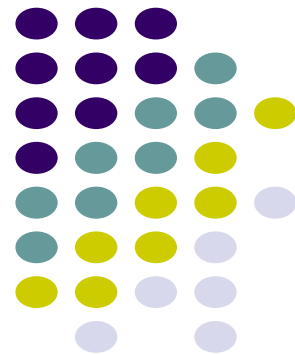


# 操作系统

## 第4章 存储器管理

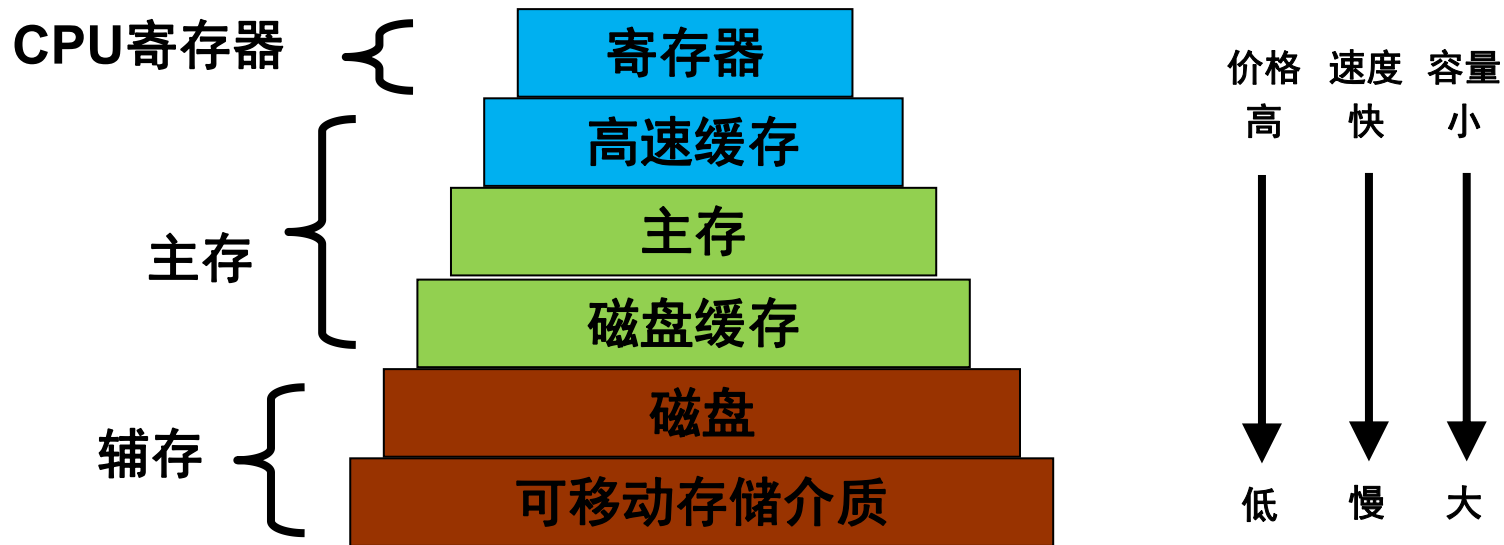




# 内容

- 4.1 程序的装入和链接
- 4.2 连续分配方式
- 4.3 基本分页存储管理方式
- 4.4 基本分段存储管理方式
- 4.5 虚拟存储器的基本概念
- 4.6 请求分页存储管理方式
- 4.7 页面置换算法
- 4.8 请求分段存储管理方式

# 存储器结构

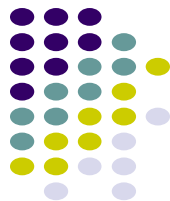


存储层次示意图

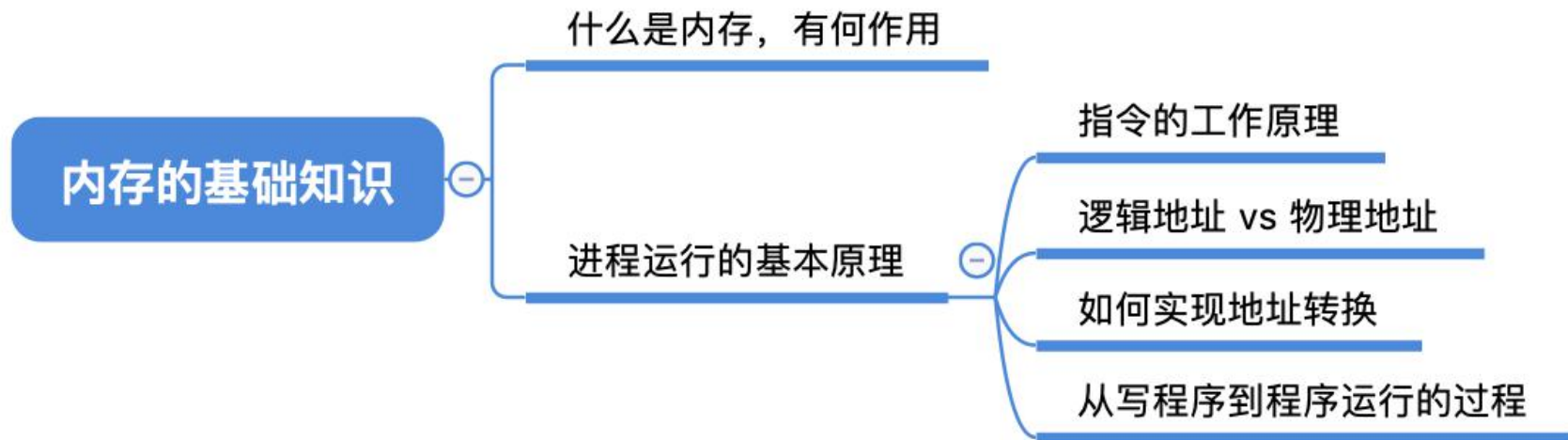


# 存储器的层次结构

- 主存储器
  - 主存也称可执行存储器。CPU可从其中取指令和数据，数据能从主存读取并装入到寄存器中，或从寄存器存入到主存。
- 寄存器
  - 访问速度最快，但价格昂贵
- 高速缓存cash
  - 容量远大于寄存器，但比主存小两三个数量级
- 磁盘缓存
  - 磁盘缓存本身并不是一种实际存储介质。
  - 实质：利用主存中的存储空间，来暂存从磁盘中读出或写入的信息



# 内存的基础知识



# 什么是内存？有何作用？





**新品** 华为 HUAWEI P30 超感光徕卡三摄麒麟980AI智能芯片全面屏屏内指纹版手机8GB+64GB亮黑色全网通双4G手机双

现在购机享受价保至6月18日、618提前购，安心购买！屏内指纹，感光徕卡三摄mate20优惠200、到手价3299起、还有赠品

**618 全球年中购物节**

京东价 **¥3988.00** 降价通知

促销 **满送** 满100元即赠热销商品，赠完即止

**满额返券** 购买此商品满10元返配件品类优惠券（送完为止）详情 >>

增值业务 **高价回收-卖了换钱** **3元1G**

配送至 **北京海淀区三环以内** 有货

由 **京东** 发货，并提供售后服务。23:00前下单，预计明天(05月28日)送达

重量 0.48kg

服务支持 **自营放心购** 免举证退换货 原厂维修 ①

京尊达 99元免基础运费(20kg内) 京准达 自提

选择颜色 **亮黑色** 天空之境 极光色 赤茶橘 珠光贝母

选择版本 **8GB+64GB** 8GB+128GB 8GB+256GB

选择版本 **标准版** 碎屏险套装版 京享无忧版 特惠版

套 装 **优惠套装1** 优惠套装2 优惠套装3 优惠套装4 优惠套装5 优惠套装6

< 关注 分享 对比

举报


# 什么是内存？有何作用？



电脑、办公 > 电脑整机 > 游戏本 > 外星人 (Alienware) > 外星人Alienware 17

自营 外星人京东自营旗舰店 联系客服 关注店铺

## ALIENWARE



ALIENWARE

GTX 1080

外星人17.3英寸机皇4K游戏笔记本电脑(i9-8950HK 32G 1T固态X2 1T GTX1080 8G独显 UHD)

【全尺寸机皇新高度】i9+GTX1080+4K超清屏,全方位无死角,双1TBSSD+1TBHDD,兼顾速度与容量。

**618 全球年中购物节**

京东价 **¥32999.00** 降价通知

累计评价 **1500+**















增值业务 [高价回收,享补贴](#)

配送至 北京海淀区三环以内 有货 支持 京尊达 99元免基础运费(20kg内)

由 京东 发货,并提供售后服务。有货 (外地跨区调货),暂免调货服务费。18:00前下单,预计**05月29日(周三)**送达

重量 7.8kg

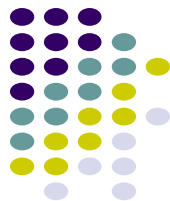
选择颜色

 i9-8950HK 16G RTX2080MQ 8G 红	 i7-8750H 16G RTX2080MQ 8G 银
 i7-8750H 16G RTX2070MQ 8G 银	 i7-8750H 16G RTX2060 6G 银
 i7-8750H 16G RTX2060 OC 6G独显	 i7-8750H 16G GTX1660Ti 6G独显
 i9-8950HK 32G GTX1080 8G独显	 i9-8950HK 32G GTX1080 8G UHD
 i7-7820HK 16G GTX1080 8G独显	 i9-8950HK 16G GTX1070 8G独显
 i7-8750H 16G GTX1060 6G独显 黑	 i7-8750H 16G GTX1070 8G独显
 i7-8750H 16G GTX1070 8G QHD	 i7-8750H 16G GTX1060 6G QHD

< 关注 分享 对比 举报

企业购更优惠

# 什么是内存？有何作用？



❤ 关注 分享 对比

举报

京东物流 三星 (SAMSUNG) DDR4 2400 2133 4G 8G 四代笔记本内存条 三星  
原厂正品 DDR4 2133 4GB

品牌机原厂内存供应商 买就买个真的的内存 京东自营仓直发 原厂正品内存 不烧机 吃鸡不蓝屏

京东价 ¥355.00 降价通知

累计评  
3700

促销 赠品 × 1 × 1 (赠完即止)

增值业务 以旧换新, 卖了换钱

配送至 北京海淀区五环到六环之间 有货 支持 货到付款 免运费

由 京东 发货, 本尚网来数码专营店 提供售后服务. 11:10前下单, 预计今天(08月03日)送达

选择颜色



增值保障



1

加入购物车





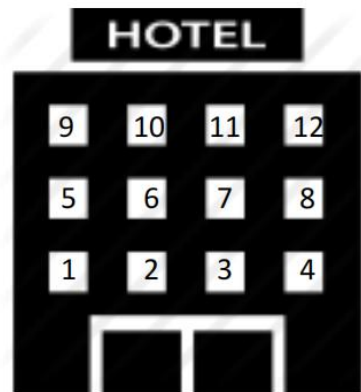
# 什么是内存？有何作用？

内存可存放数据。程序执行前需要**先放到内存中才能被CPU处理**——缓和CPU与硬盘之间的**速度矛盾**



思考：在多道程序环境下，系统中会有多个程序并发执行，也就是说会有多个程序的数据需要同时放到内存中。那么，如何区分各个程序的数据是放在什么地方的呢？

方案：给内存的存储单元编地址



内存地址从0开始，**每个地址对应一个存储单元**

地址	内存
0	“小房间”
1	“小房间”
2	.....
3	
4	

内存中也有一个一个的“小房间”，每个小房间就是一个“**存储单元**”

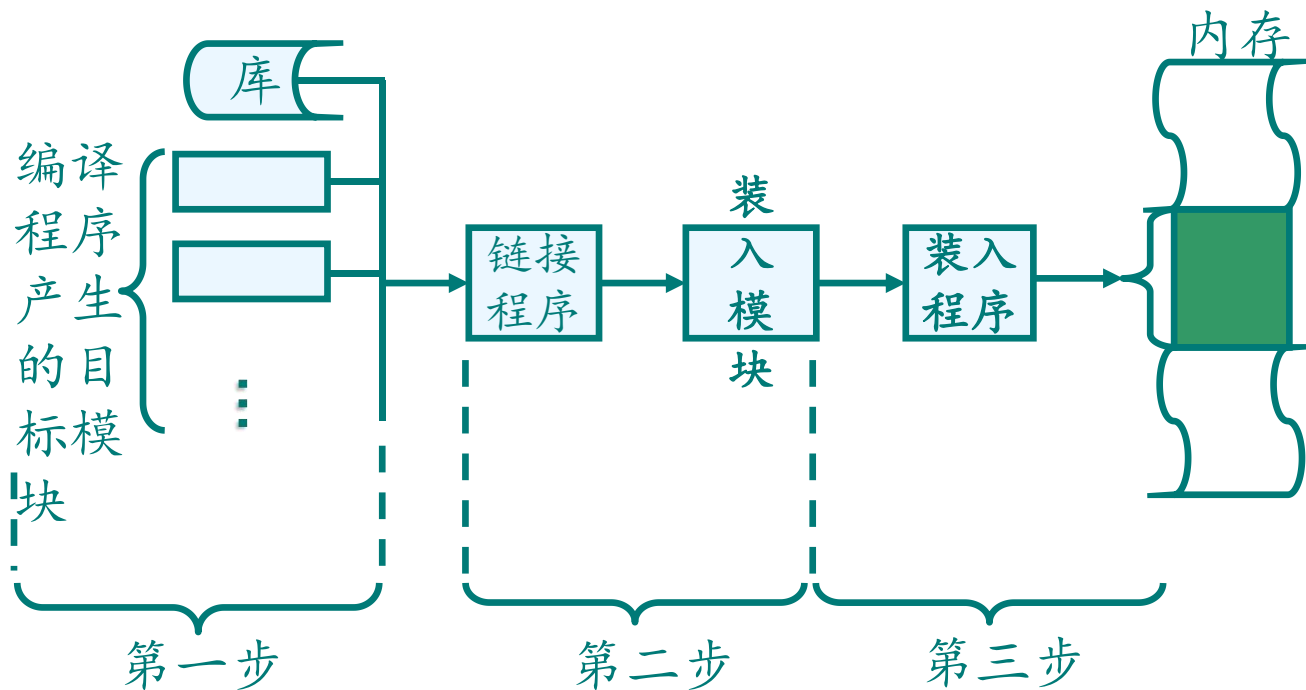
如果计算机“**按字节编址**”，则**每个存储单元大小**为**1字节**，即**1B**，即**8个二进制位**

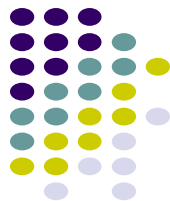


# 从源程序到程序执行

- 在多道程序环境下，要使程序运行，必须为之先建立进程。创建进程的第一件事是将程序和数据装入内存
- 将用户源程序变为可在内存中执行的程序的步骤：
  - **编译**：由编译程序将用户源代码编译成若干个目标模块
  - **链接**：由链接程序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
  - **装入**：由装入程序将装入模块装入内存

# 程序执行过程





# 指令的工作原理



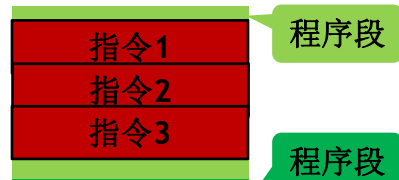
指令1 (00101100, 00000011, 01001111) 数据传送指令  
指令2 (10010010, 00000011, 00000001) 加法指令  
指令3 (00101100, 01001111, 00000011) 数据传送指令

某个寄存器：地址为 00000011 (3)  
(注：有的系统中，寄存器和内存 可能统一编址)

数据传送指令：(00101100, 00000011, 01001111)

11

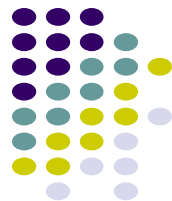
cpu



变量 x 的存放地址：  
01001111 (79)

可见，我们写的代码要翻译成CPU能识别的指令。这些指令会告诉CPU应该去内存的哪个地址读/写数据，这个数据应该做什么样的处理。在这个例子中，我们默认让这个进程的相关内容从地址#0开始连续存放，指令中的地址参数直接给出了变量 x 的实际存放地址（物理地址）。

**思考：**如果这个进程不是从地址#0 开始存放的，会影响指令的正常执行吗？



# 指令的工作原理

逻辑地址  
(相对地址)

程序经过编译、链接后生成的指令中指明的是逻辑地址（相对地址），即：相对于进程的起始地址而言的地址

1	指令0: 往地址为 79 的存储单元中写入 10
2	指令1: 把地址 79 中的 数据读入寄存器3
...	.....
179	.....

装入模块  
可执行文件 (\*.exe)

装入

物理地址  
(绝对地址)

1	指令0: 往地址为 79 的存储单元中写入 10
2	指令1: 把地址 79 中的 数据读入寄存器3
...	...
79	10
80	
...	
179	

变量 x  
存放的  
位置

C语言程序经过编译、链接处理后，生成装入模块，即可执行文件：  
`int x = 10;`  
`x = x+1;`

# 指令的工作原理

逻辑地址  
(相对地址)

程序经过编译、链接后生成的指令中指明的是逻辑地址（相对地址），即：相对于进程的起始地址而言的地址

1  
2  
...  
179

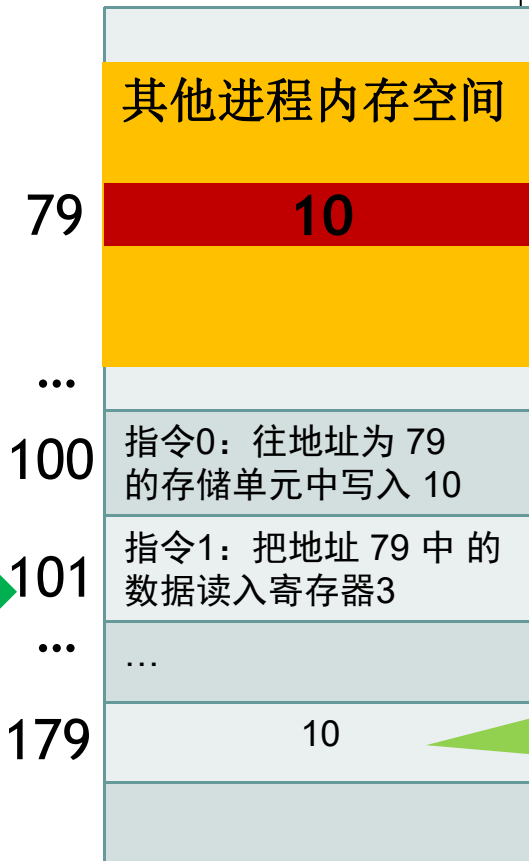
指令0: 往地址为 79 的存储单元中写入 10
指令1: 把地址 79 中的 数据读入寄存器3
.....
.....

装入模块  
可执行文件 (\*.exe)

装入

策略：三种  
装入方式

物理地址  
(绝对地址)



问题：如何  
将指令  
中的逻辑  
地址转换  
为物理地  
址？

变量 x  
存放的  
位置

C语言程序经过编译、链接处理后，生成装入模块，即可执行文件：  
`int x = 10;`  
`x = x+1;`



# 装入的三种方式——绝对装入

**绝对装入**：在编译时，如果知道程序将放到内存中的哪个位置，编译程序将产生绝对地址的目标代码。装入程序按照装入模块中的地址，将程序和数据装入内存。

**Eg**：如果知道装入模块要从地址为 100 的地方开始存放...

指令0：往地址为 79 的存储单元中写入 10
指令1：把地址 79 中的 数据读入寄存器3
.....
.....

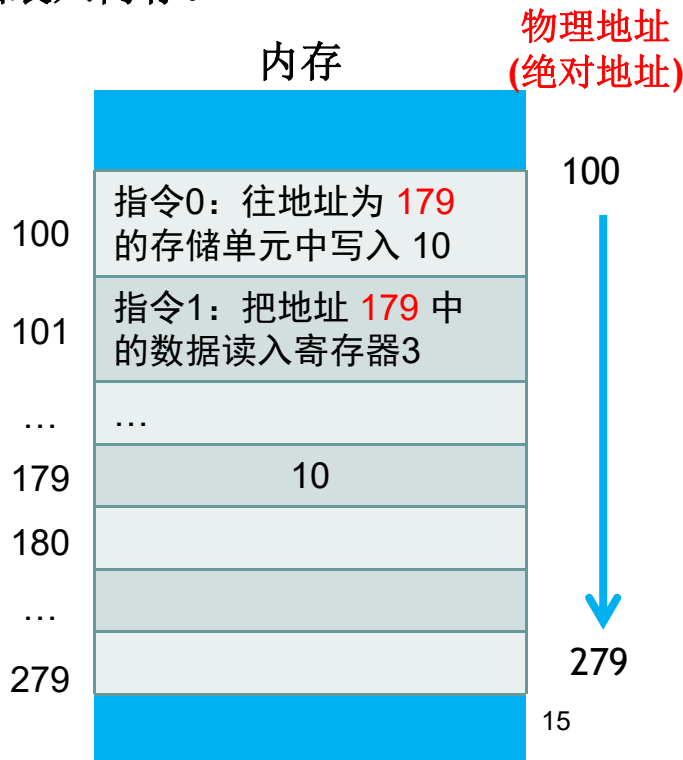


装入模块（可执行文件）

编译、链接后得到的装入模块的指令就直接使用了绝对地址

指令0：往地址为 179 的存储单元中写入 10
指令1：把地址 179 中的 数据读入寄存器3
.....
.....

