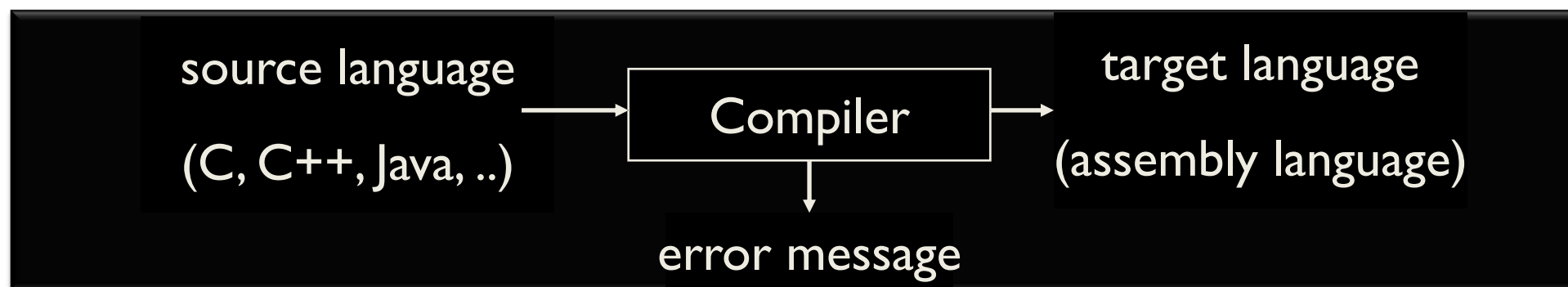


# 第八章:内存管理

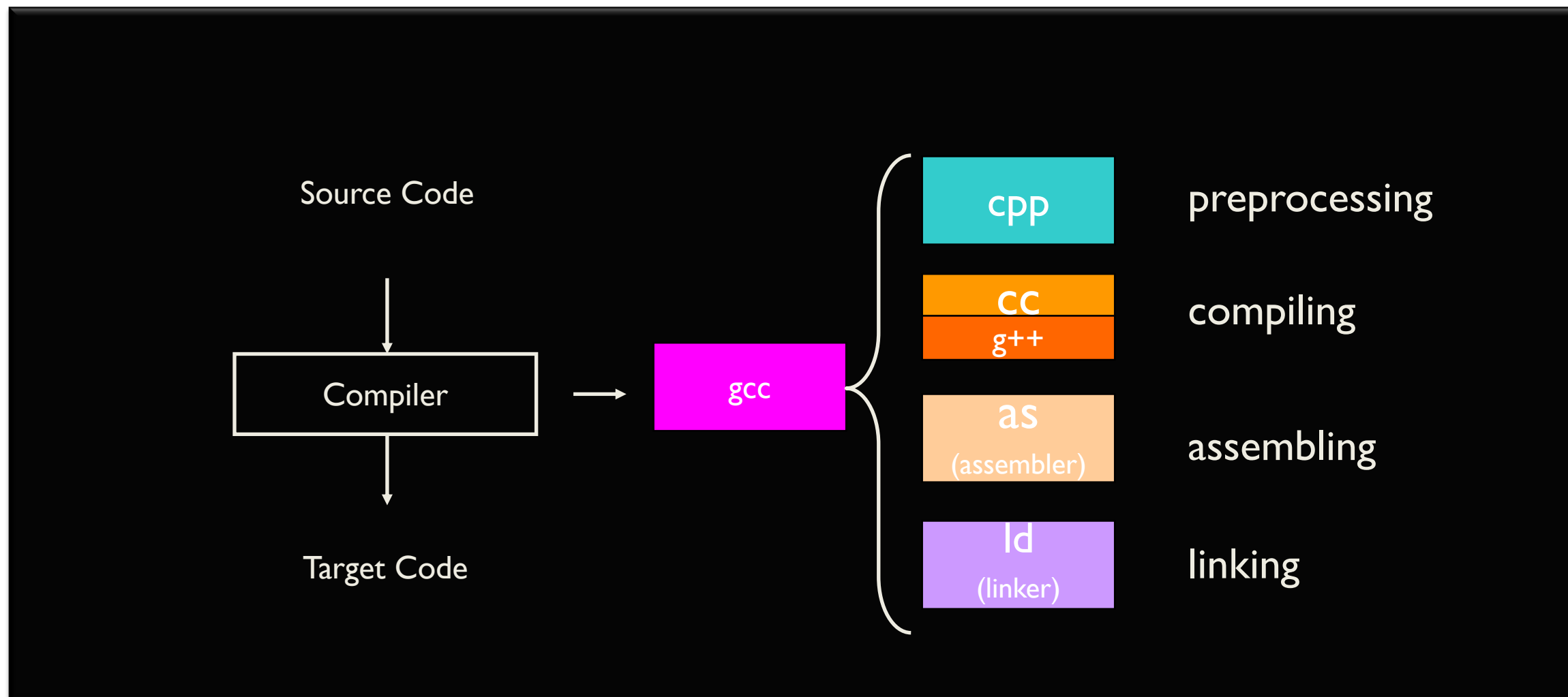
---

1. 介绍内存地址编址方法
2. 介绍内存分配方式

Compiler : Translate computer program from one language to another platform



1. Source Code is optimized for human readability
2. Target Machine Code is optimized for target hardware
3. **Goal of compiler** is that translate a source code program into an **equivalent** machine code efficiently



```
1. hello.c
2. #include <stdio>
3. void main (void)
4. {
5.     printf("hello\n");
6. }
```

## 1. Preprocessing

```
$ gcc -E -o hello.i hello.c
```

```
$ cpp hello.c hello.i
```

## 2. Compile

```
$ gcc -S -o hello.s hello.c
```

```
$ cc -S hello.i
```

## 3. Assembling

```
$ gcc -c -o hello.o hello.c
```

```
$ as -o hello.o hello.s
```

## 4. Linking

```
$ gcc -o hello hello.c
```

```
$ ld hello.o ( see the next slide )
```

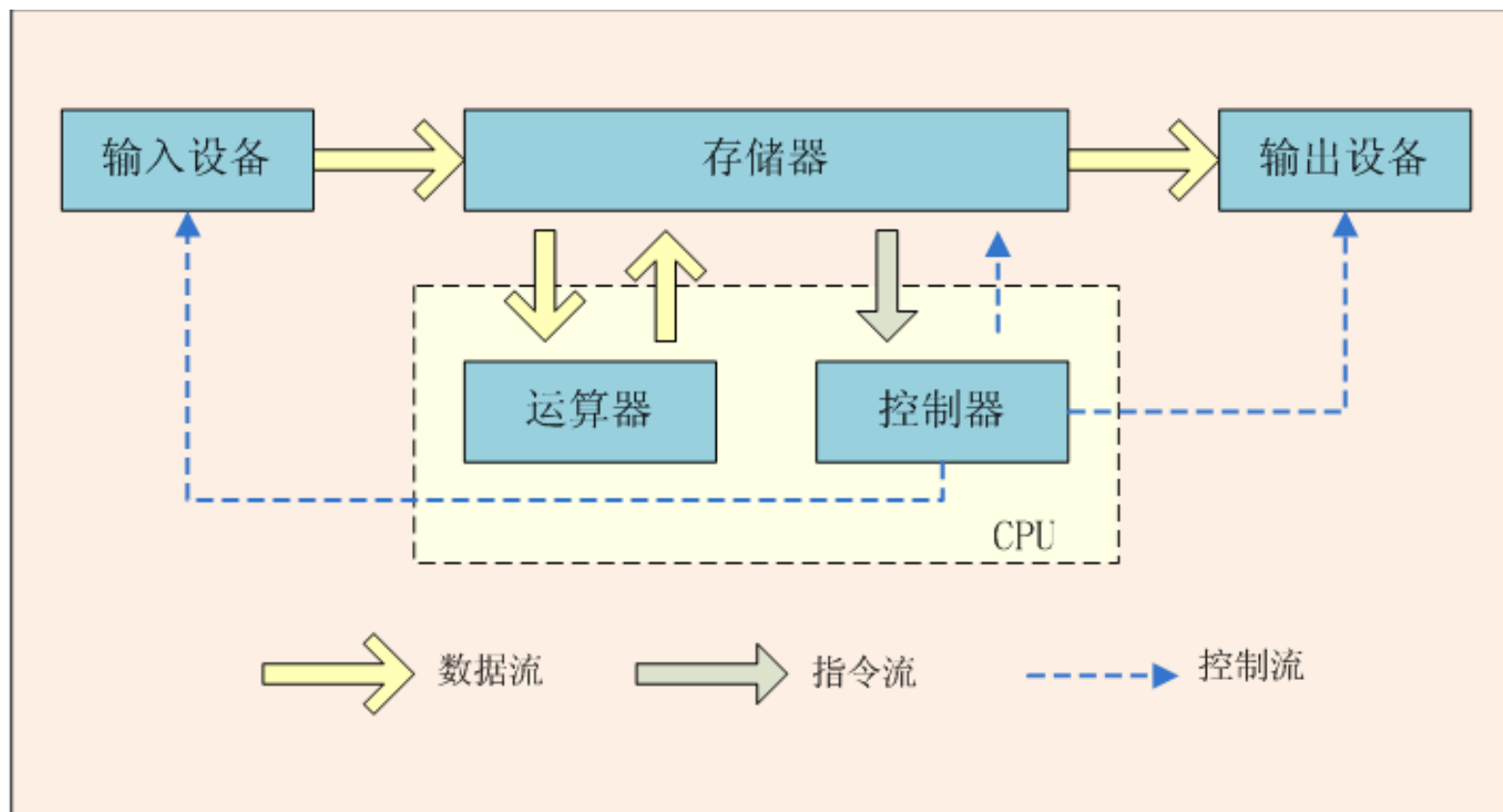
- Created hello.o file must be linked with the following object files:
  1. crt1.o, crti.o, crtn.o, crtbegin.o( for C++), crtend.o(for C++) .
  2. Dynamic link with “ld-linux-x86-64.so.2” dynamic link file
  3. With -lc option

### Command Example:

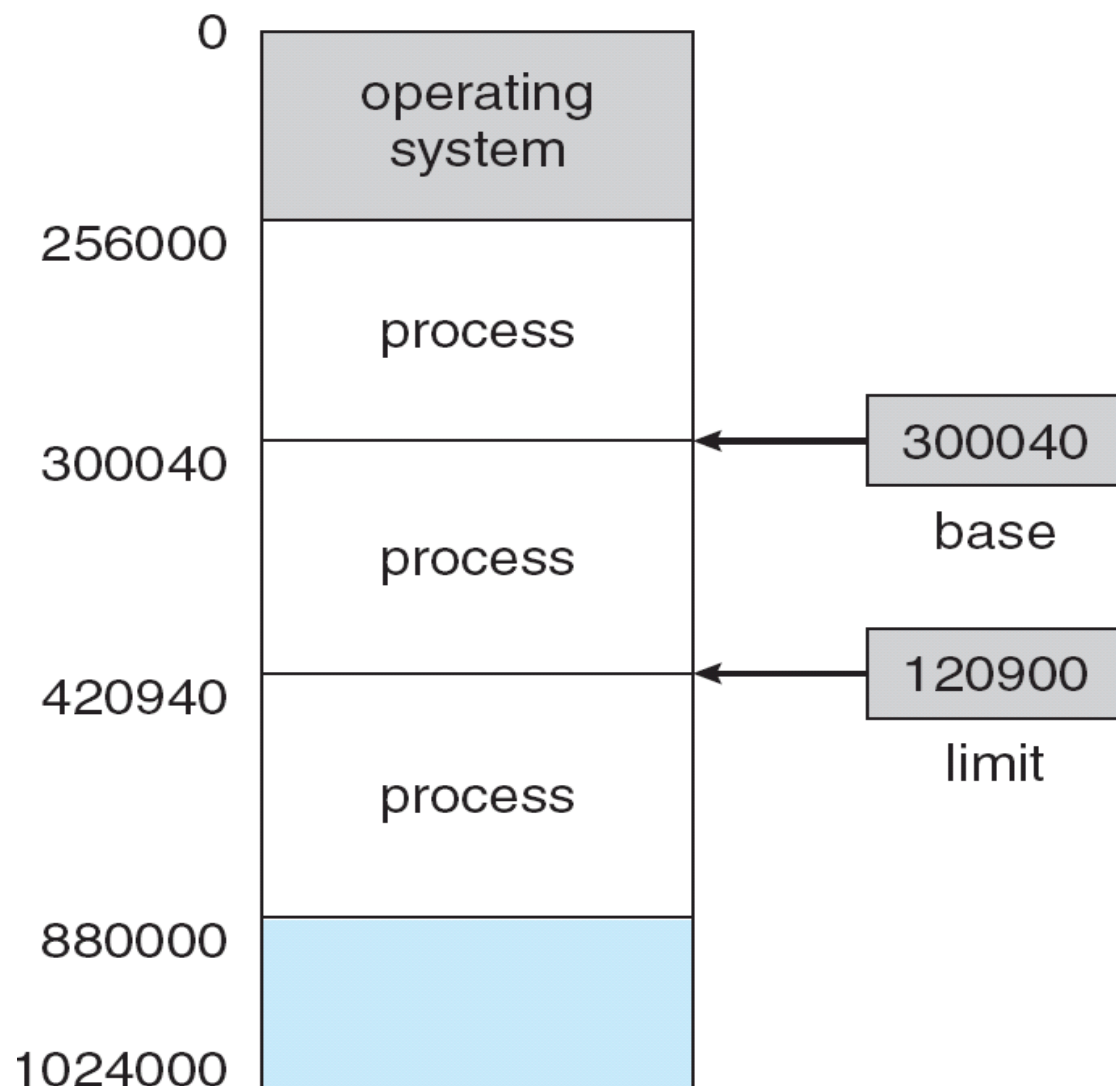
```
$ ld -o hello -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2  
hello.o /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o  
/usr/lib/gcc/x86_64-linux-gnu/4.8/crtbegin.o /usr/lib/gcc/x86_64-linux-  
gnu/4.8/crtend.o /usr/lib/x86_64-linux-gnu/crtn.o -lc
```

