第四章 存储器管理

存储器管理

- 存储器的层次结构
- 程序的装入和链接
- 连续分配方式
- 基本分页存储管理方式
- 基本分段存储管理方式

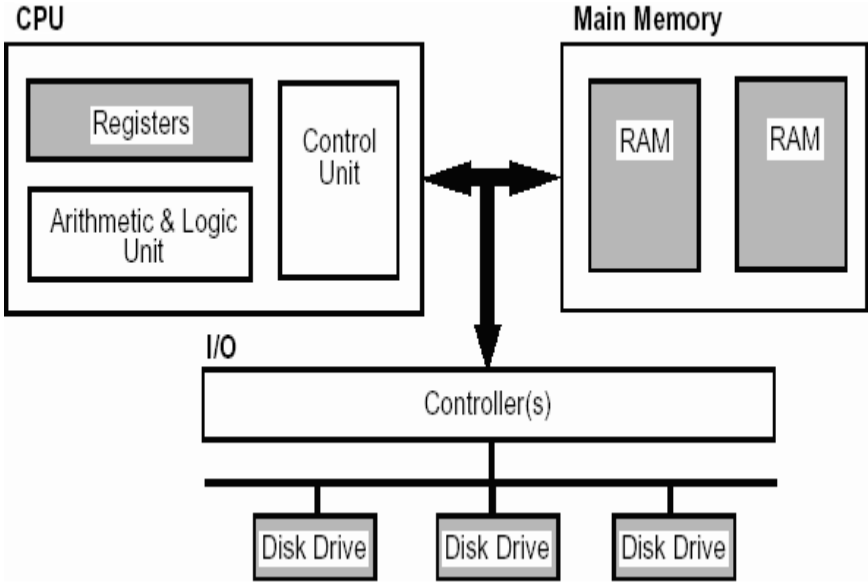
4.1 存储器的层次结构

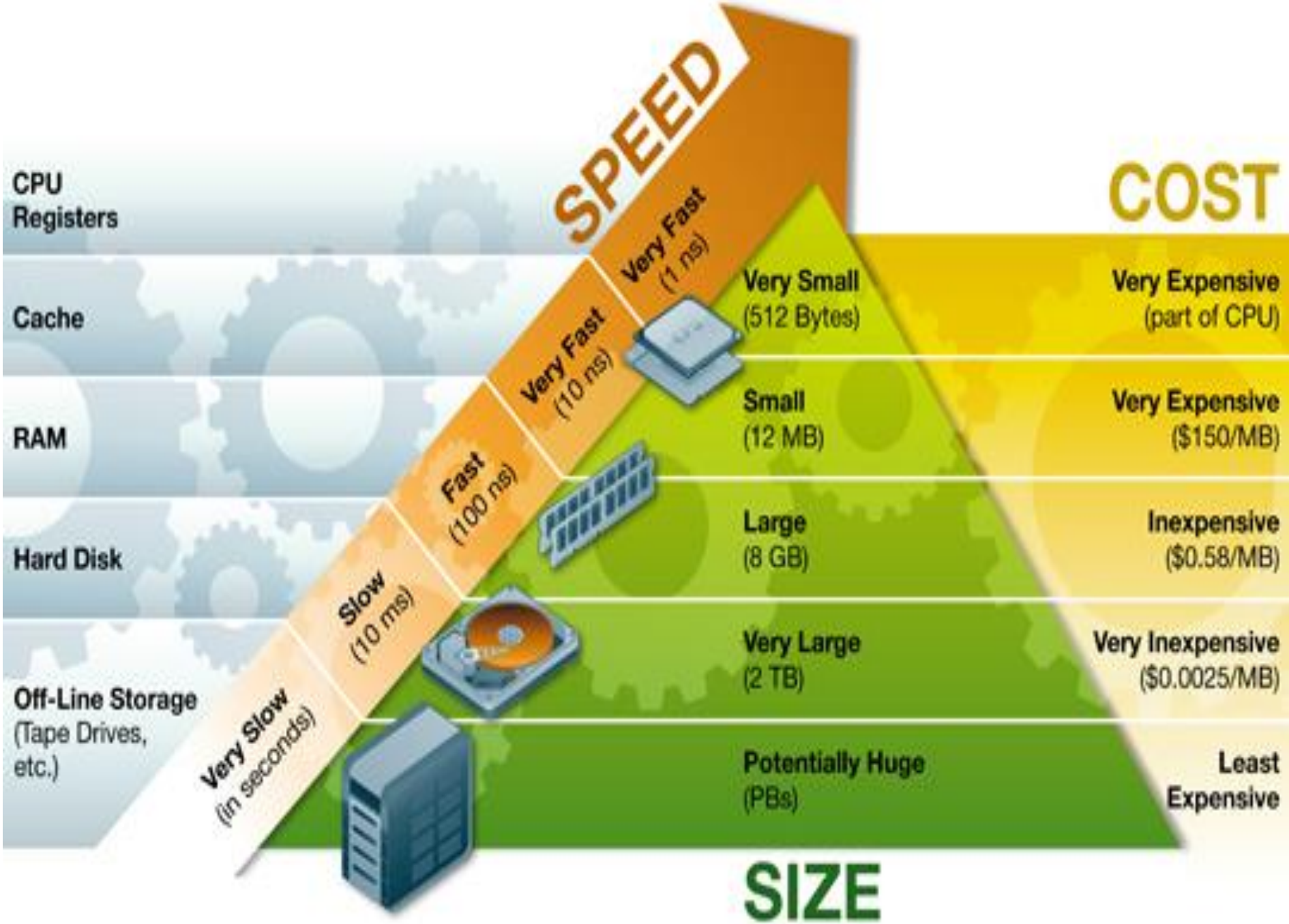
理想的存储器：

容量大，速度快，价格低

4.1.1 多级存储器结构

- 类型多种多样
  - 速度，容量，价格
  - CPU寄存器，高速缓存
  - 主存，磁盘缓存
  - 磁盘，可移动存储介质
- 分配、回收、不同层次间的移动



