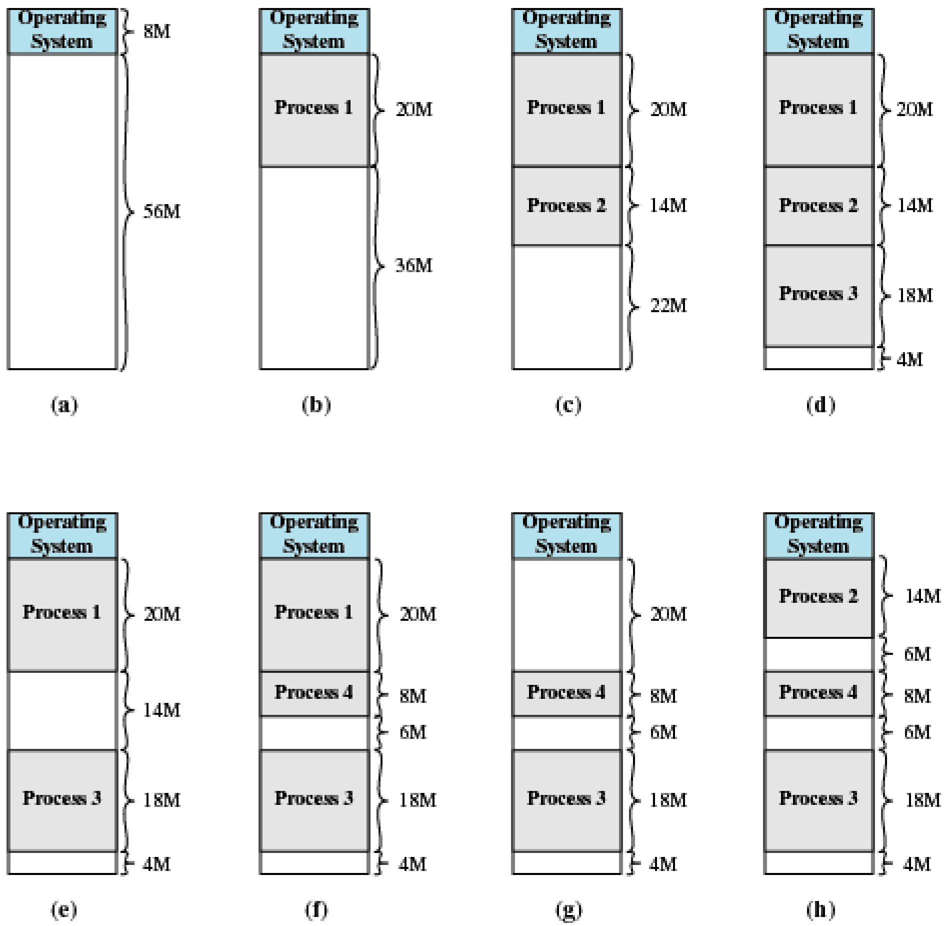


Figure 7.4 The Effect of Dynamic Partitioning

7.2.2 Dynamic Partitioning(3/6)

Dynamic Partitioning Placement Algorithm

•Three placement algorithms ：

1.Best-fit algorithm(最佳适配）性能最差

- 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

7.2.2 Dynamic Partitioning(4/6)

2. First-fit algorithm(首次适配）性能最佳
   - Scans memory form the beginning and chooses the first available block that is large enough
   - Simplest and usually fastest and best
   - May have many process loaded in the front end of memory that must be searched over when trying to find a free block

7.2.2 Dynamic Partitioning(5/6)

- Next-fit(邻近适配）性能次佳
  - Scans memory from the location of the last placement and chooses the next available block that is large enough
  - More often allocate 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

7.2.2 Dynamic Partitioning(6/6)

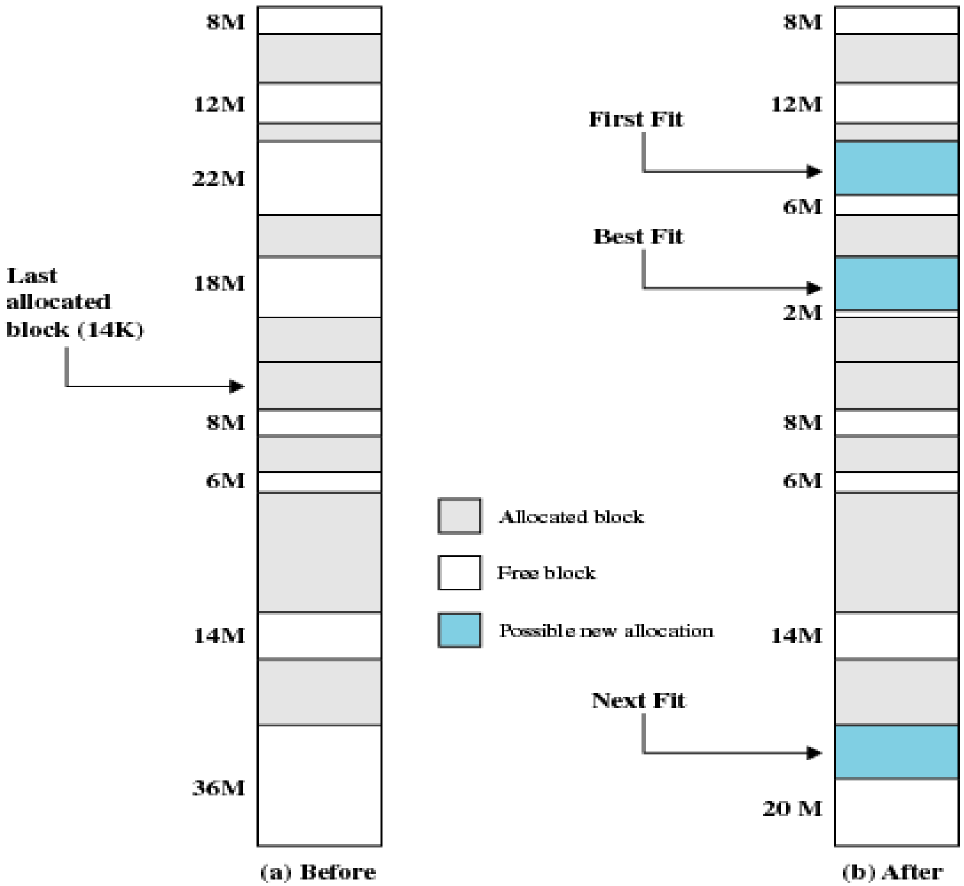


Figure 7.5 Example Memory Configuration Before and After Allocation of 16 Mbyte Block

