内存管理

内存管理是操作系统对内存的划分和动态分配

内存管理的目的

1. 为了更好地支持多道程序并发执行
2. 方便用户
3. 提高内存利用率

内存管理的功能

1.内存空间的分配与回收	**由OS完成主存储器空间的分配和管理**
2.地址转换	存储管理将逻辑地址转换为物理地址
3.内存空间的扩充	利用虚拟存储技术/自动覆盖技术，从逻辑上扩充内存
4.内存共享	允许多个进程访问内存的同一部分
4.存储保护	保证多道作业在各自的存储空间运行，互不干扰

内存管理的分配

1.连续分配	**单一连续分配-->单道发展到多道OS-->固定分区分配--为了适应大小不同的程序-->动态分区分配**
2.不连续分配	分段存储管理--->分页存储管理--->段页存储管理

程序的链接和装入

编译	编译：由编译程序将用户源代码编译成若干目标模块
链接	链接：由链接程序将目标模块和库函数链接，形成完整的装入模块 链接类别 - 静态链接 - 动态链接 - 运行时动态链接
装入	装入：是由装入程序将装入模块装入内存运行 装入类别 - 静态装入：在编程时把物理地址计算好

	- 可重定位装入：装入时把逻辑地址转换为物理地址，但装入后不能改变 - 动态重定位装入：执行时再决定装入的地址并装入，装入后有可能会换出

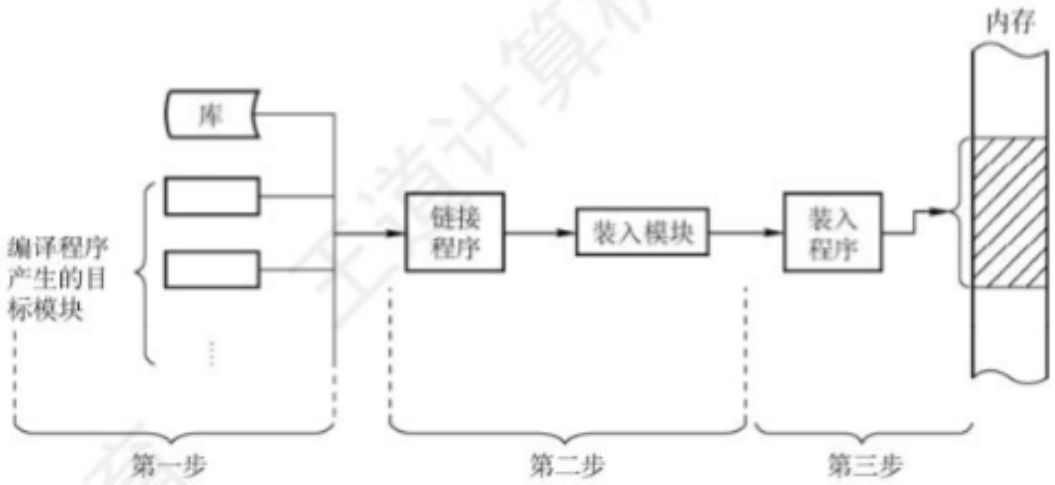


图 3.1　将用户程序变为可在内存中执行的程序的步骤

逻辑地址和物理地址

	逻辑地址	物理地址
定义	每个目标模块都从0号单元开始编址的地址	物理地址空间是指内存中物理单元的集合
特点	- 不同进程可以有相同的逻辑地址，这些逻辑地址可以映射到主存的不同位置 - 进程运行时，看到和使用的地址都是逻辑地址 - 将逻辑地址转换为物理地址的过程叫做地址重定位	- 物理地址是地址转换的最终地址

