

# 第三章 存储器管理

- 3.1 程序的执行过程
- 3.2 连续内存分配
- 3.3 非连续内存分配（分页分段）
- 3.4 虚拟内存管理（请求分段，请求分页）

测与解除。

## 5、存储器管理

程序装入与链接；连续分配管理  
理方式、基本分段存储管理方式

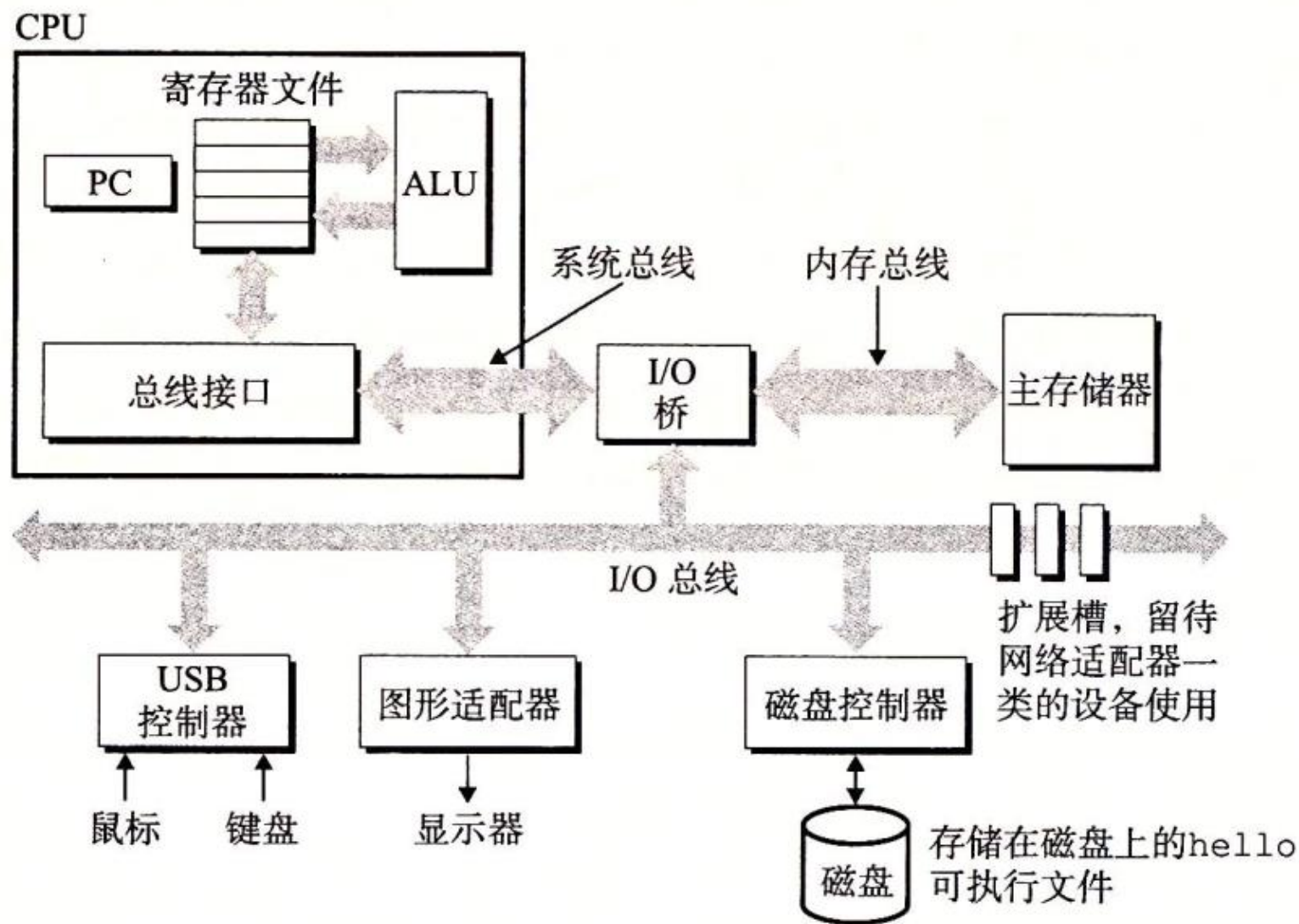


图 1-4 一个典型系统的硬件组成

CPU: 中央处理单元; ALU: 算术/逻辑单元; PC: 程序计数器; USB: 通用串行总线

## 存储体系结构的核心

作为程序员，我们肯定是希望有无限资源的快速存储器，来存放程序的数据。而现实是，快速存储器的制造成本很高，速度慢的存储器相对便宜。所以从成本角度来说，计算机的存储结构被设计成分层的，一般包括寄存器、缓存、内存、磁盘等。

其中，缓存又是整个存储体系结构的灵魂，它让内存访问的速度接近于寄存器的访问速度。所以，要想深入理解存储体系结构，我们就要围绕“缓存”这个核心来学习。

在过去的几十年，处理器速度的增长远远超过了内存速度的增长。尤其是在 2001 ~ 2005 年间，处理器的时钟频率在以 55% 的速度增长，而同期内存速度的增长仅为 7%。为了缩小处理器和内存之间的速度差距，缓存被设计出来。

