上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

L12. 课程总结

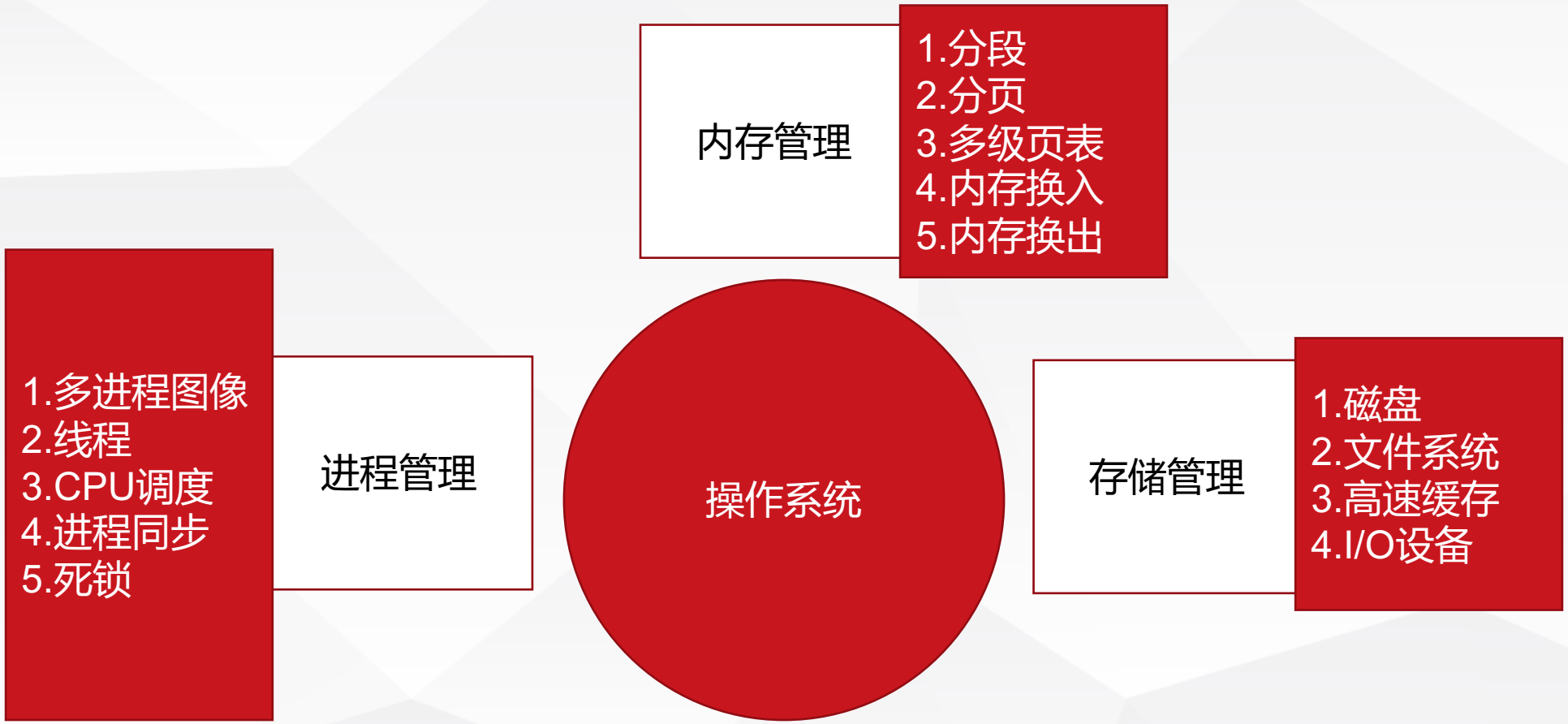
宋卓然

上海交通大学计算机系

songzhuoran@sjtu.edu.cn

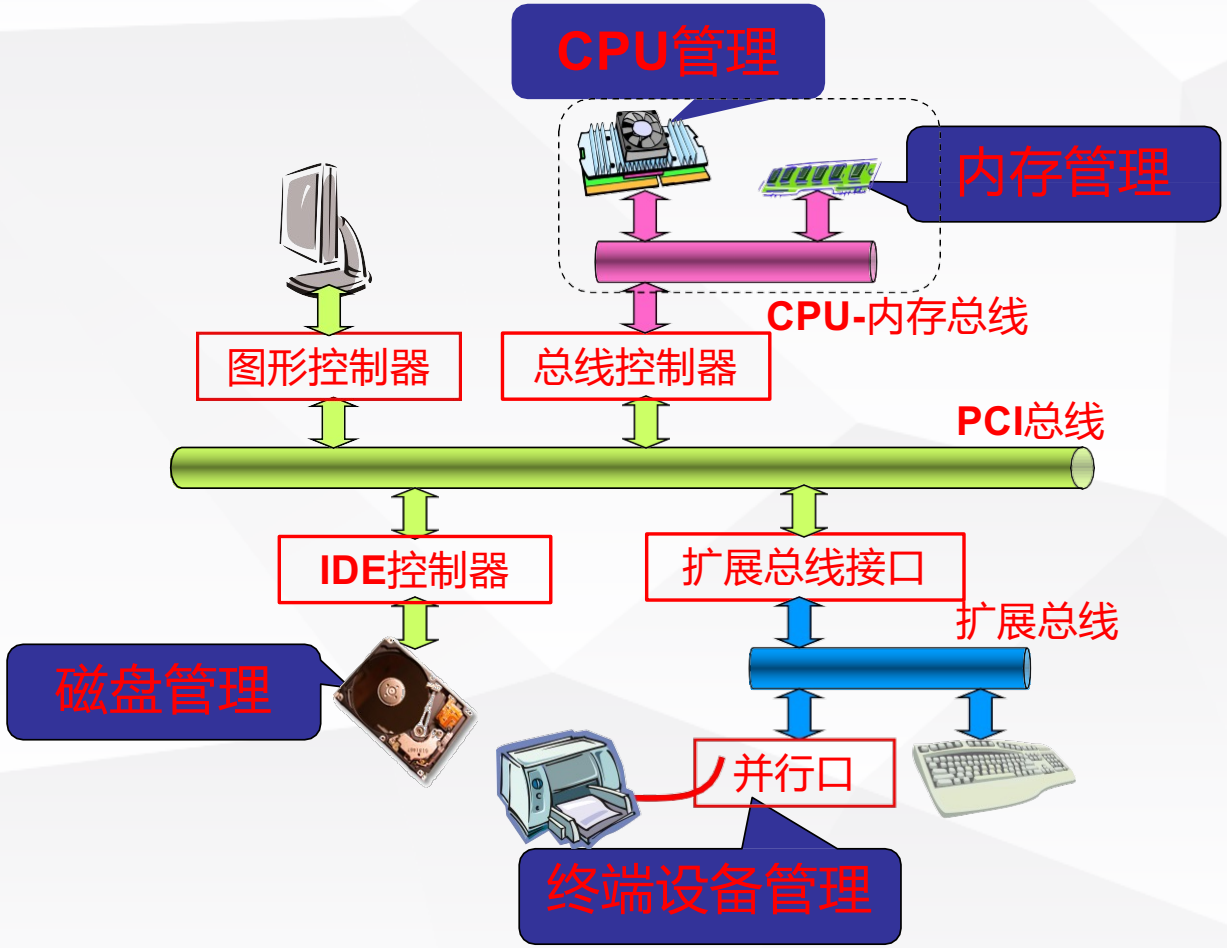
饮 水 思 源 • 爱 国 荣 校

操作系统课程总览


内存管理
1.分段
2.分页
3.多级页表
4.内存换入
5.内存换出
1.多进程图像
2.线程
3.CPU调度
4.进程同步
5.死锁
进程管理
操作系统
存储管理
1.磁盘
2.文件系统
3.高速缓存
4.I/O设备

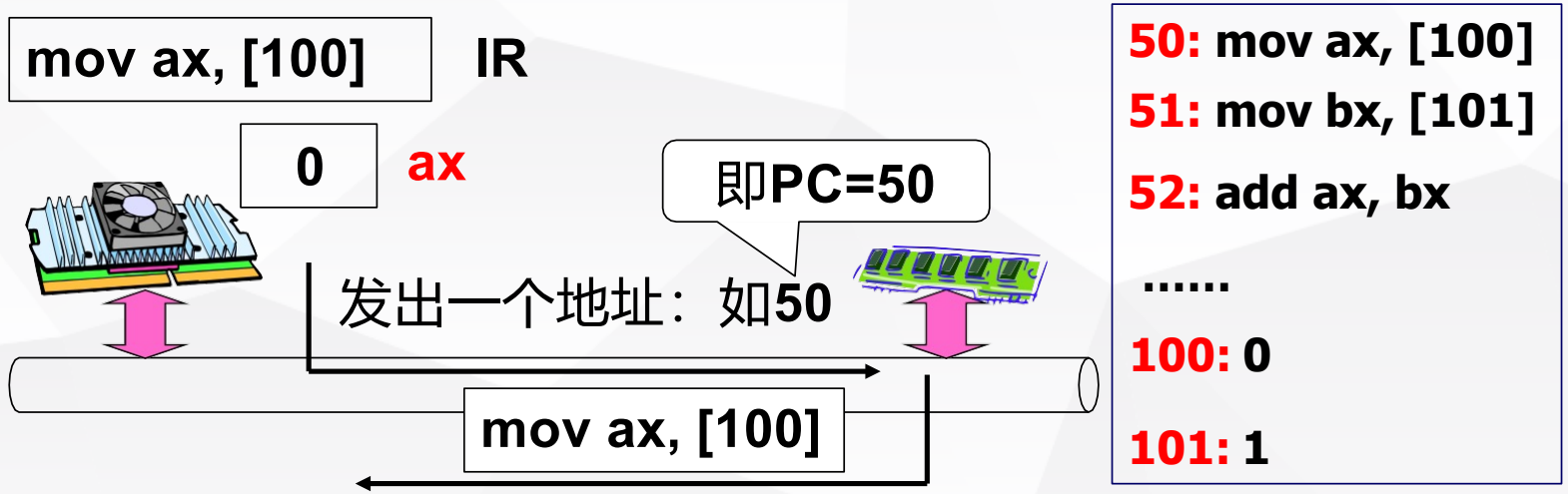
什么是操作系统

- 要管理硬件资源



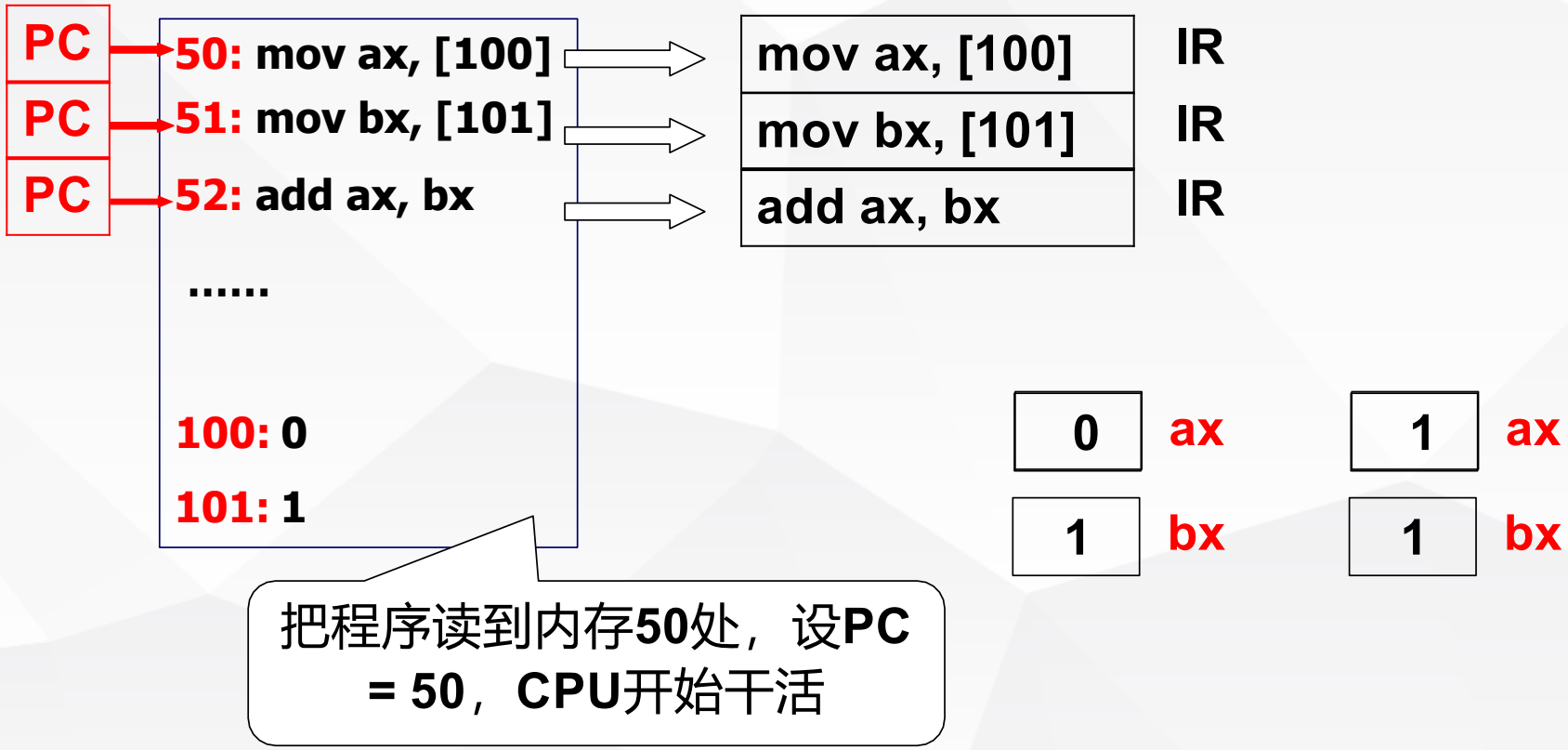
CPU的工作原理

- CPU上电以后发生了什么?
  - 自动的取指—执行
- CPU怎么工作?
- CPU怎么管理?



管理CPU的最直观方法

- 设好PC初值就完事!