# 第一节、内存地址编址方法

- 为了运行程序，必须把程序从磁盘载入到内存
- 内存和寄存器是CPU唯一能直接访问的存储器

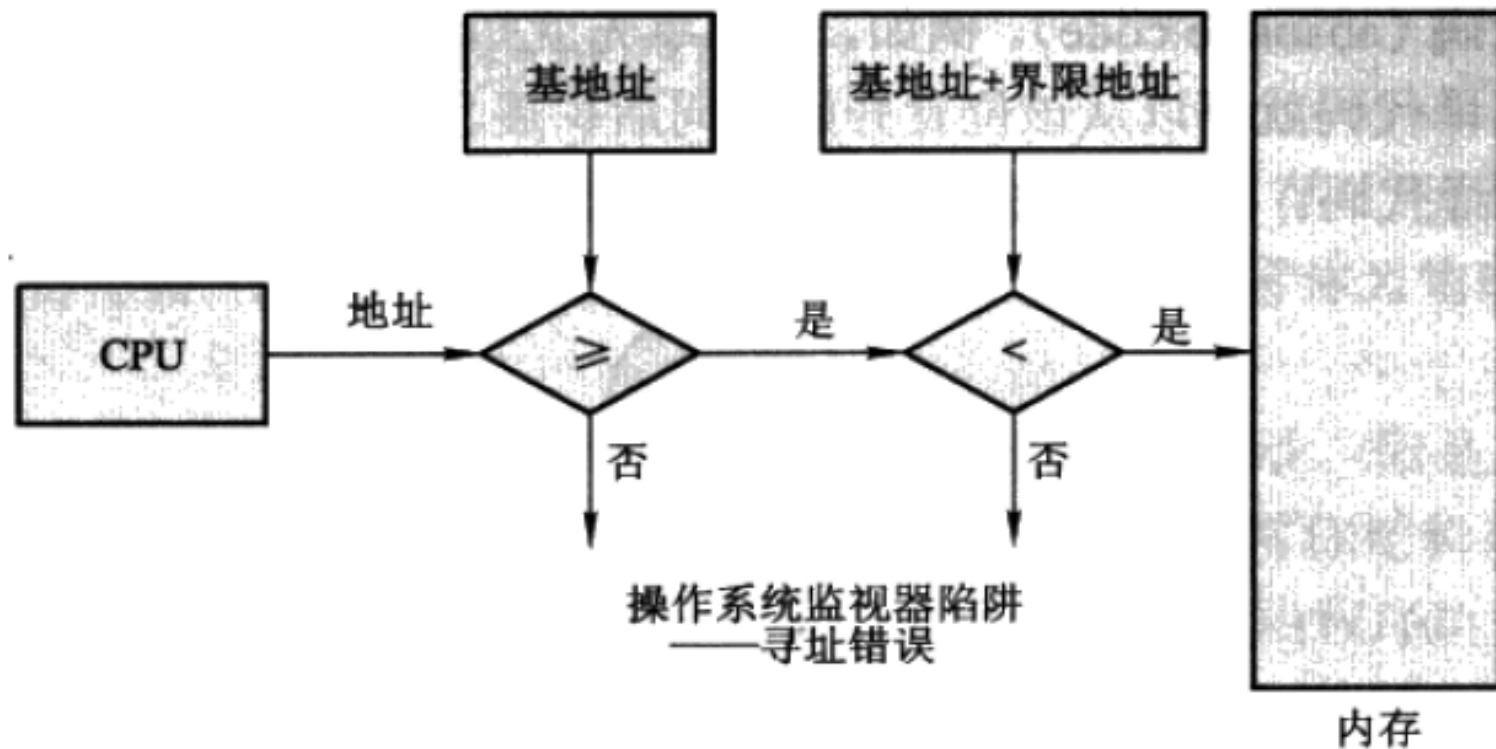


# 1.1 进程使用内存空间的界定



为确保进程只访问合法地址范围，一个进程使用的内存地址范围是由一对**基地址寄存器 (base register)**和**界限地址寄存器 (limit register)**来定义





为了确保进程正常运行必须保护内存，  
(1) 确保系统区域不被用户进程访问，  
(2) 确保用户进程不被其他进程访问。

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

地址绑定是逻辑地址到物理地址的映射

### 1、逻辑地址

- 也叫相对地址，用户程序在经过编译后形成的目标代码，其首地址一般为0，其余指令中的地址都是相对于首地址来编址。
- 不能用逻辑地址直接寻址

### 2、物理地址

- 也叫做内存地址，把内存分成很多个大小相等的存储单元，每个单元给一个编号，这个编号称为物理地址
- 物理地址可以直接寻址

/proc/\$pid/maps

```

root@hbpark-VirtualBox: /proc/12776
cat: mem: Permission denied
hbpark@hbpark-VirtualBox:/proc/12776$ su root
Password:
root@hbpark-VirtualBox:/proc/12776# cat mem
cat: mem: Input/output error
root@hbpark-VirtualBox:/proc/12776# cat maps
00400000-00401000 r-xp 00000000 08:01 305300 /home/hbpark/src/mem_str/hello
00600000-00601000 r--p 00000000 08:01 305300 /home/hbpark/src/mem_str/hello
00601000-00602000 rw-p 00001000 08:01 305300 /home/hbpark/src/mem_str/hello
7f9d0b839000-7f9d0b9f5000 r-xp 00000000 08:01 136551 /lib/x86_64-linux-gnu/libc-2.19.so
7f9d0b9f5000-7f9d0bbf4000 ---p 001bc000 08:01 136551 /lib/x86_64-linux-gnu/libc-2.19.so
7f9d0bbf4000-7f9d0bbf8000 r--p 001bb000 08:01 136551 /lib/x86_64-linux-gnu/libc-2.19.so
7f9d0bbf8000-7f9d0bbfa000 rw-p 001bf000 08:01 136551 /lib/x86_64-linux-gnu/libc-2.19.so
7f9d0bbfa000-7f9d0bbff000 rw-p 00000000 00:00 0
7f9d0bbff000-7f9d0bc22000 r-xp 00000000 08:01 136527 /lib/x86_64-linux-gnu/ld-2.19.so
7f9d0be07000-7f9d0be0a000 rw-p 00000000 00:00 0
7f9d0be1f000-7f9d0be21000 rw-p 00000000 00:00 0
7f9d0be21000-7f9d0be22000 r--p 00022000 08:01 136527 /lib/x86_64-linux-gnu/ld-2.19.so
7f9d0be22000-7f9d0be23000 rw-p 00023000 08:01 136527 /lib/x86_64-linux-gnu/ld-2.19.so
7f9d0be23000-7f9d0be24000 rw-p 00000000 00:00 0
7fff4896e000-7fff4898f000 rw-p 00000000 00:00 0 [stack]
7fff489fe000-7fff48a00000 r-xp 00000000 00:00 0 [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
root@hbpark-VirtualBox:/proc/12776#

```

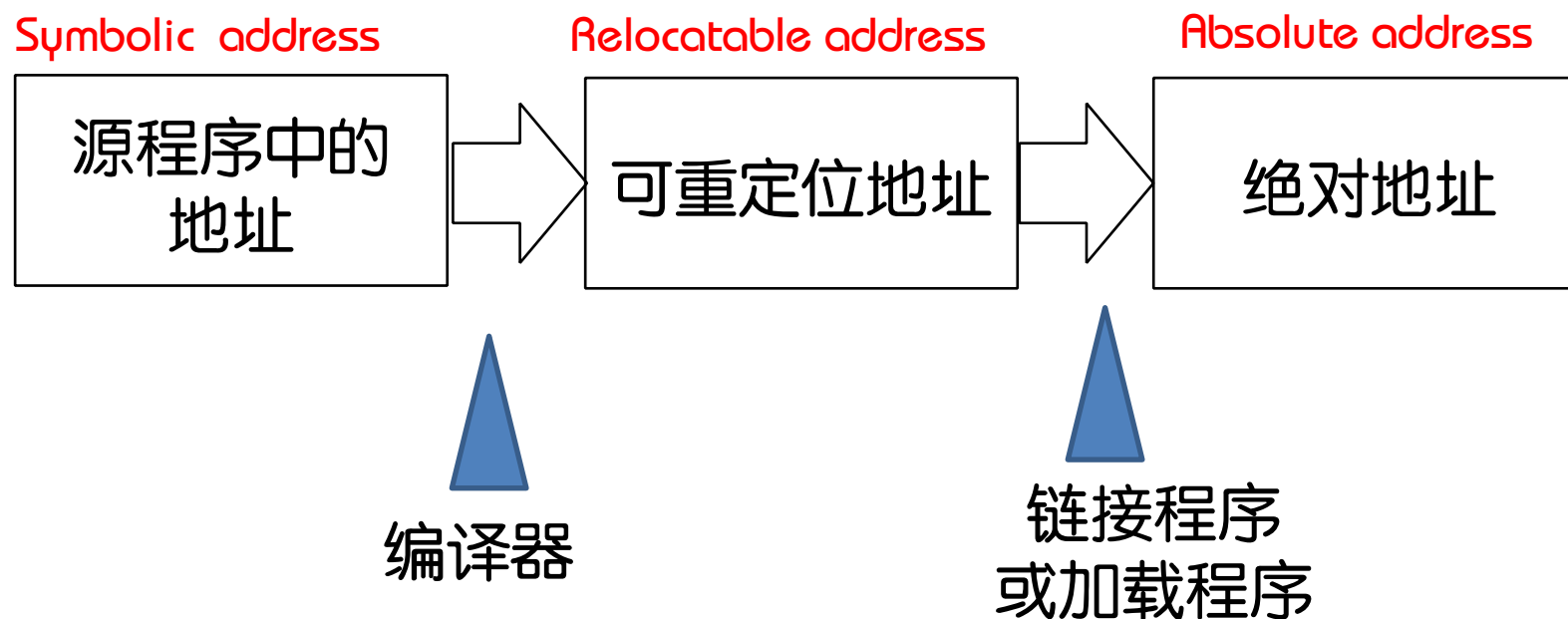
查看进程的虚拟地址空间是如何使用的。

该文件有6列，分别为：

1. 地址:库在进程里地址范围
2. 权限:虚拟内存的权限， $r$ =读， $w$ =写, $x$ =,  $s$ =共享,  $p$ =私有;
3. 偏移量:库在进程里地址范围
4. 设备:映像文件的主设备号和次设备号;
5. 节点:映像文件的节点号;
6. 路径: 映像文件的路径

- /proc/pagemap 文件中可以确认
- 这个文件允许一个用户的进程查看每个虚拟页映射到的物理页，每一个虚拟页都包含了一个64位的值

地址绑定是逻辑地址到物理地址的映射

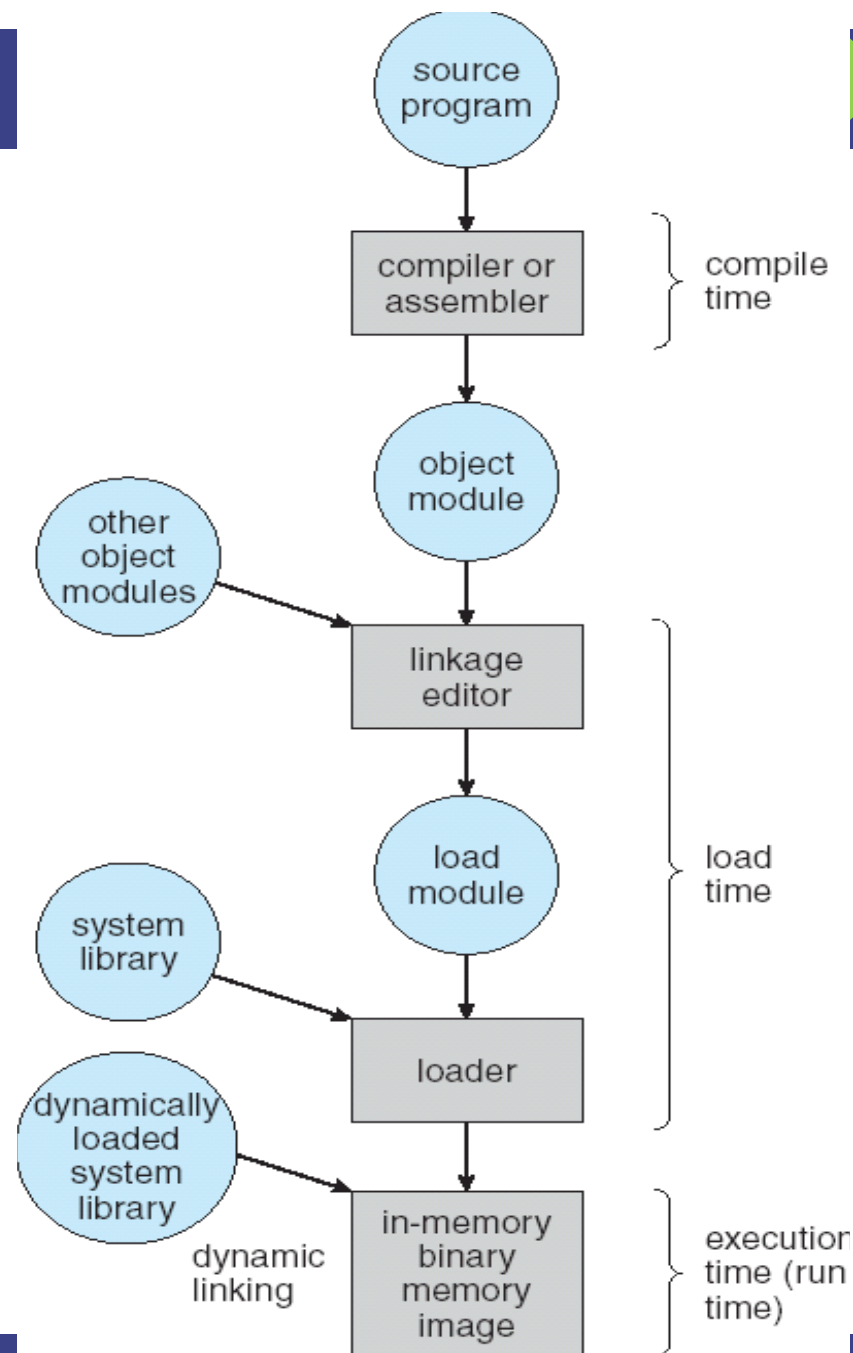


# 地址绑定

## 一个用户程序的多步骤处理过程

1. 编译器将符号地址绑定在可重定位的地址
2. 链接程序或加载程序将这些可重定位地址绑定成绝对地址

动态链接的  
系统库



通常，将指令和数据绑定到内存地址，有以下三种情况

### 1. 编译时

编译时就知道进程将内存中驻留地址

### 2. 加载时

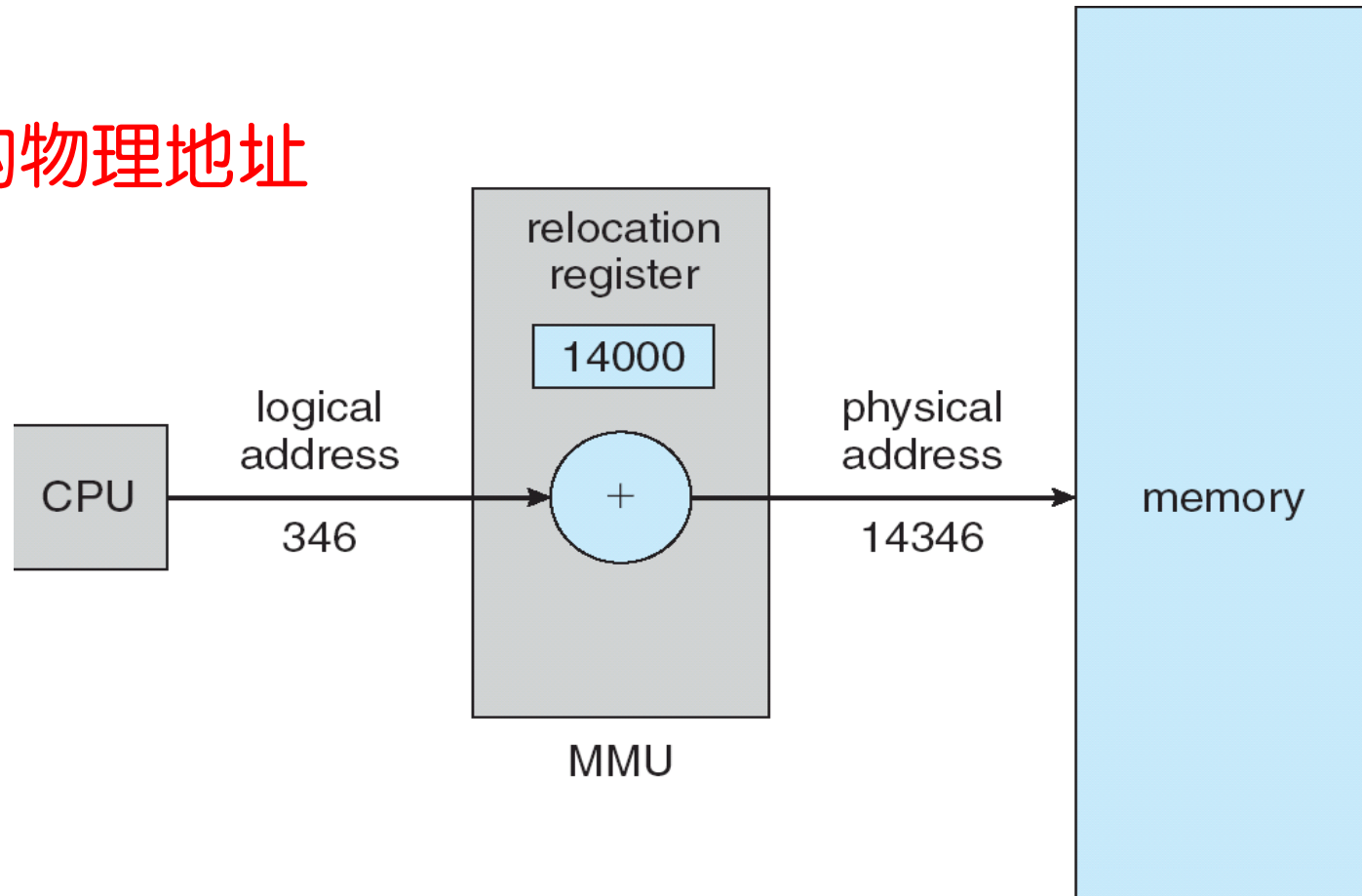
在编译时并不知道进程驻留内存地址，编译器必须生成可重定位代码(relocatable code)。对于这种情况，最后绑定延迟到加载时才进行。