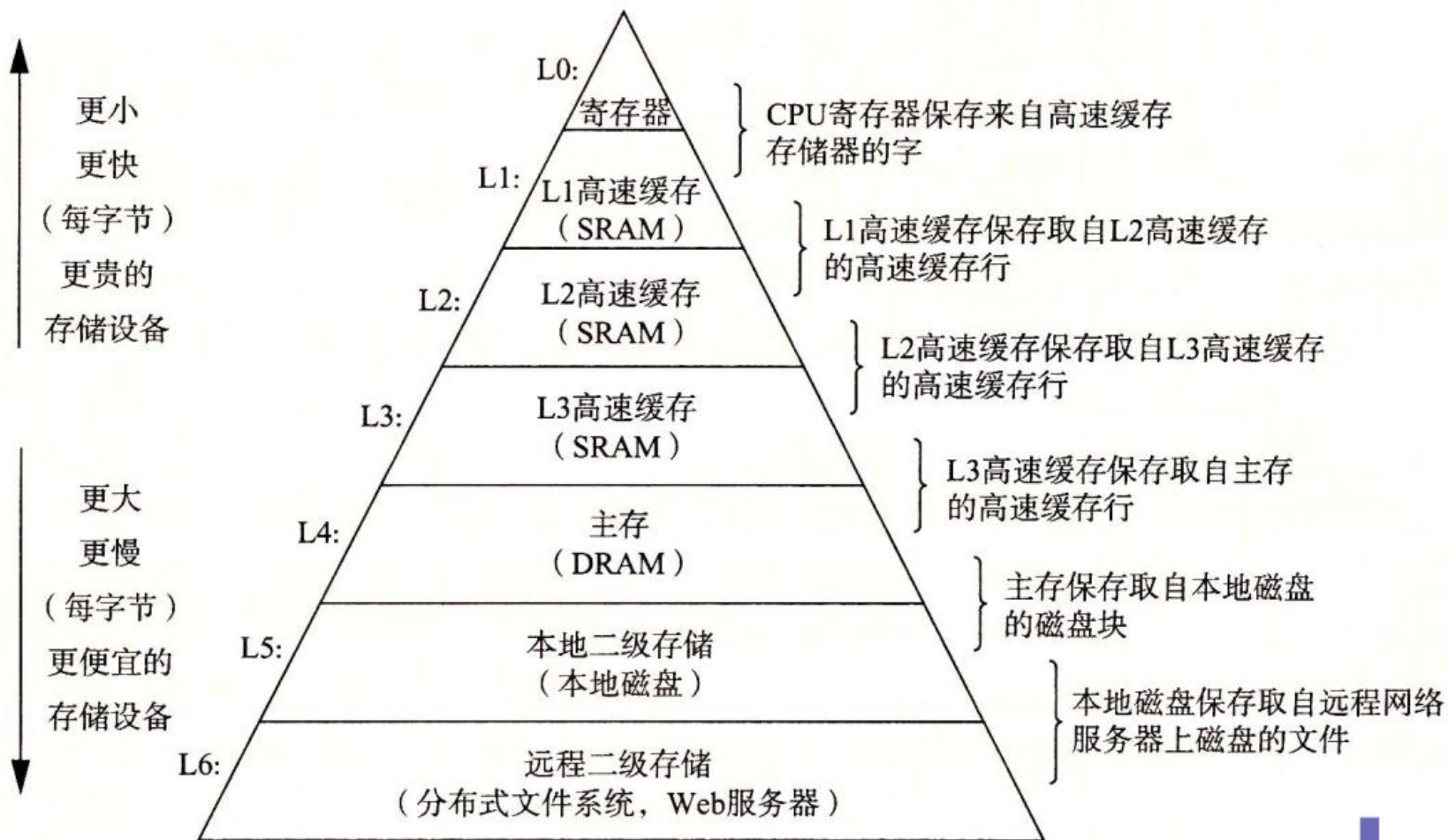


图 1-9 一个存储器层次结构的示例

造价越来越低

速度越来越慢

容量越来越大

Register 300ps
L1 Cache 1ns
L2 Cache 10ns
Memory 100ns
SSD 25us
HDD 5ms

造价越来越高

速度越来越快

容量越来越小

CPU 芯片

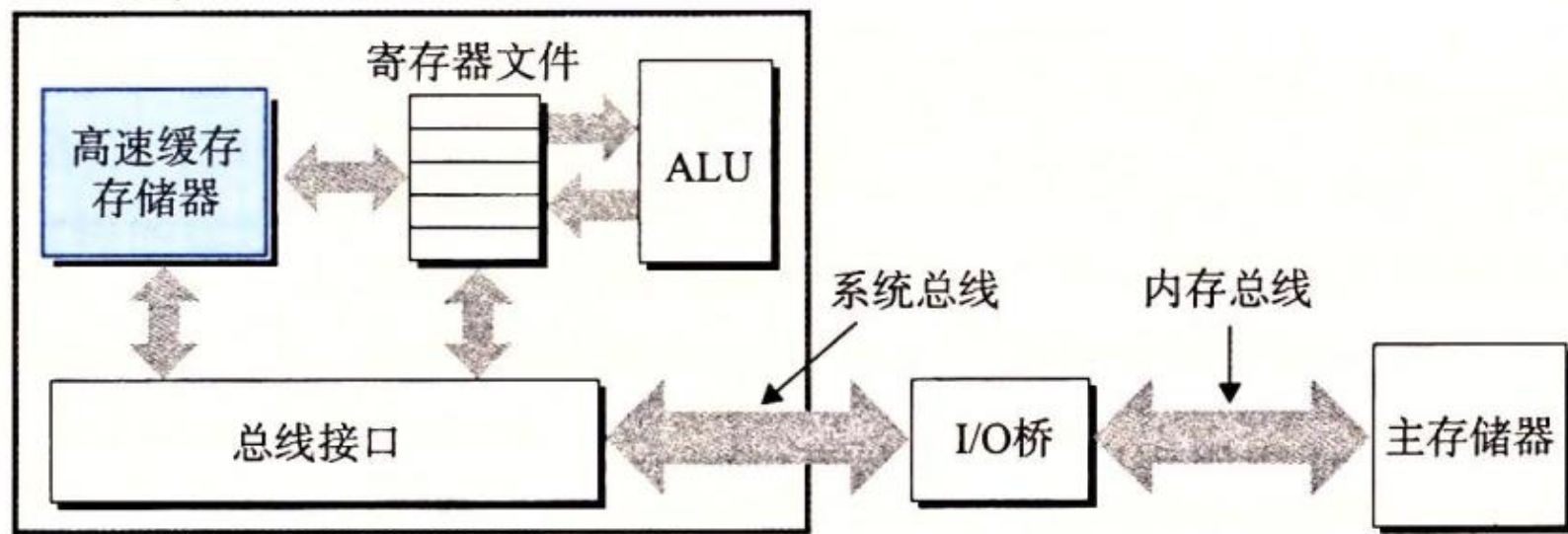
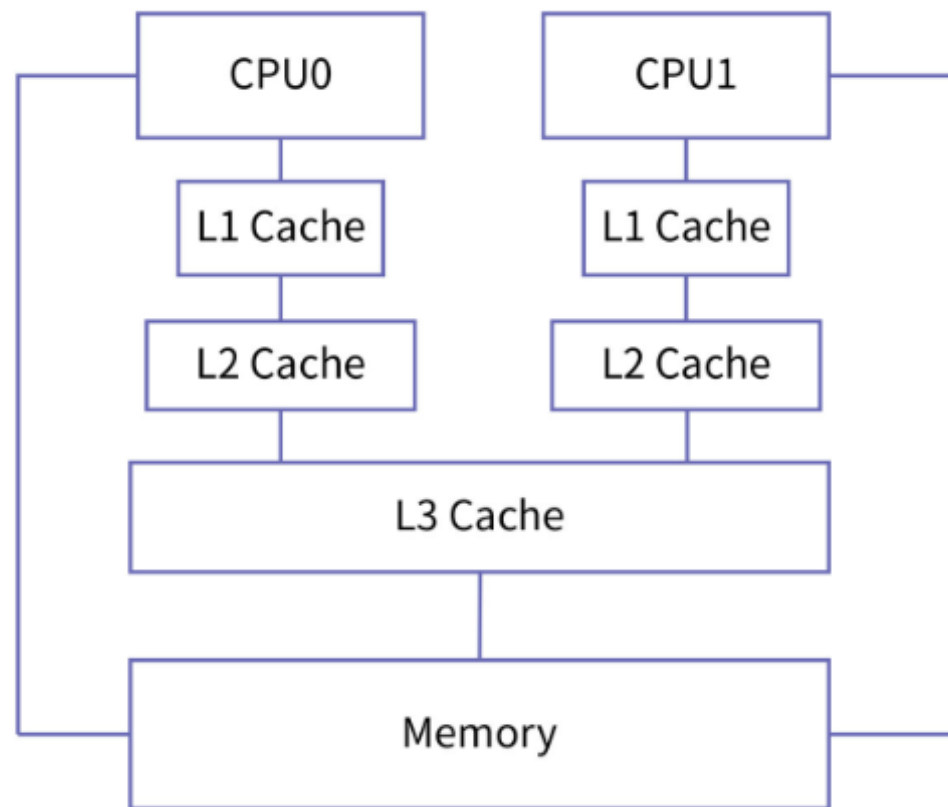



图 1-8 高速缓存存储器





## 1. 主存储器

主存储器简称内存或主存，是计算机系统中的主要部件，用于保存进程运行时的程序和数据，也称可执行存储器。通常，处理机都是从主存储器中取得指令和数据的，并将其所取得的指令放入指令寄存器中，而将其所读取的数据装入到数据寄存器中；或者反之，将寄存器中的数据存入到主存储器。早期的内存是由磁芯做成的，其容量一般为数十 KB 到数百 KB。随着 VLSI 的发展，现在的内存已由 VLSI 构成，其容量，即使是微机系统，也在数十 MB 到数 GB，而且还在不断增加。而嵌入式计算机系统一般仅有几十 KB 到几 MB。CPU 与外围设备交换的信息一般也依托于主存储器的地址空间。由于主存储器访问速度远低于 CPU 执行指令的速度，为缓和这一矛盾，在计算机系统中引入了寄存器和高速缓存。



## 2. 寄存器

寄存器具有与处理机相同的速度，故对寄存器的访问速度最快，完全能与 CPU 协调工作，但价格却十分昂贵，因此容量不可能做得很大。在早期计算机中，寄存器的数目仅为几个，主要用于存放处理机运行时的数据，以加速存储器的访问速度，如使用寄存器存放操作数，或用作地址寄存器加快地址转换速度等。随着 VLSI 的发展，寄存器的成本也在迅速降低，在当前的微机系统和大中型机中，寄存器的数目都已增加到数十个到数百个，而寄存器的字长一般是 32 位或 64 位；而在小型的嵌入式计算机中，寄存器的数目仍只有几个到十几个，而且寄存器的字长通常只有 8 位。



