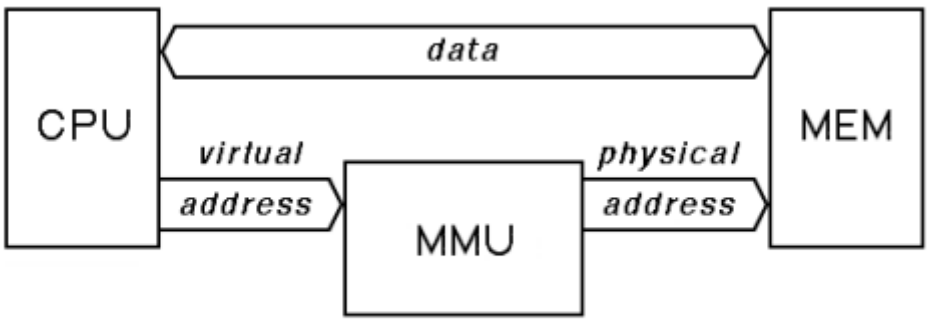
操作系统WEEK 7

1.内存Memory

1. **内存和存储：** 内存是一种用于保存数据的硬件设备。在执行过程中，程序需要先被装入内存才能被CPU处理。通常情况下，程序和数据是存储在辅助存储设备（如硬盘）中，从这些设备读取数据需要花费大量时间，因此，我们引入了内存，一种比硬盘更快的存储设备。故一般的流程为：程序/软件 -> 内存 -> CPU执行。
2. **内存存储单元（main memory unit）：负责处理和管理虚拟内存和物理内存之间的映射关系。** 在内存中，有许多小的"存储单元"，每一个单元都有一个唯一的地址。这些地址用于区分和识别内存中的数据位置。内存地址从0开始，每个地址对应一个存储单元。存储单元的大小可能根据计算机不同，有些计算机按字节进行编址（每个存储单元为一个字节，即8位），有些计算机按字进行编址（每个存储单元可能为多个字节，具体取决于计算机的字长）。



3. **常用的内存单位：** 常用的内存单位包括 KB、MB、GB 等， 而且这些单位都以2为基数，如：2^10=1KB，2^20=1MB，2^30=1GB。比如，一台配备4GB内存的计算机，可存放4x2^30个字节（如果按字节编址）。以字节存储的话，表现为4x2^30个"存储单元"（即小房间）。每个存储单元，都需要一个唯一的地址进行标识，所以至少需要32位（4字节）二进制数进行地址表示（0 ~ 2^32 - 1）。
4. **进程执行与指令操作：**

- 我们使用高级编程语言（如Java、C++、C）所写的代码，在执行前首先要通过编译，将这些代码转化为机器可以执行的指令，这些指令都是用二进制编写，可以直接被CPU执行。
- CPU根据这些指令完成各种任务，例如执行一个简单的加法操作。在这个操作中，指令中直接给出了数据存放的具体地址（物理地址）。

- 然而，实际的情况却是，在编译出机器指令的时候，并不知道该进程最终会被放入到内存的什么位置。因此，这些指令中通常会使用逻辑地址（也叫相对地址）logical address，而不是绝对物理地址。
- 逻辑地址在编译过程中产生，它只表示了数据相对于程序起始位置的相对地址。现实运行时，程序被装入内存后，系统会根据程序在内存中的起始地址，将逻辑地址转化为绝对地址（实际地址）。这种地址计算过程称为“地址重定位”。
- 因此，一旦程序被载入内存，它就知道所有变量/参数在内存中的实际位置（物理地址）。

说明：

- 相对地址：指的是地址的逻辑位置，编译过程中由编译器生成，表示的是数据和代码相对于程序起始位置的偏移量。
- 物理地址：是指数据在内存中的实际地址，运行过程中由逻辑地址加上程序在内存中的起始地址计算得出。

5. 从写程序到程序运行

一个程序从编写完成到能够运行，需要经历编译、链接和加载三个步骤。

1. **编译：** 使用编译器将源代码（通常为某种高级语言编写）编译成一个或多个的目标代码模块。也就是将人类可读的编程语言转化为机器可执行的低级语言（通常为二进制代码）。
2. **链接：** 链接器将上一步产生的一个或多个低级语言的目标文件和必要的库函数链接在一起，生成最终的可执行文件。这个过程主要解决程序内部和外部的符号引用问题，也就是确定这些代码模块在最终的可执行文件中的相对位置。
3. **加载：** 当程序需要被执行时，加载器会负责将可执行文件加载到内存中，并启动程序的执行。

在这些过程中，编译和链接过程在程序执行之前完成，所以在编写代码的时候我们并不知道程序将会被加载到内存的哪个位置，也就无法确定程序的物理地址。因此，编译完的代码中通常包含的是逻辑地址，也就是每段代码相对于整个程序起始位置的偏移量。然后在程序被加载到内存并开始运行时，逻辑地址会根据程序在内存中的具体位置被转化为物理地址。

6. **程序装载到内存：**

程序装载到内存涉及逻辑地址到物理地址的转换。程序的地址在装载到内存时都是相对地址，需要将这些相对地址转换为物理地址才能确保程序的正确执行。存在下面这三种装载方式：

绝对装载： 这种方式在编译或汇编时给出物理地址。在编译的过程中就确定了程序和数据的实际地址。它需要预知程序在内存中的装入位置。

静态重定位： 这种方式在装入时进行地址转换。先确定程序的物理起始地址，然后将这个地址加到所有的逻辑地址（相对地址）上，得到对应的物理地址。所有需要地址转换的操作都在装载时一次性完成。因此，程序装入后，地址就固定，程序运行过程中就不能改变位置，也不能动态申请内存空间。

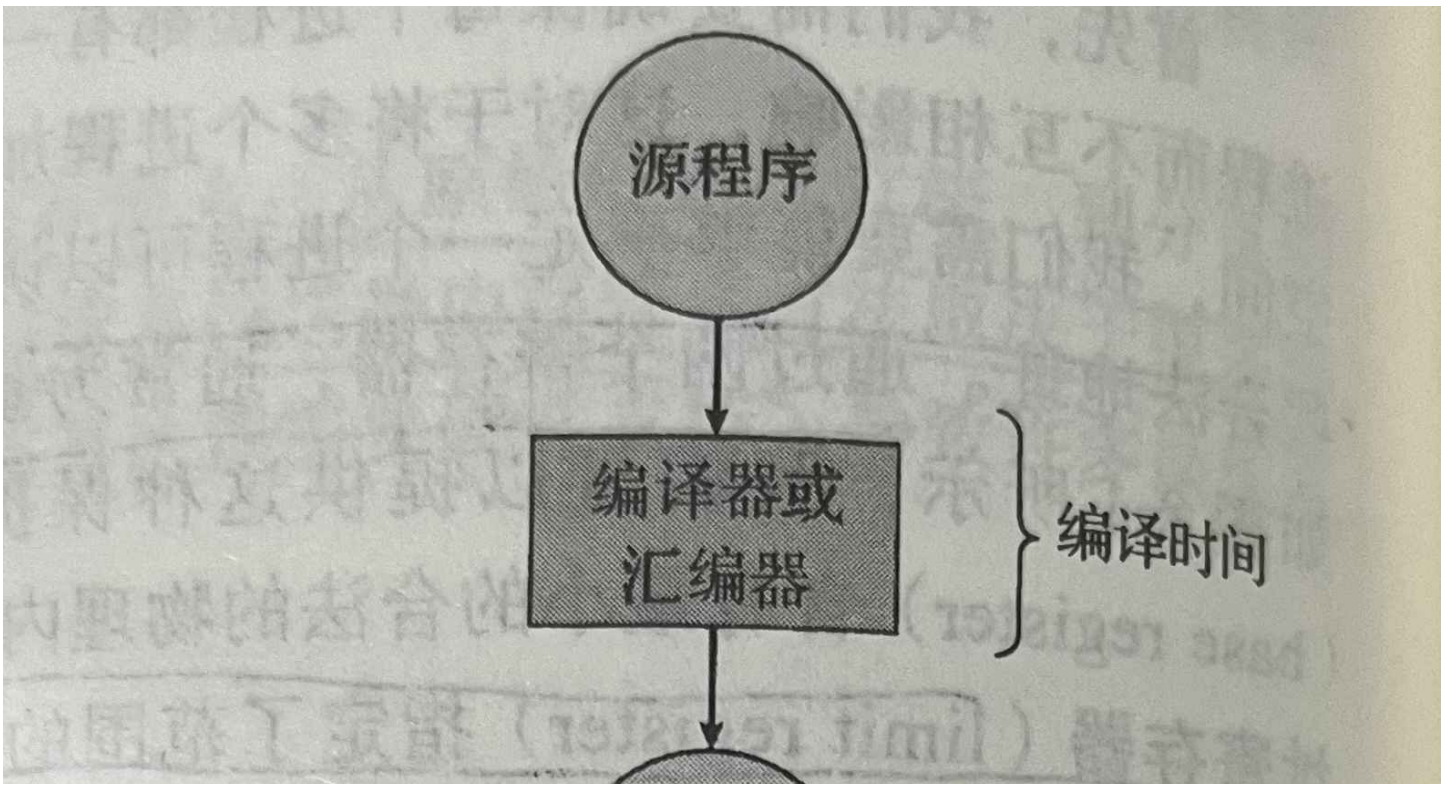
动态重定位： 这种方式在程序执行的过程中进行地址转换。装入模块的地址在程序运行时会动态改变。需要使用一个**重定位寄存器relocation register（存放装入模块在内存存放的起始地址）**来保存程序装载到内存中的起始地址，CPU在执行指令时，会将存储在重定位寄存器中的值加上指令中的地址，得到实际的物理地址。动态重定位方法让程序在内存中的位置可以动态变化，并且由于地址转换是在程序运行时完成的，可以将程序分配到不连续的存储区，并有效实现内存的动态分配和共享。