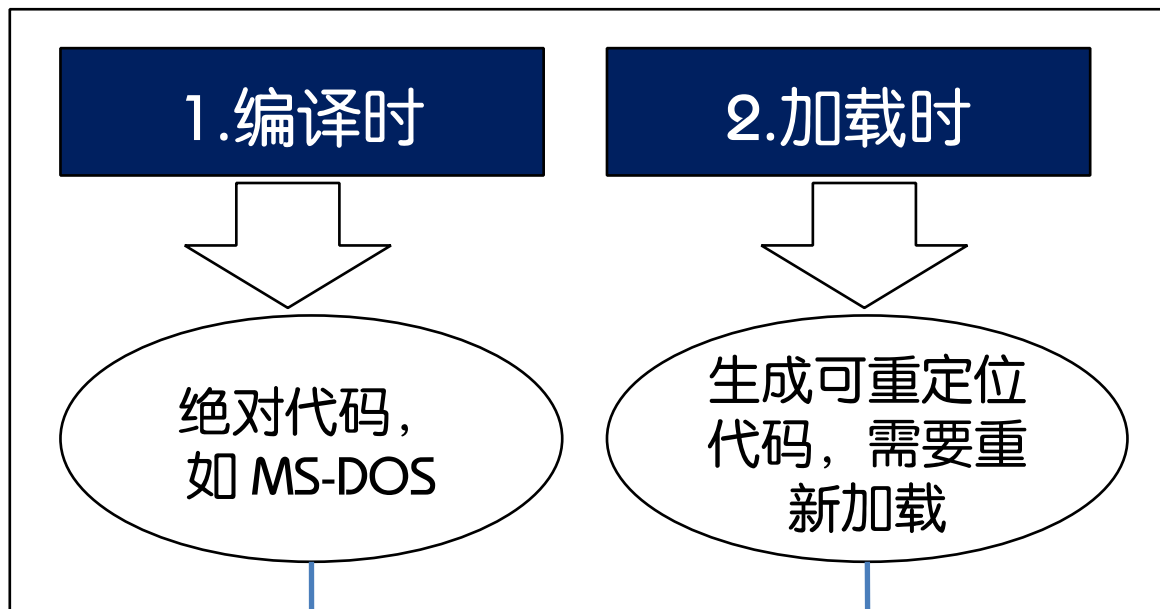
### 3. 运行时

如果进程在执行时可以从一个内存段移到另一个内存段，那么绑定必须延迟到执行时才进行，这需要特定的硬件支持(MMU)



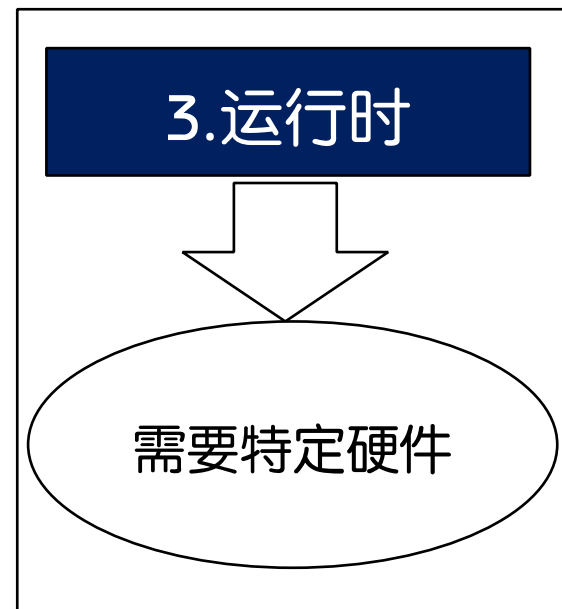
- 是映射虚拟地址为物理地址的硬件设备
- 用户进程所生成的地址在送交内存之前，都会加上重定位寄存器的值
- 用户程序看不到真正的物理地址





每次运行时驻留的  
内存地址空间一样

每次结束后，重新  
运行时驻留的内存  
地址空间有可能不  
一样



运行时(载入内存后)，  
可发生地址变化

运行时(载入内存后)，不可以发生地址变化

直到被调用之前，程序不会被载入到内存，即加载延迟到运行时

### 优点

- 内存使用率高:不使用的程序不会载入到内存
- 适合用户用大量代码来操作不常发生事件
- 不需要操作系统的特别支持，由程序员来设计

```

1  /*****
2      > File Name: test.c
3      > Author: ma6174
4      > Mail: ma6174@163.com
5      > Created Time: 2016年04月17日
6      *****/
7
8  #include <stdio.h>
9
10 void printHello()
11 {
12     printf("Hello World\n");
13 }
~

```

test.c

生成libtest.so 的动态加载库

```
$ gcc test.c -shared -fPIC -o libtest.so
```

```
1 /*****
2  > File Name: main.c
3  > Author: ma6174
4  > Mail: ma6174@163.com
5  > Created Time: 2016年04月17日 星期日 20时30分15秒
6  *****/
7
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include <dlfcn.h>
11
12 void main(void)
13 {
14     void * plib;
15     typedef void (*FUN_HELLO)();
16     FUN_HELLO funHello = NULL;
17
18     plib = dlopen("./libtest.so", RTLD_NOW| RTLD_GLOBAL);
19     if ( plib == NULL )
20         printf("error\n");
21     funHello = dlsym(plib,"printHello");
22     funHello();
23     dlclose(plib);
24 }
25
```

由程序员加载/卸  
载动态库

\$gcc main.c -o main -ldl

- 静态链接:加载程序合并到二进制程序镜像中，一直驻留在内存
- 动态链接，链接延迟到运行时
  1. 有个小程序,叫存根,用来定位适当的内存驻留库程序，或如果该程序不在内存时应如何装入库
  2. 存根首先检查所需子程序是否在内存中，如果不在，就将子程序装入内存
  3. 存根会用子程序地址来替换自己，并开始子程序
- 动态链接通常适用于系统库（如语言库），由操作系统管理

动态链接和动态加载都是需要时加载到内存。但，

- 动态链接

程序启动时建立了链接（即只有链接地址），需要时载入内存，由操作系统决定

- 动态加载

通过程序的方法来控制加载，由程序员决定

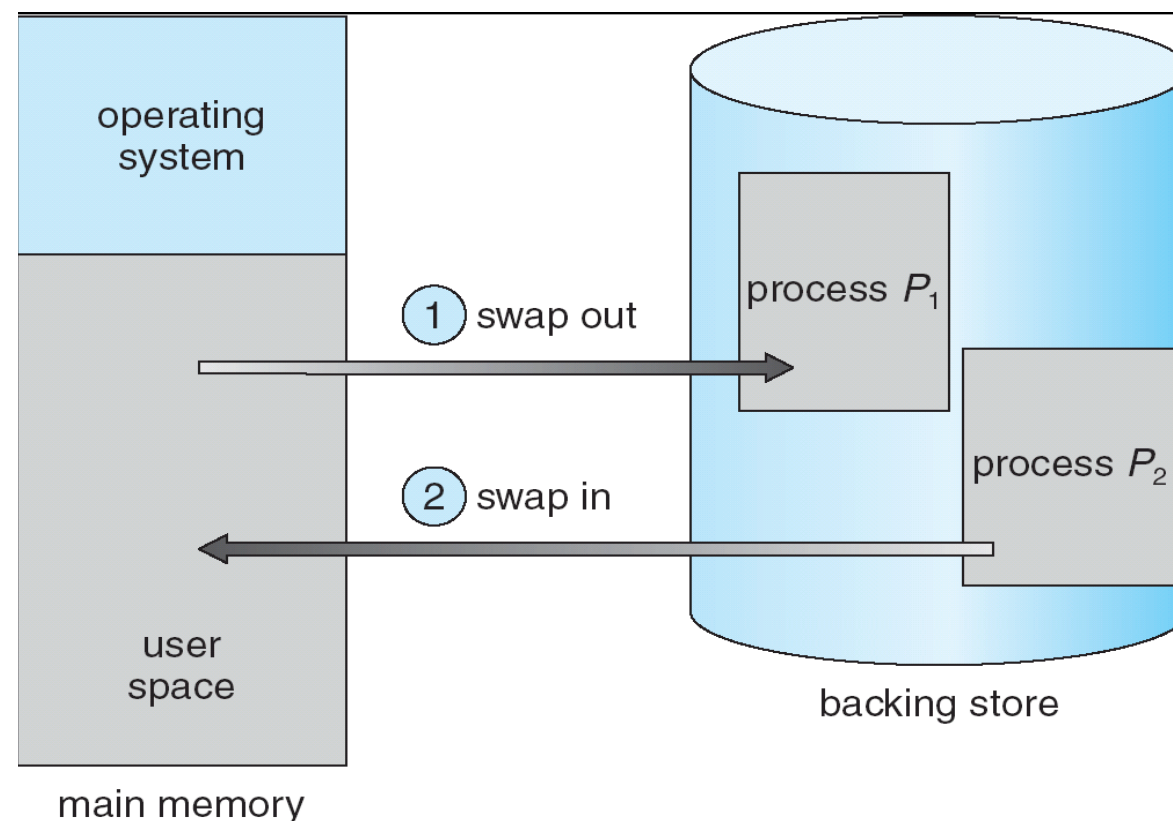
进程可以暂时从内存中交换到备份存储上(通常是快速磁盘), 当需要再次执行时再调回到内存

1. 优先级调度算法:低优先级交换出, 高优先级交换进;滚出(roll out), 滚入(roll in)
2. 交换时间(转移时间):转移时间与交换内存空间量成正比

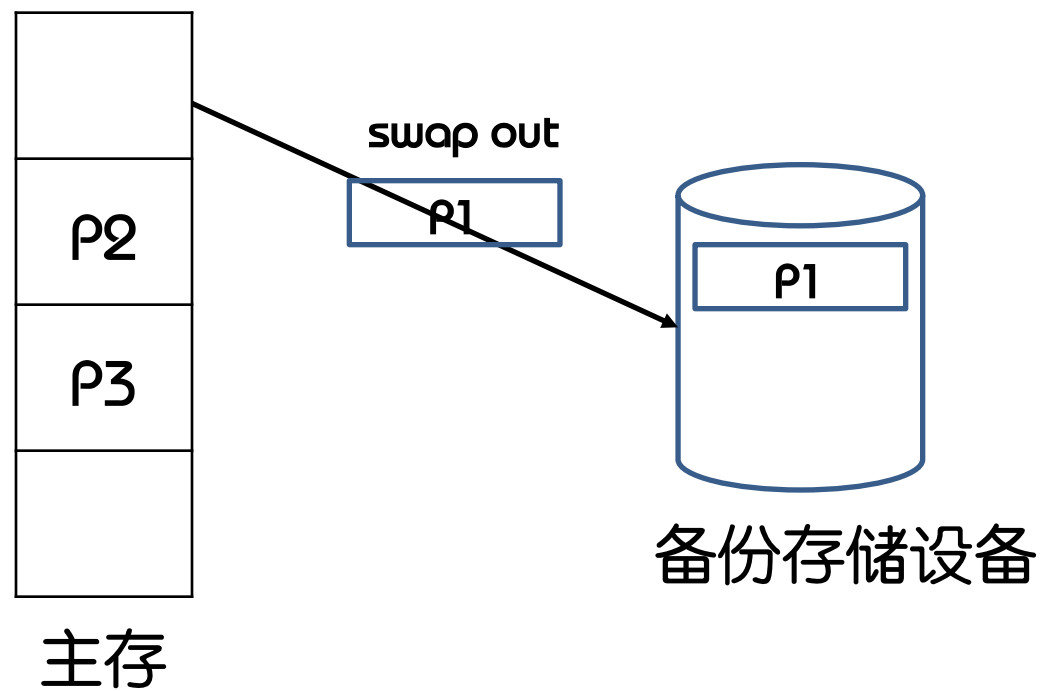
问:将交换出的进程再交换(调回)回来的时候, 应调回到哪个内存空间?应根据以下三种情况决定