装入模块（可执行文件）



绝对装入只适用于**单道程序环境**。程序中使用的绝对地址，可在编译或汇编时给出，也可由程序员直接赋予。通常情况下都是编译或汇编时再转换为绝对地址。



# 装入的三种方式——可重定位装入

**静态重定位**：又称**可重定位装入**。编译、链接后的装入模块的地址都是从0开始的，指令中使用的地址、数据存放的地址都是相对于起始地址而言的逻辑地址。可根据内存的当前情况，将装入模块装入到内存的适当位置。装入时对地址进行“**重定位**”，将逻辑地址变换为物理地址（**地址变换是在装入时一次完成的**）。

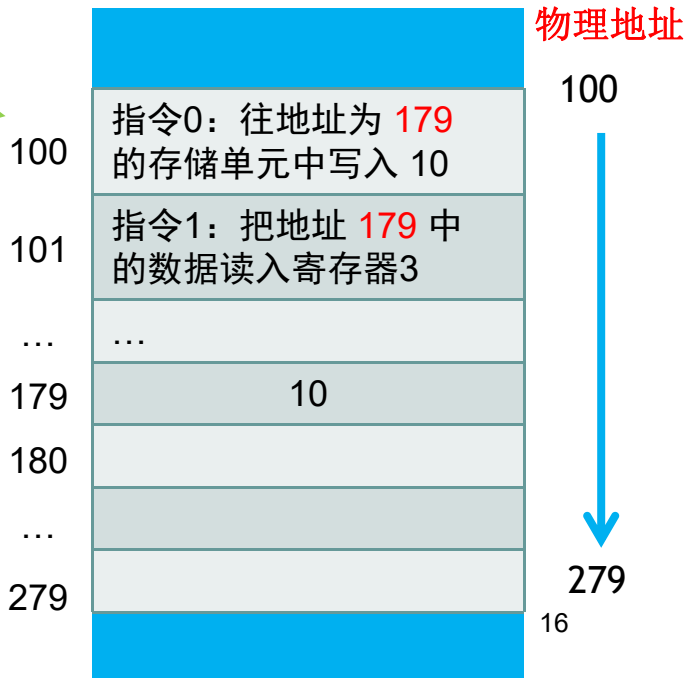
逻辑地址

1	指令0：往地址为 79 的存储单元中写入 10
2	指令1：把地址 79 中的 数据读入寄存器3
...	.....
179	.....

装入模块（可执行文件）

装入的起始物理地址为100，则所有地址相关的参数都 +100

装入



**可重定位装入**的特点是在一个作业装入内存时，必须**分配其要求的全部内存空间**，如果没有足够的内存，就不能装入该作业。作业一旦进入内存后，在**运行期间就不能再移动**，也不能再申请内存空间





# 装入的三种方式——动态运行时装入

**动态重定位**：又称**动态运行时装入**。编译、链接后的装入模块的地址都是从**0**开始的。装入程序把装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要执行时才进行。因此装入内存后所有的地址**依然是逻辑地址**。这种方式需要一个**重定位寄存器**的支持。

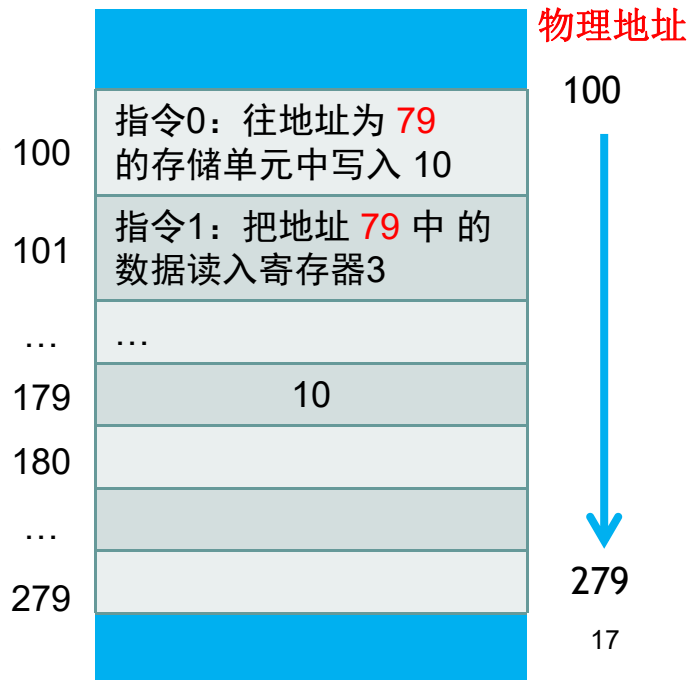
逻辑地址

1	指令0：往地址为 79 的存储单元中写入 10
2	指令1：把地址 79 中的 数据读入寄存器3
...	.....
179	.....

装入模块（可执行文件）

装入时依然保持使用**逻辑地址**

装入





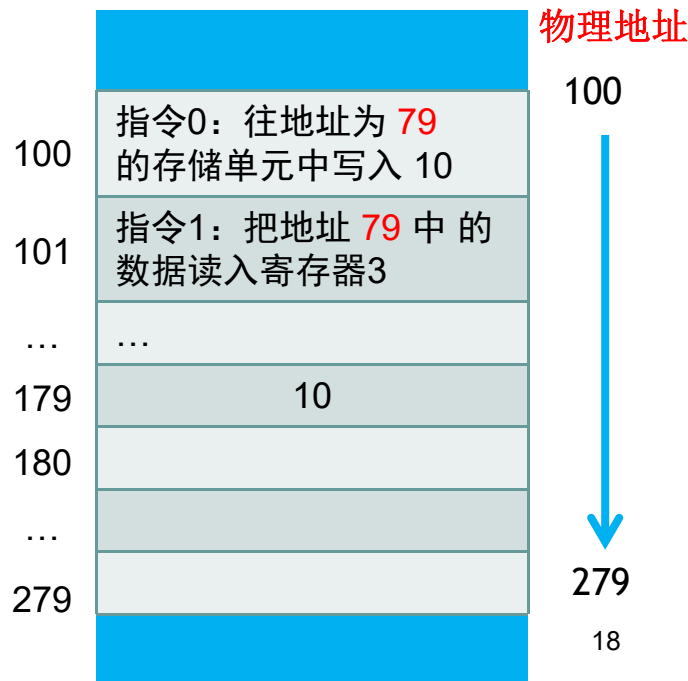
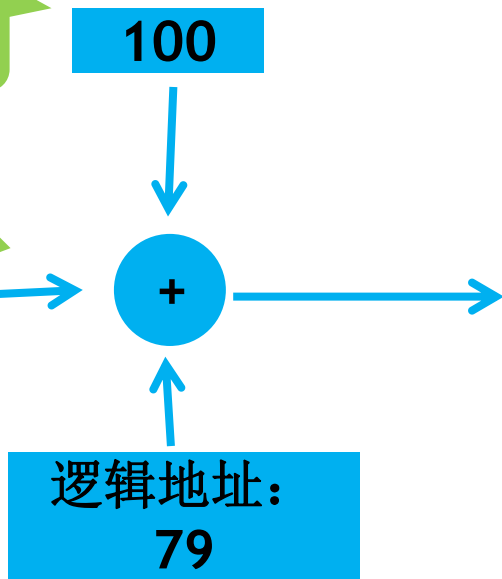
# 装入的三种方式——动态运行时装入

**动态重定位：**又称**动态运行时装入**。编译、链接后的装入模块的地址都是从**0**开始的。装入程序把装入模块装入内存后，并不会立即把逻辑地址转换为物理地址，而是把地址转换推迟到程序真正要执行时才进行。因此装入内存后所有的地址**依然是逻辑地址**。这种方式需要一个**重定位寄存器**的支持。

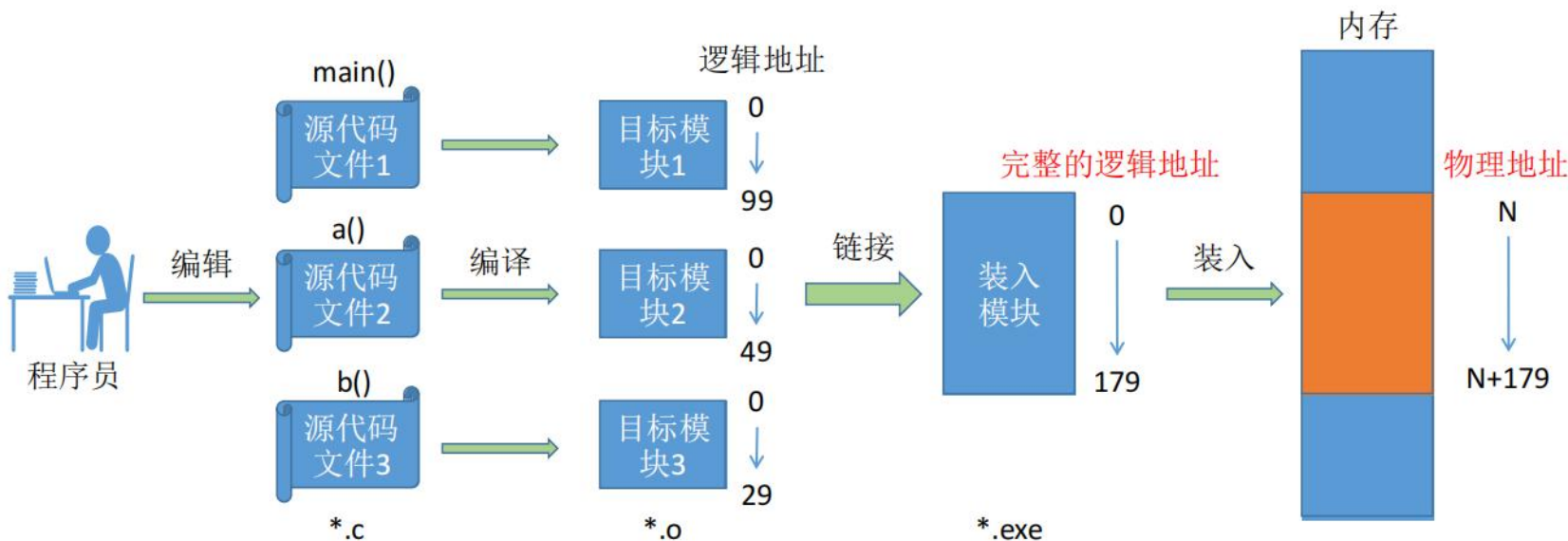
**重定位寄存器：**  
存放装入模块存放的起始地址

在程序运行前只需装入它的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，可以向用户提供一个比存储空间大得多的地址空间。

采用动态重定位时允许程序在内存中发生移动。



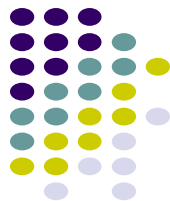
# 从写程序到程序运行



**编译:** 由编译程序将用户源代码编译成若干个目标模块（**编译就是把高级语言翻译为机器语言**）

**链接:** 由链接程序将编译后形成的一组目标模块，以及所需库函数链接在一起，形成一个完整的装入模块

**装入（装载）:** 由装入程序将装入模块装入内存运行

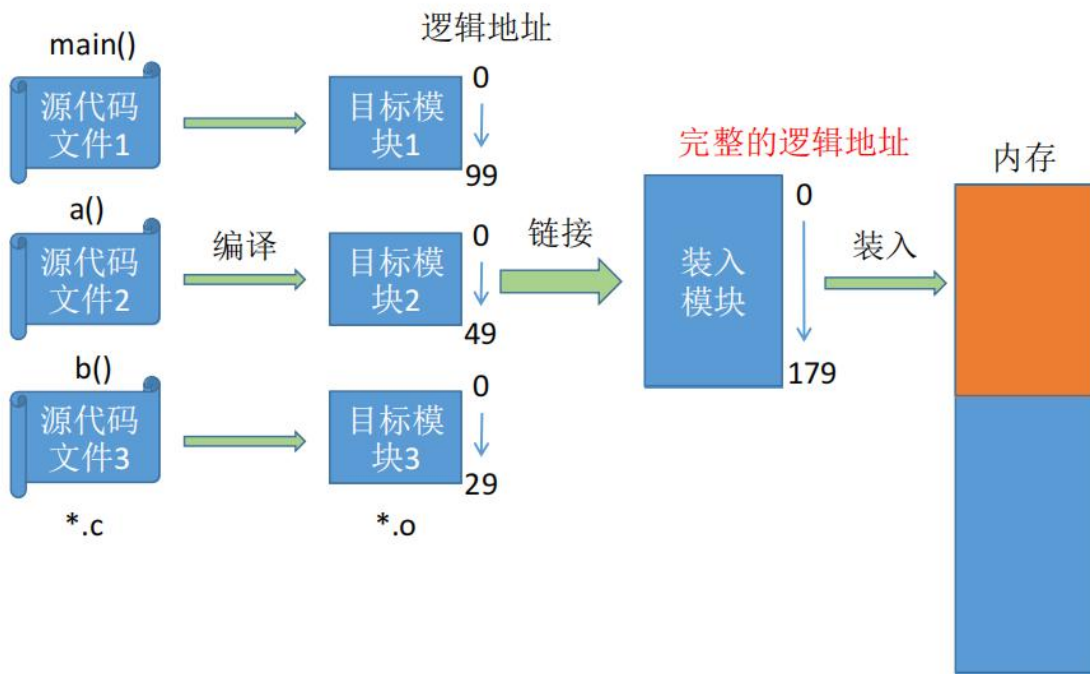


# 链接的三种方式

- 程序经过编译后得到一组目标模块，再利用链接程序将目标模块链接，形成装入模块。根据链接时间的不同，把链接分成三种：
  - **静态链接**：在程序运行前，将目标模块及所需的库函数链接成一个完整的装配模块，以后不再拆开。
  - **装入时动态链接**：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。
  - **运行时动态链接**：指对某些目标模块的链接，是在程序执行中需要该目标模块时，才对它进行链接



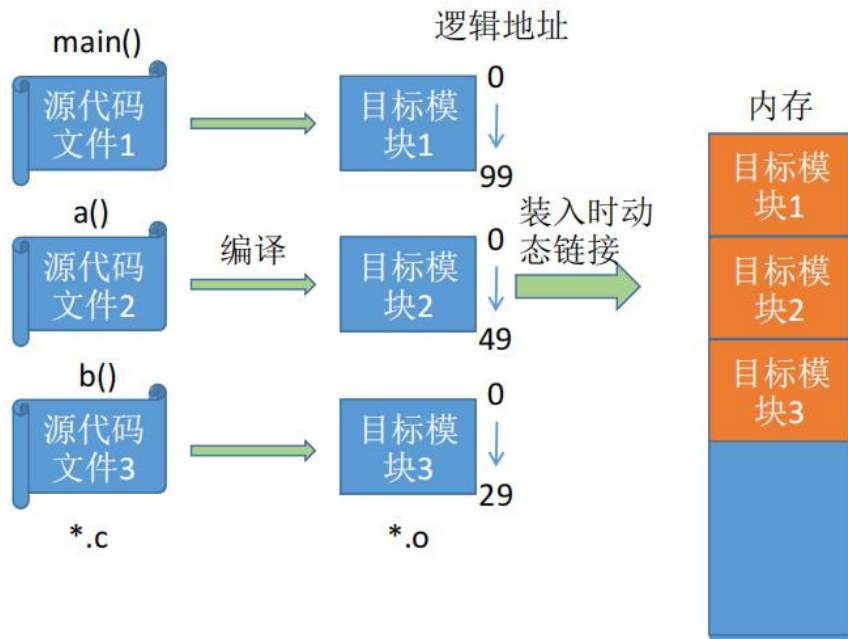
# 链接的三种方式-静态链接



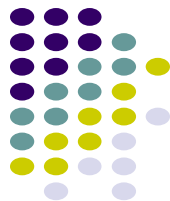
**静态链接：**在程序运行之前，先将各目标模块及它们所需的库函数连接成一个完整的可执行文件（装入模块），之后不再拆开。



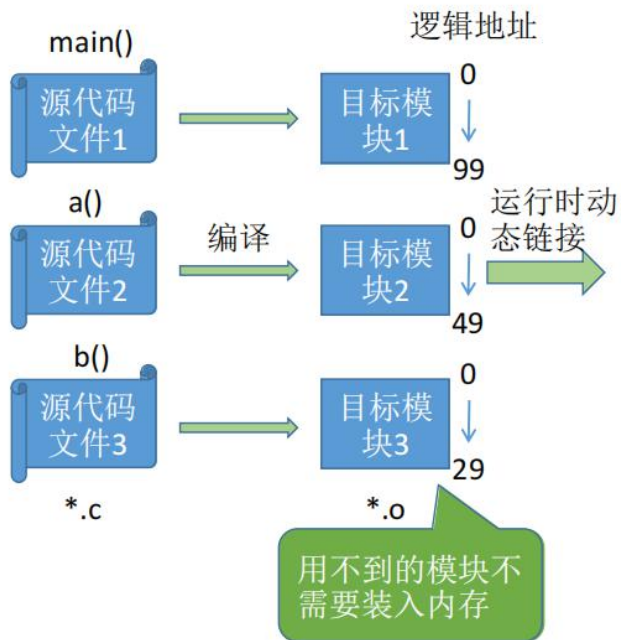
# 链接的三种方式-装入时动态链接



**装入时动态链接：**将各目标模块装入内存时，边装入边链接的链接方式

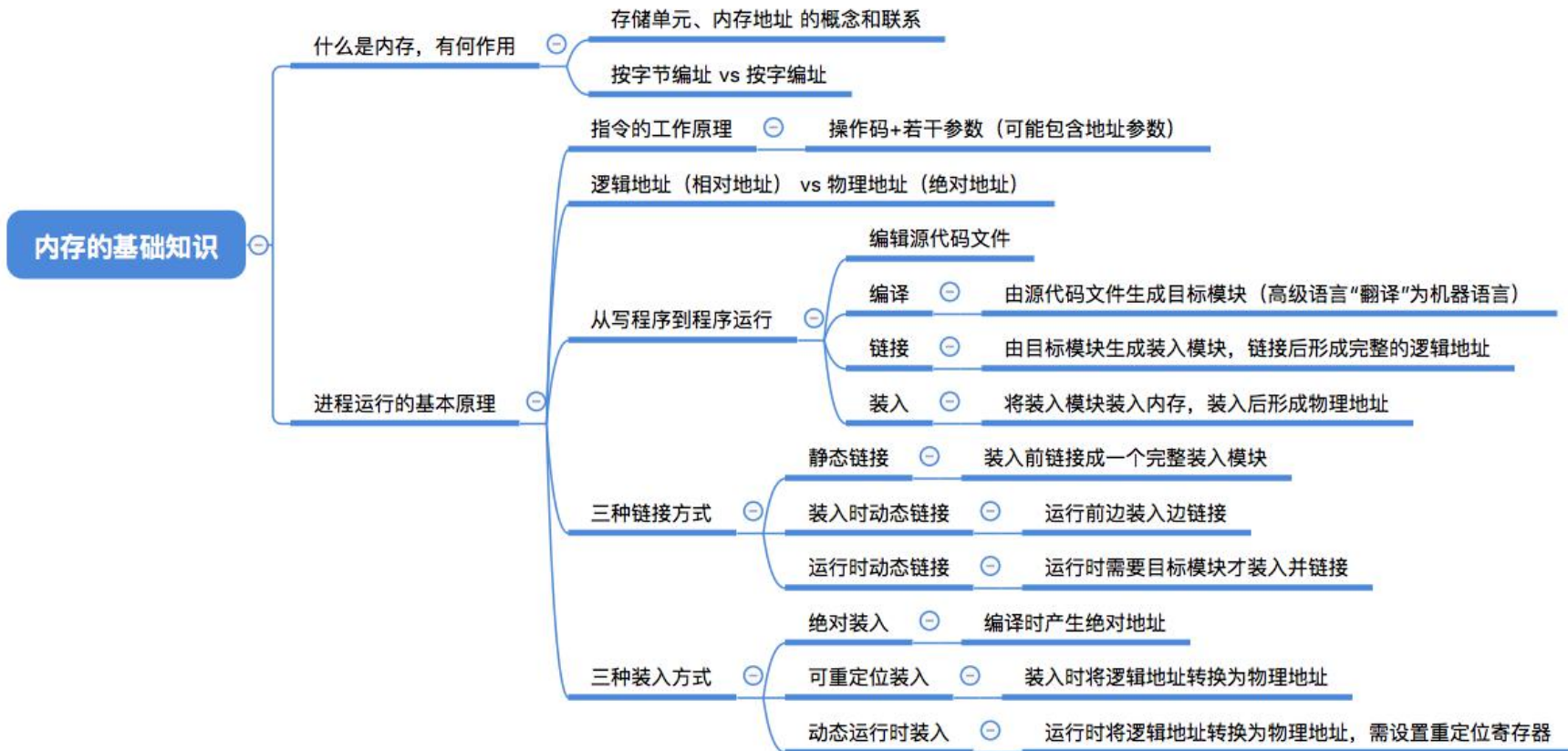


# 链接的三种方式-运行时动态链接

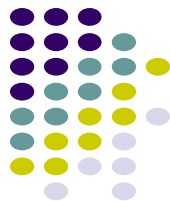


**运行时动态链接：**在程序执行中需要该目标模块时，才对它进行链接。其优点是便于修改和更新，便于实现对目标模块的共享

# 知识点回顾







# 内容

- 4.1 程序的装入和链接
- 4.2 连续分配方式
- 4.3 基本分页存储管理方式
- 4.4 基本分段存储管理方式
- 4.5 虚拟存储器的基本概念
- 4.6 请求分页存储管理方式
- 4.7 页面置换算法
- 4.8 请求分段存储管理方式