进程的内存映像

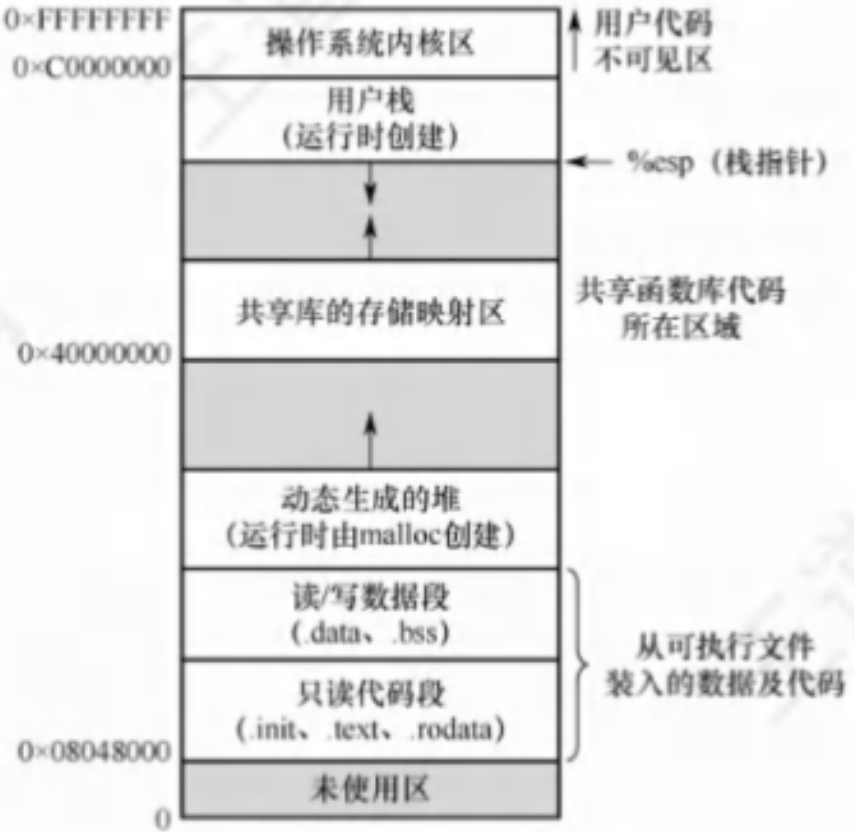


图 3.3　内存中的一个进程

1.代码段	• **代码段是只读的，可以被多个进程共享**
2.数据段	• 程序运行时加工处理的对象，包括全局变量和静态变量
3.进程控制块PCB	• 存放在系统区，OS通过PCB控制和管理进程
4.堆	• 用来存放动态分配的变量【动态的】
5.栈	• 用来实现函数调用的【动态的】

内存保护

确保每个进程都有单独的一个内存空间

方法

方法1.在CPU中设置一对上下限存储器，判断CPU访问的地址是否越界
方法2.使用重定位寄存器和界地址寄存器（只有OS内存才可以使用这两个寄存器）

- 重定位寄存器/基地址寄存器含最小的物理地址值【用于“加”】
- 界地址寄存器含逻辑机制的最大值【用于“比”】
- 逻辑地址+重定位寄存器的值=物理地址

内存共享

概念	- 只有只读区域的进程空间可用共享 - 纯代码/可重入代码 = 不能修改的代码，不属于临界资源 - 可重入程序通过减少交换数量来改善系统性能
实现方式	1.段的共享 2.基于共享内存的进程通信（第二章的同步互斥） 3.内存映射文件

内存管理方法

分页和分段的比较

	分段	分页
地址映射表	每个进程由多个不等的段组成	每个进程一张页表，且进程的页表驻留在内存中
地址结构	段号+段内偏移	页号+页内偏移
地址结构维度	二维	一维
供什么感知	供用户感知	供操作系统感知
以什么单位划分	- 以段为单位分配 - 每段是一个连续存储区 - 每段不一定等长 - 段与段之间可连续，也可不连续	- 逻辑地址分配按页分配 - 物理地址分配按内存块分配
长度是否固定	段长不固定	页长固定
访问主存次数	2次	2次
碎片情况	只产生外部碎片	产生内部碎片

连续分配

连续分配管理是为一个用户程序分配一个连续的内存空间

- 用户程序在主存中都是连续存放的
- 非连续分配的方式的存储密度 < 连续分配方式

碎片

- 内部碎片：当程序小于固定分区大小时，也要占用一个完整的内存分区，导致分区内部存在空间浪费
- 外部碎片：内存中产生的小内存块

分类

单一连续分配和固定分区分配

	1.单一连续分配	2.固定分区分配
定义	- 在此方法下，内存分为两个区 - 系统区：供OS用，在地址区 - 用户区：内存用户空间由一道程序独占	- 用户内存空间划分为固定大小(分区大小相等或不等)的区域 - 每个区装一道作业
优点	- 简单，无外部碎片 - 无需进行内存保护（内存中永远只有一道程序）	- 简单
缺点	- 只能用于单用户单任务的OS - 有内部碎片，存储器利用率极低	- 程序太大可能放不下任何一个分区，有内部碎片 - 不能实现多进程共享一个主存区，存储空间利用率低

动态分配

进程转入内存时，根据进程的实际需要，动态地分配内存；动态分区是在作业装入时动态建立的

动态分配算法	按什么次序链接的	特点
a.首次适应算法	按地址递增的次序	最简单，效果最好，速度最快
b.邻近适应算法		比首次适应算法差
c.最佳适应算法	按容量递增的次序	性能很差，会产生最多的外部碎片（也可以叫内存碎片，但是不能叫内部碎片）
d.最坏适应算法	按容量递减的次序	可能导致没有可用的大内存块，性能差