7.2 Memory Partitioning

- 7.2.1 Fixed Partitioning
- 7.2.2 Dynamic Partitioning
- 7.2.3 Buddy(伙伴) System
- 7.2.4 Relocation

7.2.3 Buddy 伙伴 System(1/3)

- Entire space available is treated as a single block of 2^U (e.g. 1M = 2^{20})
- 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

7.2.3 Buddy 伙伴 System(2/3)

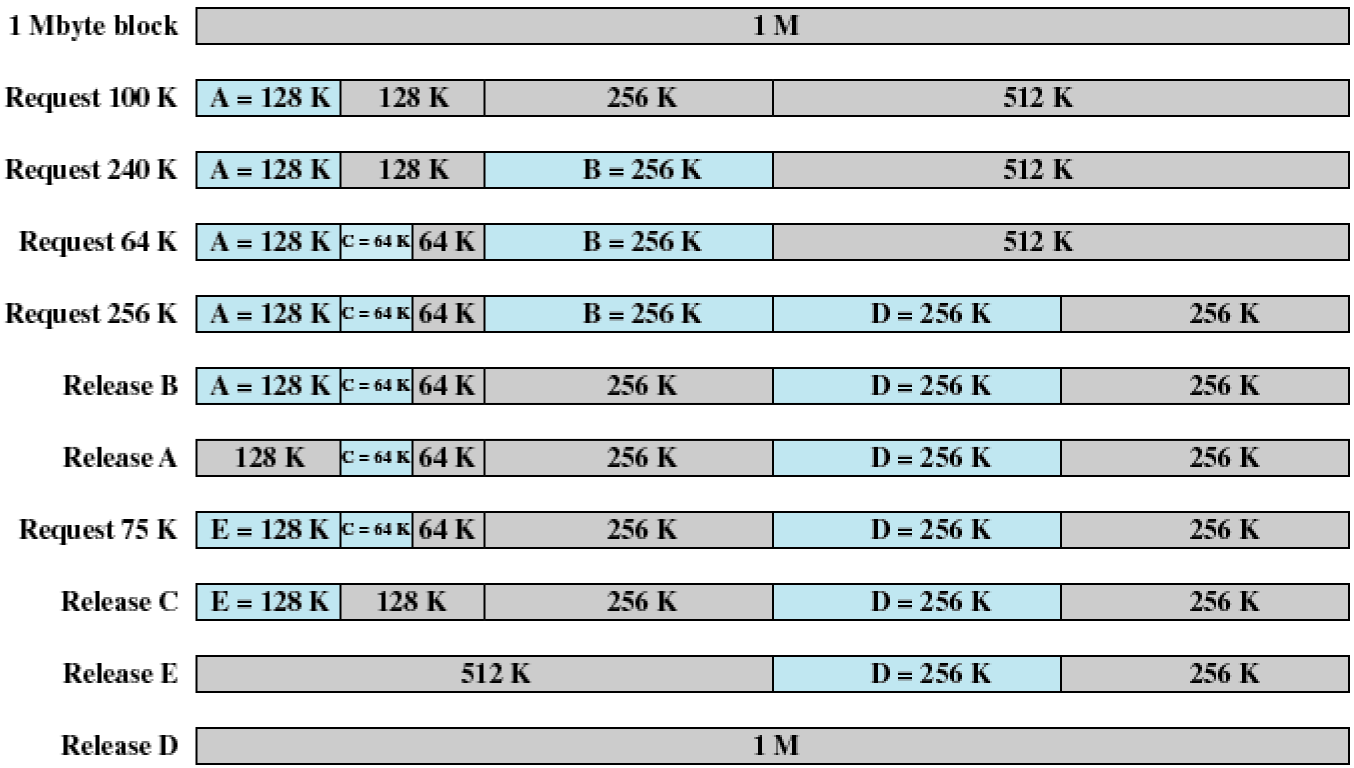


Figure 7.6 Example of Buddy System

7.2.3 Buddy 伙伴 System(3/3)

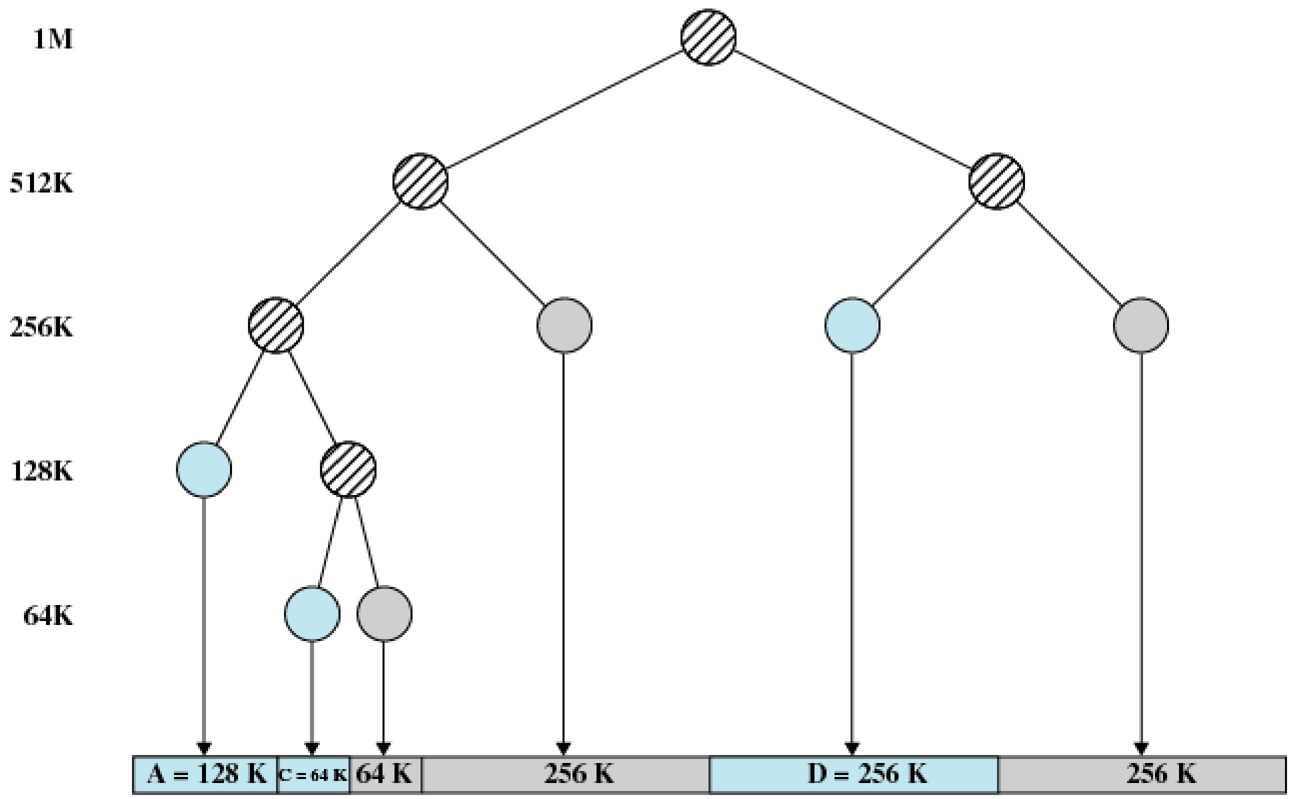


Figure 7.7 Tree Representation of Buddy System

7.2 Memory Partitioning