1:编译时绑定 2:加载时绑定 3:运行时绑定

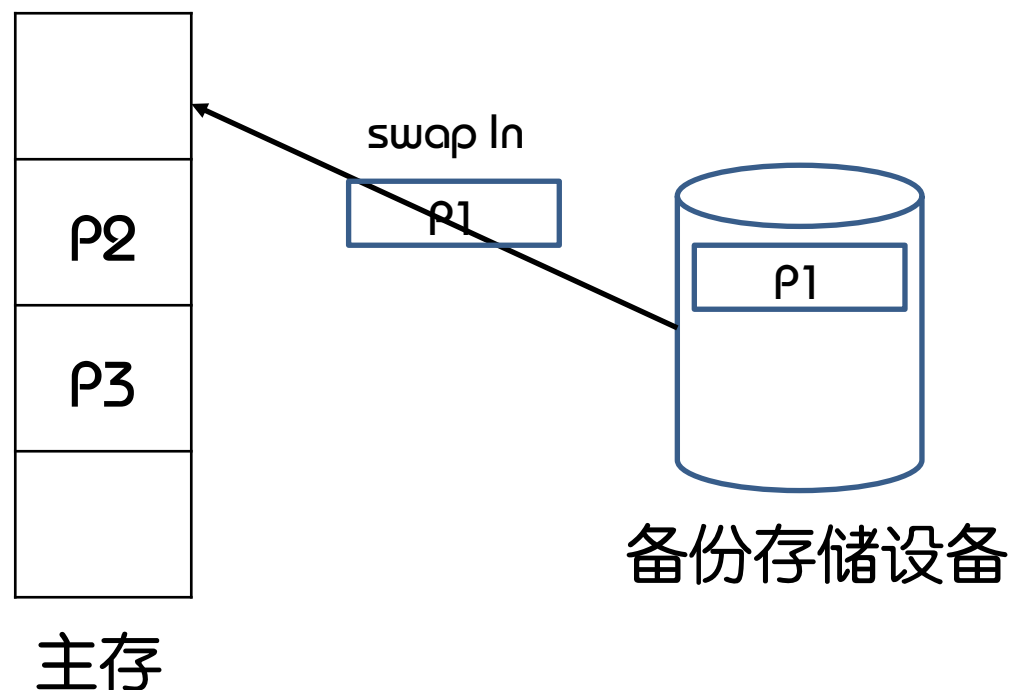




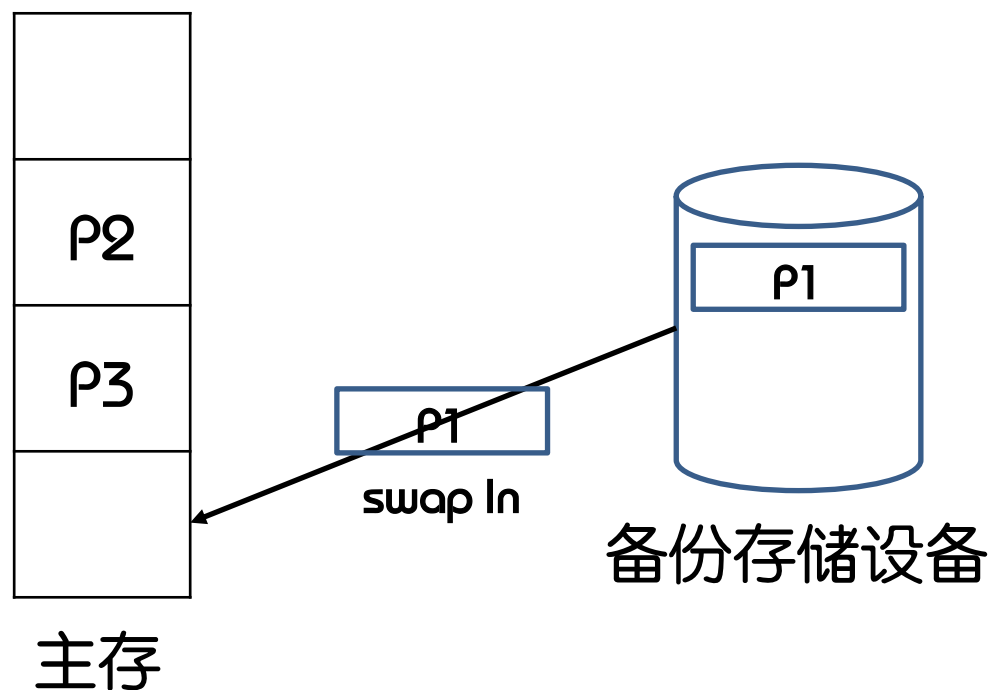
使用磁盘作为备份存储的两个进程的交换



必须回到原位，不可以移动到不同的位置（地址）  
即运行时不可以发生地址变化



可以移动到不同的地址  
即运行时，可以发生地址变化



## 第二节、内存分配方式

### 1. 连续分配

(1) 多分区分配

(2) 可变分区分配

### 2. 分页分配

### 3. 分段分配

每个进程位于连续的内存区域（多分区方法和可变分区方法）

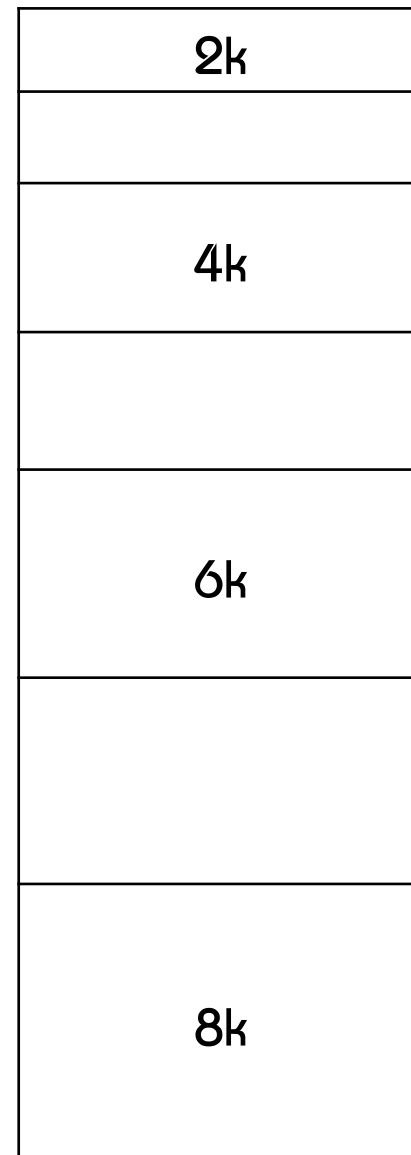
1. **多分区方法**:多个固定大小分区

划分区的方法

- a) **分区大小相同**:只适合于多个相同程序的并发执行，否则会缺乏灵活性
- b) **分区大小不同**:多个小分区，适量中分区，少量的大分区。根据程序大小，分配当前空闲的，适当大小的分区



相同大小



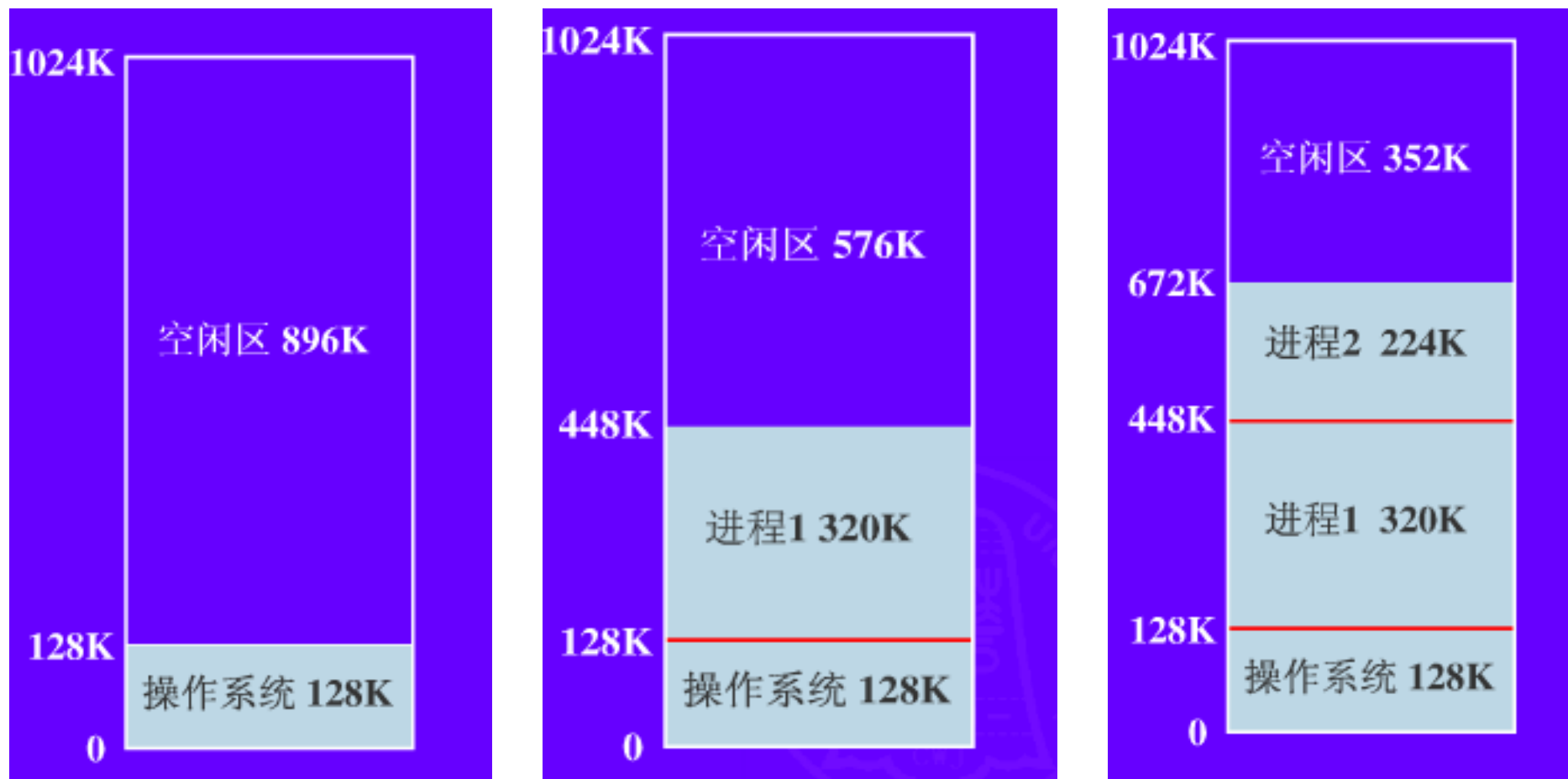
不同大小

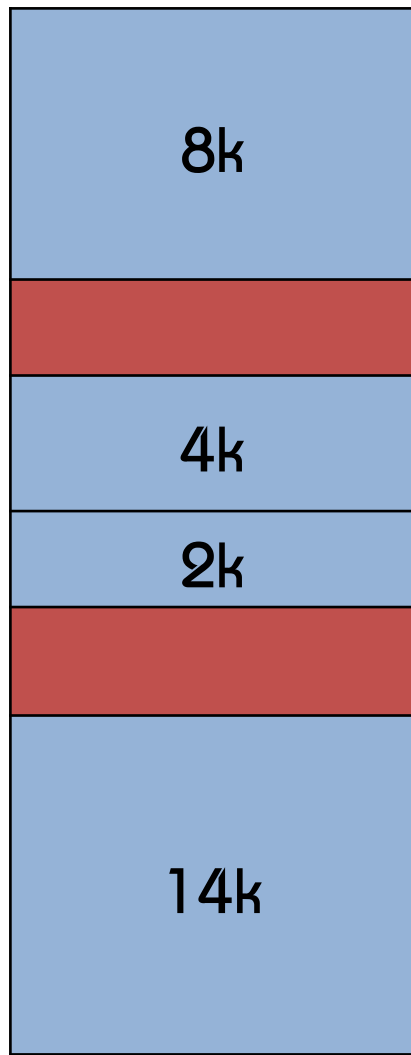


2. **可变分区方法**: 整个内存空间是就是一个大孔, 操作系统用表来记录已用和未用内存,

- 孔, 每当新的进程载入时, 分配足够大(?) 的孔
- 系统需要用表来记录已分配孔和未分配孔的信息

- I. **首次适应(first fit)**: 分配足够运行进程的第一个找到的孔
- II. **最佳适应(best fit)**: 分配足够运行进程的最小孔
- III. **最差适应(worst fit)**: 分配最大的孔

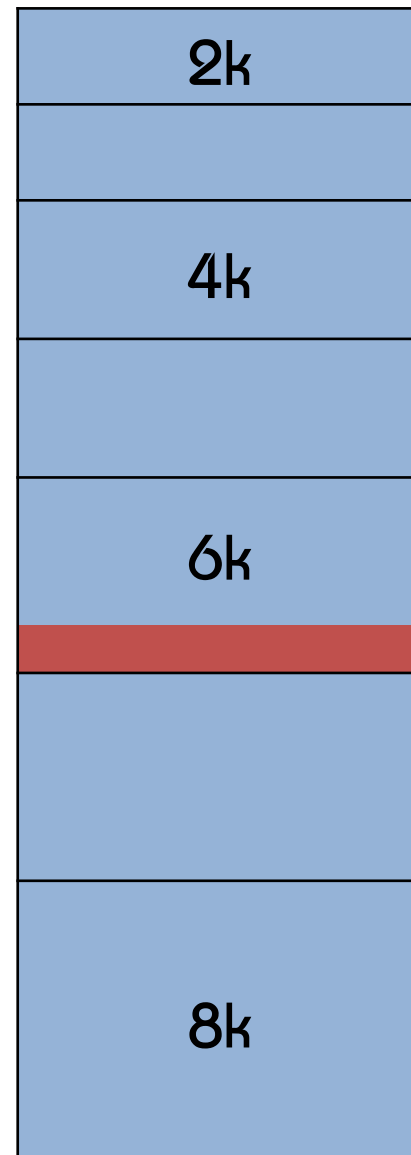




可变分区



多分区: 相同大小



多分区: 不同大小

产生碎片

## 1. 外部碎片(external)

- 区外产生的碎片。分区大小为2byte、2byte、2byte的不连续的空间，不能分配内存请求为6byte的进程

## 2. 内部碎片(internal)

- 区内产生的碎片。有18464byte大小的孔，要分配内存请求为18462byte, 会剩下2byte的孔，怎么办?

## 解决的方法(solution)

紧缩(compact):移动内存内容，把所有的空闲空间合并成一块

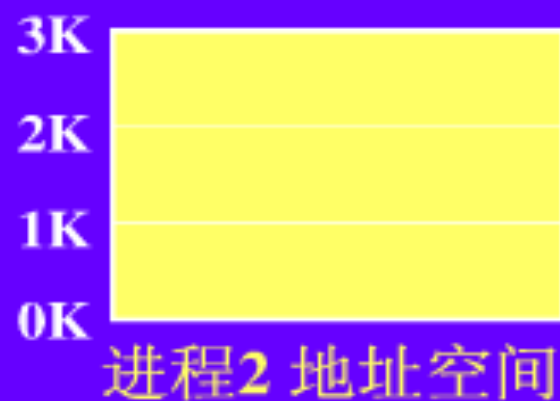
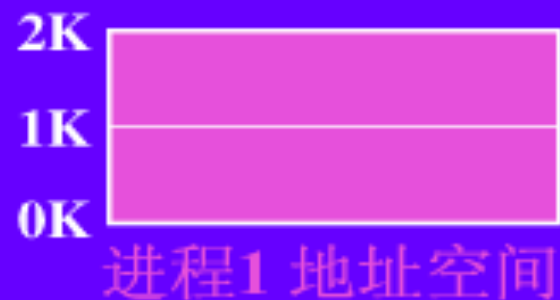
注:只适用于动态重定位的时候，不适用于静态定位，但开销大

要是不采用连续分配的方法，  
采用不连续分配方法？

- **帧**:把物理内存划分为**固定大小的块儿**，称为帧。
- **页**:把逻辑内存划分为**固定大小的块儿**，称为页。

一个页对应一个帧，当需要分配内存时，以页为单位分配内存。若程序大小为  $n$  页，则需要有  $n$  个帧来存放它，**而这些帧是不必连续的内存空间 (看下一页图)**