基于索引搜索的分配算法

按大小建立空闲分区表，对于每类空闲分区设置一个空闲分区链，设置一张索引表来管理，分区时，先查表，在得到对应的空闲分区链的头指针，从而获得空闲分区。

快速适应算法

根据进程长度->索引表->索引链表->第一块进行分配
优点：查找效率高，不产生内部碎片

缺点：合并回收区时，需要有效合并分区，算法复杂，系统开销大

伙伴系统

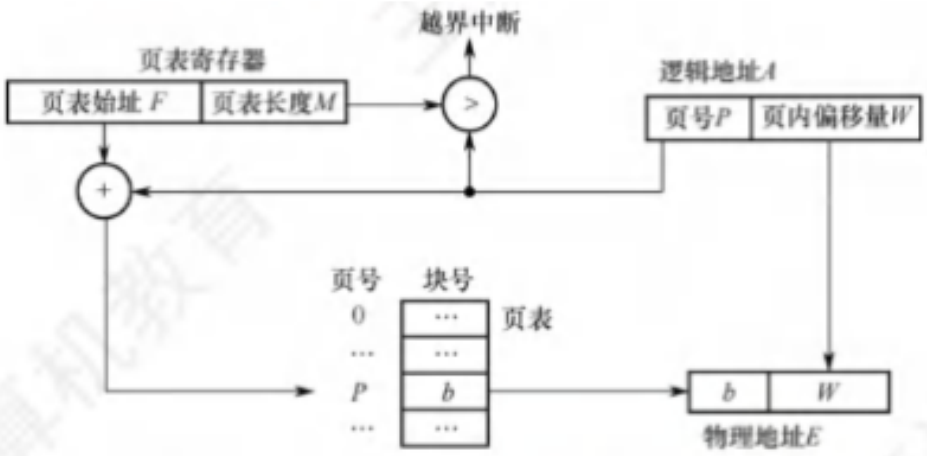
规定所有分区系统大小均为 2^k 当为进程分配大小为 n $2^{i-1} < n \le 2^i$ 的空间时,在大小为 2^i 的空闲分区链表查找，找到就分配，找不到，则去 2^{i+1}去查找，若存在则分为两个区，一个分配，一个插入i级链表。回收的时候，可能合并

哈希算法

构建以空闲分区大小为关键字的哈希表，根据哈希函数分配

分页

分页	**• 把整个虚拟和物理内存空间切成一段段固定尺寸的大小，在linux中，每一页的大小为4kB**
页/页面	• 页就是进程中的块
	• 页面大--->页内碎片增多，降低内存的利用率
	• 页面小--->进程的页面数大，页表过长，占用大量内存，增加硬件地址转换的开销，降低页面换入/出的效率
地址结构	• 地址结构 = 页号P+页内偏移量W
	• 地址结构决定了虚拟内存的寻址空间有多大
	• 完成地址转换工作的是有硬件的地址转换机构，而不是地址转换程序
页表	• 页表就是记录页面在内存中对应的物理块号
	• 页表的起始地址（是物理地址不是逻辑地址）放在页表基址寄存器PTBR中
	• 页表是存储在内存里的，内存管理单元 （MMU）就做将虚拟内存地址转换成物理地址的工作。