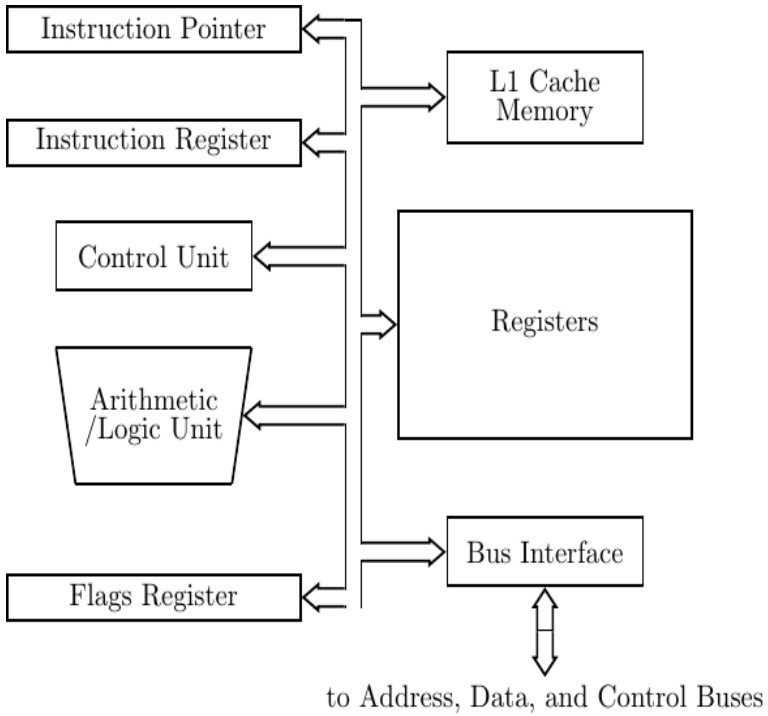

SPEED
COST
SIZE
CPU Registers
Cache
RAM
Hard Disk
Off-Line Storage (Tape Drives, etc.)
Very Fast (1 ns)
Very Fast (10 ns)
Fast (100 ns)
Slow (10 ms)
Very Slow (in seconds)
Very Small (512 Bytes)
Small (12 MB)
Large (8 GB)
Very Large (2 TB)
Potentially Huge (PBs)
Very Expensive (part of CPU)
Very Expensive ($150/MB)
Inexpensive ($0.58/MB)
Very Inexpensive ($0.0025/MB)
Least Expensive

4.1.2 主存储器和寄存器

- 1 主存储器
  - 放置进程运行时的程序和数据
  - 可执行存储器
  - 外部设备←→主存←→寄存器
- 2 寄存器
  - 速度与CPU同步

4.1.3 高速缓存和磁盘缓存

- 1 高速缓存
  - 速度高于主存
  - 存放主存中经常访问的信息



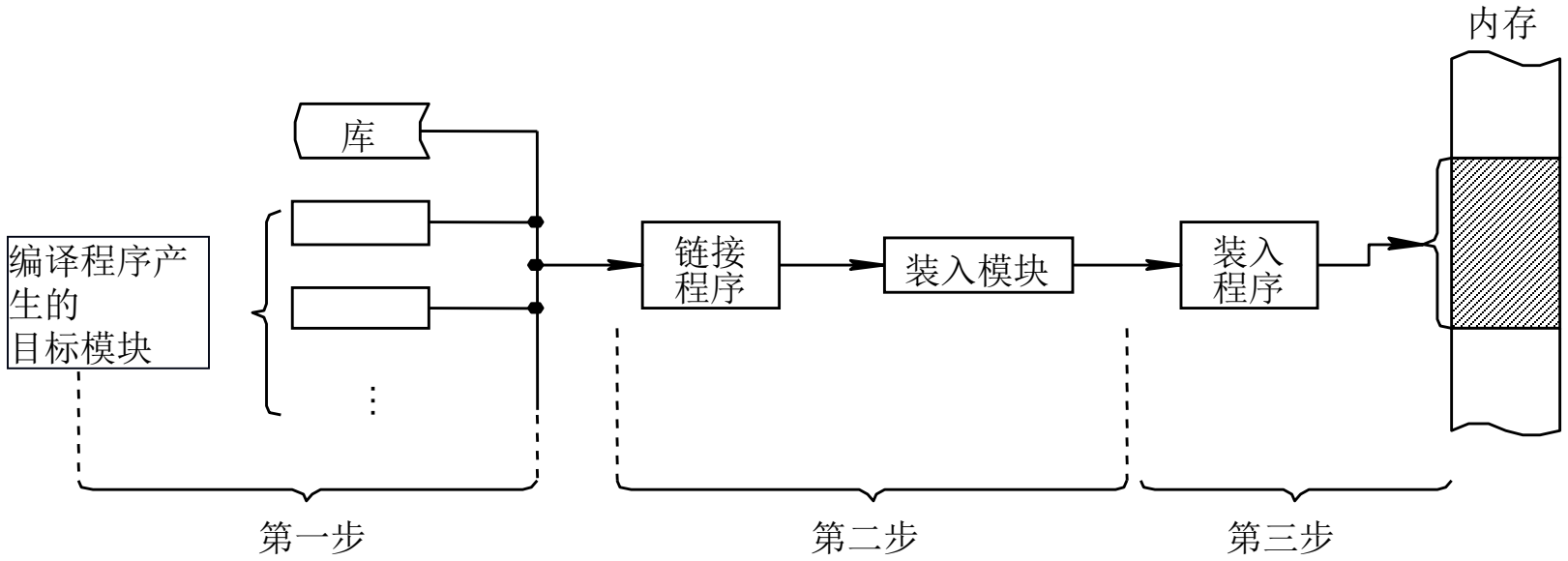
- 2 磁盘缓存
  - 主存中暂存对磁盘的读写信息
- （硬盘的缓冲区）
  - 硬盘的缓冲区是硬盘与外部总线交换数据的场所

存储器管理

- 存储器的层次结构
- 程序的装入和链接
- 连续分配方式
- 基本分页存储管理方式
- 基本分段存储管理方式

4.2 程序的装入和链接

- 将程序和数据装入内存
- 编译——链接——可装入模块



4.2.1 程序的装入

- 1. 绝对装入方式(Absolute Loading Mode
  - 程序中所使用的绝对地址既可在编译或汇编时给出， 也可由程序员直接赋予
  - 由程序员直接给出绝对地址时
    - 要求程序员熟悉内存的使用情况，而且一旦程序或数据被修改后，可能要改变程序中的所有地址
  - 在程序中采用符号地址，然后在编译或汇编时，再将这些符号地址转换为绝对地址
  - 装入到事先指定的内存地址，只适用于单道程序系统

