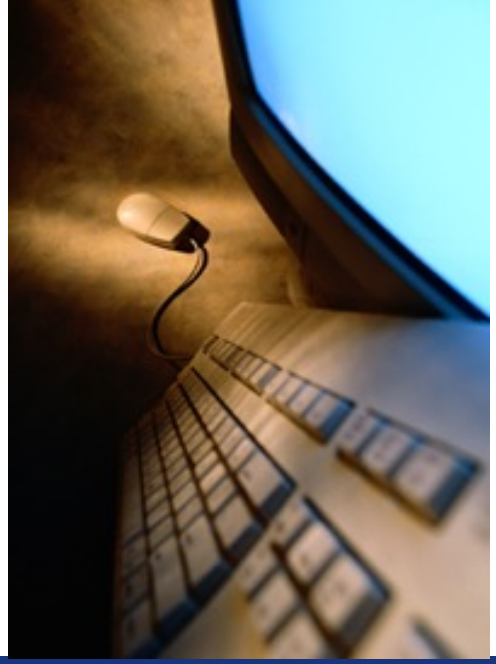
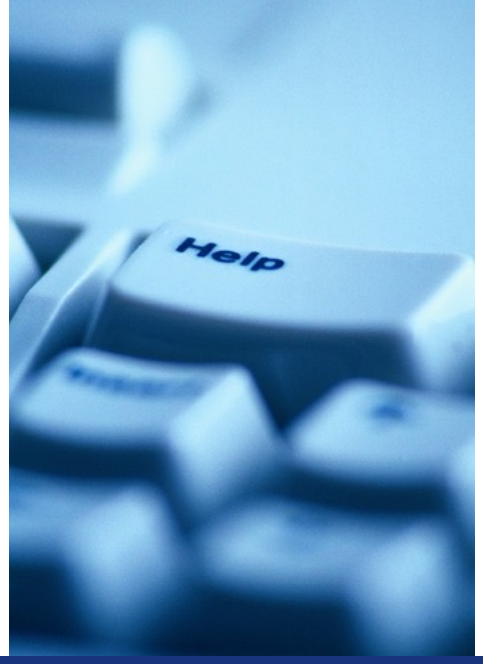
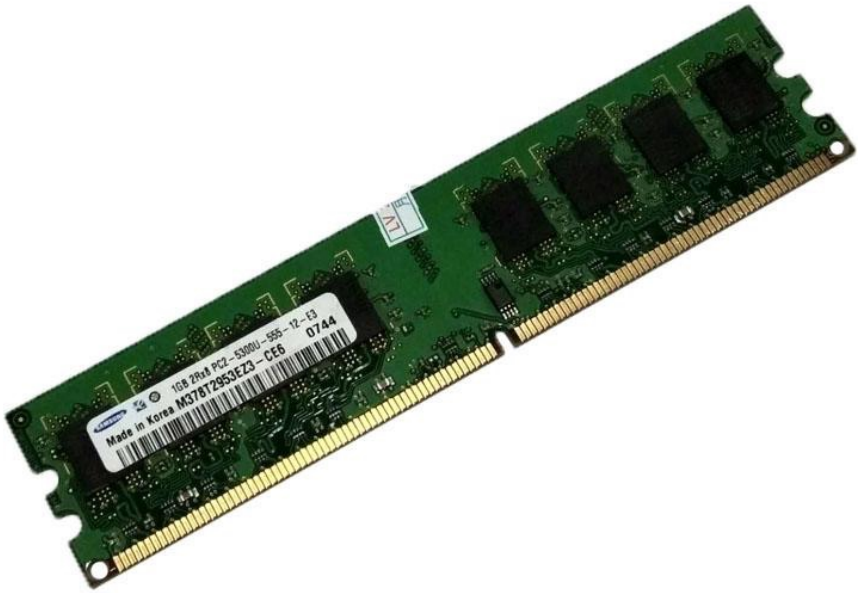



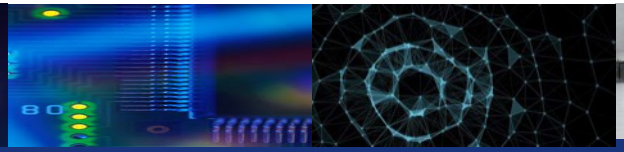
操作系统原理

Operating Systems Principles

陈鹏飞
计算机学院
2021-04-20

第八讲 — 内存管理





目标

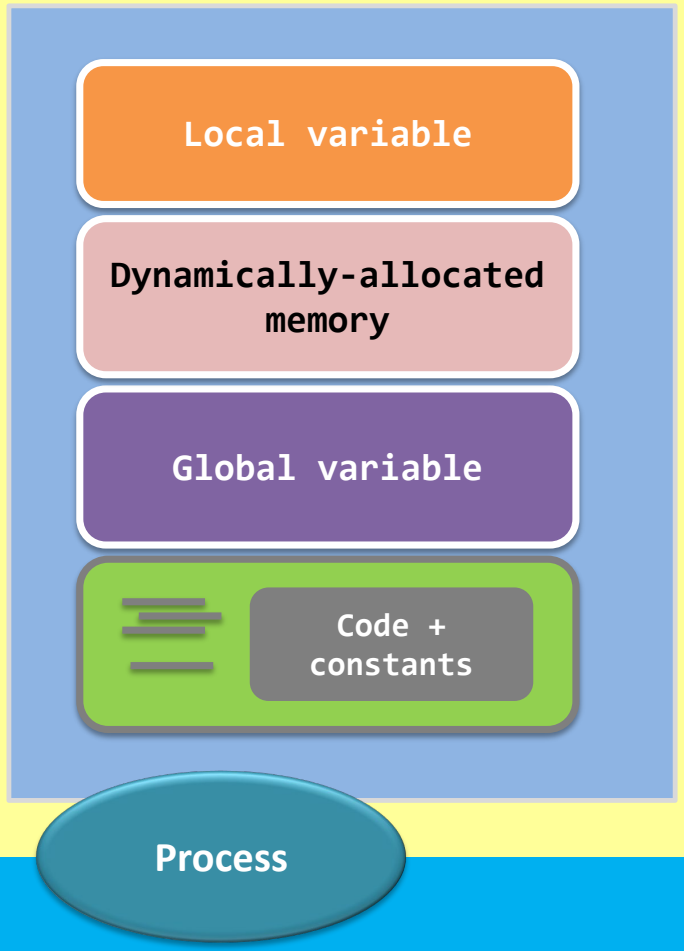
- 讨论内存管理的主要需求；
- 了解内存分区的原因并解释所用的各种技术；
- 理解并解释分页的概念；
- 理解并解释分段的概念；
- 评估分区和分段的优点；
- 了解装载和链接的概念；

Why we need memory management

- The running program code requires memory
  - Because the CPU needs to fetch the instructions from the memory for execution
- We must keep several processes in memory
  - Improve both CPU utilization and responsiveness
  - Multiprogramming

It is required to efficiently manage the memory

Part 1: User-space memory



Do you remember this?

- Content of a process (in user-space memory)

How does each part use the memory?

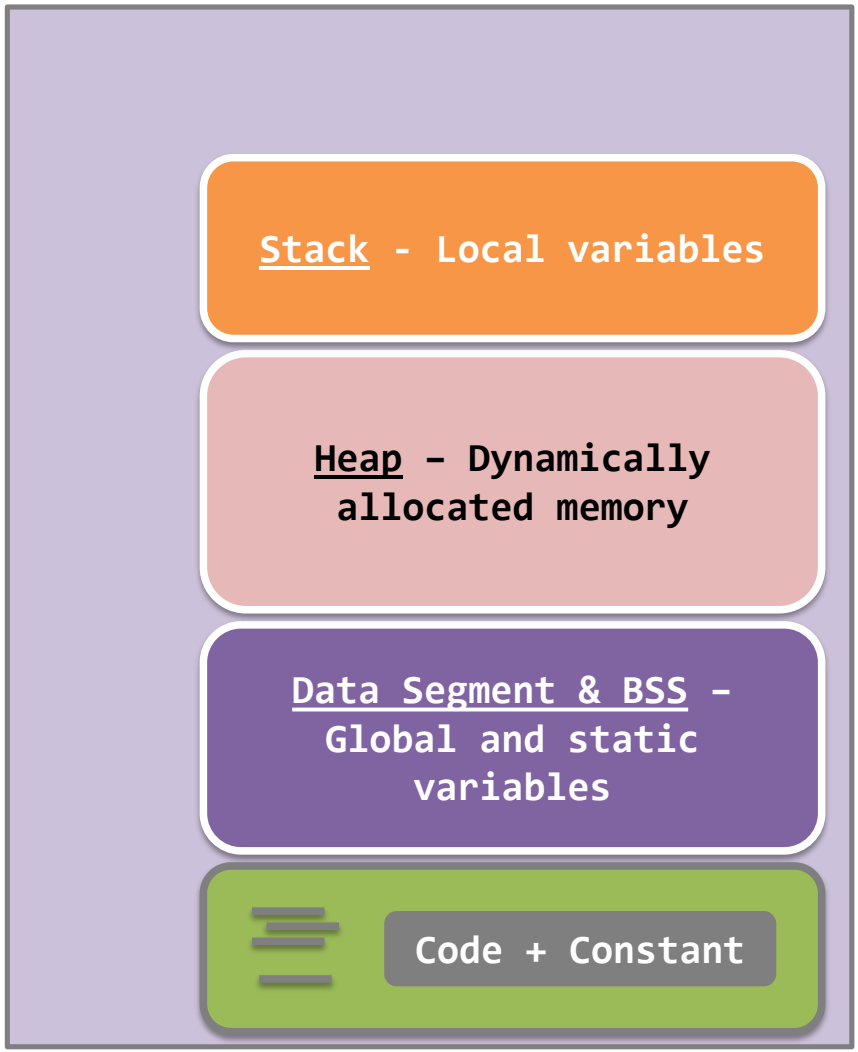
- From a programmer's perspective

Let's forget about the kernel for a moment. We are going to explore the **user-space memory** first.

Address space

How does a programmer look at the memory space?