进程的概念

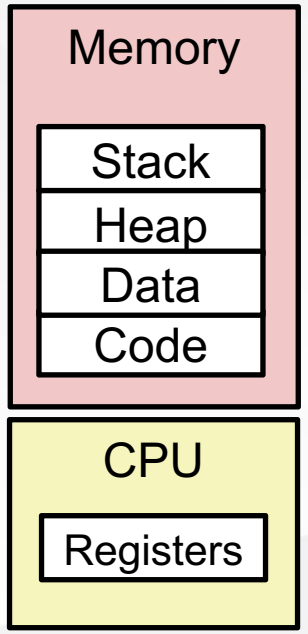
- 每一个进程都具有：

1、“看上去”独立使用CPU

- 当前的活动状态记录（PCB，上下文）
- program counter(当前的程序计数器)
- processor registers（当前的寄存器的状态）

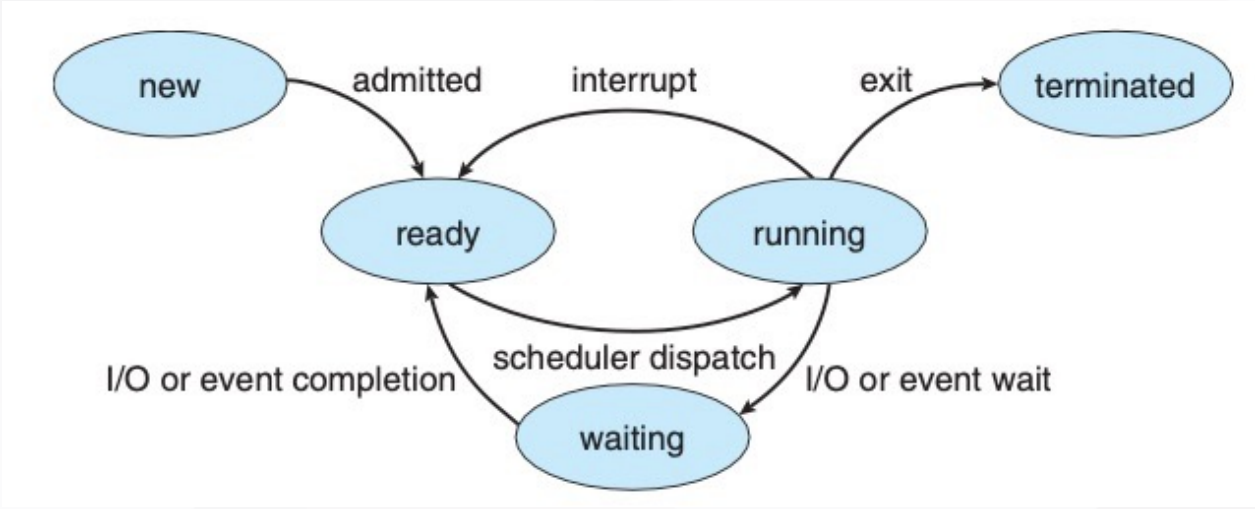
2、Private address space 私有地址空间

- “看上去”拥有整个内存



进程状态

- 进程在执行时可能会改变状态
- 右图可称为进程状态图
- 它能给出进程生存期的清晰描述
- 它是认识操作系统进程管理的一个窗口

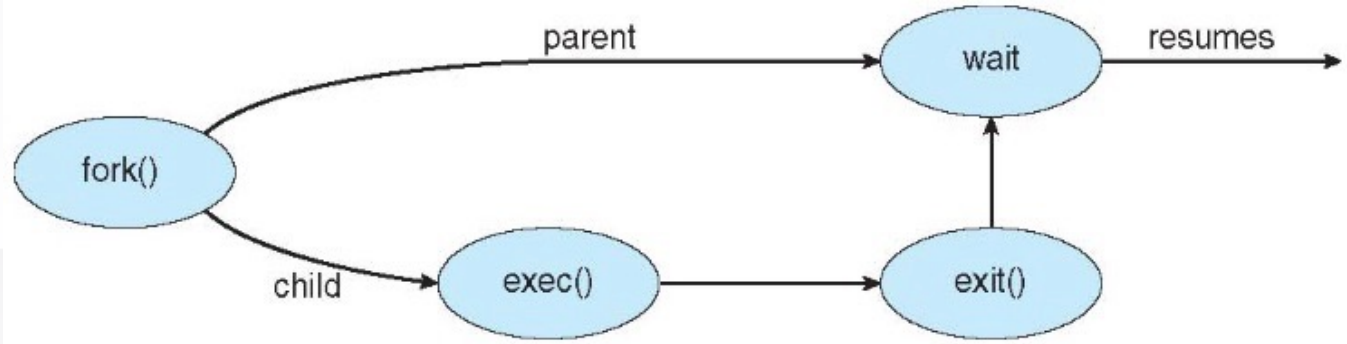


New: 刚刚创建
Running: 正在运行
Ready: 准备好被分配运行

Blocked/Waiting: 因为等待某些事件无法运行
Terminated: 执行结束

进程创建 UNIX系统实例

- fork，创建新进程
- 子进程使用exec 系统调用，用新程序来取代进程的内存空间
- wait系统调用，等待子进程完成

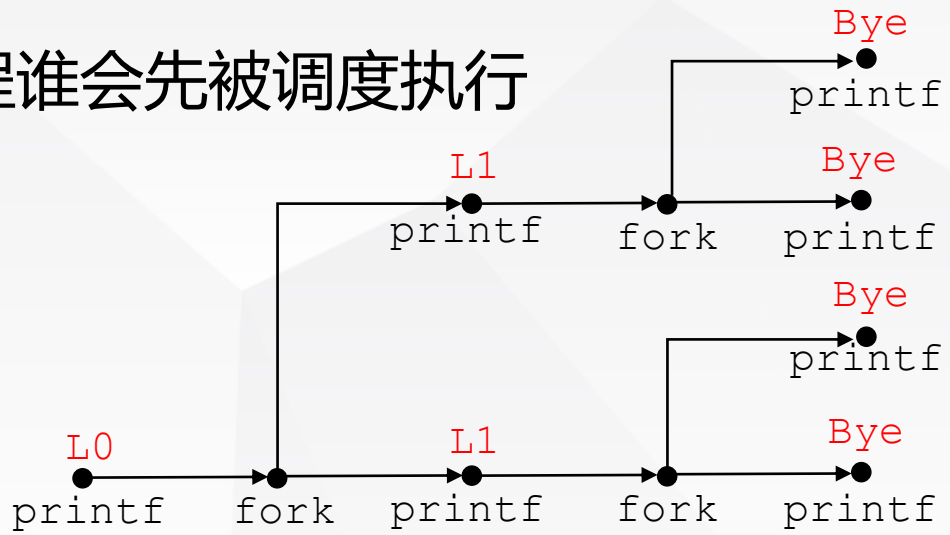



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    int i = 1;
    pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed");
          return 1;
    }
    else if (pid == 0) { /* child process */
          printf( "This is child.");
    }
    else { /* parent process */
          /* parent will wait for the child */
          wait (NULL);
          printf ("Child Complete.");
    }
    return 0;
```

fork：两个连续的fork

并发执行：无法预测父进程与子进程谁会先被调度执行

```
void fork2()                    forks.c
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



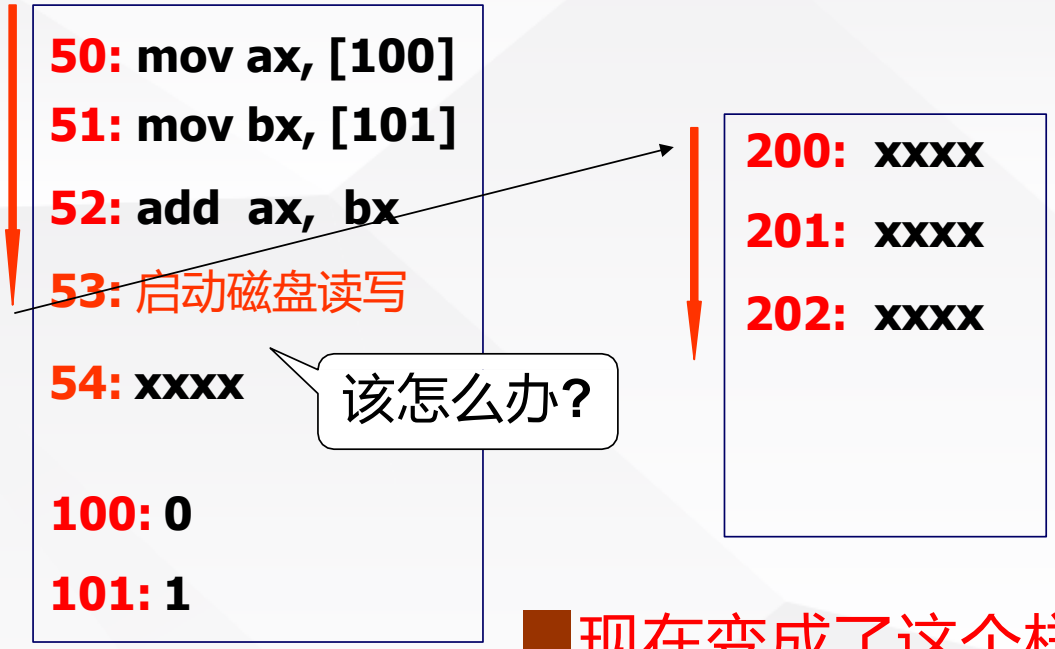
Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:

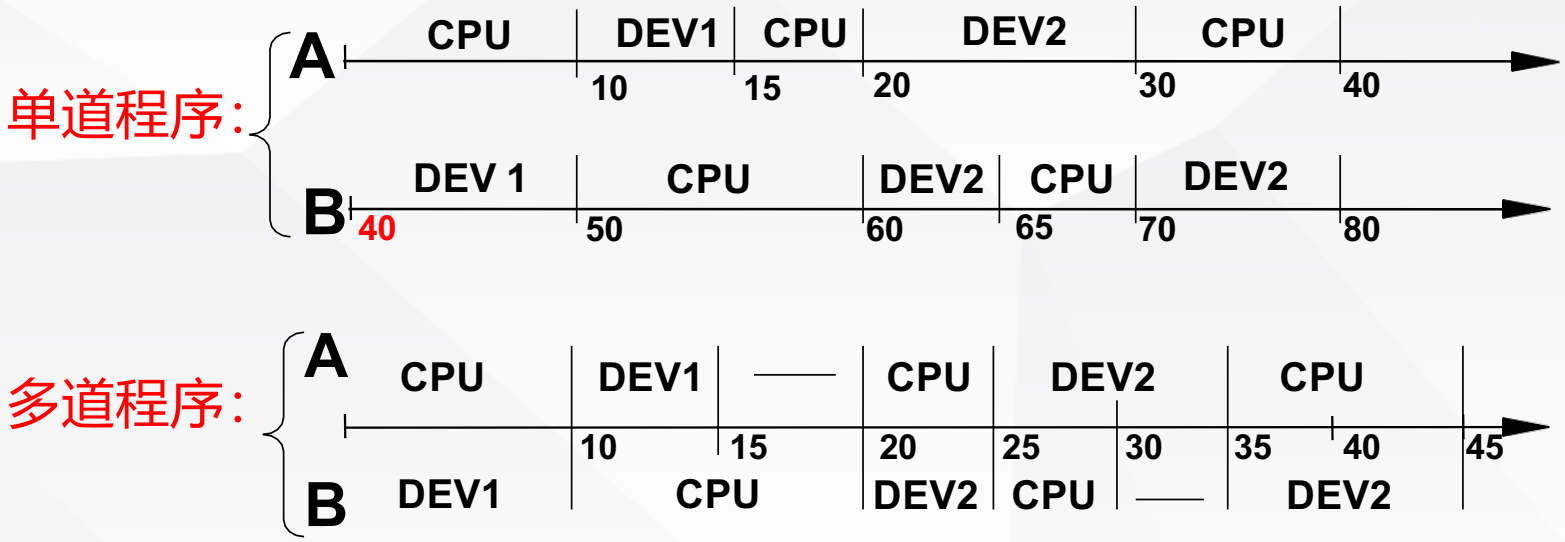
L0
Bye
L1
Bye
L1
Bye
Bye

进程切换



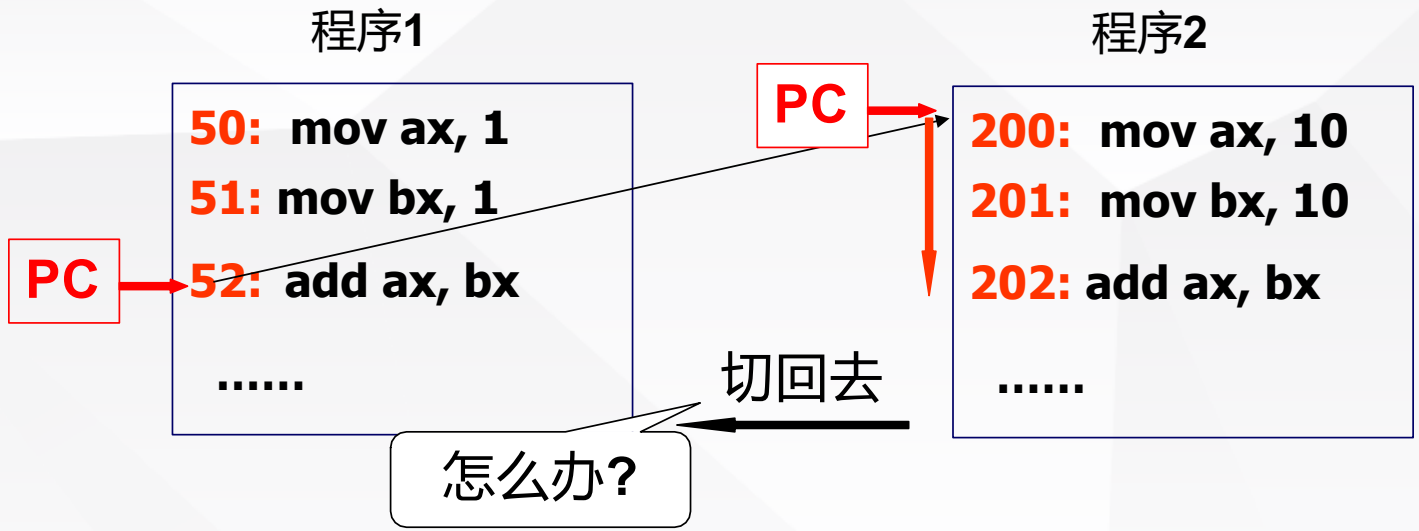
- 现在变成了这个样子

多进程的概念

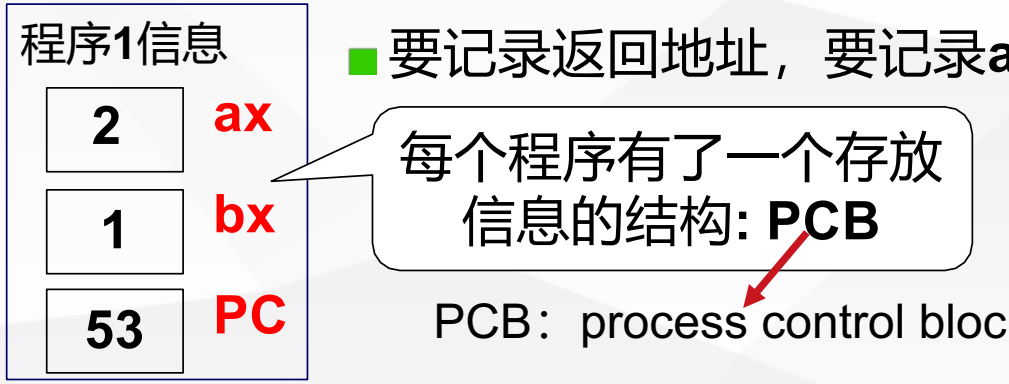


	单道程序	多道程序
CPU利用率	40/80=50%	40/45=89%
DEV1利用率	15/80=18.75%	15/45=33%
DEV2利用率	25/80=31.25%	25/45=56%