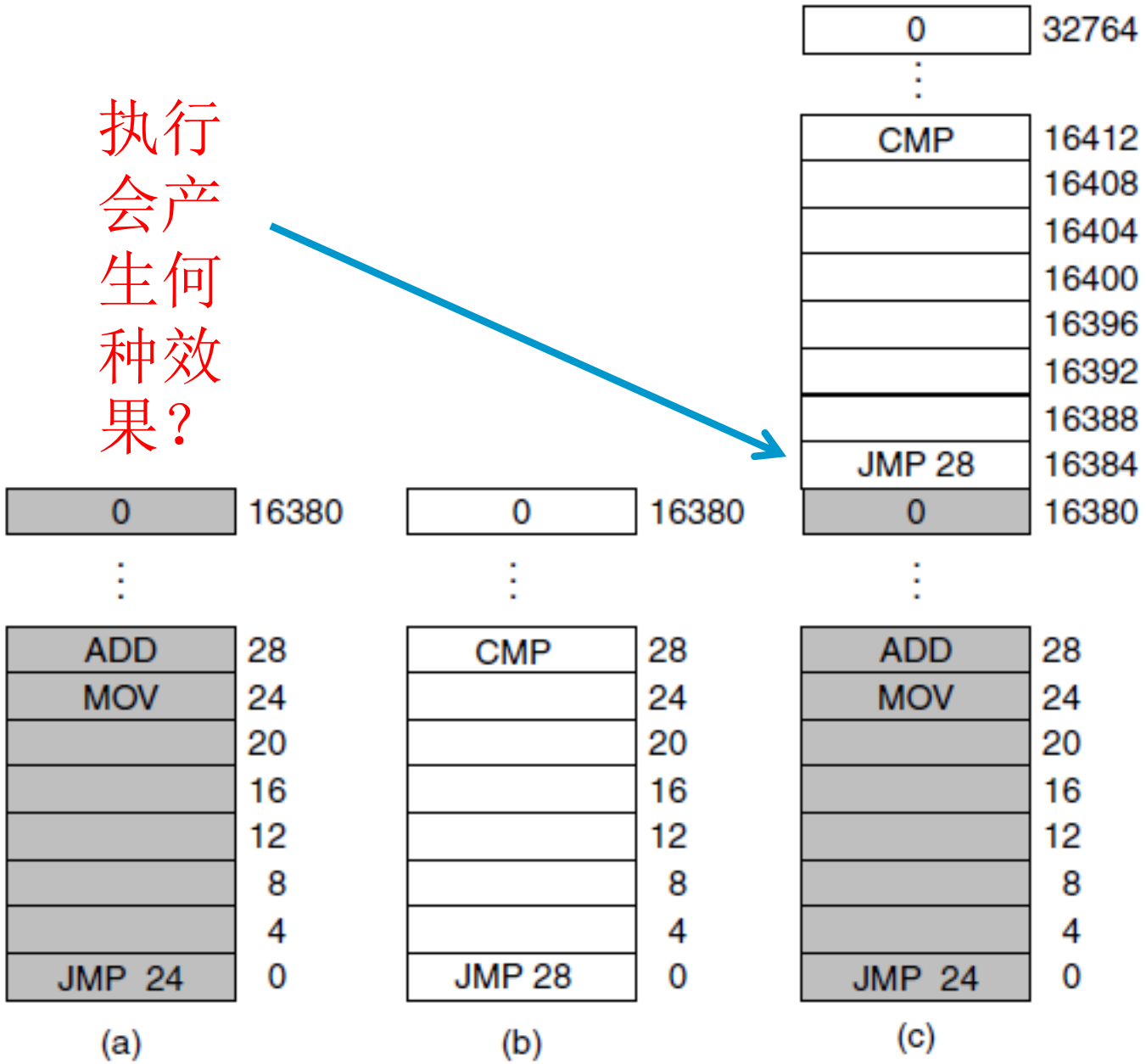
4.2.1 程序的装入

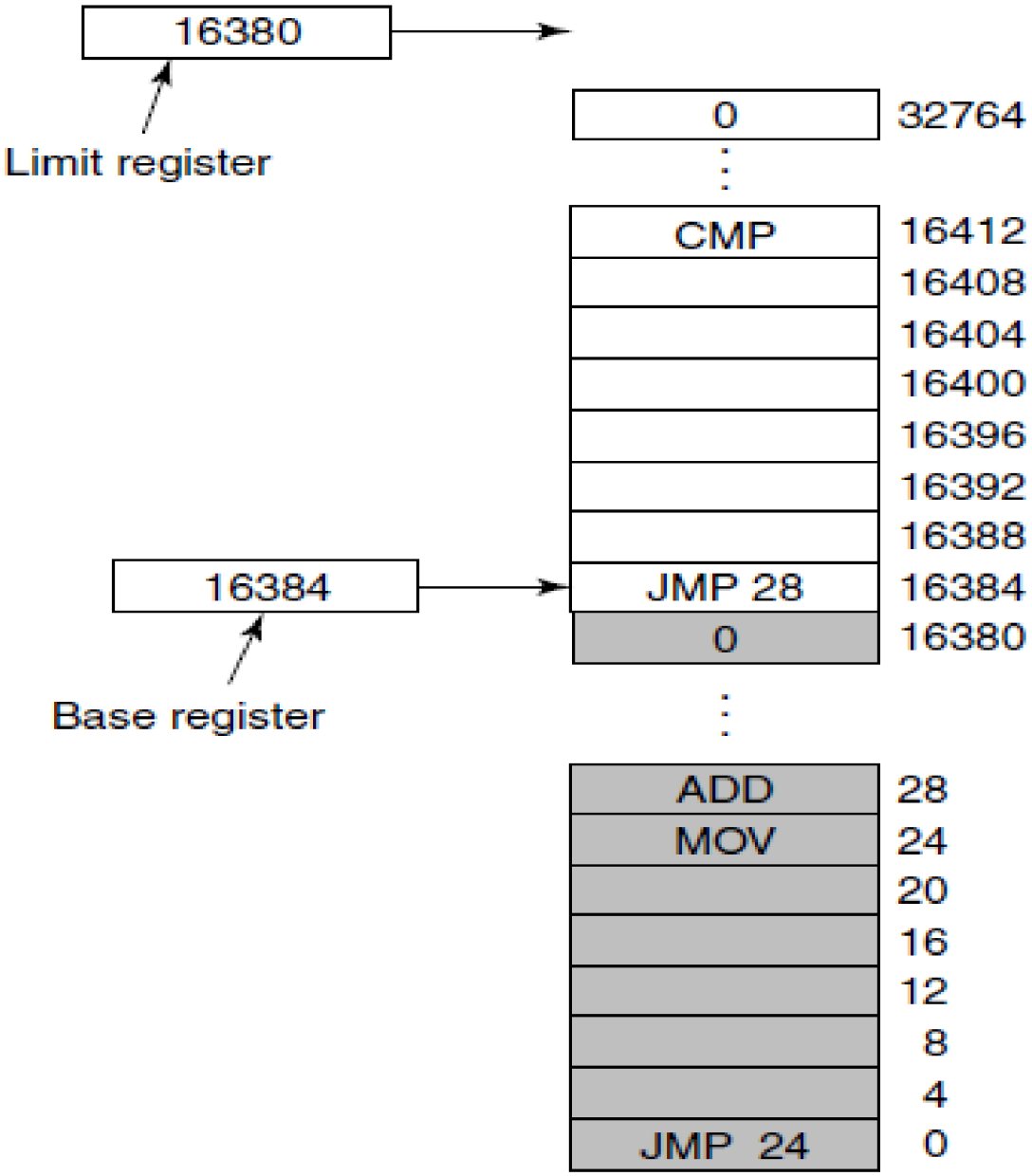
- 2 可重定位装入方式(Relocation Loading Mode)
  - 基本思想
    - 根据实际情况，装入到内存的适当位置
    - 程序装入时对目标程序中指令和数据的修改过程称为重定位。
    - 程序执行之前进行地址重定位；
    - 由操作系统的装配程序实现程序的相对地址到绝对地址的映射。
    - 在装入时一次完成，称为静态重定位。
  - 特点
    - 容易实现，无需硬件支持。
    - 程序在主存中只能连续分配；程序在运行期间不能被再移动。无法实现虚拟存储；
    - 用户之间难以共享主存的同一程序，若共享需要使用副本，浪费空间。