7. 链接的三种方式

静态链接： 在程序运行之前，将所有的目标模块及它们所需要的库文件链接成一个完整的可执行文件。这个过程会将各个目标模块的逻辑地址结合成完整的逻辑地址空间。一旦链接完成，那么这些地址就固定了，不会在程序运行时改变。

装入时动态链接： 这种方式在装入程序时进行链接，即在装入模块到内存过程中，对模块进行链接。这样可以在需要的时候加载和链接模块，可以更好地利用内存资源。

运行时动态链接： 这种方式在程序运行时进行链接，也就是说直到程序需要某个目标模块时，才对其进行链接。这种方式便于更新和修改模块，以及实现模块的共享。



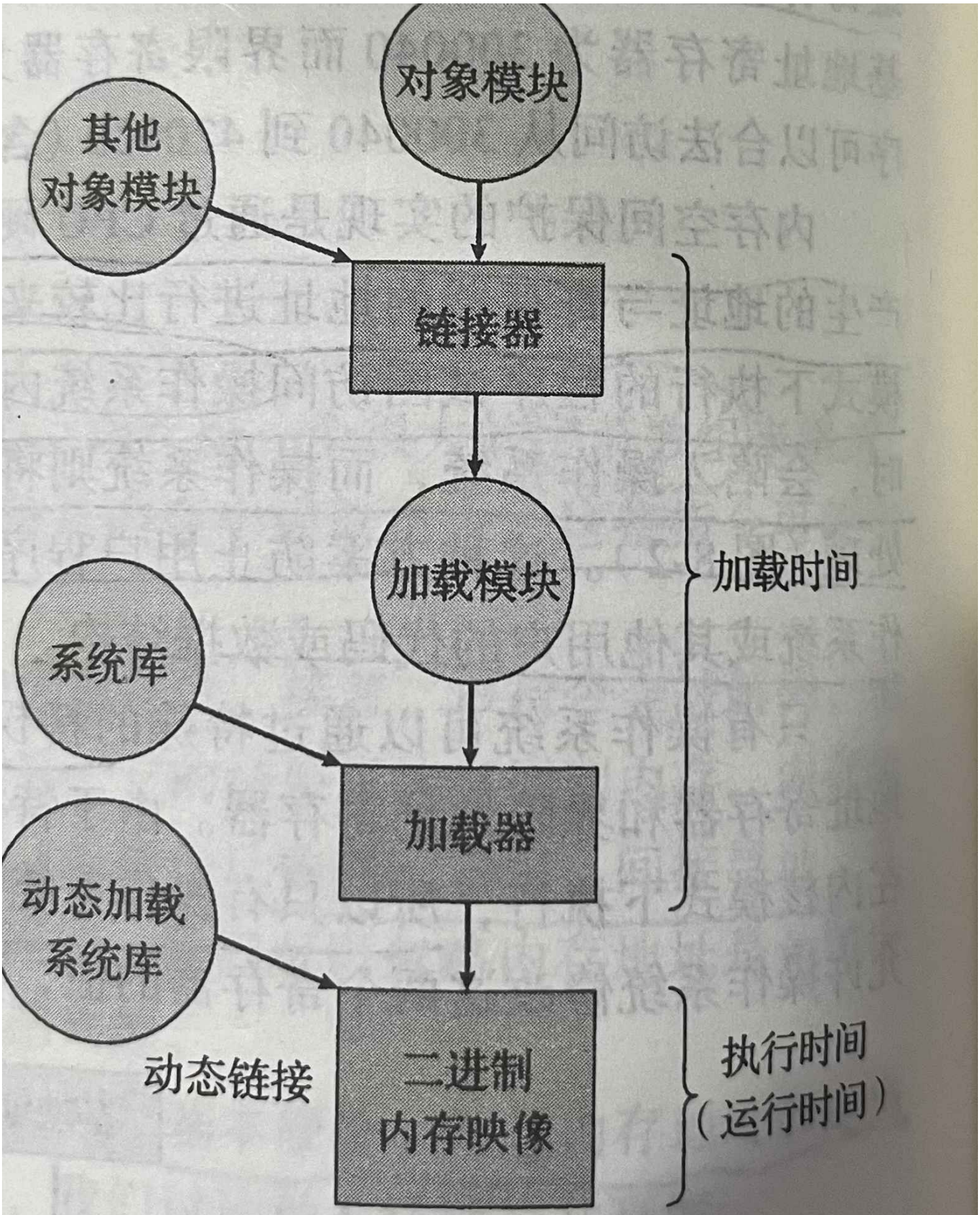


图 8-3　一个用户程序的多步骤处理

2. 内存管理

操作系统(OS)对内存的管理：

1. 操作系统负责内存空间的分配和回收。
2. 操作系统提供了各种技术在逻辑层面上对内存空间进行扩展。
3. 操作系统提供地址转换功能，进行逻辑地址到物理地址的转换。
4. 操作系统提供内存保护功能，确保多进程在内存中运行时互不干扰。

1. 内存保护：

内存保护的主要目的是确保每个进程只能访问自己的内存空间。实现内存保护的两种常见方法包括：

1. 在CPU中使用寄存器存储每个进程的上下限地址。当进程尝试访问某个地址时，CPU会先检查这个地址是否在允许的范围内。
2. 采用基址寄存器和**界址寄存器（限长寄存器）**进行边界检查。基址寄存器存储的是进程的起始物理地址，界址寄存器存储的是进程的最大逻辑地址。只有操作系统可以修改这些寄存器中的数据。