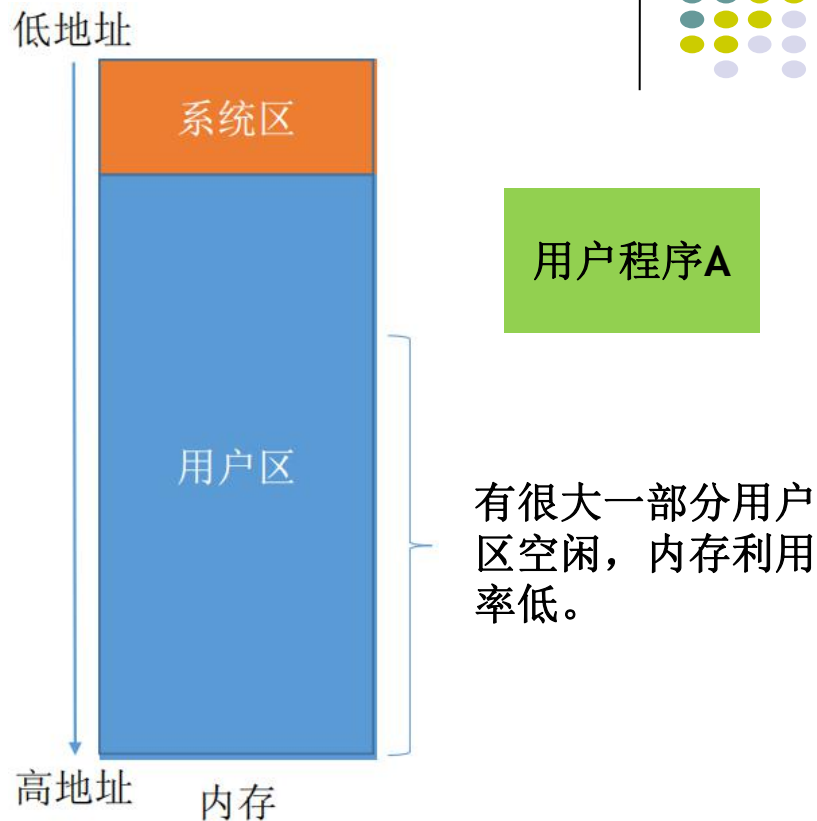


# 单一连续分配

在单一连续分配方式中，内存被分为**系统区**和**用户区**。系统区通常位于内存的低地址部分，用于存放操作系统相关数据；用户区用于存放用户进程相关数据。内存中只能有**一道用户程序**，用户程序独占整个用户区空间。

优点：**实现简单**；无外部碎片；可以采用覆盖技术扩充内存；不一定需要采取内存保护（**eg**：早期的 **PC** 操作系统 **MS-DOS**）。

缺点：只能用于**单用户、单任务**的操作系统中；有内部碎片；存储器利用率极低。



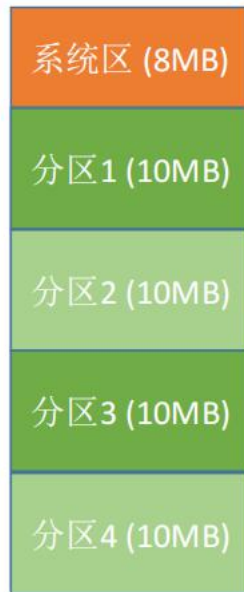


# 固定分区分配

将内存用户空间划分为若干个**固定大小**的区域，在每个分区中只装入**一道作业**，便可以有多道作业并发执行。当有一空闲分区时，便可以再从外存的后备作业队列中，选择一个适当大小的作业装入该分区，当该作业结束时，可再从后备作业队列中找出另一作业调入该分区。

**分区大小相等**：缺乏灵活性，但是很适合用于用一台计算机控制多个相同对象的场合（比如：**钢铁厂有n个相同的炼钢炉**，就可把内存分为**n个大小相等的区域**存放**n个炼钢炉控制程序**）

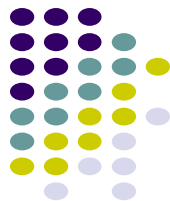
**分区大小不等**：增加了灵活性，可以满足不同大小的进程需求。根据常在系统中运行的作业大小情况进行划分（比如：**划分多个小分区、适量中等分区、少量大分区**）



内存（分区大小**相等**）



内存（分区大小**不等**）



# 固定分区分配

操作系统需要建立一个数据结构——**分区说明表**，来实现各个分区的分配与回收。每个表项对应一个分区，通常按**分区大小排列**。每个表项包括对应分区的大小、起始地址、状态（是否已分配）。

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	未分配

分区说明表

20K 32K 64K 128K 256K

操作系统
作业A
作业B
作业C
~

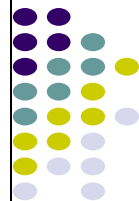
存储空间分配情况

当某用户程序要装入内存时，由操作系统内核程序根据用户程序大小检索该表，从中找到一个能**满足大小**的、**未分配**的分区，将之分配给该程序，然后修改状态为“**已分配**”。

优点：**实现简单**。

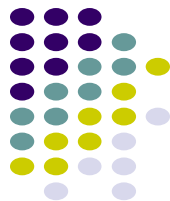
缺点：**a.** 当用户程序太大时，可能所有的分区都不能满足需求；  
**b.** 会产生**内部碎片**，**内存利用率低**。

作业1进入，大小30K      作业2进入，大小500K  
作业3进入，大小8K



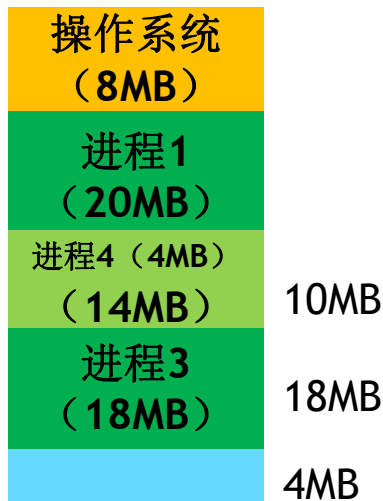
分区号	大小	起址	状态
1	8 K	512K	已使用
2	32 K	520K	已使用
3	32 K	552K	未使用
4	128 K	584K	未使用
5	512 K	712K	已使用
...	...	...	...





# 动态分区分配

**动态分区分配**又称为**可变分区分配**。这种分配方式**不会预先划分内存分区**，而是在进程装入内存时，**根据进程的大小动态地建立分区**，并使分区的大小正好适合进程的需要。因此系统分区的大小和数目是可变的。（eg: 假设某计算机内存大小为 **64MB**，系统区 **8MB**，用户区共 **56 MB**...）



1. 系统要用什么样的数据结构记录内存的使用情况？

2. 当很多个空闲分区都能满足需求时，应该选择哪个分区进行分配？

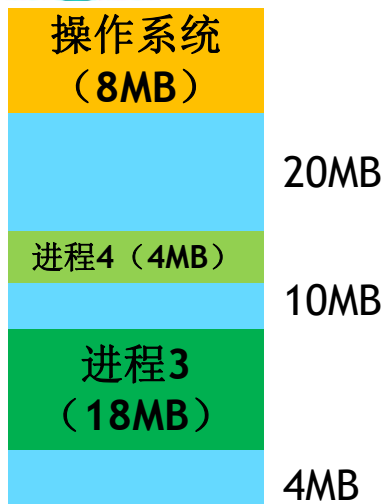
3. 如何进行分区的分配与回收操作？



# 动态分区分配



1. 系统要用什么样的数据结构记录内存的使用情况？



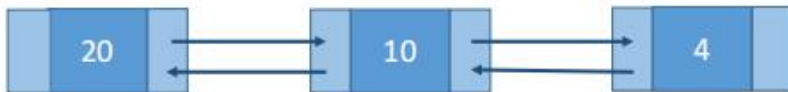
## 两种常用的数据结构

空闲分区表

空闲分区链

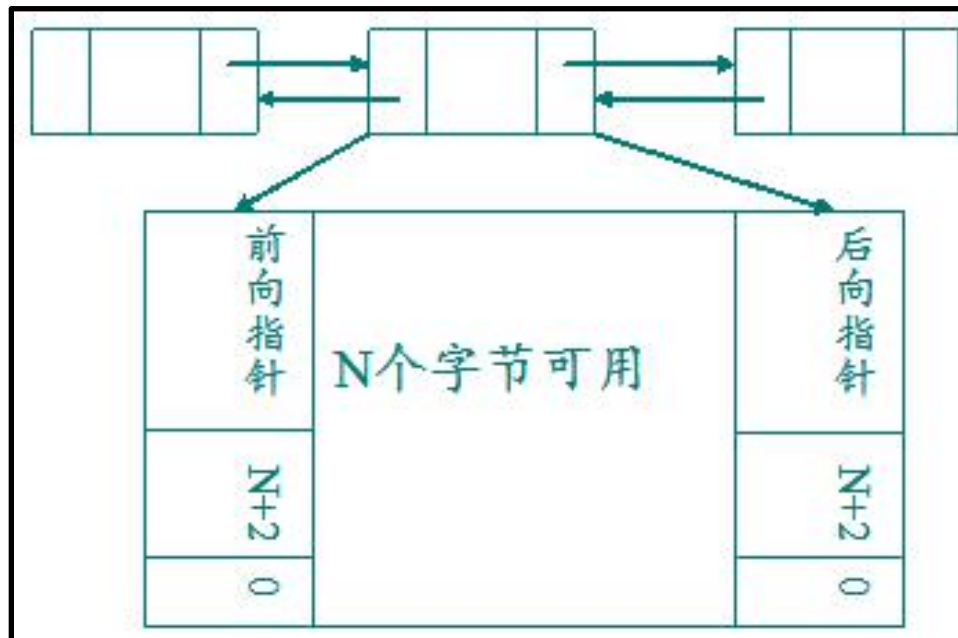
分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

空闲分区表：每个空闲分区对应一个表项。表项中包含分区号、分区大小、分区起始地址等信息

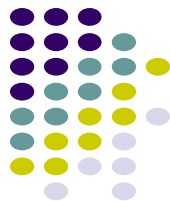


空闲分区链：每个分区的起始部分和末尾部分分别设置**前向指针**和**后向指针**。起始部分处还可记录分区大小等信息

# 动态分区分配



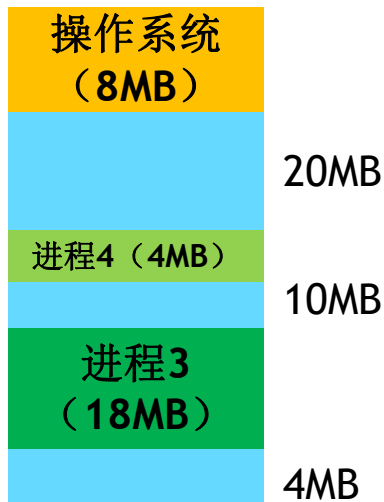




# 动态分区分配



2. 当很多个空闲分区都能满足需求时，应该选择哪个分区进行分配？



进程5 (4MB)

应该用最大的分区进行分配？还是用最小的分区进行分配？又或是用地址最低的部分进行分配？

把一个新作业装入内存时，须按照一定的**动态分区分配算法**，从空闲分区表（或空闲分区链）中选出一个分区分配给该作业。由于分配算法对系统性能有很大的影响，因此人们对它进行了广泛的研究。

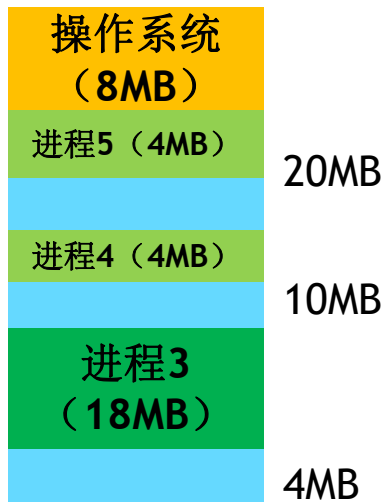


# 动态分区分配



3. 如何进行分区的分配与回收操作？ 假设系统采用的数据结构是“空闲分区表” ...

如何分配？



分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

分区号	分区大小 (MB)	起始地址 (M)	状态
1	16	12	空闲
2	10	32	空闲
3	4	60	空闲

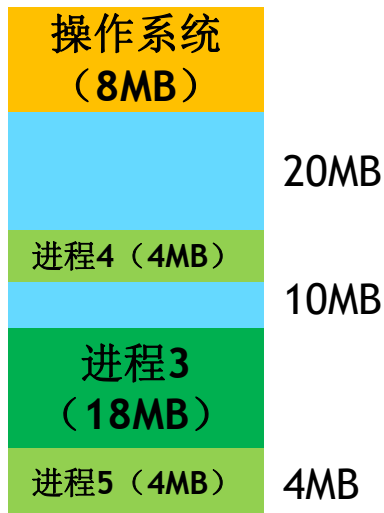


# 动态分区分配



3. 如何进行分区的分配与回收操作？ 假设系统采用的数据结构是“空闲分区表” ...

如何分配？



分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

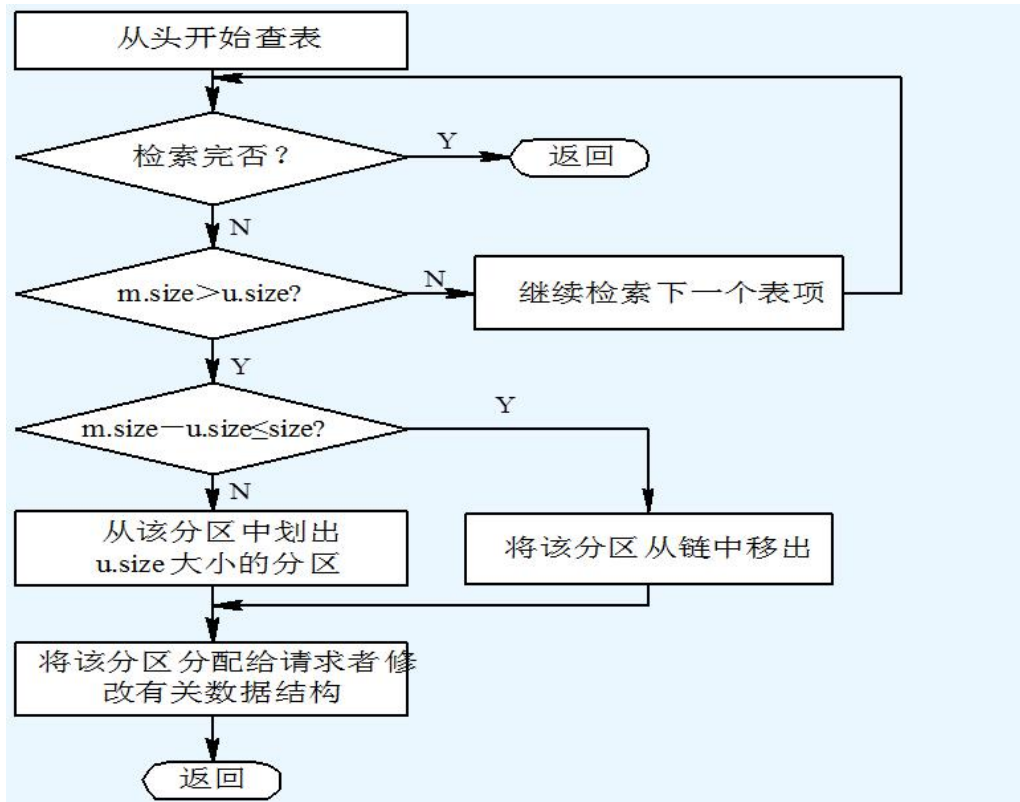
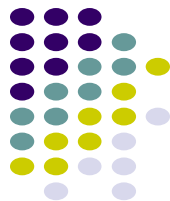
分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	10	32	空闲

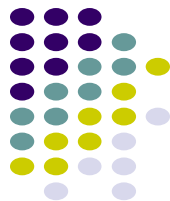


# 动态分区分配

- 分配内存

- 利用某种分配算法，从空闲分区链(表)中找到所需大小的分区。设请求的分区大小为 $u.size$ ，表中每个空闲分区的大小表示为 $m.size$ ，若 $m.size - u.size \leq size$ (规定的不再切割的分区大小)，将整个分区分配给请求者，否则从分区中按请求的大小划出一块内存空间分配出去，余下部分留在空闲链中，将分配区首址返回给调用者。





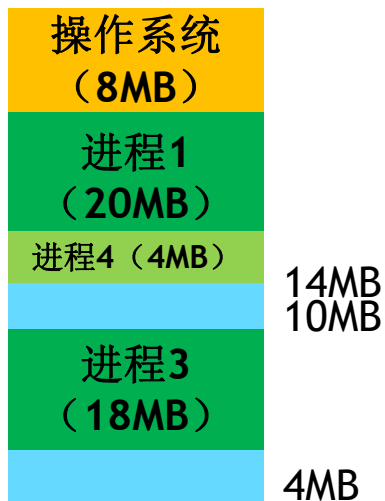
# 动态分区分配



3. 如何进行分区的分配与回收操作？ 假设系统采用的数据结构是“空闲分区表” ...

如何回收？

情况一：回收区的后面有一个相邻的空闲分区



分区号	分区大小 (MB)	起始地址 (M)	状态
1	10	32	空闲
2	4	60	空闲

分区号	分区大小 (MB)	起始地址 (M)	状态
1	14	28	空闲
2	4	60	空闲

两个相邻的空闲分区合并为一个



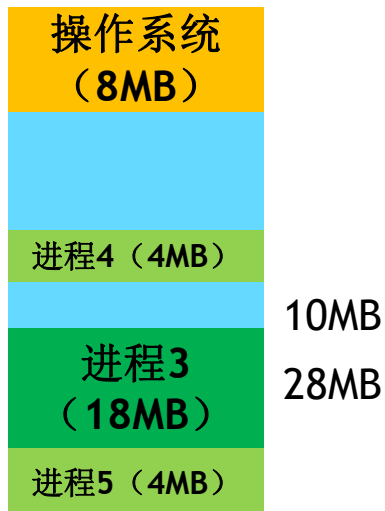
# 动态分区分配



3. 如何进行分区的分配与回收操作？ 假设系统采用的数据结构是“空闲分区表” ...

如何回收？

情况二：回收区的前面有一个相邻的空闲分区



分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	10	32	空闲

分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	28	32	空闲

两个相邻的空闲分区合并为一个



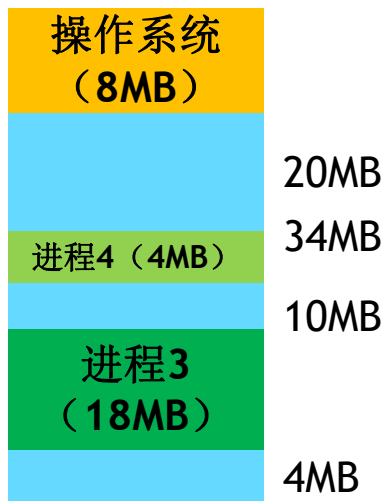
# 动态分区分配



3. 如何进行分区的分配与回收操作？ 假设系统采用的数据结构是“空闲分区表” ...

如何回收？

情况三：回收区的前、后各有一个相邻的空闲分区



分区号	分区大小 (MB)	起始地址 (M)	状态
1	20	8	空闲
2	10	32	空闲
3	4	60	空闲

分区号	分区大小 (MB)	起始地址 (M)	状态
1	34	8	空闲
2	4	60	空闲

三个相邻的空闲分区合并为一个





# 动态分区分配



3. 如何进行分区的分配与回收操作？ 假设系统采用的数据结构是“空闲分区表” ...