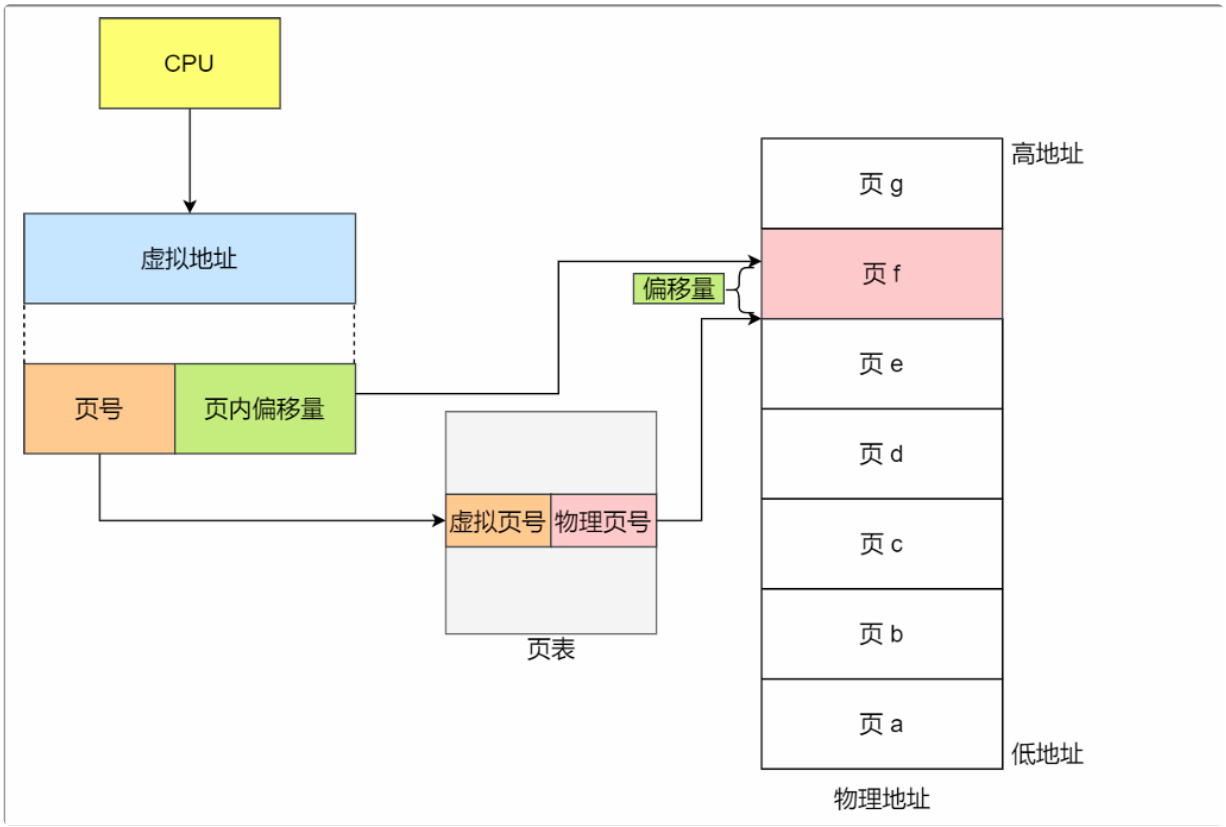


优点	缺点	特点
• 能有效地提高内存利用率	• 会产生内部碎片	• 逻辑地址分配按页分配
		• 物理地址分配按内存块分配
		• 划分的页面大小都相同
		• 所有进程都有一张页表
		• 页表存在内存中
		• 分页是面向计算机的
		• 整个系统设置一个页表寄存器用于存放页表在内存中起始地址和长度

分页的地址变换

基本地址变换

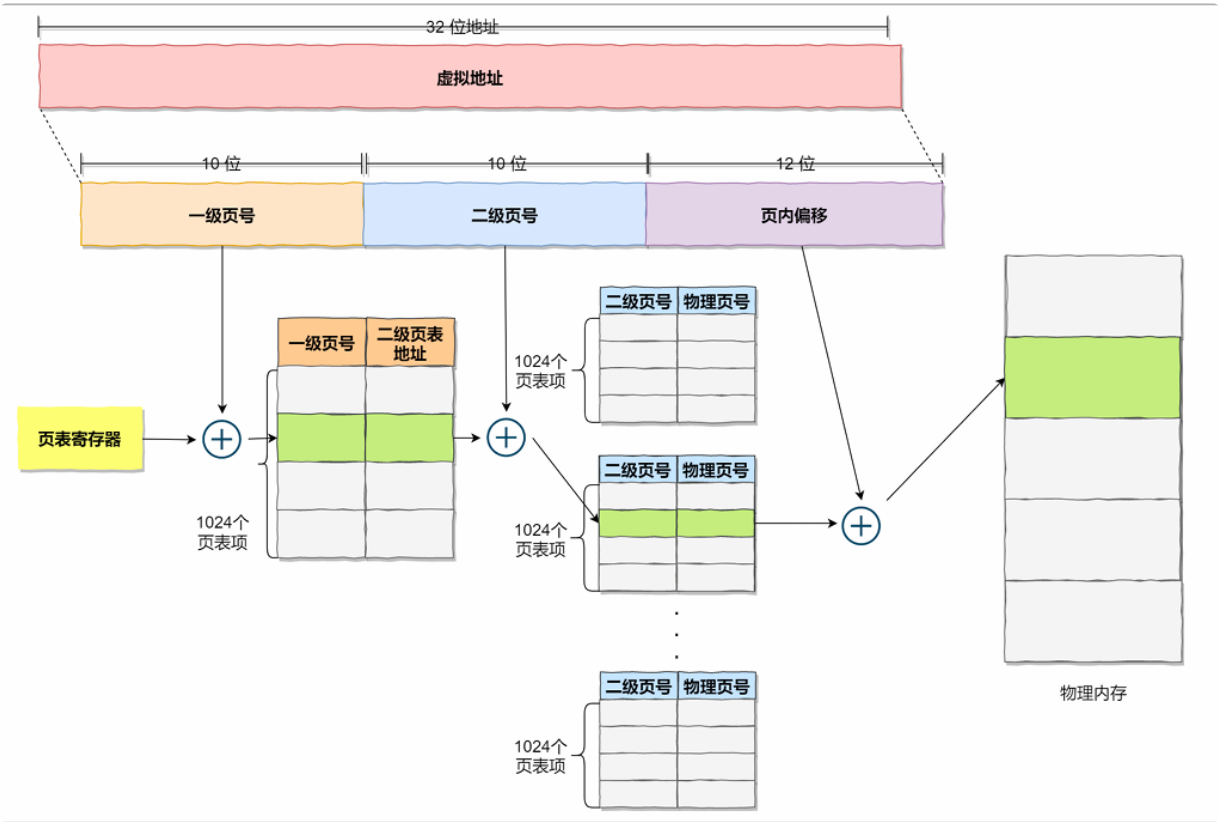
1. 把虚拟内存地址，切分成页号和偏移量
2. 根据页号，从页表里面，查询对应的物理页号
3. 直接拿物理页号，加上前面的偏移量，就得到了物理内存地址