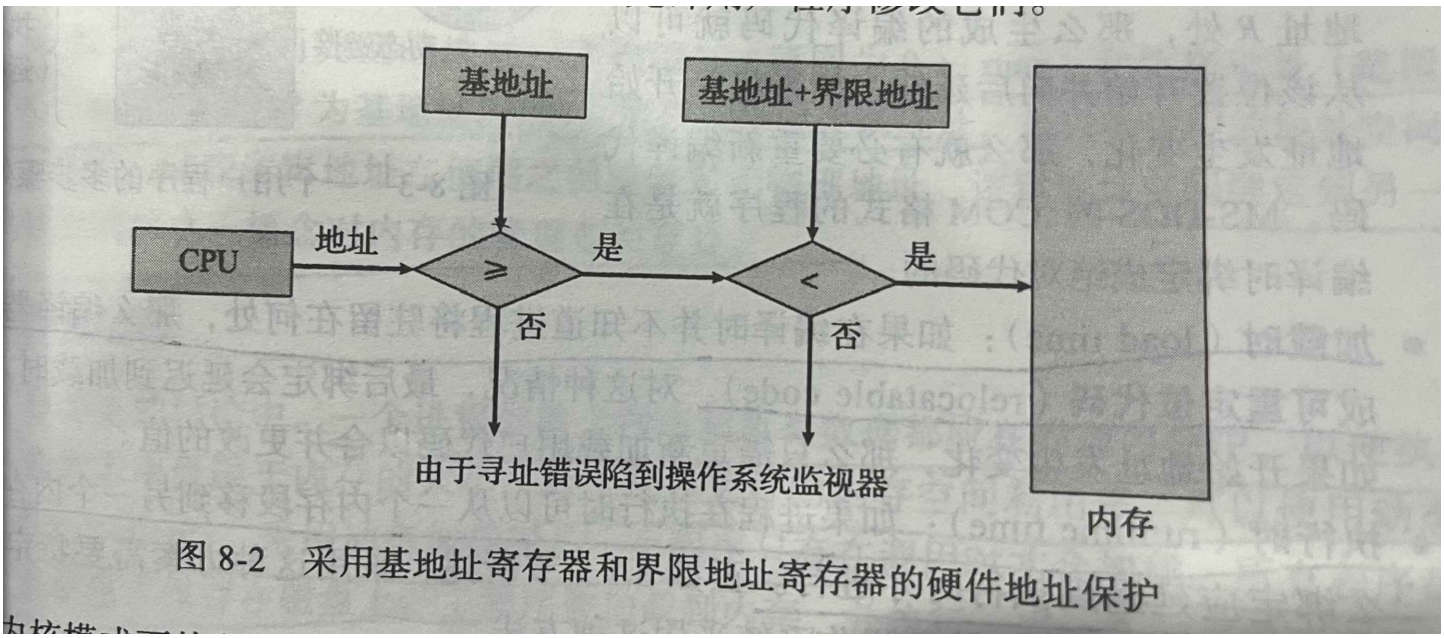


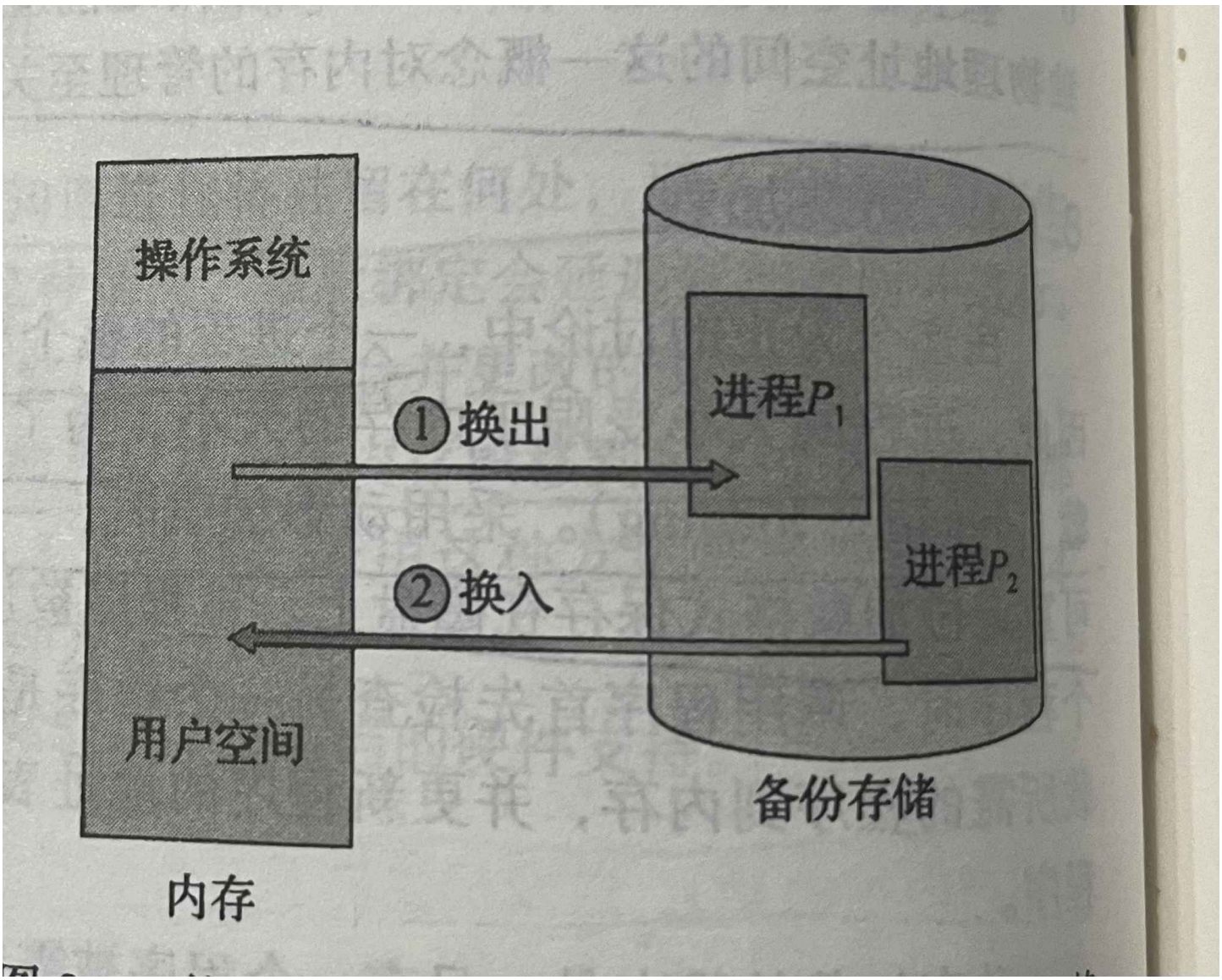
图 8-2　采用基地址寄存器和界限地址寄存器的硬件地址保护

2. 内存转换问题的解决办法：

当物理内存无法满足程序运行所需的内存空间时，可以采用以下方法：

1. **覆盖技术：** 将程序划分为多个段或模块，按需将程序段装入内存。足以放下所有程序的内存被划分为一个固定区和若干个覆盖区。需要常驻内存的程序段放在固定区，不常用的段放在覆盖区，需要时调入，用完后调出。这种方式的缺点是需要程序员手动声明调用结构，过于麻烦，现在很少使用。

2. **交换技术：** 交换技术是操作系统中的一种内存管理技术，主要用于在内存空间紧张时，将内存中的某个进程暂时换出到磁盘中，再将磁盘中已经做好运行准备的进程换入内存。这是一个动态的过程，也称为中期调度。磁盘通常被划分为文件区和交换区两部分。文件区主要用于存放文件，通常采用离散分配方式，只能顺序访问。交换区主要用于存放被换出的进程数据，常采用连续分区方式，可以随机访问，I/O速度更快。交换技术的关键问题包括：在外存中选择何处保存被换出的程序，何时发生交换以及选择换出哪个进程。



3. **覆盖技术与交换技术的区别**：覆盖技术主要应用在同一个程序或进程中，通过不断地将不同段的程序调入内存而实现程序的运行。而交换技术则是在不同的进程或作业之间调整，将内存中的进程换出到磁盘，再将磁盘的进程换入内存。

3. 内存空间的分配与回收：

内存空间的分配方法主要有连续分配和非连续分配。连续分配是指为程序分配的内存必须是一个连续的内存空间。根据分区的大小是否固定，连续分配又可以分为固定分区分配和动态分区分配。