- 7.2.1 Fixed Partitioning
- 7.2.2 Dynamic Partitioning
- 7.2.3 Buddy System
- 7.2.4 Relocation

7.2.4Relocation(1/6)(重定位)

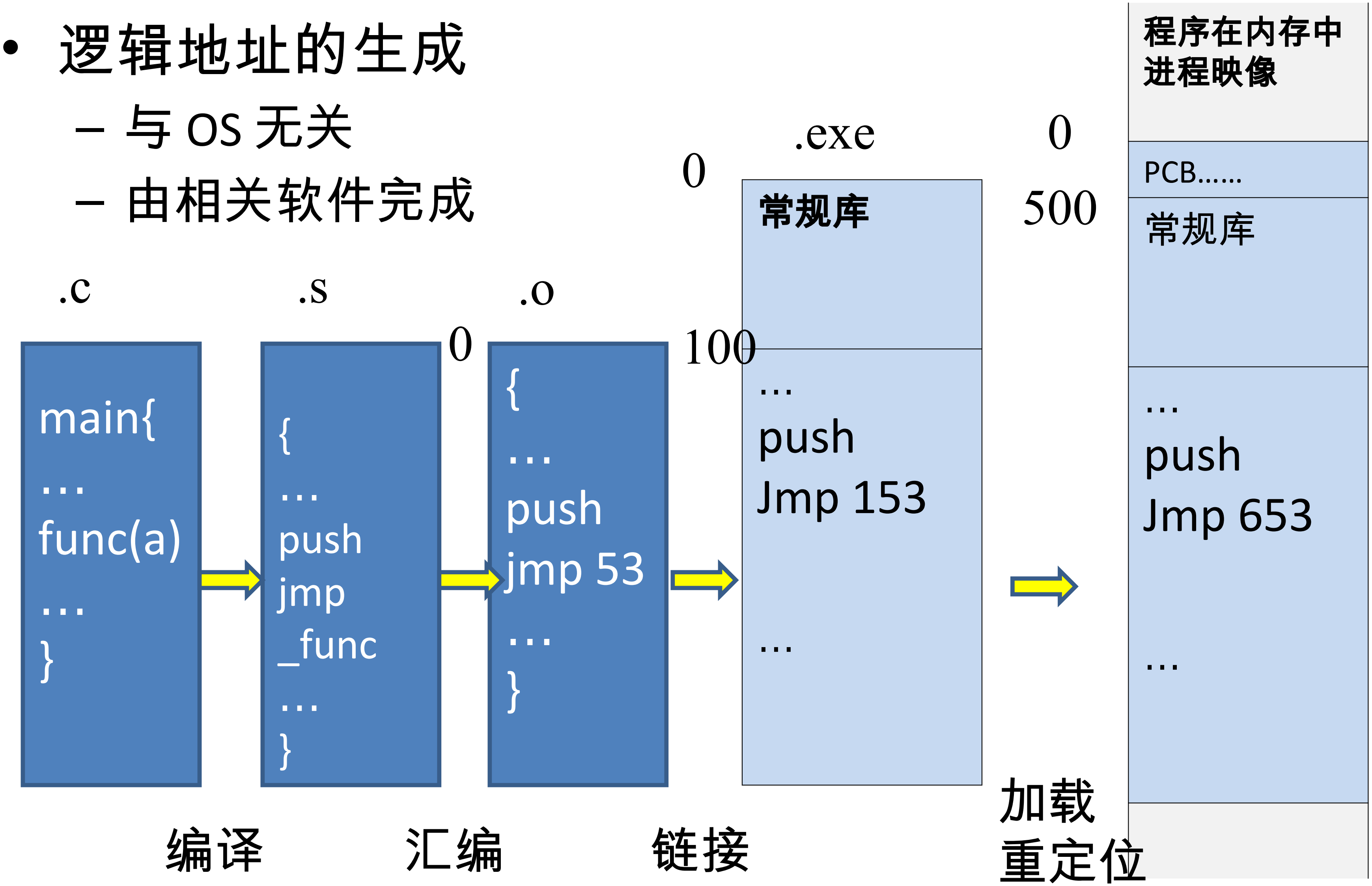
- When program loaded into(载入) memory the actual (absolute) memory locations are determined
- A process may occupy different partitions which means different absolute memory locations during execution (from swapping 交换)
- Compaction(压缩) will also cause a program to occupy a different partition which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

7.2.4Relocation(2/6)