进程切换的开销：PCB



- 运行的程序和静态程序不一样了…



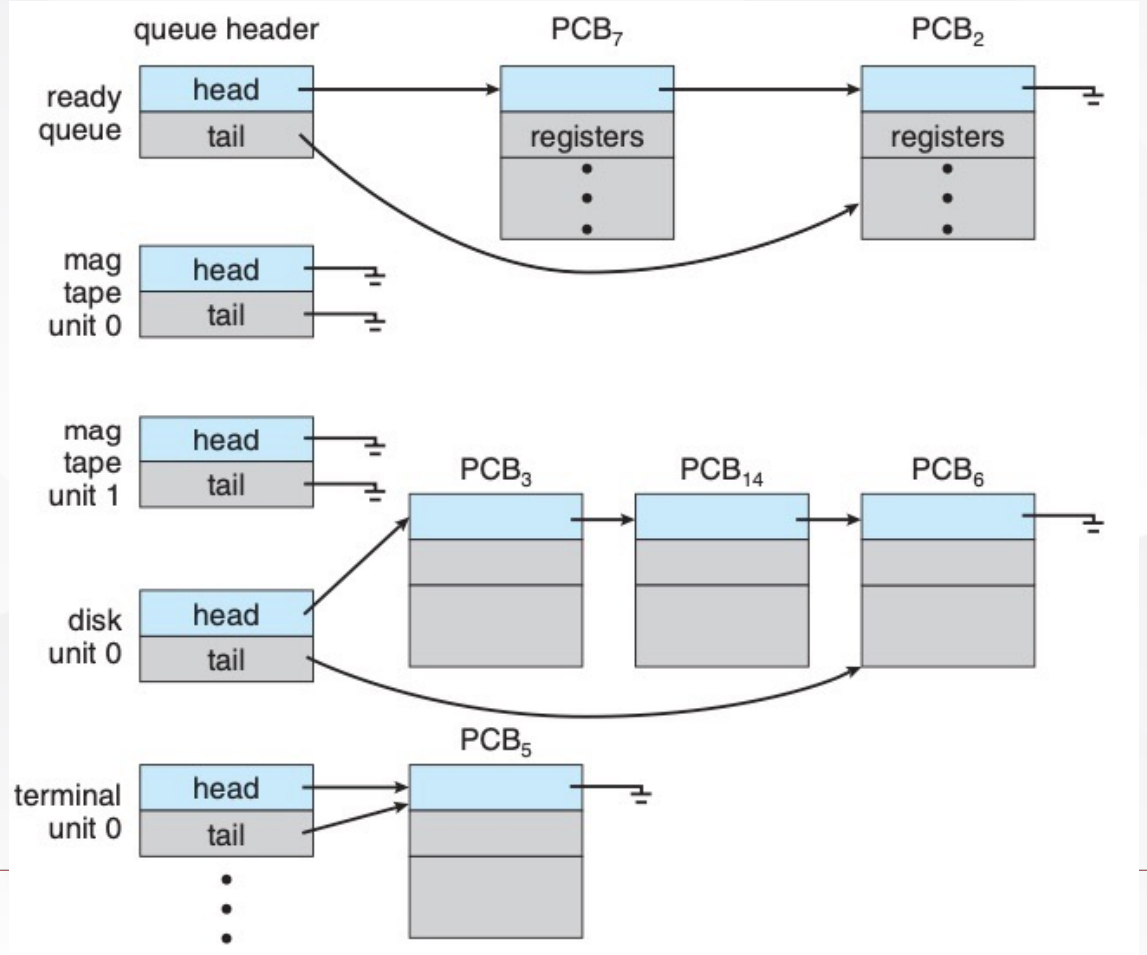
- 要记录返回地址，要记录ax…

每个程序有了一个存放信息的结构: **PCB**

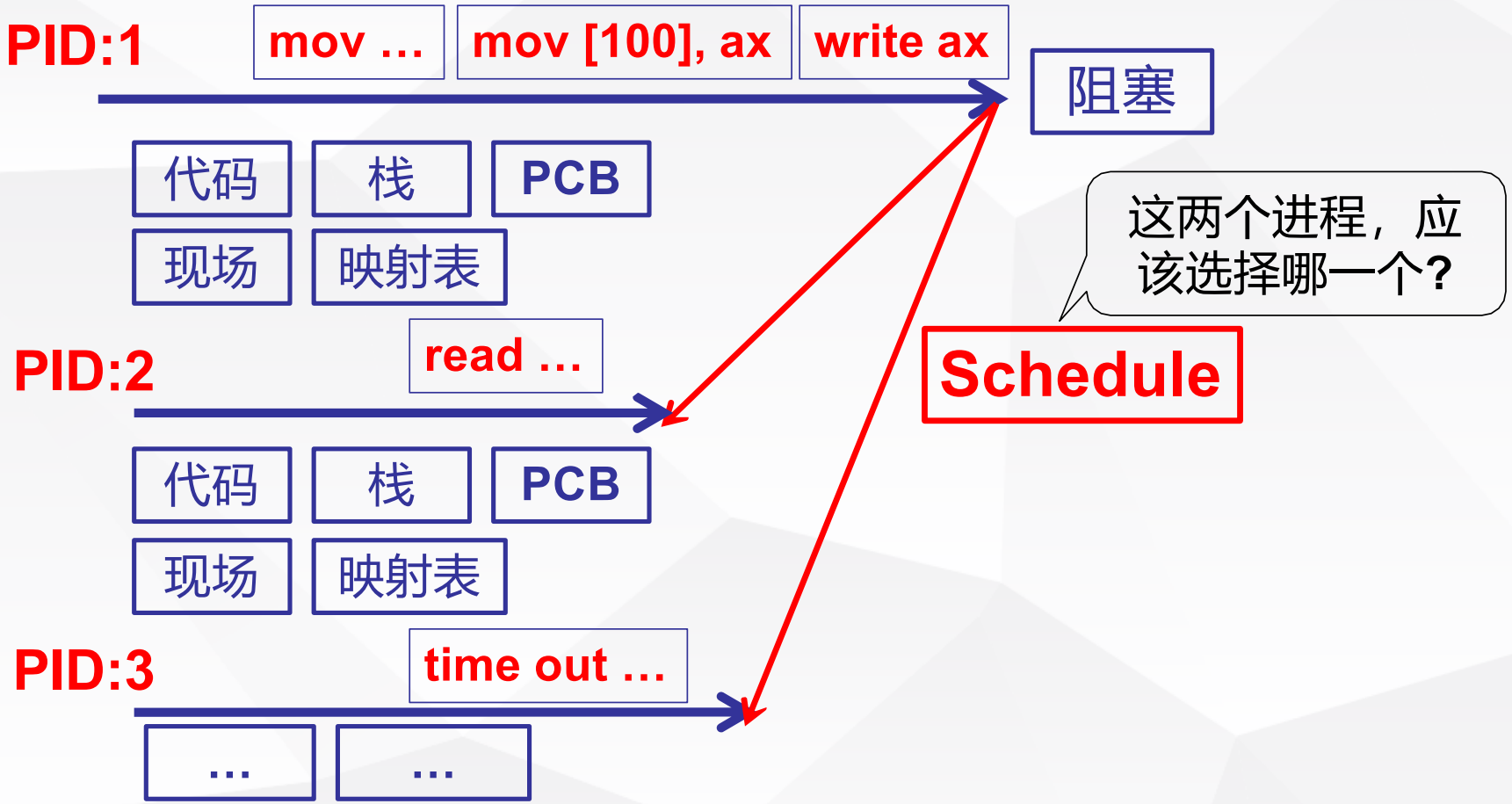
PCB：process control block 进程控制块

调度队列

- 进程在进入系统时， 会被加到作业队列，这个队列包括系统内的所有进程
  - 就绪队列：驻留在内存中、就绪的进程保存在就绪队列中
  - 设备队列：等待特定设备的进程列表



多进程->CPU调度


PID:1
mov …
mov [100], ax
write ax
阻塞
代码
栈
PCB
现场
映射表
这两个进程，应该选择哪一个?
Schedule
PID:2
read …
代码
栈
PCB
现场
映射表
PID:3
time out …
…
…

调度准则

- CPU利用率： 使CPU尽可能忙碌
- 吞吐量：一个时间单元内进程完成的数量
- 周转时间：从进程提交到进程完成的时间段称为周转时间。周转时间为所有时间段之和，包括等待进入内存、在就绪队列中等待、在CPU上执行和I/O执行
- 等待时间：在就绪队列中等待所花时间之和
- 响应时间：从提交请求到产生第一次响应的时间

最大化CPU利用率、吞吐量，最小化周转时间、等待时间和响应时间

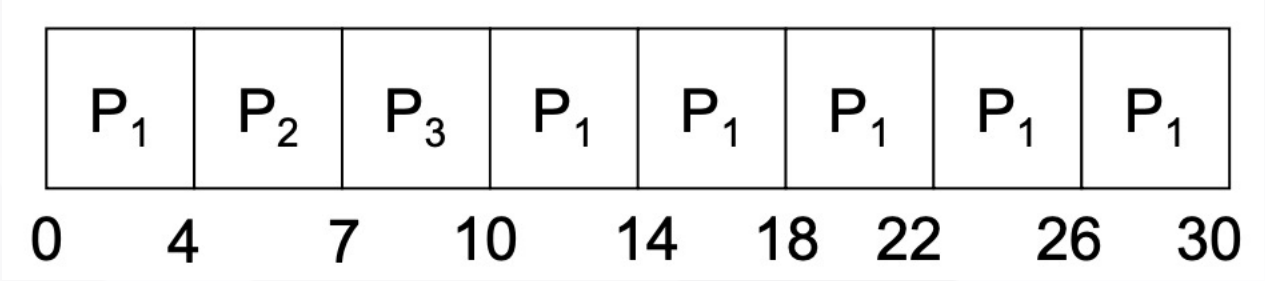
CPU调度算法

- 先到先服务调度 First-Come, First-Served (FCFS) Scheduling
- 最短作业优先调度 Shortest-Job-First (SJF) Scheduling
- 优先级调度 Priority Scheduling
- 轮转调度 Round-Robin Scheduling
- 多级队列调度 Multilevel Queue Scheduling
- 多级反馈队列调度 Multilevel Feedback Queue Scheduling

轮转调度（round-robin,RR）例子

Process	Burst Time
P_1	24
P_2	3
P_3	3

- 采用4ms的时间片，甘特图为：



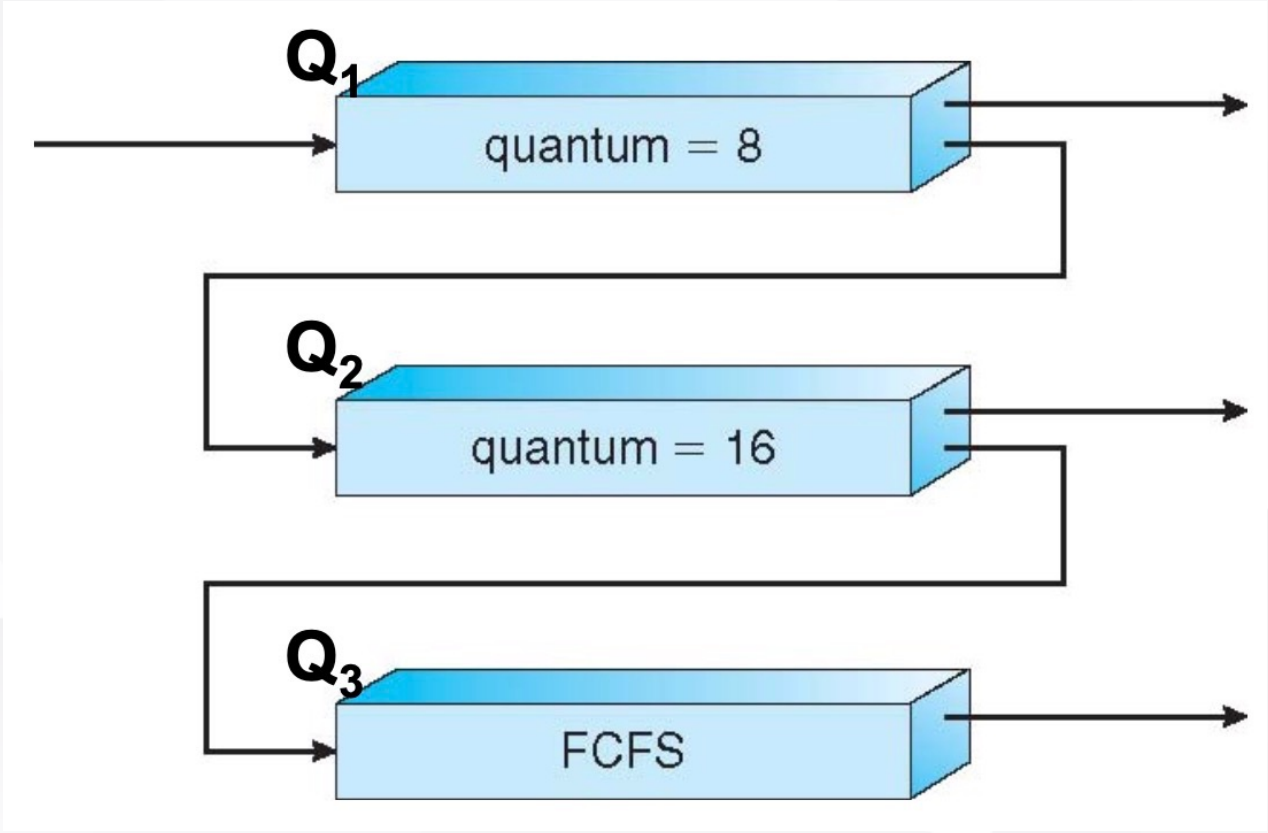
- 等待时间：P1等待10-4=6ms，P2等待4ms，P3等待7ms；平均等待时间为17/3=5.66ms