如何回收？

情况四：回收区的前、后都没有相邻的空闲分区

操作系统 (8MB)	14MB
进程1 (20MB)	
进程2 (14MB)	
进程3 (18MB)	
4MB	

分区号	分区大小 (MB)	起始地址 (M)	状态
1	4	60	空闲

分区号	分区大小 (MB)	起始地址 (M)	状态
1	14	28	空闲
2	4	60	空闲

新增一个表项

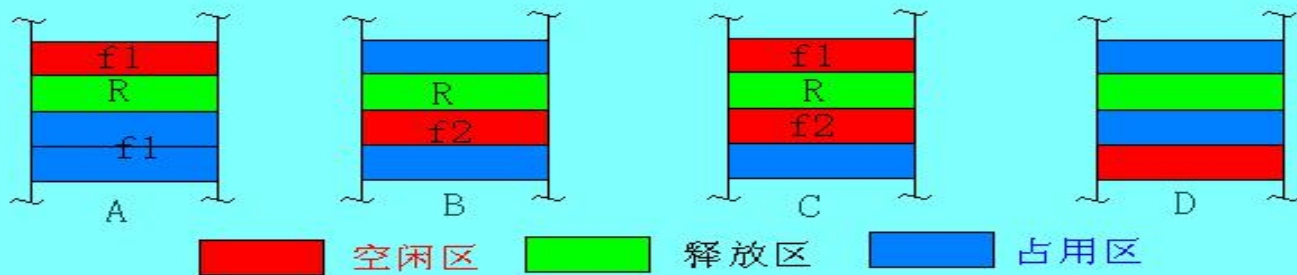
注：各表项的顺序不一定按照地址递增顺序排列，具体的排列方式需要依据动态分区分配算法来确定。



# 动态分区分配

## ● 回收内存

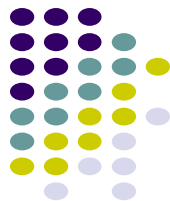
- 当进程运行完毕释放内存时，系统根据回收区首址，在空闲分区链(表)中找到相应插入点，可能有四种情况：
- (1) 回收区与插入点的前一个分区F1邻接
- (2) 回收区与插入点的后一个分区F2邻接
- (3) 回收区与插入点的前后两个分区F1、F2邻接
- (4) 回收区既不与F1邻接，又不与F2邻接





# 可重定位分区分配

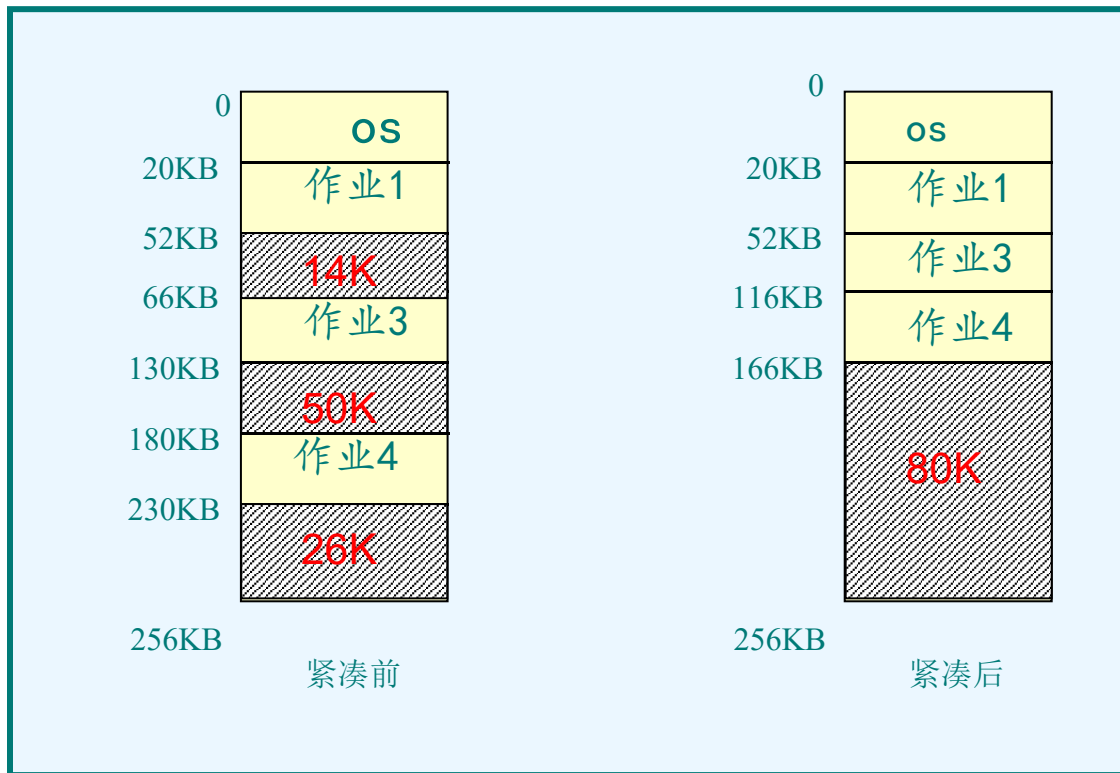
- 动态重定位的引入
  - 在连续分配方式中，必须把系统或用户程序装入一连续的内存空间。  
如果在系统中只有若干个小分区，即使它们的容量总和大于要装入的程序，但由于这些分区不相邻，也无法将程序装入内存
- 动态分区分配没有内部碎片，但是有外部碎片。
  - 内部碎片，分配给某进程的内存区域中，如果有些部分没有用上。
  - 外部碎片，是指内存中的某些空闲分区由于太小而难以利用。

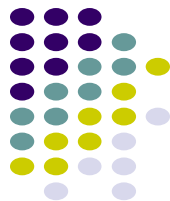


## 可重定位分区分配

- 解决方法：将内存中的所有作业进行移动，使它们全部邻接，这样把原来分散的小分区拼接成大分区，这种方法称为“拼接”或“紧凑”。
- 缺点：用户程序在内存中的地址发生变化，必须重定位。

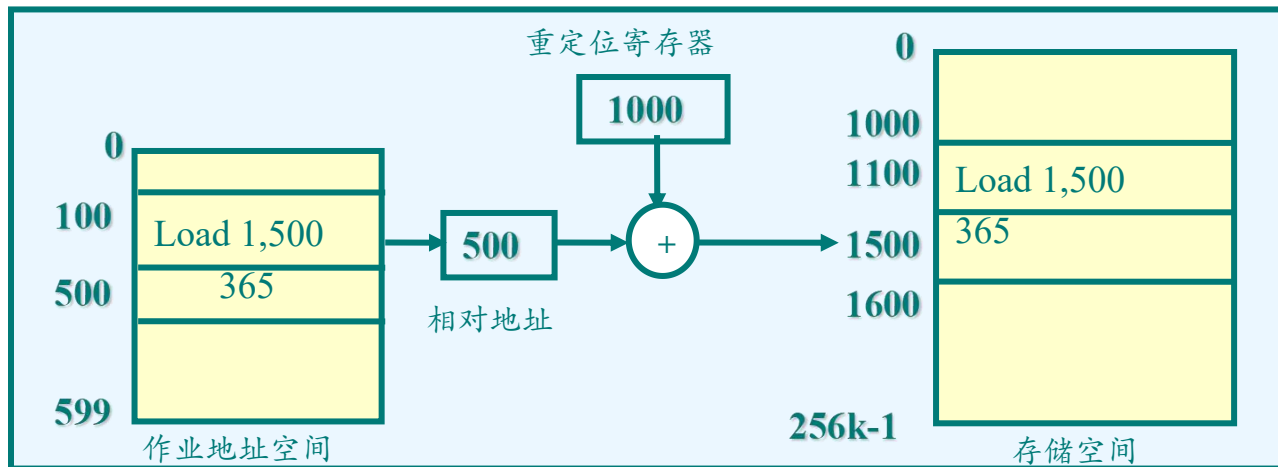
# 可重定位分区分配

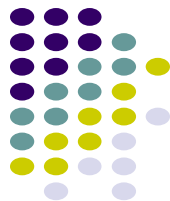




# 动态重定位的实现

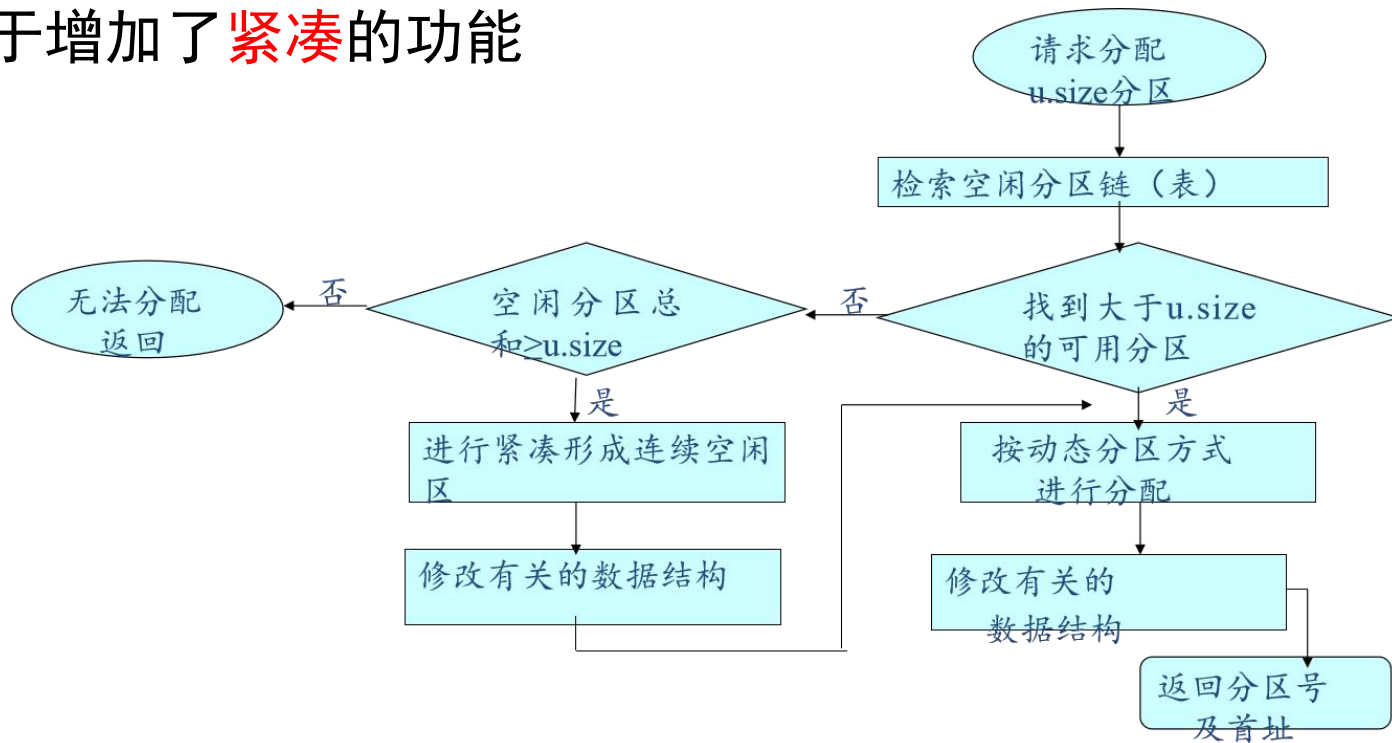
- 在动态运行时装入的方式，将相对地址转换为物理地址的工作在程序指令真正要执行时才进行。地址转换需要重定位寄存器的支持。程序执行时访问的内存地址是**相对地址**与**重定位寄存器**中的地址相加而成





# 动态重定位分区分配算法

- 动态重定位分区分配算法与动态分区分配算法基本相同，差别在于增加了**紧凑**的功能



# 知识回顾



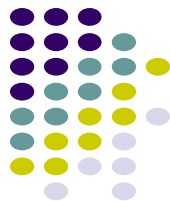
## 连续分配管理



为用户进程分配的必须是一个连续的内存空间

另外，需要对用于管理空闲分区的数据结构有印象——空闲分区表、空闲分区链





# 动态分区分配算法

算法思想：每次都从低地址开始查找，找到第一个能满足大小的空闲分区

## ● 首次适应算法FF

- FF算法要求空闲分区表以**地址递增**的次序排列。在分配内存时，从表首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止；然后按照作业的大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲分区表中。若从头到尾不存在满足要求的分区，则分配失败。
- 优点：优先利用内存低址部分的内存空间,保留了高址部分的大空闲区。
- 缺点：低址部分不断划分，产生**小碎片**（内存碎块、零头）；每次查找从低址部分开始，增加了查找的开销



# 动态分区分配算法

算法思想：避免地址部分留下很多小的空闲分区，减少查找可用空闲分区的开销。

- 循环首次适应算法

- 在分配内存空间时，从上次找到的空闲分区的下一个空闲分区开始查找，直到找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。
- 为实现算法，需要：
  - 设置一起始查寻指针
  - 采用循环查找方式
- 优点：使内存空闲分区分布均匀，减少查找的开销
- 缺点：缺乏大的空闲分区



# 动态分区分配算法

算法思想：由于动态分区分配是一种连续分配方式，为各进程分配的空间必须是连续的一整片区域。因此为了保证当“大进程”到来时能有连续的大片空间，可以尽可能多地留下大片的空闲区，即，优先使用更小的空闲区。

- 最佳适应算法

- 所谓“最佳”是指每次为作业分配内存时，总是把能满足要求、又是最小的空闲分区分配给作业，避免“大材小用”。
- 要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。
- 缺点：产生许多难以利用的小空闲区



# 动态分区分配算法

算法思想：为了解决最佳适应算法的问题——即留下太多难以利用的小碎片，可以在每次分配时 优先使用最大的连续空闲区，这样分配后剩余的空闲区就不会太小，更方便使用

- 最坏适应算法(worst fit)
  - 最坏适应分配算法要扫描整个空闲分区表或链表，总是挑选一个**最大的空闲区**分割给作业使用。该算法要求将所有的空闲分区按其**容量从大到小**的顺序形成一空闲分区链，查找时只要看第一个分区能否满足作业要求。
  - 优点：剩下的空闲区还可以利用，同时查找效率很高。
  - 缺点：缺乏大的空闲分区。



# 动态分区分配算法

算法思想：基于顺序搜索的动态分区分配算法，不适用于较大的系统。为了提高搜索空闲分区的速度，在大型、中型系统中，往往采用基于索引搜索的动态分配算法

- 快速适应算法(quick fit)
  - 该算法又称为分类搜索法，是将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表，系统中存在多个空闲分区链表，同时在内存中设立一张管理索引表，该表的每一个表项对应了一种空闲分区类型，并记录了该类型空闲分区链表表头的指针



# 动态分区分配算法

- 快速适应算法(quick fit)

- 优点是**查找效率高**，仅需要根据进程的长度，寻找到能容纳它的最小空闲区链表，并取下第一块进行分配即可。在进行空闲分区分配时，不会对任何分区产生分割，所以能保留大的分区，满足对大空间的需求，也不会产生内存碎片。

- 缺点：

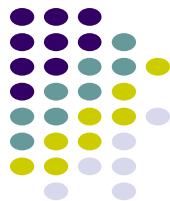
在分区归还主存时算法复杂，系统开销较大。

一个进程占用一个分区，存在浪费现象

# 算法对比



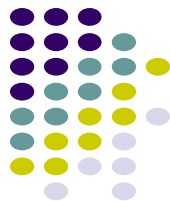
算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	空闲分区以地址递增次序排列	算法开销小，保留大容量分区	产生很多小的空闲分区（碎片）
循环首次适应	从上次找到的空闲分区的下一个空闲分区开始查找	同上	空闲分区分布均匀，减少查找开销	缺乏大的空闲分区
最佳适应	优先使用更小的分区，以保留更多大分区	空闲分区以容量递增次序排列	会有更多的大分区被保留下来，更能满足大进程需求	会产生很多小的碎片
最坏适应	优先使用更大的分区，以防止产生太小的不可用的碎片	空闲分区以容量递减次序排列	可以减少难以利用的小碎片，提高查找效率	缺乏大的空闲分区
快速适应	基于分类搜索算法	根据分区容量分类	查找效率高，保留大分区，不会产生内存碎片	算法复杂，系统开销较大；浪费空间



# 对换

- 对换的引入
  - 阻塞进程占据大量内存空间
  - 许多作业在外存而不能进入内存运行
- 对换：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**的进程和进程所需要的程序和数据调入内存
- 对换的类型
  - **整体对换**：以进程为单位的对换
  - **部分对换**：以“页”或“段”为单位的对换
- 实现进程对换，系统必须具备的功能：**对换空间的管理、进程的换出、进程的换入**





# 对换空间的管理

- 一般从磁盘上划出一块空间作为对换区使用

	存储内容	驻留时间	主要目标	分配方式
文件区	文件	较长久	提高文件存储空间 的利用率	离散
对换区	从内存换出的 进程	短暂	提高进程换入和 换出的速度	连续

外存的划分：文件区、对换区

管理方式：空闲分区表、空闲分区链

分配算法：首次适应法、循环首次适应法、最佳适应法

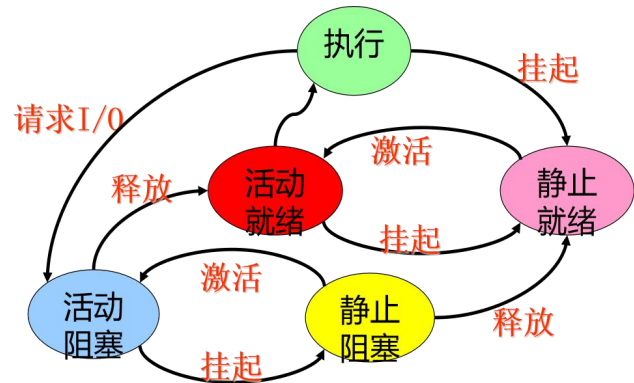
# 进程的换出与换入

- 进程的换出

- 选择处于阻塞或睡眠状态且优先级最低的进程
- 将该进程的程序和数据传送到磁盘的对换区上
- 回收内存空间，修改该进程的PCB

- 进程的换入

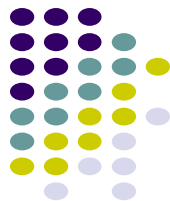
- 定时查看进程状态
- 将处于就绪态的，换出时间最久的进程换入内存





# 内容

- 4.1 程序的装入和链接
- 4.2 连续分配方式
- 4.3 基本分页存储管理方式
- 4.4 基本分段存储管理方式
- 4.5 虚拟存储器的基本概念
- 4.6 请求分页存储管理方式
- 4.7 页面置换算法
- 4.8 请求分段存储管理方式



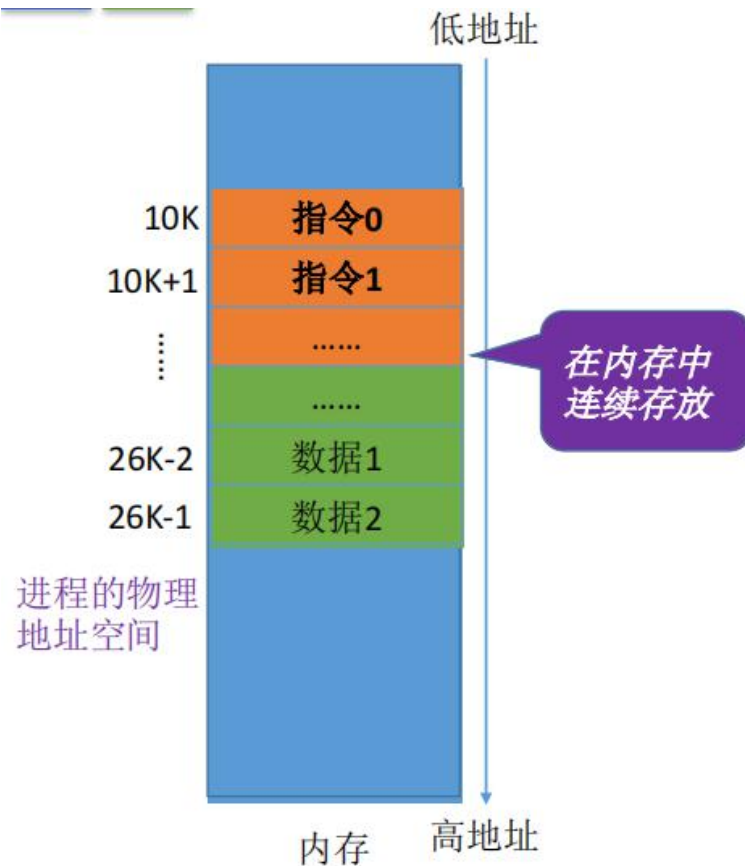
# 引入

- 连续分配方式会形成“碎片”，虽然可以通过“紧凑”解决，但开销大。如果允许将一个进程直接分散地装入许多不相邻的分区中，则无需“紧凑”，由此产生离散分配方式。
- 分类：
  - **分页存储管理方式**：离散分配的基本单位是页
  - **分段存储管理方式**：离散分配的基本单位是段
- 基本的分页存储管理方式(或纯分页存储管理方式)：不具备页面对换功能，不具有支持实现虚拟存储器的功能，要求把每个作业全部装入内存后方能运行。