- An array of bytes?
- Memory of a process is divided into segments
- This way of arranging memory is called **segmentation**



Address space

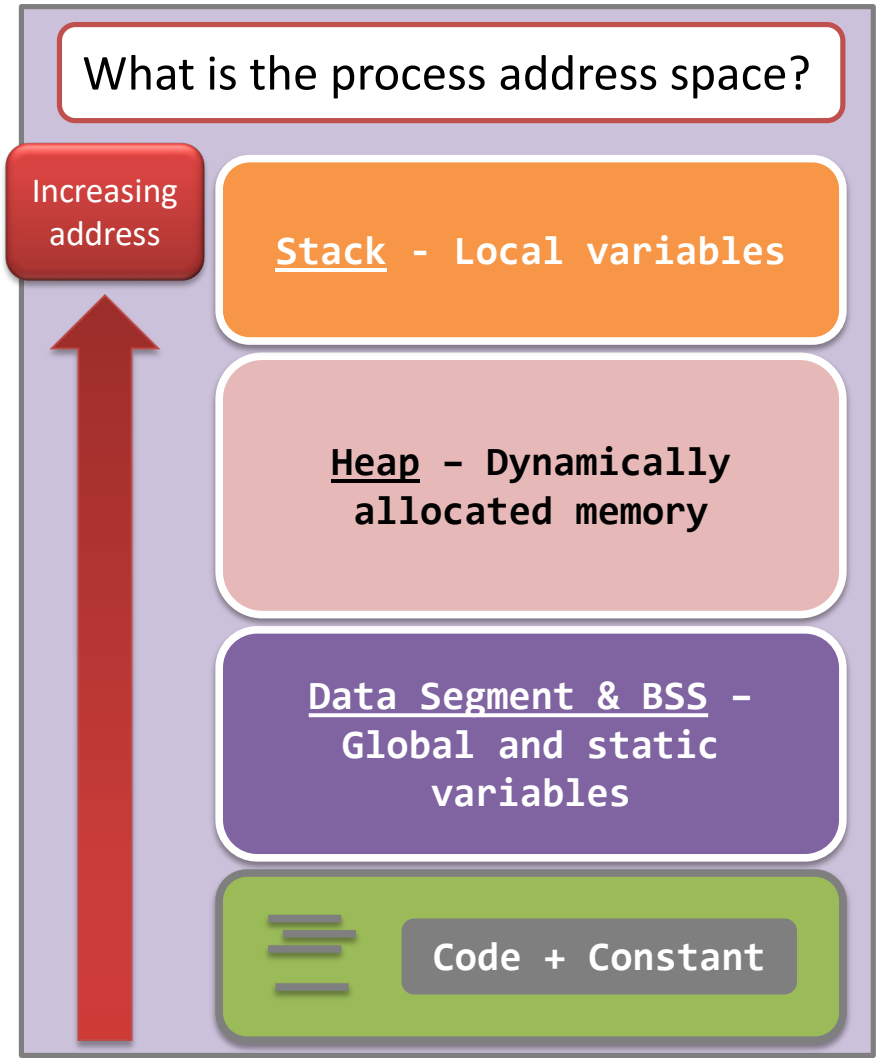
```c
int main(void) {
    int *malloc_ptr = malloc(4);
    char *constant_ptr = "hello";

    printf("Local variable  = %15p\n", &malloc_ptr);
    printf("malloc() space  = %15p\n", malloc_ptr);
    printf("Global variable = %15p\n", &global_int);
    printf("Code & constant = %15p\n", constant_ptr);

    return 0;
}
```

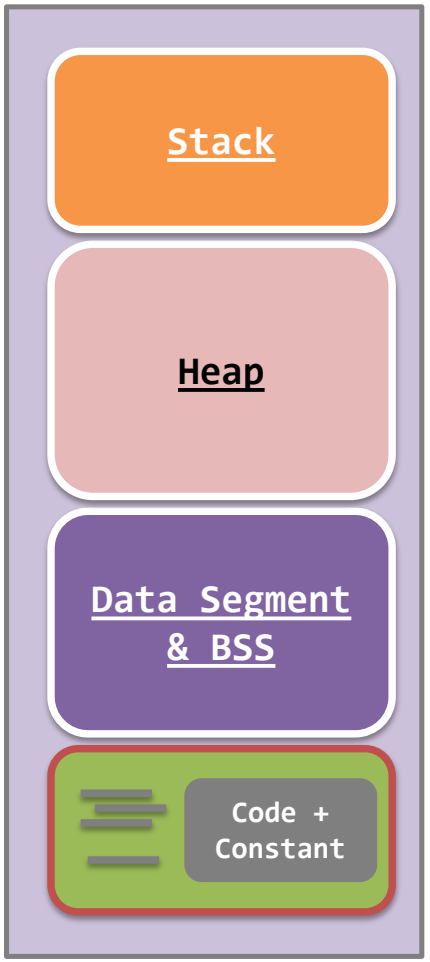
```
$ ./addr
Local variable  = 0xbfa8938c
malloc() space  =  0x915c008
Global variable =  0x804a020
Code & constant =  0x8048550
$ _
```

Note
The addresses are not necessarily the same in different processes



Program code & constants

```
1  int main(void) {
2      char *string = "hello";
3      printf("\"hello\"       = %p\n", "hello");
4      printf("String pointer = %p\n", string);
5      string[4] = '\0';
6      printf("Go to %s\n", string);
7      return 0;
8  }
```



- Constants are stored in code segment.
  - The memory for constants is decided by the program code
  - Accessing of constants are done using addresses (or pointers).

- Codes and constants are both **read-only**.

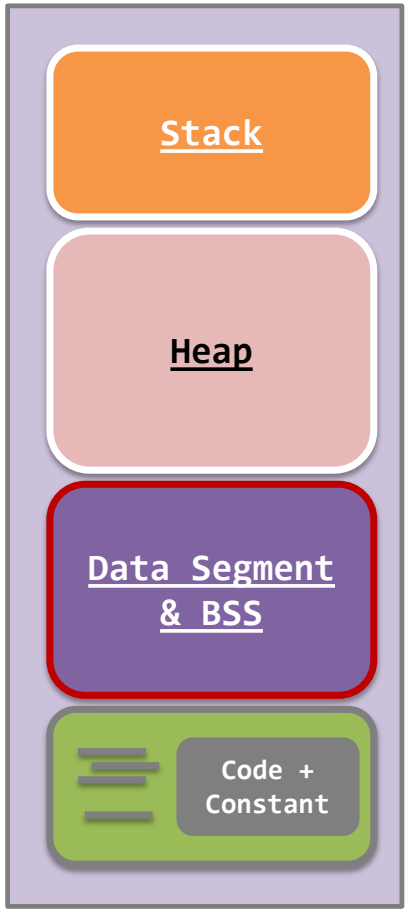
Data Segment & BSS – properties

```
int global_int = 10;
int main(void) {
    int local_int = 10;
    static int static_int = 10;
    printf("local_int  addr = %p\n", &local_int );
    printf("static_int addr = %p\n", &static_int );
    printf("global_int addr = %p\n", &global_int );
    return 0;
}
```

```
$ ./global_vs_static
local_int  addr = 0xbf8bb8ac
static_int addr = 0x804a018
global_int addr = 0x804a014
$_
```

They are stored next to each other.

This implies that they are **in the same segment!**

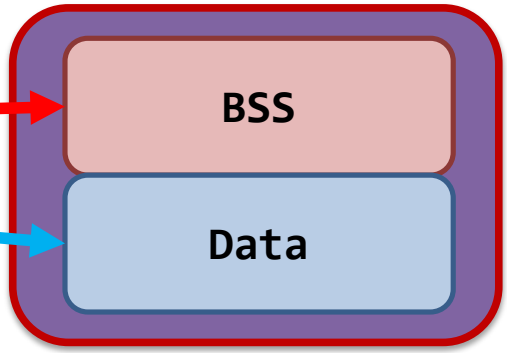


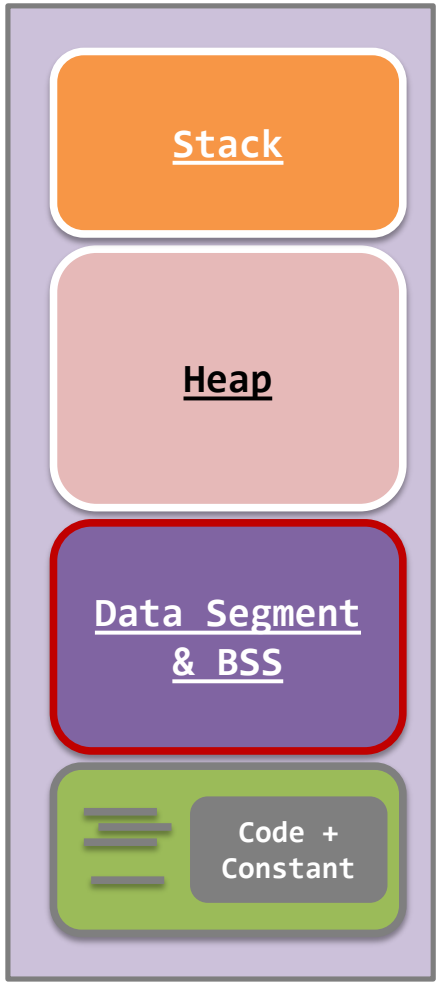
Note: A static variable is treated as the same as a global variable!

