上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L8. 内存管理

宋卓然

上海交通大学计算机系

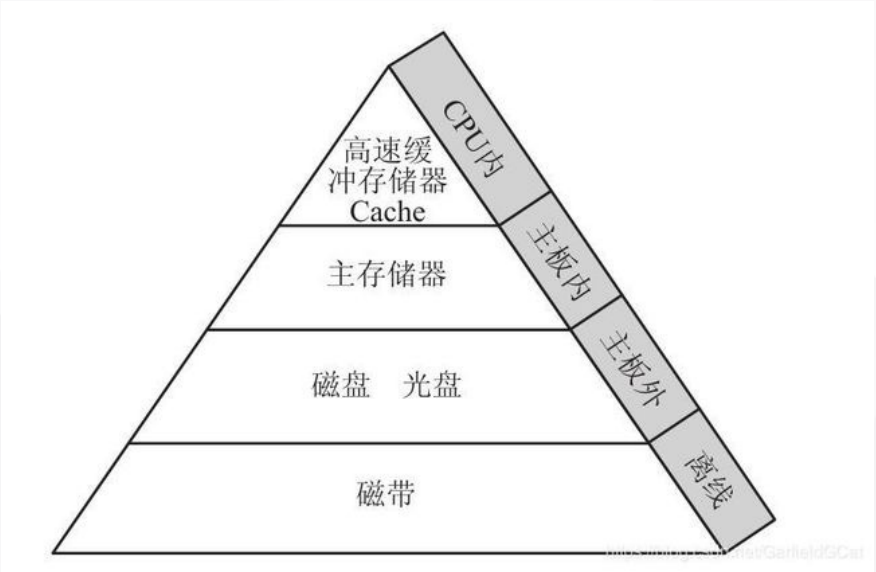
songzhuoran@sjtu.edu.cn

背景

- 程序必须（从磁盘）导入内存并放置在进程中才能运行
  - 取指-译码-执行-访存-写回
- CPU 可以直接访问的通用存储只有内存和处理器内置的寄存器
- 内存单元只能看到地址流，而不知道这些地址如何产生（由指令计数器PC、索引、间接寻址、常量地址等）或它们是什么（指令或数据）的地址

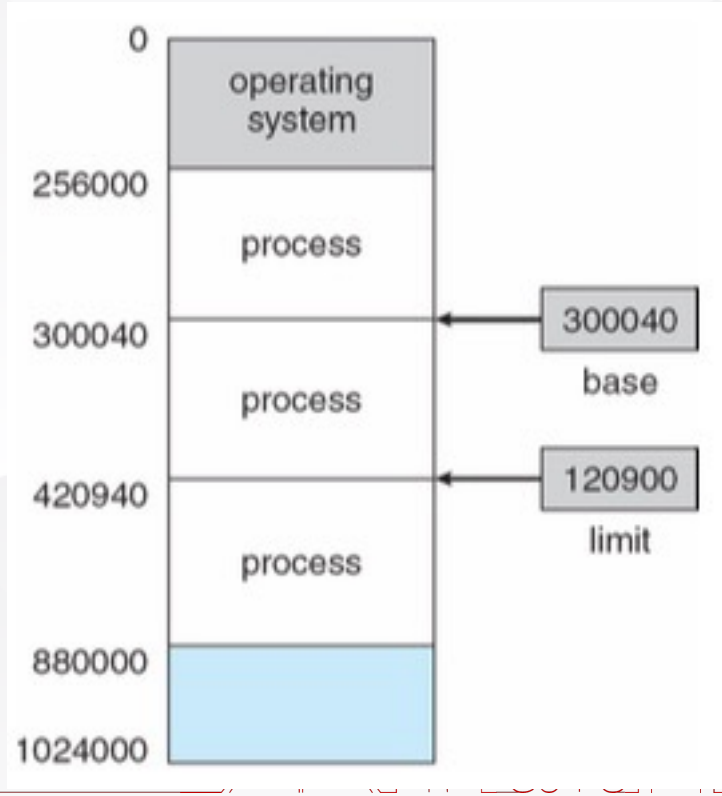
背景

- 寄存器可以在一个 CPU时钟周期内被访问，而内存则需要多个时钟周期才能被访问
  - 金字塔结构
  - 寄存器-Cache-主存-磁盘-磁带（从快到慢、容量从小到大）



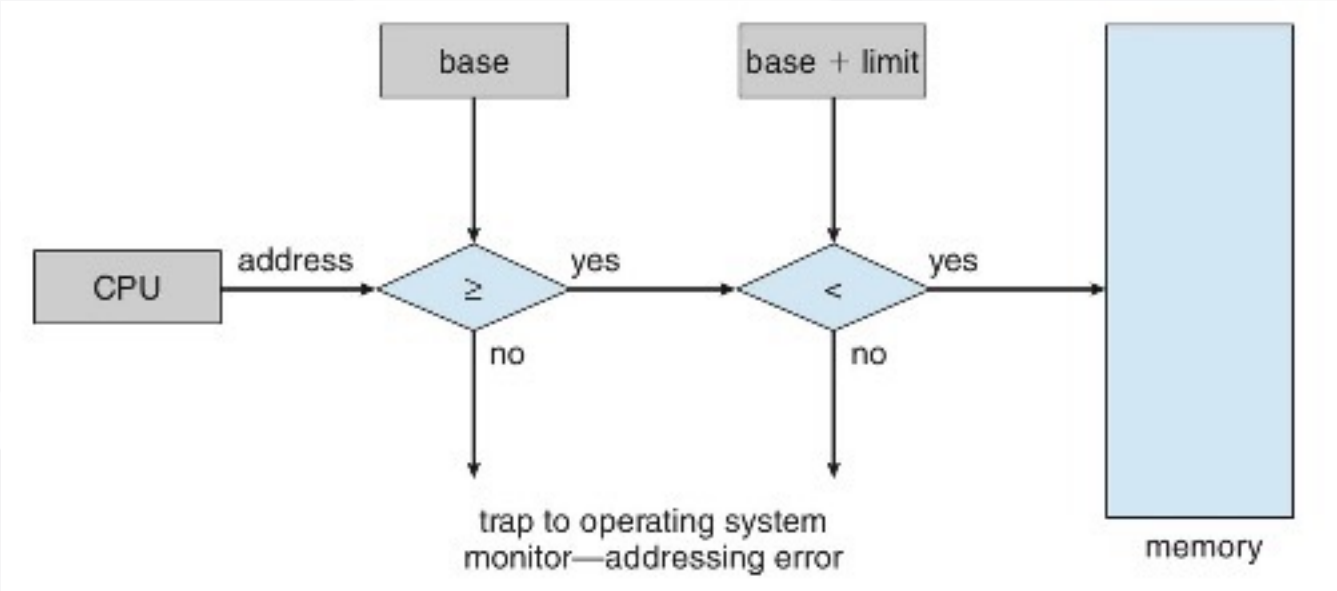
基地址与界限地址寄存器

- 为了保证用户进程之间不会互相影响，需要确保每个进程都有一个单独的内存空间
- 两个寄存器
  - 基地址寄存器：含有最小的合法的物理内存地址
  - 界限地址寄存器：指定了范围的大小



内存空间保护

- 内存空间保护的实现是通过CPU硬件对在用户模式下产生的地址与寄存器的地址进行比较来完成的
  - 当用户试图访问操作系统内存或其他用户内存，则系统将其视作致命错误来处理



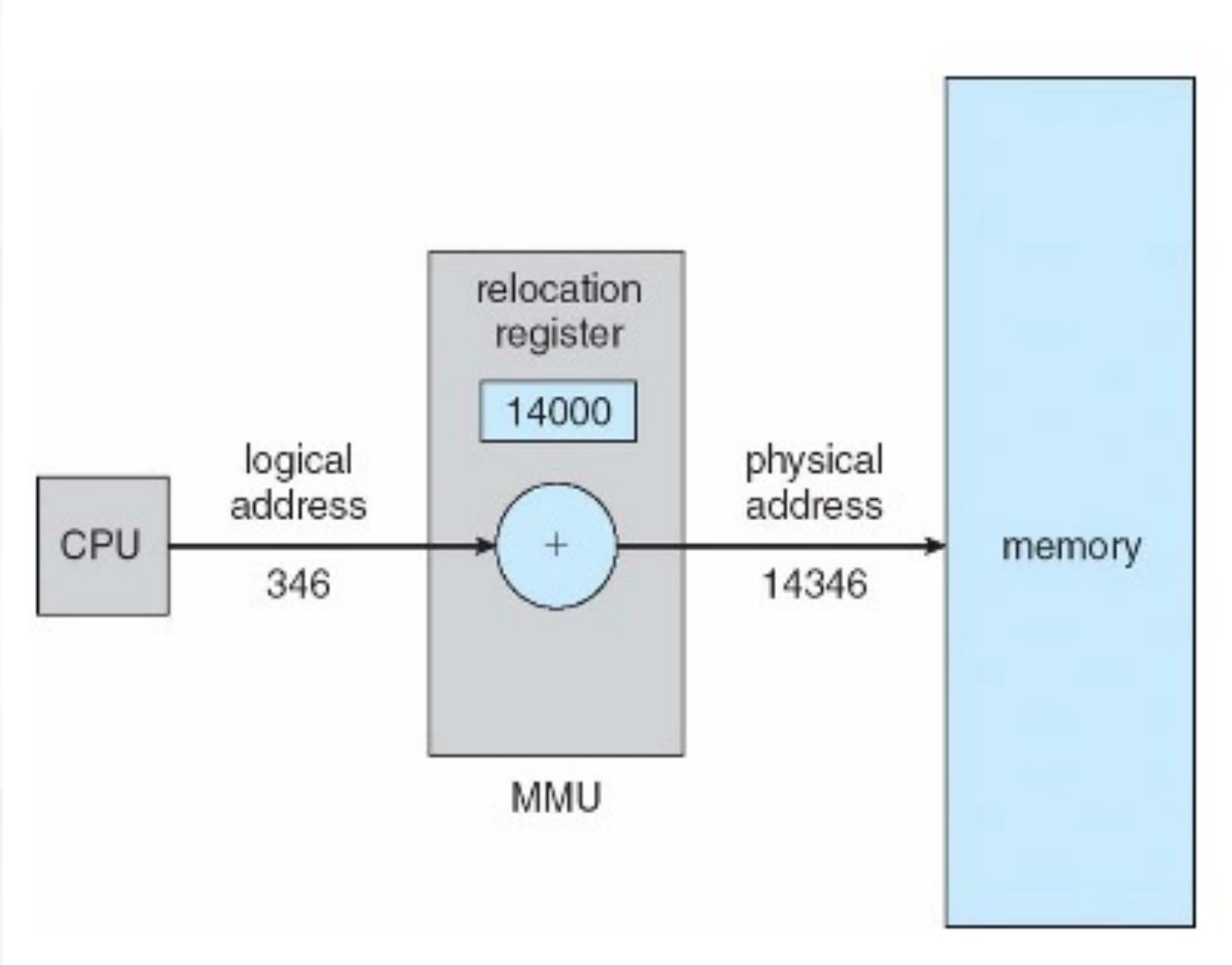
逻辑 vs 物理地址空间

- CPU生成的地址通常称为逻辑地址
- 内存单元看到的地址通常称为物理地址
- 由程序生成的所有逻辑地址的集合称为逻辑地址空间
- 物理地址空间是这些逻辑地址对应的物理地址的集合