两级页表变换

1. 一级页表覆盖到全部虚拟地址空间，二级页表在需要时创建

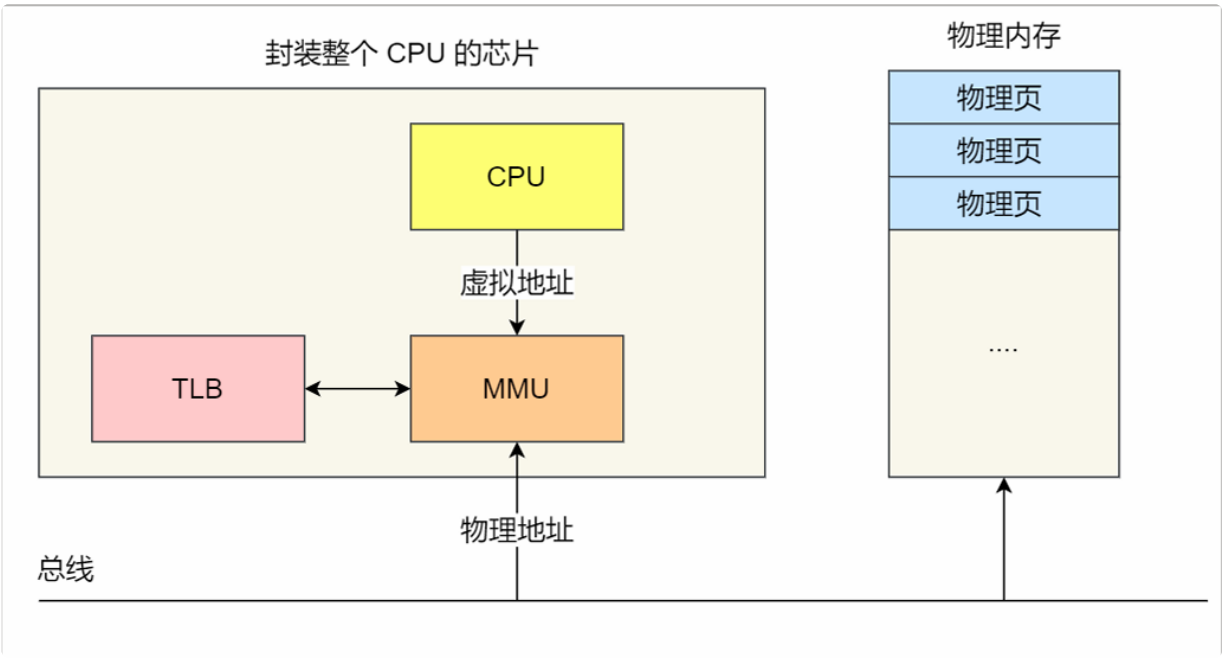
2. 建立多级页表的目的在于建立索引，以便不用浪费主存空间区存储无用的页表项，也不用盲目地顺序式查找页表项
3. 页表寄存器存放的是一级页表起始物理地