连续内存分配：CONTIGUOUS ALLOCATION

连续内存分配意味着为用户程序分配的内存必须是连续的。以下是三种不同的连续内存分配方式：

1. **单一连续分区分配：** 操作系统将内存分为系统区和用户区。系统区存放系统相关的数据，而用户区则用于存放用户程序相关的数据。这种方式的优点在于其实现简单，不会产生外部碎片，并且可以使用覆盖技术来扩展内存空间。但是它的缺点也很明显，即内存利用率低，只能在单用户单任务的操作系统中使用，且会产生内部碎片，这就是指在为某些进程分配的内存区域中，有些空间没有得到充分利用的情况。
2. **固定分区分配Fixed/Static Partitioning：** 在这种方式中，操作系统会将用户区划分为若干个固定大小的分区，并在每个分区中只装入一个程序。其优点在于实现简单，无外部碎片；缺点在于可能会产生内部碎片，因为不能完美地适应每个用户程序的大小，即使当大程序无法在任何一块分区内满足其内存需求时，也无法改变分区的大小。
3. **动态分区分配Variable/Dynamic Partitioning：** 在动态分区分配中，操作系统将用户区进行动态的不等大小的划分，以适应不同进程的内存需求。操作系统会维护一个数据结构（如空闲区域表）来记录和管理各个分区的使用情况，以实现内存的动态分配和回收。在用户程序装入内存时，操作系统会找到一个空闲区域能满足需求的最小区域进行分配。虽然这种方式比较复杂，但是它比单一连续分区和固定分区分配更加灵活，并且减小了内部碎片的发生。在动态分区分配方式下，内存分区的大小和数量都是可变的。当进程装入内存时，根据进程的大小动态地建立分区，并确保产品的大小正好满足进程的需要。整个过程中，操作系统会使用数据结构（如空闲区表）来记录可用内存的使用情况。

分区的选择：当有多个空闲分区都能满足需求时，可以使用动态分区分配算法来决定使用哪个分区进行分配。

分区的分配与回收：如果进程的大小小于空闲分区的大小，则直接修改空闲区表中的数据（即，大小和地址更新为进程的大小）。如果进程的大小小于空闲分区的大小，则在表中增加对应的条目。在

进行回收操作时，会将相邻的空闲分区和回收的分区合并，并修正相应的表项。

内部和外部碎片Internal fragmentation external fragmentation：内部碎片是指分配给某个进程的内存区域中，有些空间没有被使用。而外部碎片则是指内存中的一些小块空闲分区太小而无法被利用。尽管这些小块空闲分区的总计空间可能足够一个进程使用，但由于这些空闲分区不连续，因此不能被进程使用。这种问题可以使用内存紧缩技术（compaction）来解决，即通过将所有的进程移动到内存的一端，将所有的空闲分区移动到另一端，从而形成一个连续的空闲分区，避免了外部碎片的产生。

动态分区分配算法

1. **首次适应（First-fit）：**首次适应策略是从空闲分区表的最低地址开始查找，找到第一个大小能满足要求的空闲分区进行分配。其缺点是可能会留下许多小的几乎不可用的内存块，这种小的内存块就是外部碎片。
2. **最佳适应（Best-fit）：**最佳适应策略是在所有可用的空闲分区中找到大小最适合进程的那一个进行分配。它的目的是为了留下尽可能大的连续内存空间以满足可能到来的大进程。但是，这种策略可能会导致大量的小块内存分区，从而增加外部碎片的数量。
3. **最坏适应（Worst-fit）：**最坏适应策略是在所有可用的空闲分区中找到最大的那一个进行分配。虽然这样可以保留更大的连续内存空间，但这种策略也可能会导致大块的内存空间被过快的耗尽，导致后续的大进程不能得到满足。
4. **近邻适应（Next-fit）：**近邻适应策略是为了降低搜索的开销，每次不从头开始查找，而是从上次分配结束的位置开始查找到满足进程需求的空闲分区进行分配。这种策略减少了查找时间，但是可能会导致内存的分布不均，也可能生成大量的外部碎片。

连续内存分配灵活性低。

非连续内存分配：NON-CONTIGUOUS MEMORY ALLOCATION

在非连续内存分配方式下，一个进程可以被分配到很多分散的内存空闲区域，避免了连续分配方式经常出现的外部碎片问题。相较于连续分配方式，非连续分配方式具有更好的灵活性，特别是在处理大型和动态改变大小的进程时。

非连续内存分配虽然在理论上可以提高内存的利用率，但在实际实现中，面临的主要挑战是需要复杂的内存管理数据结构来跟踪内存分配情况，以及较高的CPU开销来处理额外的内存管理操作。

内存和进程的划分：

在段页式存储管理系统中，物理内存被分为许多固定大小的块，这些被称为“框”或“帧”。同时，进程被划分为和框相同大小的块，这些被称为“页”。进程A可能被分配到区域2，在该区域中的最后一个页面可能没有被完全使用，因此页框不能太大，以避免产生过大的内部碎片。如果页面太大，会导致内部碎片的产生。为了减少内部碎片，可以将页面大小设置得较小，使得进程的每个页面可以分别放置在不同的页框中。

地址转换：

将进程划分为页后，需要实现逻辑地址到物理地址的转换。这可以通过重定位寄存器来实现。例如，进程A的逻辑地址是在地址空间中散布在不同的页中，当CPU执行指令时，会将逻辑地址与相应的页的起始地址相加，然后加上偏移量来找到要访问的位置。偏移量是逻辑地址中的偏移部分，计算页号时，需要将逻辑地址除以页面大小，然后再加上偏移量。举例： 如果逻辑地址为80，页面大小为50，计算页号和偏移量：页号 = 80 / 50 = 1，偏移量 = 80 % 50 = 30。因此，逻辑地址80对应的页号为1，偏移量为30。

页号计算方法： 为了方便计算页号和页内偏移量，通常会将页面大小设置为2的整数次幂，这样可以用二进制位来表示逻辑地址。例如，假设逻辑地址用32位二进制表示，页面大小为4KB（即2^12），那么一个页可以有4096个字节。页面的逻辑地址由页号和页内偏移量组成，页号是逻辑地址除以页面大小的商，页内偏移量是逻辑地址除以页面大小的余数。

页号page-number和页内偏移量offset是内存管理中重要的概念。

1. 页号： 在页式存储管理中，逻辑地址通常由页号和页内偏移量组成。页号用来标识页面在内存中的位置。当CPU需要访问内存时，通过逻辑地址的页号可以找到相应的物理页面。每个页面都有一个唯一的页号，用来标识该页面在内存中的位置。
2. 页内偏移量： 页内偏移量是指逻辑地址中指定页内部位置的偏移量。它表示了页面中某个字节相对于页面起始位置的偏移量。当CPU访问内存时，会根据逻辑地址的页内偏移量确定需要访问的具体字节。页内偏移量的大小取决于页面的大小，通常用二进制数表示。

在给定的逻辑地址范围内，例如8192~12297，可以将这个范围的逻辑地址用二进制表示。然后根据逻辑地址的分布情况，可以确定每个地址对应的页号和页内偏移量。

例如，如果逻辑地址是8192，转换为二进制是00000000000000000000000100000000，假设页大小为4096字节（2^12），那么逻辑地址的前12位表示页号，后面的位数表示页内偏移量。页号为000000000000000001，即1，页内偏移量为000000000000，即0。