# 1. 高速缓存

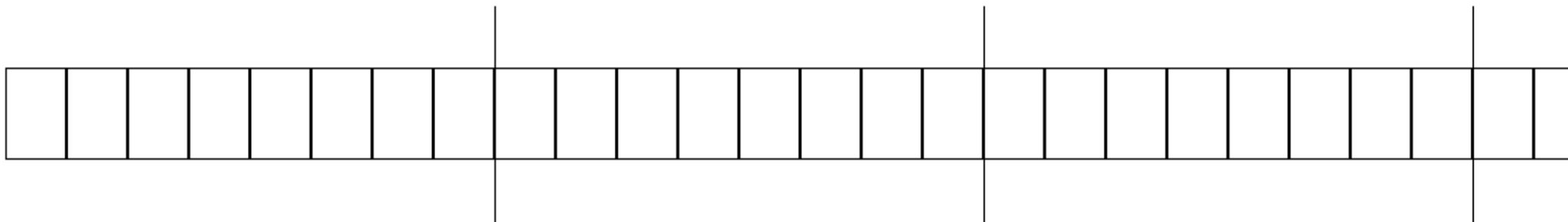


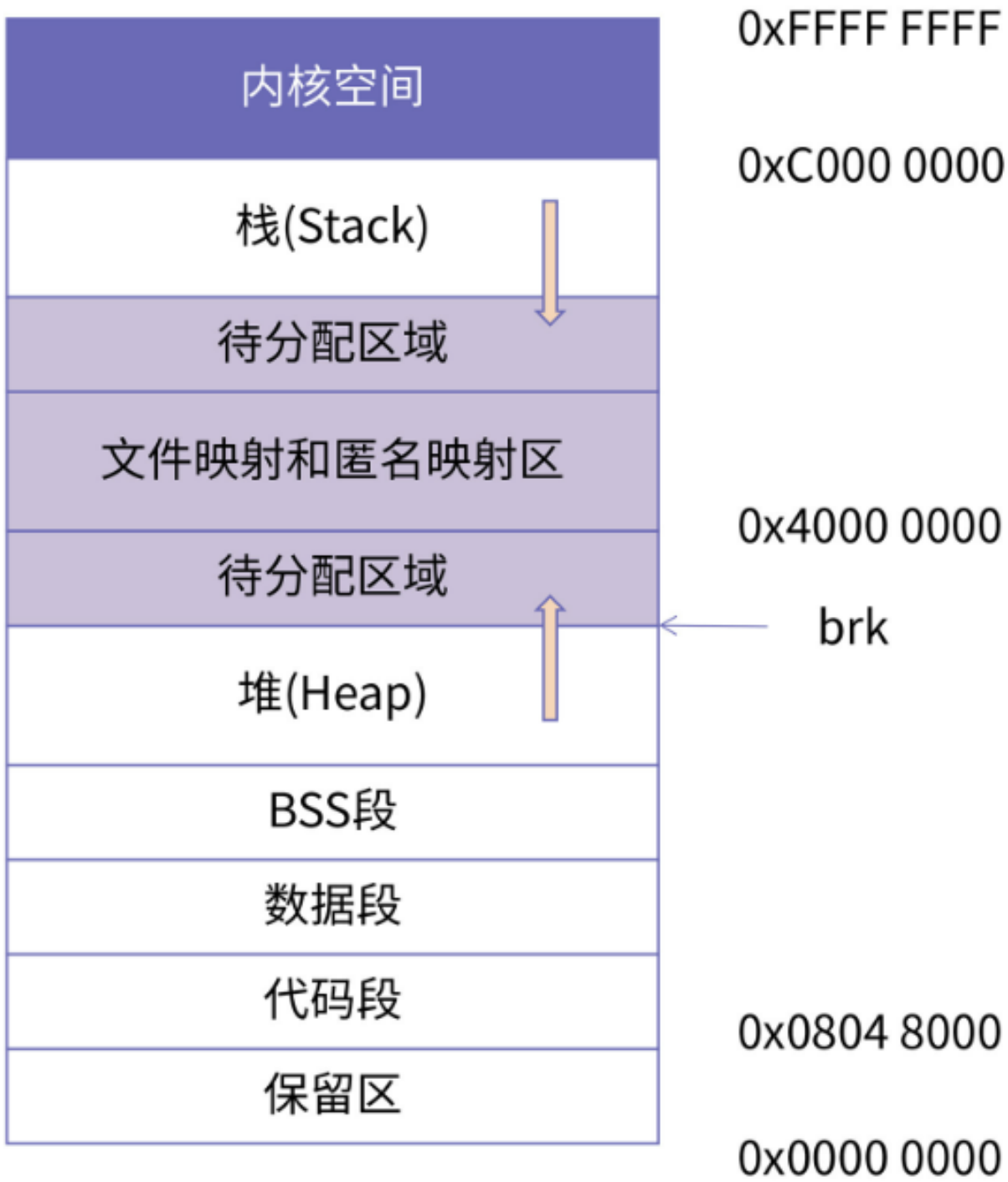
高速缓存是现代计算机结构中的一个重要部件，它是介于寄存器和存储器之间的存储器，主要用于备份主存中较常用的数据，以减少处理机对主存储器的访问次数，这样可大幅度地提高程序执行速度。高速缓存容量远大于寄存器，而比内存约小两到三个数量级左右，从几十 KB 到几 MB，访问速度快于主存储器。在计算机系统中，为了缓和内存与处理机速度之间的矛盾，许多地方都设置了高速缓存。在以后各章中将会经常遇见各种高速缓存的，届时再对它们进行详细的介绍。

利用率	速度	基准速度:	2.21 GHz
11%	3.88 GHz	插槽:	1
进程	线程	内核:	6
248	3688	逻辑处理器:	12
	句柄	虚拟化:	已启用
	119449	L1 缓存:	384 KB
正常运行时间		L2 缓存:	1.5 MB
0:12:50:10		L3 缓存:	9.0 MB

## 2.1 信息存储

大多数计算机使用 8 位的块，或者字节(byte)，作为最小的可寻址的内存单位，而不是访问内存中单独的位。机器级程序将内存视为一个非常大的字节数组，称为虚拟内存(virtual memory)。内存的每个字节都由一个唯一的数字来标识，称为它的地址(address)，所有可能地址的集合就称为虚拟地址空间(virtual address space)。顾名思义，这个虚拟地址空间只是一个展现给机器级程序的概念性映像。实际的实现(见第 9 章)是将动态随机访问存储器(DRAM)、闪存、磁盘存储器、特殊硬件和操作系统软件结合起来，为程序提供一个看上去统一的字节数组。





# 引入16进制

方便。二进制表示法太冗长，而十进制表示法与位模式的互相转化很麻烦。替代的方法是，以 16 为基数，或者叫做十六进制(hexadecimal)数，来表示位模式。十六进制(简写为“hex”)

十六进制数字	0	1	2	3	4	5	6	7
十进制值	0	1	2	3	4	5	6	7
二进制值	0000	0001	0010	0011	0100	0101	0110	0111
十六进制数字	8	9	A	B	C	D	E	F
十进制值	8	9	10	11	12	13	14	15
二进制值	1000	1001	1010	1011	1100	1101	1110	1111

图 2-2 十六进制表示法。每个十六进制数字都对 16 个值中的一个进行了编码



### 2.1.3 寻址和字节顺序

对于跨越多字节的程序对象，我们必须建立两个规则：这个对象的地址是什么，以及在内存中如何排列这些字节。在几乎所有的机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。例如，假设一个类型为 `int` 的变量 `x` 的地址为 `0x100`，也就是说，地址表达式 `&x` 的值为 `0x100`。那么，（假设数据类型 `int` 为 32 位表示）`x` 的 4 个字节将被存储在内存的 `0x100`、`0x101`、`0x102` 和 `0x103` 位置。

假设变量 `x` 的类型为 `int`，位于地址 `0x100` 处，它的十六进制值为 `0x01234567`。地址范围 `0x100~0x103` 的字节顺序依赖于机器的类型：





```
- int main() {  
    SeqList seqList;  
    seqList.arr[0] = 1;  
    seqList.arr[1] = 2;  
    seqList.arr[2] = 3;  
    for (int i = 0; i < 3; i++)  
        cout << &seqList.arr[i] << endl;  
    return 0;  
}
```

00EFFB88

00EFFB8C

00EFFB90



## 4.2 程序的装入和链接

用户程序要在系统中运行，必须先将它装入内存，然后再将其转变为一个可以执行的程序，通常都要经过以下几个步骤：

- (1) 编译，由编译程序(Compiler)对用户源程序进行编译，形成若干个目标模块(Object Module)；
- (2) 链接，由链接程序(Linker)将编译后形成的一组目标模块以及它们所需要的库函数链接在一起，形成一个完整的装入模块(Load Module)；
- (3) 装入，由装入程序(Loader)将装入模块装入内存。