具有快表的地址变换

- 快表是相联存储器(Translation Lookaside Buffer，TLB)
- 快表也叫页表缓存，转址旁路缓存
- 快表专门存放程序最常访问的页表项的Cache
- 快表位于CPU芯片中，用于加速地址变换的过程
- CPU芯片中，封装了MMU(内存管理单元)

- MMU用来完成地址转换和TBL的访问与交互



分段存储

分段 = 程序由若干个逻辑分段组成，不同的段有不同的属性，用分段的方式把程序进行分离
段表 = 一张逻辑空间与内存空间映射的表

地址变换结构

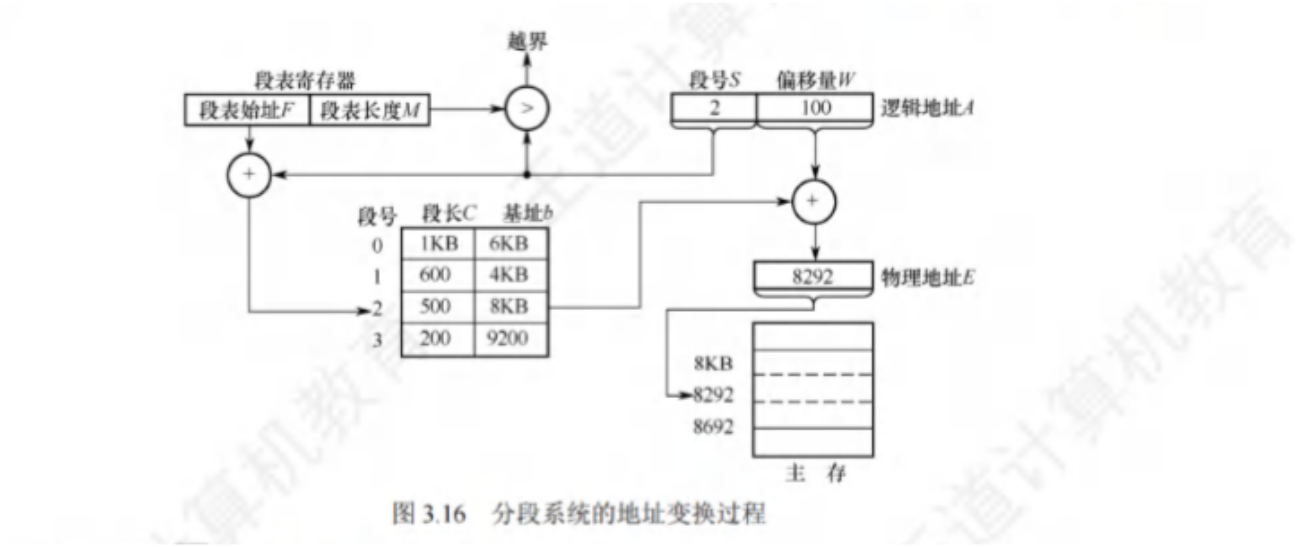


图 3.16 分段系统的地址变换过程

段的共享和保护

- 段的共享 = 通过两个作业的段表中相应表项指向被共享的段的同一物理副本
- 段的保护有两种
  - 存储控制保护

- 地址越界保护

优点	缺点
- 能产生连续的内存空间 - 分段存储管理能反映程序的逻辑结构并有利于段的共享和保护 - 程序的动态链接与逻辑结构有关，分段存储管理有利于程序的动态链接	- 会产生外部碎片 - 内存交换的效率低