1. 需要掌握空闲空间(空闲帧)信息
2. 为了把逻辑地址转变为物理地址，需要设置一个页表



逻辑地址分为两个部分:页号和页偏移

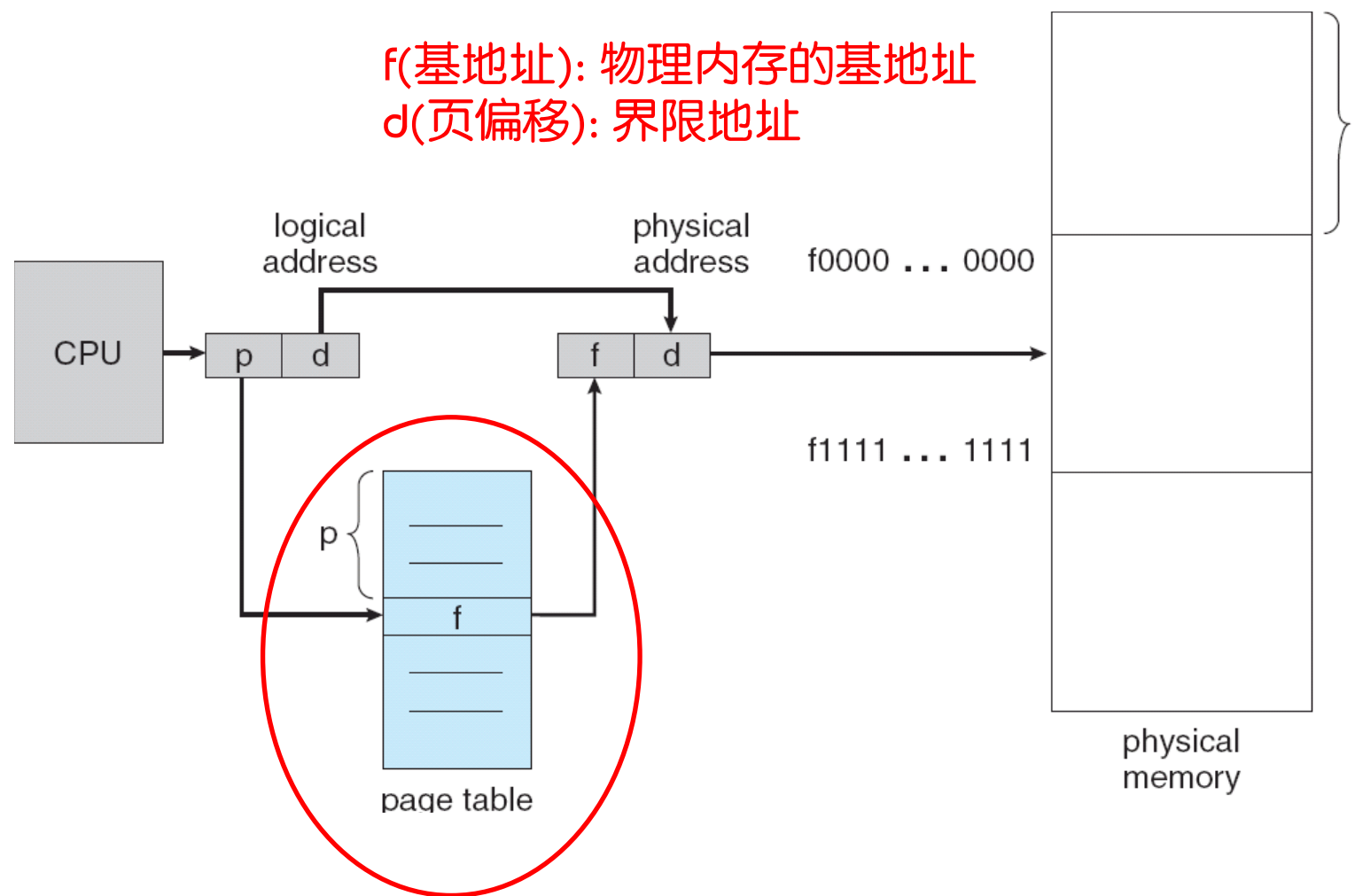
1. **页号( $p$ )**: 是页表的索引, 相应页表中存储着该页所在物理内存的基地址
2. **页偏移( $d$ )**: 结合基地址形成物理地址



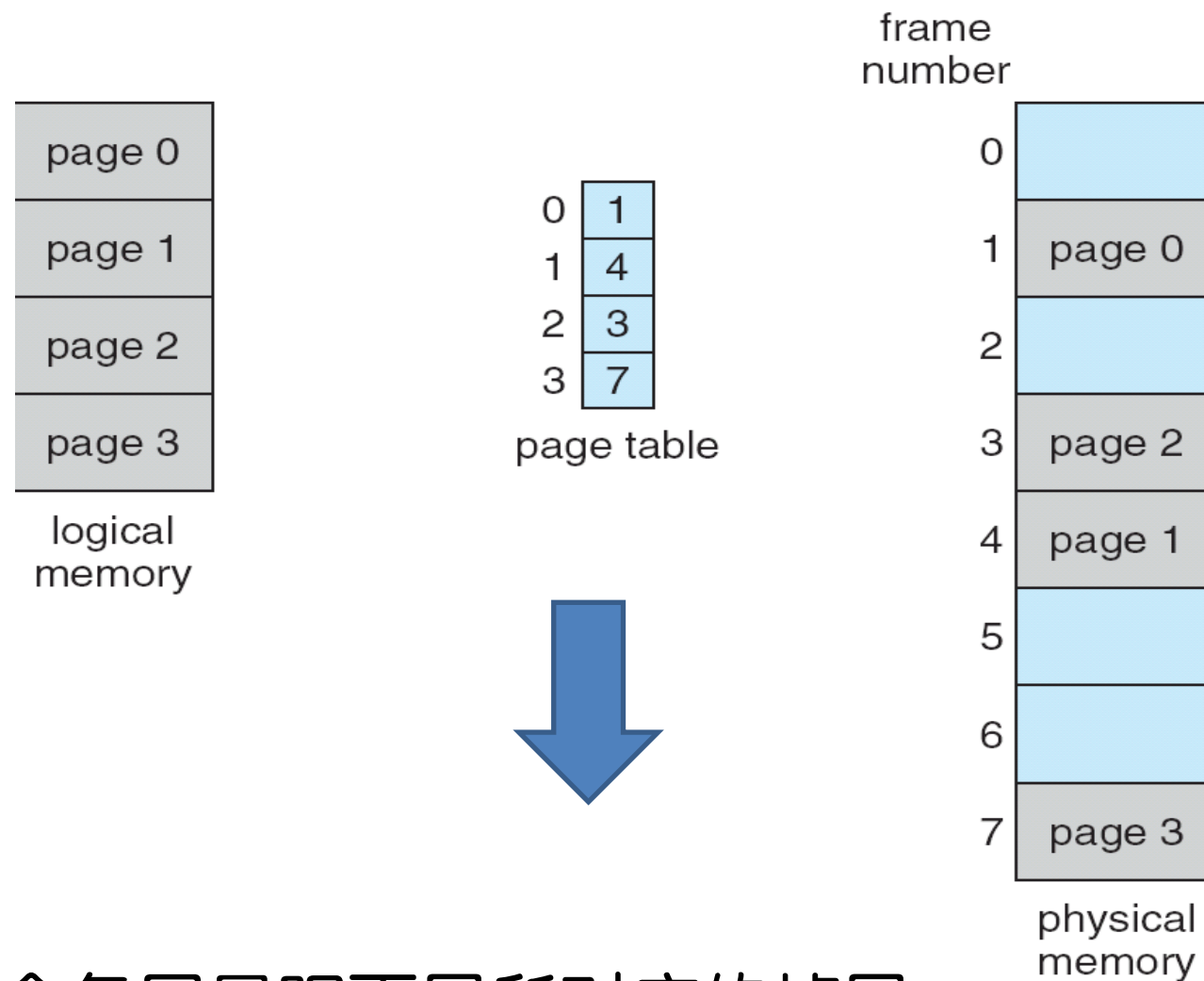
20位页号可指向 $2^{20}$ 个帧, 如每个帧大小 4k

那么, 32位逻辑地址可覆盖的内存范围是  $2^{20} \times 2^{12} = 2^{32} = 4GB$

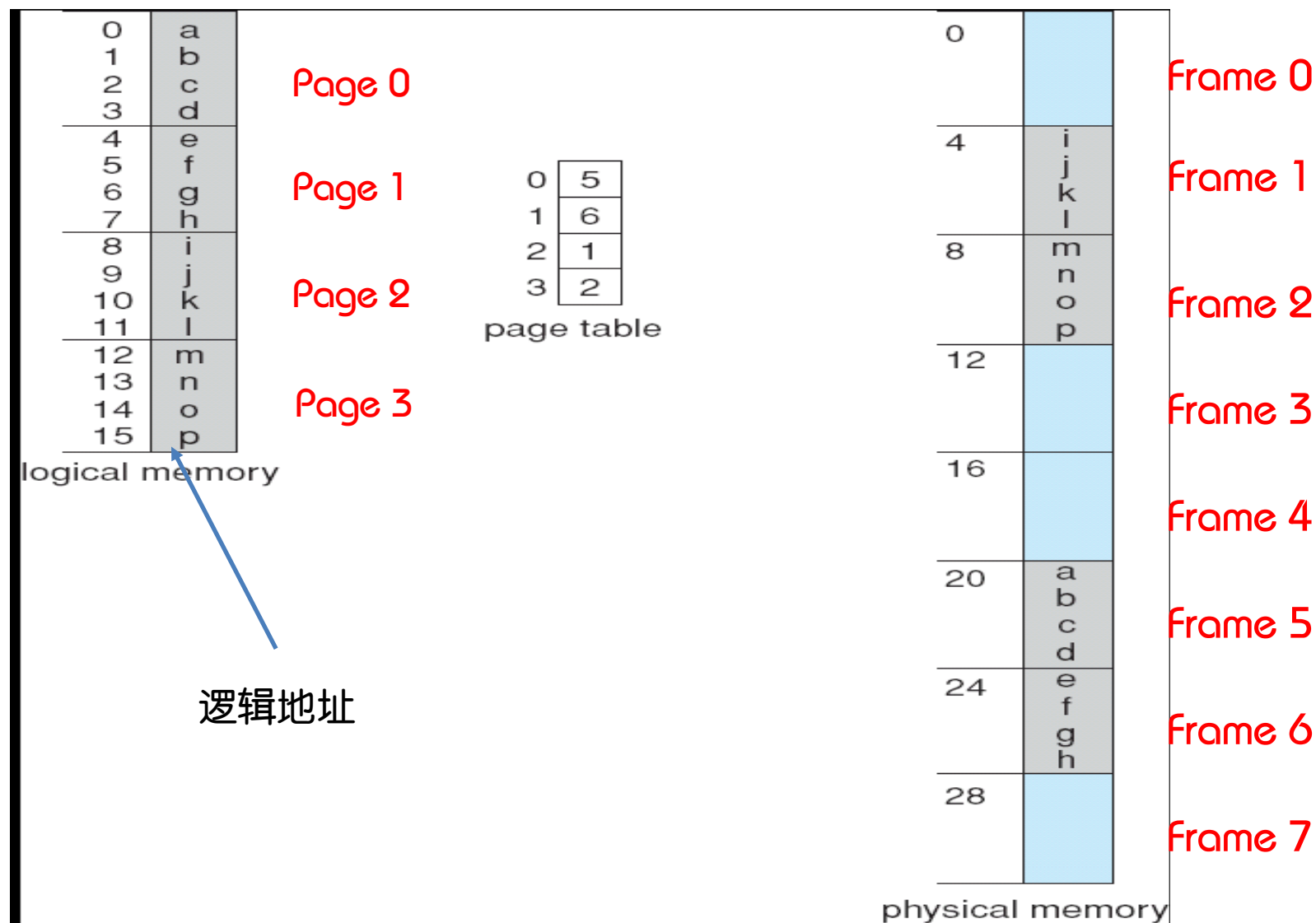


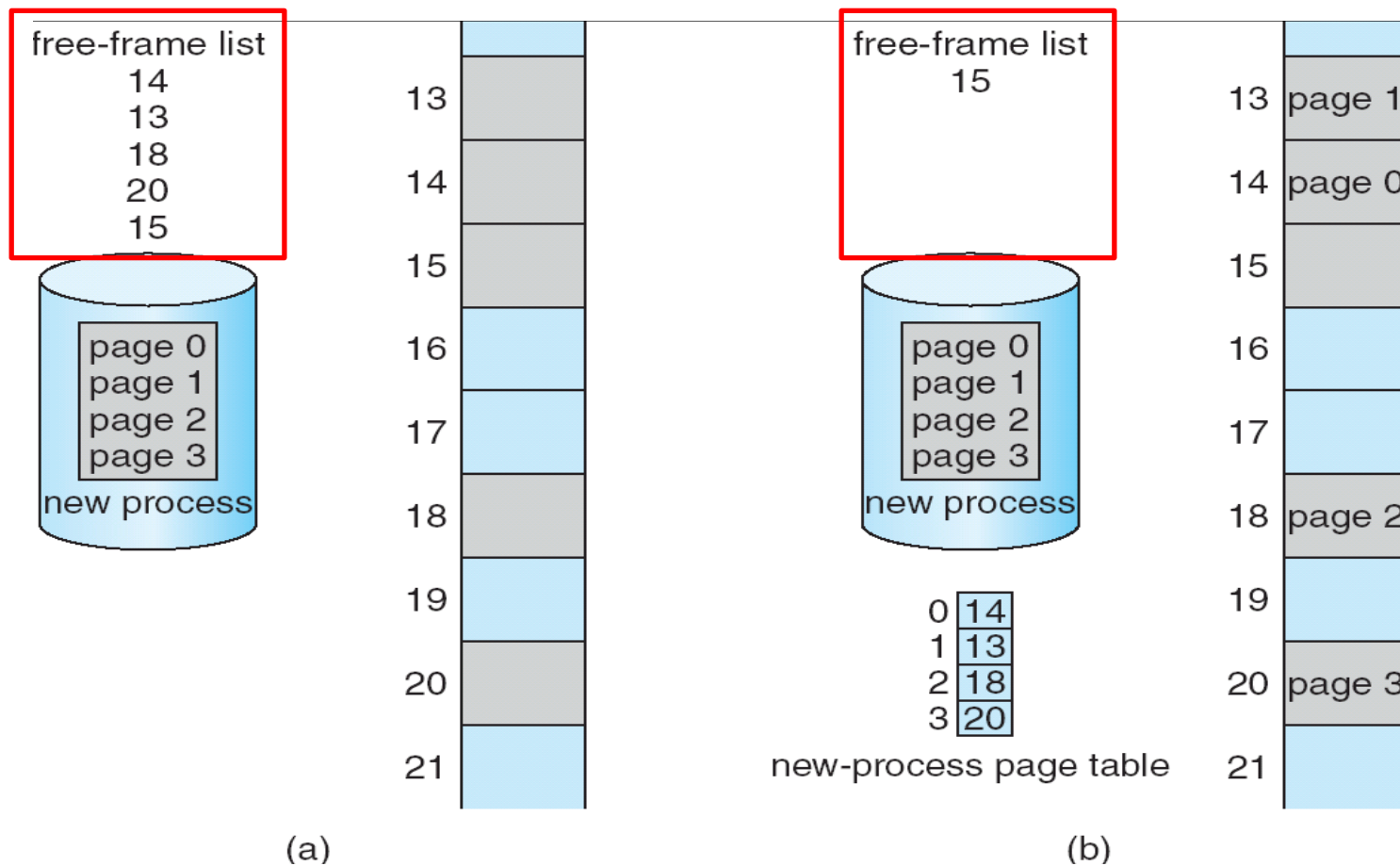


页表每个条目是跟页号所对应的帧号



页表每个条目是跟页号所对应的帧号





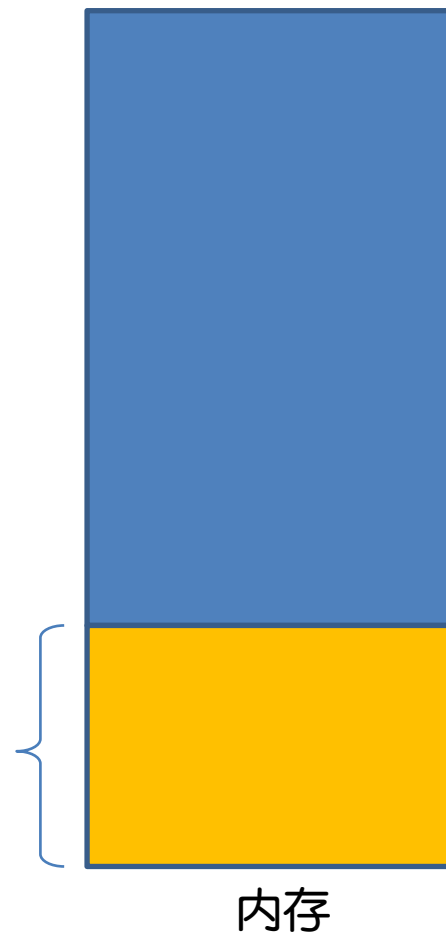
分配前

分配后

1. 页表可以用一组专用寄存器来保存,
2. 也可以将页表放入到内存中;  
(这需要页表基寄存器和需要页表长寄存器)

若把页表存入内存时,  
可能会出现的问题是什么?