# 什么是“地址空间”

- 概念回顾：
  - 逻辑地址（相对地址）
  - 物理地址（绝对地址）



# 基本分页存储管理方式

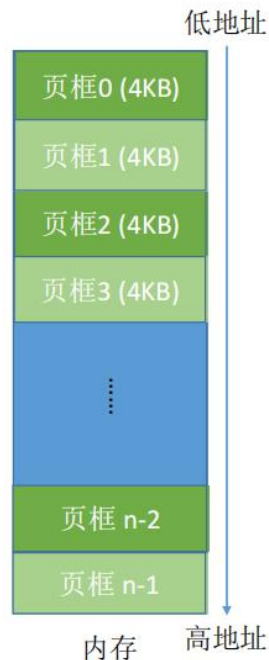


将内存空间分为一个个**大小相等**的分区（比如：每个分区**4KB**），每个分区就是一个“**页框**”（**页框=页帧=内存块=物理块=物理页面**）。每个页框有一个编号，即“**页框号/块号**”（**页框号=页帧号=内存块号=物理块号=物理页号**），页框号**从0开始**

将**进程的逻辑地址空间**也分为与页框大小相等的一个个部分，每个部分称为一个“**页**”或“**页面**”。每个页面也有一个编号，即“**页号**”，页号也是**从0开始**。

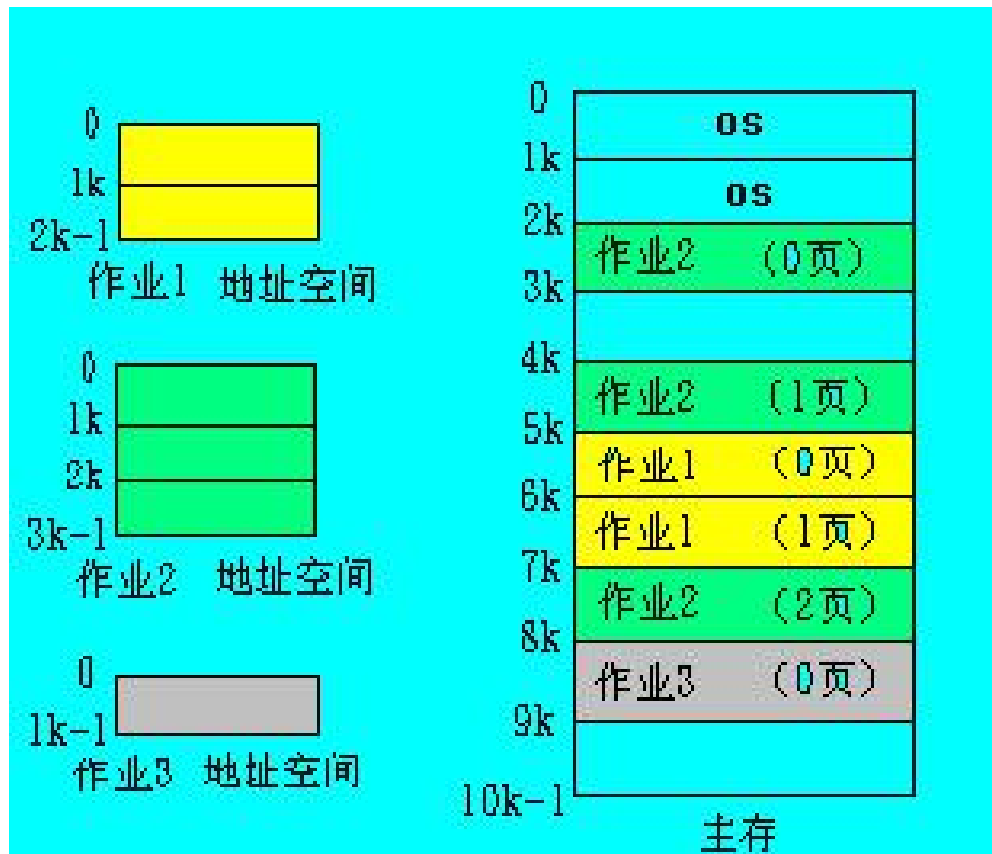
操作系统以**页框**为单位为各个进程分配内存空间。进程的每个**页面**分别放入一个页框中。也就是说，进程的页面与内存的页框有**一一对应**的关系。各个页面不必连续存放，可以放到不相邻的各个页框中

进程的最后一个页面可能没有一个页框那么大。也就是说，分页存储有可能产生**页内碎片**，因此页框不能太大，否则可能产生过大的页内碎片造成浪费





# 基本分页存储管理方式

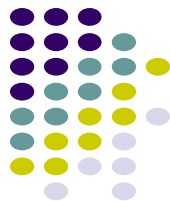




# 页面大小的选择

- 某一种机器只能采用一种大小的页面
  - 小：内碎片小，内存利用率高，但页面数目多，使页表过长，占大量内存，管理开销大
  - 大：页表短，管理开销小，内碎片大，内存利用率低
  - 页面大小应适中，是2的幂，通常是1KB~8KB





# 地址结构

- 逻辑地址



- 物理地址



地址结构包含两个部分：前一部分为页号，后一部分为页内偏移量  $W$ 。在上图所示的例子中，地址 长度为 32 位，其中 0~11 位 为“**位移量**”，或称“**页内地址**”；12~31 位为“**页号**”。

如果有  $K$  位表示“**位移量**”，则说明该系统中一个页面的大小是  $2^K$  个内存单元；  
如果有  $M$  位表示“**页号**”，则说明在该系统中，一个进程最多允许有  $2^M$  个页面

重要重要重要!!!

页面大小  $\leftrightarrow$  页内偏移量位数  $\rightarrow$  逻辑地址结构

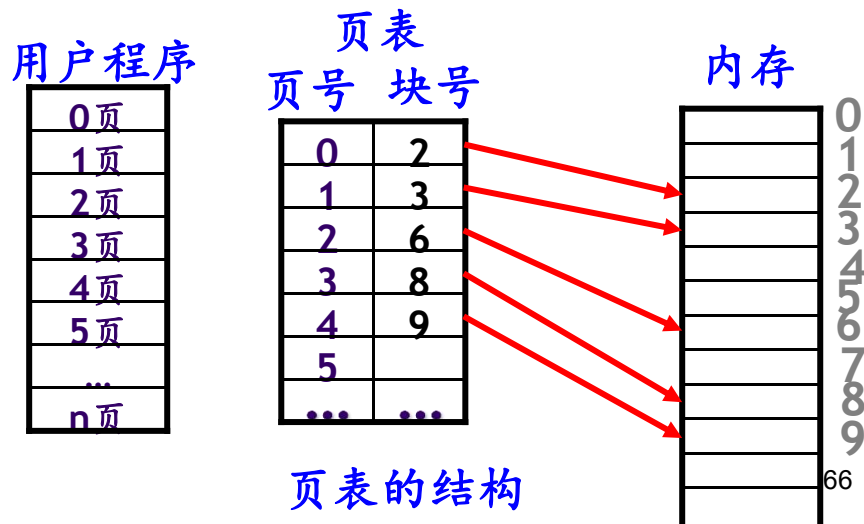


# 基本分页式存储管理的实现

## ● 页表

- 进程的每一页**离散地**存储在内存的任一存储块中，为方便查找，系统为每一进程建立一张页面映像表，简称**页表**
- **页表实现了从页号到物理块号的地址映射，页表通常存在PCB（进程控制块）中**

1. 一个进程对应一张页表
2. 进程的每个页面对应一个页表项
3. 每个页表项由“页号”和“块号”组成
4. 页表记录进程页面和实际存放的内存块之间的映射关系
5. 每个页表项的长度是相同的



## 用户程序

第0页
第1页
第2页
第3页
第4页
第5页
第6页

## 页表

页号	块号
0	2
1	7
2	3
3	5
4	9
5	10
6	11

## 内存

	0
	1
第0页	第0页
第2页	第2页
	4
第3页	第3页
	6
第1页	第1页
	8
第4页	第4页
第5页	第5页
第6页	第6页

0000,0000,0000,0000,0101,0001,0000,0000

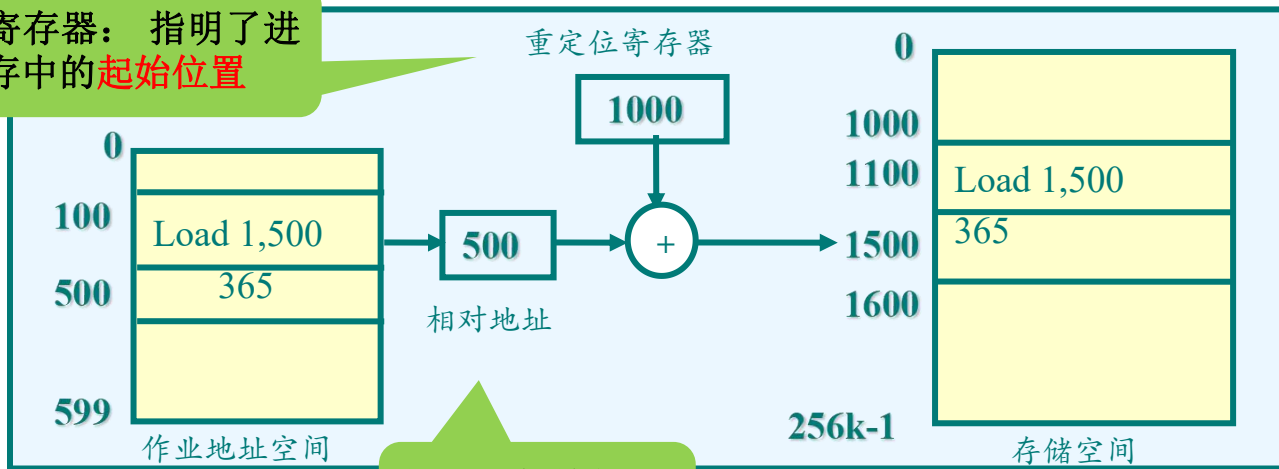
第3页256字节的物理地址（页大小：4K）

# 如何实现地址的转换



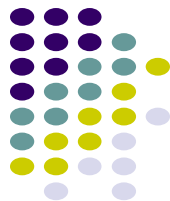
进程在内存中**连续存放**时，操作系统是如何实现逻辑地址到物理地址的转换的？

重定位寄存器：指明了进程在内存中的**起始位置**



相对于起始位置的**“偏移量”**

# 如何实现地址的转换

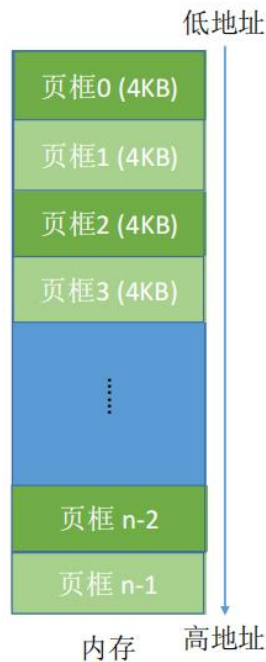


进程地址空间**分页**之后，操作系统该如何实现逻辑地址到物理地址的转换？

特点：虽然进程的各个页面是离散存放的，但是页面内部是连续存放的

如果要访问逻辑地址 **A**，则

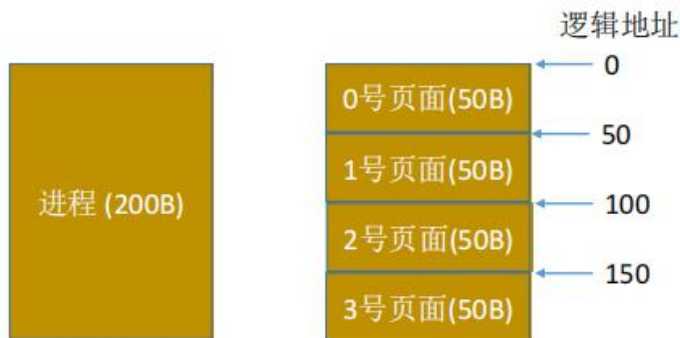
- ①确定逻辑地址**A** 对应的“页号” **P**
- ②找到**P**号页面在内存中的起始地址（需要查页表）
- ③确定逻辑地址**A** 的“页内偏移量” **W** 逻辑地址**A** 对应的物理地址 = **P**号页面在内存中的起始地址+页内偏移量**W**





# 如何确定页号、页内偏移量？

**Eg:** 在某计算机系统中，页面大小是**50B**。某进程逻辑地址空间大小为**200B**，则逻辑地址 **110** 对应的页号、页内偏移量是多少？



**页号** = 逻辑地址 / 页面长度 （取除法的整数部分）

**页内偏移量** = 逻辑地址 % 页面长度 （取除法的余数部分）

页号 =  $110 / 50 = 2$  ; 页内偏移量 =  $110 \% 50 = 10$

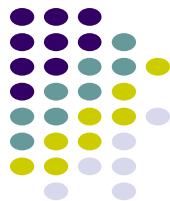
**逻辑地址**可以拆分为（**页号**，**页内偏移量**）通过页号查询页表，可知页面在内存中的起始地址

页面在内存中的起始地址+页内偏移量 = 实际的物理地址



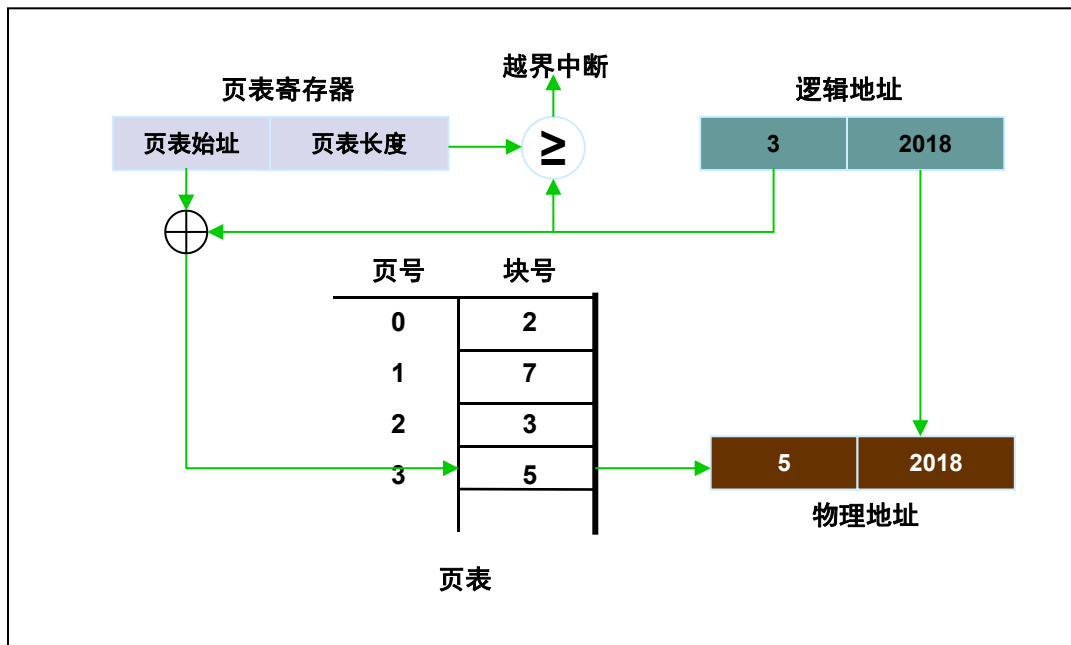
# 地址变换机构

- 为了能将用户地址空间的**逻辑地址**变换为内存空间的**物理地址**，在系统中必须设置地址变换机构
- 地址变换机构实现从逻辑地址到物理地址的转换，由于**页内地址与物理地址是一一对应的**，因此，地址变换机构的任务是借助于页表，将逻辑地址中的**页号**转换为内存中的**物理块号**
- **基本的地址变换机构**
  - 页表存放在内存系统区的一个连续空间中；
  - PCB和页表寄存器PTR中存有页表在内存的**首地址**和**页表长度**；



# 地址变化过程

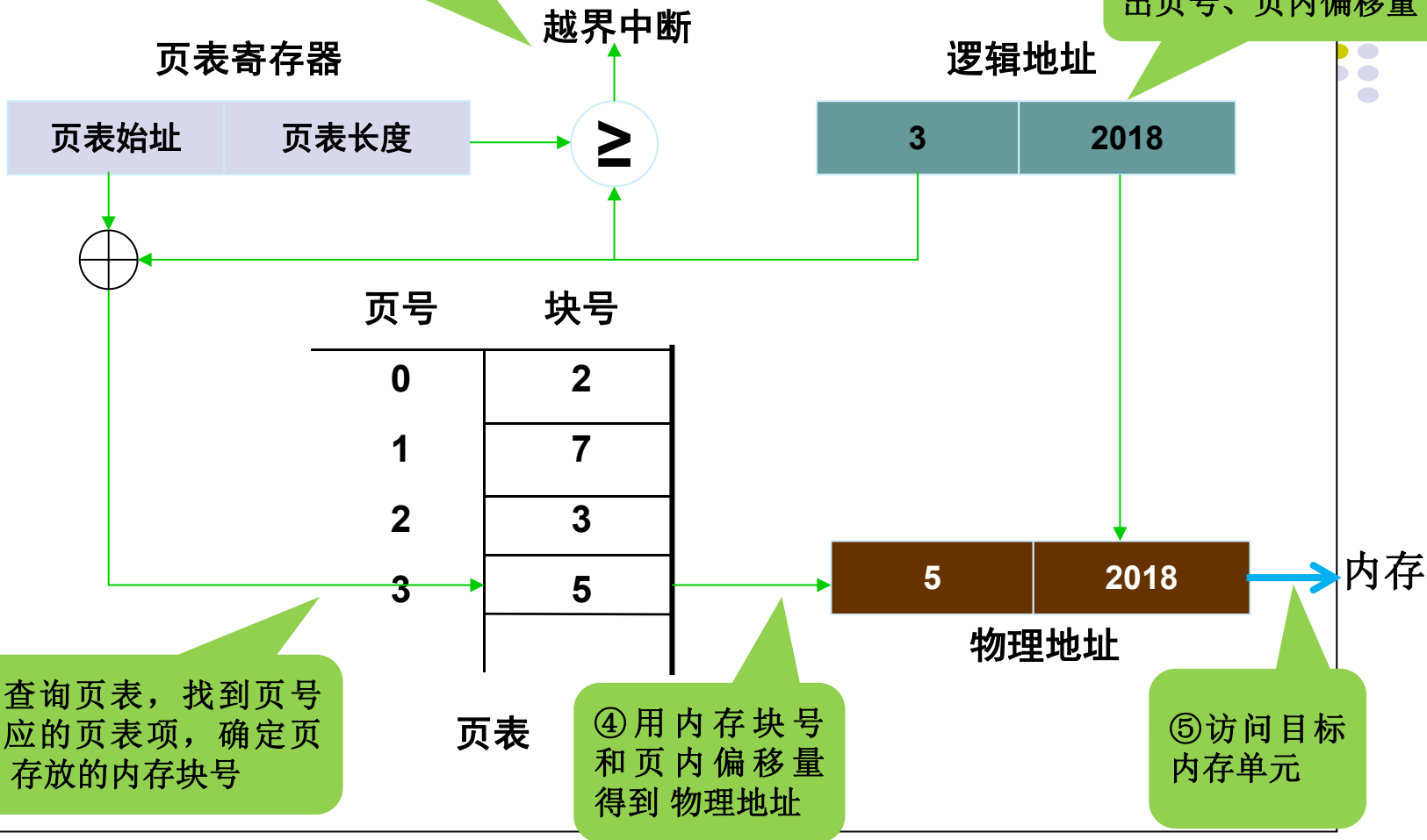
- 自动的将逻辑地址分为页号和页内地址
- 根据页号查询页表：页表首地址+页号\*表项长度





②判断页号是否越界

①根据逻辑地址计算出页号、页内偏移量





# 地址变化过程

设页面大小为 $L$ ，逻辑地址 $A$ 到物理地址 $E$ 的变换过程如下：

- ①计算页号 $P$ 和页内偏移量 $W$ （如果用十进制数手算，则  $P=A/L$ ， $W=A\%L$ ）
- ②比较页号 $P$ 和页表长度 $M$ ，若  $P \geq M$ ，则产生越界中断，否则继续执行。（注意：页号是从0开始的，而页表长度至少是1，因此  $P=M$  时也会越界）
- ③页表中页号 $P$ 对应的页表项地址 = 页表起始地址 $F$  + 页号 $P$  \* 页表项长度，取出该页表项内容 $b$ ，即为内存块号。（注意区分页表项长度、页表长度、页面大小的区别。页表长度指的是这个页表中总共有几个页表项，即总共有几个页；页表项长度指的是每个页表项占多大的存储空间； 页面大小指的是一个页面占多大的存储空间）
- ④计算  $E = b * L + W$ ，用得到的物理地址 $E$ 去访存。（如果内存块号、页面偏移量是用二进制表示的，那么把二者拼接起来就是最终的物理地址了）

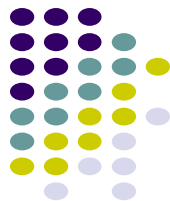


## 例题

若页面大小L 为 1K 字节，页号2对应的内存块号  $b = 8$ ，将逻辑地址  $A=2500$  转换为物理地址E。等价描述：某系统**按字节寻址**，逻辑地址结构中，**页内偏移量占10位**，页号2对应的内存块号  $b = 8$ ，将逻辑地址  $A=2500$  转换为物理地址E。