内存管理单元

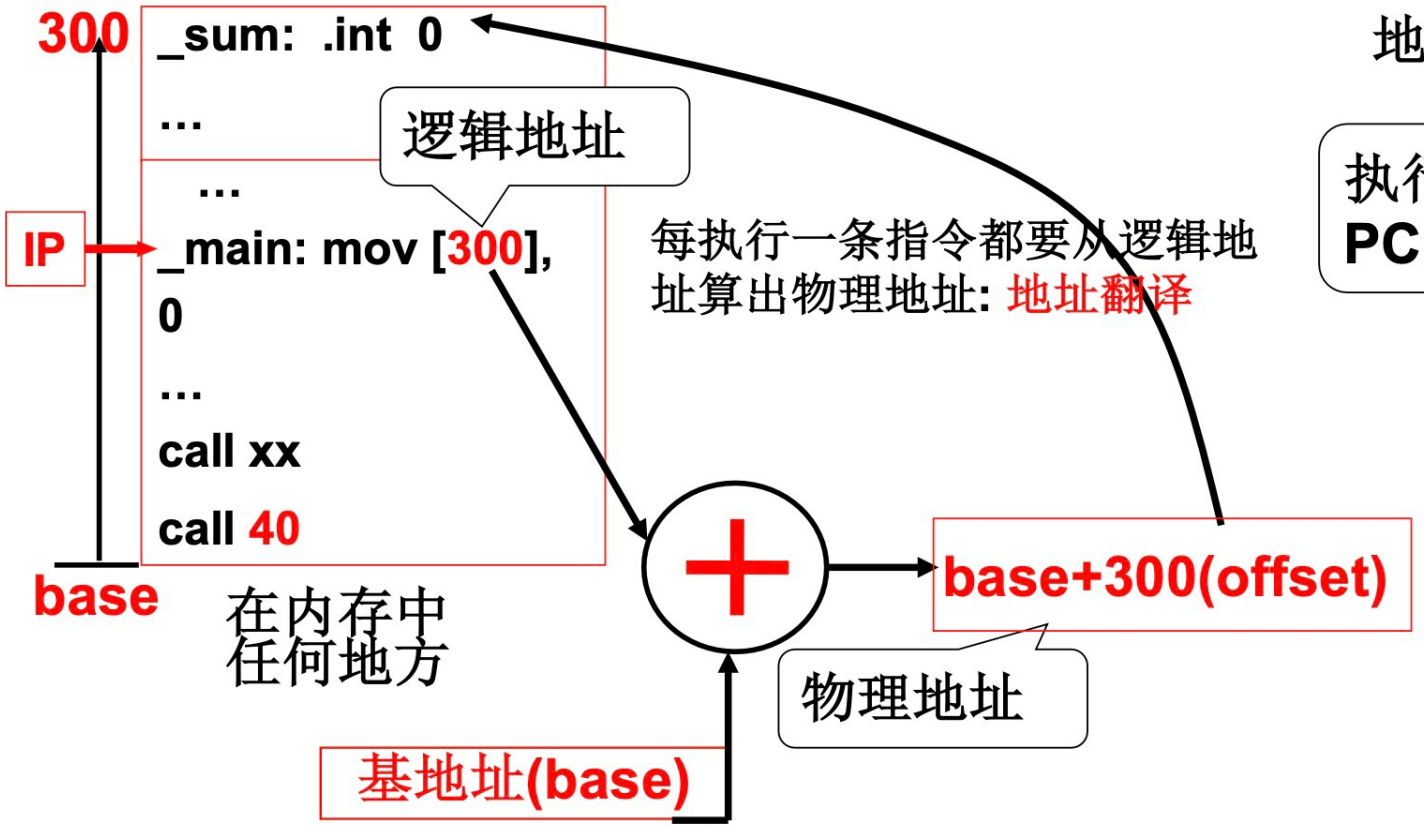
- 从虚拟地址到物理地址的运行时映射是由内存管理单元（MMU）的硬件设备来完成
- 有多种映射方法
- 最简单的MMU方案：重定位寄存器。将用户进程所生成的地址在交给内存前，加上重定位寄存器的值
- 用户程序不会看到真实物理地址

内存管理单元


relocation
register
14000
CPU
logical
address
346
+
physical
address
14346
memory
MMU