存储页表信息,  
至少需要两个寄存器



为了访问数据，需要两次访问内存

- 第一次查页表，第二次访问地址

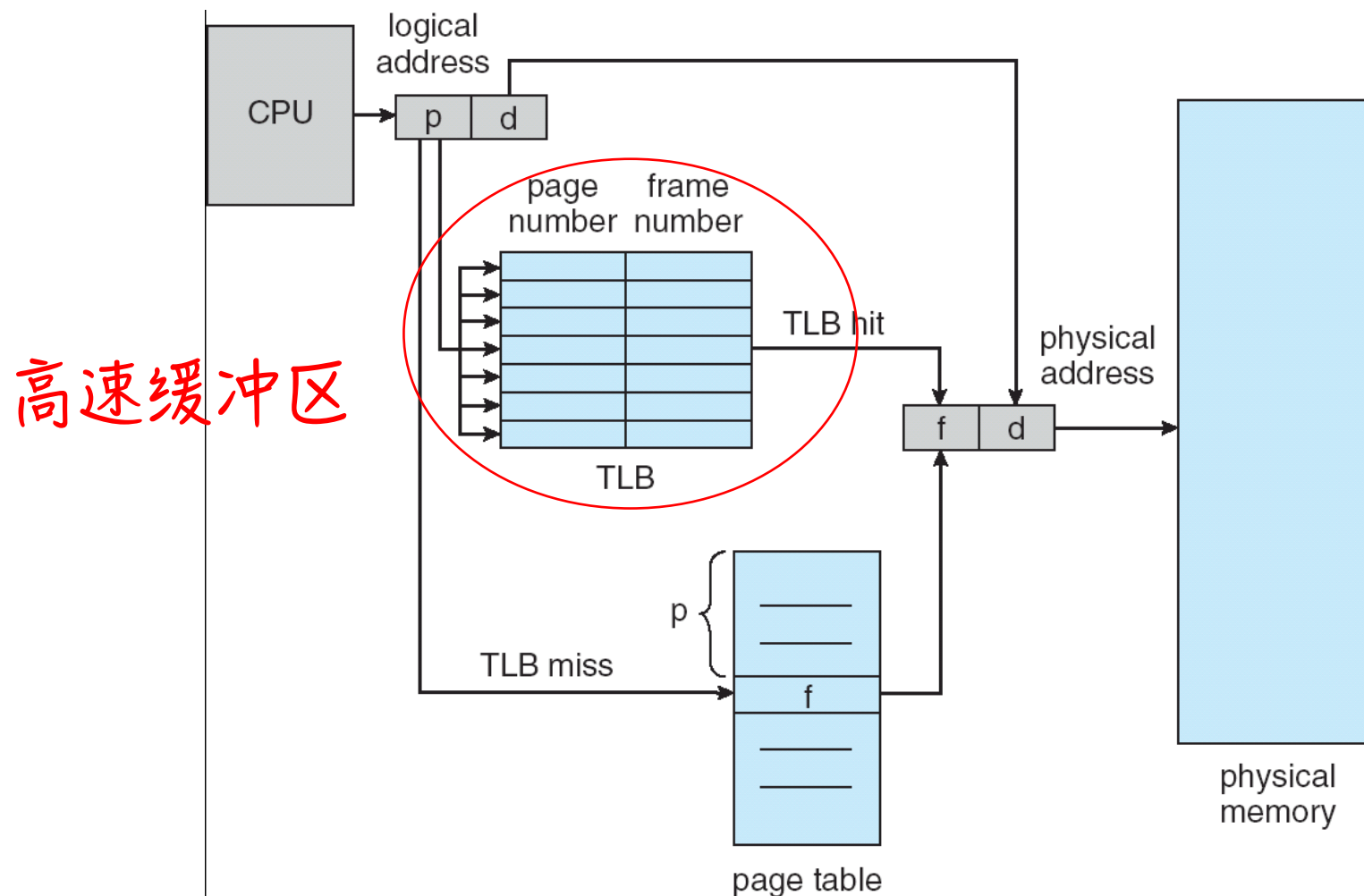
需要两次访问内存，这会导致数据访问速度慢

解决的办法

硬件缓冲，称为地址转换旁观缓冲

(Translation Look-aside Buffer: TLB)

—键（标签），值

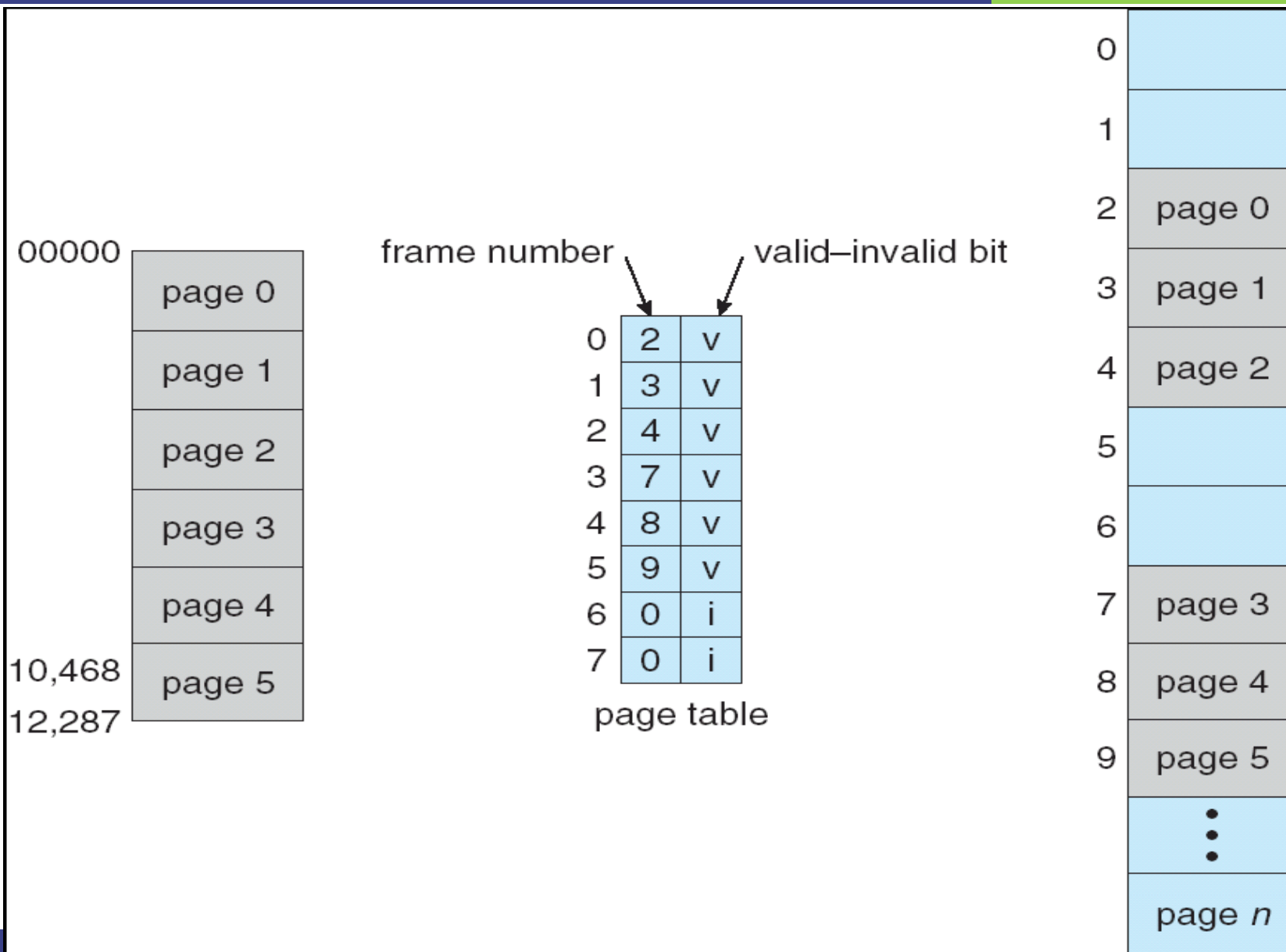


问:在分页环境下， 怎么实现内存的保护?

答:与每个帧关联一个的**保护位**，建立与页表中的每一条目相关联的有效/无效位。

1. 有效位 (v) 表示相关的页在进程的逻辑地址空间内 (合法)
2. 无效位 (i) 表示相关的页不在进程的逻辑地址空间内 (非法)





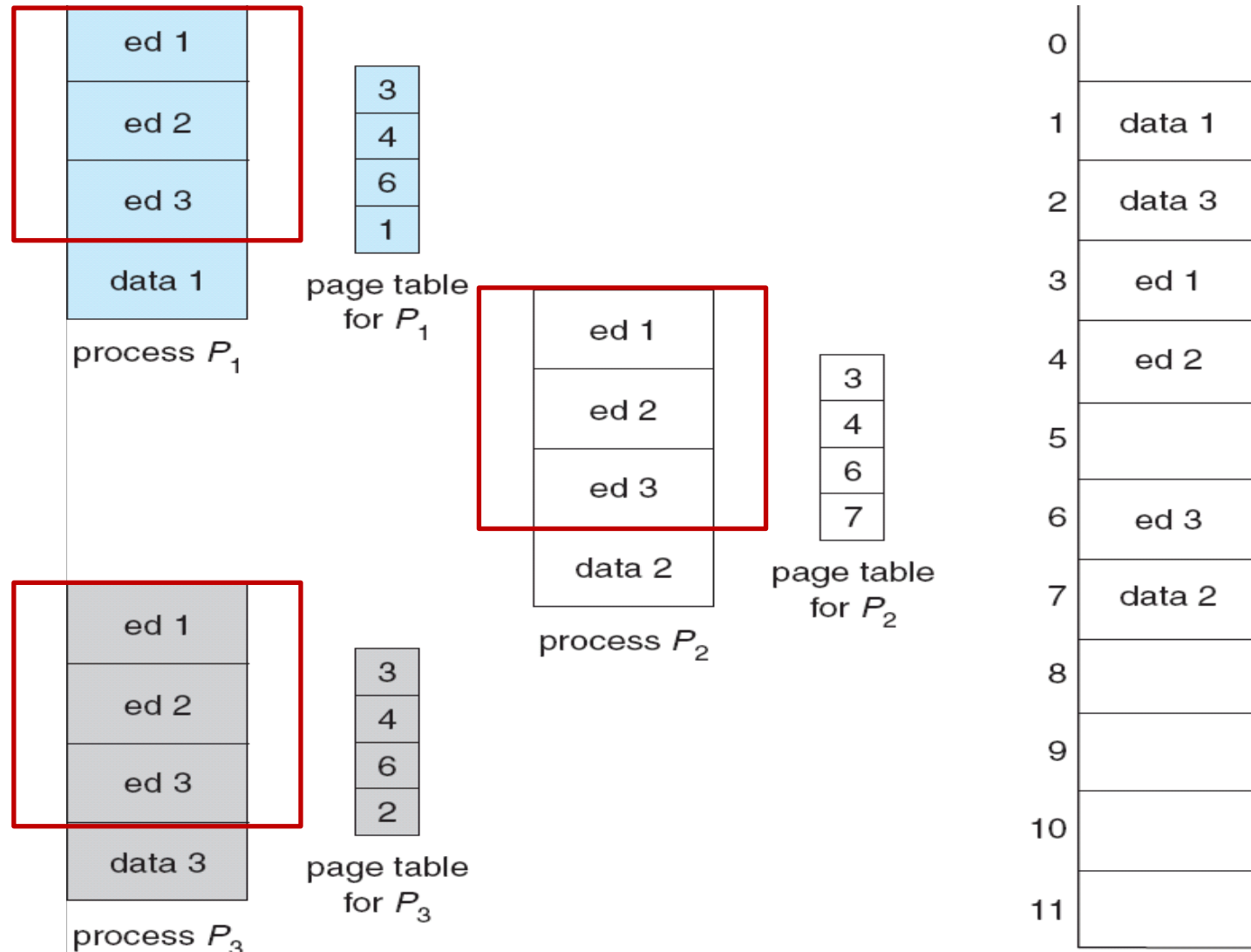
通过共享页实现代码共享。进程的代码可分为可以共享的代码和不可共享的代码，可共享的代码通过共享页表的方式实现。