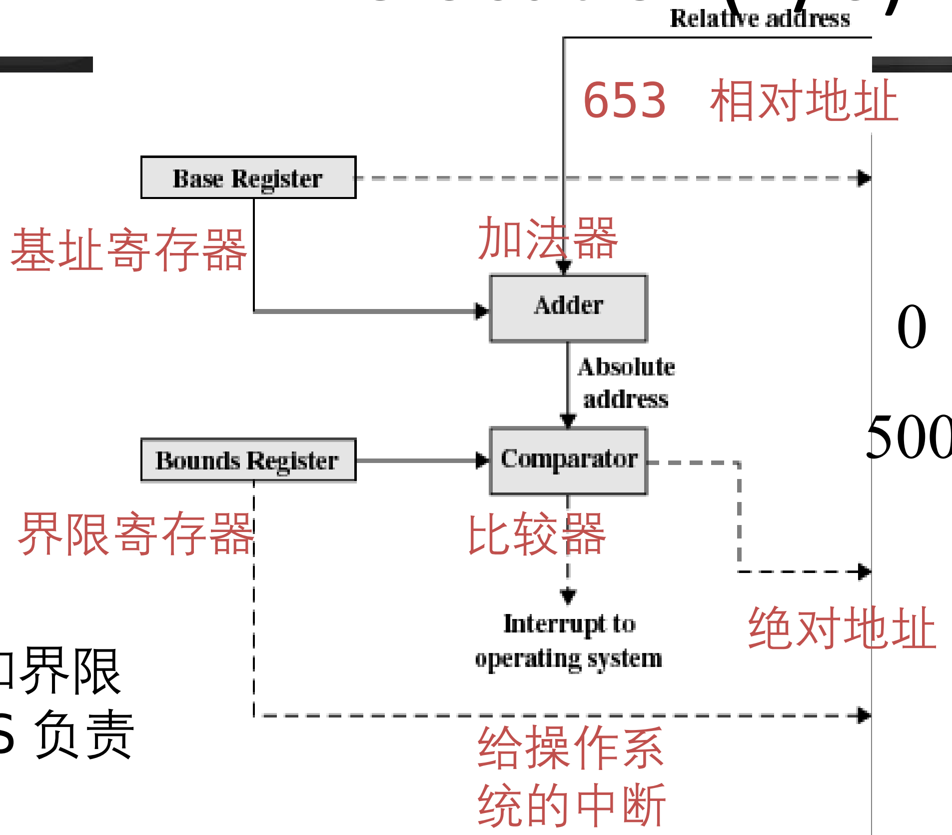
- Addresses
  - Logical Address(逻辑地址)
    - Reference to a memory location independent of the current assignment of data to memory(与当前数据在物理内存中分配无关的访问地址)
    - Textbook 8.1 进程中的所有内存访问都是逻辑地址
    - Translation must be made to the physical address
  - Relative Address(相对地址)
    - Address expressed as a location relative to some known point
  - Physical Address(物理地址)
    - The absolute address(绝对地址) or actual location in main memory

7.2.4Relocation(3/6)

- 逻辑地址的生成
  - 与 OS 无关
  - 由相关软件完成



7.2.4Relocation(4/6)



对进程进行了内存保护

1. 图中，基址和界限寄存器的值有 OS 负责生成
2. 但具体的逻辑地址生成方法见第 8 章，OS 的主要工作是完成逻辑到物理地址的映射表（段表页表）
3. CPU 内的硬件 MMU 完成地址转换

Figure 7.8 Hardware support for relocation

7.2.4Relocation(5/6)

- Registers Used during Execution
  - Base register(基址寄存器)
    - Starting address for the process
  - Bounds register(界限寄存器)
    - Ending location of the process
  - Whose job? OS: These values are set when the process is loaded(加载) or when the process is swapped in(换入)

7.2.4Relocation(6/6)

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

Agenda

- 7.1 Memory Management Requirements
- 7.2 Memory Partitioning
- 7.3 Paging
- 7.4 Segmentation
- 7.5 Summary

7.3 Paging(1/6) (分页)

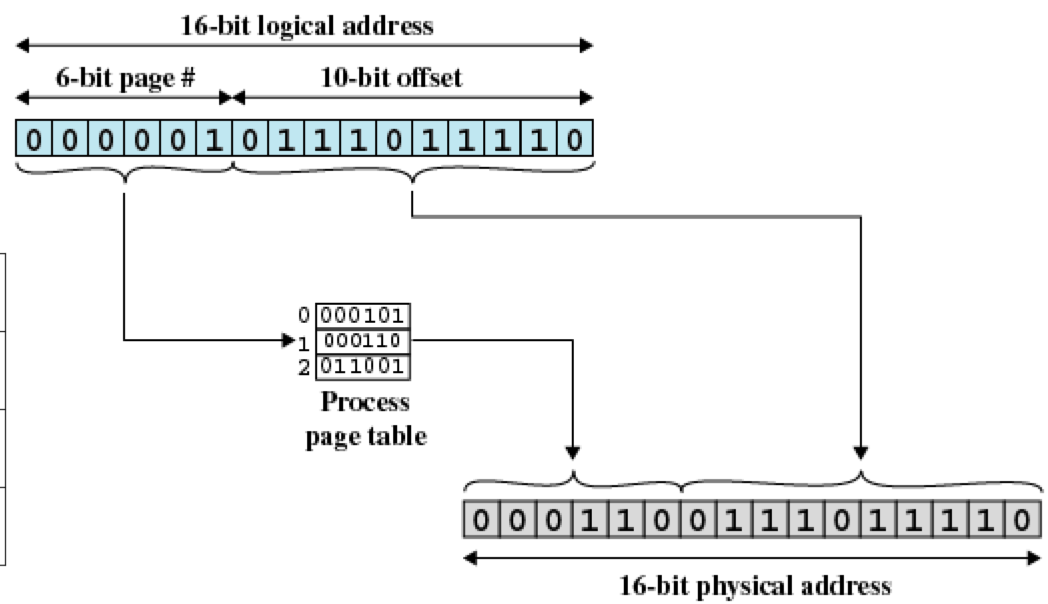
- Partition memory into small equal fixed-size chunks(块) which are called frames(帧)
- Divide each process into small equal fixed-size chunks which are called pages(页).
- The size of pages == the size of frames
- Memory address consist of a page number(页号) and offset(偏移量) within the page
- Operating system maintains a page table(页表) for each process ：帧页对应关系表
  - Contains the frame location(帧位置) for each page in the process