运行时重定位

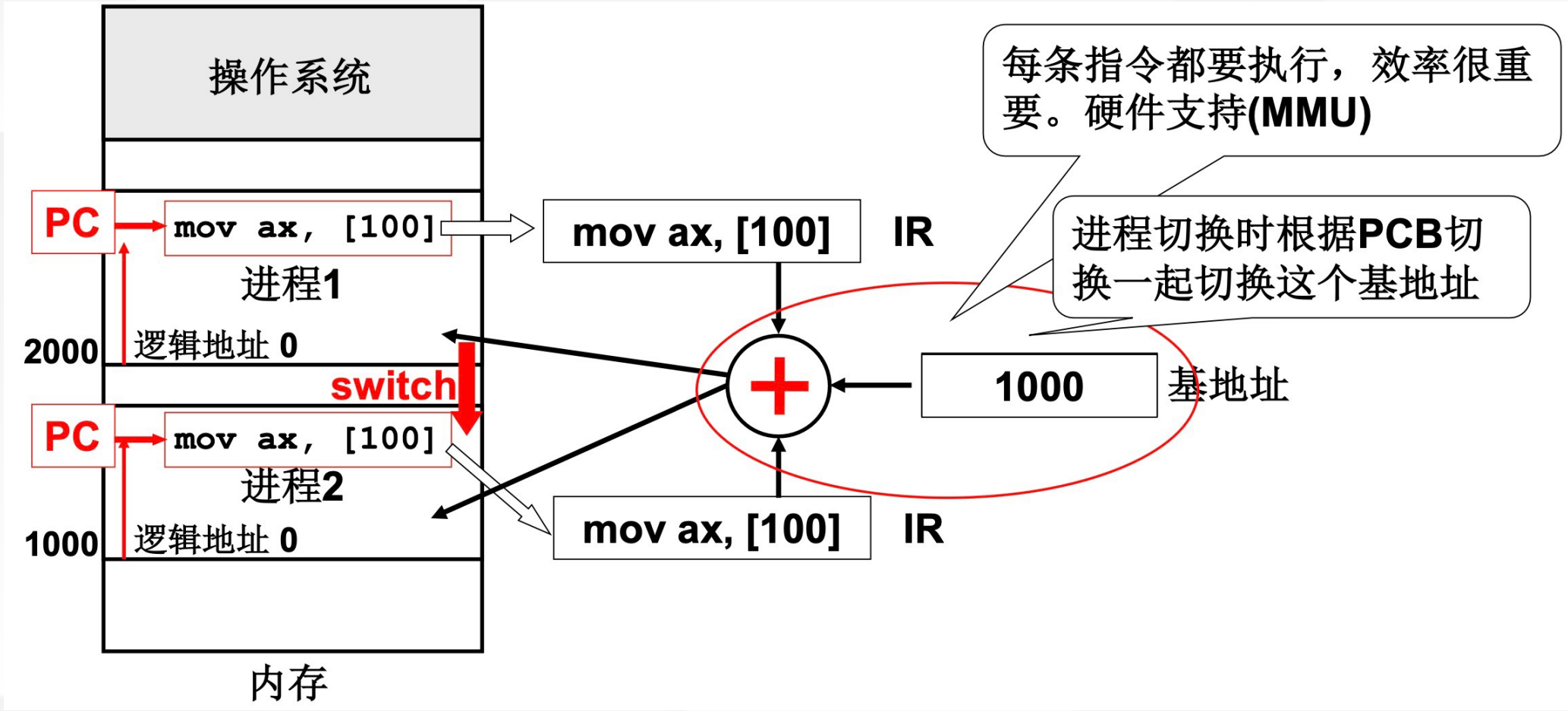
- 在运行每条指令时才完成重定位



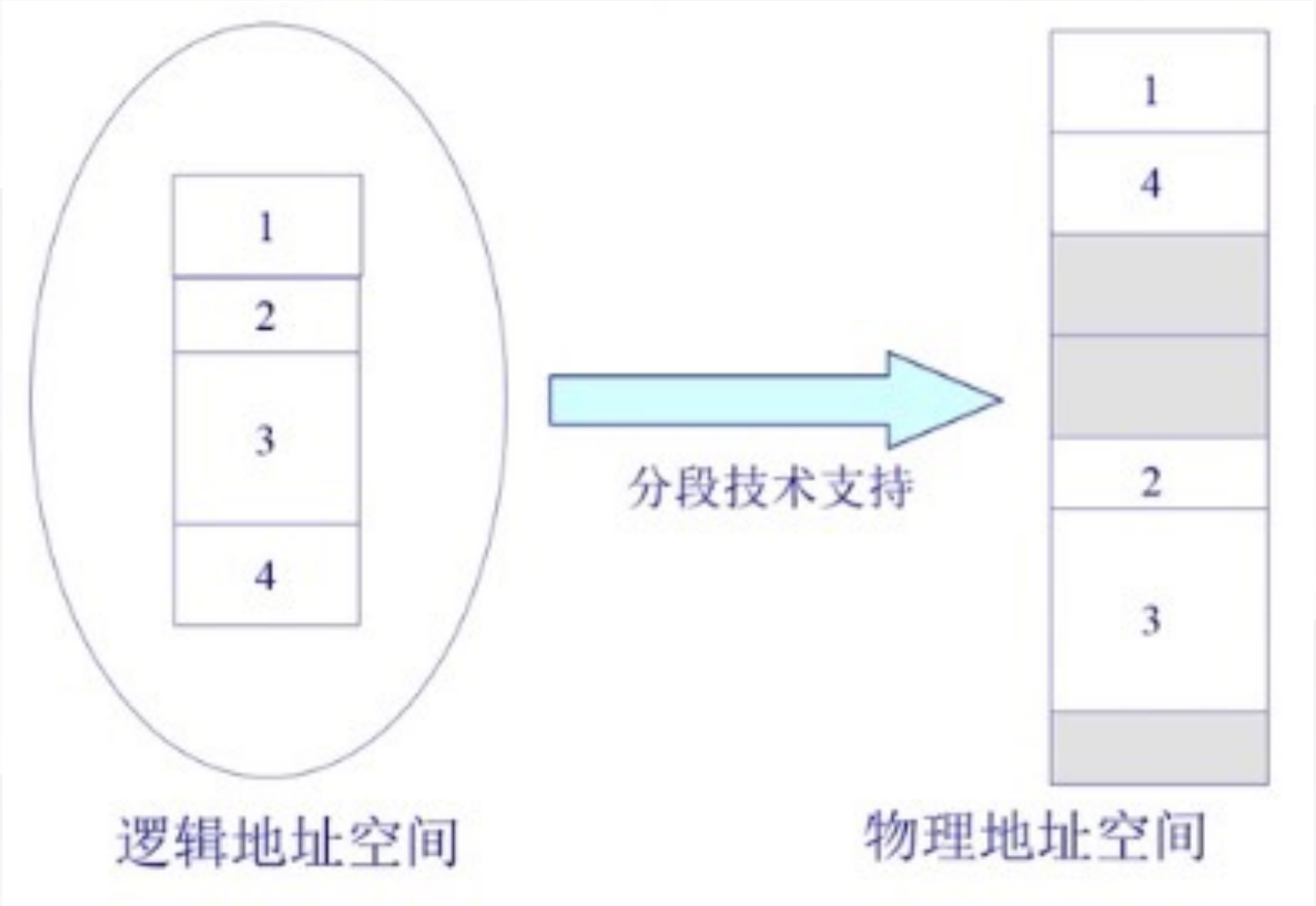
- 每个进程有各自的基地址，放在哪里？ PCB

执行指令时第一步先从PCB中取出这个基地址

运行时重定位+进程切换

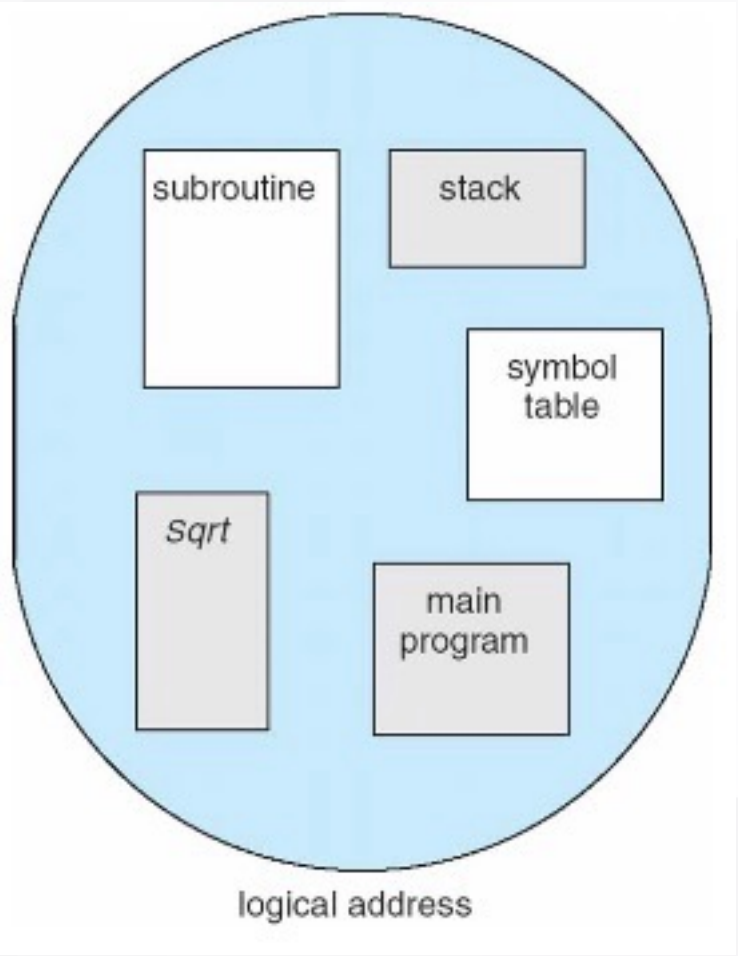

操作系统
PC
mov ax, [100]
进程1
2000
逻辑地址 0
switch
PC
mov ax, [100]
进程2
1000
逻辑地址 0
内存
mov ax, [100]
IR
mov ax, [100]
IR
1000
基地址
每条指令都要执行，效率很重要。硬件支持(MMU)
进程切换时根据PCB切换一起切换这个基地址

分段的逻辑视图



分段

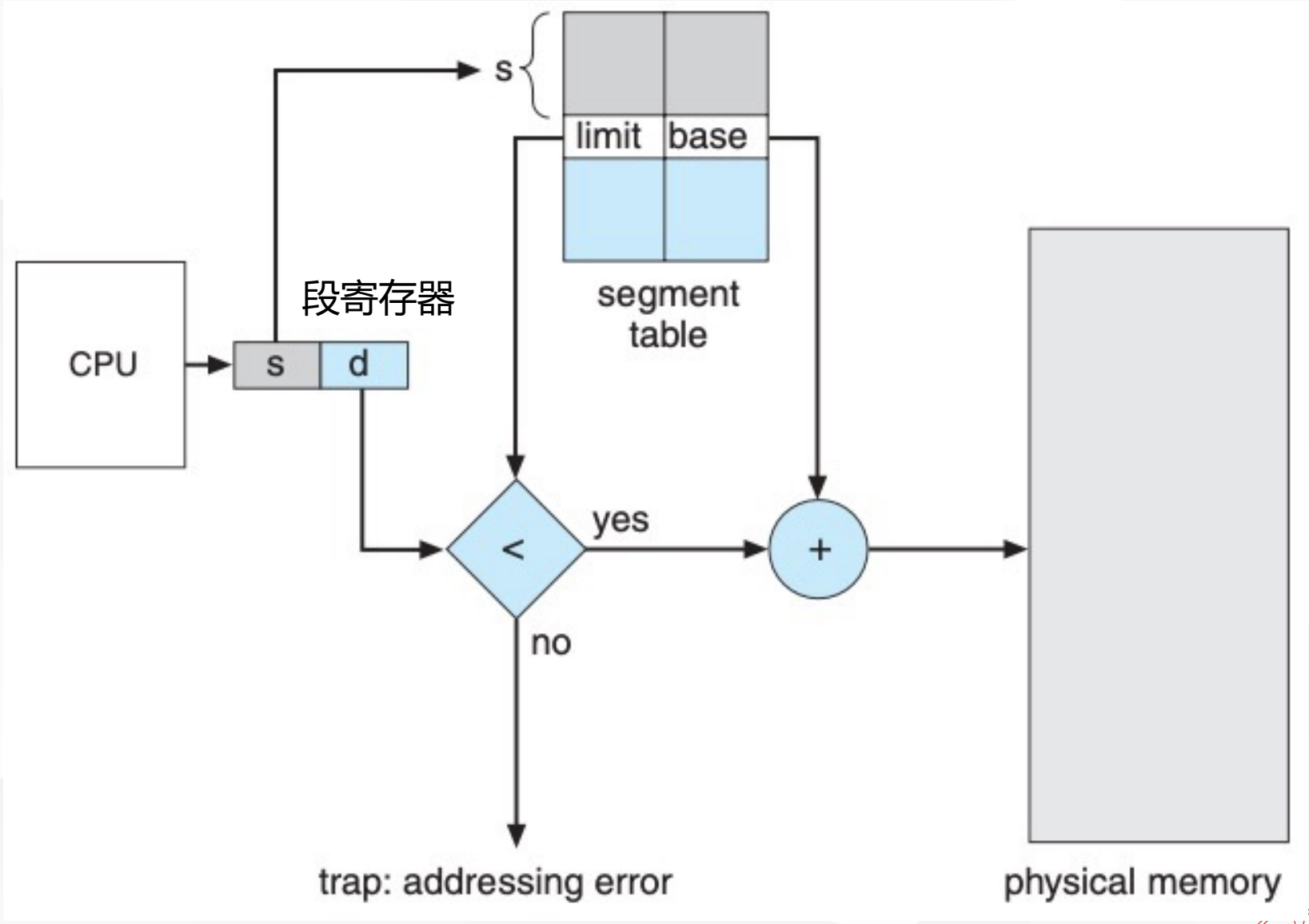
- 程序是一组段的集合
  - main program
  - procedure
  - function
  - method
  - object
  - local variables
  - global variables
- 分段是支持这种用户视图的内存管理方案



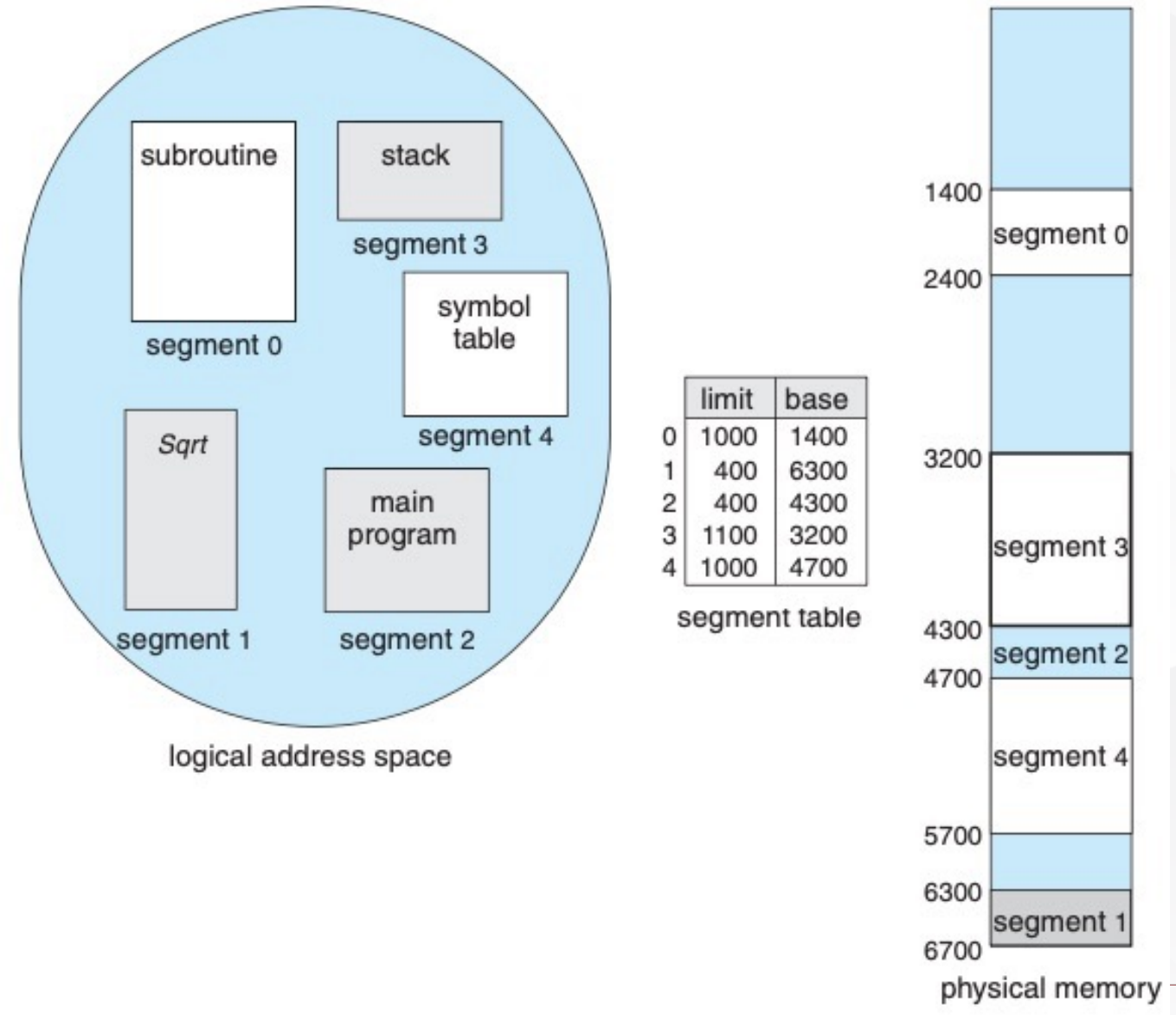
分段

- 逻辑地址空间由一组段构成。每个段有名称和长度。地址指定来段名称和段内偏移。用户通过两个量来指定地址：段名称和段偏移---由有序对（two tuple）：
  - <段号，偏移>
- 段表：用于映射用户定义的二维地址到物理地址
  - 段基地址：包含该段在内存中的开始物理地址
  - 段界限：该段所分配的长度