### (1) 可共享的代码 - 可重入代码

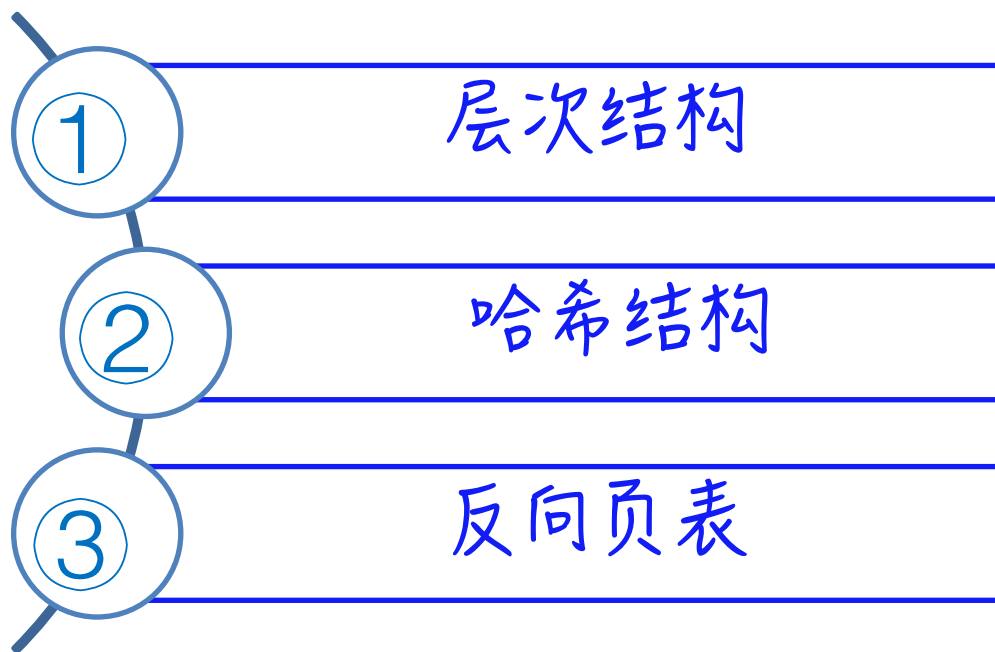
- 每个进程共享代码段的(页)逻辑地址相同

### (2) 不可共享的代码 - 私有代码和数据

- 针对私有代码和数据的(页)逻辑地址不同

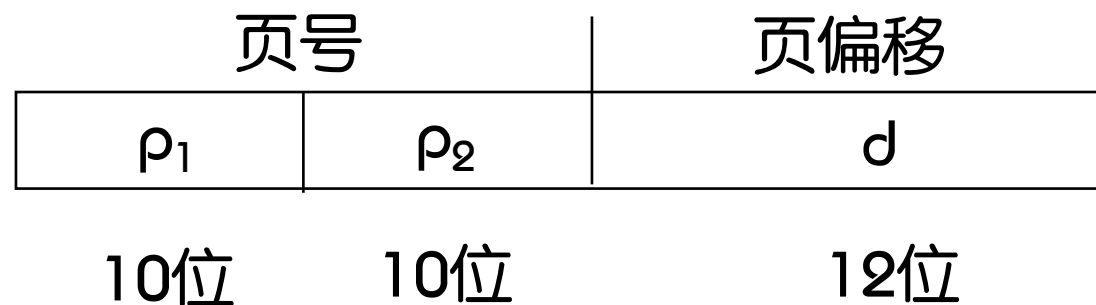


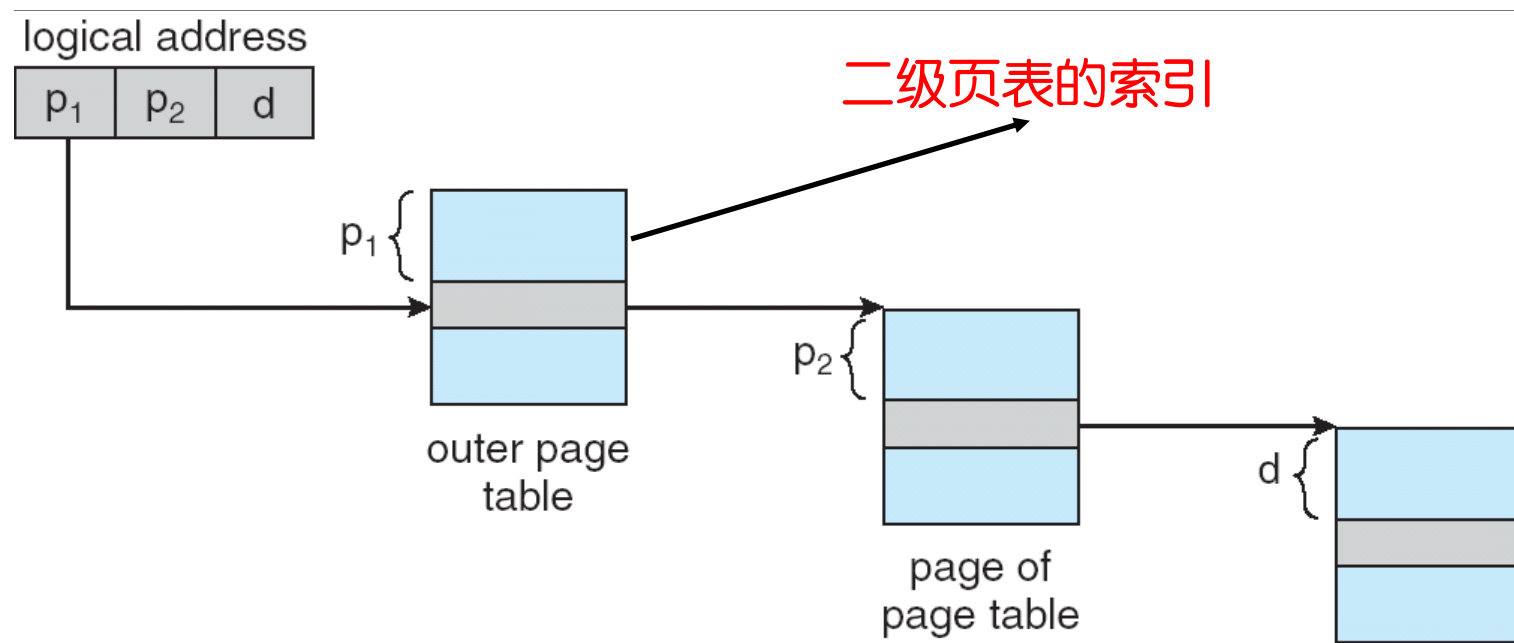
- 把逻辑地址映射为物理地址的过程中，分页方式需要系统维护页表信息
- 在实现页表方式上，有以下几种方式



把逻辑地址编址设为多层次，即多层次的页表。

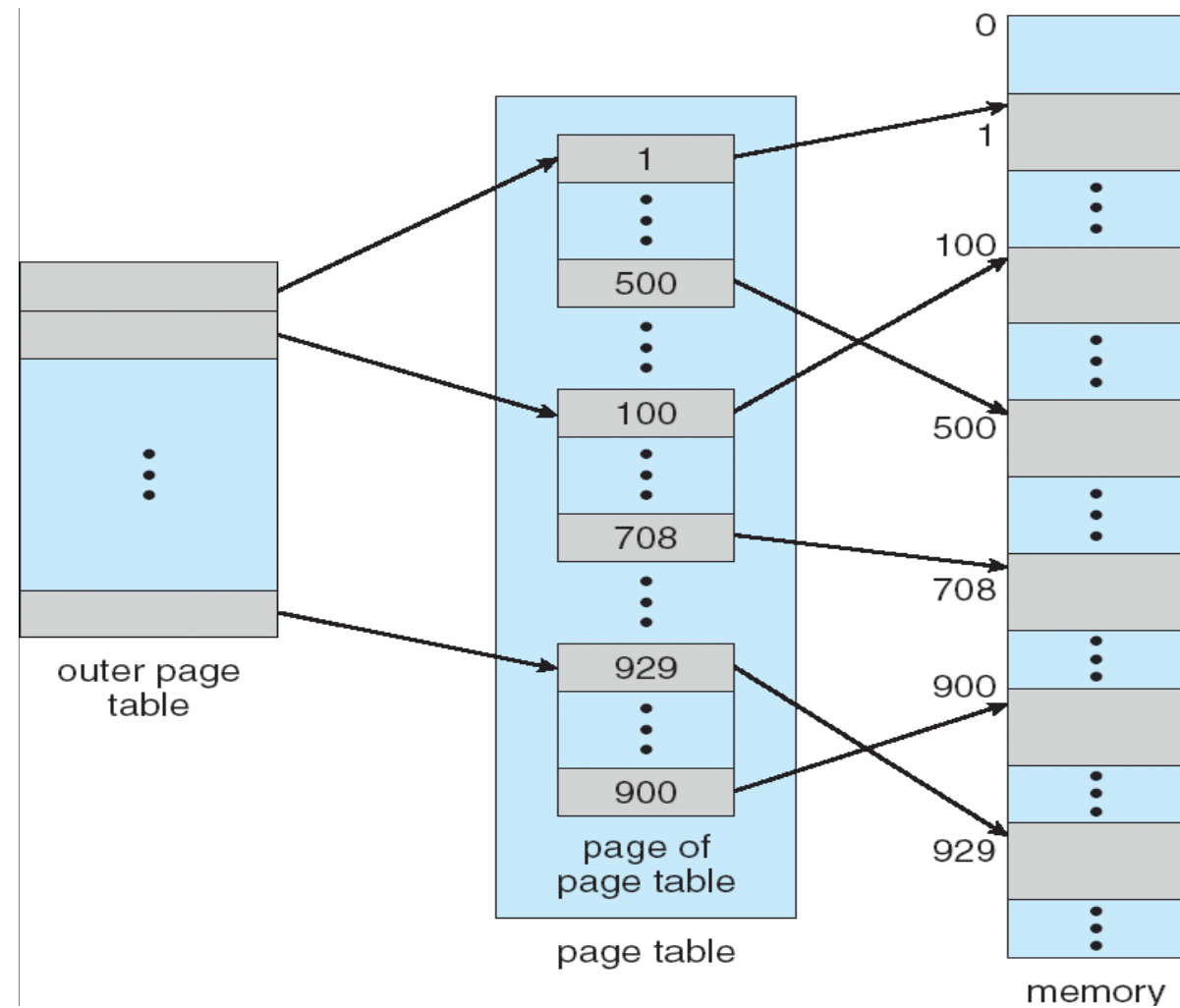
比如，32位系统的二级页表结构，逻辑地址分为20位页号和10位页偏移，在把20位页号分为两个10位页号





采用层次结构页表存储结构有什么有点?

1. 可以实现页表的不连续存储
2. 可以节约内存空间（问:为什么?）



假设,

- 页表是单层次结构
- 内存空间大小为  $2^{32} = 4\text{Gb}$
- 页大小为  $2^{12} = 4\text{k}$
- 每个页表条目大小为 4byte

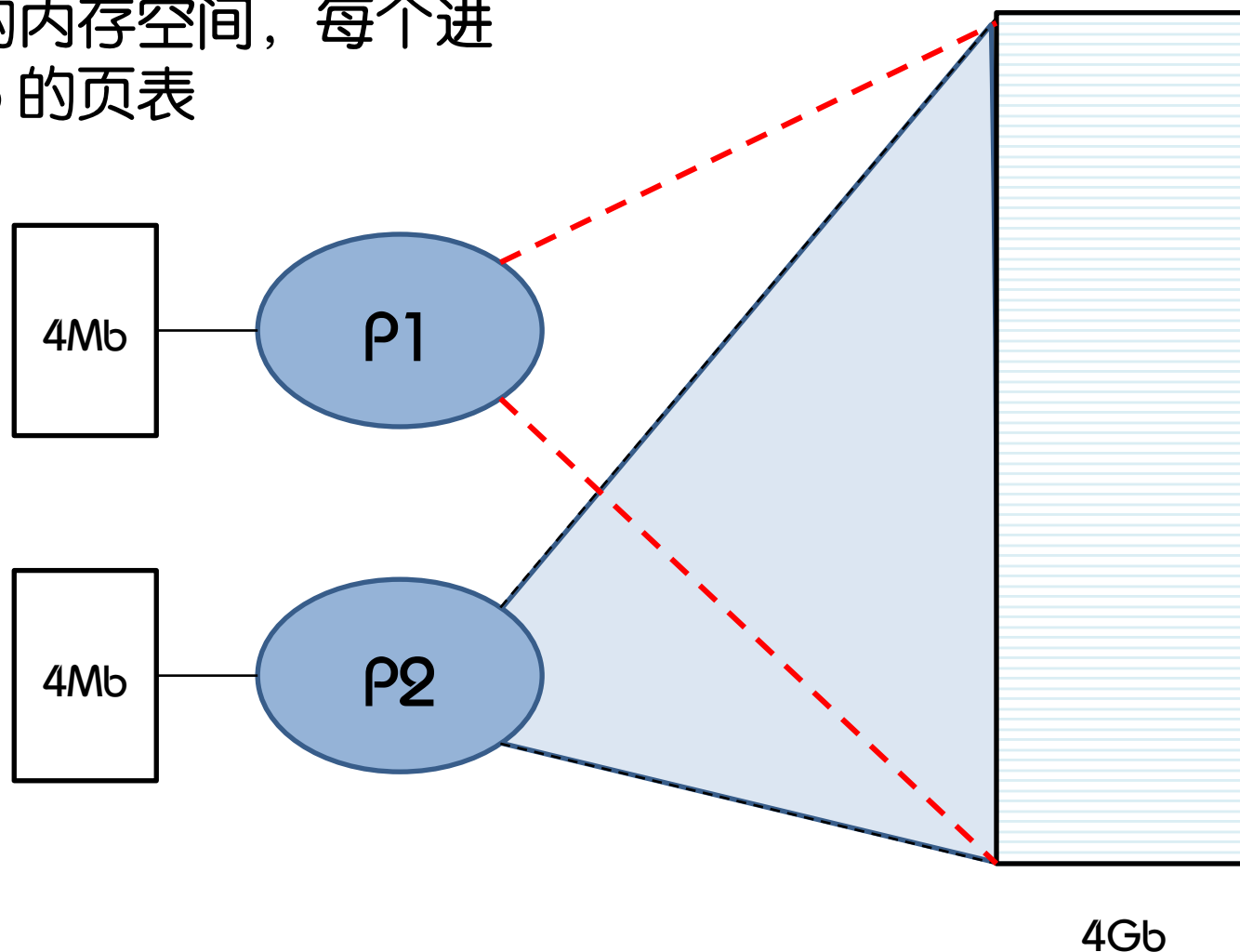
页号	页偏移
p	d
20 bits	12 bits

那么, 我们为覆盖4Gb的内存, 每个进程要维持 4Mb的页表

1. 页表条目的数量  $= 2^{32}/2^{12}=2^{20}$
2. 需要的页表大小  $2^{20} \times 4\text{byte}$
3.  $= 4\text{Mb}$

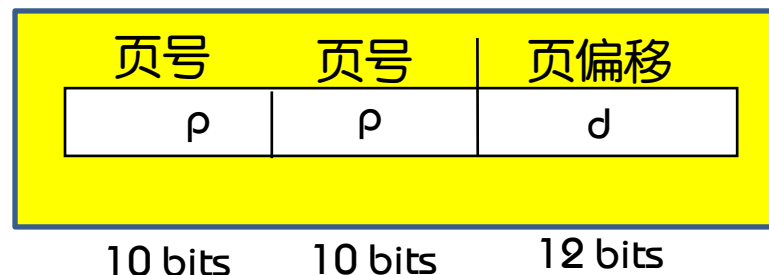


为了覆盖4Gb 的内存空间，每个进程必须持有4Mb 的页表



假设,

- 页表是两层结构
- 内存空间大小为  $2^{32} = 4\text{Gb}$
- 页大小为  $2^{12} = 4\text{k}$
- 每个页表条目大小为 4byte



Homework:

那么, 我们为覆盖4Gb的内存, 进程要维护的页表大小是多少?