7.3 Paging(2/6)

- Memory address ： page number(页号) + offset(偏移量)
- Operating system maintains a page table(页表) for each process
- Contains the frame location(帧位置) for each page in the process

进程 A

A.0page
A.1page address ： 0x05DE
A.2page
A.3page



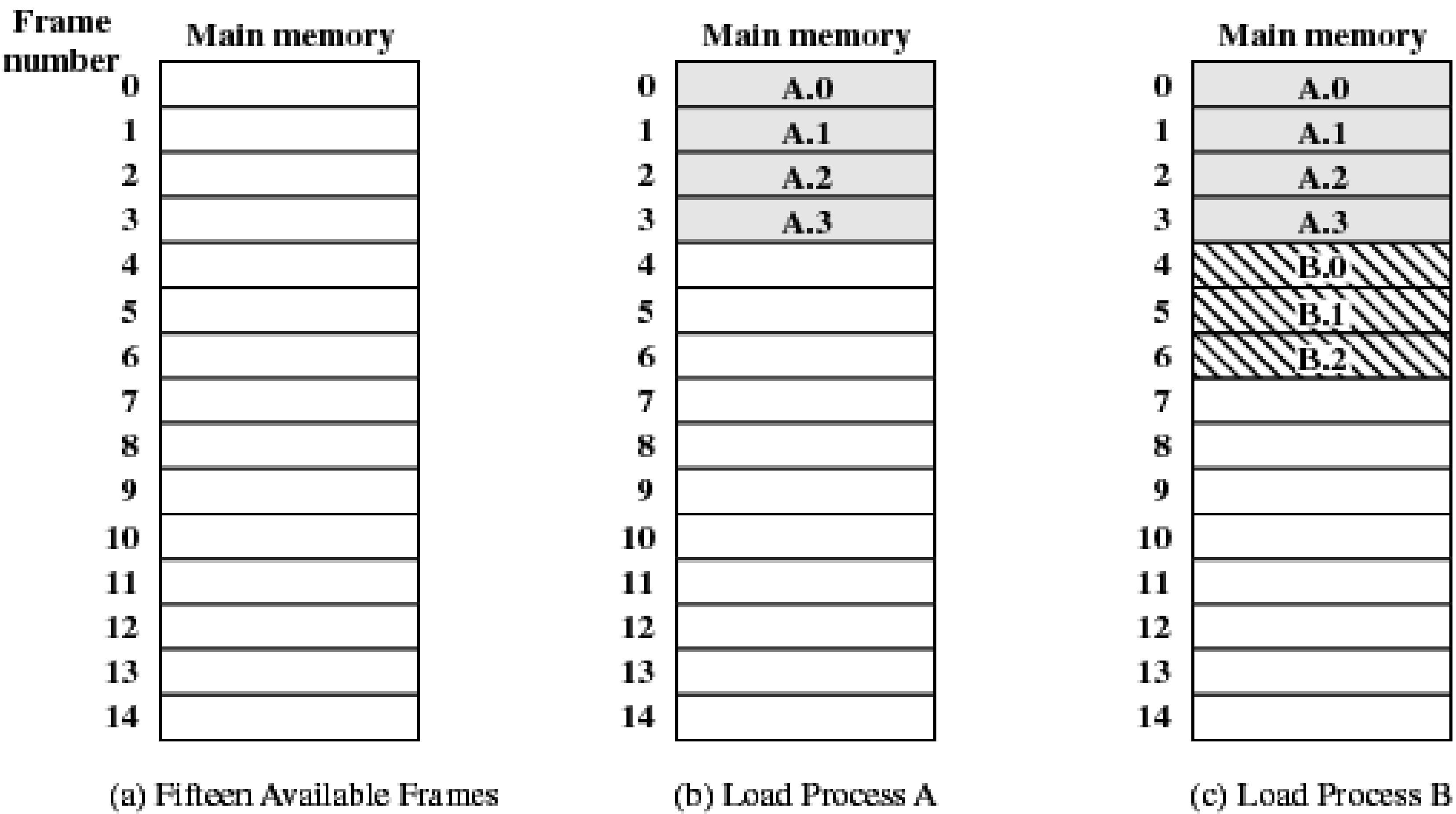
(a) Paging

7.3 Paging(3/6)

地 Frame no	址 frame	内存
000000		
000001		
000010		
000011		
000100		
000101	0000000000 - 1111111111	
000110	0000000000 -0111011110 1111111111	
		…

地 Frame no	址 frame	进程映像
000000	0000000000 - 1111111111	
000001	0000000000 -0111011110 1111111111	
000010	0000000000 - 1111111111	
000011	0000000000 - 1111111111	