Data Segment & BSS – locations

```
 1  int global_bss;
 2  int global_data = 10;
 3  int main(void) {
 4      static int static_bss;
 5      static int static_data = 10;
 6      printf("global  bss = %p\n", &global_bss  );
 7      printf("static  bss = %p\n", &static_bss  );
 8      printf("global data = %p\n", &global_data );
 9      printf("static data = %p\n", &static_data );
10  }
```

```
$ ./data_vs_bss
global  bss = 0x804a028
static  bss = 0x804a024
global data = 0x804a014
static data = 0x804a018
$_
```





内存管理

➢单道程序的内存划分为：操作系统内存和正在执行的程序所占的内存；

➢多道程序中，必须对内存进一步细分出“用户”内存，以满足多进程的要求，细分工作由操作系统完成，称为内存管理；

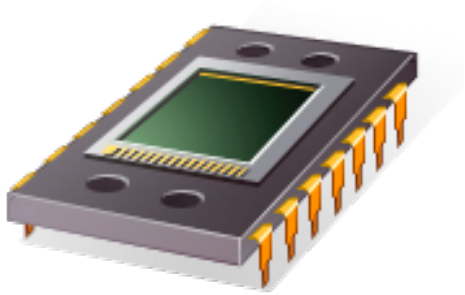
➢必须有效地分配内存来保证适量的就绪进程能占用这些可用的处理器时间；

Frame	**A fixed-length block of main memory.**
Page	**A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.**
Segment	**A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).**

内存管理术语

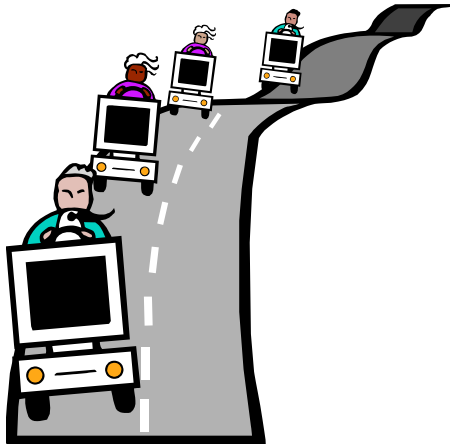
内存管理需求

- 内存管理的需求如下：
  - 重定位
  - 保护
  - 共享
  - 逻辑组织
  - 物理组织



重定位

- 在多道系统中，通常情况下，程序员事先并不知道在某个程序执行期间会有其他哪些程序驻留在内存中；
- 活跃的进程能够被换入换出内存，进而使处理器的利用率最大化；
- 程序换出到磁盘后，下一次换入的时候要放到与换出前相同的内存区域中会很困难。
  - 相反我们需要重定位到不同的内存区域中



重定位

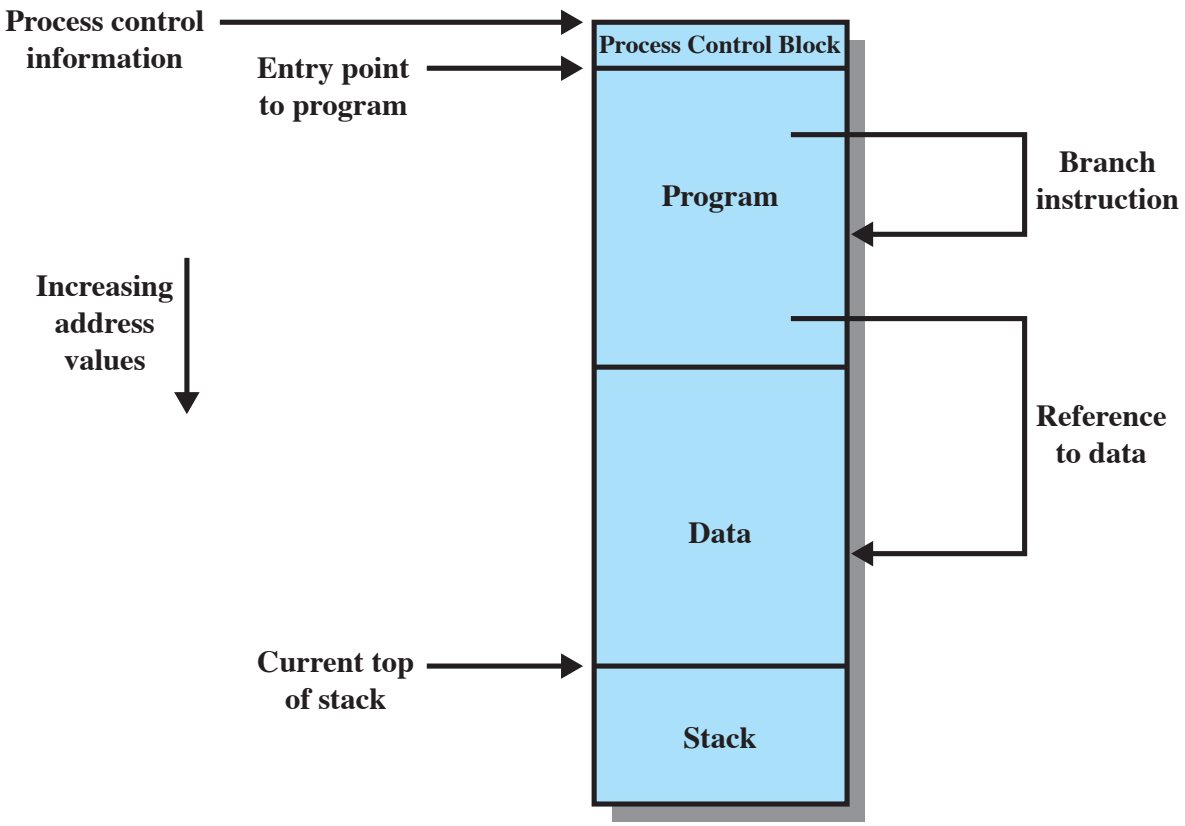
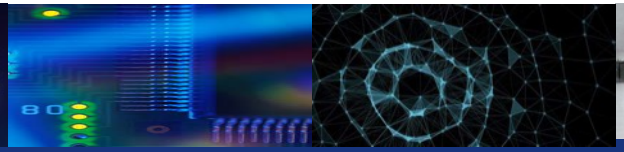


Figure 7.1 Addressing Requirements for a Process

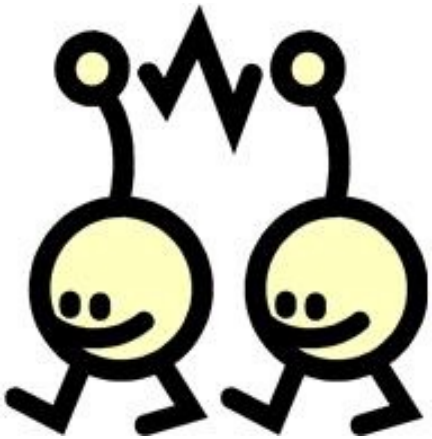


保护

- 某进程以外的其他进程中的程序不能未经授权地访问（进行读写操作）该进程的内存单元；
- 程序在内存中的位置是不确定的；
- 必须在运行时检测进程产生的内存访问，以确保它们只访问分配给该进程的内存空间；
- 支持重定位也支持保护需求的机制已经存在；

共享

- 多个进程正在执行同一个程序时，允许每个进程访问该程序的同一个副本，要比让每个进程都有自己单独的副本更有优势。
- 内存管理系统在不损害基本保护的前提下，必须允许对内存共享区域进行受控访问。
- 用于支持重定位的机制也支持共享；



逻辑组织

- 计算机系统的内存被组织成线性（或一维）的地址空间；

大多数程序被组织称模块

- 模块可以被独立的编写和编译；
- 通过适度的额外开销，可以为不同的模块提供不同的保护级别（只读、只执行）；
- 可以引入某种机制，使得模块可以被多个进程共享。模块级的共享符合用户视角；

满足这些需求的工具是分段，也是本节课要探讨的内存管理技术

物理组织

- 内存、外存两级存储，存储之间的信息移动需要由系统负责

Cannot leave the programmer with the responsibility to manage memory

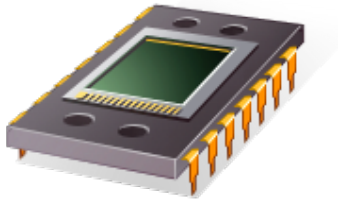
Memory available for a program plus its data may be insufficient

Programmer does not know how much space will be available

overlaying allows various modules to be assigned the same region of memory but is time consuming to program

内存分区

- 内存管理的主要操作是处理器把程序装入内存中运行；
  - 涉及虚存；
  - 虚存是基于分段和分页两种技术，或是其中一种；
- 分区
  - 曾用在很多已经过时的操作系统中；
  - 并不涉及虚存；



内存分区

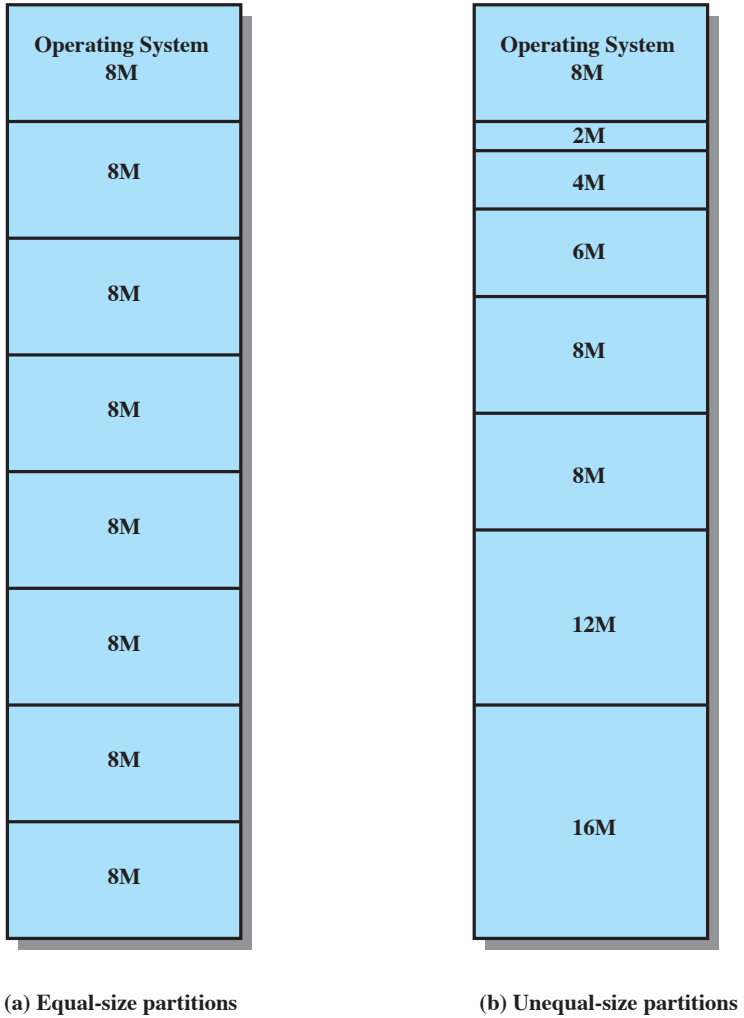
Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to 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.