分段 硬件

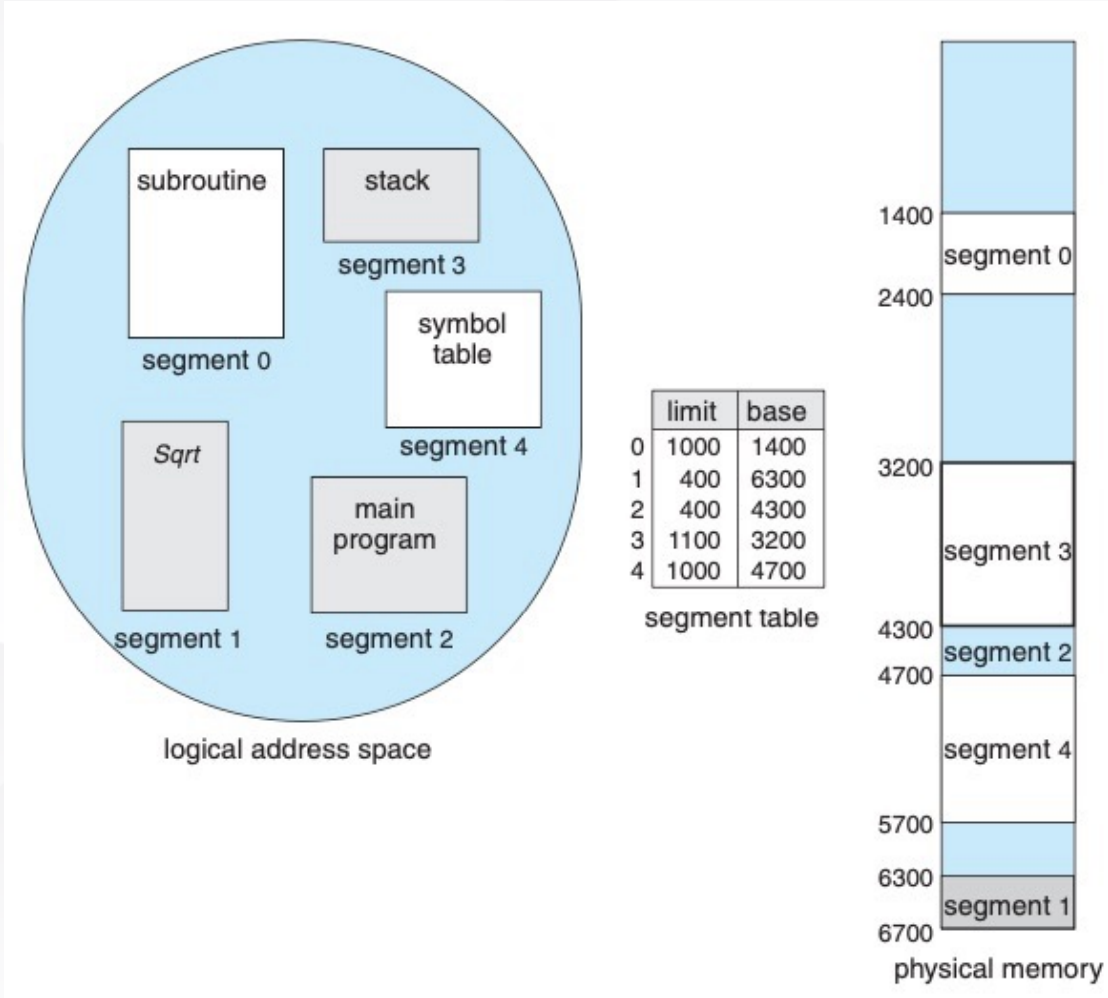

s
limit
base
segment table
段寄存器
CPU
s
d
<
yes
+
no
trap: addressing error
physical memory

分段 例子


subroutine
stack
segment 3
segment 0
symbol table
segment 4
Sqrt
main program
segment 1
segment 2
logical address space
limit base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
physical memory

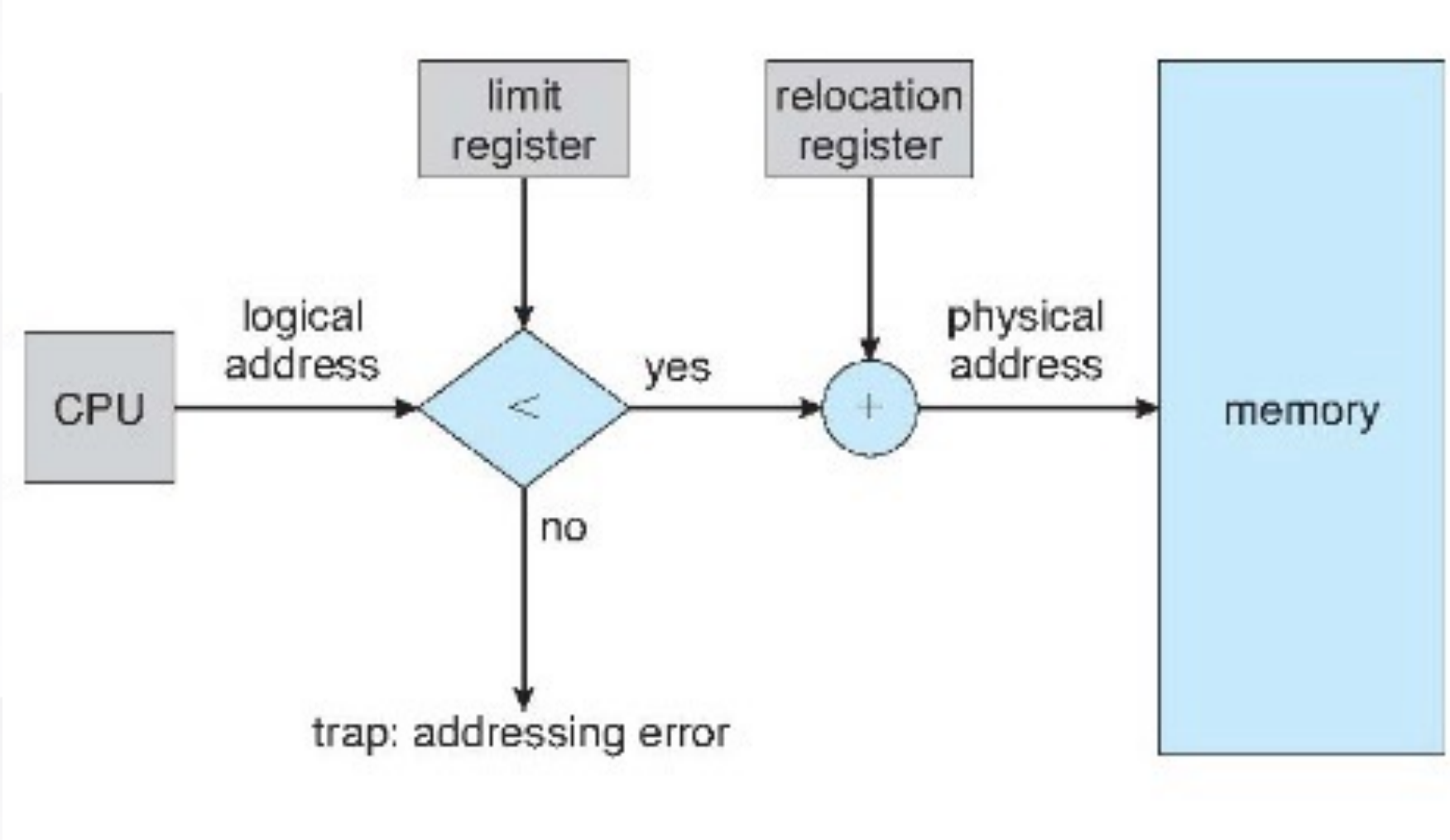
分段的优点

- 分段灵活
  - 当某一个段数据增长，可以扩展
- 符合程序员的编程习惯



内存保护 重定位和界限寄存器的硬件支持

界限寄存器=74600，重定位寄存器=100050

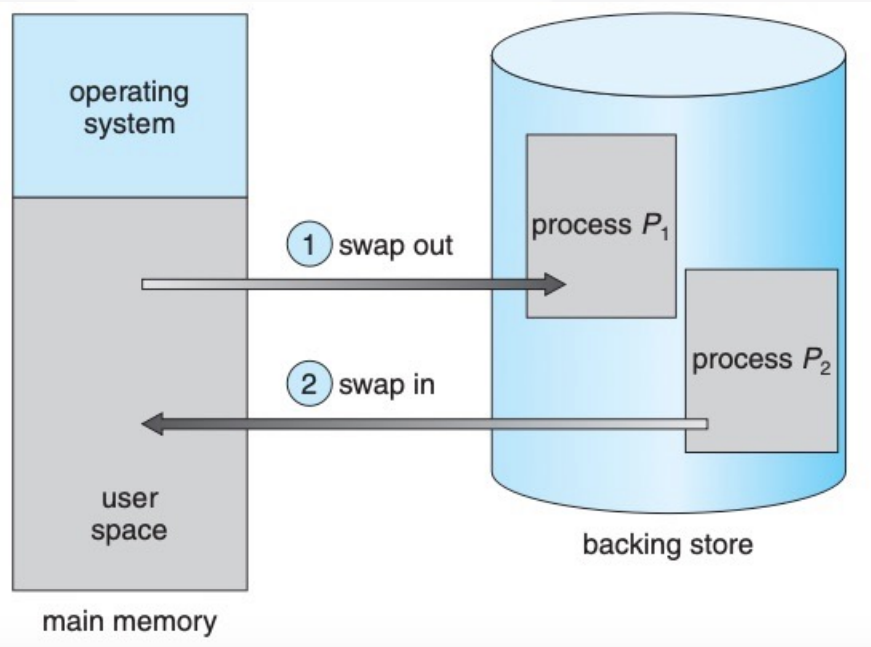


动态加载

- 在迄今为止的讨论中，一 个进程的整个程序和所有数据都应在物理内存中，以便执行。因此，进程的大小受限于内存的大小
- 为了获得更好的内存空间利用率，可以使用动态加载
  - 一个程序只有在调用时才被加载
  - 所有程序最初都保存在磁盘上
- 优点：节约内存空间

交换

- 进程可以暂时从内存交换到备份存储，当再次执行再调回内存
- 标准交换：在内存与备份存储之间移动进程，备份存储通常是快速磁盘
  - 系统维护一个可运行的所有进程的就绪队列
  - 查看进程是否在内存、以及是否有空闲内存区域



交换

- 标准交换的开销较大
  - 假设换入换出一个100MB的进程需要4s，对于一个3GB的内存，全部交换需要60s
  - 因此，明确具体要交换什么进程、内存使用情况是很重要的。可通过系统调用request_memory()和release_memory()来得知
- 约束
  - 要换出进程，应确保该进程是完全处于空闲的。例如等待I/O的进程，若换出进程，I/O操作可能试图操作其他进程的内存，两种解决方案：
    - 不换出等待I/O的进程
    - I/O操作的执行只能使用操作系统的缓冲（双缓冲）

交换 移动系统

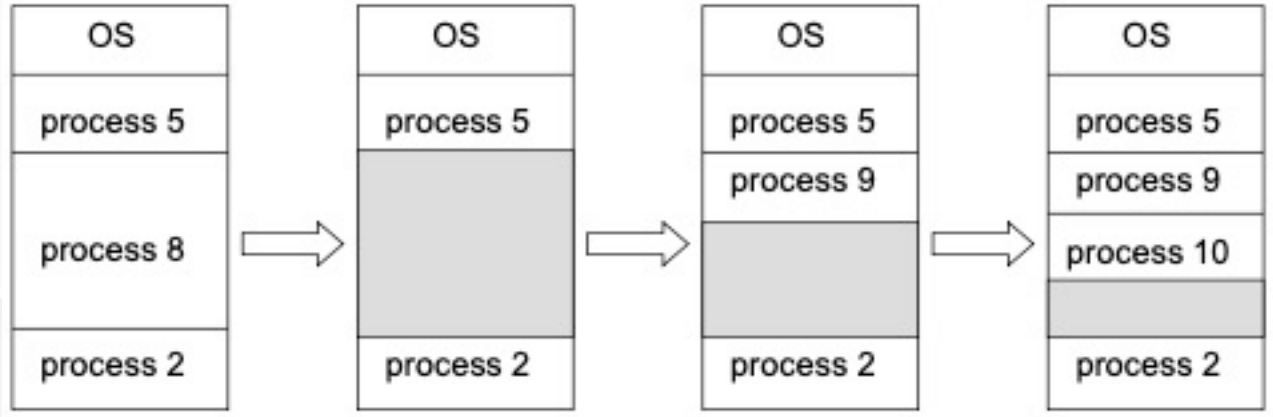
- 移动系统通常不支持交换，原因：
  - 通常采用闪存，空间约束较大
  - 闪存写入次数有限
  - 闪存与内存之间的吞吐量差
- 当空闲内存降低到一定阈值
  - 苹果的IOS，不是采用交换，而是要求应用程序自愿放弃分配的内存
  - Android也不支持交换，而是终止进程；在终止进程之前，Android将其应用程序状态(application state)写到闪存，以便它能快速重新启动