7.3 Paging(4/6)



Assignment of Process Pages to Free Frames

7.3 Paging(5/6)

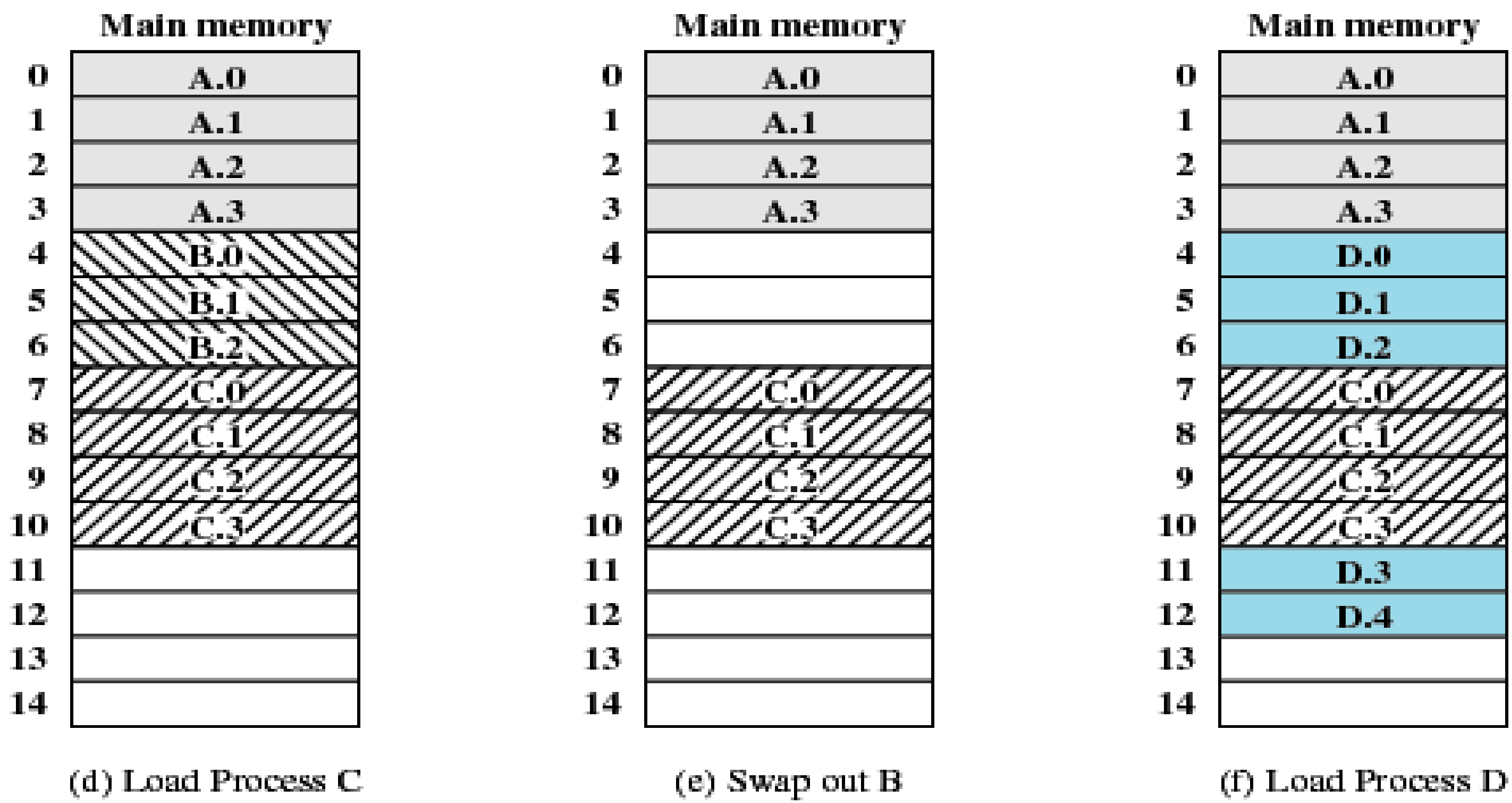
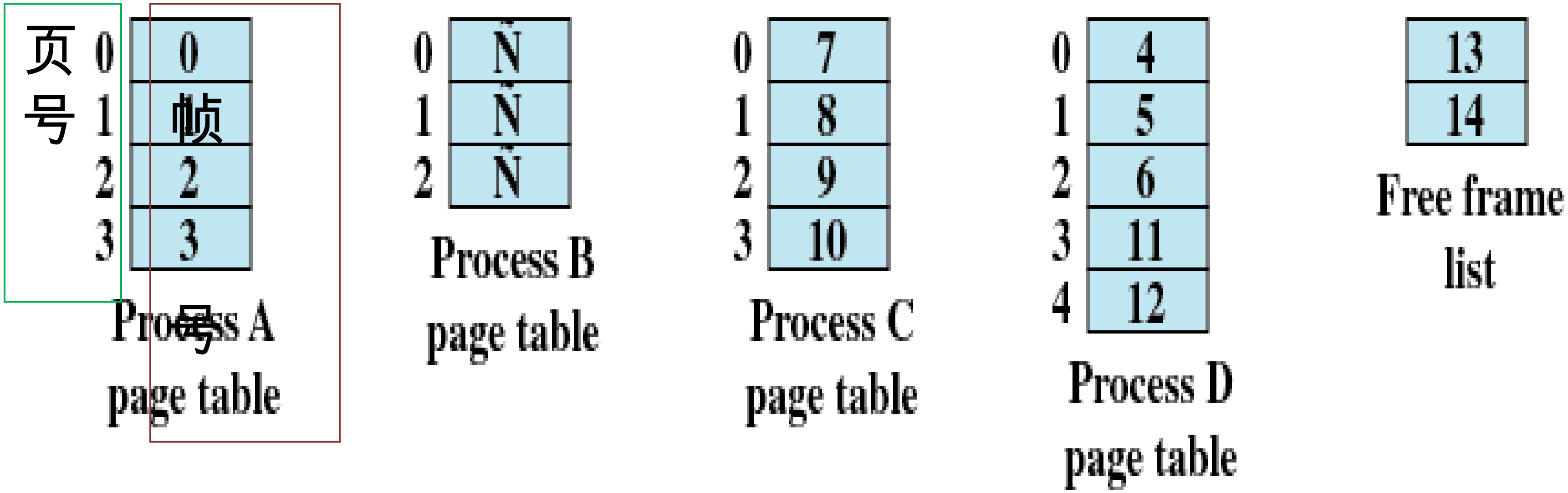


