Operating system

计算机系统概述

操作系统的基本概念

- 操作系统是控制、管理系统的硬软件资源，合理组织、调度资源并提供方便接口与环境的程序集合。

操作系统的特征

并发和共享是操作系统最基本的两个特征

- 并发：两个或多个事件在同一时间间隔内发生；
- 共享：系统的资源可以共多个并发的进程所使用，包括互斥访问以及同时访问；
- 虚拟：将物理上的实体变为逻辑上的虚拟物，从而对有限的资源进行逻辑上的扩展；
- 异步：进程实际上会以不可知的速度进行推进。

操作系统的目标和功能

i. 作为计算机系统资源的管理者
   - 处理机管理（进程的管理等）
   - 存储器管理
   - 文件管理
   - 设备管理

ii. 作为用户与计算机之间的接口
   - 命令接口
     - 联机命令接口：通过终端输入命令与系统交互
     - 脱机命令接口：写好一份作业的操作说明书之后交给系统控制，期间不能干预系统运行
   - 程序接口
     - 由系统调用（广义指令）组成

操作系统的发展历程

i. 批处理阶段

- 单道批处理系统：作业以脱机方式输入磁带，每次内存仅放一道作业，CPU等IO
  - 自动性
  - 顺序性
  - 单道性
- 多道批处理系统：资源利用率高，但不提供人机交互
  - 多道相互独立的程序同时运行
  - 宏观上并行
  - 微观上串行

ii. 分时操作系统

- 同时性：多个用户同时使用一台计算机
- 交互性：通过人机对话方式控制程序
- 独立性：用户独立地进行操作
- 及时性：用户的请求在短时间获得响应

iii. 实时操作系统

- 比分时操作系统更加注重处理程序的实时性而不是浪费时间在人机交互上
- 硬实时系统：严格限时
- 软实时系统：稍微放宽一点限制

iv. 网络操作系统以及分布式操作系统

v. 个人计算机操作系统

操作系统的运行环境

处理器运行模式

- 特权指令：不允许用户直接使用
- 非特权指令：允许用户直接使用
- 用户态请求系统服务时采用访管指令（非特权指令），切换到用户态（切换的指令是特权指令）、
- 时钟管理：计时提供标准系统时间+帮助进程切换
- 中断机制：IO、进程管理、设备驱动、文件访问都需要中断机制
- 原语：操作系统底层可被调用的公用小程序，具有原子性

中断和异常的概念

1. 中断：外中断，来自CPU执行指令外部的事件
    - 可屏蔽中断：通过改变屏蔽字可以实现多重中断，更灵活
    - 不可屏蔽中断：紧急硬件故障，必须一口气完成
2. 异常：内中断，来自CPU执行指令内部的事件

系统调用

1. 系统调用的功能
    - 设备管理
    - 文件管理
    - 进程控制
    - 进程通信
    - 内存管理
2. 系统调用过程
    - 参数压入堆栈
    - 下陷转入核心态
    - 保存程序执行现场
    - 分析调用类型并转入相应的子程序
    - 系统调用子程序执行结束后恢复被中断的CPU现场

操作系统结构

分层法

- 分层法是将操作系统分为若干层，底层(层0)为硬件，顶层(层N)为用户接口，每层只能调用紧邻它的低层的功能和服务(单向依赖)。

模块化

- 模块化是将操作系统按功能划分为若干具有一定独立性的模块。每个模块具有某方面的管理功能，并规定好各模块间的接口，使各模块之间能够通过接口进行通信。

宏内核

- 宏内核，也称单内核或大内核，是指将系统的主要功能模块都作为一个紧密联系的整体运行在核心态，从而为用户程序提供高性能的系统服务。因为各管理模块之间共享信息，能有效利用相互之间的有效特性，所以具有无可比拟的性能优势。