多级反馈队列调度

- 多级队列调度的灵活度较低：进程被永久分配到某个队列
- 多级反馈队列调度：允许进程在队列间迁移，在较低优先级队列中等待过长的进程会被移到更高优先级队列，阻止饥饿
- 由以下参数定义：
  - 队列数量
  - 每个队列的调度算法
  - 用以确定何时升级到更高优先级队列的方法
  - 用以确定何时降级到更低优先级队列的方法
  - 用以确定进程在需要服务时将会进入哪个队列的方法

多级反馈队列调度 例子

- 三个队列，从0到2
  - Q1-RR；Q2-RR；Q3-FCFS



多级反馈队列调度 例子

- Q1队列中的进程
  - 具有8ms的时间片
  - 若无法在8ms内完成，则被移动到Q2队列
- Q2队列中的进程
  - 当Q1空时执行，具有16ms的时间片
  - 若无法在16ms内完成，则被移动到Q3队列
- Q3队列中的进程
  - 当Q1、Q2均空时执行，采用FCFS调度

进程间的临界区问题

- 假设某个系统有n个进程
- 每个进程有一段代码，称为临界区
  - 进程在执行该区时可能修改公共变量、更新一个表、写一个文件
  - 当一个进程在临界区内执行时，其他进程不允许在它们的临界区内执行
- 临界区问题是设计一个协议以便协作进程
- 在进入临界区前， 每个进程应请求许可。实现这一请求的代码区段称为进入区。临界区之后可以有退出区。其他代码为剩余区

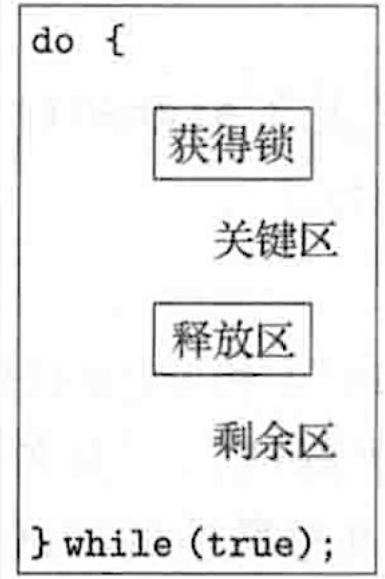
```
do {
        entry section          临界区
            critical section
        exit section           剩余区
            remainder section
} while (TRUE);
```

临界区问题的解决方案

- Peterson解决方案
- 硬件同步
- 互斥锁
- 信号量

互斥锁

- 采用硬件同步较复杂，不能让程序员直接使用
- 软件工具---互斥锁（mutex lock）
  - acquire()获得锁
  - release()释放锁



信号量

- 两个标准原子操作，用于修改S
  - wait()，最初被称为P()，把信号量减1
  - signal()，最初被称为V()，把信号量加1
- 信号量
  - 二进制信号量：0、1，类似于互斥锁
  - 计数信号量

信号量

- 核心：当执行操作wait()并发现信号量不为正时，不是忙等待，而是阻塞自己
- 定义进程链表

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

- wait()

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                sleep();
        }
}
```

- signal()

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0 && S->list != NULL) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

“第一”读者-作者问题的解答方法 （读者优先）

- 共享数据结构
  - wrt=1, readcount=0（有多少个读者），mutex=1
- 一个读者在wrt上等待，n-1个读者在mutex上等待

```
do {
        wait (wrt) ;

        //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

作者

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
            wait (wrt) ;
        signal (mutex)

        // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```

确保对readcount互斥保护

确保没有作者在写入

确保对readcount互斥保护

所有读者完成读操作释放锁

读者

“第二”读者-作者问题的解答方法（作者优先）

- 共享数据结构
  - int readcount = 0, writecount = 0;
  - mutexrc = 1, mutexwc = 1, wrt = 1, rd = 1;

作者

```
do {
    wait (mutexwc);
    writecount ++;
    if (writecount == 1)
        wait (rd);
    signal (mutexwc);

    wait (wrt);
    // writing is performed
    signal(wrt);

    wait (mutexwc);
    writecount - -;
    if (writecount == 0)
        signal (rd);
    signal (mutexwc);
} while (TRUE);
```

确保没有读者在读

```
do {
    wait (rd);
    wait (mutexrc);
    readcount ++;
    if (readcount == 1)
        wait (wrt);
    signal (mutexrc);
    signal (rd);

    //reading is performed

    wait (mutexrc);
    readcount - -;
    if (readcount == 0)
        signal (wrt);
    signal (mutexrc);
} while (TRUE);
```

增加了rd变量，只有当作者都被处理完，才会进入下面的wait(mutexrc)

读者

进程间的死锁问题

- 系统中有两个磁盘，进程P1和P2分别拥有一个，并需要另一个
- 信号量S和Q，均初始化为1

P_0	P_1
① wait (S);	② wait (Q);
③ wait (Q);	④ wait (S);

- 死锁：一系列被阻塞的进程持有部分资源，同时需要另一部分资源，这些资源被在该系列中的进程所持有
- 当死锁时，进程永远不能完成，系统资源被阻碍使用，以致于阻止了其他作业开始执行

死锁特征

- 如果一个系统中以下四个条件同时成立，就能引起死锁
  - 互斥：至少有一个资源必须处于非共享模式，即一次只能有一个进程可使用
  - 占有并等待：一个进程应占有至少一个资源，并等待另一个资源，而该资源为其他进程所占有
  - 非抢占：资源不能被抢占
  - 循环等待：有一组等待进程（P0-Pn），P0等待的资源被P1占有，P1等待的资源被P2占有，P2等待的资源被P3占有，...

死锁处理方法

- 处理死锁问题有三种方法
  - 通过协议来预防或避免死锁，确保系统不会进入死锁状态
  - 可以允许系统进入死锁状态，然后检测它，并加以恢复
  - 可以忽视这个问题，认为死锁不可能在系统内发生
- 死锁预防：确保至少有一个必要条件不成立，通过限制申请资源
- 死锁避免：通过事先得到有关进程申请资源和使用资源的额外信息，确定进程是否应该等待
- 死锁检测+恢复
- 死锁预防、避免等手段昂贵，可以通过重启计算机来恢复

死锁避免

- 死锁预防的方法是通过限制资源申请的方法来预防死锁，导致设备利用率、系统吞吐量低
- 死锁避免：获得额外信息，包括进程需要申请的资源、顺序等，系统从而可以决定，在每次请求时进程是否应等待以避免可能的死锁
  - 每个进程应声明可能需要的每种资源的最大数量
  - 死锁避免算法动态检查资源分配状态，以确保循环等待条件不能成立
  - 资源分配状态包括可用的资源、已分配的资源及进程的最大需求

安全状态

- 如果系统能按一定顺序为每个进程分配资源（不超过它的最大需求），仍然避免死锁，那么系统的状态就是安全的
- 只有存在一个安全序列<P1,P2,...,Pn>，系统才处于安全状态。对于每个进程Pi，Pi仍然可以申请的资源数小于当前可用资源+所有进程Pj（j<i）所占有的资源
  - 进程Pi所需要的资源即使不能立即可用，Pi可以等待知道所有Pj释放资源
  - 当Pj完成，Pi可以得到所需要的资源，完成任务，最后终止
  - 当Pi终止时，Pi+1可得到它所需要的资源，如此进行

银行家算法

- 前提条件
  - 适用于每个资源类型拥有多个实例的系统
  - 每个进程应声明可能需要的资源实例的最大数量
  - 当进程请求一个资源，就不得不等待
  - 当进程获得所有资源就必须在一段有限时间内释放它们
- 银行家算法试图寻找允许每个进程获得最大资源并结束的进程请求的一个理想执行序列，来决定状态是否安全
- 不存在这满足条件的执行序列的状态都是不安全的

银行家算法 实例

- 5个进程：P0-P4
- 3个资源类型：A（10个实例），B（5个实例），C（7个实例）
- T0时刻：

	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
P1	3 2 2	2 0 0	1 2 2	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

系统是否处于安全状态?

银行家算法 实例 P1请求（1 0 2）

	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	3 3 2
P1	3 2 2	2 0 0	1 2 2	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

执行安全算法，找到安全序列< P1->P3->P0->P2->P4 >

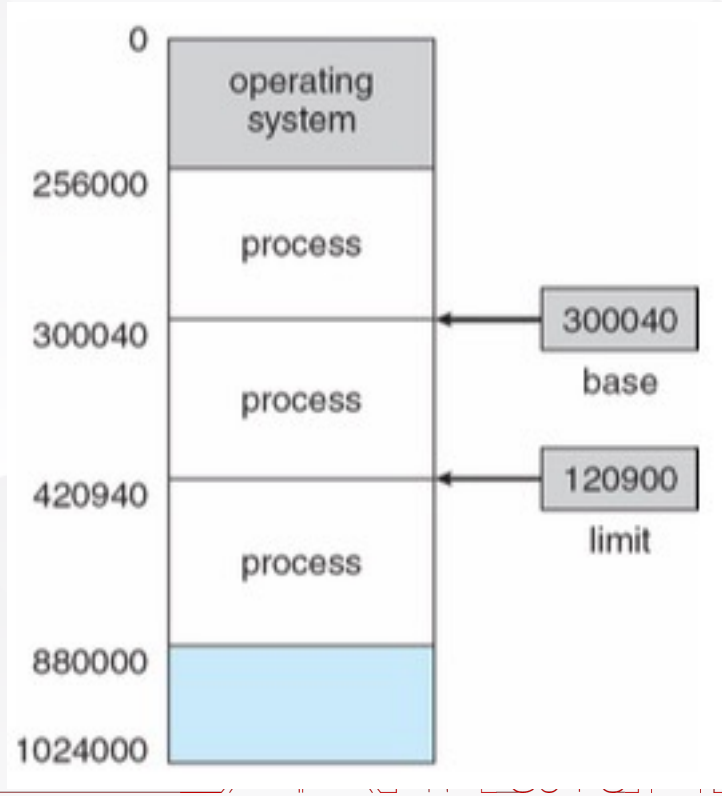
	最大需求	当前持有	需要	当前空闲
	A B C	A B C	A B C	A B C
P0	7 5 3	0 1 0	7 4 3	**2 3 0**
P1	3 2 2	**3 0 2**	**0 2 0**	
P2	9 0 2	3 0 2	6 0 0	
P3	2 2 2	2 1 1	0 1 1	
P4	4 3 3	0 0 2	4 3 1	