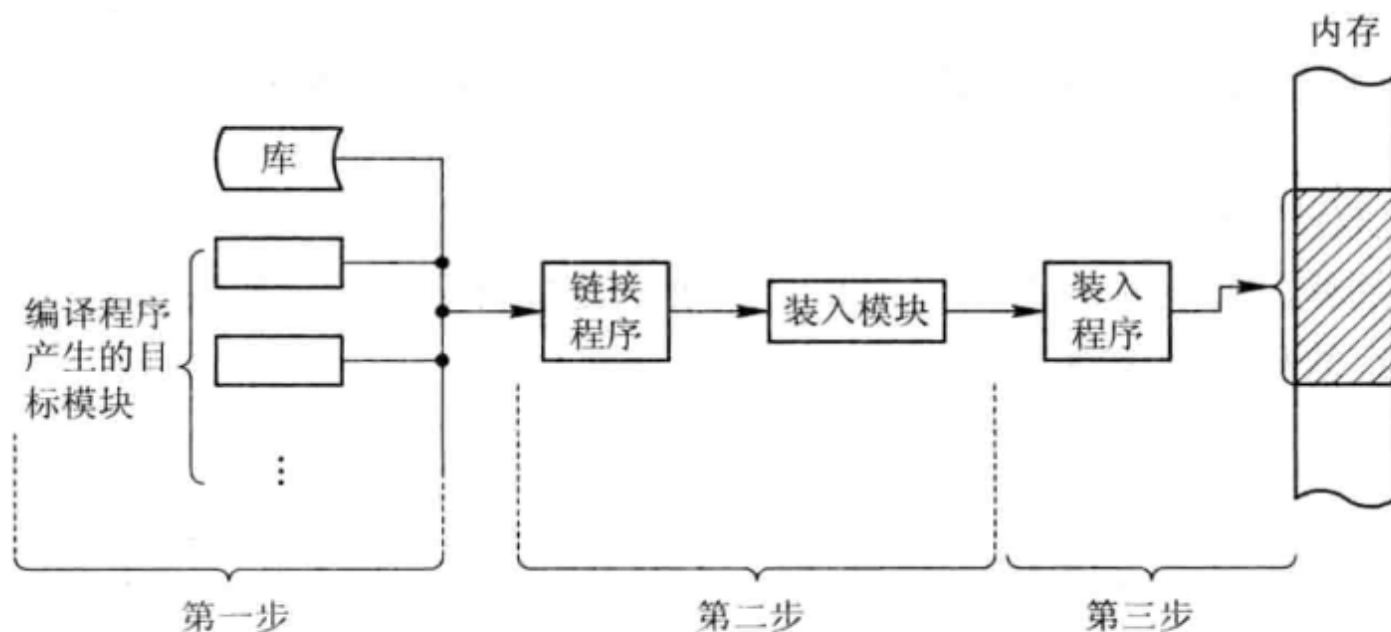


图 4-2 对用户程序的处理步骤

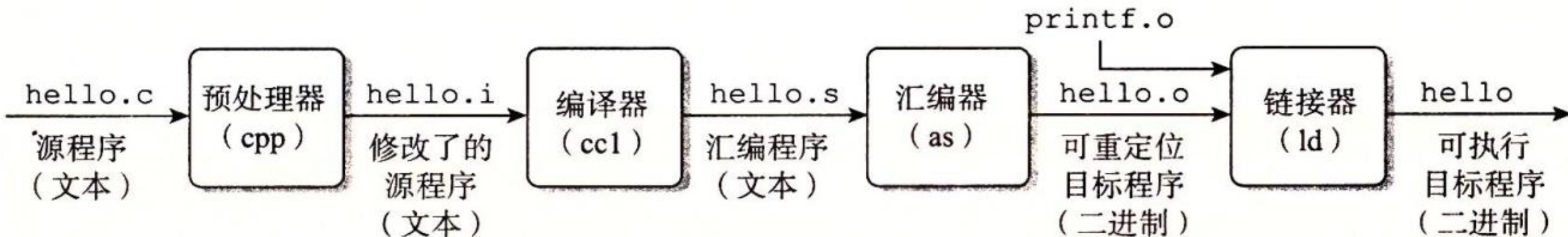


图 1-3 编译系统

- 预处理阶段。预处理器(cpp)根据以字符#开头的命令，修改原始的 C 程序。比如 `hello.c` 中第 1 行的 `#include <stdio.h>` 命令告诉预处理器读取系统头文件 `stdio.h` 的内容，并把它直接插入程序文本中。结果就得到了另一个 C 程序，通常是以 `.i` 作为文件扩展名。
- 编译阶段。编译器(cc1)将文本文件 `hello.i` 翻译成文本文件 `hello.s`，它包含一个汇编语言程序。该程序包含函数 `main` 的定义，如下所示：

```
1  main:
2      subq    $8, %rsp
3      movl    $.LC0, %edi
4      call    puts
5      movl    $0, %eax
6      addq    $8, %rsp
7      ret
```



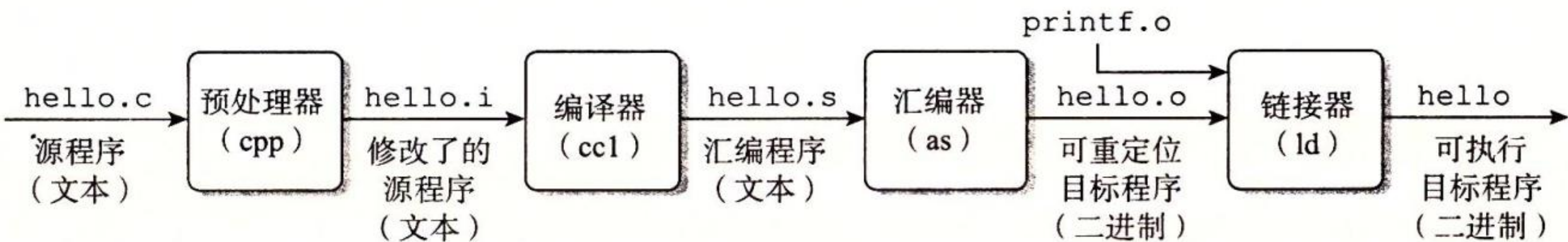


图 1-3 编译系统

- 汇编阶段。接下来，汇编器(as)将 hello.s 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序(relocatable object program)的格式，并将结果保存在目标文件 hello.o 中。hello.o 文件是一个二进制文件，它包含的 17 个字节是函数 main 的指令编码。如果我们在文本编辑器中打开 hello.o 文件，将看到一堆乱码。
- 链接阶段。请注意，hello 程序调用了 printf 函数，它是每个 C 编译器都提供的标准 C 库中的一个函数。printf 函数存在于一个名为 printf.o 的单独的预编译好了的目标文件中，而这个文件必须以某种方式合并到我们的 hello.o 程序中。链接器(ld)就负责处理这种合并。结果就得到 hello 文件，它是一个可执行目标文件(或者简称为可执行文件)，可以被加载到内存中，由系统执行。

## 4.2.2 程序的链接

源程序经过编译后，可得到一组目标模块。链接程序的功能是将这组目标模块以及它们所需要的库函数装配成一个完整的装入模块。在对目标模块进行链接时，根据进行链接的时间不同，可把链接分成如下三种。

静态链接， 装入时动态链接， 运行时动态链接

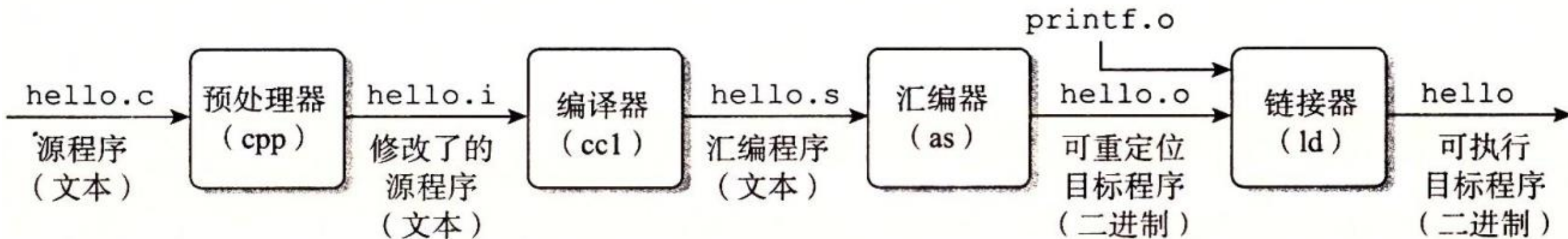


图 1-3 编译系统



## 1. 静态链接(Static Linking)方式

在程序运行之前，先将各目标模块及它们所需的库函数链接成一个完整的装配模块，以后不再拆开。我们把这种事先进行链接的方式称为静态链接方式。我们通过一个例子来说明在实现静态链接时应解决的一些问题。在图 4-4(a)中示出了经过编译后所得到的三个目标模块 A、B、C，它们的长度分别为 L、M 和 N。在模块 A 中有一条语句 CALL B，用于调用模块 B。在模块 B 中有一条语句 CALL C，用于调用模块 C。B 和 C 都属于外部调用符号，在将这几个目标模块装配成一个装入模块时，须解决以下两个问题：