如果一个内存元素中有M位用于表示页号，那么系统最多可以管理2^M个页面。

Paging *Example* for a 32-byte memory with 4-byte pages

- the size of the logical address space is **2^m = the process size**
- the **page size** is **2^n bytes**

page number	page offset
p	d
m -n	n

<page-number, offset>

<m - n, n>

Here, in the logical address: ***m* = 4 and *n* = 2;**

m - n = page number = 2

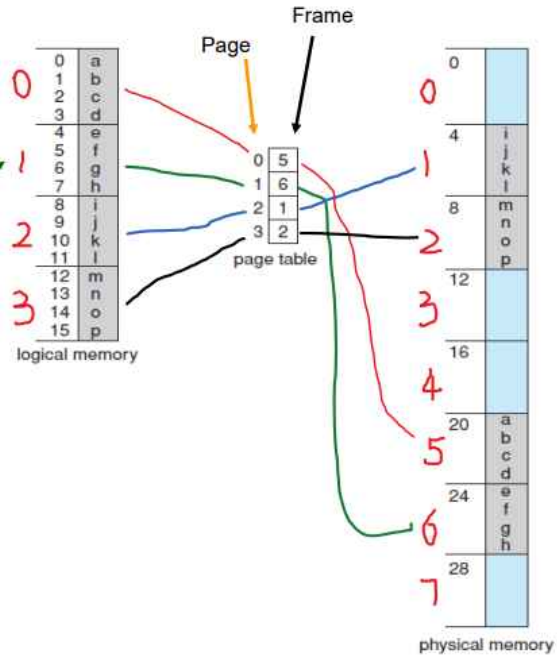
no. of pages = 2^4 / 2^2 = 2^2 = **4 pages** – **0, 1, 2, 3**

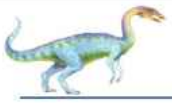
Number of pages the process is divided = Process size / Page size

page size = frame size = 4 bytes

32-byte physical memory

physical memory / frame size = 32 / 4 = 8 frames – 0, …, 7





Paging - *Example*

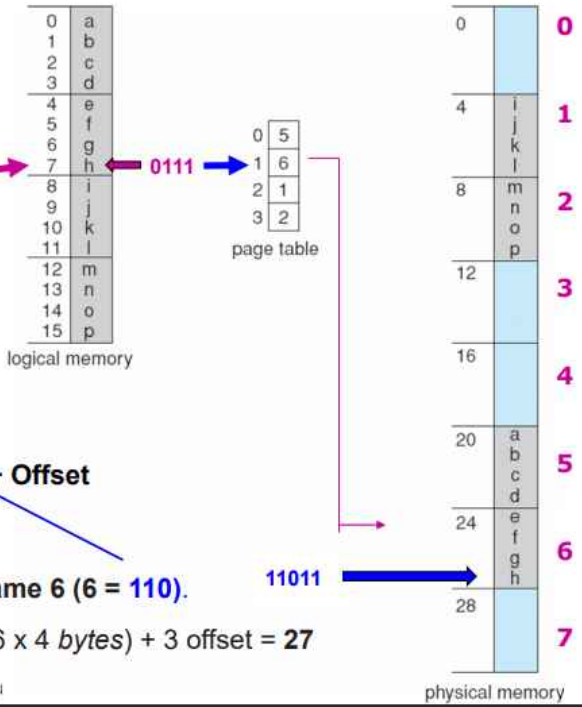
Page size = 4 bytes

Logical address space = 4 pages

Physical address space = 8 frames

Logical address 7: < 1, 3 > = **0111**

Physical address 27: < 6, 3 > = **11011**



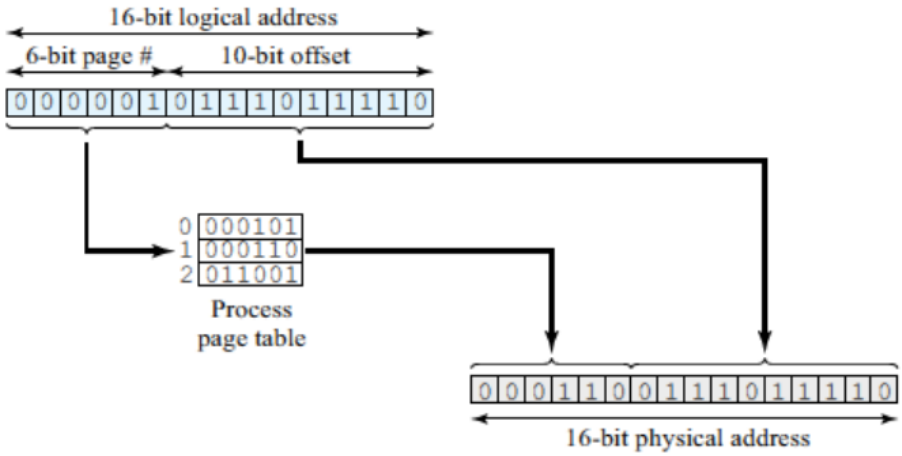
Physical memory address is: **(Frame # * Page size) + Offset**

Logical address **7** (page 1 offset 3)

- according to the page table, **page 1** is mapped to **frame 6 (6 = 110)**.
- Physical memory address for logical address 7 is : (6 x 4 *bytes*) + 3 offset = **27**

Paging - *Example*

Suppose the **logical address 0000010111011110**, which is **page number 1**, **offset 478**.
Suppose this page is residing in main **memory frame 6 = binary 000110**.
Then the physical address is frame number 6, offset 478 = **0001100111011110**



页表page table是管理进程内存分配的重要数据结构，具体特点如下：

1. 每个进程对应一个页表： 每个进程都有自己的页表，用来存放该进程的页面在内存中的地址信息。
2. 页表顶： 每个进程的页表都有一个页表顶，用来记录页表项的起始地址。这样可以根据页表顶和页表项的偏移量来访问具体的页表项。
3. 页表项组成： 每个页表项通常由页号和块号组成。页号用来标识页面的逻辑位置，块号则指向页面在内存中的实际存放位置。
4. 页表记录页面和内存块的对应关系： 页表记录了进程的页面和实际内存块之间的对应关系，使得操作系统可以根据页表来进行内存的访问和管理。
5. 页表项长度相同： 每个页表项的长度通常是相同的，这样可以方便地对页表进行管理和访问。
6. 内存块大小决定页表项数量： 内存块大小决定了每个页面所占用的内存空间，从而决定了页表项的数量。例如，如果内存大小为4GB，页面大小为4KB，那么总共有2的20次方个内存块，需要20位来表示每个内存块的位置。
7. 页表项连续存放： 页表项通常会连续地存放在内存中，这样可以通过页表顶和偏移量快速找到所需的页表项。
8. 隐式页号： 由于页表项连续存放，可以通过页表顶和偏移量来快速找到所需的页表项，因此页表项中的员号通常是隐式的，不需要额外存储。