Figure 7.9 Assignment of Process Pages to Free Frames

7.3 Paging(6/6)



Lab09
paging.c

Page Tables for Example

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

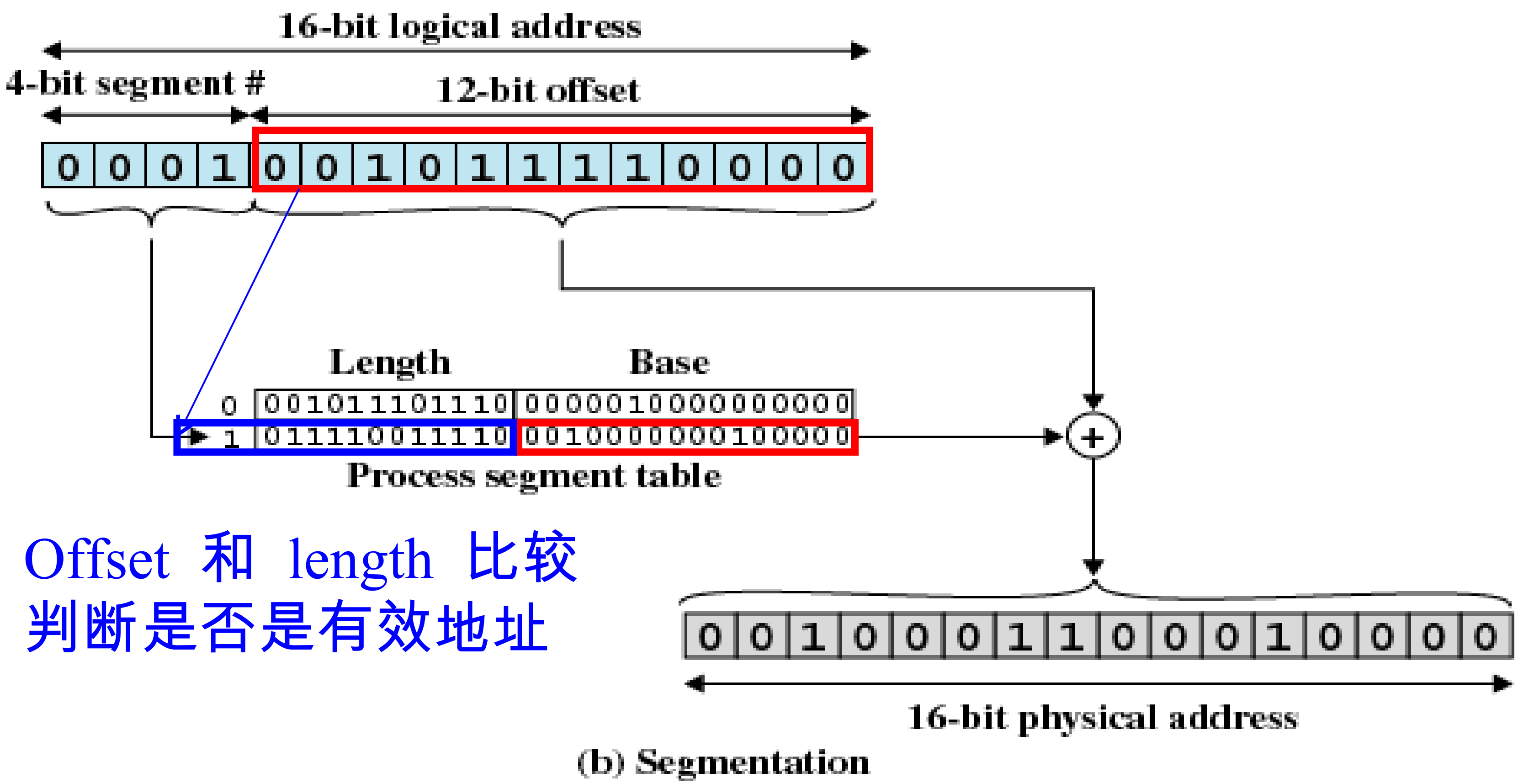
Agenda

- 7.1 Memory Management Requirements
- 7.2 Memory Partitioning
- 7.3 Paging
- <u>7.4 Segmentation</u>
- 7.5 Summary

7.4 Segmentation(1/3)(分段)

- Program and its data can be divided into a number of ***segmentation***, 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(段号)
  - offset(偏移量)
- Since segments are not equal, segmentation is similar to dynamic partitioning(动态分区)