4.2.1 程序的装入

- 3 动态运行时装入方式 Dynamic Run-time Loading
  - 基本思想
    - 在程序运行期间进行重定位，即在CPU访问存储单元时才进行程序相对地址到物理地址之间的映射。
  - 技术实现
    - 必须硬件支持，如BR基址寄存器（占用内存空间的首地址），VR程序虚拟地址寄存器（虚拟地址）.内存地址＝BR+VR。
  - 优点
    - 目标模块装入主存时无需任何修改
    - 程序运行时也可以不把地址空间全部装入主存，可以在程序运行期间动态分配，因而主存使用更加灵活；
    - 几个进程可以共享程序的单个副本。

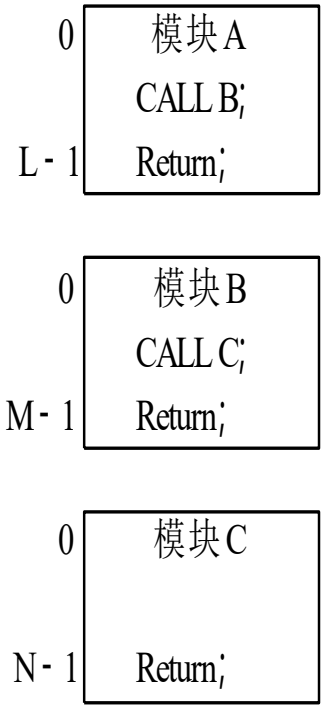
执行会产生何种效果？


0
32764
CMP
16412
16408
16404
16400
16396
16392
16388
JMP 28
16384
0
16380
ADD
28
MOV
24
20
16
12
8
4
JMP 24
0
(a)
0
16380
CMP
28
24
20
16
12
8
4
JMP 28
0
(b)
0
16380
ADD
28
MOV
24
20
16
12
8
4
JMP 24
0
(c)

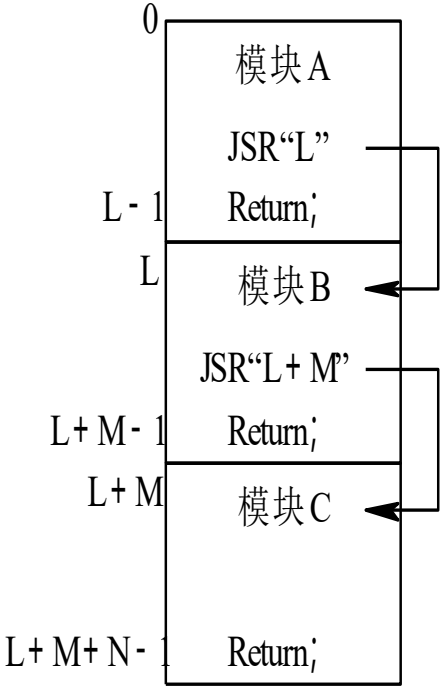

16380
Limit register
0
32764
CMP
16412
16408
16404
16400
16396
16392
16388
16384
JMP 28
16384
0
16380
Base register
ADD
28
MOV
24
20
16
12
8
4
JMP 24
0

4.2.2 程序的链接

- 程序编译后得到一组目标模块，链接程序将这组目标模块链接后形成转入模块
- 1. 静态链接方式 (Static Linking)
  - (1) 对相对地址（模块内逻辑地址）进行修改
  - (2) 变换外部调用符号
  - 预先链接成一个完整的可装入模块，一般称为可执行文件。



(*a*) 目标模块



(*b*) 装入模块

4.2.2 程序的链接

- 2. 装入时动态链接(Loadtime Dynamic Linking)
  - 装入内存时再链接
  - (1) 便于修改和更新
  - (2) 便于实现对目标模块的共享。

4.2.2 程序的链接

- 3. 运行时动态链接(Run-time Dynamic Linking)
  - 将对某些模块的链接推迟到执行时才执行
  - 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， 把它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，这样不仅可加快程序的装入过程，而且可节省大量的内存空间。

存储器管理

- 存储器的层次结构
- 程序的装如和链接
- 连续分配方式
- 基本分页存储管理方式
- 基本分段存储管理方式

4.3 连续分配方式

为一个用户程序分配一个连续的内存空间

4.3.1 单一连续分配

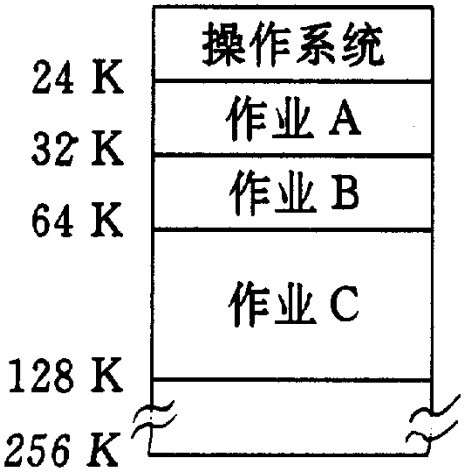
- 只能用于单用户、单任务的操作系统中。
- 把内存分为系统区和用户区两部分，系统区仅提供给OS使用；用户区是指除系统区以外的全部内存空间， 提供给用户使用。
- 简单容易实现，但浪费存储资源。