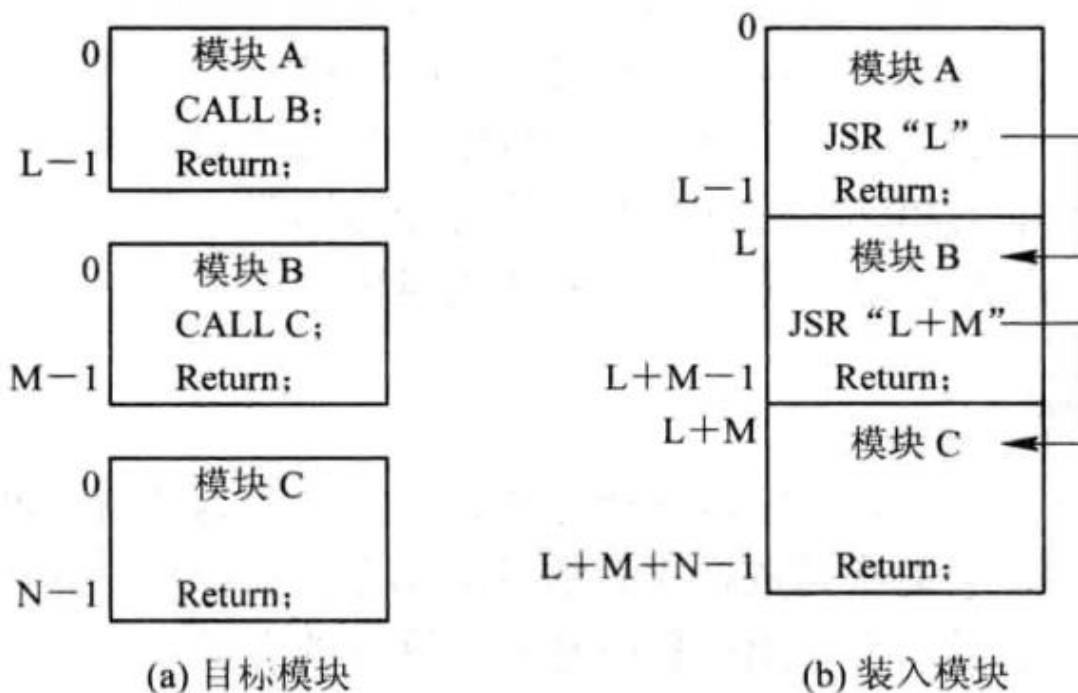
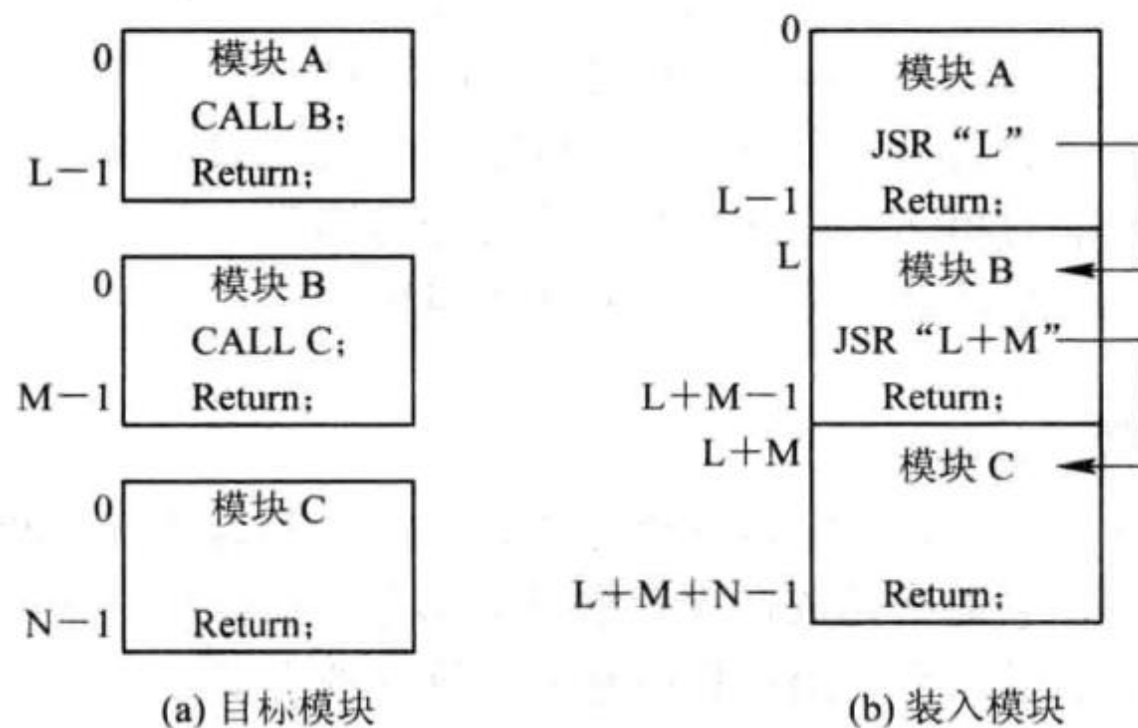
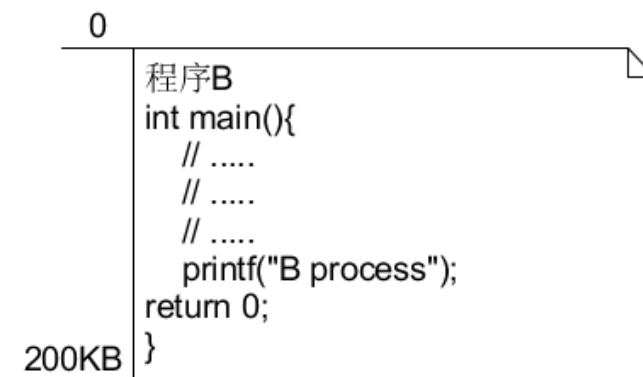
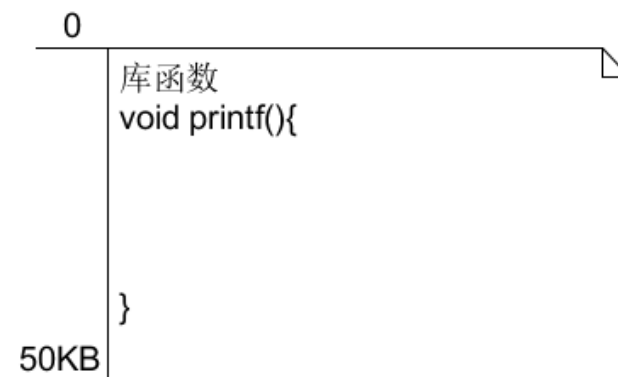
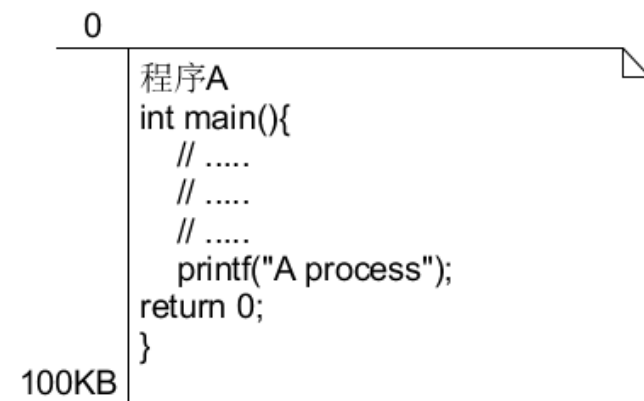