Table 7.2

Memory Management Techniques

固定分区

- 操作系统占据内存中的固定部分；
- 用户内存空间的最简单方案是对它分区，以形成若干边界固定的区域；
- 分区大小：使用大小相等的分区；使用大小不等的分区；



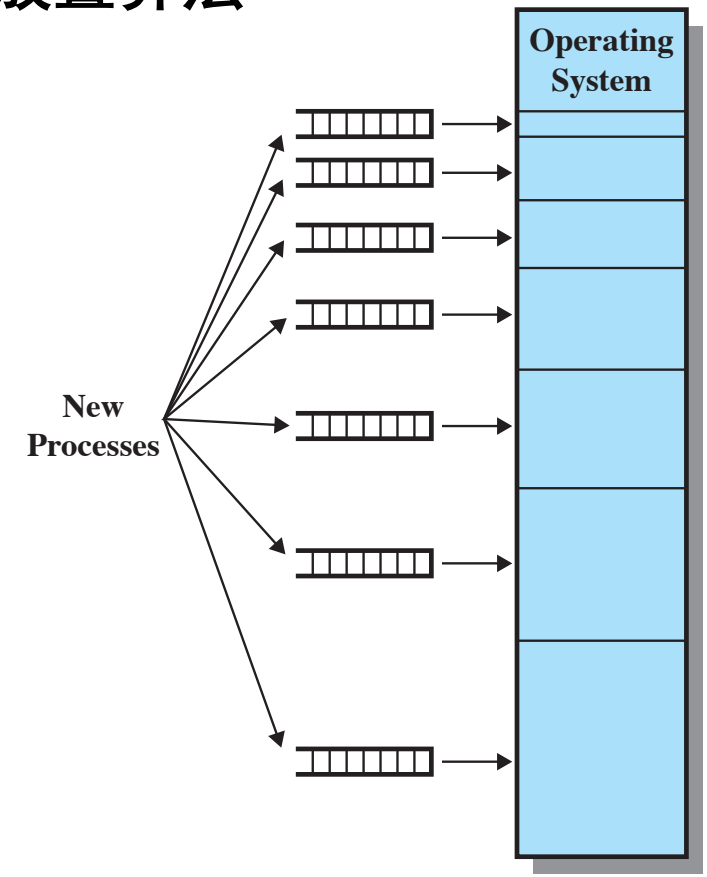
(a) Equal-size partitions　　(b) Unequal-size partitions

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

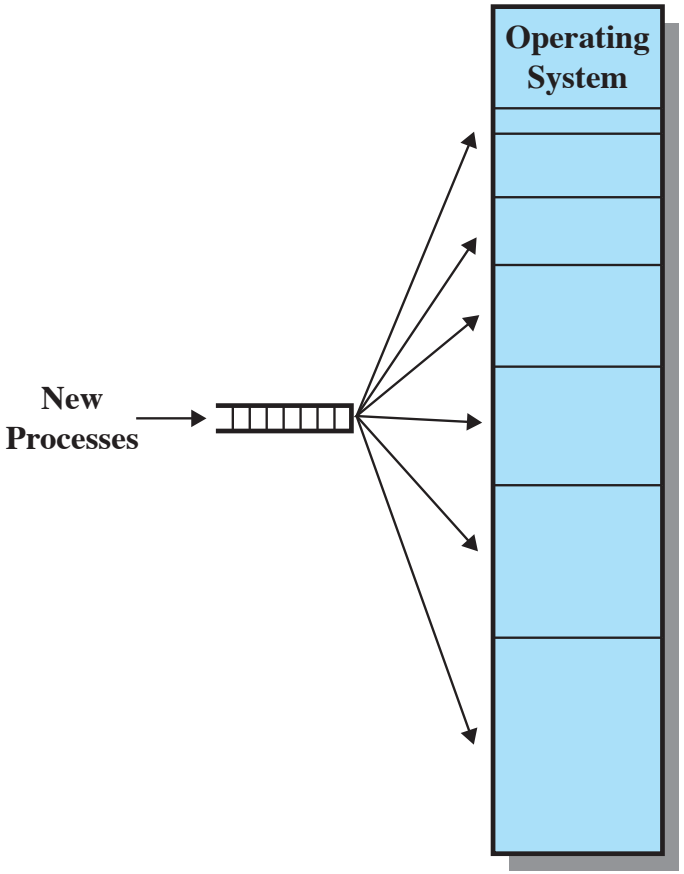
等大小固定分区的缺点

- 程序太大不能放到一个分区中；
  - 程序需要使用覆盖技术设计程序；
- 内存的利用率低；
  - 任何程序，即使很小，都需要占据一个完成的分区；
  - *内存碎片*
    - *由于装入的数据块小于分区大小，因而导致分区内部存在空间浪费的现象*

放置算法



(a) One process queue per partition



(b) Single queue

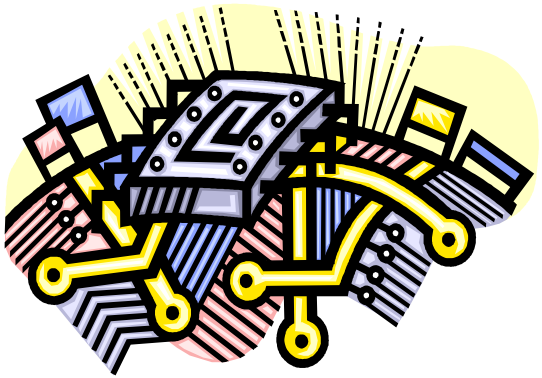
Figure 7.3 Memory Assignment for Fixed Partitioning