连续内存分配

- 内存通常分为两个区域：一个用于驻留操作系统，另一个用于用户进程
- 操作系统可以放在低内存，也可放在高内存。影响中断向量的位置
  - 程序员通常将操作系统放在低内存
- 多个进程放在内存中，可以采用连续内存分配的方式，每个进程位于一个连续内存区域，与下一个进程内存相连

内存分配

- 固定分区：最初用于IBM OS/360操作系统，现在已不再使用
  - 多道程序的程度受限于分区数
  - 当一个分区空闲时，可以从输入队列中选择一个进程，调入空闲分区
  - “孔”：一整块可用内存区域，用一个列表记录
  - 当一个进程进入，将其分配给一个孔，直到没有足够大的可用孔



动态存储分配问题

- 如何根据一组空闲孔来分配大小为n的请求？
  - 首次适应：分配首个足够大的孔
  - 最优适应：分配最小的足够大的孔。需要遍历整个列表，这种方法会产生最小剩余孔
  - 最差适应：分配最大的孔。需要遍历整个列表，这种方法会产生最大剩余孔

动态存储分配问题

- 问题：如果某操作系统的段内存请求很不规则，有时候需要很大的块，有时候需要很小的块，那选择哪种分区算法最好？

A 最先适配

B 最佳适配

C 最差适配

答案：B

碎片

- 用于内存分配的首次适应和最优适应算法都会产生外部碎片的问题
  - 存储被分成了大量的小孔
- 采用首次适应方法的统计说明，假设有N个可分配块，那么可能有0.5N个块是外部碎片。即1/3的内存可能不能使用。这一特性被称为50%规则
- 为了避免维护极小的孔，一般会按固定大小的块来分配内存（而不是以字节为单位），因此，进程所分配的内存可能比所需的要大，两者之差被称为内部碎片

碎片

- 解决外部碎片问题的一种方法：紧缩
  - 通过移动内存内容，将所有空闲空间合并成一整块
    - 频繁地数据移动
  - 只有当重定位是动态的，并且在运行时进行的，紧缩会导致“死机”
    - 进程在此时无法运行
  - 花费大量时间：假如复制速度1M/1秒，则1G内存需要1000秒（17分钟）进行一次紧缩
- 另一种方法的解决方案：允许进程的逻辑地址空间不连续！
  - 分页

分页

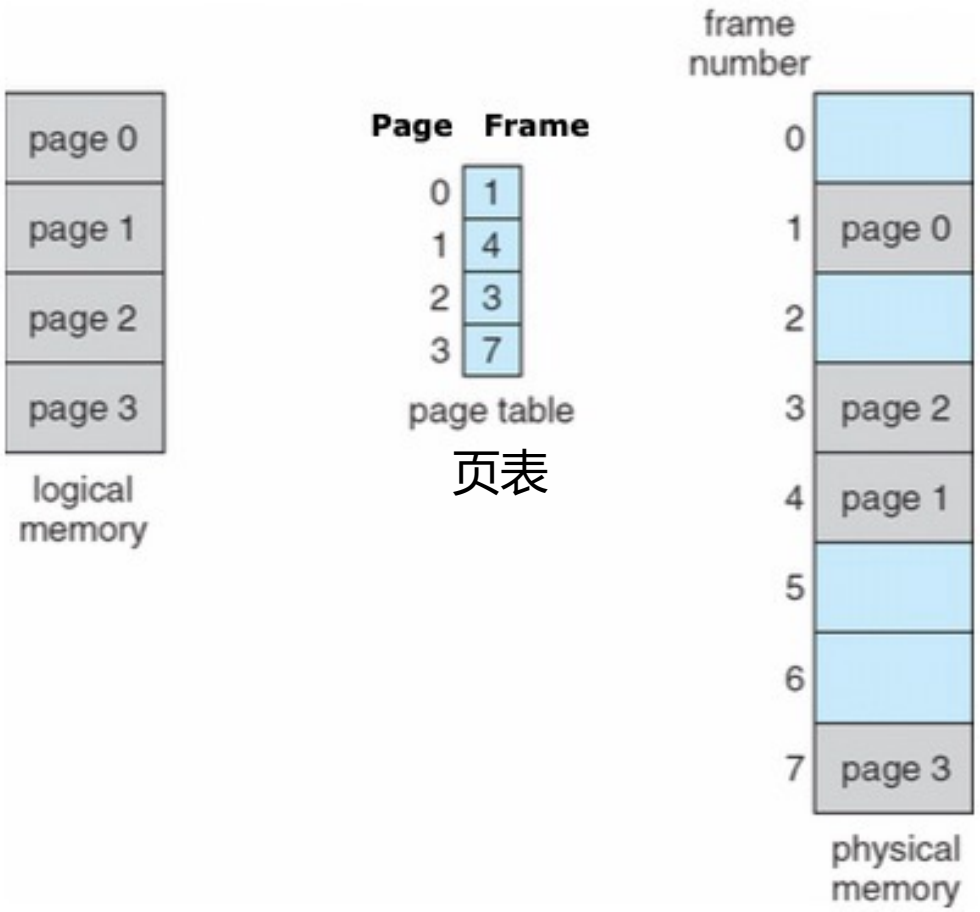
- 思考切面包：为什么面包基本没有碎片，因为将面包切成片
- 针对每个段内存请求，切分成多页（逻辑、物理）
- 问题：最多浪费的内存是多少？需要内存紧缩吗？
  - 一页！4K
  - 不需要

分页

- 分页也允许进程的物理地址空间是非连续的，并且能避免外部碎片和紧缩，而分段不可以，更优化，常用于大型机和智能手机的操作系统
- 将物理内存分为固定大小的块，称为帧或页帧（frame）
- 将逻辑内存也分为同样大小的块，称为页或页面（page）
- 当需要运行一个含有N页的程序时，需要寻找N个空闲帧，然后加载程序
- 构建页表：将逻辑地址翻译成物理地址

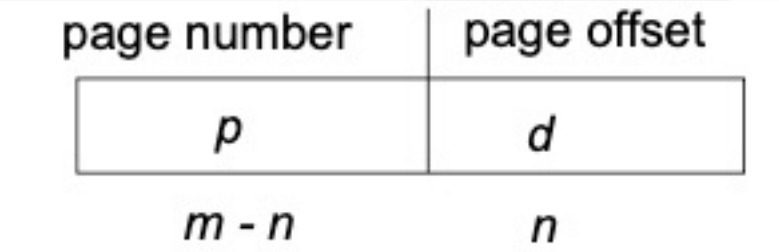
分页

- 逻辑地址 50
- 页大小 100
- 对应逻辑页0，物理地址 页帧 1
- 如何找到对应的物理地址？



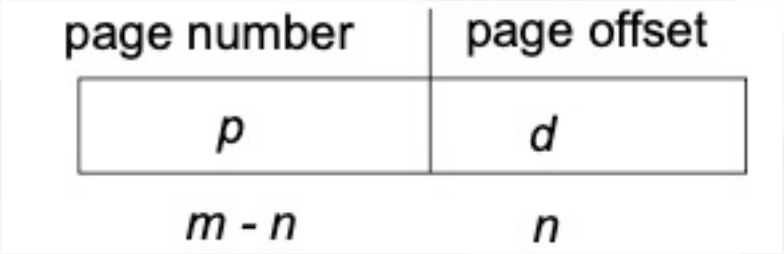
地址翻译

- 由CPU生成的**逻辑地址**分为两部分
  - 页码（page number，p）：作为页表的索引，包含了每页在物理内存中的基地址
  - 页偏移（page offset，d）：与基地址相结合形成物理内存地址，发送到内存单元，进行数据访存

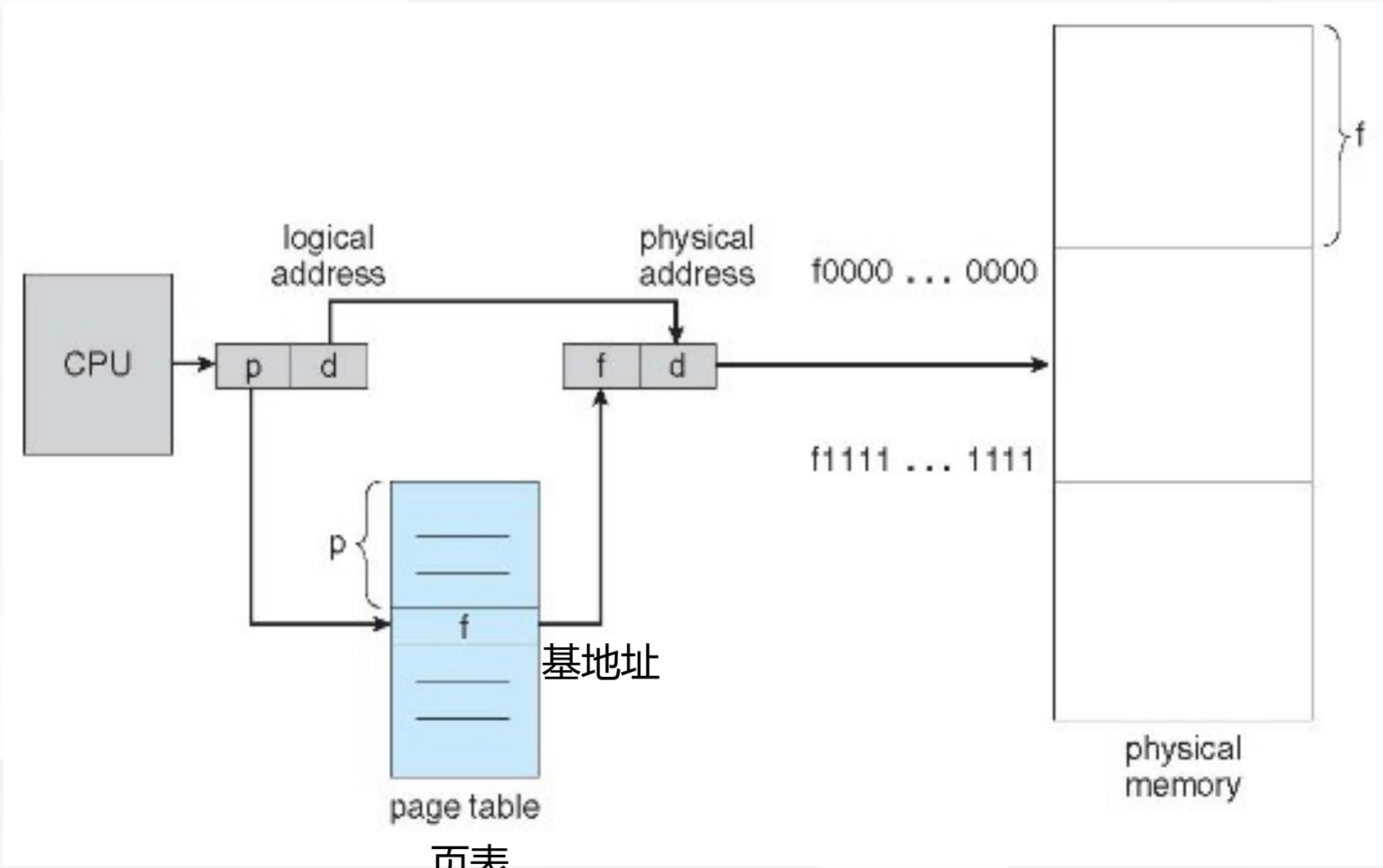


地址翻译

- 页的大小为2的幂，通常在512B-1GB之间，方便转换页码和页偏移
- 如果逻辑地址空间为2^m，且页大小为2^n字节，那么逻辑地址m-n位表示页码，而低n位表示页偏移。逻辑地址如下所示（其中p作为页表的索引，d作为页的偏移）：