- ①计算页号、页内偏移量 页号  $P = A/L = 2500/1024 = 2$ ；  
页内偏移量  $W = A \% L = 2500 \% 1024 = 452$
- ②根据题中条件可知，页号2没有越界，其存放的内存块号  $b = 8$
- ③物理地址  $E = b * L + W = 8 * 1024 + 425 = 8644$

在分页存储管理（页式管理）的系统中，只要确定了每个页面的大小，逻辑地址结构就确定了。即，只要给出一个逻辑地址，系统就可以自动地算出页号、页内偏移量两个部分，并不需要显式地告诉系统这个逻辑地址中，页内偏移量占多少位。



# 具有快表的地址变换机构

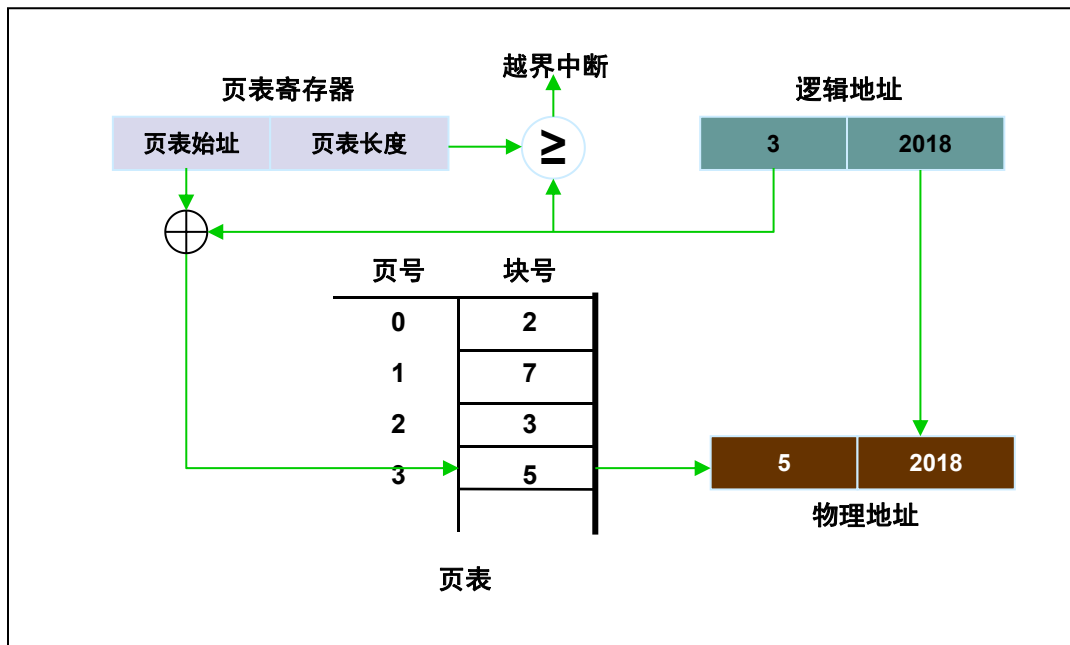


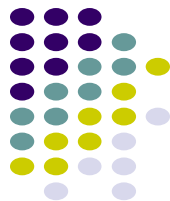
思考：CPU在每存取一个数据时，需要访问几次内存？

**第一次：**访问页表，找到指定页的物理块号，将块号与页内偏移量拼接形成物理地址。

**第二次：**从第一次所得地址中获得所需数据，或向此地址中写入数据

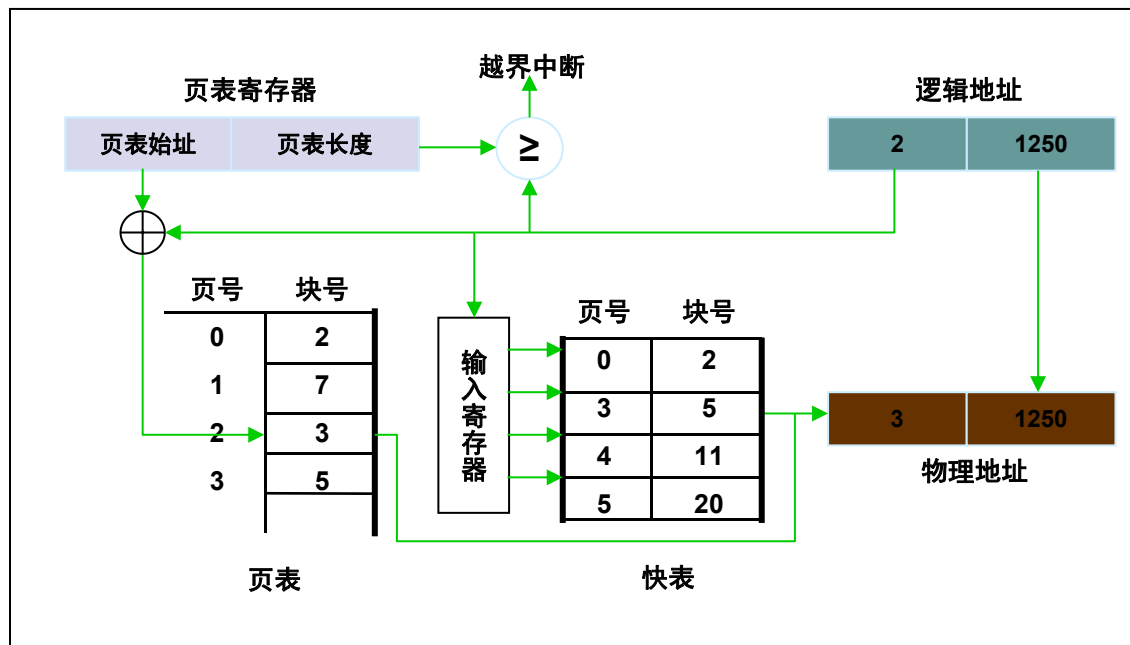
# 具有快表的地址变换机构





# 具有快表的地址变换机构

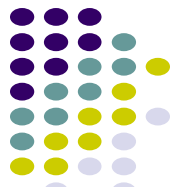
**快表**，又称**联想寄存器**（TLB，translation lookaside buffer），是一种访问速度比内存快很多的高速缓存（TLB不是内存！），用来存放**最近访问的页表项的副本**，可以加速地址变换的速度。与此对应，内存中的页表常称为慢表



## 地址映射过程:

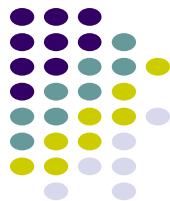
将页号P送入快表，若有此页号，则读出该页对应的物理块号；若无，则访问页表

将物理块号送入地址寄存器，并将此页表项存入快表。若快表已满，则换出一个不再用的页表项



# 知识回顾

	地址变换过程	访问一个逻辑地址的访存次数
基本地址变换机构	<ul style="list-style-type: none"><li>①算页号、页内偏移量</li><li>②检查页号合法性</li><li>③查页表，找到页面存放的内存块号</li><li>④根据内存块号与页内偏移量得到物理地址</li><li>⑤访问目标内存单元</li></ul>	两次访存
具有快表的地址变换机构	<ul style="list-style-type: none"><li>①算页号、页内偏移量</li><li>②检查页号合法性</li><li>③查快表。若命中，即可知道页面存放的内存块号，可直接进行⑤；若未命中则进行④</li><li>④查页表，找到页面存放的内存块号，并且将页表项复制到快表中</li><li>⑤根据内存块号与页内偏移量得到物理地址</li><li>⑥访问目标内存单元</li></ul>	快表命中，只需一次访问内存 快表未命中，需要两次访存



# 两级页表引入

某计算机系统按字节寻址，支持 32 位的逻辑地址，采用分页存储管理，页面大小为4KB，页表项长度为 4B

4KB =  $2^{12}$ B，因此页内地址要用12位表示，剩余 20 位表示页号。

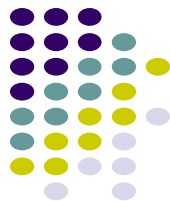
因此，该系统中用户进程最多有  $2^{20}$  页。相应的，一个进程的页表中，最多会有  $2^{20} = 1\text{M} = 1,048,576$  个页表项，所以一个页表最大需要  $2^{20} * 4\text{B} = 2^{22}$  B，共需要  $2^{22}/2^{12} = 2^{10}$ 个页框存储该页表。

根据页号查询页表的方法： $K$  号页对应的页表项存放位置 = 页表始址 +  $K * 4$   
要在所有的页表项都连续存放的基础上才能用这种方法找到页表项

## 两级页表

- 解决大页表占用大的连续存储空间的问题；
- 将在内存中离散分配的页表再建立一张页表，称为外层页表





# 如何解决单级页表的问题？

问题一：页表必须连续存放，因此当页表很大时，需要占用很多个连续的页框。

问题二：没有必要让整个页表常驻内存，因为进程在一段时间内可能只需要访问某几个特定的页面



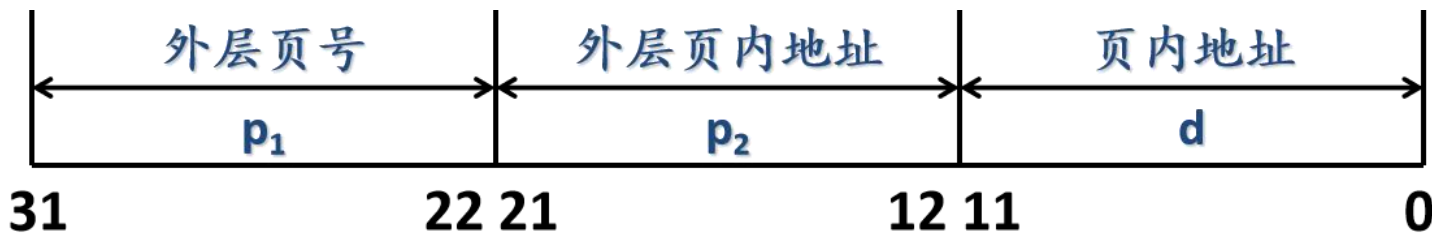
把页表再分页并离散存储，然后再建立一张页表记录页表各个部分的存放位置，称为页目录表，或称外层页表，或称顶层页表

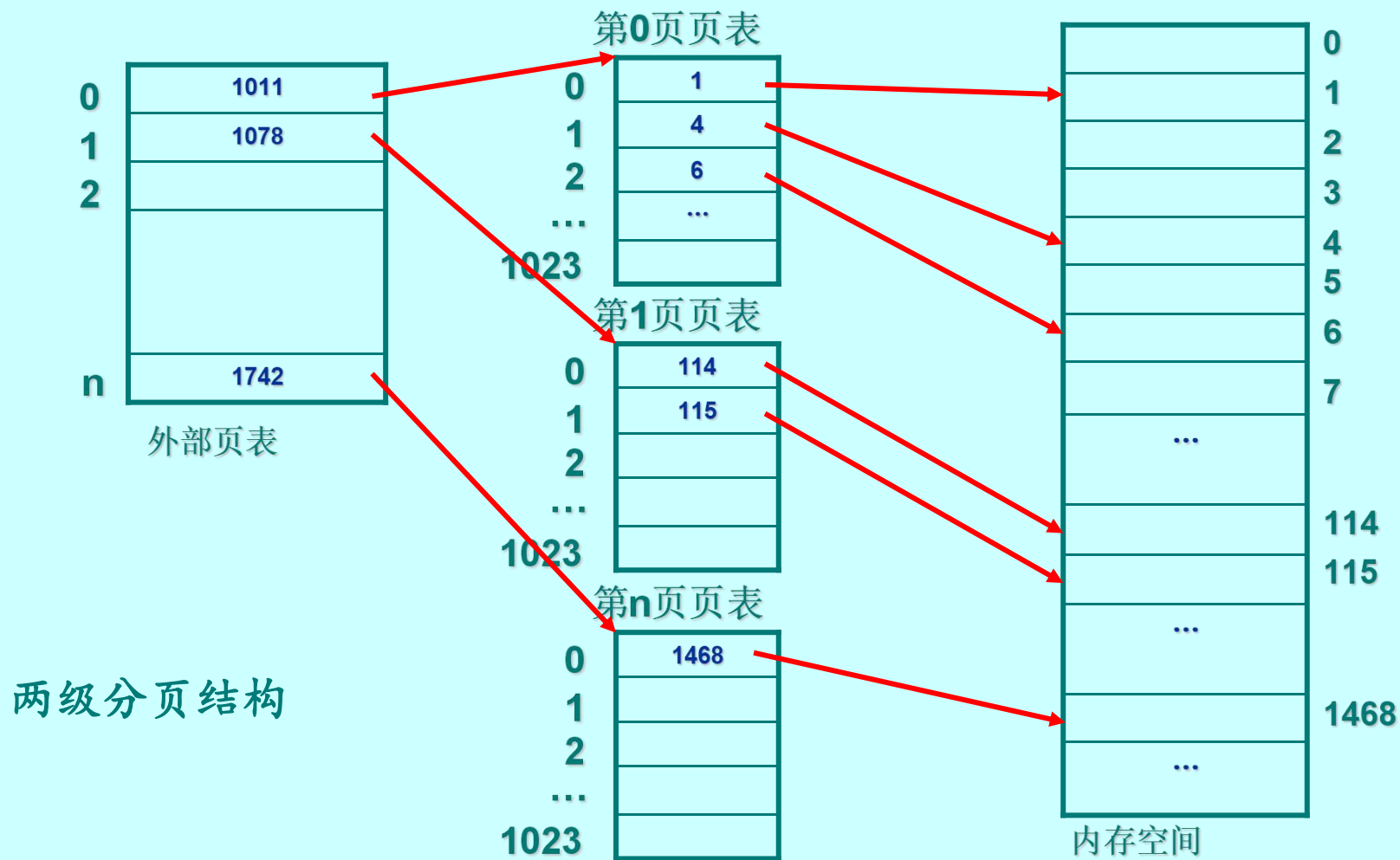


## 两级页表

- 将页表分页，并离散地将各个页面分别存放在不同的物理块中，同时为离散分配的页表再建立一张页表，称为**外层页表**，其每个页表项记录了页表页面的物理块号

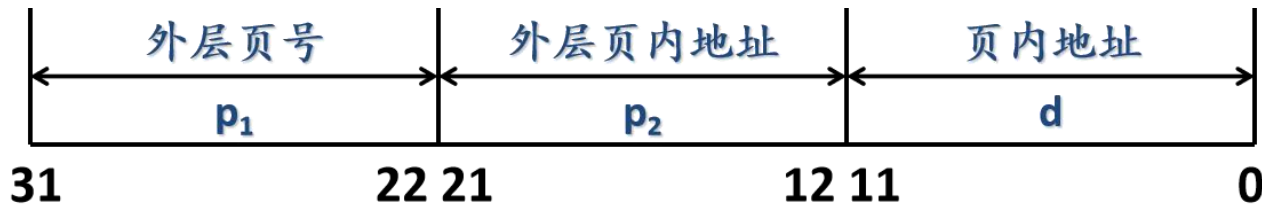
例如：**32位**逻辑地址空间，页面大小为**4KB**(即**12位**)，若采用一级页表机构，应有**20位**页号，即页表项应有**1M**个；在采用两级页表机构时，再对页表进行分页，使每页包含 **$2^{10}$** (即**1024**)个页表项，最多允许有 **$2^{10}$** 个页表分页。即







# 如何实现地址变换

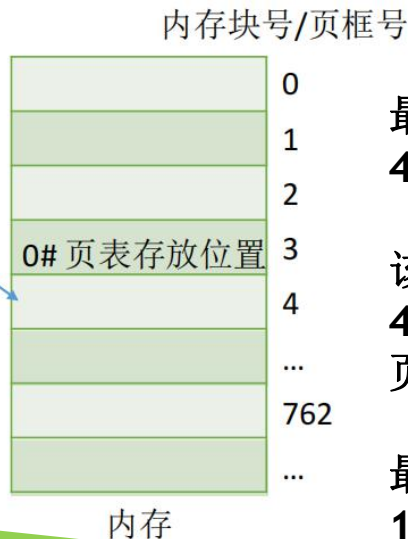


内存块号

0	3
1	...
.....	
1023	...

内存块号

0	2
1	4
.....	
1023	...



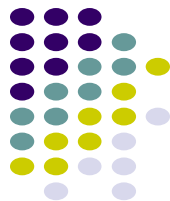
最终要访问的内存块号为  
4

该内存块的起始地址为  
 $4 * 4096 = 16384$   
页内偏移量为 4095

最终的物理地址为  
 $16384 + 4095 = 20479$

- ①按照地址结构将逻辑地址拆分成三部分
- ②从PCB中读出页目录表始址，再根据一级页号查页目录表，找到下一级页表在内存中的存放位置
- ③根据二级页号查二级页表，找到最终想访问的内存块号
- ④结合页内偏移量得到物理地址

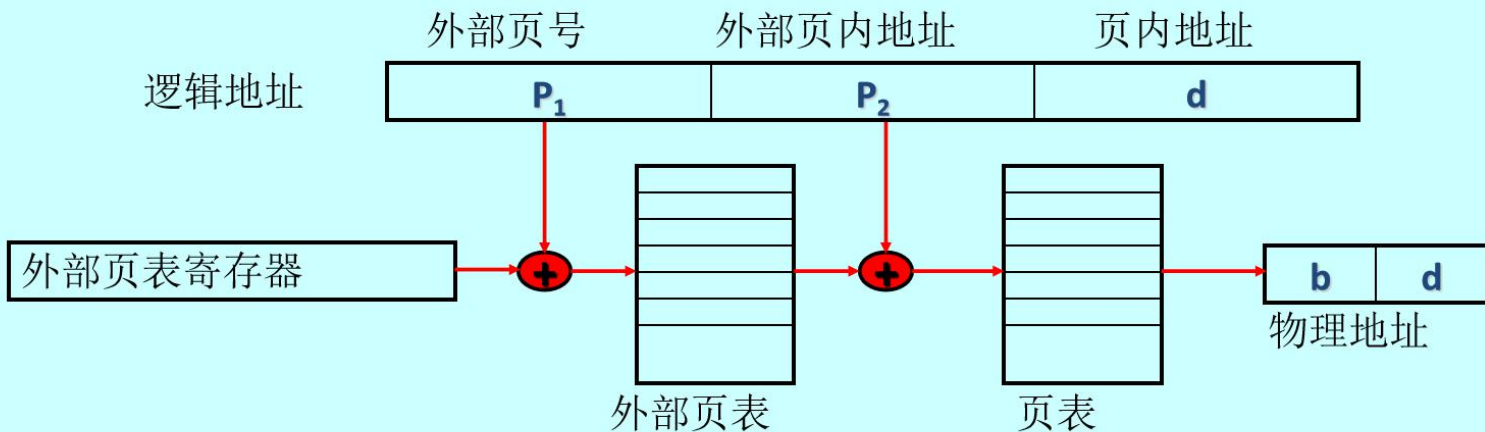
页表项存放位置：  
 $3 * 4096 + 1 * 4 = 12,292$



## 两级页表地址变化机构

为实现方便，在地址变换机构中需设一**外层页表寄存器**，用于存放**外层页表的始址**，并利用逻辑地址中的外层页号，作为外层页表的索引，从中找到指定页表分页的始址，再利用**p2**作为指定页表分页的索引，找到指定的页表项，其中即含有该页在内存的物理块号，用该块号和页内地址**d**即可构成访问的内存物理地址。

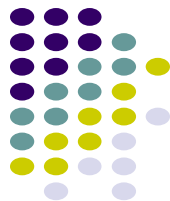
具有两级页表的地址变换机构：





## 两级页表

- 上述方法用离散分配空间解决了大页表无需大片存储空间的问题，但并未减少页表所占的内存空间
- **解决方法**:是把当前需要的一批页表项调入内存，以后再根据需要陆续调入。在采用两级页表结构的情况下，对正在运行的进程，必须将其外层页表调入内存，而对页表则只需调入一页或几页。
- 这就需要在外层页表项中增设一个状态位S，用于表示该页表是否调入内存。



# 如何解决单级页表的问题？

问题：没有必要让整个页表常驻内存，因为进程在一段时间内可能只需要访问某几个特定的页面。

可以在需要访问页面时才把页面调入内存（虚拟存储技术）。可以在页表项中增加一个标志位，用于表示该页面是否已经调入内存

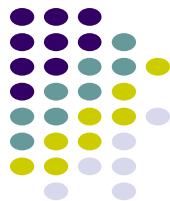
一级页号	内存块号	是否在内存中
0	3	是
1	8	否
.....	.....	...
1023		

若想访问的页面不在内存中，则产生缺页中断（内中断/异常），然后将目标页面从外存调入内存

二级页号	内存块号	是否在内存中
0	2	是
1	4	是
.....		
1023	...	

	0
	1
	2
0# 页表存放位置	3
	4
	...
	762
	...