内存管理的背景

- 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
- 进程拥有自己的内存地址空间

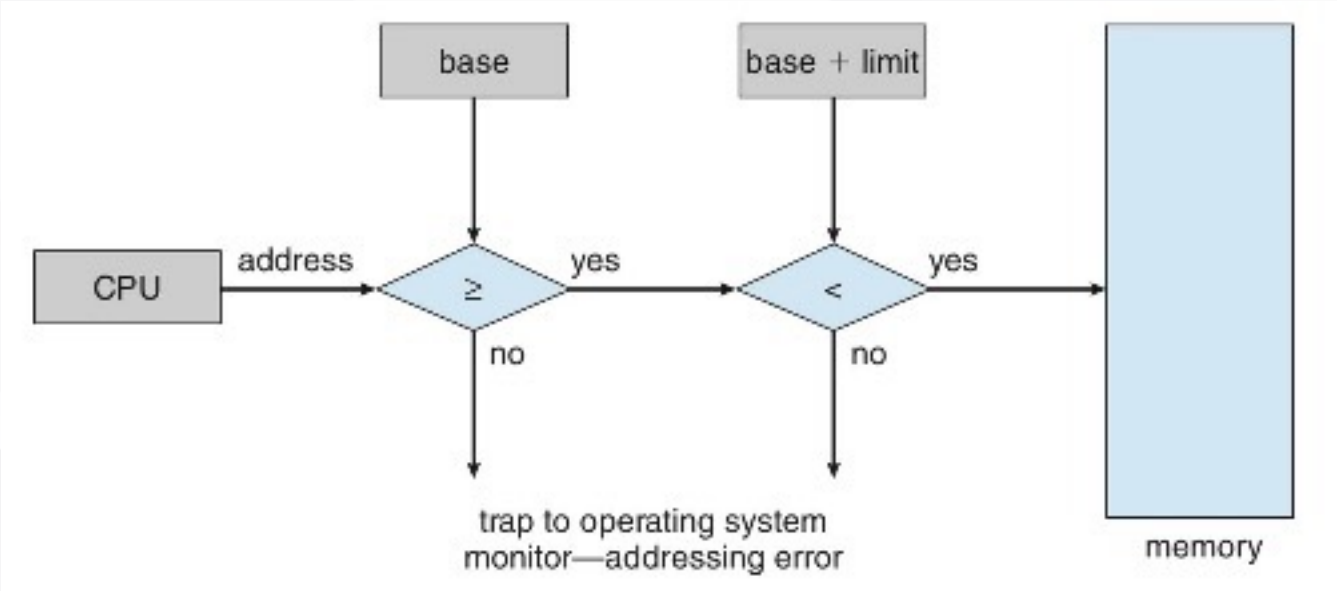
基地址与界限地址寄存器

- 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



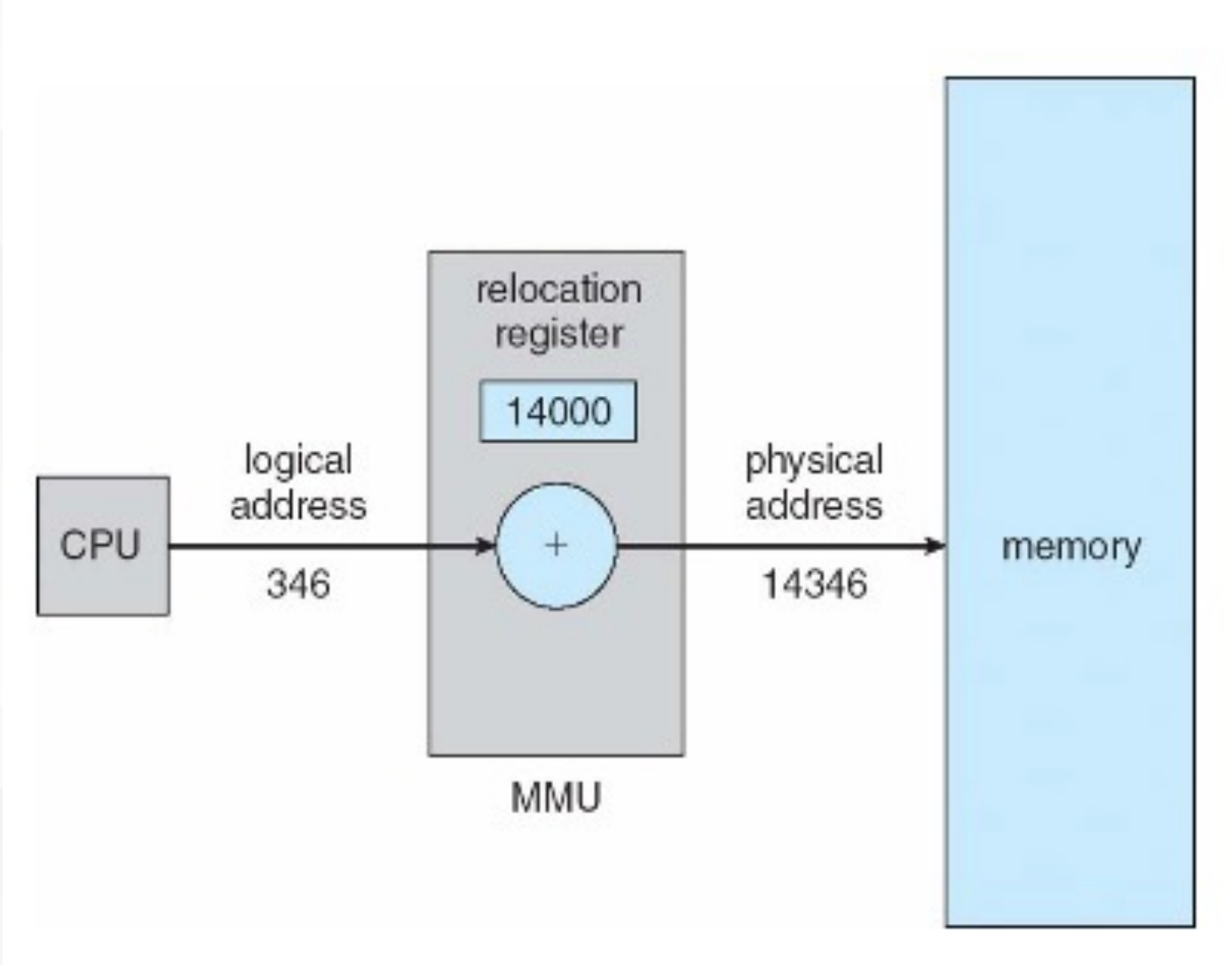
逻辑 vs 物理地址空间

- 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
- 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

内存管理单元

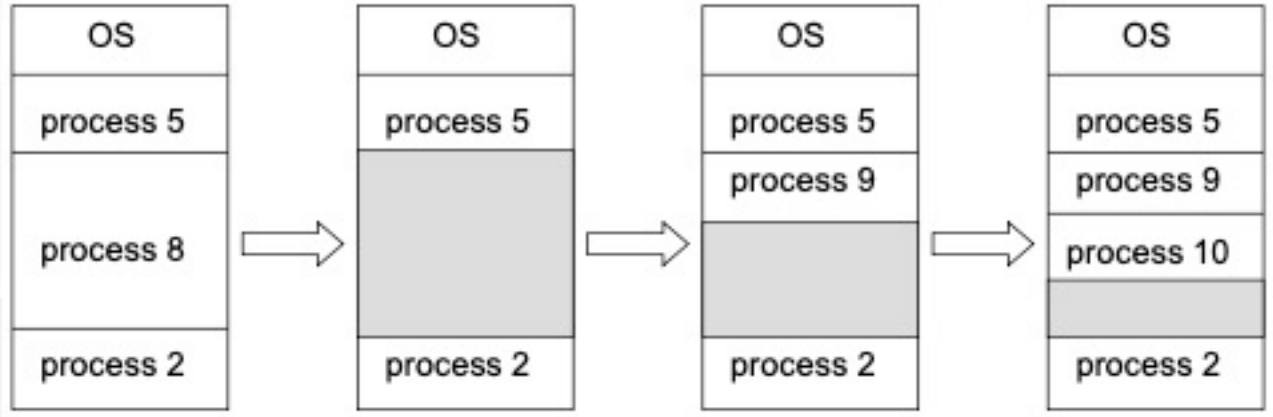
- 从虚拟地址到物理地址的运行时映射是由内存管理单元（MMU）的硬件设备来完成
- 有多种映射方法
- 最简单的MMU方案：重定位寄存器。将用户进程所生成的地址在交给内存前，加上重定位寄存器的值
- 用户程序不会看到真实物理地址

内存管理单元


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

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

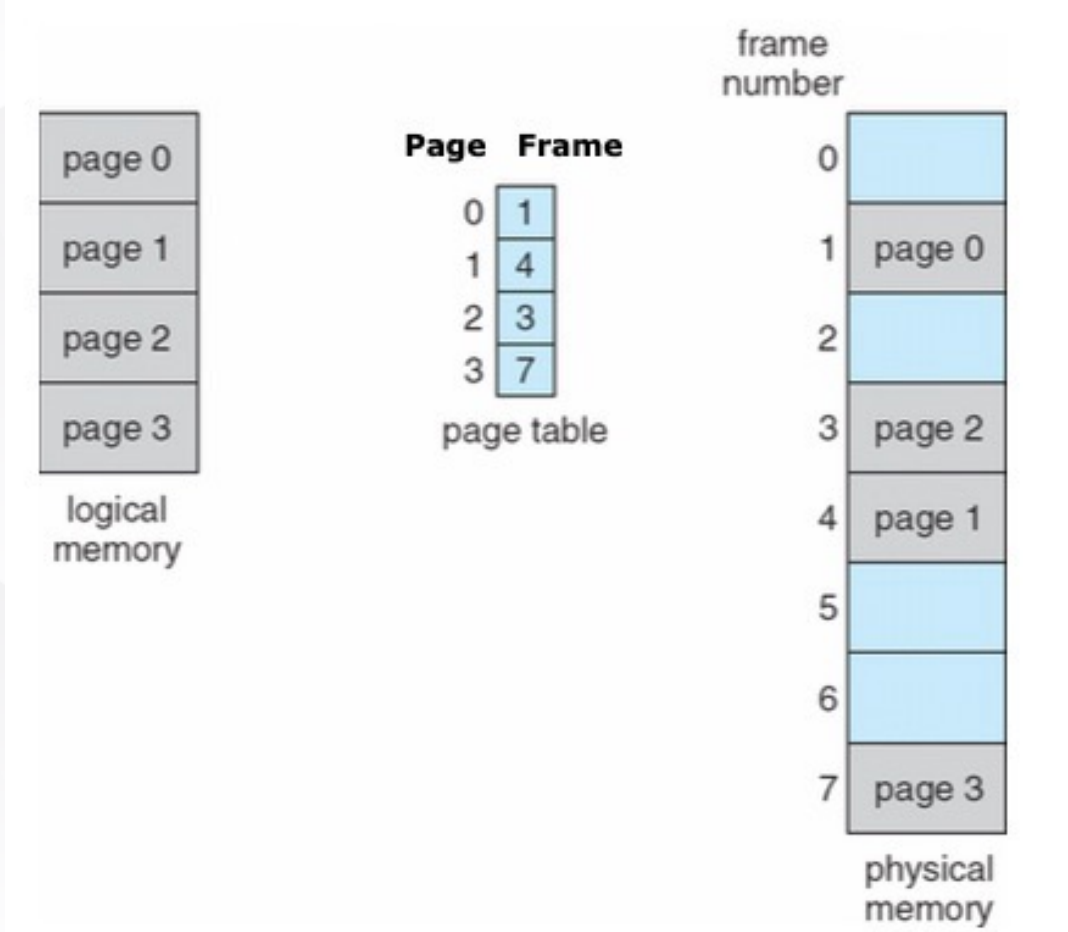
- 如何根据一组空闲孔来分配大小为n的请求?
  - 首次适应：分配首个足够大的孔
  - 最优适应：分配最小的足够大的孔。需要遍历整个列表，这种方法会产生最小剩余孔
  - 最差适应：分配最大的孔。需要遍历整个列表，这种方法会产生最大剩余孔

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

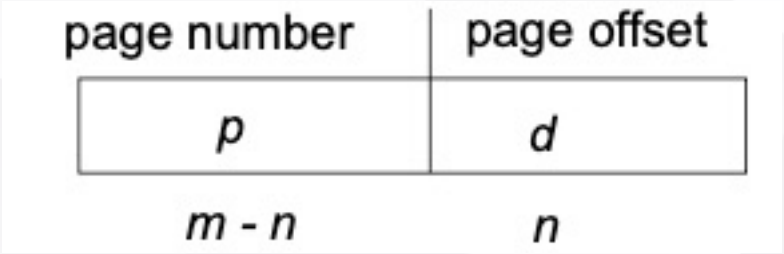
分页

- 针对每个段内存请求，切分成多页（逻辑、物理）
  - 一页：4K
- 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?