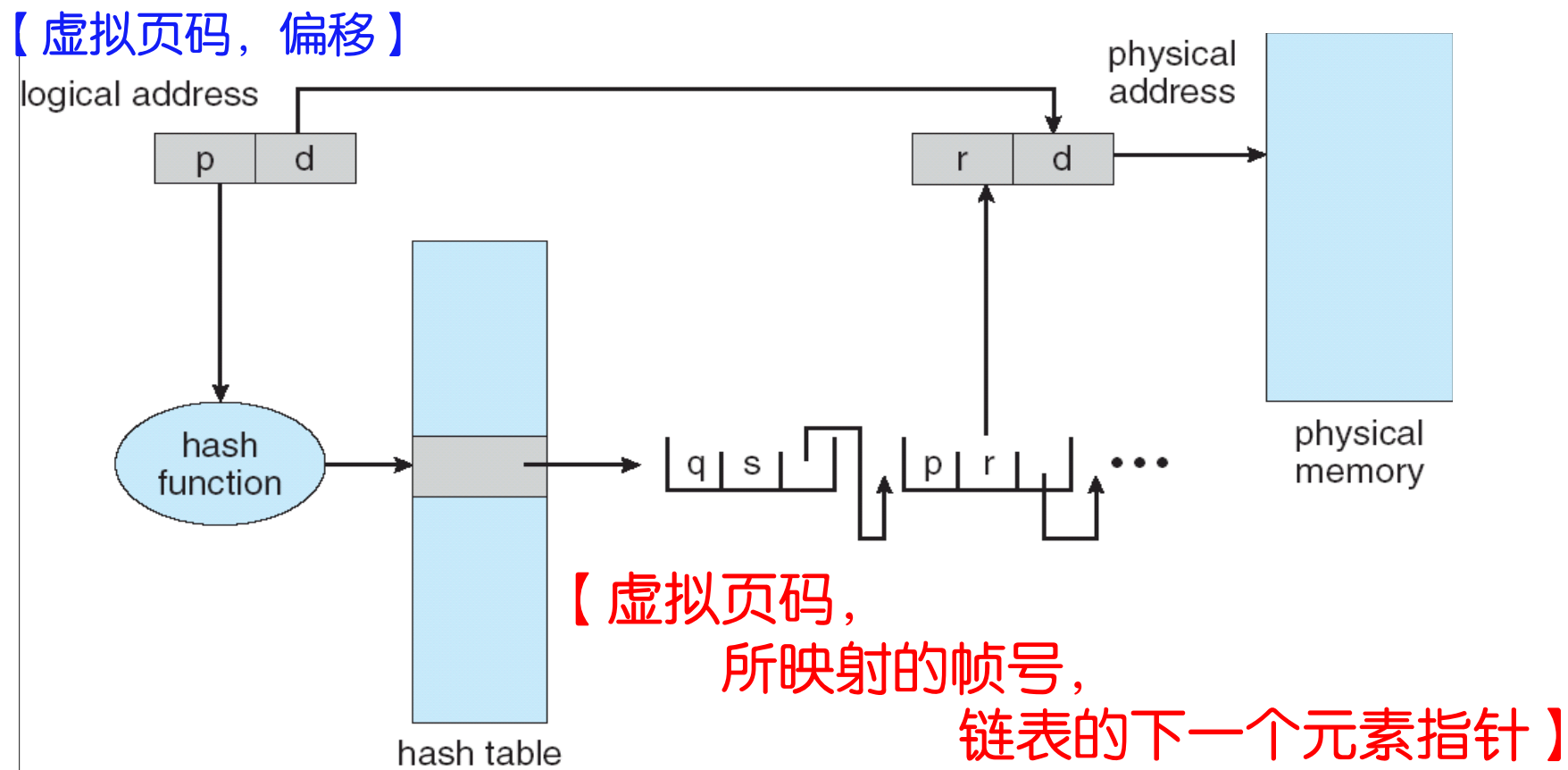
- $2^{10} \times 4\text{byte} + 2^{10} \times 4\text{byte} = 8\text{k}$

outer page	inner page	offset
$p_1$	$p_2$	$d$
42	10	12

2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

- 处理超过32位地址空间的常用方法
- 逻辑地址的定义为【虚拟页码，偏移量】
  - 把虚拟页码作为哈希值(key)
- 哈希页表的每一条目是链表，链表的每个元素包括【虚拟页码，所映射的帧号，下一个元素指针】
- 根据虚拟页码（哈希值）找到匹配的条目，并在链表里找到相对应的帧号

通过页号转换到哈希表中，用链表指针找到想对应的帧号，并用帧号和页偏移形成物理地址



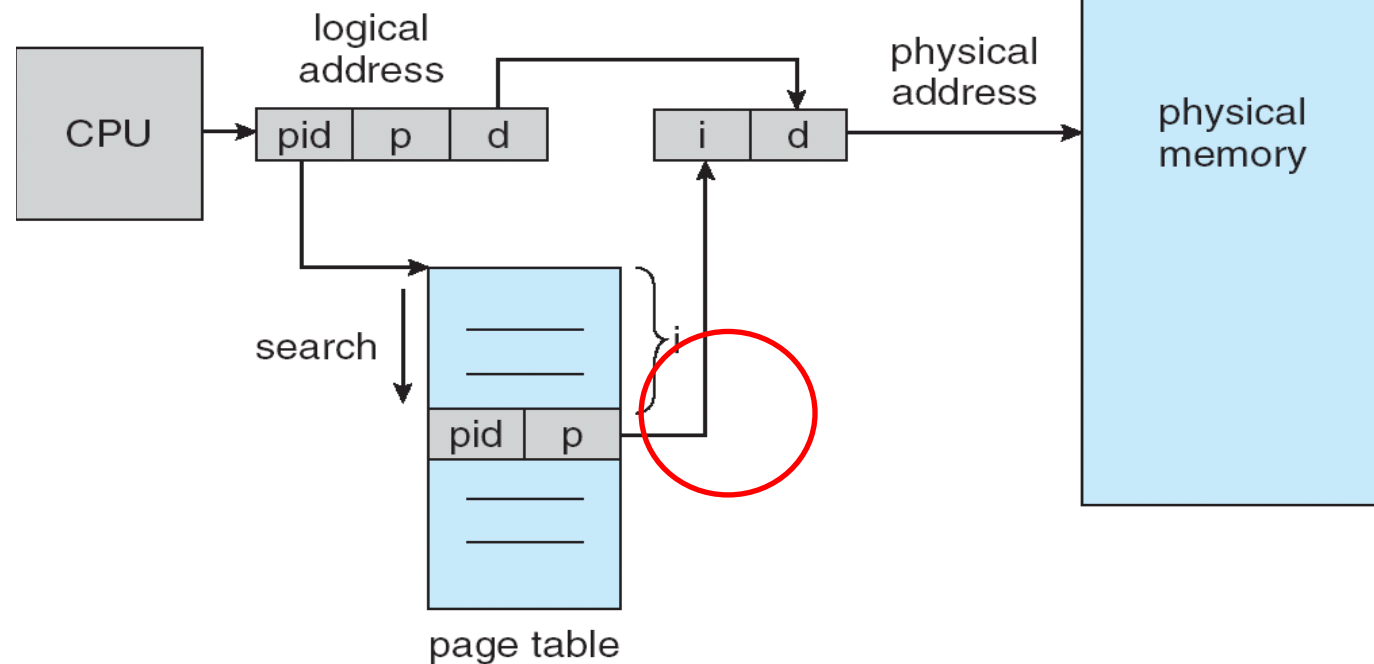
为了减少页表消耗的内存空间而采用的方法

即，整个系统只有一个页表，页表对于每个真正的内存页或帧才建立一个条目

- 地址信息:需要用页码和页偏移来确定空间
- 进程信息:需要用进程标识符确定进程信息
- 反向页表编址:

【进程标识符，页码，页偏移】

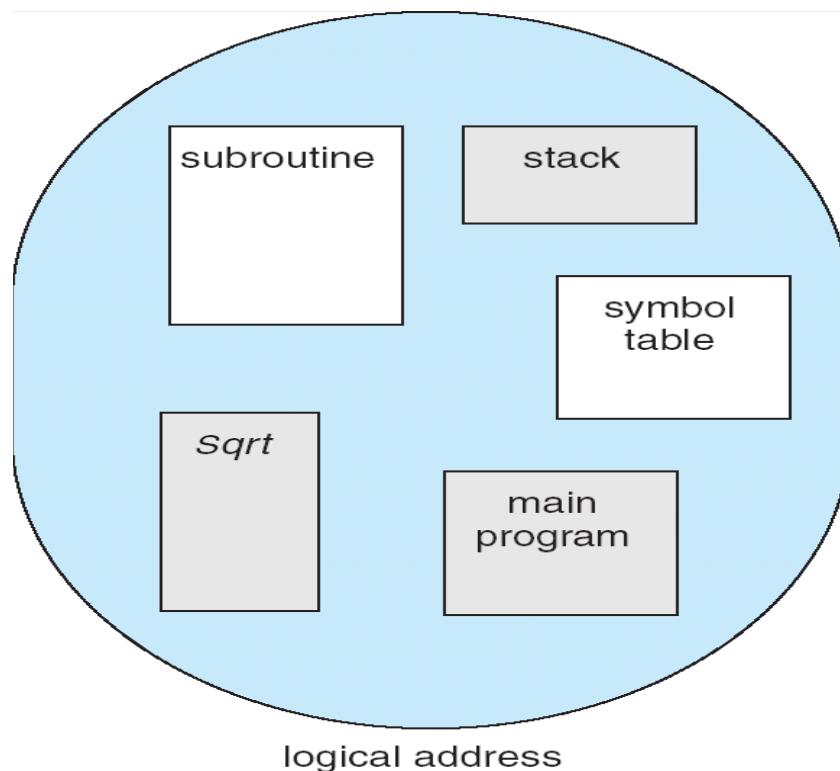
【进程标识符，页码，页偏移】



问题是查找页表，费时

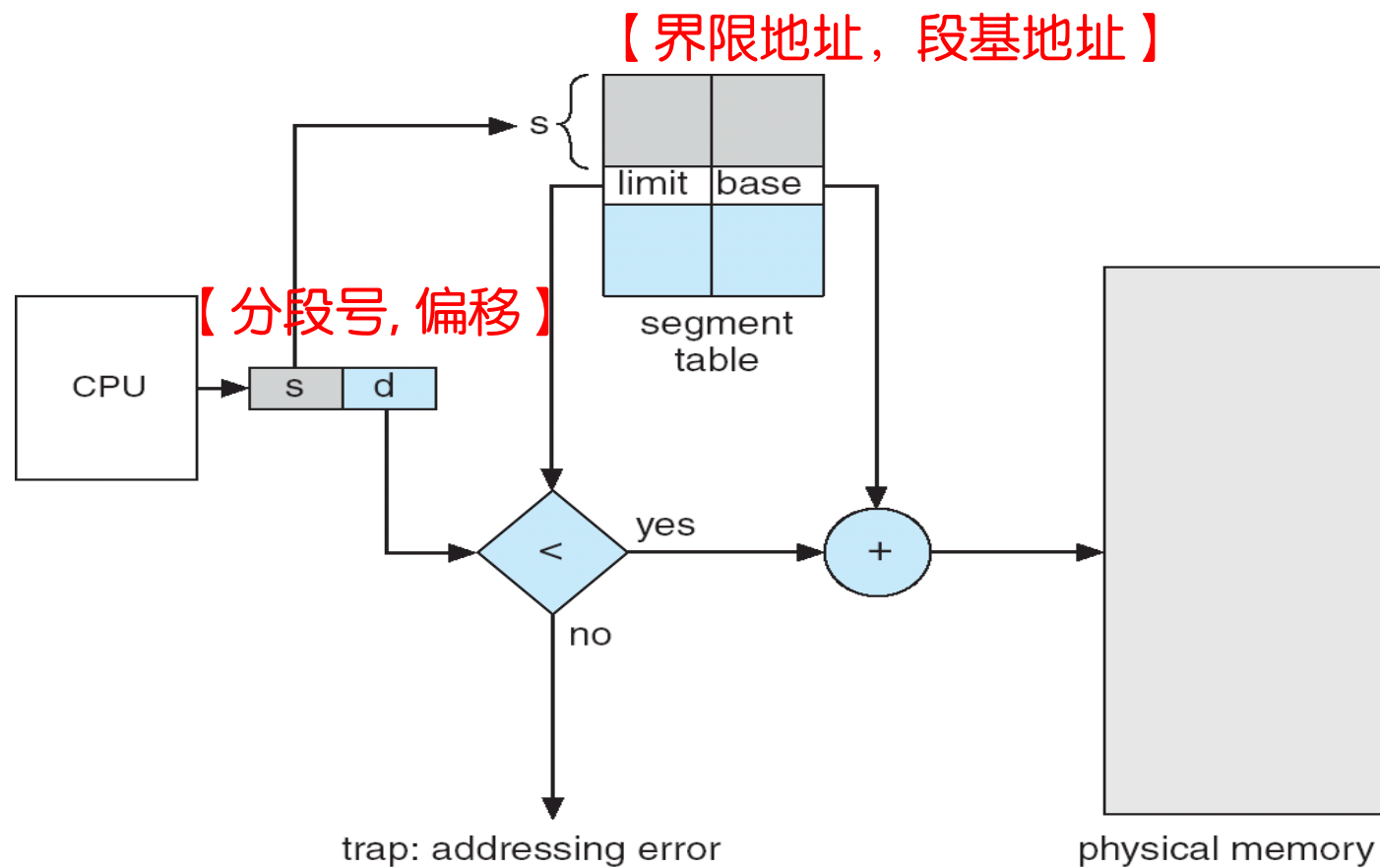
以用户视角去管理内存的方式，一个程序由多个逻辑段组成：  
一般组成:主程序、函数、方法、对象、堆、栈、局部变量、全局变量等

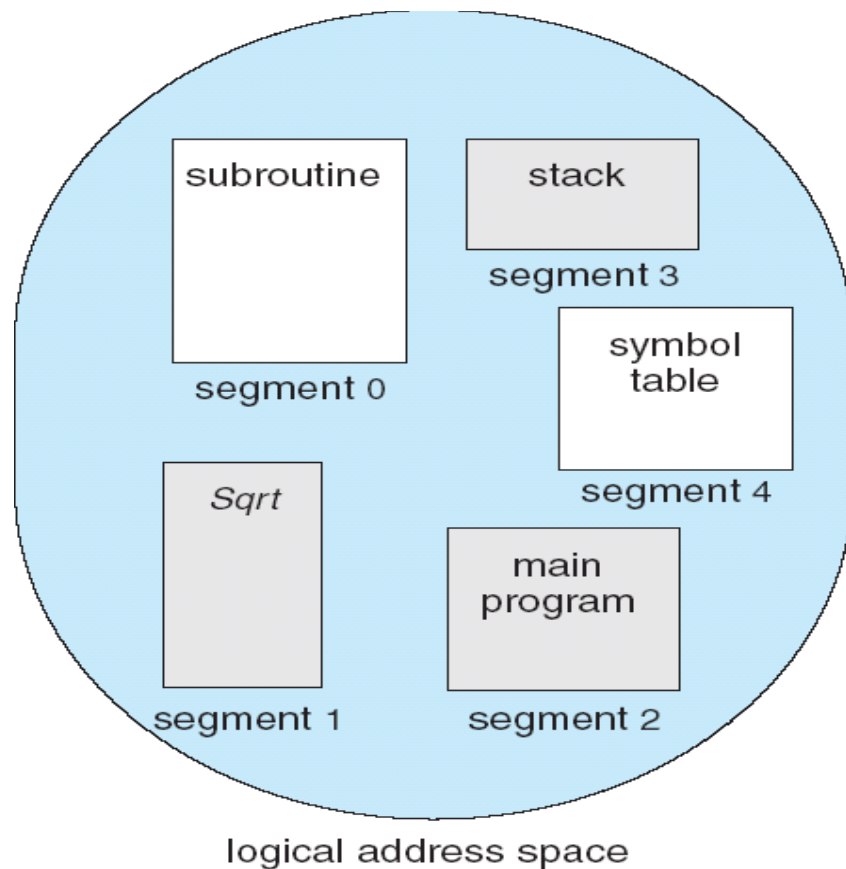
用户眼中的  
内存空间





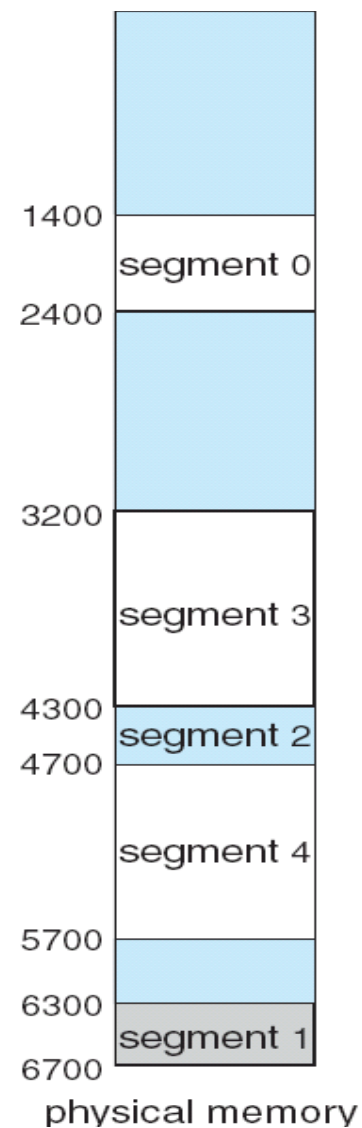
- 逻辑地址的定义:  
【分段号, 偏移】
- 需要段表, 段表条目包括
  - 段基地址
  - 段界限





	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



# Q&A