图 4-4 程序链接示意图

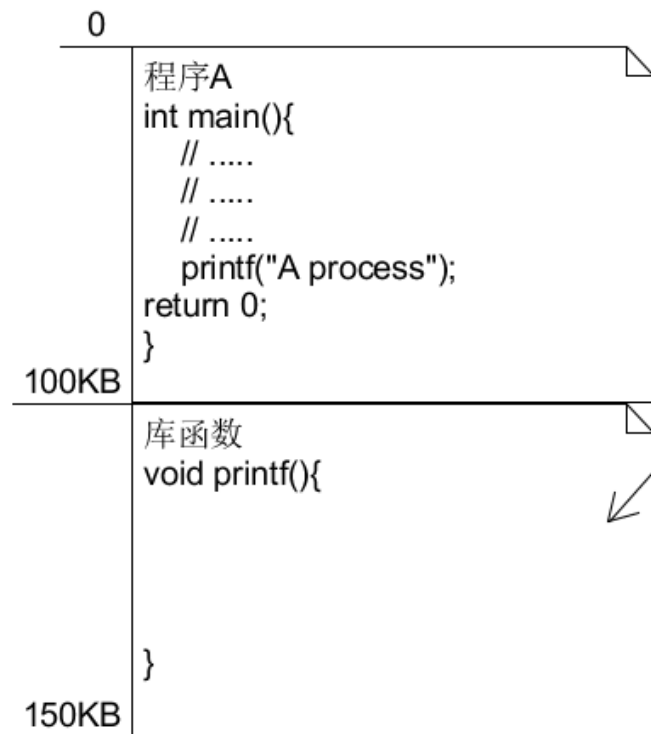
(1) 对相对地址进行修改。在由编译程序所产生的所有目标模块中，使用的都是相对地址，其起始地址都为 0，每个模块中的地址都是相对于起始地址计算的。在链接成一个装入模块后，原模块 B 和 C 在装入模块的起始地址不再是 0，而分别是 L 和 L + M，所以此时须修改模块 B 和 C 中的相对地址，即把原 B 中的所有相对地址都加上 L，把原 C 中所有相对地址都加上 L + M。

(2) 变换外部调用符号。将每个模块中所用的外部调用符号也都变换为相对地址，如把 B 的起始地址变换为 L，把 C 的起始地址变换为 L + M，如图 4-4(b)所示。这种先进行链接所形成的一个完整的装入模块，又称为可执行文件。通常都不再把它拆开，要运行时可直接将它装入内存。把这种事先进行链接而以后不再拆开的链接方式称为静态链接方式。

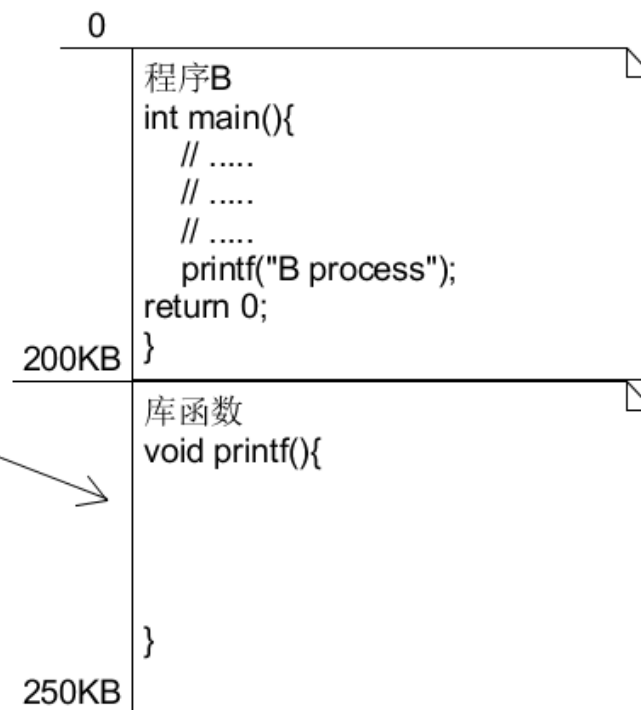




编译  
静态链接



模块printf重复，  
造成存储空间的浪费



## 什么是动态链接

要想解决静态链接的问题，可以把共享的部分抽离出来，组成新的模块。为了让一些公共的库函数能够被多个程序，在运行的过程中进行共享，我们可以让程序在链接和运行过程中，也拆分成不同的模块，即共享模块和私有模块。共享模块用来存放供所有进程公共使用的库函数，私有模块存放本进程独享的函数与数据。



## 2. 装入时动态链接(Load-time Dynamic Linking)

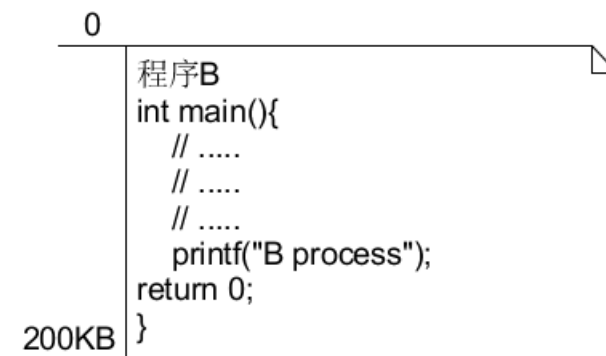
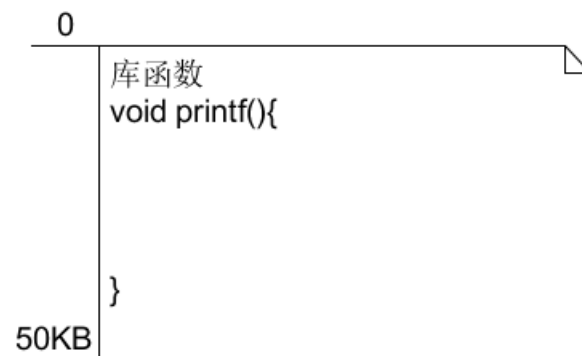
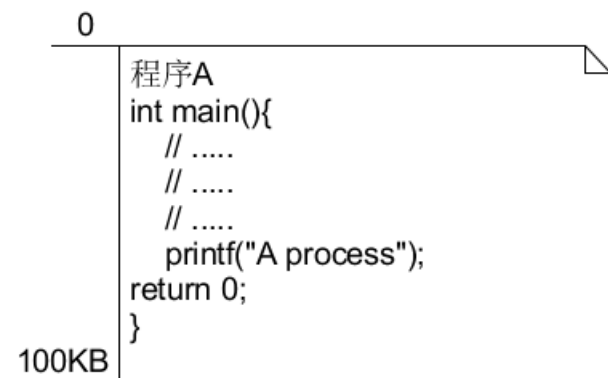
这是指将用户源程序编译后所得的一组目标模块，在装入内存时，采用边装入边链接的链接方式。即在装入一个目标模块时，若发生一个外部模块调用事件，将引起装入程序去找出相应的外部目标模块，并将它装入内存，还要按照图 4-4 所示的方式修改目标模块中的相对地址。装入时动态链接方式有以下优点：

(1) 便于修改和更新。对于经静态链接装配在一起的装入模块，如果要修改或更新其中的某个目标模块，则要求重新打开装入模块。这不仅是低效的，而且有时是不可能的。若采用动态链接方式，由于各目标模块是分开存放的，所以要修改或更新各目标模块是件非常容易的事。

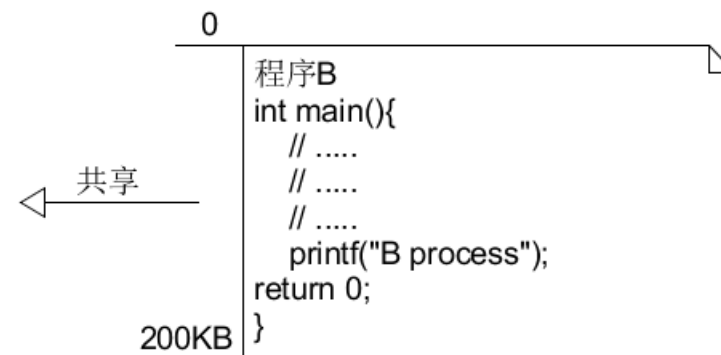
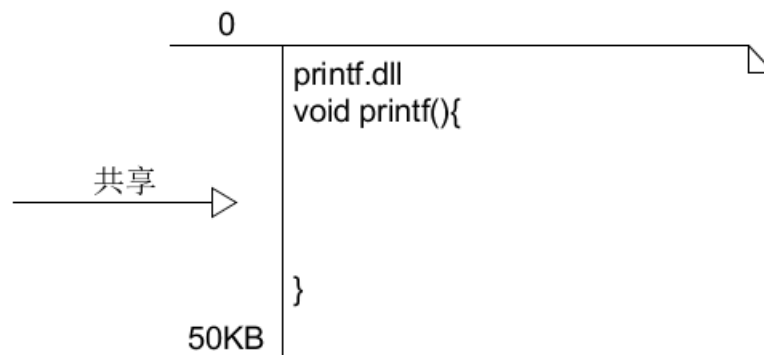
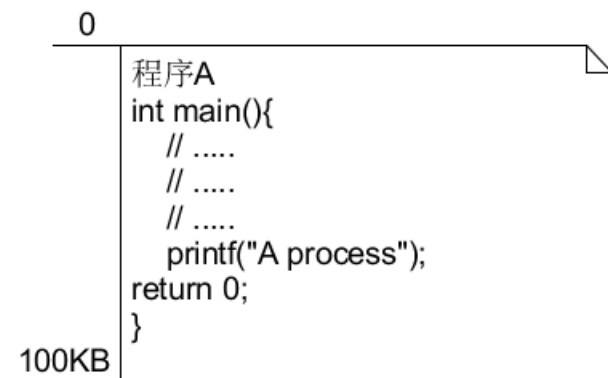
(2) 便于实现对目标模块的共享。在采用静态链接方式时，每个应用模块都必须含有其目标模块的拷贝，无法实现对目标模块的共享。但采用装入时动态链接方式时，OS 就很容易将一个目标模块链接到几个应用模块上，实现多个应用程序对该模块的共享。