4.3.2 固定分区分配

- 将内存用户区划分为若干个固定大小的区域，每个区域装入一道作业
- 为实现分区的保护，系统硬件需设置基址和界限寄存器

分区号	大小(K)	起址(K)	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	已分配

(a) 分区说明表



(b) 存储空间分配情况

4.3.3 动态分区分配

- 根据进程的实际需要，动态地为其分配合适大小的内存空间
- 1 分区分配中的数据结构
  - 如何描述空闲分区和已分配的分区
  - (1)空闲分区表
    - 每个空闲分区占一个表目，分区序号、始址、长度
  - (2) 空闲分区链
    - 所有空闲分区链成双向链表

4.3.3 动态分区分配

- 2. 顺序搜索分区分配算法
  - (1) 首次适应算法
    - 基本思想
      - 空闲分区链按分区起始地址递增的顺序排列；
      - 内存请求时，从链表第一个分区控制块开始线性搜索，找到一个分区长度大于等于请求容量的空闲区，即进行分配。
      - 分配时从空闲区中划分出用户所要求的内存长度。

- 优点
  - 因空闲分区按实际地址递增排序，释放分区时易于合并相邻为较大空闲分区。
  - 该算法实质是尽可能利用存储器低端地址，而在高端地址保留较多或者较大空闲分区以满足后继大作业存储要求。
- 缺点
  - 低端地址难以利用的小空闲分区增多
  - 分区分配时搜索次数增加。

4.3.3 动态分区分配

- (2) 循环首次适应算法(下次匹配算法)
  - 由首次适应算法演变而成的
  - 分配内存时，不是从链首搜索，而是从上次找到的空闲分区的下一个空闲分区开始查找，直到找到满足要求的空闲分区
- 采用循环查找的方式
- 内存中空闲分区分布得会更均匀，减少了查找空闲分区的时间开销，但缺乏大的空闲分区

4.3.3 动态分区分配

- (3) 最佳适应算法
  - 从空闲分区中找到能满足要求容量的最小分区；
  - 为提高速度，空闲区的可用表或可用链表按分区从小到大递增顺序链接；
  - 分配时，从表头开始查找，第一个满足要求的空闲区。
- 特点：可以保留大分区；但剩余很小的块可能不可用。

4.3.3 动态分区分配

- (4) 最坏适应算法
  - 从最大空闲分区中裁取块分配；然后为该分区剩下的存储空间建立空闲分区控制块，插入可用表中。
  - 空闲分区按大小递减顺序链接。
  - 分配时比较可用表中的第一个空闲可用区，若大于等于需求内存大小，则分配相应大小空间，调整空闲区可用表。否则失败。

- 优点
  - 仅比较一次就可确定是否能满足作业要求的容量；分配后剩余空闲空间可能较大可再次利用。
- 缺点
  - 各空闲分区均匀减小，当有大作业时可能难以满足；释放分区时合并相邻空闲分区也困难

4.3.3 动态分区分配

- 3. 索引搜索分区分配算法
  - 快速适应算法。
    - 分类搜索算法
    - 根据空闲分区的容量分类，容量一般取进程常用的空间大小；每类设一个空闲分区链表
    - 管理索引表：记录各类型空闲分区表的头指针
    - 分配内存时，根据需求内存长度，查管理索引表，找到满足要求的最小空闲分区链表，取下第一块分配。
    - 查找效率高；空闲区分配时不对分区分割
    - 分区回收复杂；一个分区属于一个进程，产生内部碎片
    - 空间换时间的方法