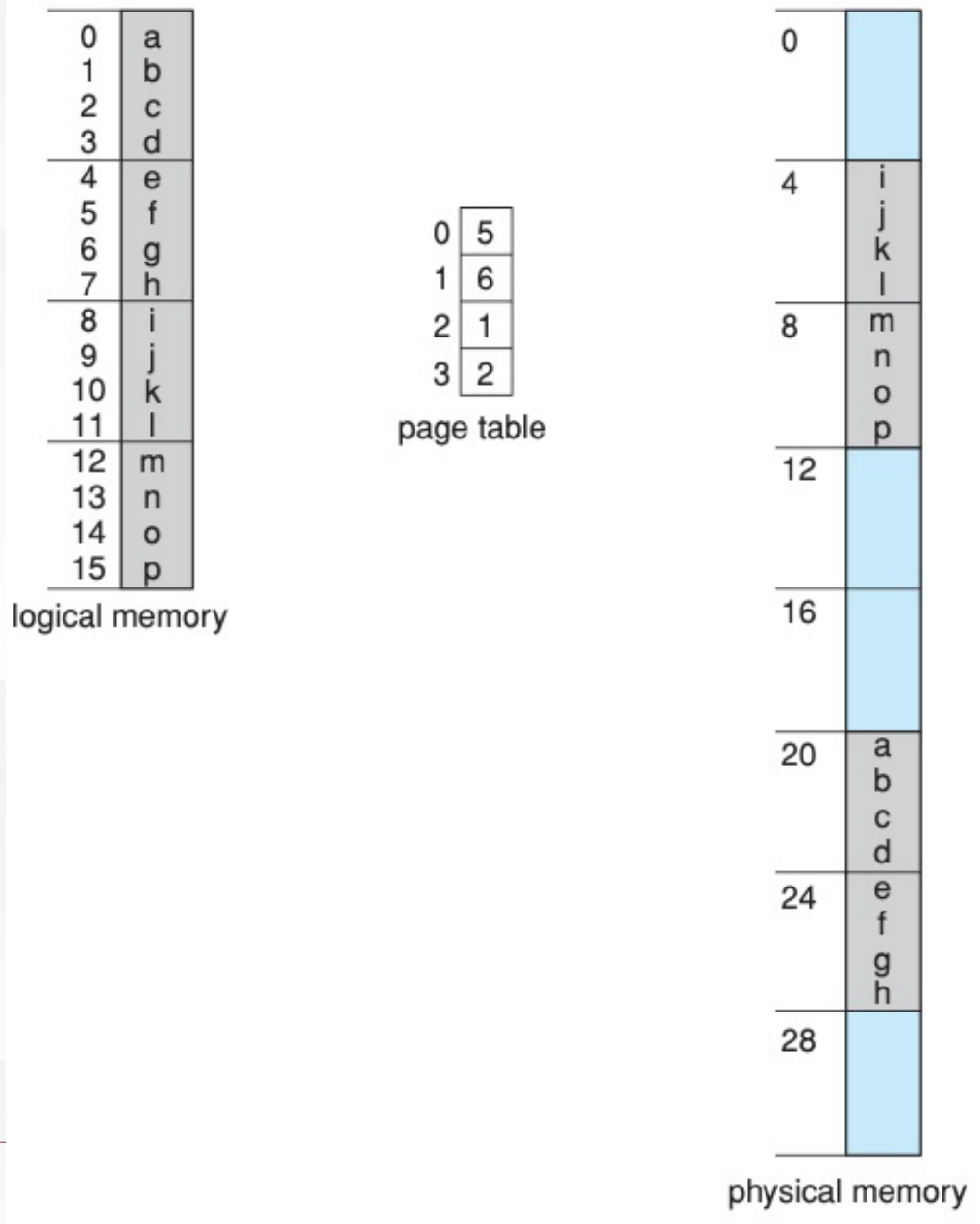


分页的硬件支持


CPU
logical address
p
d
physical address
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
基地址
page table
页表
physical memory

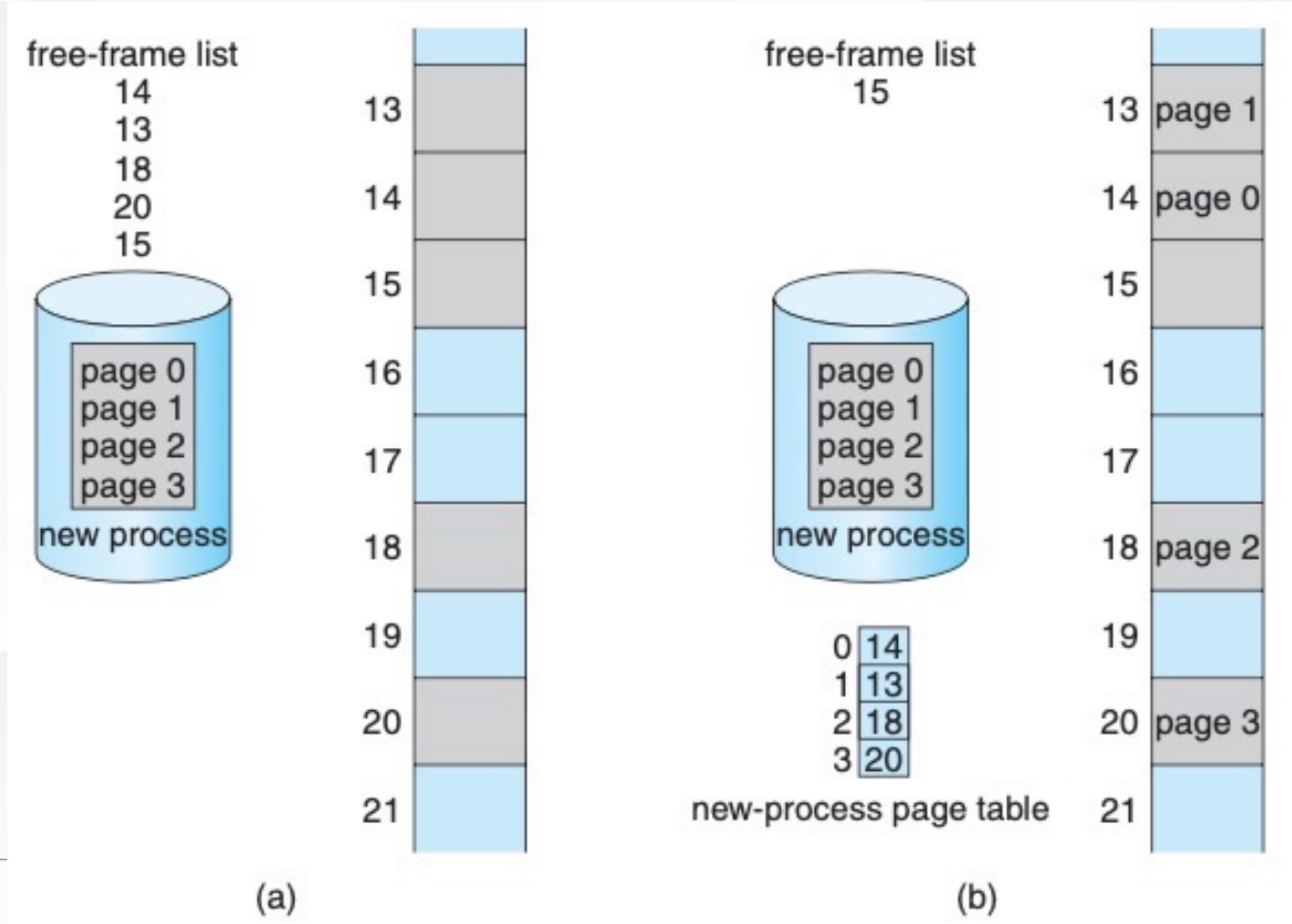
分页 实例

- n=2，采用大小为4字节（2^2）的页
- 逻辑地址0[=0000]的页码为0，页偏移为0，根据页表，可以查到页码为0对应帧5[=101]，因此逻辑地址0映射为物理地址20[=10100]
- 逻辑地址4[=0100]的页码为1，页偏移为0，根据页表，可以查到页码为1对应帧6[=110]，因此逻辑地址0映射为物理地址24[=11000]



空闲帧分配

- 当进程需要执行，检查该进程大小（按页）；如果进程需要n页，那么内存中至少要有n个空闲帧（用帧表记录），如果有，则将其分配给新进程



页表的实现

- 保存页表的方法
  - 为每个进程分配一个页表，页表的指针，与指令计数器PC等信息一起存入PCB
- 页表的硬件实现有多种方法
  - 将页表作为一组专用的寄存器来实现（高效逻辑电路），但是太贵
  - 将页表存放在内存中，使用页表基地址寄存器（Page-Table Base Register，PTBR）指向页表
    - 根据所得的帧码，加上页偏移，可得到物理地址
    - 但需要进行两次内存访问！延迟无法忍受

是否可以采用交换机制？

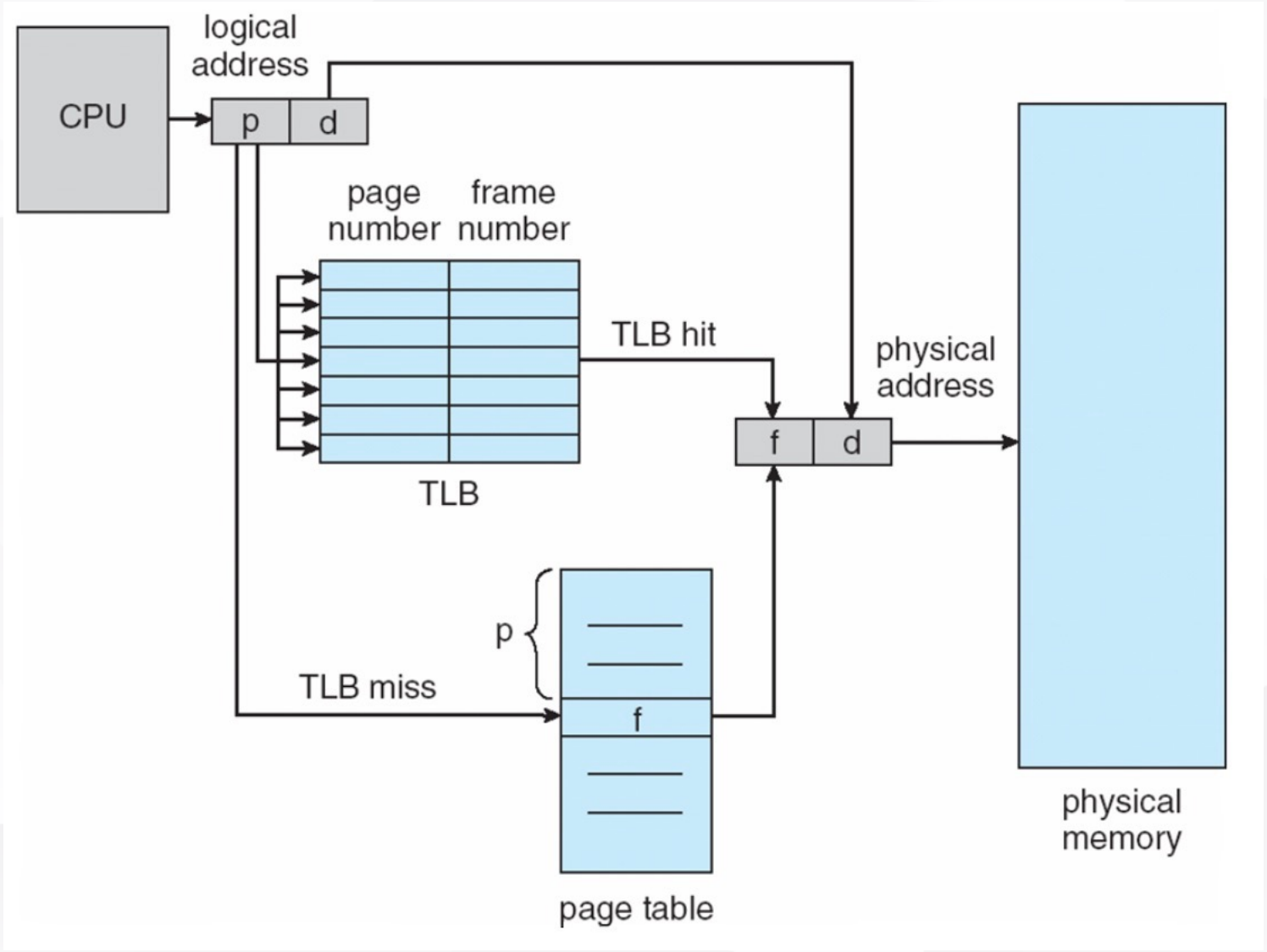
页表的实现

- 采用专用的、小的、查找快速的高速硬件缓冲，称为转换表缓冲区（Translation Look-aside Buffer，TLB）
- TLB由两部分组成：键（标签）和值
- 当关联内存根据给定值查找时，它会同时与所有键进行比较，得到相应值。由于并行比较，搜索速度快
- 昂贵，通常很小，能存放64-1024个条目
- TLB存放部分帧码
  - CPU产生逻辑地址后，将页码发送到TLB，如果找到，则帧码立即可用
    - 可以避免访问内存！