固定分区的缺点

- 分区的数量在系统生成阶段已经确定，因而限制了系统中活动（未挂起）进程的数量；
- 由于分区是在系统生成阶段事先设置的，因而小作业不能有效利用分区空间

动态分区

- 对于动态分区，分区的长度和数量是可变的；
- 系统会给进程分配一块与其所需容量完全相等的内存空间；
- 这个技术最早由IBM主机操作系统所使用的；



动态分区

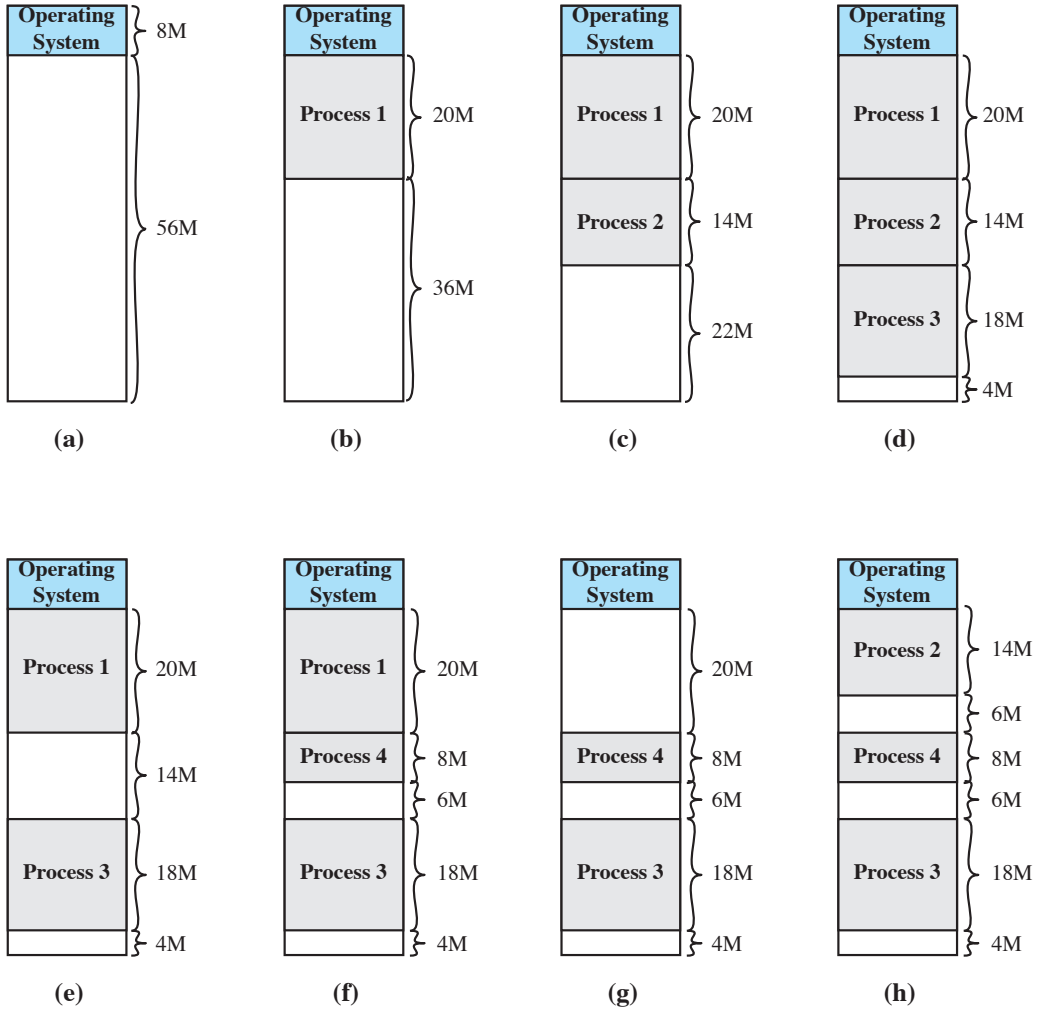


Figure 7.4 The Effect of Dynamic Partitioning

这种分配算法最初不错，但是最终形成很多小空洞，外部碎片；

动态分区

External Fragmentation（外部碎片）

- memory becomes more and more fragmented
- memory utilization declines（内存利用率）

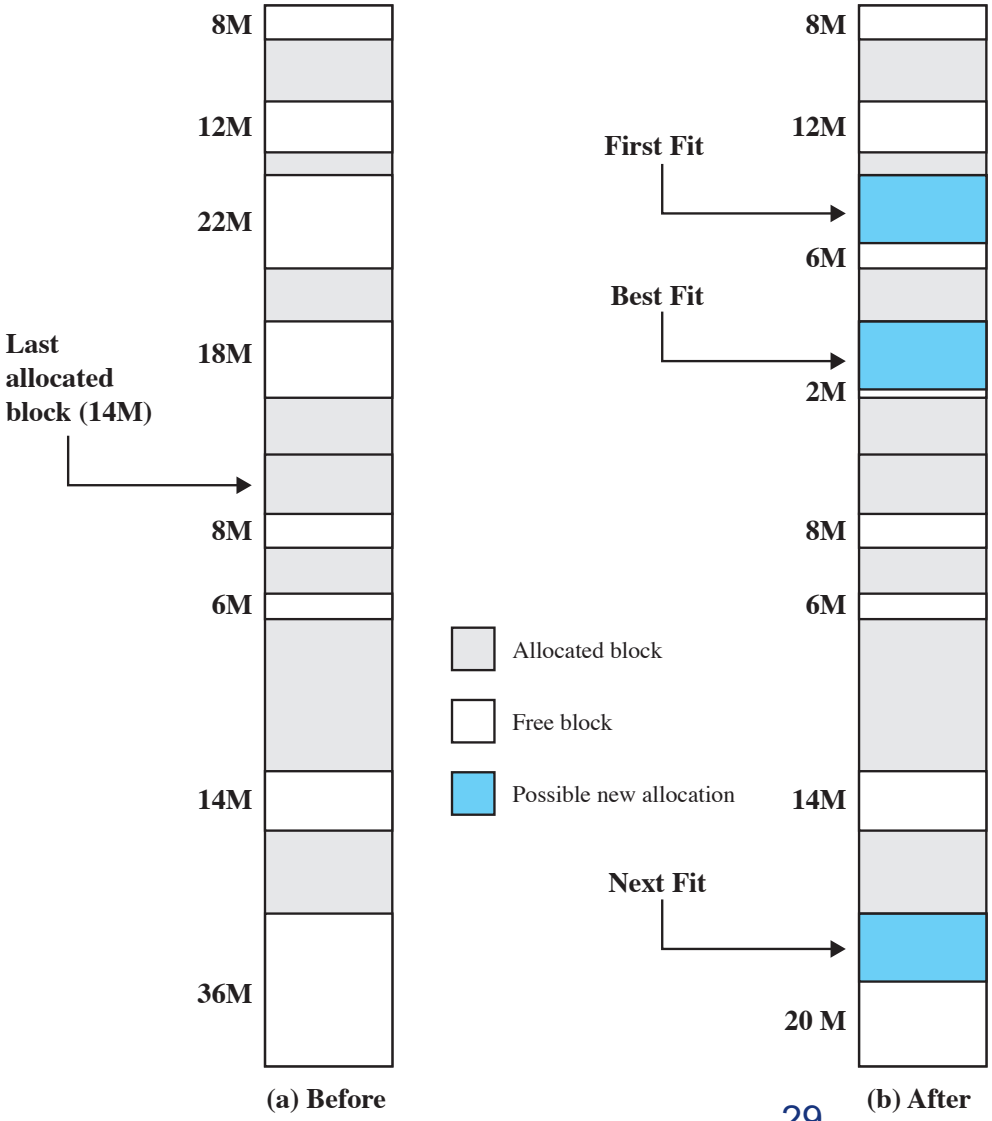
Compaction（压缩）

- technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- free memory is together in one block
- time consuming and wastes CPU time

放置算法

Best-fit	First-fit	Next-fit
• chooses the block that is closest in size to the request	• begins to scan memory from the beginning and chooses the first available block that is large enough	• begins to scan memory from the location of the last placement and chooses the next available block that is large enough

放置算法

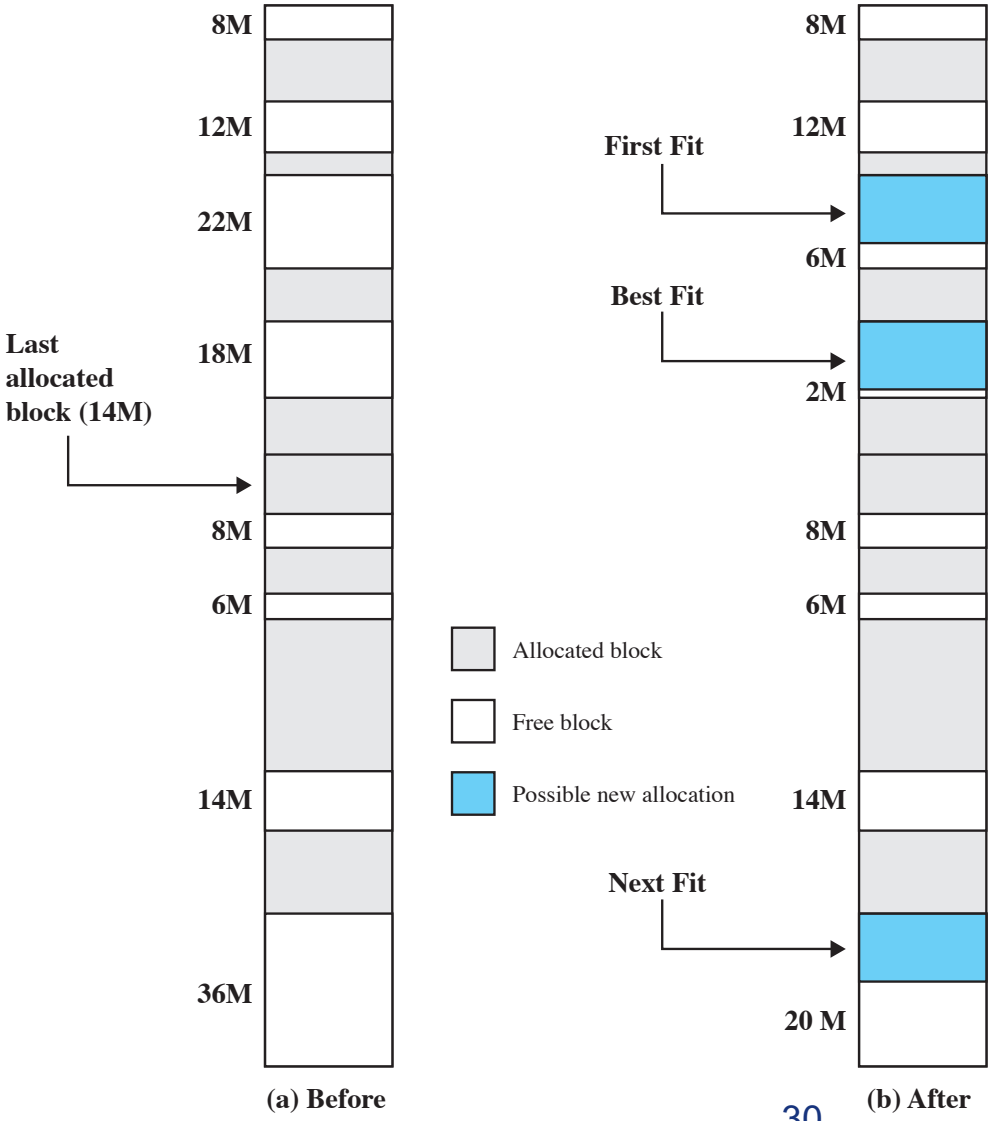


结论： 首次适配算法不仅是最简单的，而且通常是最好和最快的；

首次适配算法会使得内存的前端出现很多小空闲区；

最佳适配通常性能是最差的，产生很多小到无法满足任何内存分配请求的小块，需要更频繁的内存压缩；

置换算法



- 内存中所有进程都可能阻塞；
- 操作系统将一个阻塞进程换出内存，给新进程或者处于就绪-挂起态的进程；
- 操作系统必须决定要替换哪个进程；

伙伴系统（Buddy System）

- 动态分区和固定分区都存在明显的缺点，伙伴系统融合了固定和动态分区；
- 可用于分配的整个空间看成一个块；
- 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