内存



# 多级页表

若分为两级页表后，页表依然很长，则可以采用**更多级页表**，一般来说**各级页表的大小不能超过一个页面**

例：某系统按字节编址，采用 **40 位**逻辑地址，页面大小为 **4KB**，页表项大小为 **4B**，假设采用纯页式 存储，则要采用（ ）级页表，页内偏移量为（ ）位？

页面大小 = **4KB** =  $2^{12}\text{B}$ ，按字节编址，因此页内偏移量为**12位**

页号 = **40 - 12 = 28 位**

页面大小 =  $2^{12}\text{B}$ ，页表项大小 = **4B**，则每个页面可存放  $2^{12}/4 = 2^{10}$ 个页表项  
因此各级页表最多包含  $2^{10}$ 个页表项，需要 **10位**二进制位才能映射到  $2^{10}$ 个页表项，因此每一级的页 表对应页号应为**10位**。总共**28位**的页号至少要分为**三级**

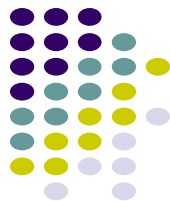
逻辑地址：



逻辑地址：

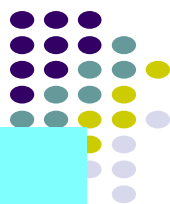




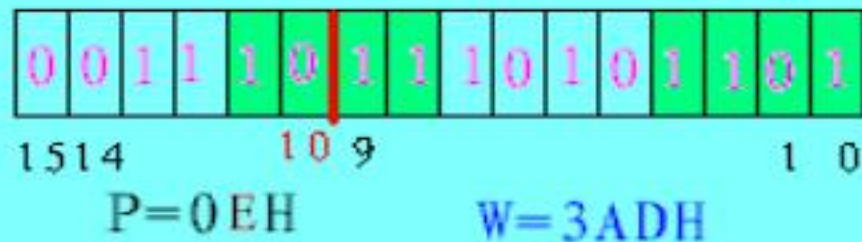


## 例题1

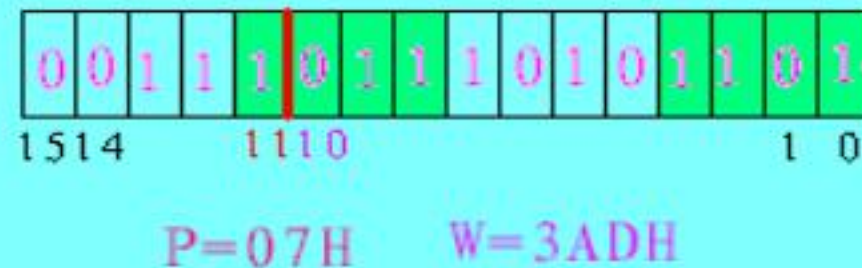
- 例1. 设页面大小为1KB，将逻辑地址3BADH划分为页号和页内偏移量两部分。用16进制表示。
- 例2. 设页面大小为2KB，将逻辑地址3BADH划分为页号和页内偏移量两部分。用16进制表示。



例 1：页面大小是 1KB，虚地址是 3BADH



例 2：页面大小是 2KB，虚地址是 3BADH





## 例题2

- 有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、A、5块，试将逻辑地址0AFEH转换成内存地址

虚地址 0AFEH

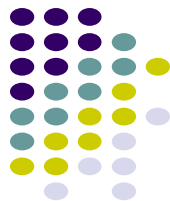
0000 1010 1111 1110

P=1 d=010 1111 1110

MR=0100 1010 1111 1110

=4AFEH

页号	块号
0	7
1	9
2	A
3	5



## 例题3

- 有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、10、5块，试将虚地址3412D转换成内存地址。

虚地址 3412

$$P = 3412 \% 2048 = 1$$

$$d = 3412 \bmod 2048 = 1364$$

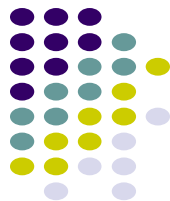
$$MR = 9 * 2048 + 1364 = 19796$$

虚地址3412的内存地址是：19796

解：页表：

页号 块号

0	7
1	9
2	10
3	5



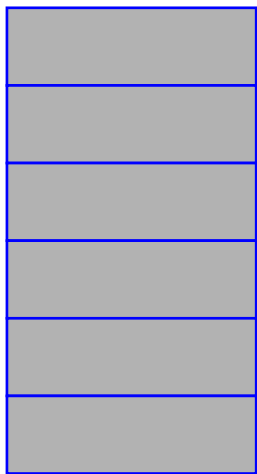
# 内容

- 4.1 程序的装入和链接
- 4.2 连续分配方式
- 4.3 基本分页存储管理方式
- 4.4 基本分段存储管理方式
- 4.5 虚拟存储器的基本概念
- 4.6 请求分页存储管理方式
- 4.7 页面置换算法
- 4.8 请求分段存储管理方式



# 分段存储管理方式的引入

- 分页从根本上克服了外零头（地址空间、物理空间都分割）。内存利用率提高。



} 最后一个页面可能未占满，还有内零头，  
平均浪费 $PL/2$ 大小的空间

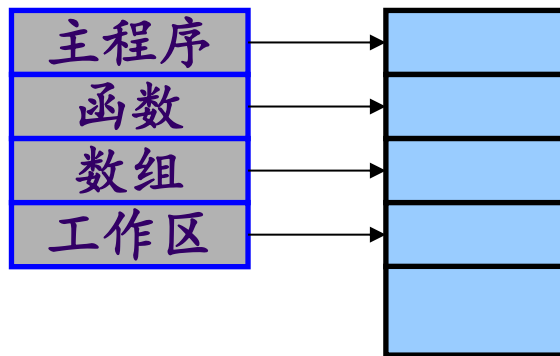


# 分页的缺点

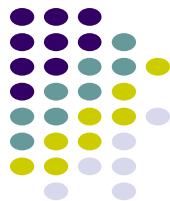
- 无论信息内容如何，按页长分割，分割后装入内存，有可能使逻辑完整的信息分到不同的页面，--执行速度降低
- 所以考虑以逻辑单位分配内存。如：



进程



内存



# 分段存储管理方式的引入

- 引入分段存储管理方式，主要是为了满足用户和程序员的需要
- 便于编程
  - 用户常把自己的作业按逻辑关系划分成若干个段，每段都有自己的名字，且都从零开始编址，这样，用户程序在执行中可用段名和段内地址进行访问。例如：LOAD 1, [A] | <D> 。
- 信息共享
  - 在实现程序和数据共享时，常常以信息的逻辑单位为基础，而分页系统中的每一页只是存放信息的物理单位，其本身没有完整的意义，因而不便于实现信息的共享，而段却是信息的逻辑单位，有利于信息的共享。





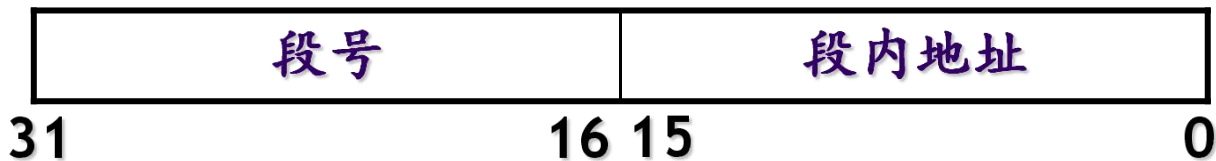
# 分段存储管理方式的引入

- **分段保护:**
  - 信息保护是对相对完整意义的**逻辑单位**(段)进行保护
- **动态链接:**
  - 当运行过程中又需要调用某段时，再将该段（目标程序）调入内存并链接起来。所以，动态链接是以段为基础的。
- **动态增长:**
  - 在实际系统中，有些数据段会不断地增长，而事先却无法知道数据段会增长到多大，分段存储管理方式能较好地解决这个问题



# 分段系统的基本原理

- 在分段存储管理方式中，作业地址空间被划分为若干个段，每个段定义了一组逻辑信息，都有自己的名字。通常用段号代替段名，每段从0开始编址，并采用一段连续地址空间。段长由逻辑信息组的长度决定。整个作业的地址空间分成多个段，逻辑地址由段号(段名)和段内地址所组成。



该地址结构允许一个作业最长有64K个段，每段的最大长度为64KB。

段号的位数决定了每个进程最多可以分几个段  
段内地址位数决定了每个段的最大长度是多少



# 基本分段式存储管理的实现-段表

- 为使程序正常运行，须在系统中为每个进程建立一张段映射表，简称“段表”。每个段在表中占有一个表项
- **段表结构：段号；段在内存中的起始地址(基址)；段长**
- 段表可以存放在寄存器中，但更多的是存放在内存中。
- 段表用于实现从逻辑段到物理内存区的映射

LOAD 1, [D] | <A>; //将分段D中A单元内的值读入寄存器1  
STORE 1, [X] | <B>; //将寄存器1的内容存入X 分段的B单元中

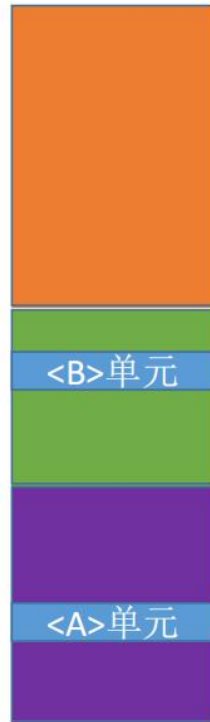
写程序时使用的  
段名 [D]、[X] 会  
被编译程序翻译  
成对应段号

<A>单元、<B>  
单元会被编译程  
序翻译成段内地  
址

段名: MAIN  
→段号: 0  
main 函数

段名: X  
→段号: 1  
某个子函数

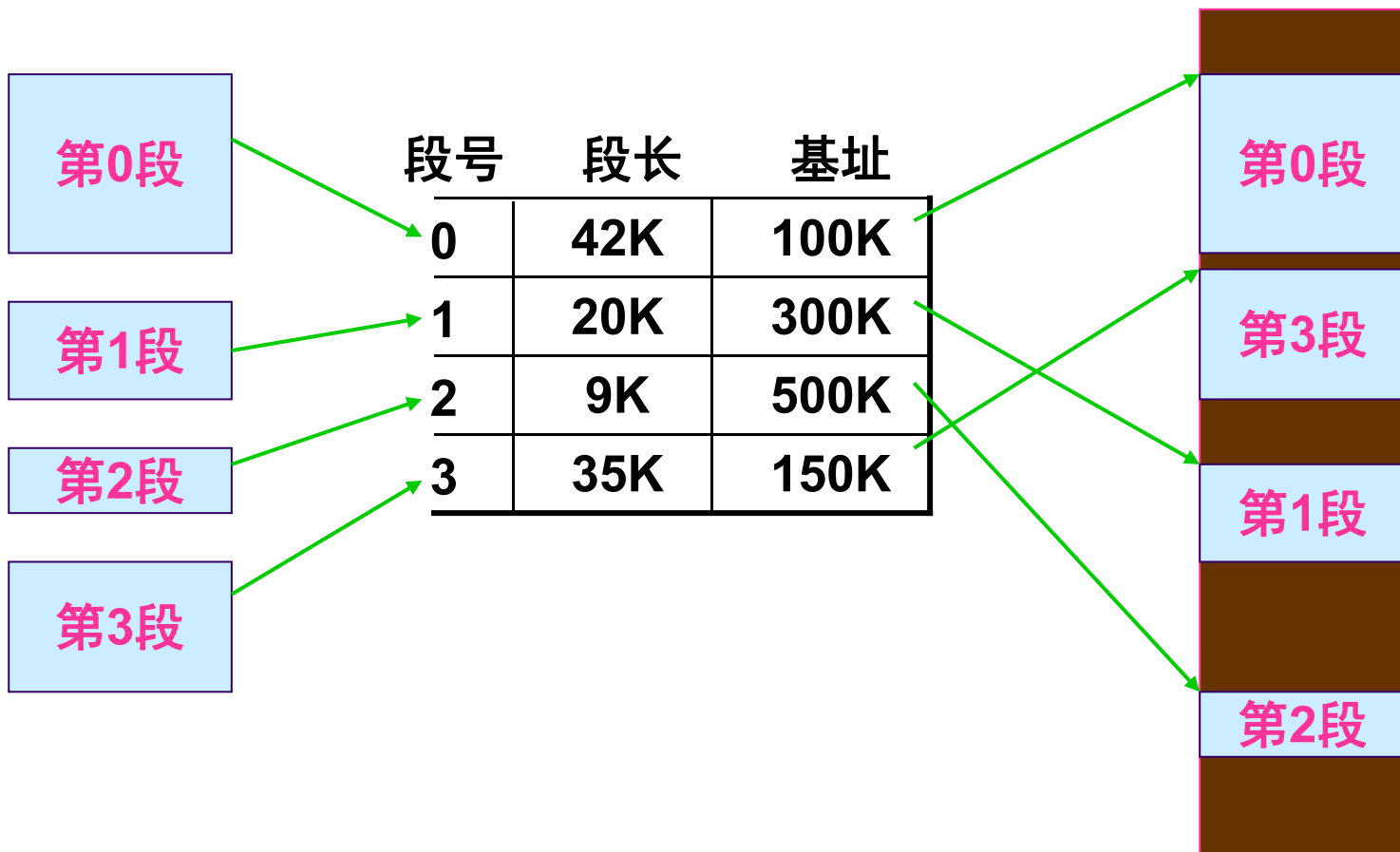
段名: D  
→段号: 2  
保存全局变量



用户作业

段表

内存





# 基本分段管理的地址变换

- 地址变换机构
- 在系统中设置段表寄存器，用于存放段表始址和段表长度，以实现从进程的逻辑地址到物理地址的变换。
- 当段表存放在内存中时，每访问一个数据，都需访问两次内存，降低了计算机的速率。
- 解决方法：设置联想寄存器，用于保存最近常用的段表项

注意：段表长度至少是1，而段号从0开始

## 段表寄存器

段表始址

段表长度M (4)

越界中断

≥

N

Y

②判断段号是否越界。若  $S \geq M$ ，则产生越界中断，否则继续执行

①根据逻辑地址得到段号、段内地址

## 逻辑地址

段号S 3

段内地址W 105

⑤ 计算得物理地址

段基址+段内地址

物理地址

153705

⑥ 访问目标内存单元

内存

段号 段长 基址

0	42K	100K
1	20K	300K
2	9K	500K
3	35K	150K

## 段表

④检查段内地址是否超过段长。若  $W \geq C$ ，则产生越界中断，否则继续执行

③查询段表，找到对应的段表项，段表项的存放地址为  $F+S \times \text{段表项长度}$



# 分段与分页的主要区别

- 相似点：
  - 采用离散分配方式，通过地址映射机构实现地址变换
- 不同点：
  - 页是信息的物理单位，分页是为了满足系统的需要；段是信息的逻辑单位，含有意义相对完整的信息，是为了满足用户的需要。
  - 页的大小固定且由系统确定，由系统把逻辑地址分为页号和页内地址，由机器硬件实现；段的长度不固定，取决于用户程序，编译程序对源程序编译时根据信息的性质划分。
  - 分页的作业地址空间是一维的；分段的作业地址空间是二维的,需要给出段名和段内地址。

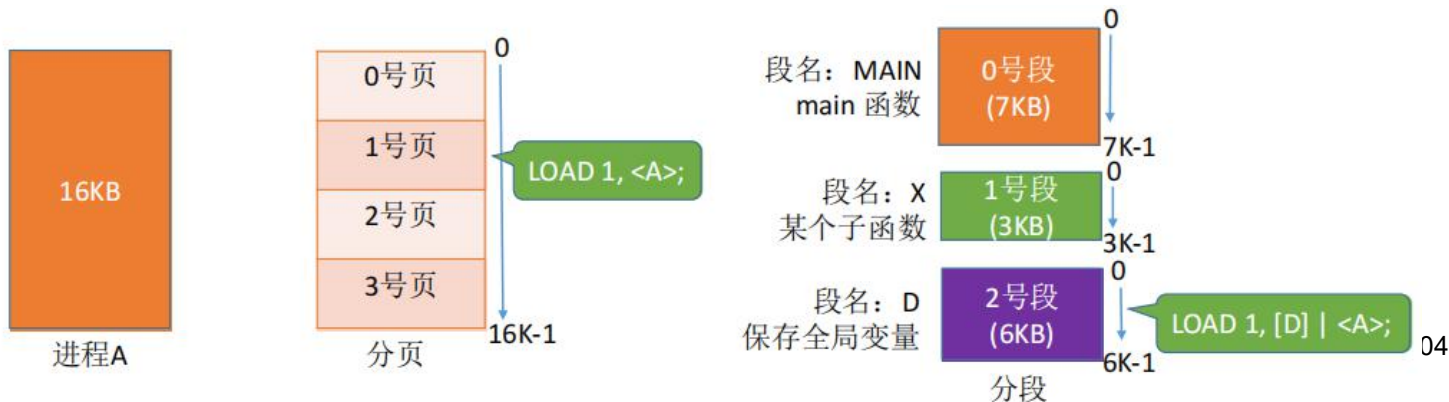


# 分段与分页的主要区别

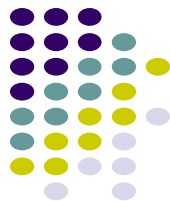
**页是信息的物理单位。**分页的主要目的是为了实现离散分配，提高内存利用率。分页仅仅是系统管理上的需要，完全是系统行为，**对用户是不可见的。**

**段是信息的逻辑单位。**分段的主要目的是更好地满足用户需求。一个段通常包含着一组属于一个逻辑模块的信息。分段**对用户是可见的**，用户编程时需要显式地给出段名。页的大小固定且由系统决定。段的长度却不固定，决定于用户编写的程序。

分页的用户进程地址空间是**一维**的，程序员只需给出一个记忆符即可表示一个地址。  
分段的用户进程地址空间是**二维**的，程序员在标识一个地址时，既要给出段名，也要给出段内地址

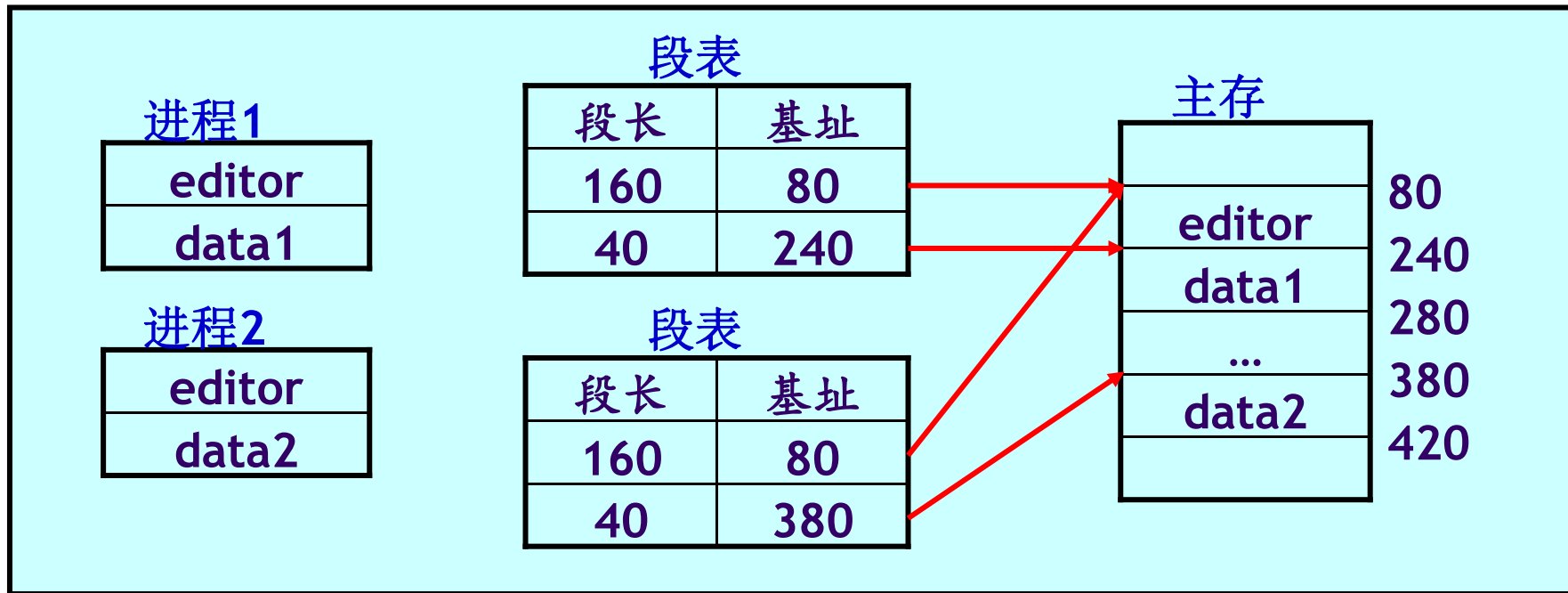






# 信息共享

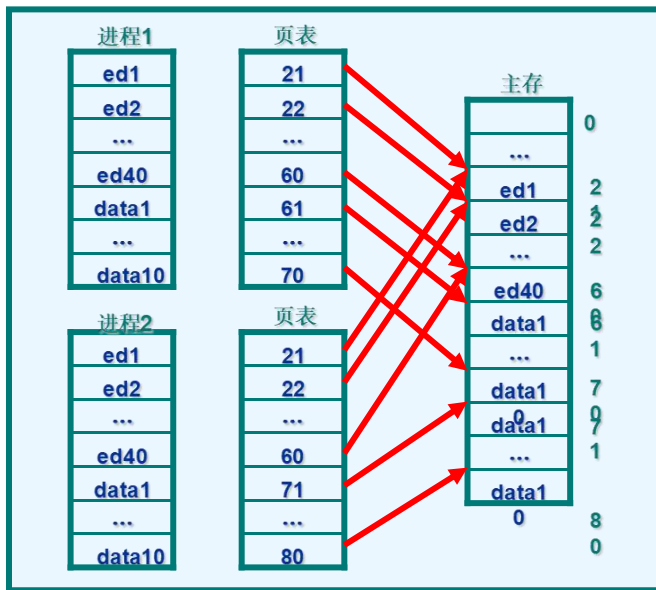
分段比分页更容易实现信息的共享和保护。不能被修改的代码称为**纯代码**或**可重入代码**（不属于临界资源），这样的代码是可以共享的。可修改的代码是不能共享的（比如，有一个代码段中有很多变量，各进程并发地同时访问可能造成数据不一致）