4.3.3 动态分区分配

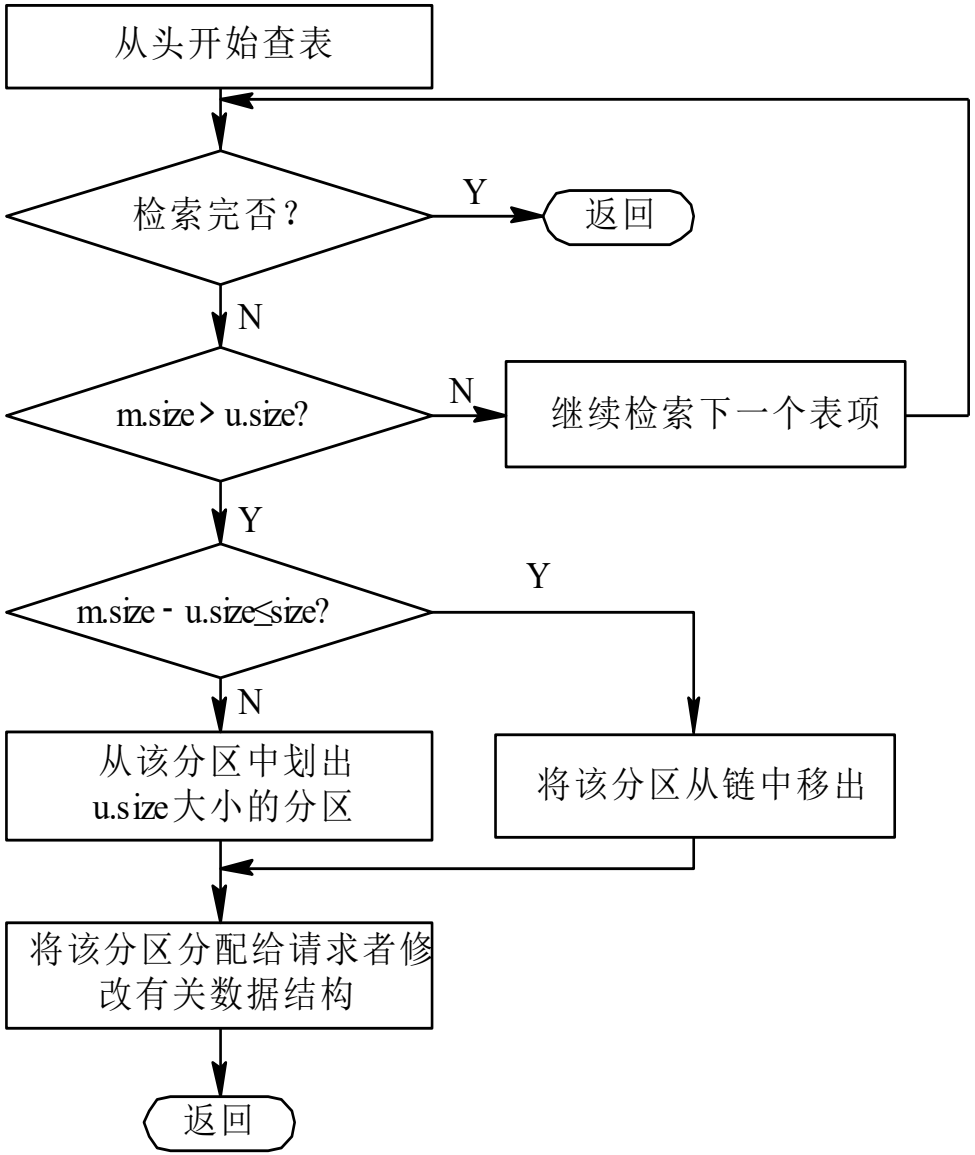
伙伴系统

2的幂次大小, 启始地址x, 大小2^k块的伙伴

$$\mathrm{Buddy}_k(x)=\begin{cases} x+2^k & (\text{若 } x \bmod 2^{k+1}=0) \\ x-2^k & (\text{若 } x \bmod 2^{k+1}=2^k) \end{cases}$$