伙伴系统（Buddy System）

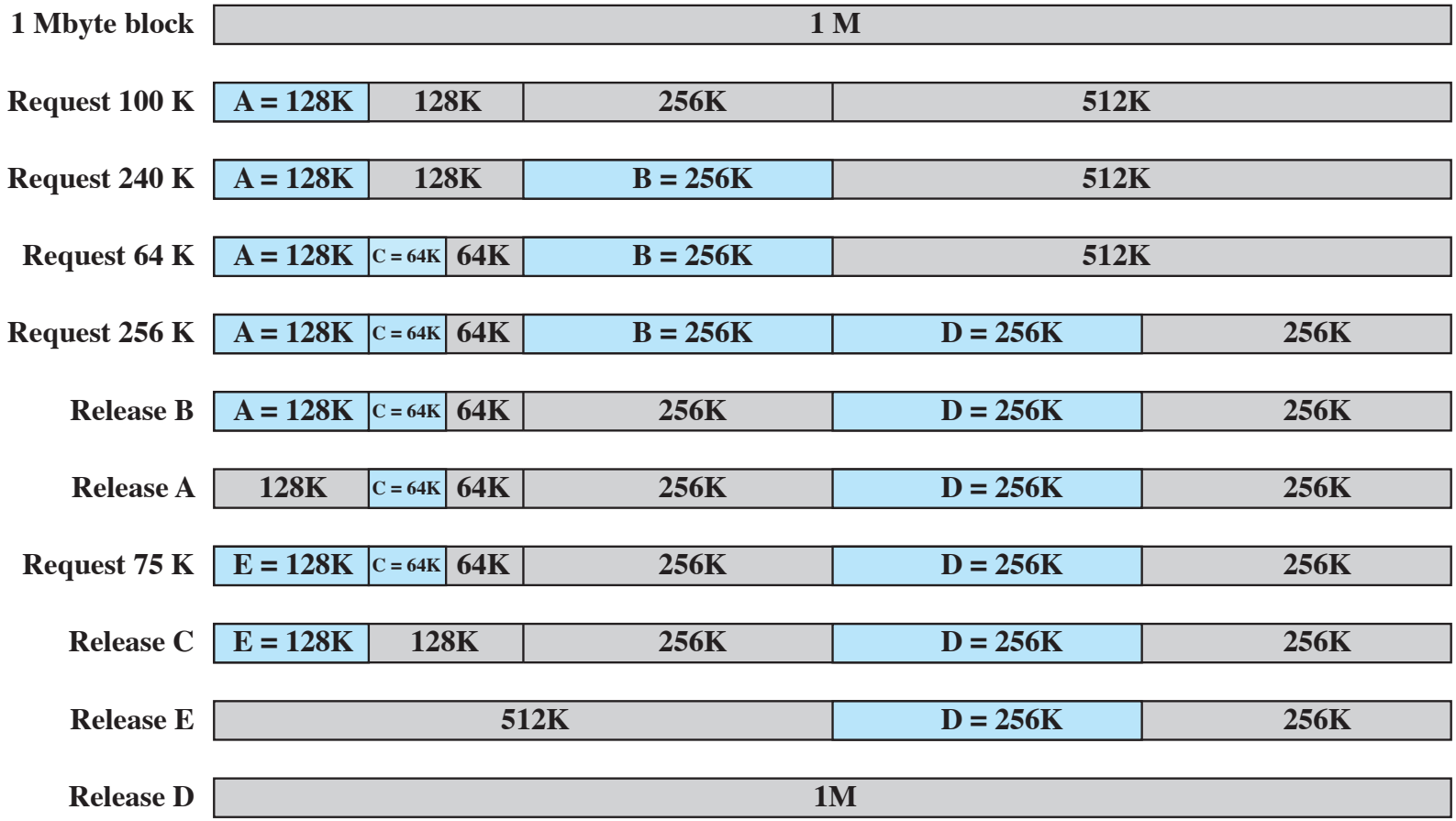


Figure 7.6 Example of Buddy System

伙伴系统的树状表示

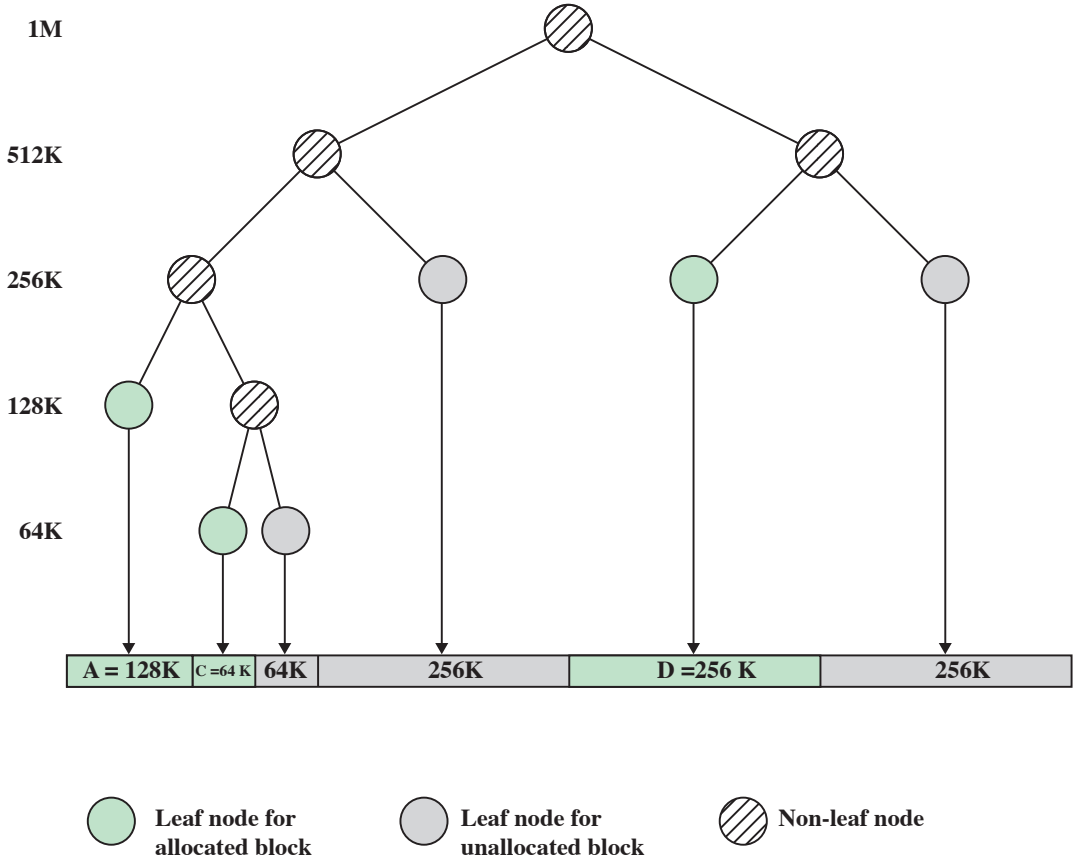


Figure 7.7 Tree Representation of Buddy System

重定位

逻辑地址

- reference to a memory location independent of the current assignment of data to memory

相对地址

- address is expressed as a location relative to some known point

物理或者绝对地址

- actual location in main memory

重定位的硬件支持

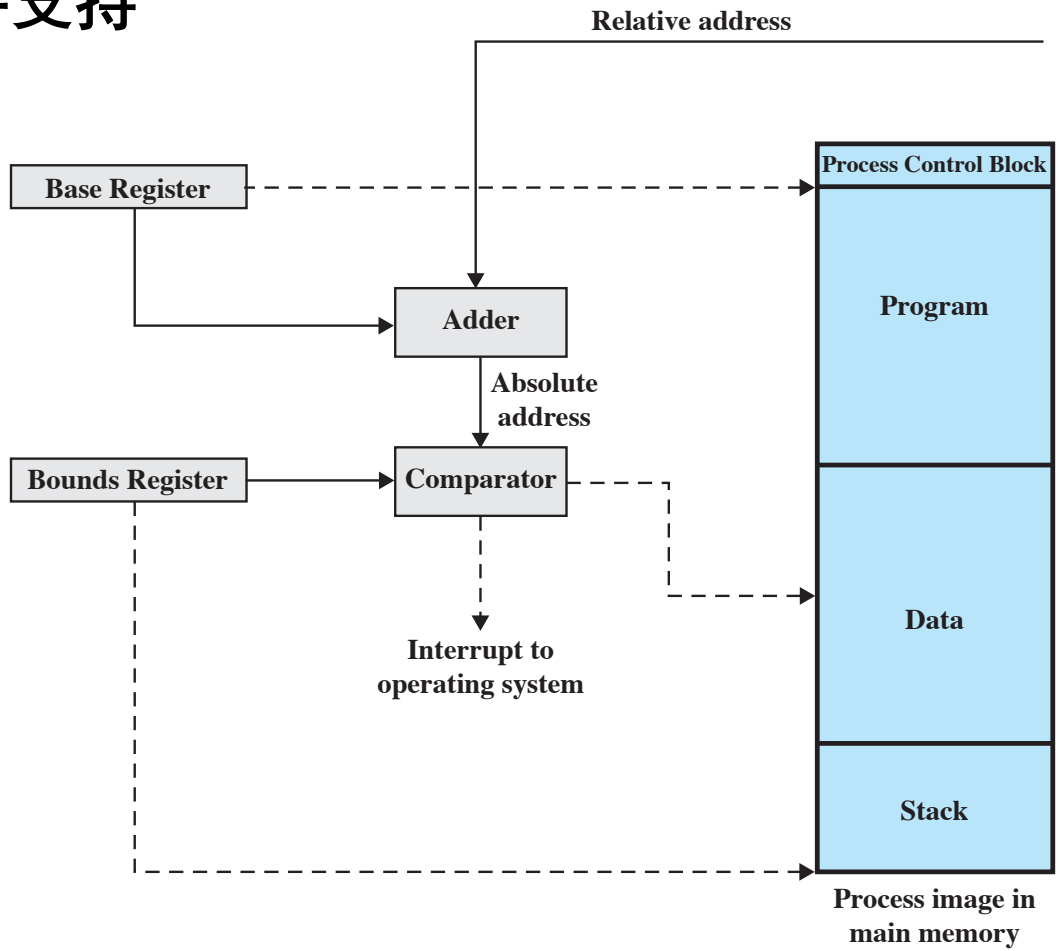


Figure 7.8 Hardware Support for Relocation

分页

- 将内存划分成大小固定、相等的块，且块相对较小
- 每个进程也被分成同样大小的块；
- 仅有每个进程最后一页的一小部分形成的内部碎片，没有任何外部碎片；

Pages（页）	Frames（帧）
• chunks of a process	• available chunks of memory

分页

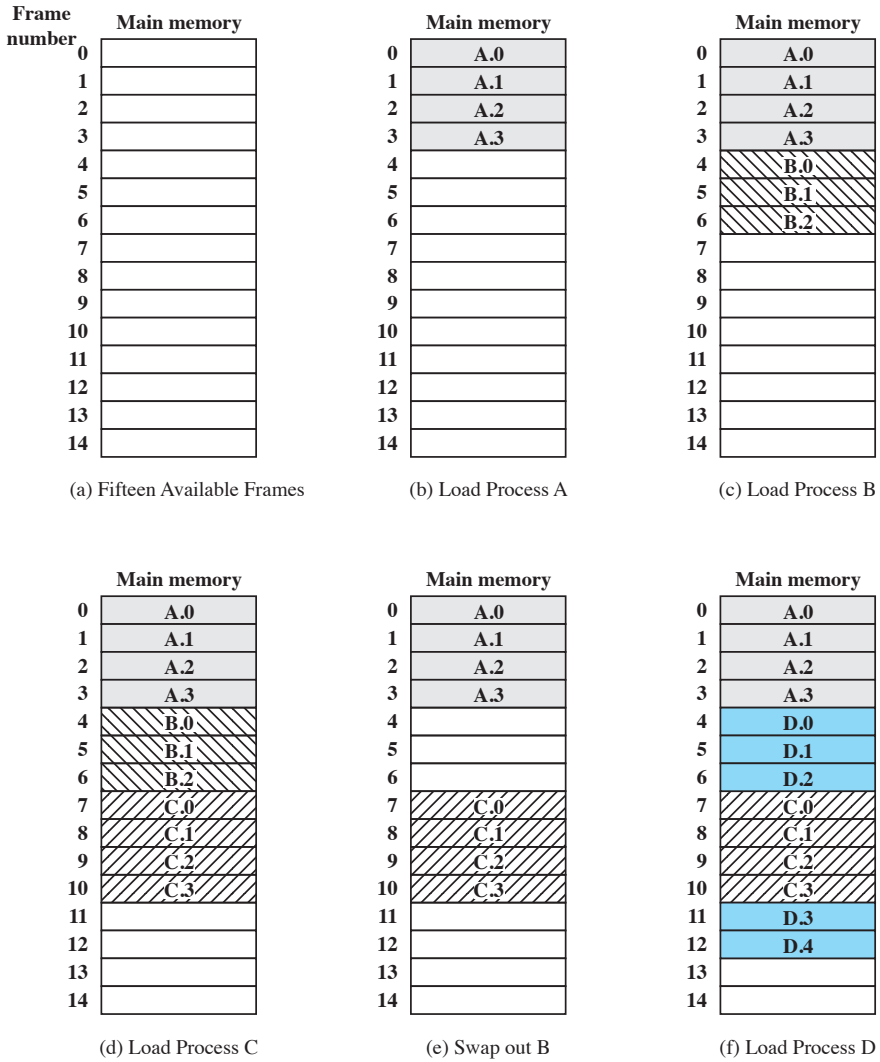
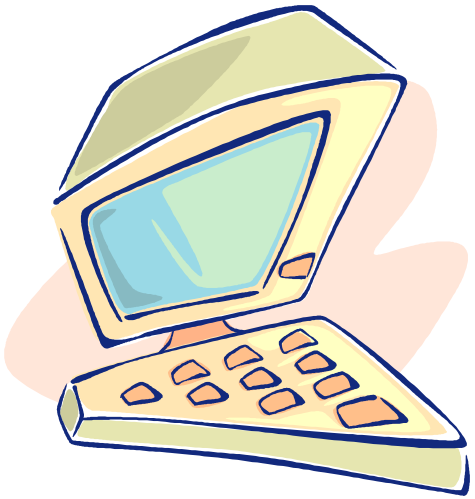