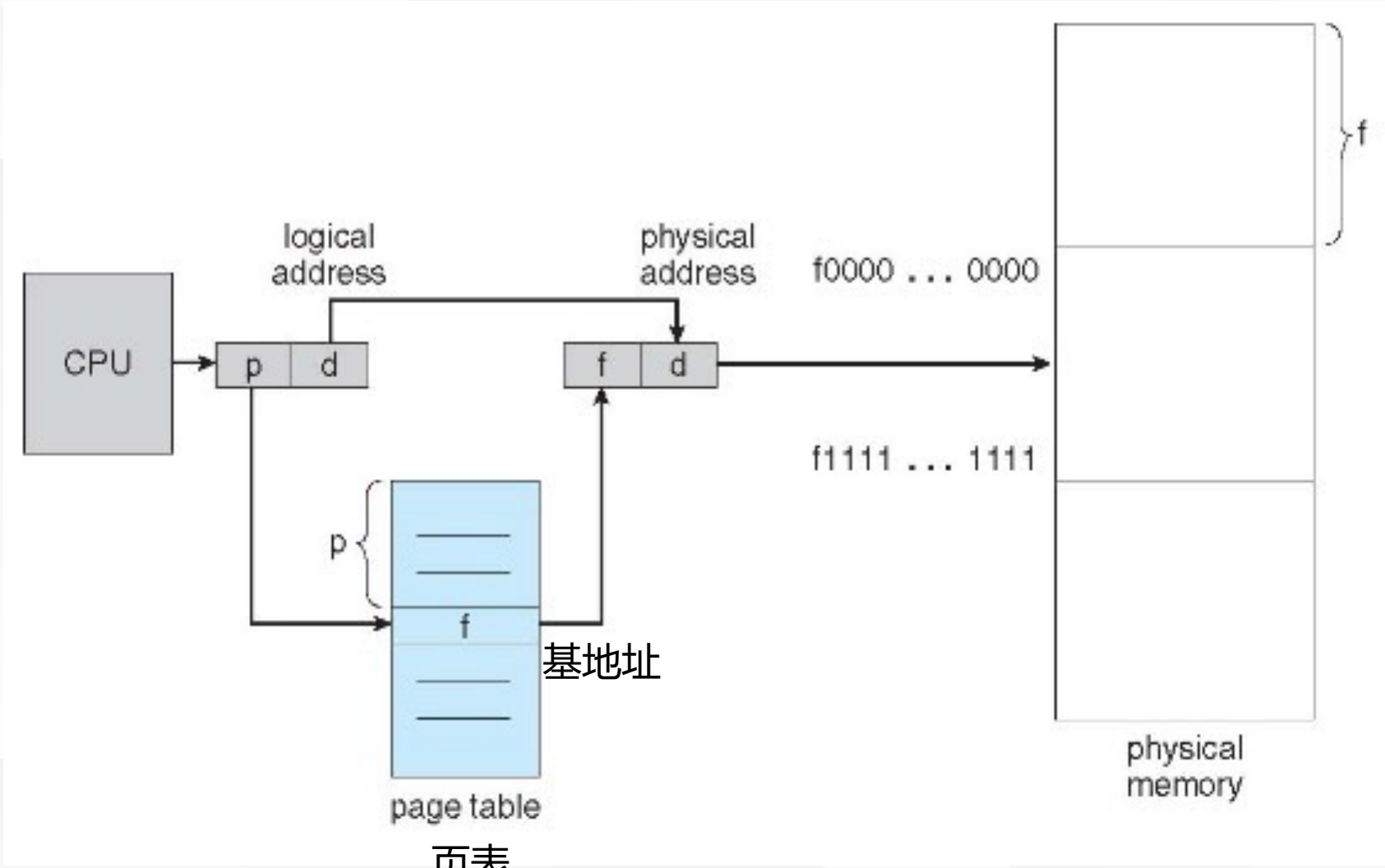


地址翻译

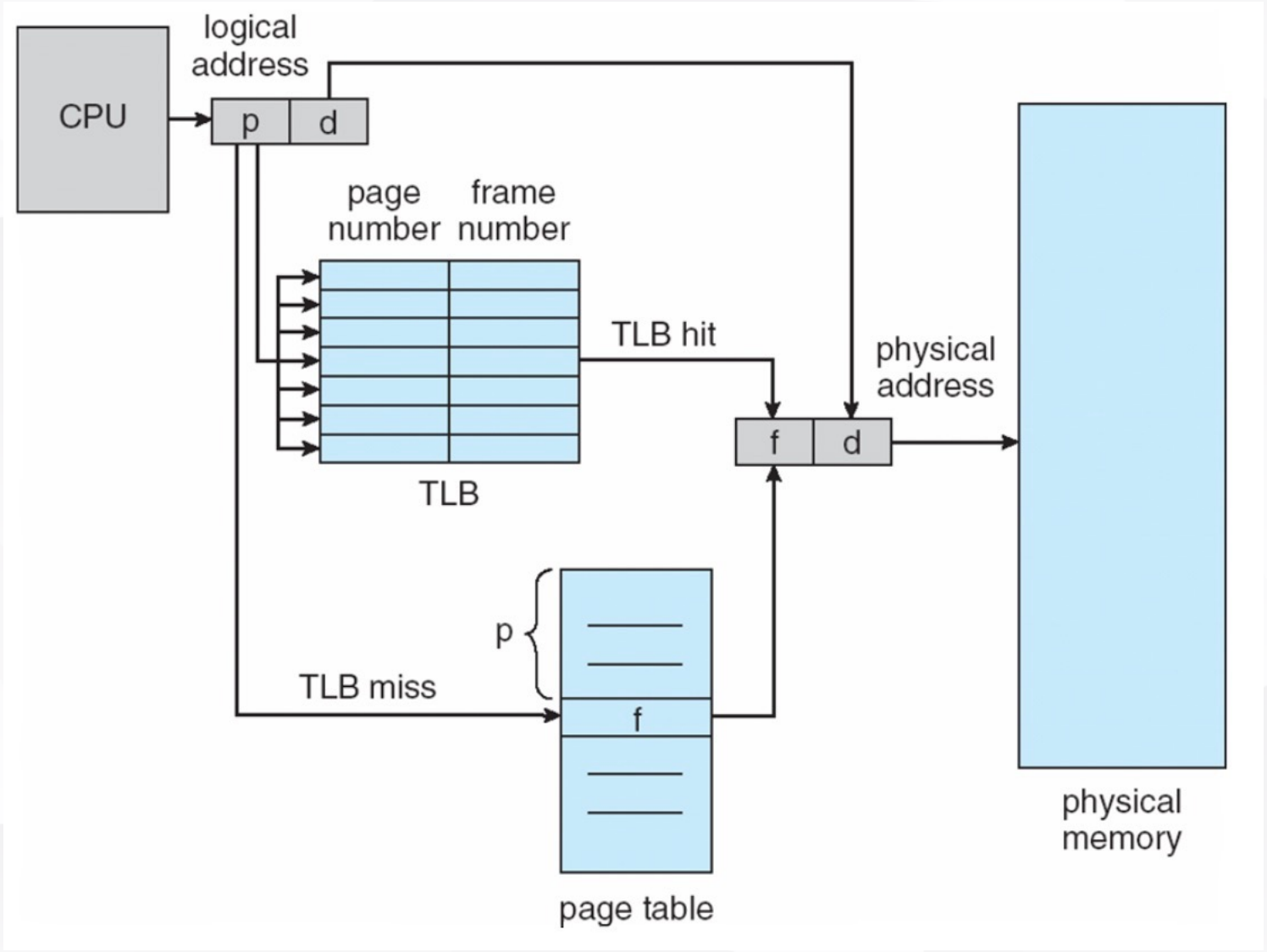
- 由CPU生成的**逻辑地址**分为两部分
  - 页码（page number，p）：作为页表的索引，包含了每页在物理内存中的基地址
  - 页偏移（page offset，d）：与基地址相结合形成物理内存地址，发送到内存单元，进行数据访存



分页的硬件支持

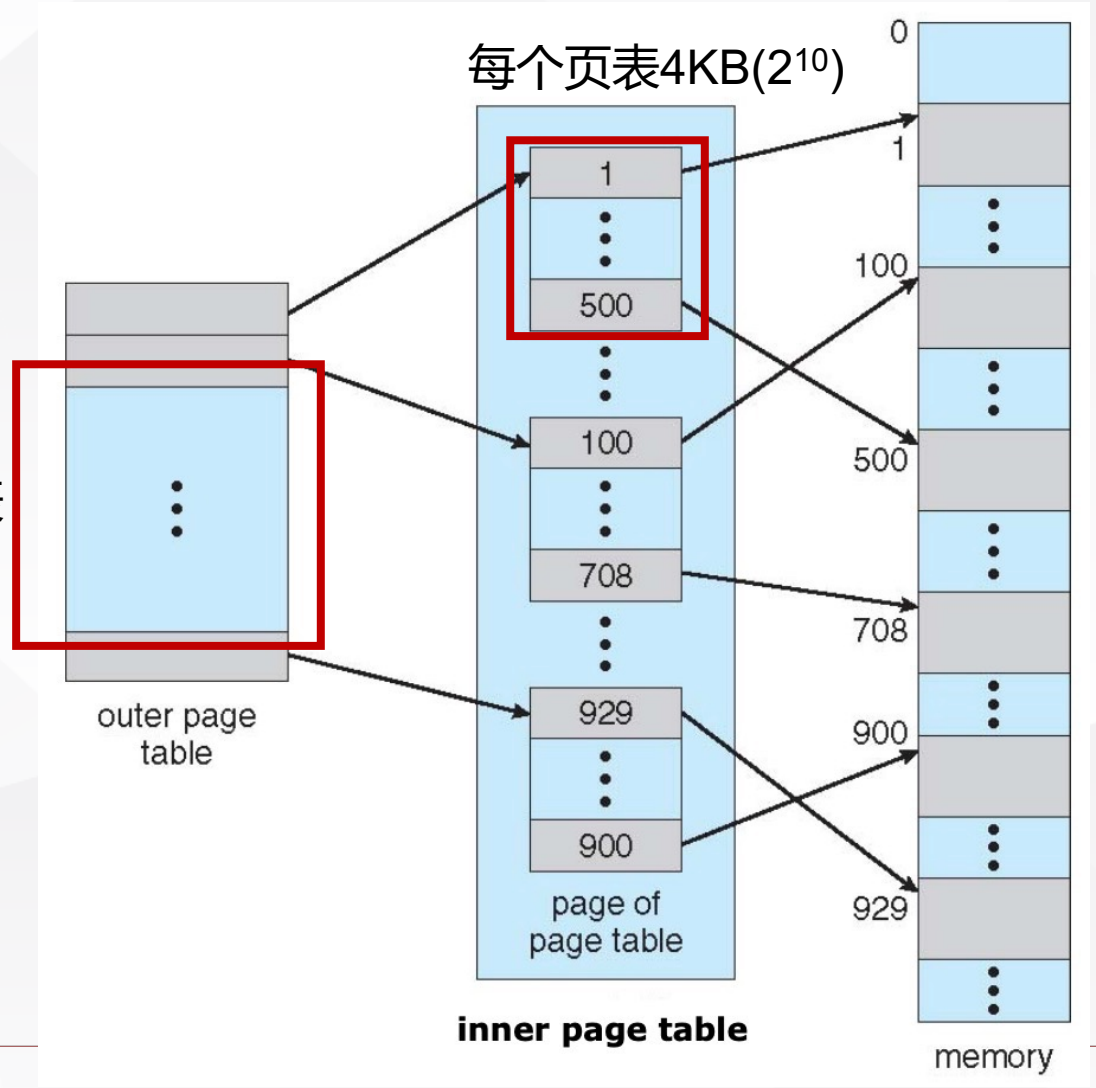

CPU
logical address
p
d
physical address
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
基地址
page table
页表
physical memory

分页的硬件支持：进阶TLB



分层页表

- 使用两层页表算法：第一级页表（章）+第二级页表（节）



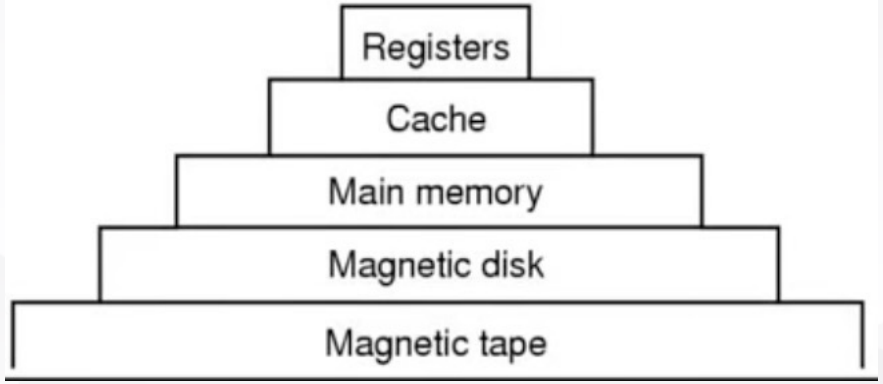
现在需要的页表共计
4KB+12KB=16KB<<4MB

虚拟内存的背景

- 程序很大，内存空间不够
- 金字塔型存储器层次结构
- 难以直接把程序放在硬盘上执行，速度太慢

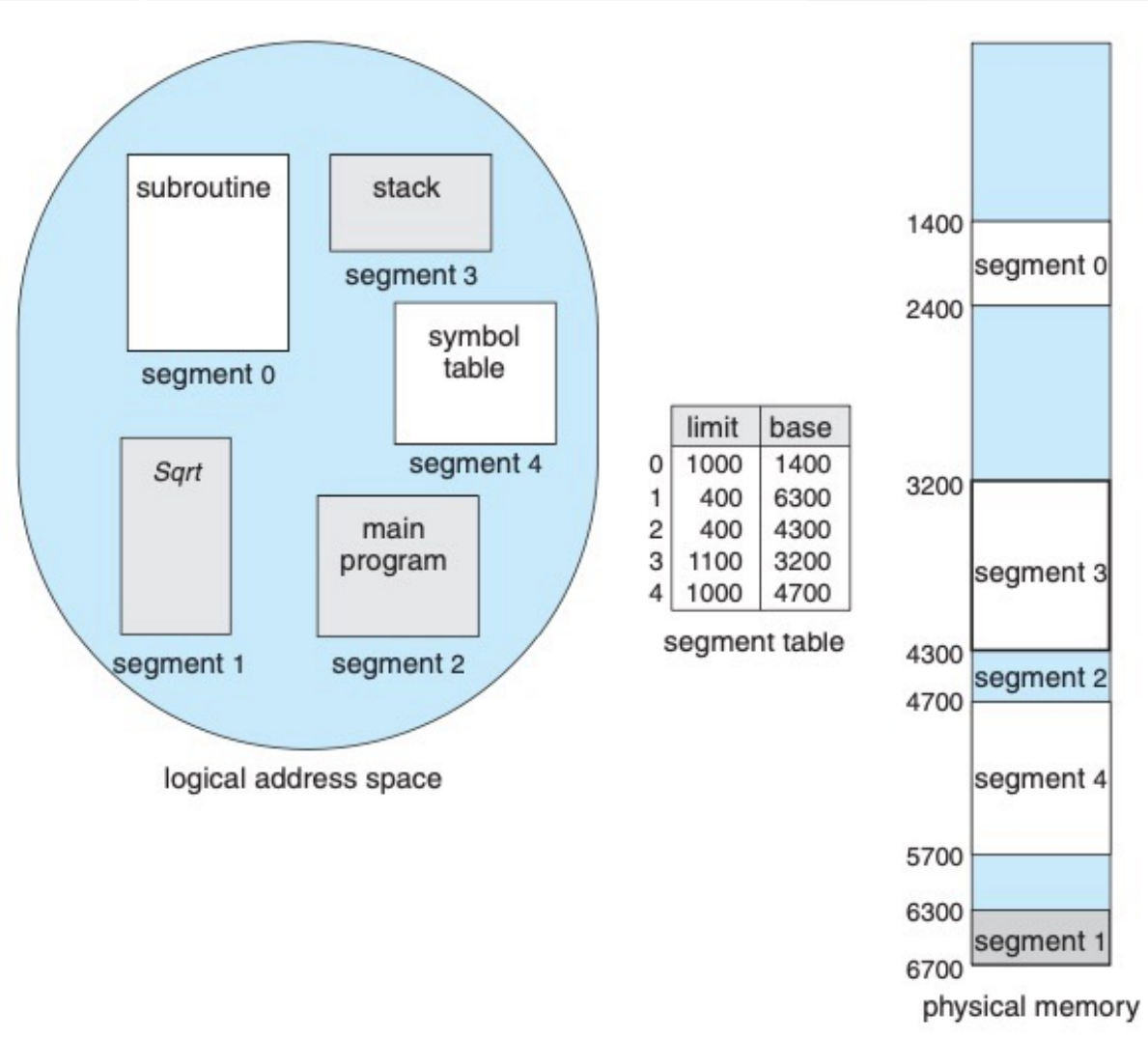
电脑游戏

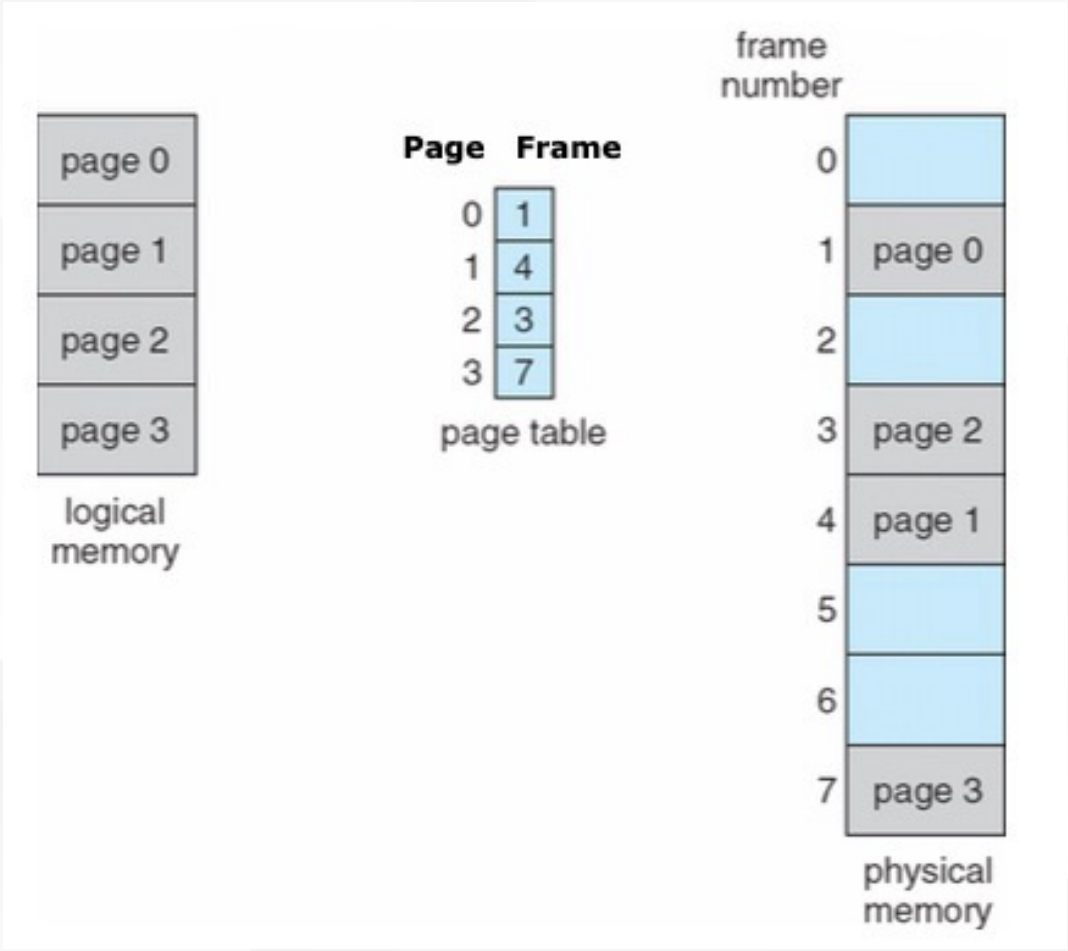
一代	二代	三代	四代	五代	六代	七代	八代
437K	883K	1.9M	6M	6.3M	59M	100M	138M