微内核

- 微内核构架，是指将内核中最基本的功能保留在内核，而将那些不需要在核心态执行的功能移到用户态执行，从而降低内核的设计复杂性。那些移出内核的操作系统代码根据分层的原则被划分成若干服务程序，它们的执行相互独立，交互则都借助于微内核进行通信。

外核

- 在底层，一种称为外核(exokernel)的程序在内核态中运行。它的任务是为虚拟机分配资源，并检查这些资源使用的安全性，以确保没有机器会使用他人的资源。每个用户的虚拟机可以运行自己的操作系统，但限制只能使用已经申请并且获得分配的那部分资源。

操作系统引导

操作系统引导是指计算机利用CPU运行特定程序，通过程序识别硬盘，识别硬盘分区，识别硬盘分区上的操作系统，最后通过程序启动操作系统，一环扣一环地完成上述过程。

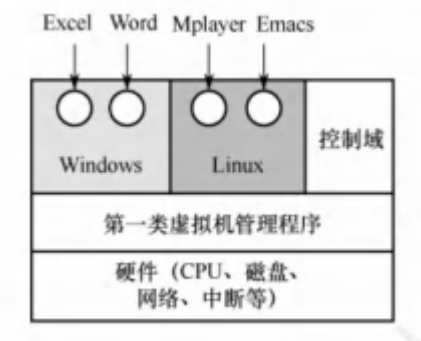
引导过程

- 激活CPU
- 硬件自检
- 加载带有操作系统的硬盘
- 加载主引导记录，作用是告诉CPU去硬盘的哪个分区找操作系统
- 扫描硬盘分区表
- 加载分区引导记录，作用是激活根目录下用于引导操作系统的程序
- 加载启动管理器
- 加载操作系统

虚拟机

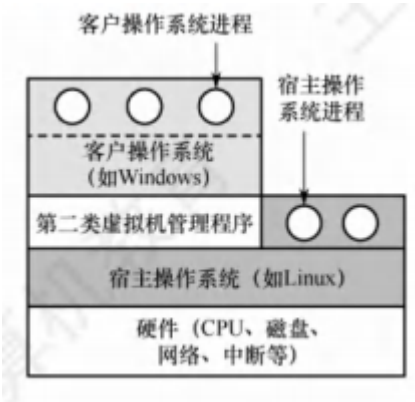
第一类虚拟机管理程序

- 第一类虚拟机管理程序就像一个操作系统，因为它是唯一一个运行在最高特权级的程序。它在裸机上运行并且具备多道程序功能。虚拟机管理程序向上层提供若干虚拟机，这些虚拟机是裸机硬件的精确复制品。由于每台虚拟机都与裸机相同，所以在不同的虚拟机上可以运行任何不同的操作系统。



第二类虚拟机管理程序

- 第二类虚拟机管理程序像一台刚启动的计算机那样运转，期望找到的驱动器可以是虚拟设备。然后将操作系统安装到虚拟磁盘上(其实只是宿主操作系统中的一个文件)。客户操作系统安装完成后，就能启动并运行。



进程与线程

进程 & 线程

进程的概念和特征

1. 进程的概念
   - 进程是进程实体的运行过程，是系统进行资源分配和调度的一个基本单位；
   - 程序段、相关数据段、PCB三部分构成了进程实体。
2. 进程的特征
   - 动态性：最基本的特征
   - 并发性：多个进程会共同存在于内存中
   - 独立性：进程是能独立运行、独立获得资源和独立接受调度的基本单位
   - 异步性：进程各自以不可预知的速度推进