Figure 7.9 Assignment of Process Pages to Free Frames

页表

- Maintained by operating system for each process;
- Contains the frame location for each page in the process;
- Processor must know how to access for the current process (CR3);
- Used by processor to produce a physical address;



页表

- 分页机制类似于固定分区，不同之处在于：采用分页技术的分区相当小，一个程序可以占据多个分区，并且这个分区不需要是连续的；

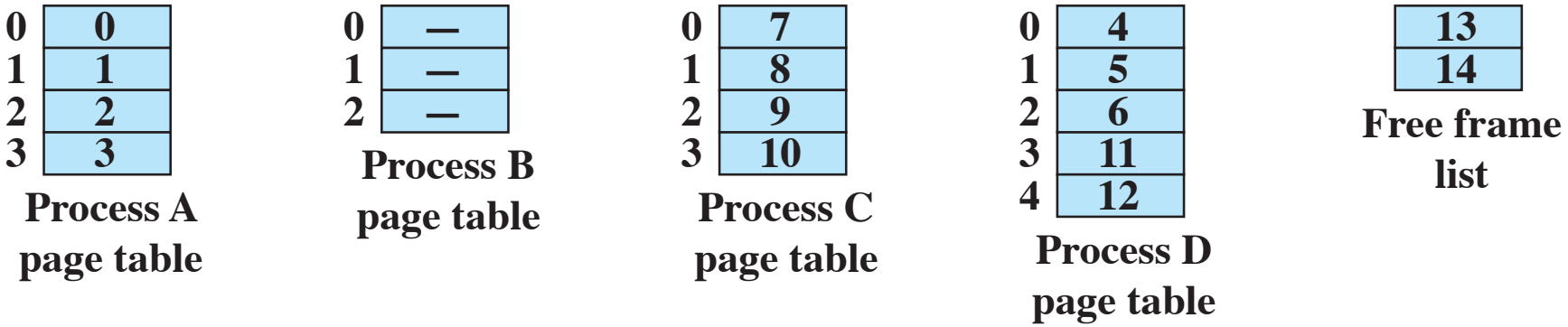


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

页表

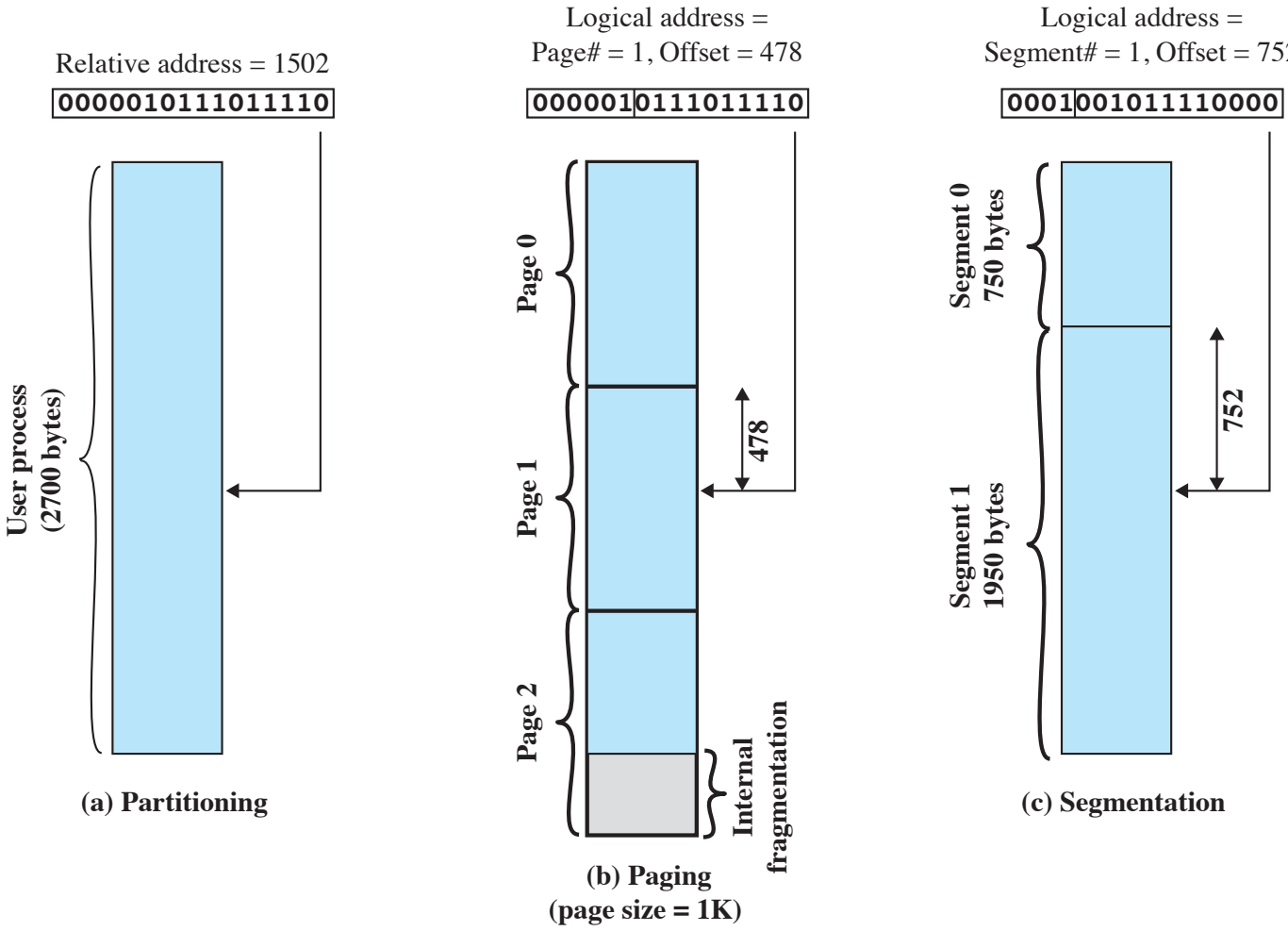
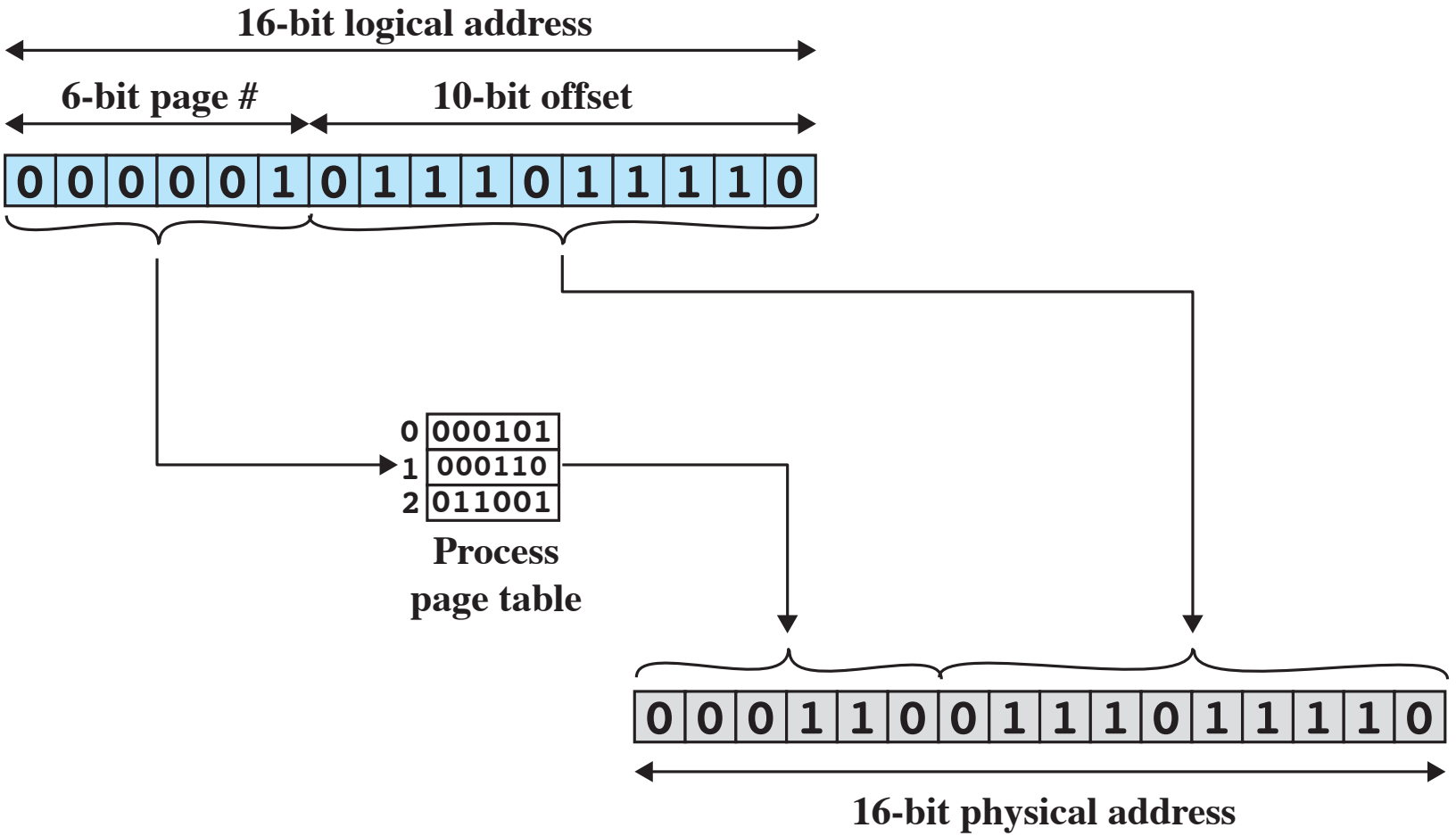