# 信息共享

- 分段系统的一个突出优点是易于实现段的共享和保护，允许若干个进程共享一个或多个分段，且对段的保护十分简单易行。
- 分页系统中虽然也能实现程序和数据共享，但远不如分段系统方便。





# 知识回顾

## 基本分段存储管理

分段

将地址空间按照程序自身的逻辑关系划分为若干个段，每段从0开始编址

每个段在内存中占据连续空间，但各段之间可以不相邻

逻辑地址结构：（段号，段内地址）

段表

记录逻辑段到实际存储地址的映射关系

每个段对应一个段表项。各段表项长度相同，由段号（隐含）、段长、基址组成

地址变换

1. 由逻辑地址得到段号、段内地址
2. 段号与段表寄存器中的段长度比较，检查是否越界
3. 由段表始址、段号找到对应段表项
4. 根据段表中记录的段长，检查段内地址是否越界
5. 由段表中的“基址+段内地址”得到最终的物理地址
6. 访问目标单元

分段 VS 分页

分页对用户不可见，分段对用户可见

分页的地址空间是一维的，分段的地址空间是二维的

分段更容易实现信息的共享和保护（纯代码/可重入代码可以共享）

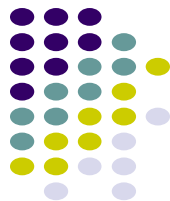
分页（单级页表）、分段访问一个逻辑地址都需要两次访存，分段存储中也可以引入快表机构



例：对于如下所示的段表，请将逻辑地址（0，137），（1，4000），（2，3600），（5，230）转换成物理地址。

段号	内存始址	段长
0	50k	10k
1	60k	3k
2	70k	5k
3	120k	8k
4	150k	4k

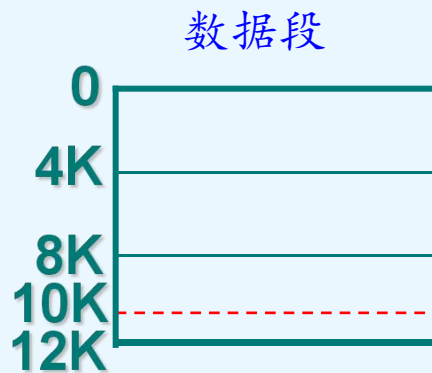
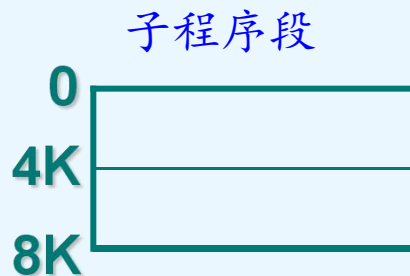
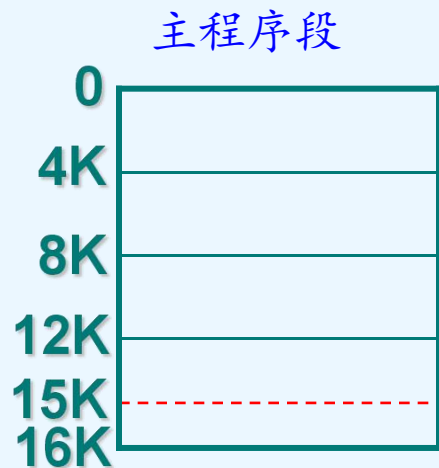
解：(4) 段号5 = 段表长，故段号不合法。产生越界中断。



# 段页式存储管理方式

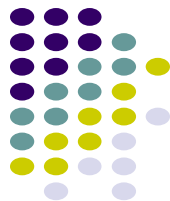
- 分段和分页存储管理方式各有优缺点。把两者结合成一种新的存储管理方式--**段页式存储管理方式**，具有两者的长处
  - 分页：**有效提高内存利用率**
  - 分段：**很好的满足用户需求**
- 基本原理
  - 先将用户程序分成若干段，再把每个段分成若干页，并为每个段赋予一个段名
- 地址结构与地址变换
  - **段号、段内页号、页内地址三部分**
  - 段表和页表

作业地址空间：

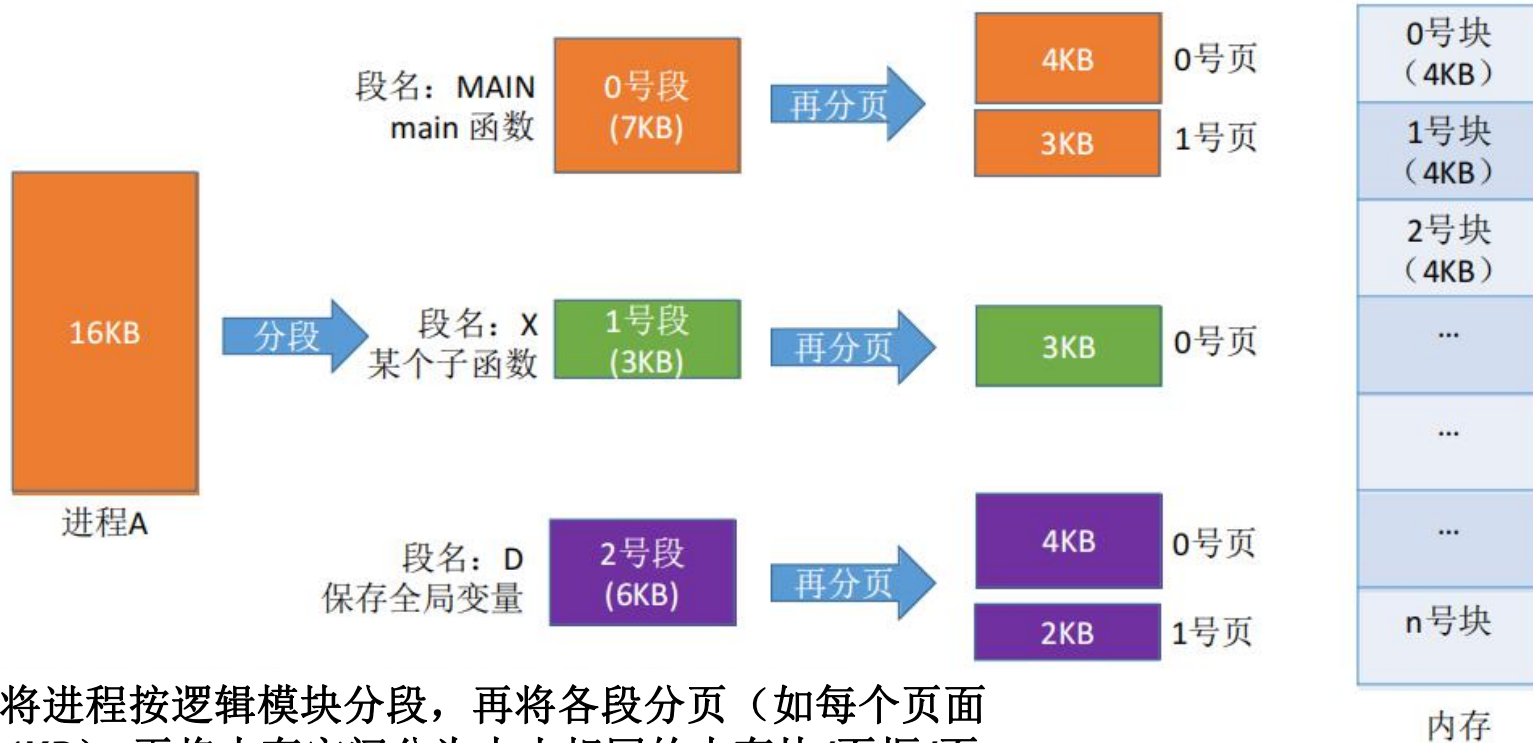


地址结构（逻辑地址）：

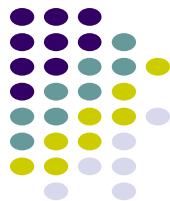
段号(S)	段内页号(P)	页内地址(W)
-------	---------	---------



# 段页式存储管理方式



将进程按逻辑模块分段，再将各段分页（如每个页面4KB）再将内存空间分为大小相同的内存块/页框/页帧/物理块，各页面分别装入各内存块中



# 段页式管理的逻辑地址结构



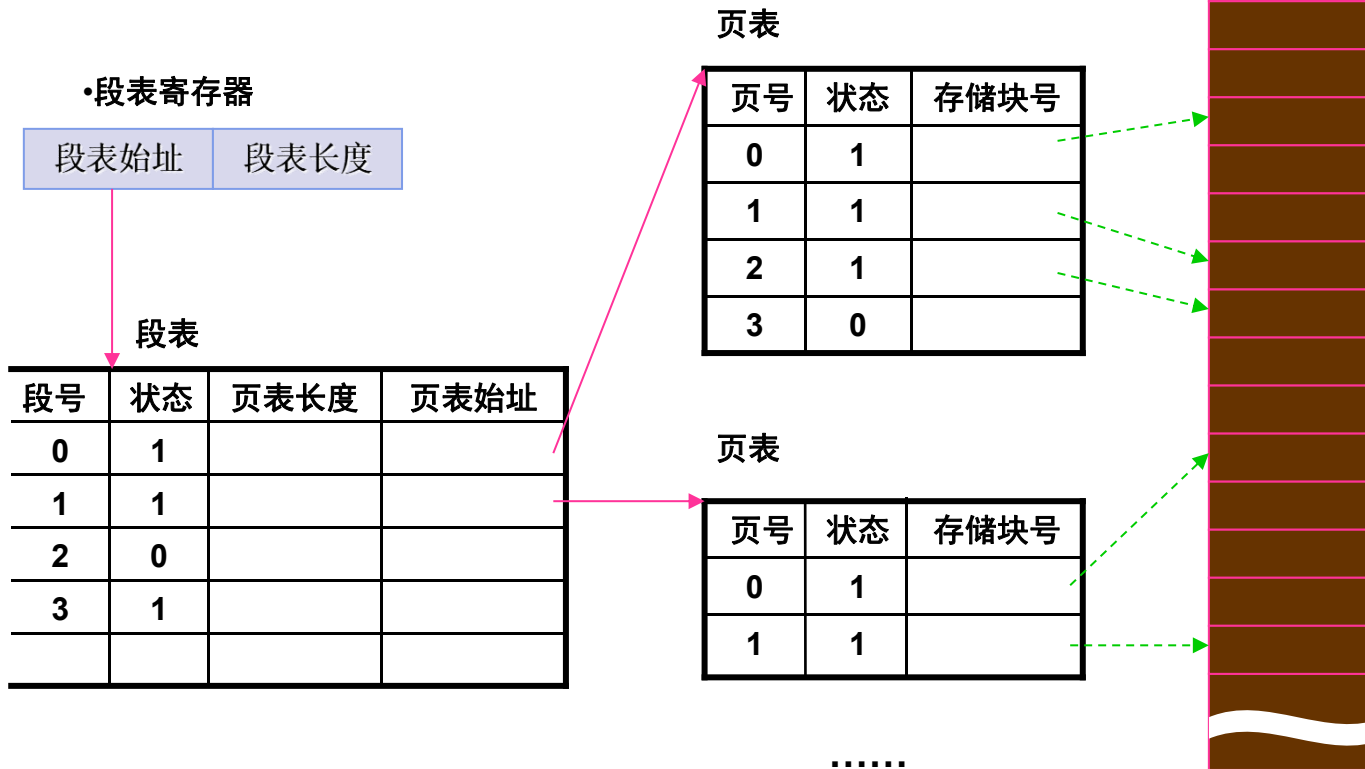
段号的位数决定了每个进程最多可以分几个段 页号位数决定了每个段最大有多少页 页内偏移量决定了页面大小、内存块大小是多少

在上述例子中，若系统是按字节寻址的，则段号占16位，因此在系统中，每个进程最多有  $2^{16} = 64K$  个段 页号占4位，因此每个段最多有  $2^4 = 16$  页 页内偏移量占12位，因此每个页面\每个内存块大小为  $2^{12} = 4096 = 4KB$

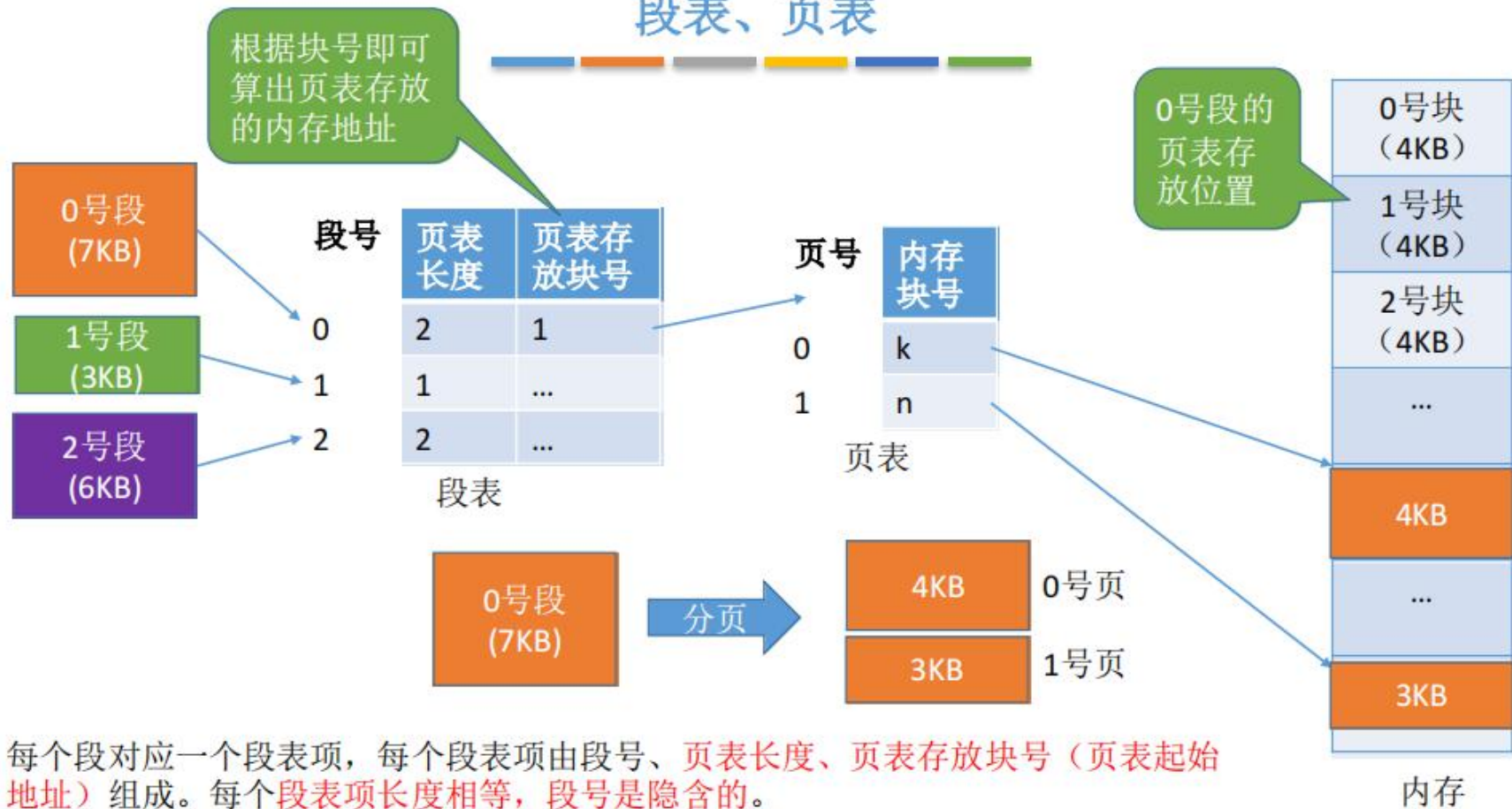




# 段页式管理地址映射



## 段表、页表

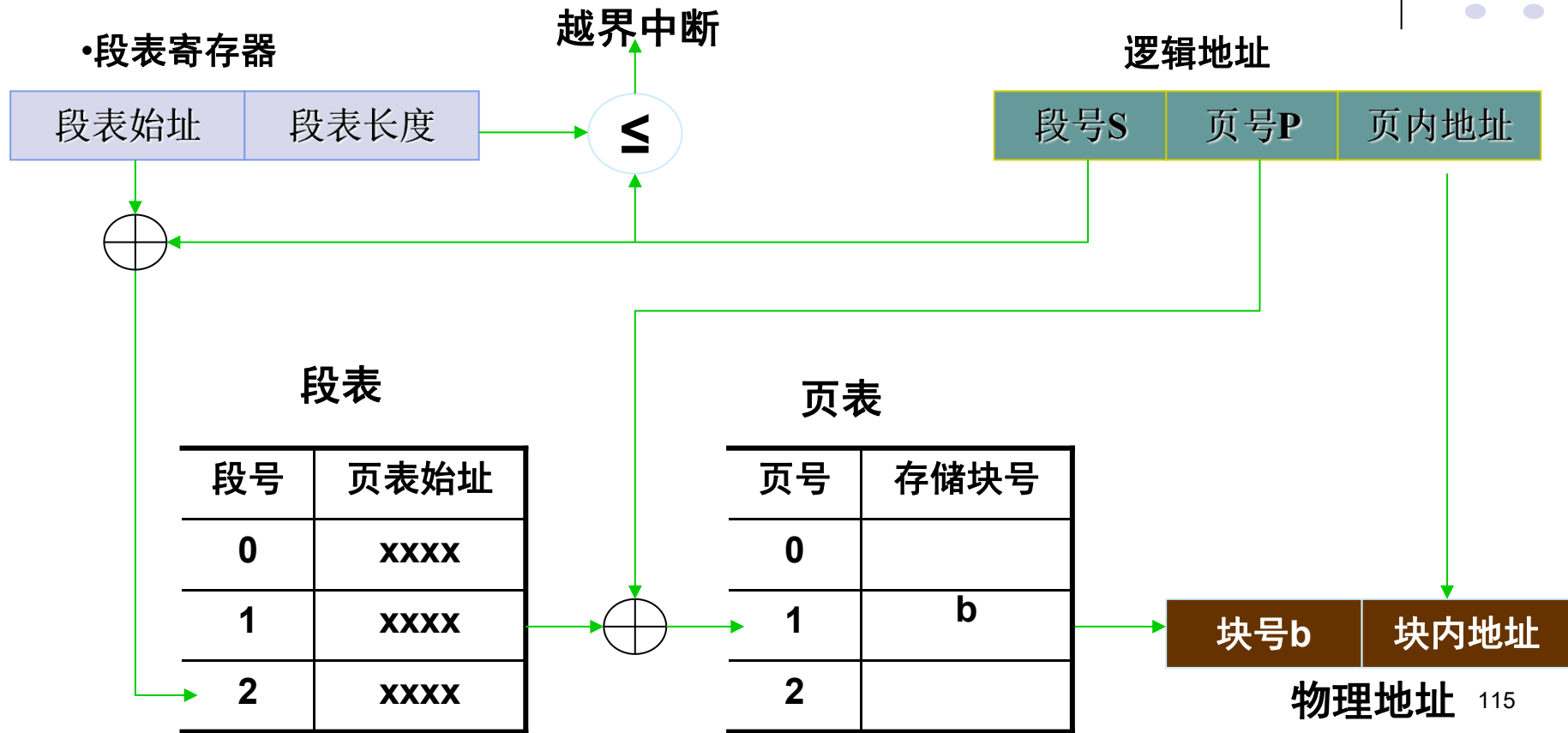


每个段对应一个段表项，每个段表项由段号、页表长度、页表存放块号（页表起始地址）组成。每个段表项长度相等，段号是隐含的。

每个页面对应一个页表项，每个页表项由页号、页面存放的内存块号组成。每个页表项长度相等，页号是隐含的。



# 段页式系统的地址变换机构

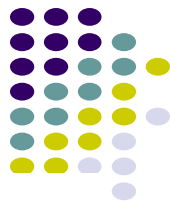




# 段页式存储管理的优缺点

- 同时具备分段和分页管理的优点：
  - 分散存储，内存利用率较高；
  - 便于代码或数据共享，支持动态链接等。
- 访问效率下降：
  - 一次访问转换成了三次访问。

# 知识回顾



## 段页式管理