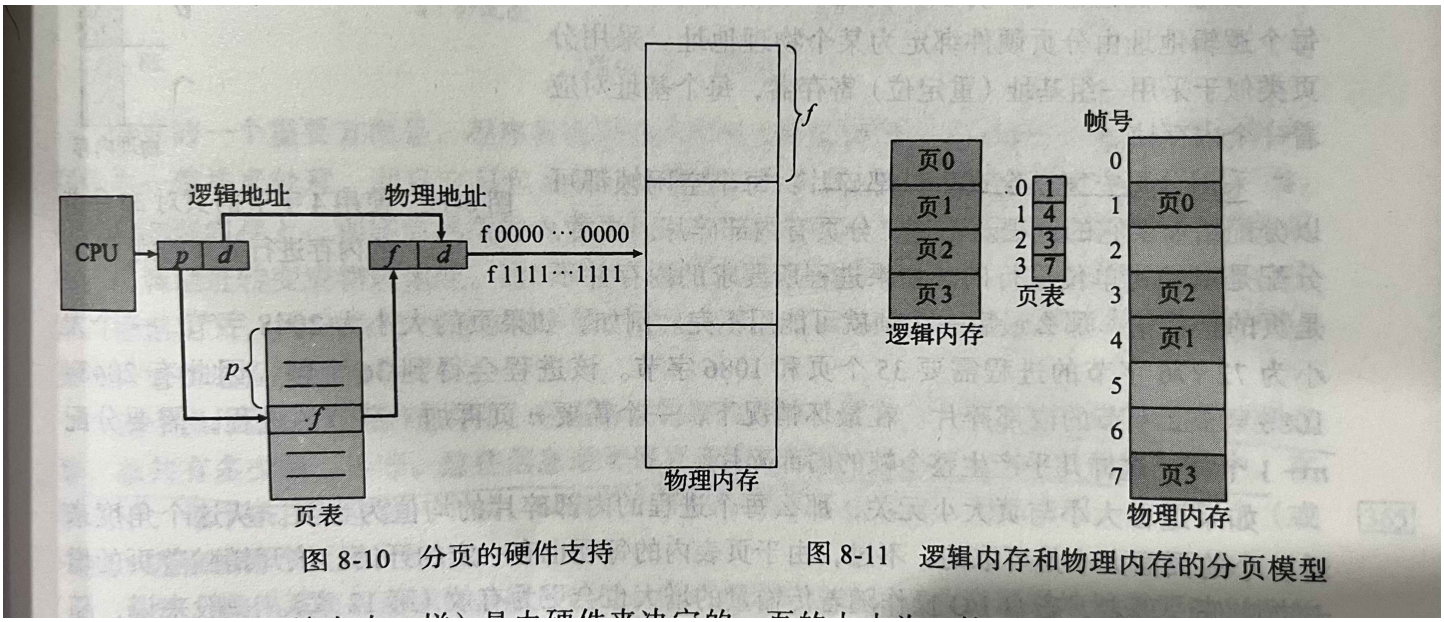
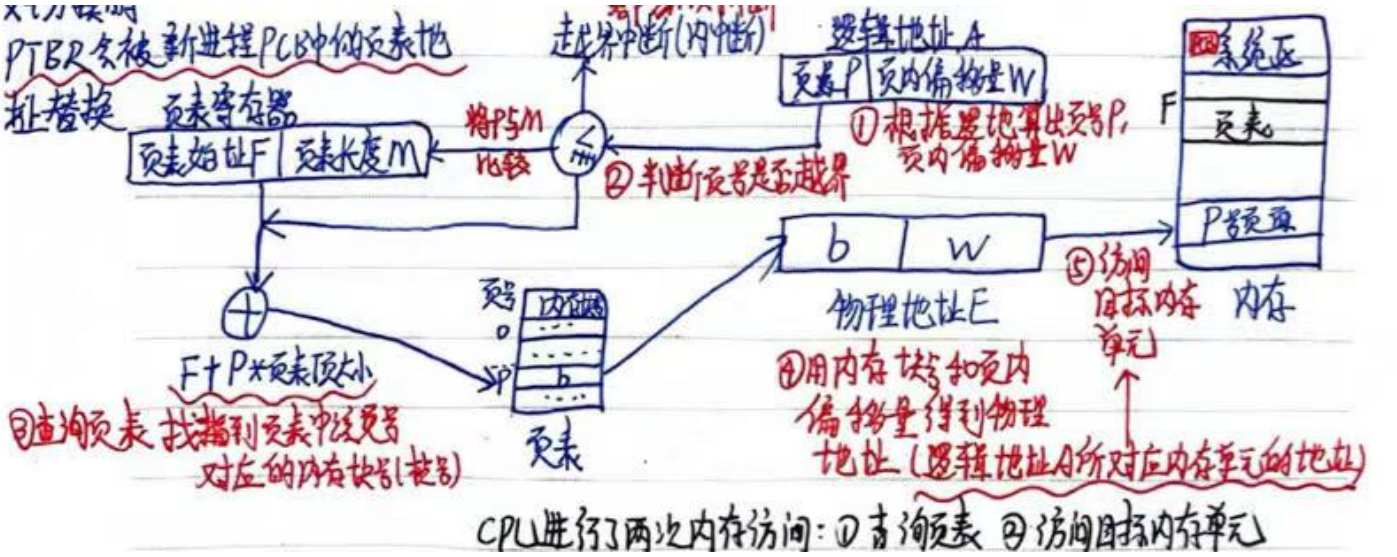


图 8-10　分页的硬件支持　　图 8-11　逻辑内存和物理内存的分页模型

非连续内存分配方式基于页表的地址变换机构，通过内存管理单元（MMU）来实现从逻辑地址到物理地址的转换。以下是基于页表的地址变换机构的工作流程：

1. 页表存放在内存中： 页表通常存放在内存中，它将逻辑地址转换为物理地址。每当一个进程被加载到内存中时，其页表的起始地址被存储在进程控制块中。
2. 页表寄存器（PTR）： 当处理器需要访问页表时，页表的起始地址被加载到页表寄存器（PTR）中。此时，页表的起始地址和基址寄存器中的地址相加，从而得到页表在内存中的实际地址。
3. 访问页表： 处理器使用逻辑地址中的页号查询页表，以找到相应的页表项。
4. 计算物理地址： 通过页表项中的块号和页内偏移量，可以计算出逻辑地址对应的物理地址。块号表示内存块在内存中的位置，而页内偏移量则表示逻辑地址在内存块中的偏移量。
5. 内存访问： 处理器进行两次内存访问。首先，它通过查询页表目录来获取页表项的地址，然后再次查询页表项以获取实际的物理地址。
6. 越界检查： 在进行地址转换时，需要进行越界检查以确保访问的页面在合法范围内。如果页号超出了页表的范围，则会触发越界中断。
7. 计算页内偏移量： 根据页面内偏移量和页表项的起始地址，可以计算出目标内存单元的地址。



TLB（Translation Look-aside Buffer）是一个快表的地址变换机构，是基于基址寄存器的改进。

局部性原理指出，程序在执行期间存在时间局部性和空间局部性。时间局部性指的是一条指令可能在程序执行期间被反复访问，因此相邻的存储位置可能会在同一个页面中。地址变换机构中，每次访问一个逻辑地址通常需要对内存进行两次访问。为了减少访问更表的次数，引入了快表（TLB）。快表是一种特殊的高速缓冲器，存储了当前正在访问的页表项。与内存中的页表相对应，TLB中存储了已经访问过的内存块号和页号。在CPU查询页表之前，会先查询TLB，如果要访问的地址已经在快表中，则可以直接访问相应的内存单元，节省了一次访存时间。

TLB中存放的是页表的副本内容，每次查询页表后，会自动将相应的页号内存块号复制到TLB中。如果TLB已满，则根据一定的规则替换TLB中的表项。TLB命中表示找到了对应的号，无需再次访问内存；若未命中，则CPU需要对页表进行访问，并将其存入TLB中。

一般情况下，TLB的命中率可以达到90%以上，TLB命中的时间远远低于查询页表的时间，可以显著节省时间。

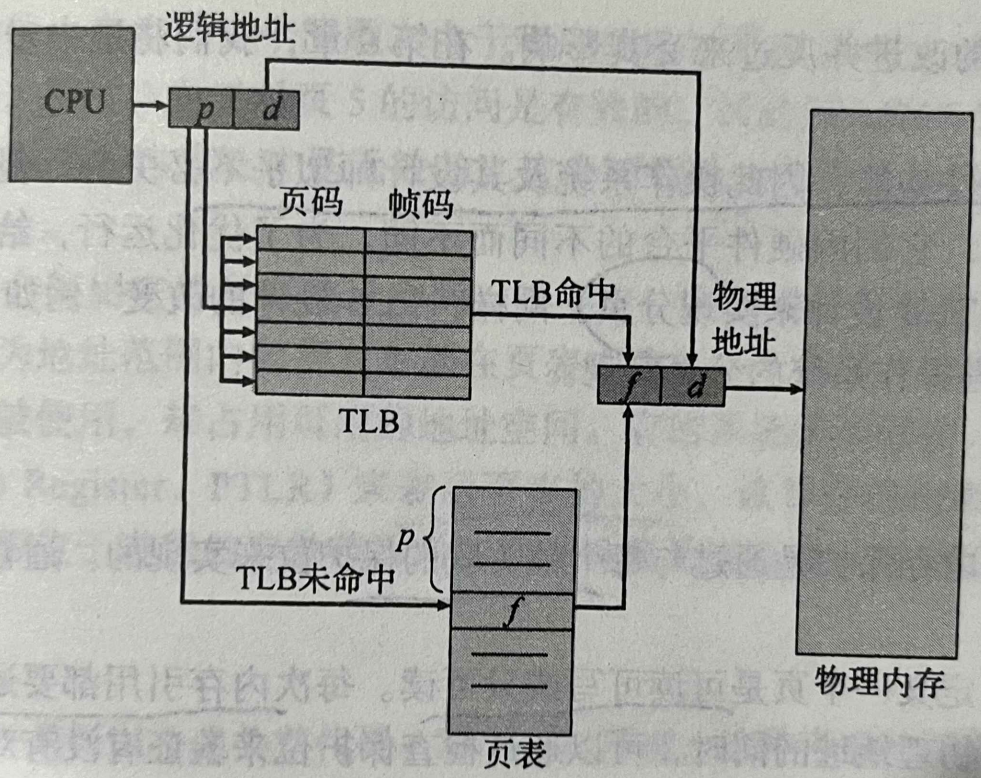


图 8-14　带 TLB 的分页硬件

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**.