Figure 7.11 Logical Addresses

页表定址



(a) Paging

分段

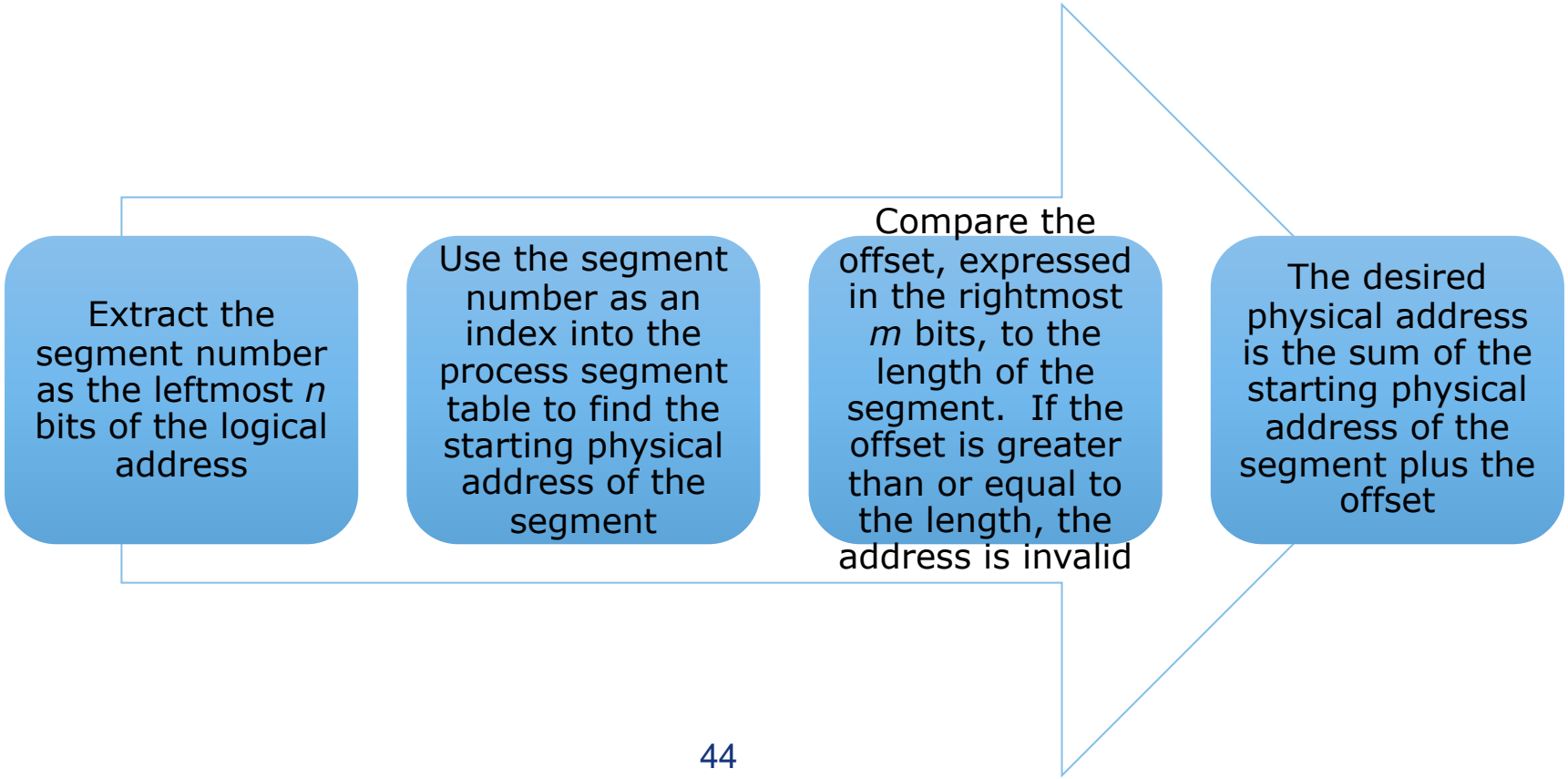
- 程序和其相关的数据划分到几个段（segment）
  - 所有段的长度不一定相等；
  - 段有最大长度；
- 逻辑地址也有两部分构成：
  - 段号
  - 偏移量
- 分段似于动态分区；
- 需要将程序的所有分段都装入内存；
- 一个程序占据多个分区，并且这些分区不要求是连续的；
- 消除了内部碎片，但会产生外部碎片；

分段

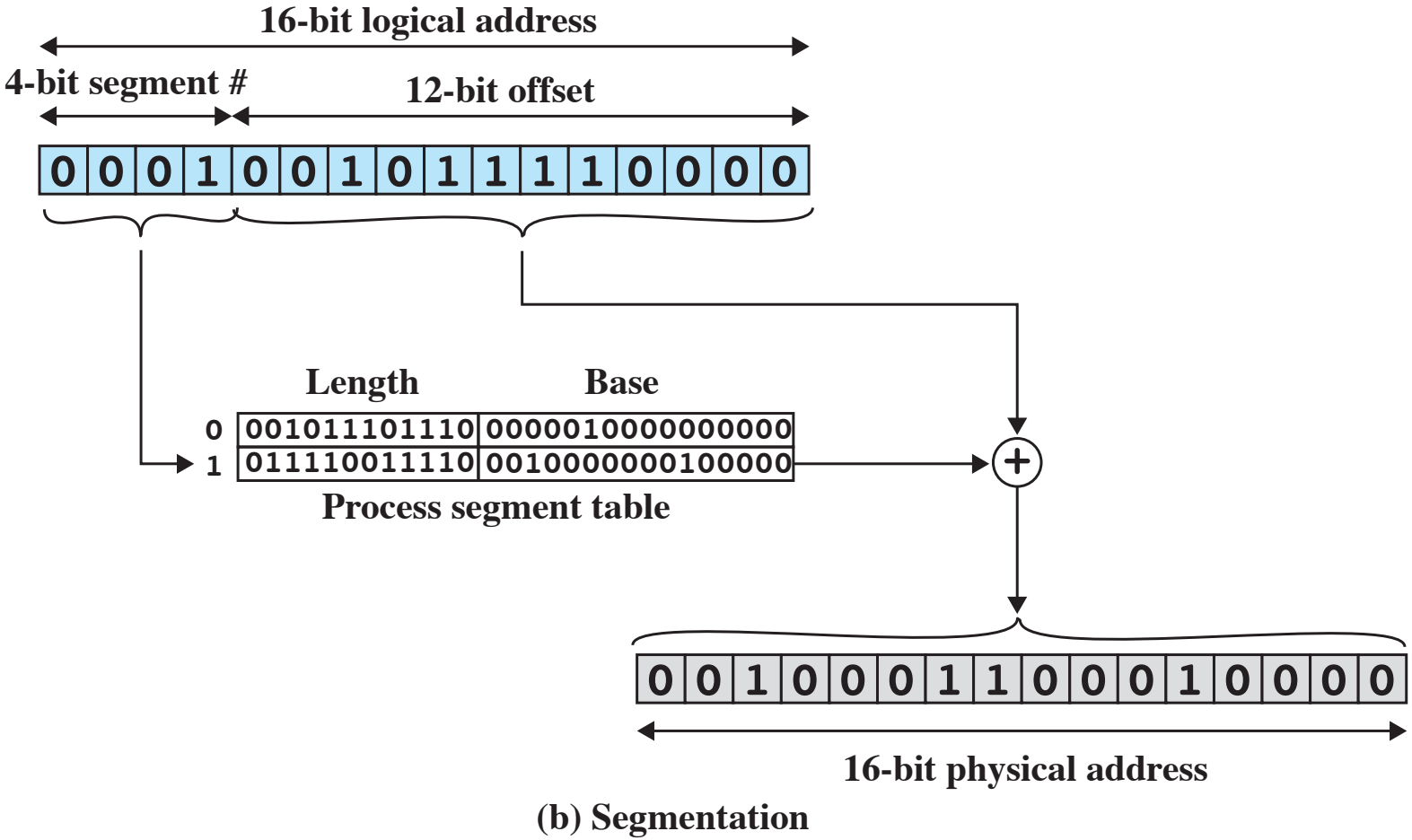
- **Usually visible （可见的）；**
- **作为组织程序和数据的一种方便手段提供给程序员；**
- **程序员或者编译器会把程序和数据指定到不同的分段；**
- **为了实现模块化程序设计的目的，程序和数据可能会进一步划分成多个段；**
  - 这种方法最大的不方便是程序需要知道段的最大长度限制；

地址转换

- Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses
- The following steps are needed for address translation:

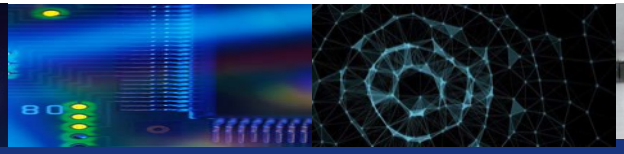


地址转换



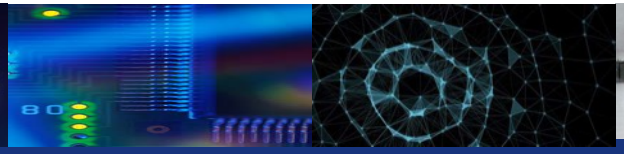
(b) Segmentation

Figure 7.12 Examples of Logical-to-Physical Address Translation



总结

- **Memory management requirements**
  - relocation
  - protection
  - sharing
  - logical organization
  - physical organization
- **Paging**
- **Memory partitioning**
  - fixed partitioning
  - dynamic partitioning
  - buddy system
  - relocation
- **Segmentation**



谢谢