页表的实现

- TLB存放部分帧码
  - CPU产生逻辑地址后，将页码发送到TLB，如果找到，则帧码立即可用
  - 如果没有找到，则称为TLB未命中（TLB miss），则需要去内存访问页表
- 当TLB内条目已满，如何替换？
  - 替换策略很多：最近最少使用替换LRU、轮转替换、随机替换等
  - 有些条目固定，不被替换。例如重要的内核代码条目

页表的实现


CPU
logical address
p
d
page number
frame number
TLB hit
TLB
physical address
f
d
p
TLB miss
f
page table
physical memory

有效内存访问时间

- 命中率（hit ratio）=α
  - 在TLB中找到感兴趣页码的次数的百分比
- 访问内存的时间=β
- 有效内存访问时间（effective memory-access time，EAT）
  - EAT= αxβ+$(1-\alpha)$x2β
- 假设α=80%，β=100ns，则EAT=0.8x100+0.2x200=120ns
  - 比平均内存访问时间多了20%
- 假设α=99%，β=100ns，则EAT=0.99x100+0.01x200=101ns
  - 仅比平均内存访问时间多了1%
- 命中率对TLB、内存访问性能至关重要！

页表结构

- 页表结构的另一大问题是空间开销巨大
  - 假设具有32位逻辑地址空间的一个计算机系统
  - 页的大小为4KB（2^{12}）
  - 页表具有100多万个条目（$2^{32}/2^{12}$）
  - 如果每个条目有4Bytes，那么每个进程需要4MB物理地址空间来存储页表
    - 开销过大
    - 不希望在内存中连续分配这个页表
- 分层页表
- 哈希页表
- 倒置页表

分层页表

- 页表结构的空间开销巨大的核心原因是大部分逻辑地址根本不会用到

页号	帧号	保护	有效
0	5	R	1
1	1	R/W	1
2			0
3	6	R	1

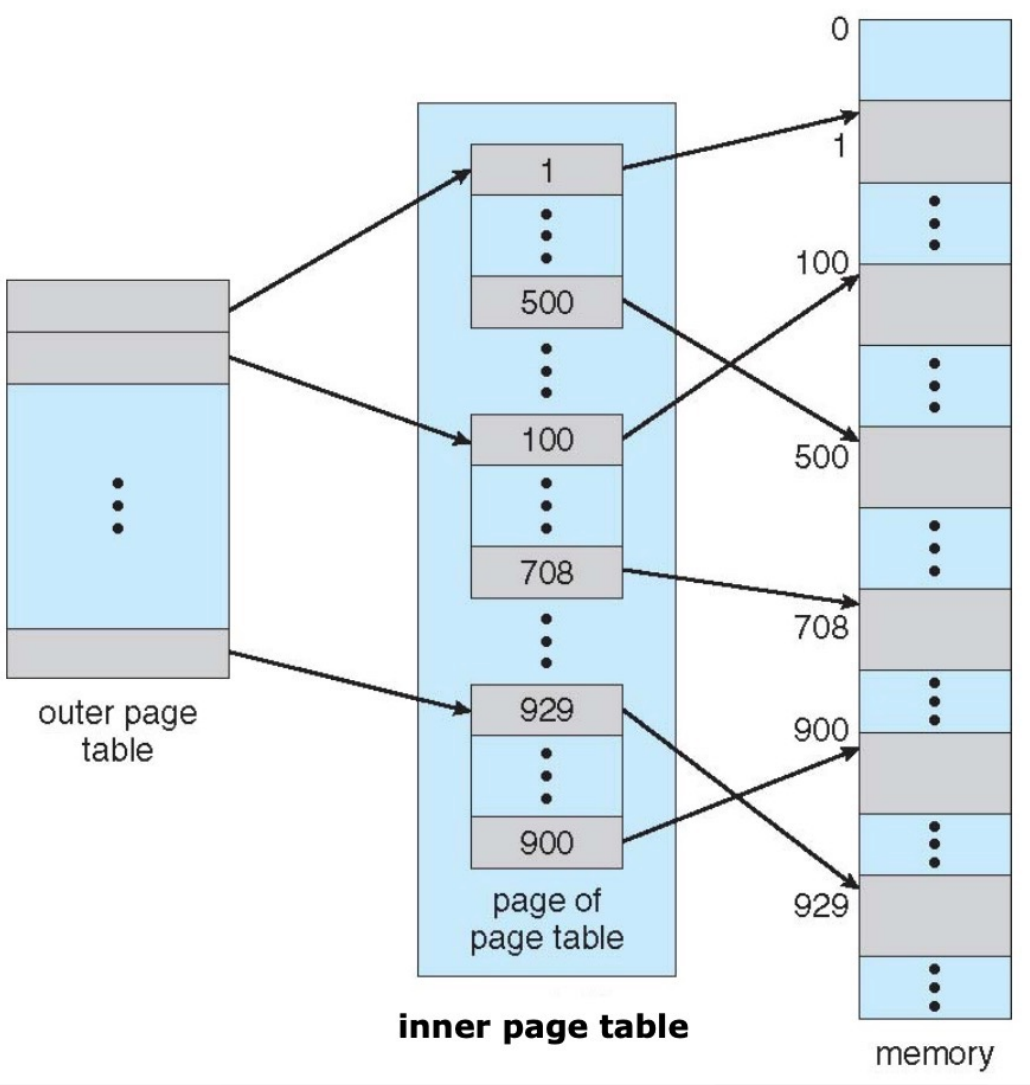
→

页号	帧号	保护
0	5	R
1	1	R/W
3	3	R

- 但如果直接删除不需要的逻辑地址，会导致存储空间不连续，需要逐个搜索对比，以找到帧号，显然不合理！
- 既要连续又要让页表占用内存少，怎么办？
- 用书的章目录和节目录来类比思考...