存储器层次结构

段页结合的虚拟内存

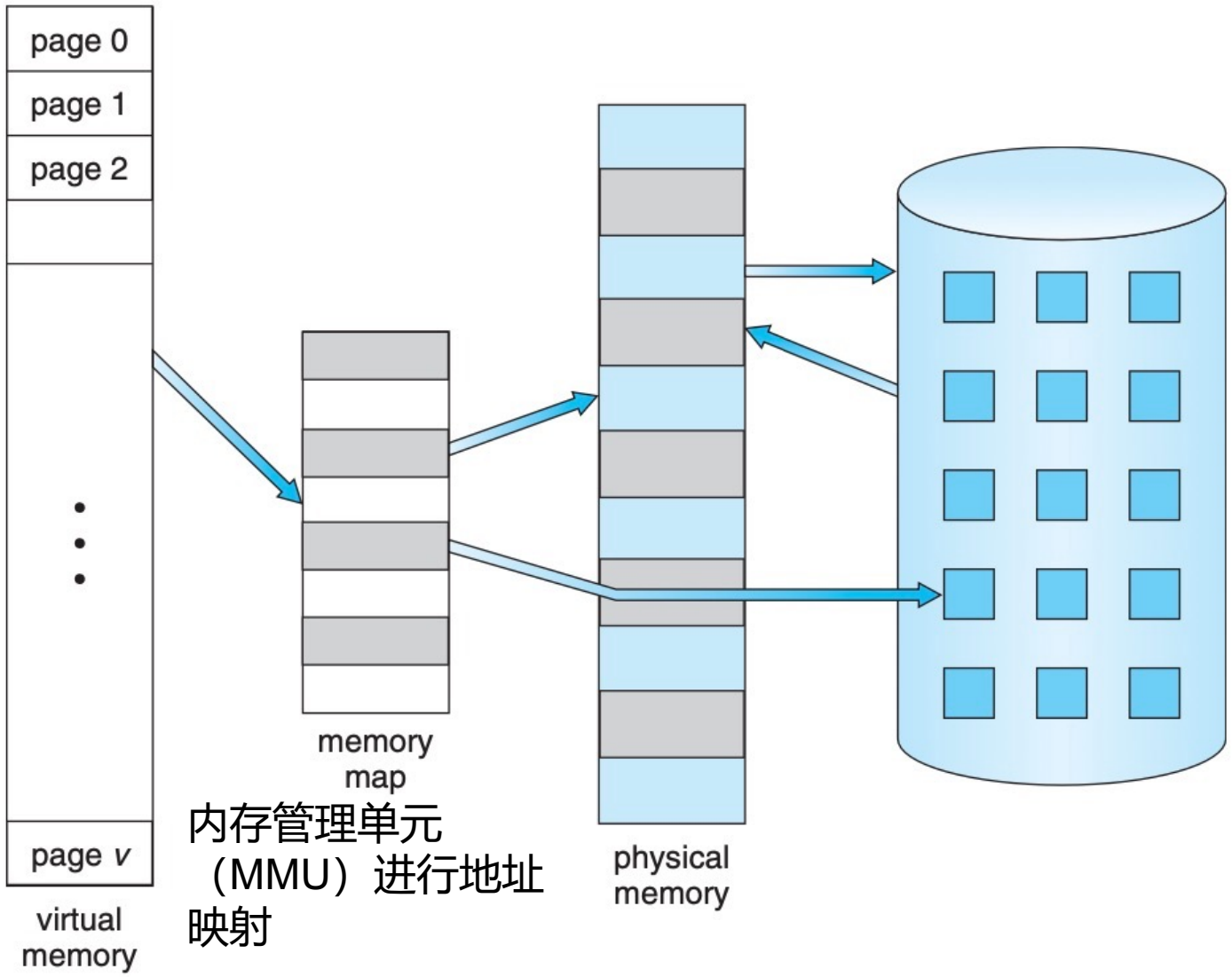




分段：从用户的角度将程序分段，存储到虚拟内存

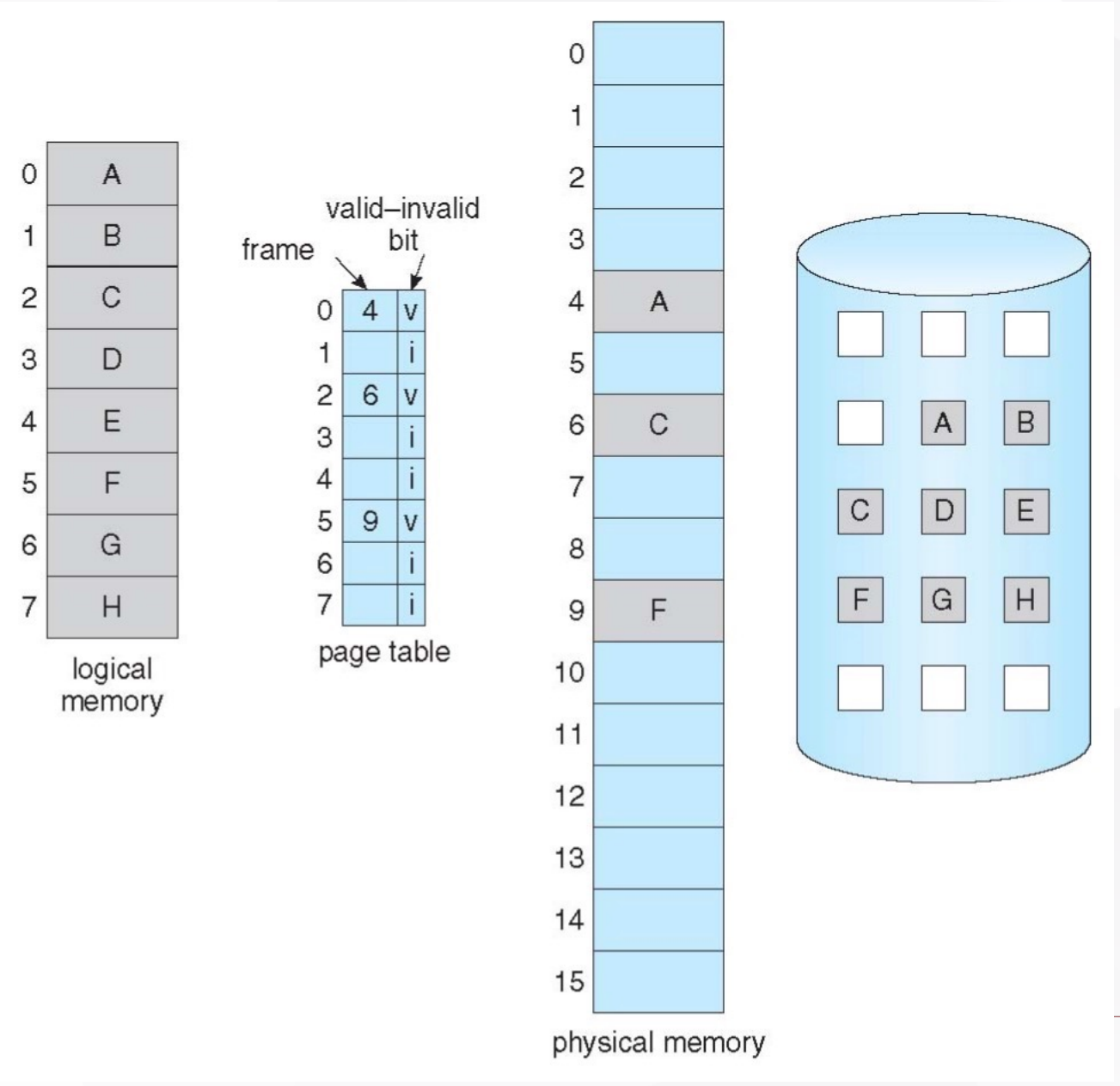
分页：从物理内存来看，将虚拟内存的段分页

虚拟内存大于物理内存



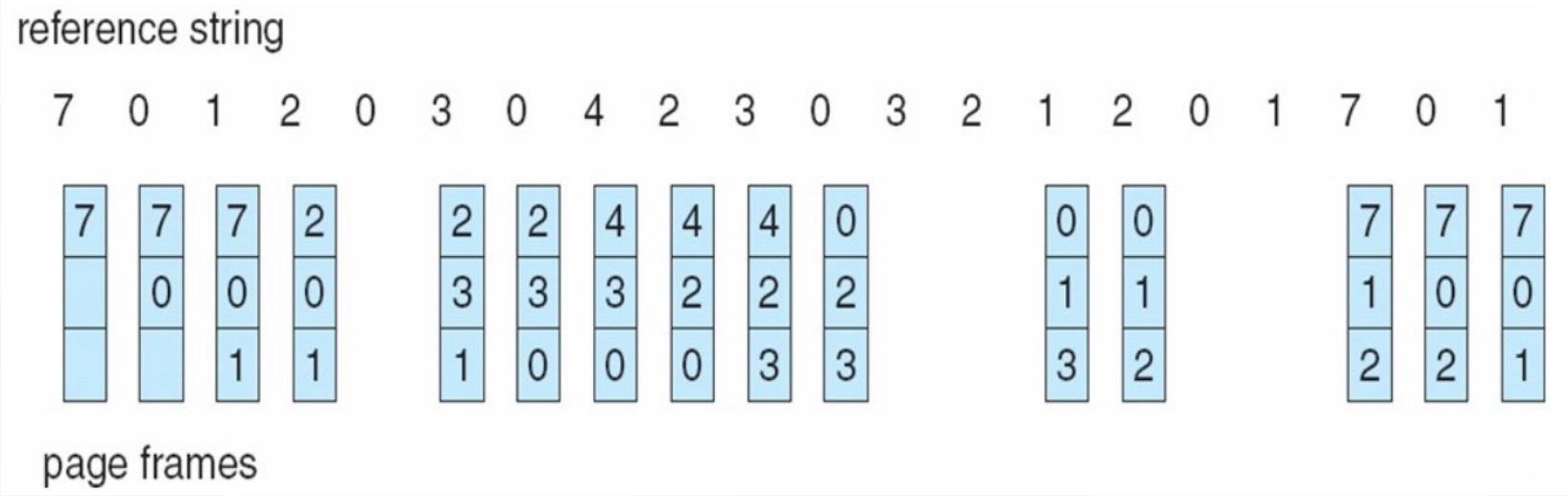
内存管理单元（MMU）进行地址映射

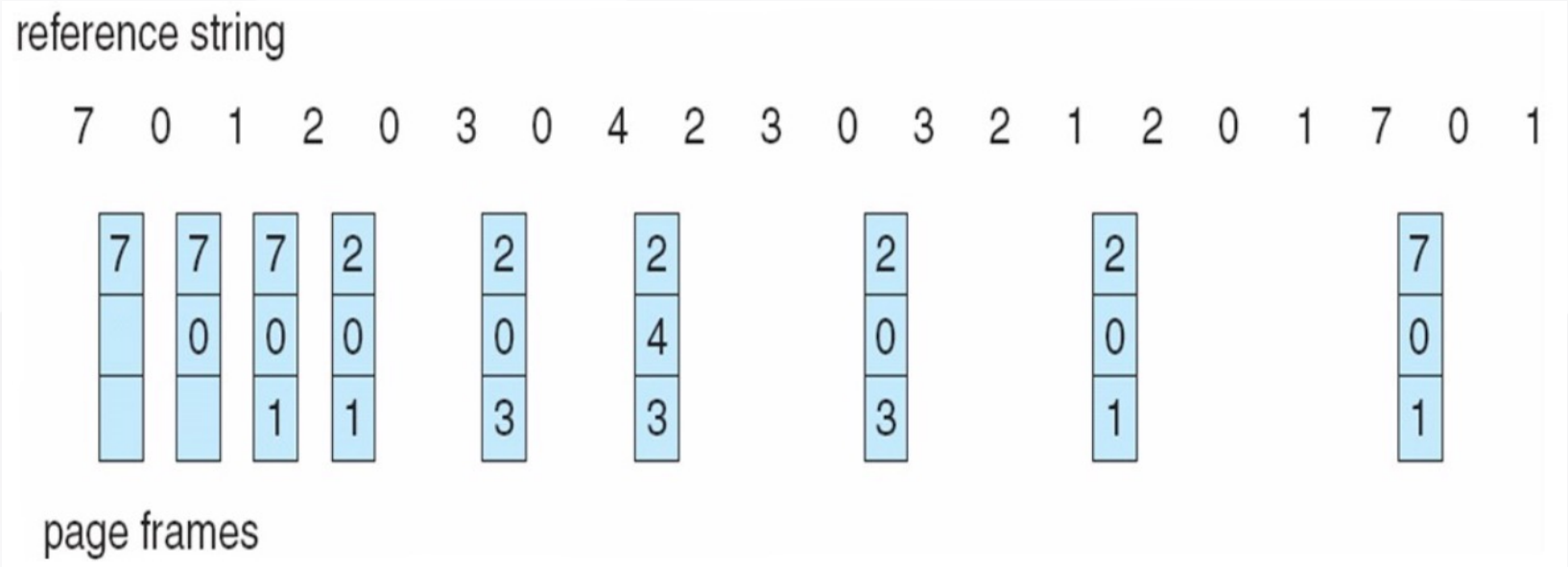
部分页面不在内存的页表



页面置换算法

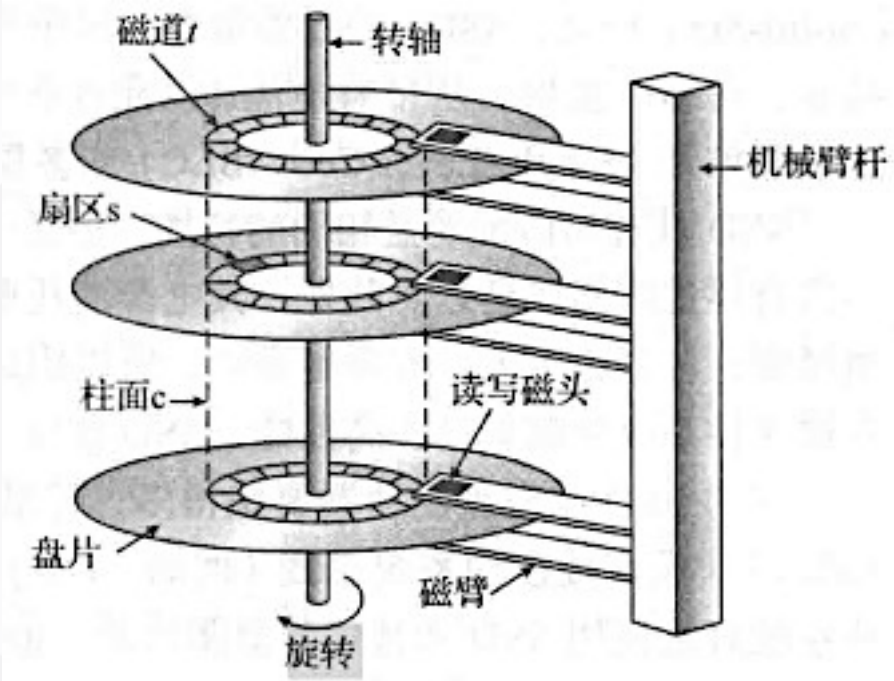
- FIFO页面置换
- 最优页面置换
  - 最长时间不会使用的页面
- LRU页面置换
  - 最近最少使用的页面