- **TLB lookup time (E)**,
- **memory access time (M)** and
- **hit ratio (A)**

EFFECTIVE MEMORY-ACCESS TIME (EAT) - estimation of the impact of the TLB on the execution speed of the computer

$$EAT = A(E + M) + (1 - A)(E + 2 \times M)$$

1 - A = TLB **miss ratio**

多级页表是一种解决大内存中页表过大而导致连续内存消耗的方案。这里提到了两级页表作为示例，但也提到了可能需要更多级的情况。以下是关于多级页表的解释：

1. 原理： 多级页表是将单级页表分层次管理的一种方式。它通过将页表分成多个级别，每个级别都包含一定数量的页表。通过多级的结构，可以减少单个页表的大小，从而节省内存空间。
2. 地址结构： 逻辑地址被划分为三个部分：一级页表号、二级页表号和页内偏移量。根据逻辑地址的结构，可以从一级页表开始逐级查询，直到找到最终的物理地址。
3. 页表结构： 一级页表的结构与二级页表相似，但它们记录的不是物理地址，而是下一级页表的帧号。最后一级页表中存放的是物理页框号和页内偏移量。
4. 解决方案： 多级页表的出现解决了单级页表过大的问题，同时也减少了连续内存的消耗。通过分级管理，即使系统中有大量页表，也可以将其有效地存储在内存中。
5. 访存次数： 在两级页表的情况下，访问内存需要进行三次。而在更多级的情况下，每增加一级，访问内存的次数就会增加一次。
6. 内存管理： 多级页表是一种基本的内存管理机制，通过将地址空间划分为若干个段，每个段都有自己的逻辑地址空间。这种方式使得程序可以更加灵活地管理内存，并根据程序自身的逻辑关系来划分地址空间。