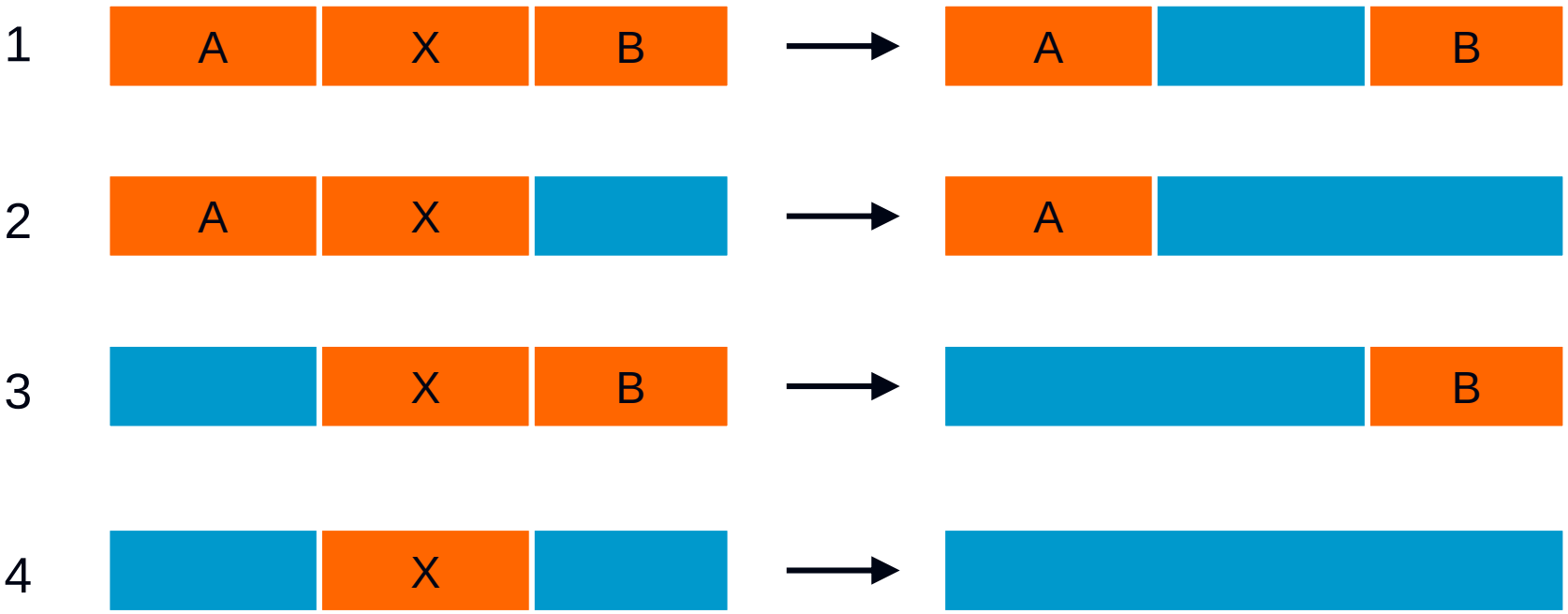
4.3.3 动态分区分配

- 如何分配内存和回收内存
- 1) 分配内存
  - 请求分区大小：u.size
  - 空闲分区大小：m.size



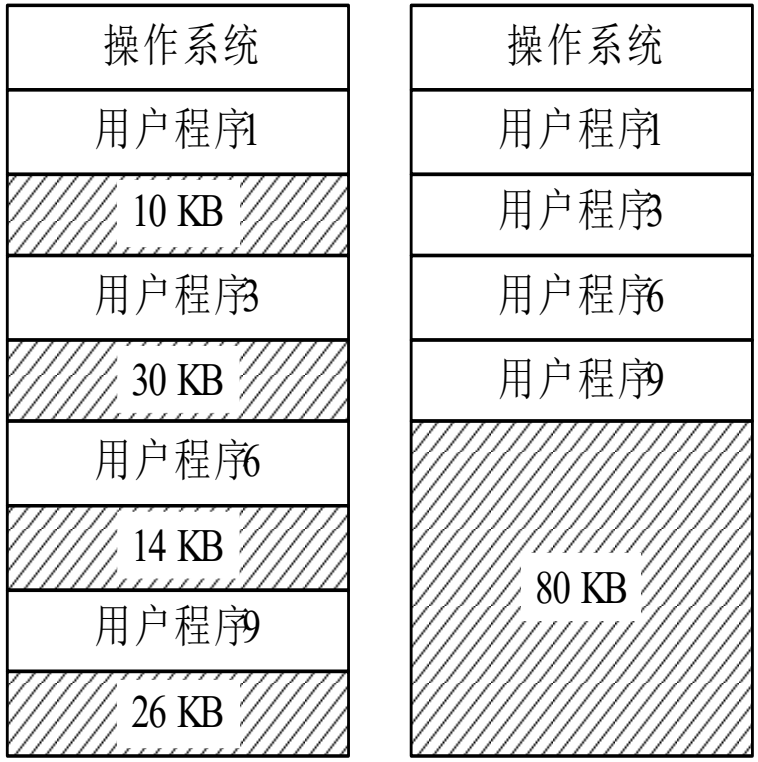
4.3.3 动态分区分配

- 2) 回收内存
  - （分区的合并）



4.3.4 可重定位分区分配

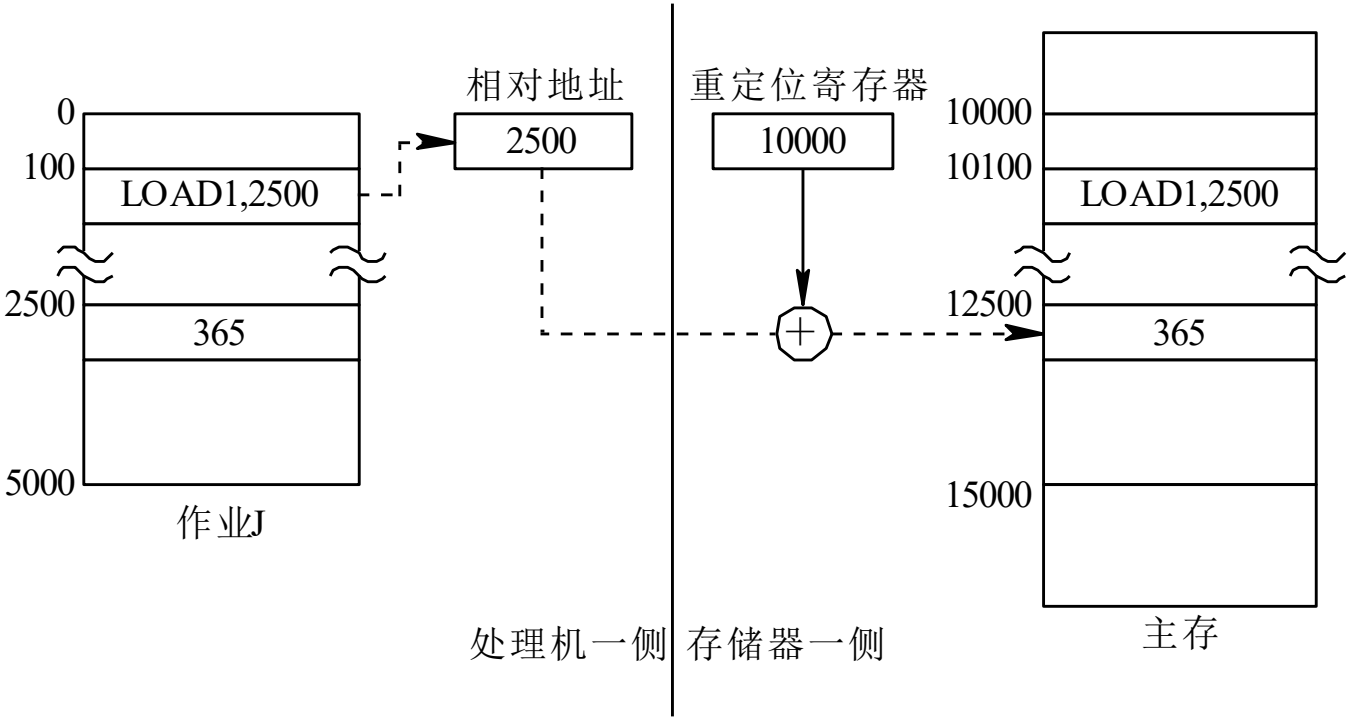
- 1. 动态重定位的引入
  - 进程连续分配需要连续的内存空间
  - 碎片，零头
    - 系统中小的空闲分区的总和大于要装入的作业，但因不连续而无法使用
  - 紧凑
    - 作业移动，把分散的小分区拼接成大分区
  - 紧凑后程序的内存地址发生了变化，必须修改地址，即重定位