磁盘

- 磁盘用于存储数据
- 具有盘片，为图中平的圆状
- 普通盘片直径为1.8-3.5英寸
- 盘片两面都涂有磁质材料，通过在盘片上进行磁性记录可以保存信息
- 读写磁头在盘片上滑动，磁头附着在磁臂上，磁臂将所有磁头作为整体一起移动
- 盘片的表面逻辑地分成圆形磁道
- 磁道分为**扇区**，是读写的基本单位
- 同一磁臂位置的磁道集合形成了柱面

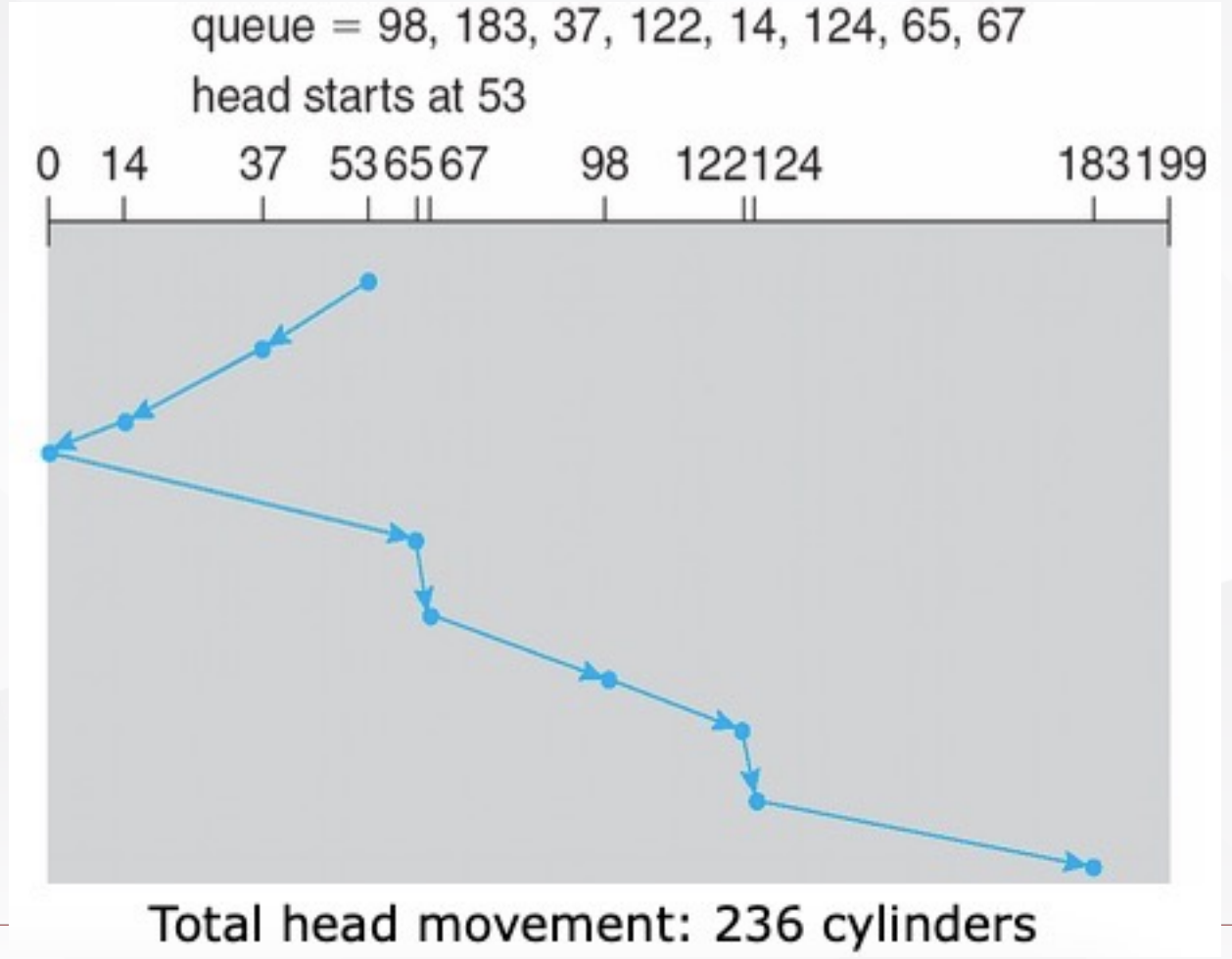


磁盘调度

- 磁盘队列中可能有多个待处理的请求，选择哪个？采用磁盘调度算法
  - 先来先服务(First-Come First Served, FCFS)算法
  - 最短寻道时间优先(Shortest-Seek- Time-First, SSTF) 算法
  - 扫描算法(SCAN algorithm)
  - 循环扫描(Circular SCAN,C-SCAN)调度
  - LOOK调度

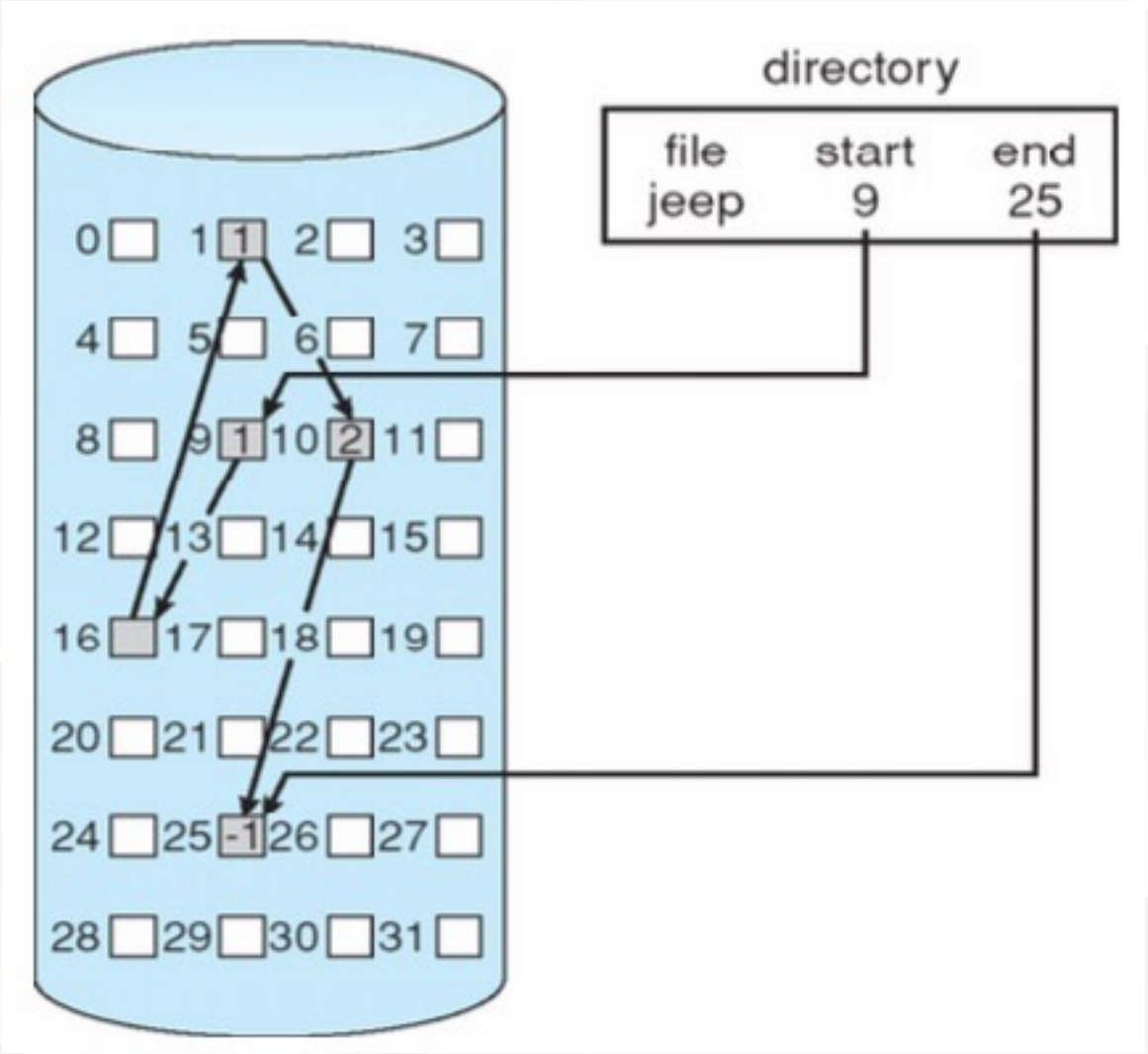
扫描算法(SCAN algorithm)

- 磁臂从磁盘的一端开始，向另一端移动；在移过每个柱面时，处理请求。当到达磁盘另一端时，开始反向移动



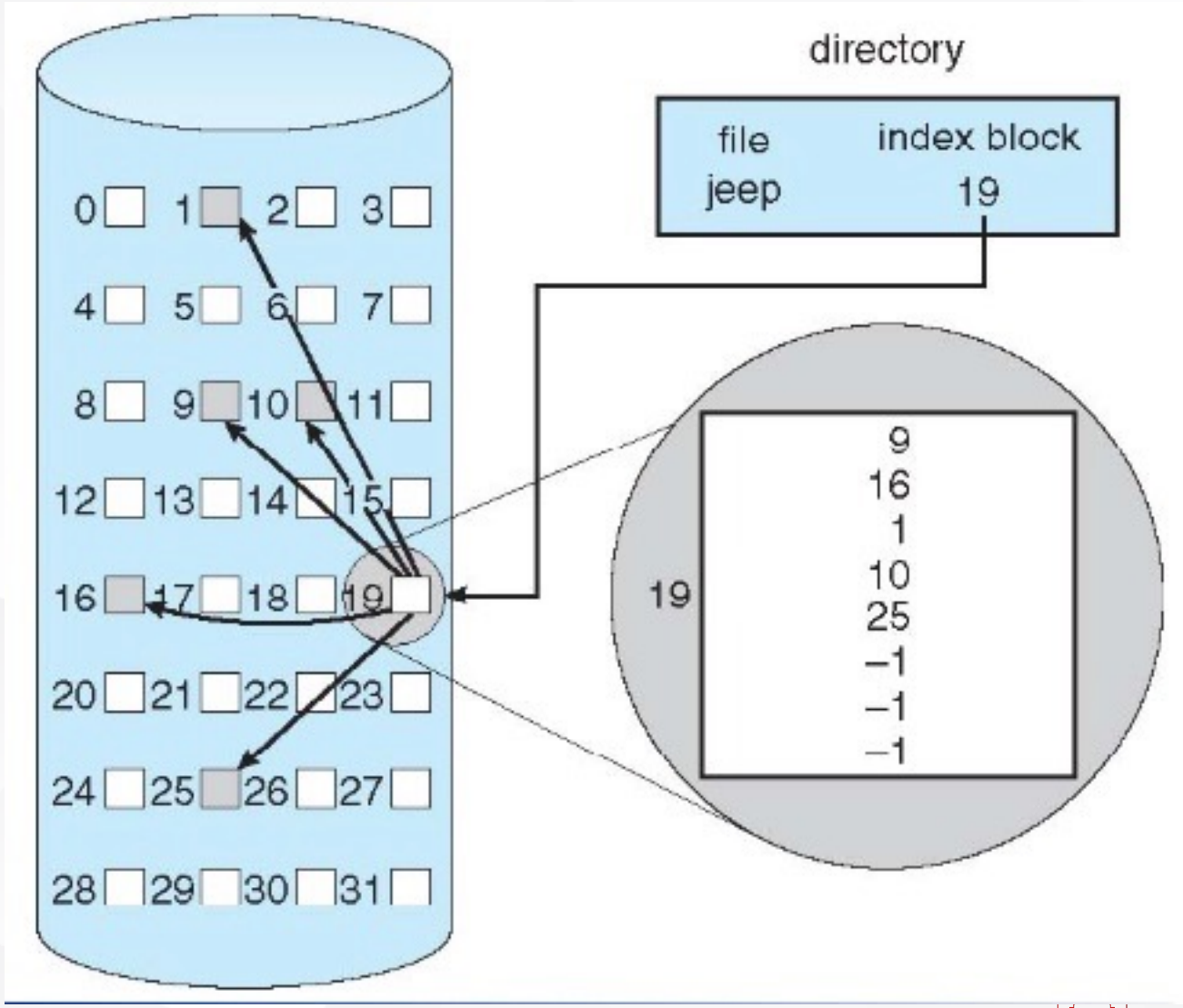
磁盘中文件的分配方法

- 分配方式
  - 连续分配
  - 链式分配
  - 索引分配
- 指标
  - 高效：如存储利用（外部碎片）
  - 表现：如访问速度



索引分配

- 每个文件都有自己的**索引块**，是一个磁盘块地址的数组
- 当创建文件时，索引块的所有指针设为NULL
- 当首次写入第i块，先从空闲空间管理器中获得一块，再将其磁盘地址写到索引块的第i个条目



索引分配-组合方案

- 组合方案，常见于早期的UNIX文件系统
  - 将索引块的前几个(如15) 指针存在文件的inode中
  - 这些指针的前12个指向直接块，即包含存储文件数据的块的地址
    - 小文件不需要多级索引
  - 剩余3个指针指向间接块
    - 第一个指向一级间接块
    - 第二个指向二级间接块
    - 第二个指向三级间接块

I/O硬件

- PCI总线（PCI bus）将处理器内存子系统连到快速设备
- 扩展总线（expansion bus）连接相对较慢的设备，如键盘、USB