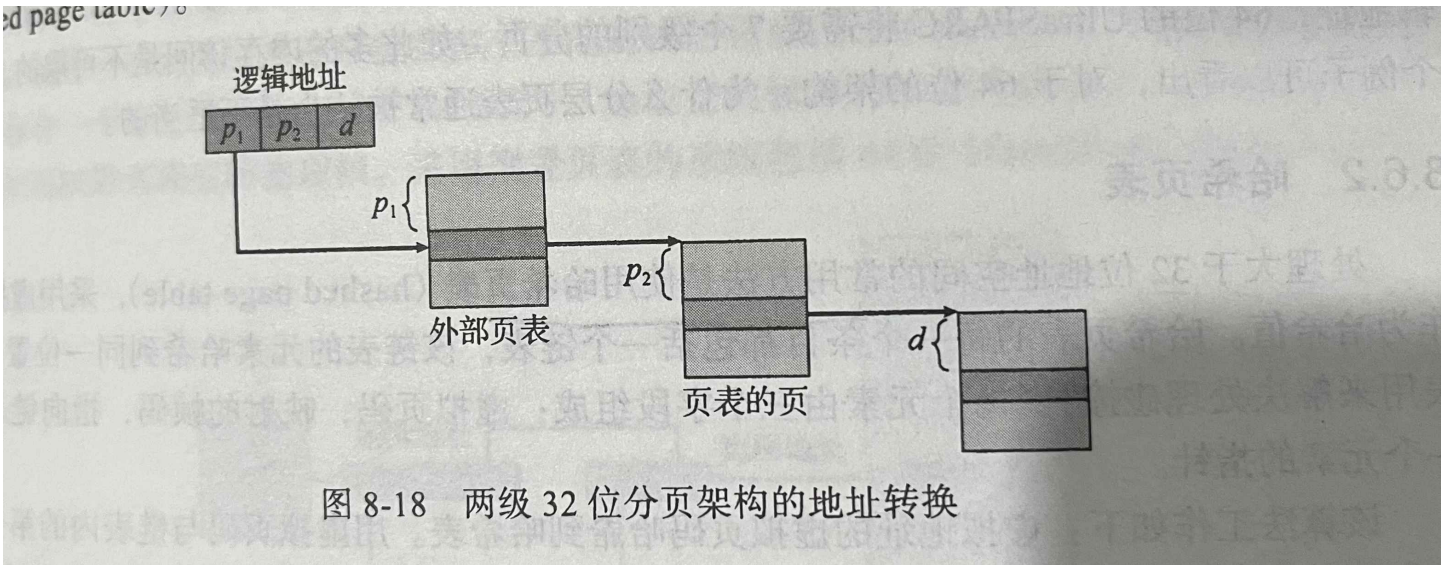


图 8-18　两级 32 位分页架构的地址转换

内存分段规则：将内存按段进行划分，每个段在内存中是连续的，但各段的大小可以不相同。

由于提高了逻辑功能模块的灵活性和用户编程的便利性，将内存分为多个段，每个段有自己的段名和长度。

通过段名来访问内存中的段，这样更容易理解，也更具可读性。

内存分段系统的逻辑地址结构由段号（段名）和段内地址（段内偏移量）组成。段号的位数决定了进程可以拥有多少段。

内存大小为4GB（2^32B），因此每个段表项占用6B（32位段号 + 16位段内地址）。

段表记录了各个段在内存中的存放位置，每个段有自己的段表项，记录了段在内存中的起始位置（基址）。

根据逻辑地址得到段号和段内地址，根据段号查询段表，得到段的起始位置，再根据段内地址得到物理地址。

判断逻辑地址是否越界：

- 如果段号超出了段表的长度，则越界。
- 如果段内地址超出了段的长度，则越界。

根据逻辑地址得到物理地址的公式为：物理地址 = 段基址 + 段内地址。

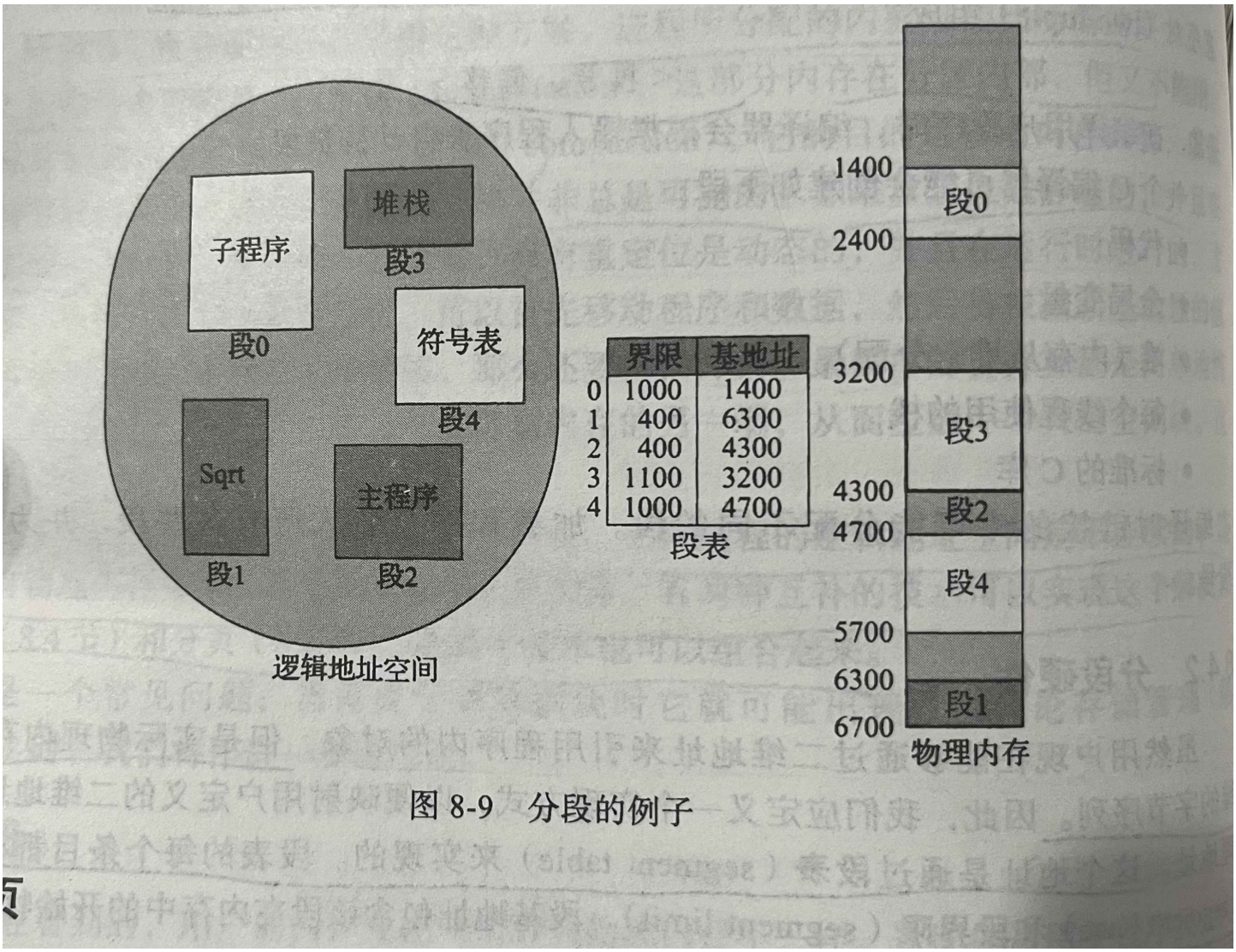
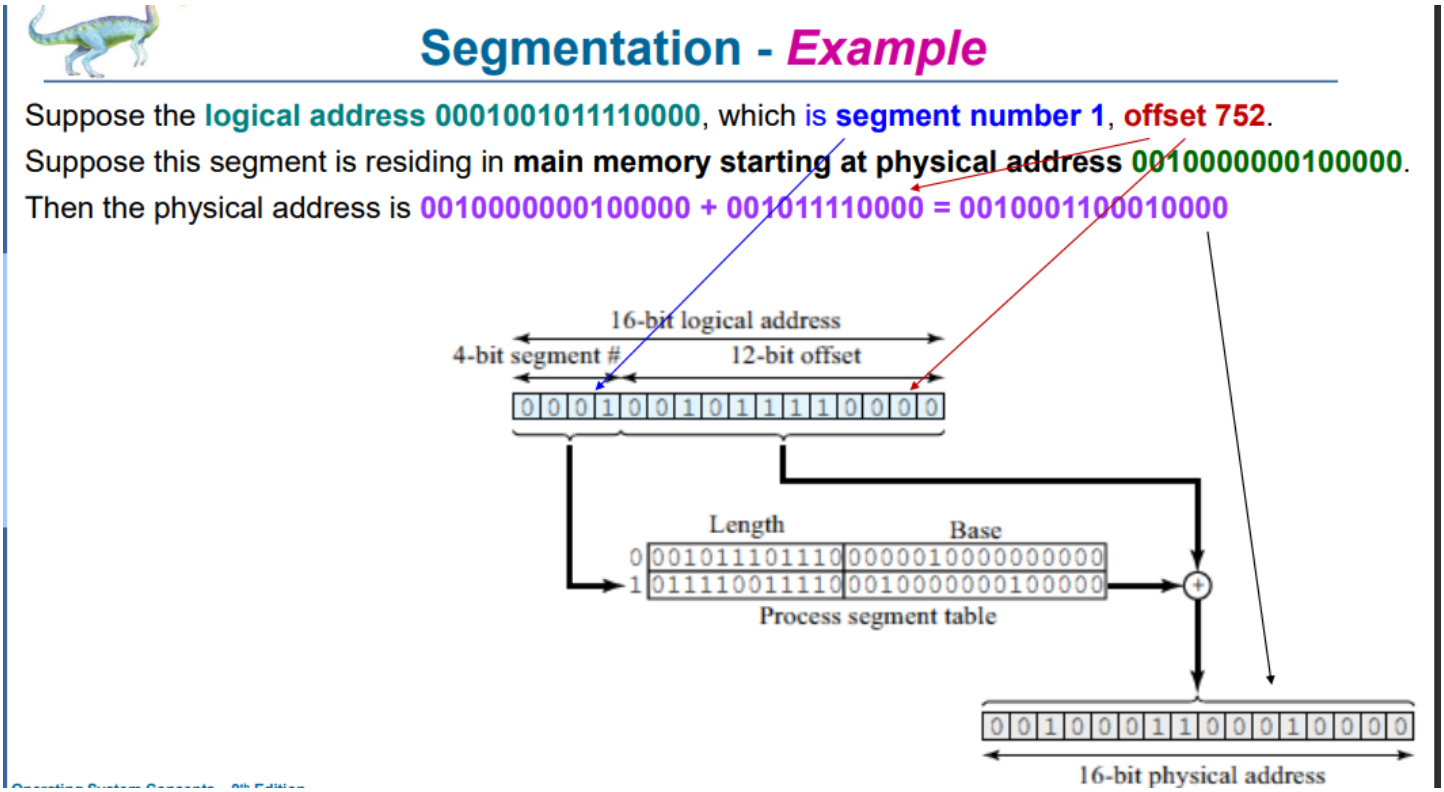


图 8-9 分段的例子



1. 分页和分段管理的对比：
   - 页是信息的物理单位，主要目的是为了将离散的信息划分为固定大小的块，以提高内存的利用率。 页的大小是由系统决定的，用户无法改变。用户程序中只需给出逻辑地址，物理地址的转换由系统完成。
   - 段是信息的逻辑单位，主要是为了满足用户需求，一个段通常包含一个逻辑模块的信息。 用户在编程时需要显式地给出段名，并且要定义段内地址。用户通过定义段名和段内地址来告诉系统如何进行地址转换。段的长度是由用户定义的。
2. 分段比分页更容易实现信息的共享和保护：
   - 段管理更容易实现信息的共享和保护。 两个进程可以同时访问同一个段，只需修改进程的段表即可实现共享。但对于页，没有逻辑关系，很难确定哪些页可以共享，哪些页不能。段的功能是由用户定义的，因此可以明确地控制哪些段可以共享，哪些段不可以。
3. 分页管理和分段管理的性能比较：
   - 分页管理需要两次内存访问（查询页表和访问物理地址），而分段管理需要一至两次内存访问（查询段表和访问物理地址）。 因此，分段管理在性能上略优于分页管理。
   - 分段管理利用率高，不会产生外部碎片，但可能会有少量的内部碎片。 分页管理虽然不容易产生内部碎片，但可能会有外部碎片。分页管理对内存的分配更加灵活，但不太容易实现信息共享和保护。分段管理虽然容易实现信息共享和保护，但如果段很大，连续的内存分配可能会产生外部碎片。
   - 通过“紧凑”技术可以解决外部碎片的问题，但会增加时间代价。