(*a*) 紧凑前　　(*b*) 紧凑后

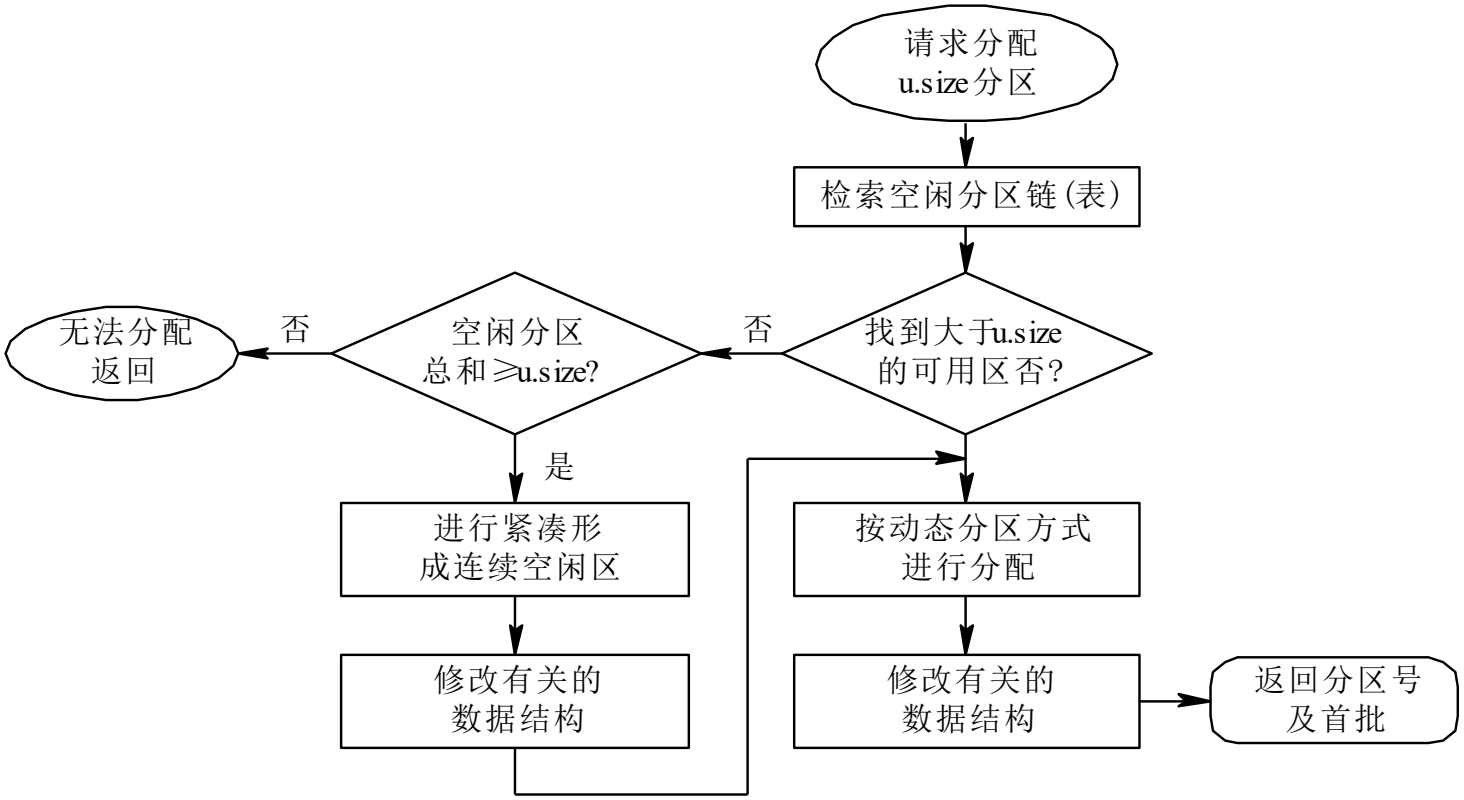
4.3.4 可重定位分区分配

- 2. 动态重定位的实现
  - 程序装入后仍然为相对地址，在指令执行时再变换为物理地址
  - 重定位寄存器
    - 开始地址
  - 不需要修改程序



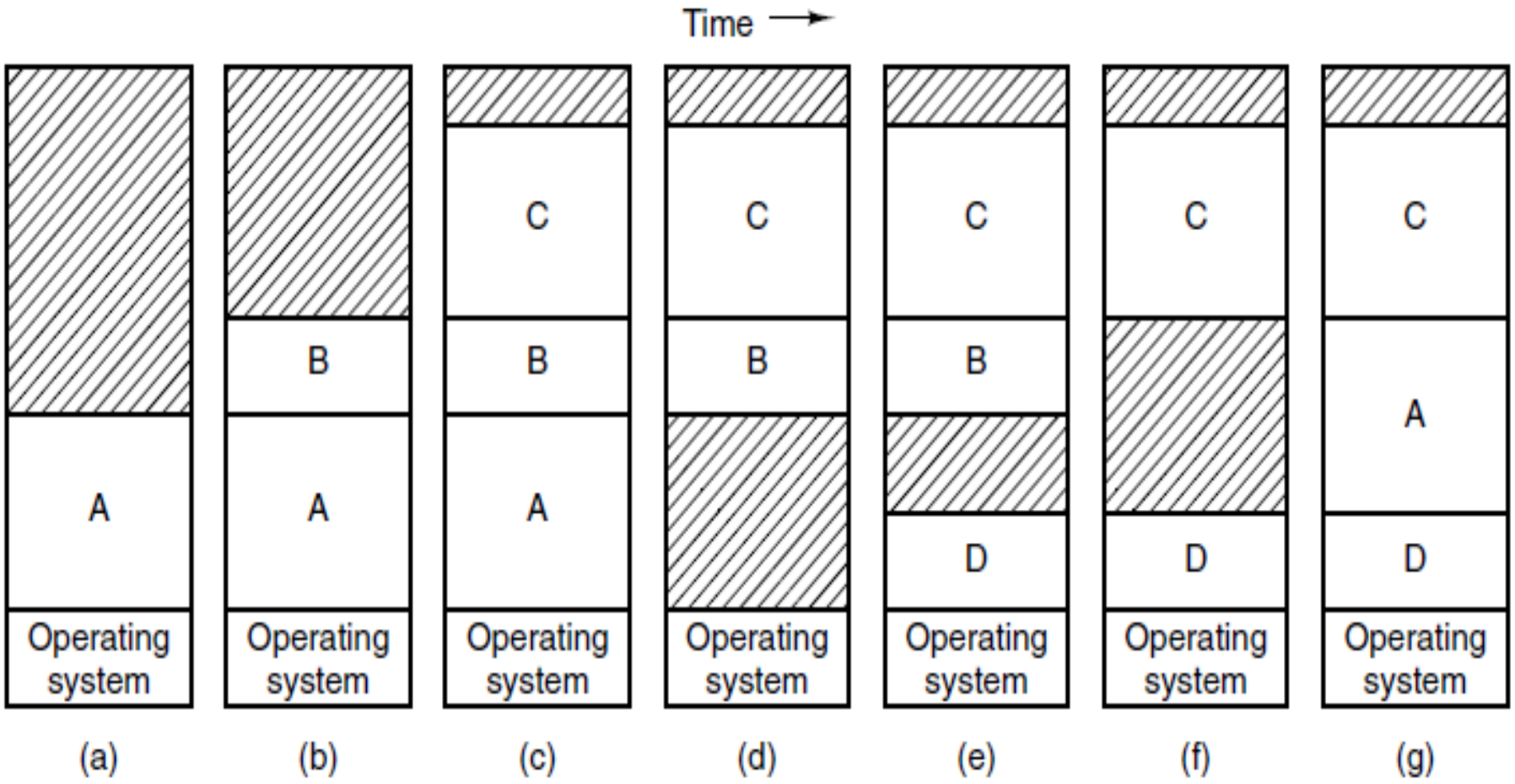
4.3.4 可重定位分区分配

- 3. 动态重定位分区分配算法
  - 无足够大空闲分区时需要紧凑



4.3.5 对换(Swapping)

- 1. 对换的引入
  - 阻塞进程占用内存，浪费资源
  - 把内存中暂时不能运行的进程或者暂时不用的程序和数据，调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据，调入内存。
  - 整体对换，部分对换
  - 提高内存利用率的有效措施。



一个对换过程的示意图

4.3.5 对换(Swapping)

- 2. 对换空间的管理
  - 外存分为文件区和对换区
  - 文件区离散分配；对换区连续分配
  - 对换区管理与内存动态分区管理类似

4.3.5 对换(Swapping)

- 3. 进程的换出与换入
  - (1) 进程的换出
    - 当一进程由于创建子进程而需要更多的内存空间，但又无足够的内存空间等情况发生时，系统应将某进程换出。
    - 选择处于阻塞状态且优先级最低的进程作为换出进程。将该进程的程序和数据传送到磁盘的对换区上。回收该进程所占用的内存空间，并对该进程的进程控制块做相应的修改。
  - (2) 进程的换入
    - 定时地查看所有进程的状态，从中找出“就绪”状态但已换出的进程，将其中换出时间最久的进程作为换入进程，将之换入，直至已无可换入的进程或无可换出的进程为止。