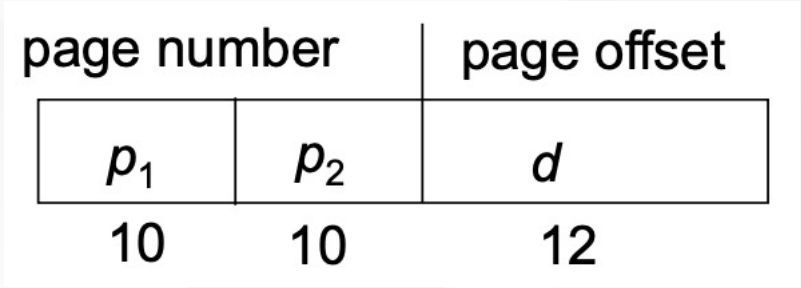
分层页表

- 使用两层页表算法：第一级页表（章）+第二级页表（节）



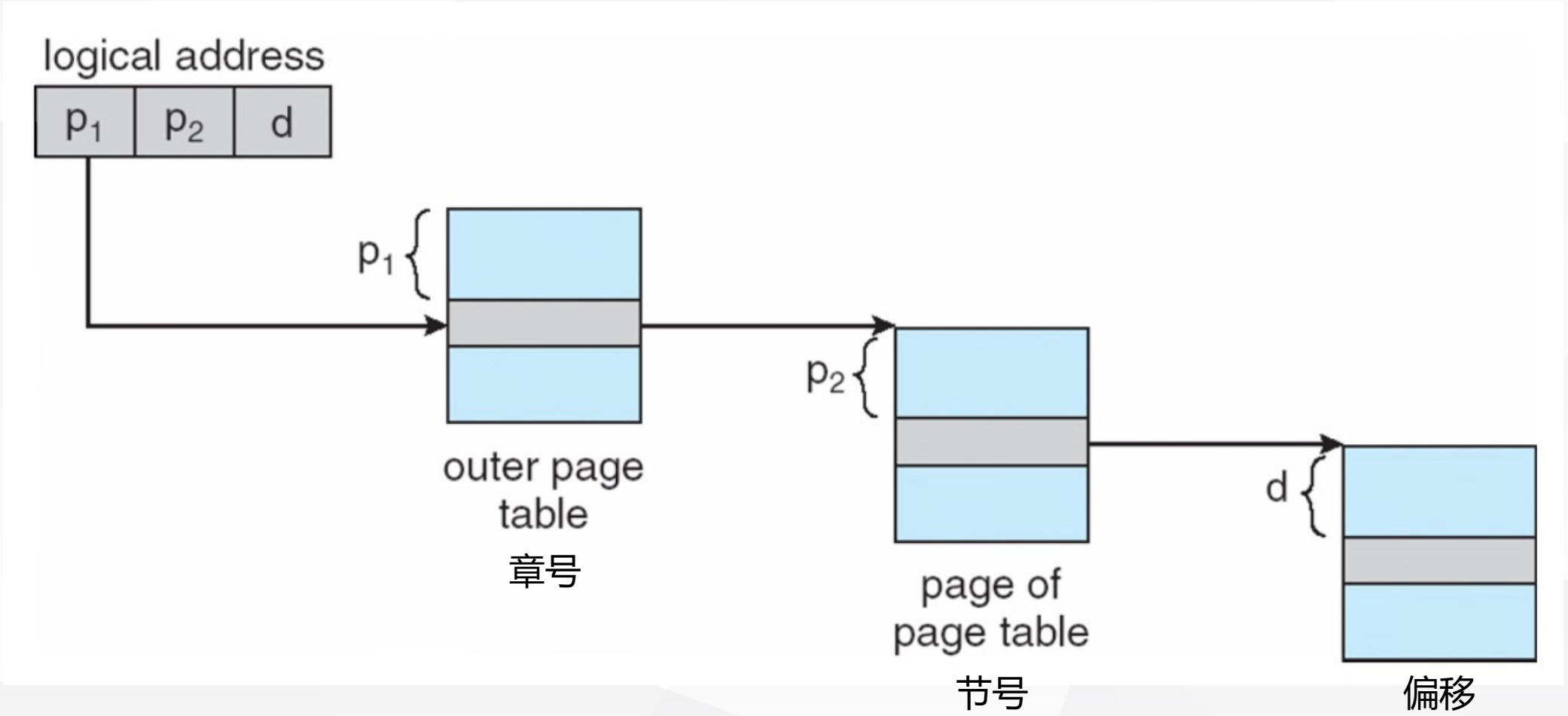
分层页表 实例

- 逻辑地址被分成两部分
  - 页码：20bit；页偏移：12bit
- 分层页表，将页码进行再次划分



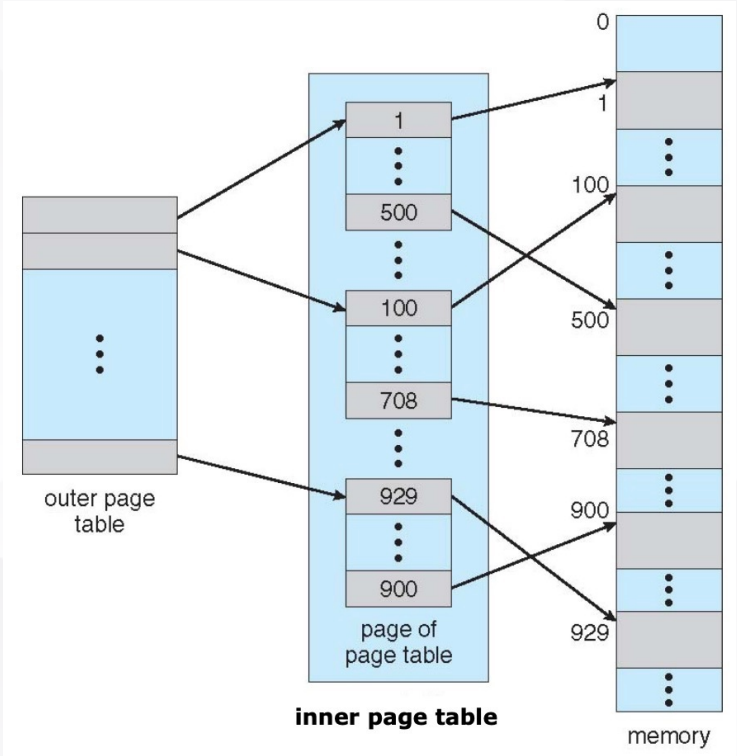
- 其中，p1是用于索引一级页表的，p2是二级页表的页偏移
- 这种方案也称为向前映射页表

分层页表 实例


logical address
p1
p2
d
p1
outer page table
章号
p2
page of page table
节号
d
偏移

分层页表的好处

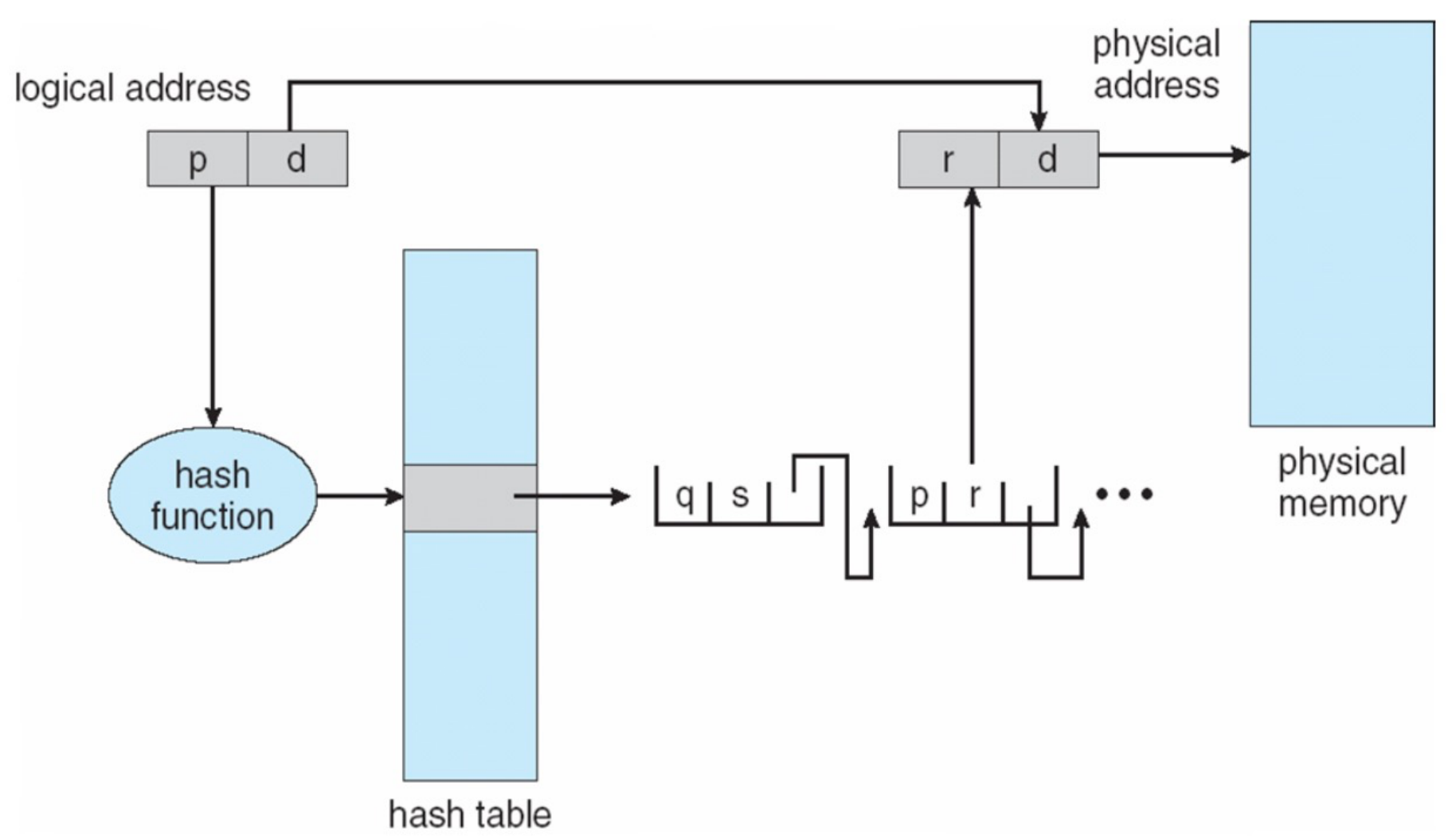
- 不需要的数据可以不在二级页表中存储，大幅减少空间浪费
- 一个页表是2^{10}x4字节=4K空间
- 一共4个页表，共需要16K空间<<4M



哈希页表

- 大于32位地址空间常用哈希页表
- 采用虚拟页码作为哈希值
- 哈希页表的每个条目包括一个链表，链表的元素哈希到同一位置
  - 处理碰撞，很多虚拟页码都哈希到同一位置
  - 链表中的每个元素由三个字段组成：虚拟页码、映射的帧码、指向链表中下一个元素的指针
- 工作原理：虚拟地址的虚拟页码哈希到哈希表；用虚拟页码与链表中的第一个元素的第一个字段比较，若匹配，那么相应帧码就形成物理地址；否则，那么与链表的后续元素的第一个字段比较，查找匹配的页码

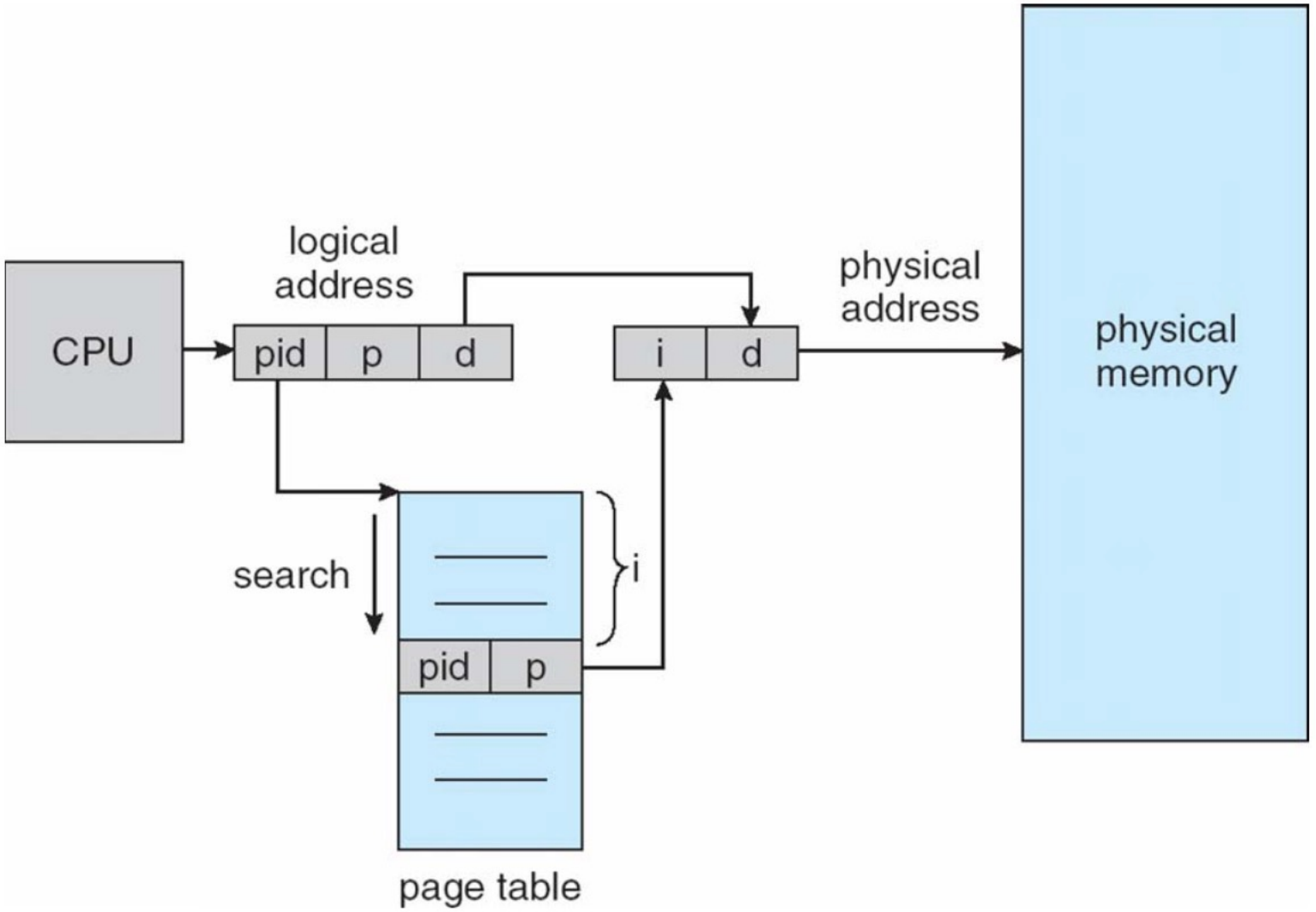
哈希页表


logical address
p
d
physical address
r
d
hash function
q
s
p
r
physical memory
hash table

倒置页表

- 不是每个进程拥有一个关联页表，而是对于每个真正的内存页/帧，倒置页表才有一个条目
- 每个条目包含保存在真正内存位置上的页的虚拟地址，以及拥有该页的进程信息
  - 整个系统只有一个页表
- 可以减少空间开销，但增加了搜索相应帧码的时间开销
- 通过哈希表可以减少搜索成本，通过TLB可以加速搜索
- 应用于IBM system 38、IBM power CPU

倒置页表


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table

课题习题

1. 给定6个内存分区：300KB、600KB、350KB、200KB、750KB和125KB，分别采用首次适应、最优适应、最差适应算法，如何放置大小分别为115KB、500KB、358KB、20KB和375KB（按顺序)的进程？根据它们使用内存的效率对算法进行排序
2. 页表分页的目的是什么？