段页

step1：将程序划分为多个有逻辑意义的段【分段】
step2：对分段划分出来的连续空间，再划分固定大小的页【分页】

地址变换

- 作业的逻辑地址划分为：段号，段内页号，页内偏移量
- 对内存的管理以存储块为单位，地址空间是二维的

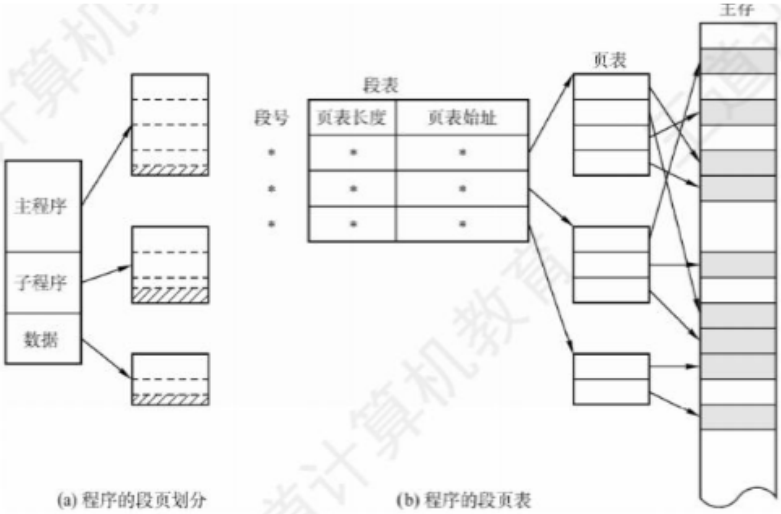


图 3.17　段页式管理方式

在段页式系统中，进程的逻辑地址分为三部分：段号、页号和页内偏移量，如图 3.18 所示。

图 3.18　段页式系统的逻辑地址结构

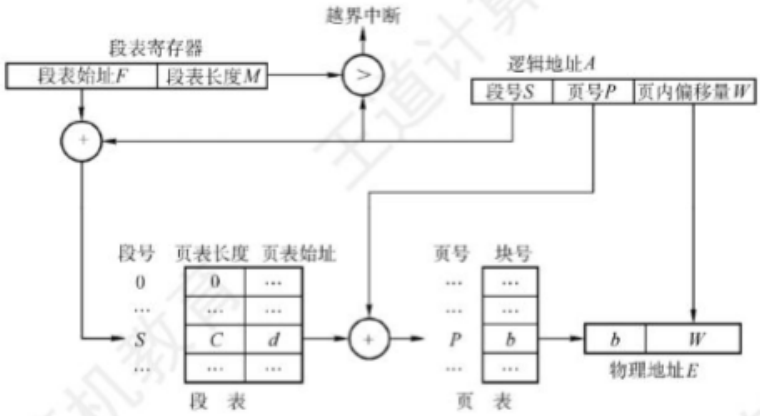


图 3.19 段页式系统的地址变换机构

虚拟内存管理

Abstract

- 虚拟存储器的最大容量是由计算机的地址结构决定的，与主存容量和外存容量没有关系
- 虚拟存储技术基于程序的局部性原理，局部性越好，虚拟存储系统越能更好地发挥作用
- 虚拟存储技术只能基于非连续分配技术
- 使用覆盖，交换方法可以实现虚拟存储

基本概念

局部性原理

- 时间局部性是指如果程序中的某条指令一旦执行，则不久之后该指令可能再次被执行；如果某数据被访问，则不久之后该数据可能再次被访问
- 空间局部性是指一旦程序访问了某个存储单元，则不久之后。其附近的存储单元也将被访问

虚拟存储器

定义	**• 系统为用户提供的一个比实际内存容量大得多的存储器**
特征	• 多次性 = 即只需将当前运行的那部分程序和数据装入内存即可开始运行【最重要的特征】
	• 对换性 = 即作业无需一直常驻内存，要用时换入，不要用时换出
	• 虚拟性 = 从逻辑上扩充内存的容量【最重要的目标】

实现方法

方式(离散分配)	**1.请求分页存储管理**
	2.请求分段存储管理
	3.请求段页式存储管理
需要的东西	• 一定的硬件支持，一定容量的内存和外存
	• 页表/段表机制作为主要的数据结构
	• 中断机制，当程序要访问的部分还未调入内存时，产生中断
	• 地址变换机构

请求分页管理方式

页表=页号+页框号+状态位P+访问字段A+修改位M+外存地址L

状态位/合法位P	**标记该页是否已被调入内存中**	**供程序访问时参考，用于判断是否触发缺页异常**
访问字段A	记录本页在一段时间内被访问的次数	供置换算法换出页面时参考
修改位M	标识该页在调入内存后是否被修改过	当页面被淘汰时，若页面数据没有修改，则不用写回外存
外存地址L	用于指出该页在外存上的地址，通常是物理块号	供写回外存和从外存中调入此页时参考

缺页中断

定义	**• 缺页是在CPU执行某条指令过程中，进行取指令或读写数据时发生的一种故障，是内中断或者叫做异常**
产生时间	• 每当要访问的页面不在内存中时，便产生一个缺页中断，请求OS将所缺的页调入内存
	• 缺页中断是访存指令引起的，说明所要访问的页面不在内存中
	• 进行缺页中断处理并调入所要访问的页后，访存指令应该重新执行
特点	• 在指令执行期间而非一条指令执行完后产生和处理中断信号
	• 一条指令在执行期间，可能产生多次缺页中断