7.4 Segmentation(2/3)(分段)



Offset 和 length 比较
判断是否是有效地址

Figure 7.12 Examples of Logical-to-Physical Address Translation

7.4 Segmentation(3/3)(分段)

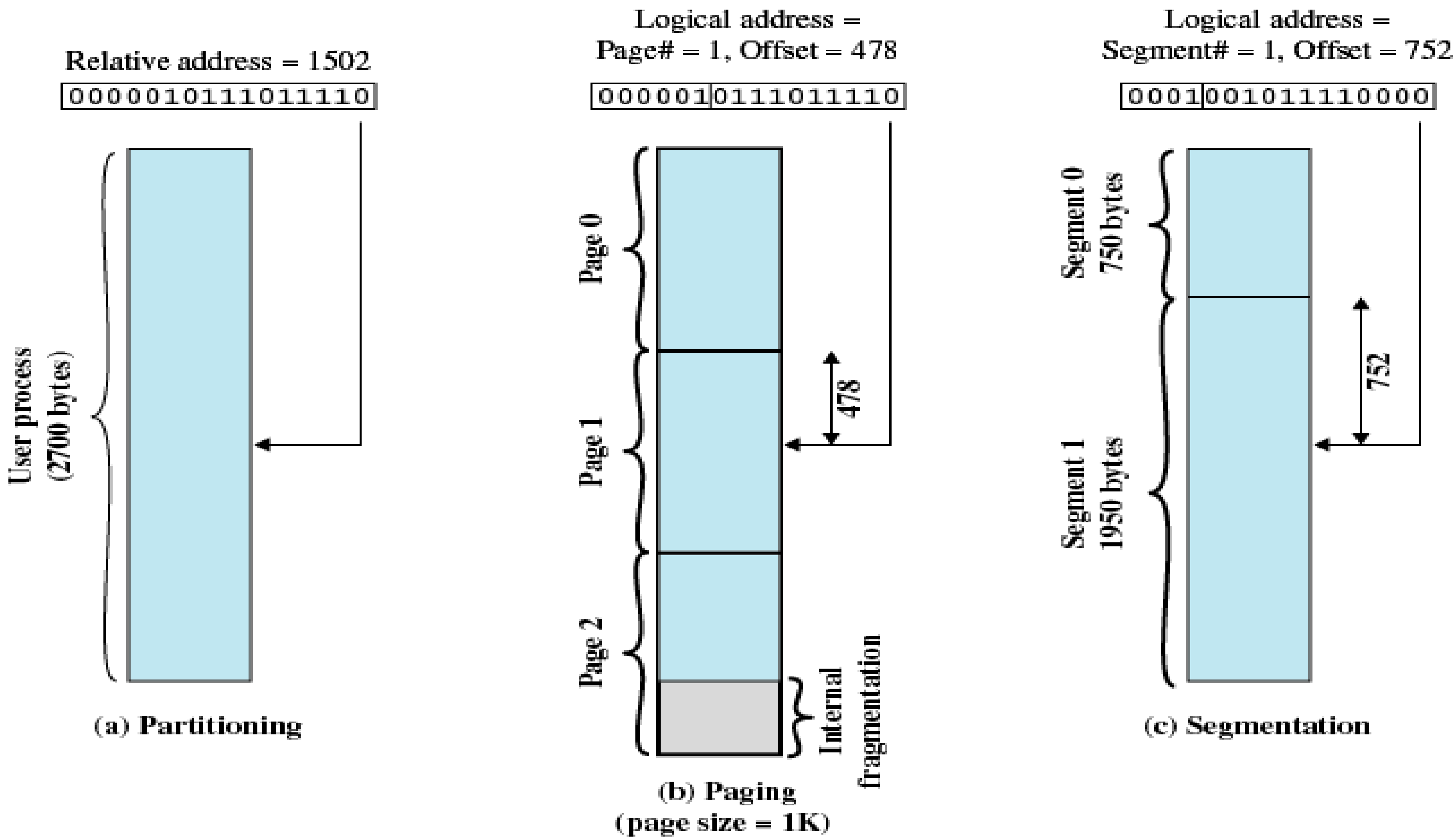


Figure 7.11 Logical Addresses

Agenda

- 7.1 Memory Management Requirements
- 7.2 Memory Partitioning
- 7.3 Paging
- 7.4 Segmentation
- <u>7.5 Summary</u>

Summary

- 解决的问题
  - 进程可以分割为模块装载
  - 程序员无关
- 尚未解决的问题
  - 页表，段表大小，检索速度问题

拓展： bitmap(1/9)

- Q：如何表示空间是否可用？

- 位图（ bitmap) 为 n 个二进制位的串，用于表示 n 项的状态。
  - 如有 n 个资源，第 i 个 bit 表示资源 i 的可用性,0 表示可用，而 1 表示不可用（或相反）。
    - 例如，现有如下位图： 00101110
    - 第 1、2、3、5 资源是不可用的，第 0、4、6 和 7 资源是可用的。

7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0

拓展：bitmap(2/9)

- 当考虑空间效率时，位图优势明显。如果所用的布尔值是 8 位的而不是 1 位的，那么最终的数据结构将会是原来的 8 倍。因此，当需要表示大量资源的可用性时，通常采用位图。
- 磁盘驱动器就是这么工作的。一个中等大小的磁盘可以分成数千个单元，称为磁盘块（ disk block ）。每个磁盘块的可用性就可通过位图来表示。

拓展： bitmap(3/9)

- 某文件管理系统在磁盘上建立了位示图（ Bitmap ），记录磁盘的使用情况。若磁盘上的物理块依次编号为 0 、 1 、 2 、…，系统中字长为 32 位，每一位对应文件存储器上的一个物理块，取值 0 和 1 分别表示空闲和占用，如下所示。假设将 4195 号物理块分配给某文件，那么该物理块的使用情况在位示图中的第（　）个字（从 0 编号）中描述；选择（ ）
- A . 128
  B . 129
  C . 130
  D . 131

拓展： bitmap(4/9)

- 因为物理块编号是从 0 开始的，所以 4195 号物理块其实就是第 4196 块。因为字长为 32 位，也就是说，每个字可以记录 32 个物理块的使用情况。 4196/32=131.125 ，所以， 4195 号物理块应该在第 131 个字中（字的编号也是从 0 开始计数）。那么，具体在第 131 个字的哪一位呢。到第 130 个字为止，共保存了 131×32=4192 个物理块（ 0 ～ 4191 ），所以，第 4195 块应该在第 131 个字的第 3 位记录（要注意： 0 是最开始的位）。因为系统已经将 4195 号物理块分配给某文件，所以其对应的位要置 1 。
- 如何判断某个物理块是否占用？

拓展： bitmap(5/9)

- 假设用 char bitmap[8] ；来描述 64 个块的分配情况

bitmap[0]	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**
	7	6	5	4	3	2	1	0
bitmap[1]	0	0	0	0	0	1	0	1
	15	14	13	12	11	10	9	8

假设左高位右位低

- 如修改 n=14 为占用
- 1. 先找到 n 在 bitmap 数组中的下标 index ，显然 index = 1
  - index = n / 8 = n >> 3 ，即 1110>>3 得到 1
- 2. 找到 n 在 bitmap[index](bitmap[1]) 中的位置 position ，这里 position = 6 。
  - position = n % 8 , 即 n & 0x07 。

拓展： bitmap(6/9)

- 3. 对对应 bit 做或操作
  - bitmap[1] |= 1<<position

位移

0	1	0	0	0	0	0	0

或

0	0	0	0	0	1	0	1
15	14	13	12	11	10	9	8

结果

0	1	0	0	0	1	0	1
15	14	13	12	11	10	9	8

拓展： bitmap(7/9)

- *// 用 >> 的操作是，运算会比较快*
- **int** index = n >> 3;
- **int** position = n & 0x07;
- bitmap[index] |= 1 << position;//set to 1
- bitmap[index] &= ~(1 << position);// set to 0
- (bitmap[index] & (1 << position)) != 0;// 判断是否存在

拓展： bitmap(8/9)

- void bit_set(bits bit, unsigned int pos, unsigned char value)
- {
- unsigned char mask = 0x1 << (pos & 0x7);
- if (value)
-     bit-> bitmap[pos>>3] |= mask;
- else
-     bit-> bitmap[pos>>3] &= ~mask;
- }

拓展： bitmap(9/9)

- char bit_get(bits bit, unsigned int pos)
- {
-         unsigned char mask = 0x1 << (pos & 0x7);
-          return (mask & bit-> bitmap[pos>>3]) == mask ? 1 : 0;
- }