在运行时，只有当EXE程序确实要调用这些DLL模块的情况下，系统才会将它们装载到内存空间中。这种方式不仅减少了EXE文件的大小和对内存空间的需求，而且使这些DLL模块可以同时被多个应用程序使用。





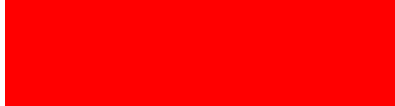
编译  
动态链接  
▽



### 3. 运行时动态链接(Run-time Dynamic Linking)

在许多情况下，应用程序在运行时，每次要运行的模块可能是不相同的。但由于事先无法知道本次要运行哪些模块，故只能是将所有可能要运行到的模块全部都装入内存，并在装入时全部链接在一起。显然这是低效的，因为往往会有部分目标模块根本就不运行。比较典型的例子是作为错误处理用的目标模块，如果程序在整个运行过程中都不出现错误，则显然就不会用到该模块。

近几年流行起来的运行时动态链接方式，是对上述装入时链接方式的一种改进。这种链接方式是，将对某些模块的链接推迟到程序执行时才进行。亦即，在执行过程中，当发现一个被调用模块尚未装入内存时，立即由 OS 去找到该模块，并将之装入内存，将其链接到调用者模块上。凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅能加快程序的装入过程，而且可节省大量的内存空间。

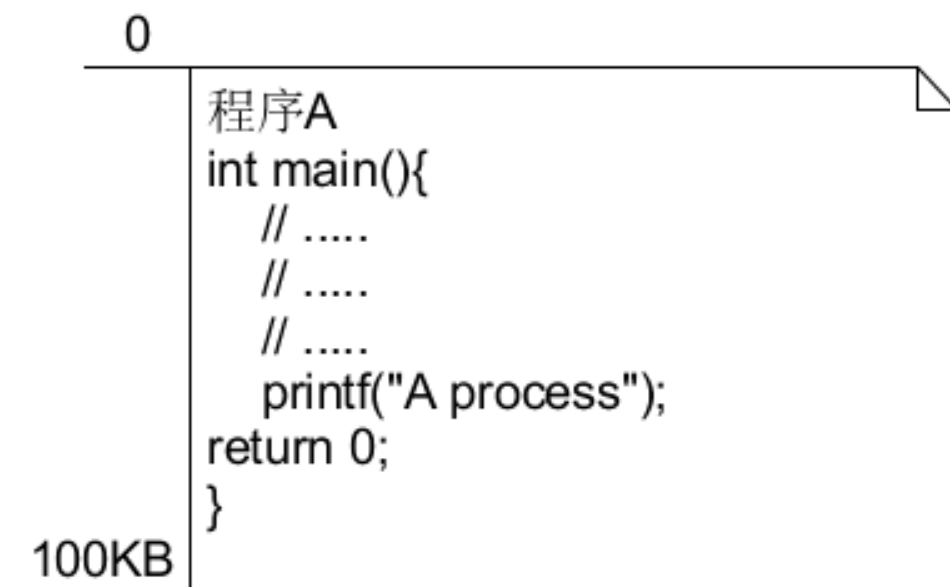


## 4.2.1 程序的装入

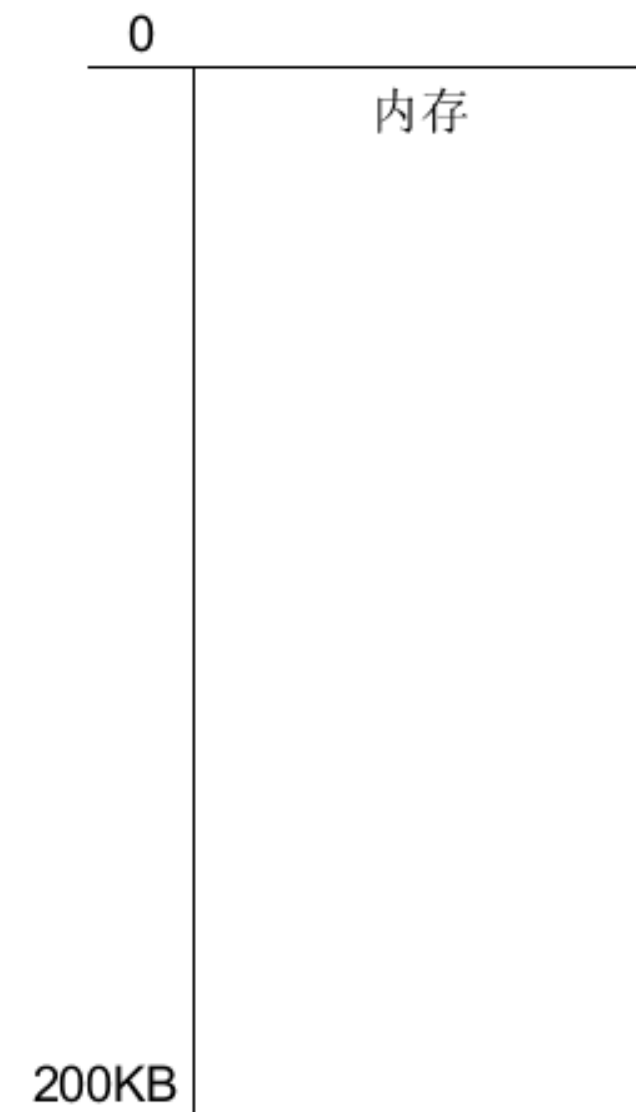
为了阐述上的方便，我们先介绍一个无需进行链接的单个目标模块的装入过程。该目标模块也就是装入模块。在将一个装入模块装入内存时，可以有如下三种装入方式：

### 1. 绝对装入方式(Absolute Loading Mode)

当计算机系统很小，且仅能运行单道程序时，完全有可能知道程序将驻留在内存的什么位置。此时可以采用绝对装入方式。用户程序经编译后，将产生绝对地址(即物理地址)的目标代码。例如，事先已知用户程序(进程)驻留在从  $R$  处开始的位置，则编译程序所产生的目标模块(即装入模块)，便可从  $R$  处开始向上扩展。绝对装入程序便可按照装入模块中的地址，将程序和数据装入内存。装入模块被装入内存后，由于程序中的相对地址(即逻辑地址)与实际内存地址完全相同，故不需对程序 and 数据的地址进行修改。