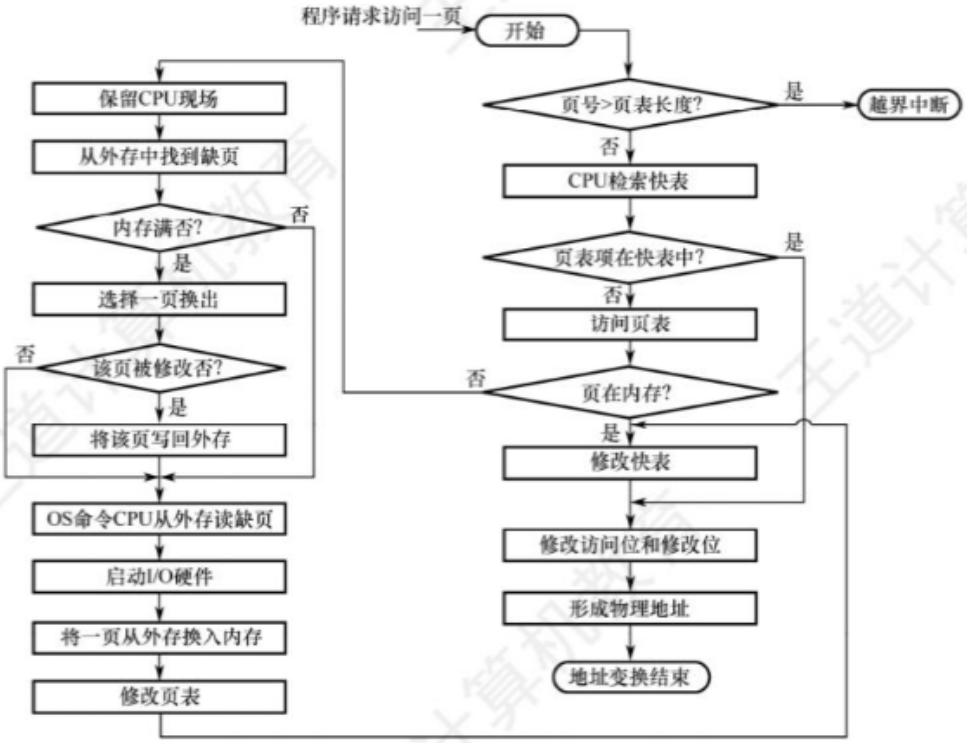


图 3.21　请求分页中的地址变换过程

页框分配

驻留集 给一个进程分配的物理页框的集合

驻留集大小

1. 分配给进程的页框越少，驻留在内存的进程就越多，CPU的利用率就越高
2. 进程在主存中的页面过少，缺页率相对较高
3. 分配的页框过多，对进程的缺页率没有大的影响

分配策略

固定分配局部置换	**·物理块固定，缺页时先换出一个线程再调入所缺页**
可变分配全局置换	·物理块可变，缺页时增加物理块再调入所缺页
可变分配局部置换	·物理块可变，若不频繁缺页则用局部置换，频繁缺页再用全局置换

物理块分配算法

1.平均分配算法

2.按比例分配算法

3.优先权分配算法

调入页面的时机

预调页策略 = 运行前的调入	**• 主要用于进程的首次调入，由程序员指出应先调入哪些页**
请求调页策略 = 运行时的调入	• 调入的页一定会被访问，策略易于实现
	• 每次仅调入一页，增加了磁盘I/O开销

请求分页系统的外存组成

预调页策略 = 运行前的调入	**• 主要用于进程的首次调入，由程序员指出应先调入哪些页**
请求调页策略 = 运行时的调入	• 调入的页一定会被访问，策略易于实现
	• 每次仅调入一页，增加了磁盘I/O开销

如何调入页面

情况1：所访问的页面不在内存时--->缺页中断--->无空闲物理块--->决定淘汰页--->调出页面--->调入所缺页面

情况2：所访问的页面不在内存时--->缺页中断--->有空闲物理块--->调入所缺页面

页面置换算法

	最佳置换算法OPT	先进先出置换算法FIFO	最近最久未使用置换算法LRU	时钟置换算法CLOCK
英文全名	Optimal replacement	First In First Out	Least Recently Use	Clock
被淘汰的页面	以后永不使用的	在内存中驻留时间最久的页面	最近最长时间未访问过的页面	根据顺序找第一个访问位为0的页面 当指针指向访问位为1的页面时，先把访问位置0，再继续寻找
特点	- 基于队列实现的 - 该算法无法实现 - 只能用于评价其他算法	- 会出现Belady异常 （分配的物理块数增大但页故障数不减反增）	- 性能好，但实现复杂 - 需要寄存器和栈道硬件支持 - 堆栈类算法	- FIFO和LRU的结合 - 改进型CLOCK算法需要使用位和修改位

		- 性能差，但实现简单	- 耗费高因为要对所有页排序	