绝对装入



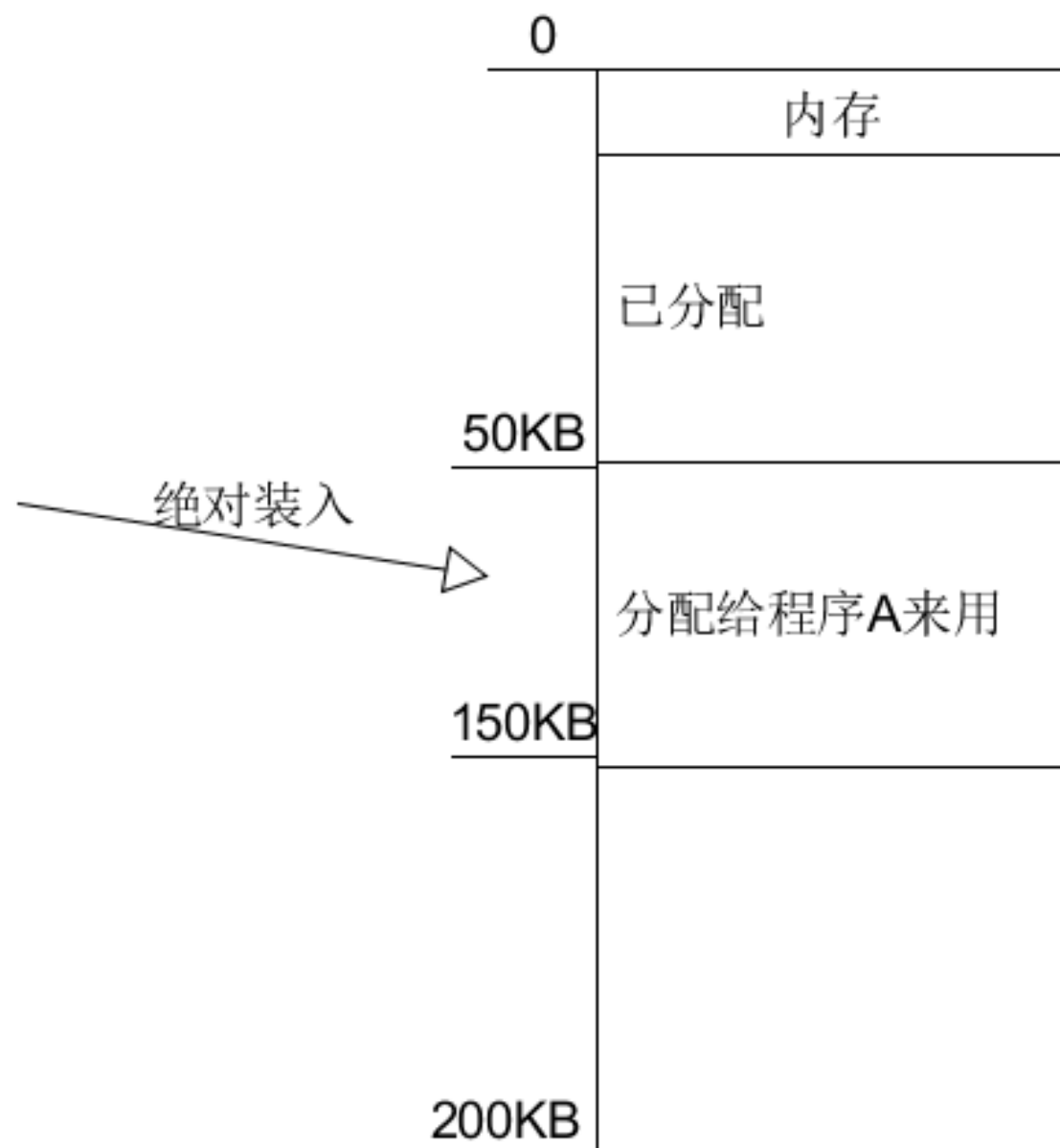
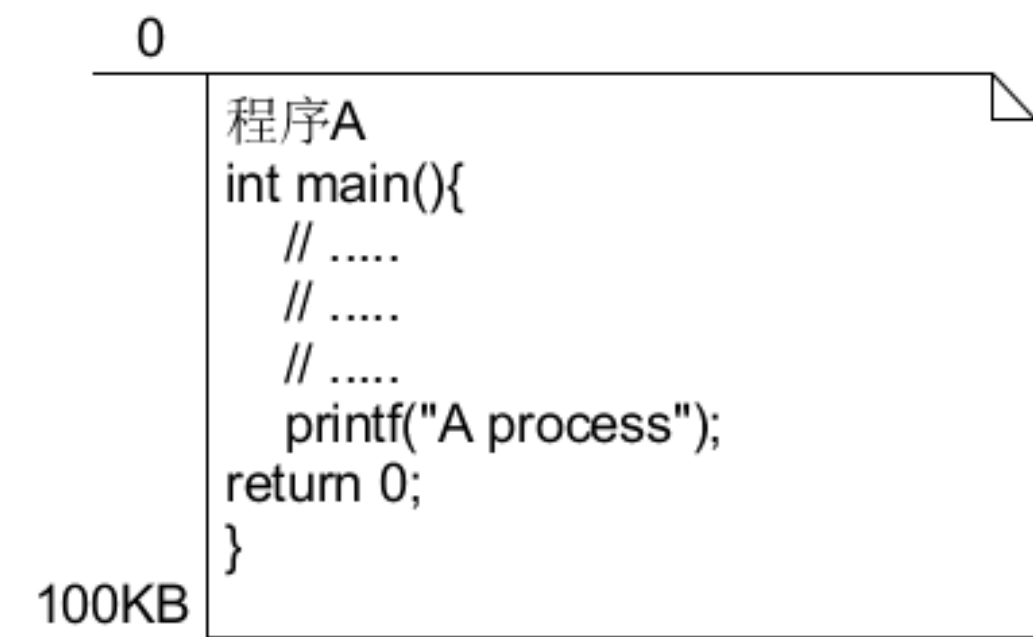
## (2) 可重定位装入

在多道程序环境下，多个目标模块的起始地址通常都从 0 开始，程序中的其他地址都是相对于起始地址的，此时应采用可重定位装入方式。根据内存的当前情况，将装入模块装入内存的适当位置。在装入时对目标程序中指令和数据地址的修改过程称为重定位，又因为地址变换通常是在进程装入时一次完成的，故称为静态重定位，如图 3.2(a)所示。

当一个作业装入内存时，必须给它分配要求的全部内存空间，若没有足够的内存，则无法装入。此外，作业一旦进入内存，整个运行期间就不能在内存中移动，也不能再申请内存空间。

缺点





### (3) 动态运行时装入

也称动态重定位。程序在内存中若发生移动，则需要采用动态的装入方式。装入程序把装入模块装入内存后，并不立即把装入模块中的相对地址转换为绝对地址，而是把这种地址转换推迟到程序真正要执行时才进行。因此，装入内存后的所有地址均为相对地址。这种方式需要一个重定位寄存器的支持，如图 3.2(b)所示。

动态重定位的优点：可以将程序分配到不连续的存储区；在程序运行之前可以只装入部分代码即可投入运行，然后在程序运行期间，根据需要动态申请分配内存；便于程序段的共享。

优点

# 真题


- 2014.3 静态链接 名词解释
- 2017.1 存储器管理的主要任务

# 作业

- 阅读课本4.1存储器的层次结构&4.2程序的装入和链接 P120
- 预习分段 分页内存管理
- 1.可采用哪几种方式将程序装入内存？它们分别适用于那种场合？
- 2.什么是静态链接？静态链接需要解决两个什么问题？
- 3.什么是装入时动态链接？装入时动态链接方式有何优点？
- 4.什么是运行时动态链接？有何优点？




- 1.首先由编译程序将用户源代码编译成若干目标模块，再由链接程序将编译后形成的目标模块和所需的库函数链接在一起，组成一个装入模块，再由装入程序将装入模块装入内存；
- 绝对装入方式适用于单道程序环境下；
- 可重定位方式适用于多道程序环境下；
- 动态运行时装入方式也适用于多道程序环境下。
- 2.在程序运行之前，先将各目标模块及它们所需的库函数连接成一个完整的装配模块，以后不再拆开。这种事先进行连接的方式称为静态链接方式。
- 要解决两个问题：对相对地址进行修改；变换外部调用符号
- 3.这是指将用户源程序编译后所得的一组目标模块，在装入内存时，采用边装入边链接的链接方式。
- 优点：便于修改和更新；便于实现对目标模块的共享。
- 4.将某些模块的链接推迟到程序执行时才进行。
- 优点：加快程序的装入过程；节省大量的内存空间。



### 3.1.1 内存管理的基本原理和要求

内存管理（Memory Management）是操作系统设计中最重要和最复杂的内容之一。虽然计算机硬件技术一直在飞速发展，内存容量也在不断增大，但仍然不可能将所有用户进程和系统所需的全部程序与数据放入主存，因此操作系统必须对内存空间进行合理的划分和有效的动态分配。操作系统对内存的划分和动态分配，就是内存管理的概念。

有效的内存管理在多道程序设计中非常重要，它不仅可以方便用户使用存储器、提高内存利用率，还可以通过虚拟技术从逻辑上扩充存储器。



内存管理的主要功能有：

- 内存空间的分配与回收。由操作系统完成主存储器空间的分配和管理，使程序员摆脱存储分配的麻烦，提高编程效率。
- 地址转换。在多道程序环境下，程序中的逻辑地址与内存中的物理地址不可能一致，因此存储管理必须提供地址变换功能，把逻辑地址转换成相应的物理地址。
- 内存空间的扩充。利用虚拟存储技术或自动覆盖技术，从逻辑上扩充内存。
- 内存共享。指允许多个进程访问内存的同一部分。例如，多个合作进程可能需要访问同一块数据，因此必须支持对内存共享区域进行受控访问。
- 存储保护。保证各道作业在各自的存储空间内运行，互不干扰。

在进行具体的内存管理之前，需要了解进程运行的基本原理和要求。

# 内存管理的任务和功能

给多道程序提供良好的运行环境，方便用户使用内存，提高内存的利用率。

功能：

1. 内存的分配和回收
2. 地址映射
3. 内存扩充
4. 内存的保护和共享



- 连续内存分配/malloc