进程的组成

1. 进程控制块

- 进程描述信息
- 进程控制和管理信息
- 资源分配清单
- 处理机相关信息

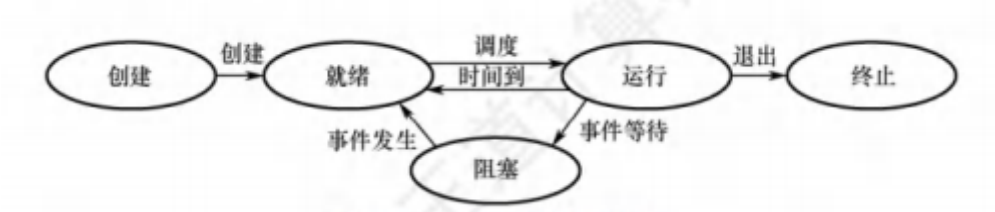
进程描述信息	进程控制和管理信息	资源分配清单	处理机相关信息
进程标识符（PID）	进程当前状态	代码段指针	通用寄存器值
用户标识符（UID）	进程优先级	数据段指针	地址寄存器值
	代码运行入口地址	堆栈段指针	控制寄存器值
	程序的外存地址	文件描述符	标志寄存器值
	进入内存时间	键盘	状态字
	CPU 占用时间	鼠标	
	信号量使用		

2. 程序段
   - 能被进程调度程序到CPU执行的程序代码段
   - 可以被共享
3. 数据段
   - 进程对应的程序加工的原始数据

进程的状态与转换

三种基本状态：就绪态、阻塞态、运行态

1. 运行态
2. 就绪态
3. 阻塞态
4. 创建态
5. 终止态



进程控制

1. 进程的创建

- 允许一个进程创建另一个进程，此时创建者称为父进程，被创建的进程称为子进程。子进程可以继承父进程所拥有的资源。当子进程被撤销时，应将其从父进程那里获得的资源还给父进程。此外，在撤销父进程时，通常也会同时撤销其所有的子进程。
- 创建进程时的操作：
  i. 为新进程分配一个唯一的进程标识号，并申请一个空白PCB(PCB是有限的)。若PCB申请失败，则创建失败。
  ii. 为进程分配其运行所需的资源，如内存、文件、I/O设备和CPU时间等(在PCB中体现)。这些资源或从操作系统获得，或仅从其父进程获得。如果资源不足(如内存),则并不是创建失败，而是处于创建态，等待内存资源。
  iii. 初始化PCB,主要包括初始化标志信息、初始化CPU状态信息和初始化CPU控制信息，以及设置进程的优先级等。
  iv. 若进程就绪队列能够接纳新进程，则将新进程插入就绪队列，等待被调度运行。

2. 进程的终止
   - 引起进程终止的事件：①正常结束，表示进程的任务已完成并准备退出运行；②异常结束，表示进程在运行时，发生了某种异常事件，使程序无法继续运行，如存储区越界、保护错、非法指令、特权指令错、运行超时、算术运算错、10故障等；③外界干预，指进程应外界的请求而终止运行，如操作员或操作系统干预、父进程请求和父进程终止。
   - 终止进程时的操作：
     i. 根据被终止进程的标识符，检索出该进程的PCB,从中读出该进程的状态。
     ii. 若被终止进程处于运行状态，立即终止该进程的执行，将CPU资源分配给其他进程。若该进程还有子孙进程，则通常需将其所有子孙进程终止(有些系统无此要求)。
     iii. 将该进程所拥有的全部资源，或归还给其父进程，或归还给操作系统。将该PCB从所在队列(链表)中删除。
3. 进程的阻塞和唤醒
   - 正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新任务可做等，进程便通过调用阻塞原语(Block),使自己由运行态变为阻塞态。可见，阻塞是进程自身的一种主动行为，也因此只有处于运行态的进程(获得CPU),才可能将其转为阻塞态。
   - 阻塞进程时的操作：
     i. 找到将要被阻塞进程的标识号(PID)对应的PCB。
     ii. 若该进程为运行态，则保护其现场，将其状态转为阻塞态，停止运行。将该PCB插入相应事件的等待队列，将CPU资源调度给其他就绪进程。

进程的通信

1. 共享存储
    - 通信的进程之间存在一块可直接访问的共享空间，通过对这片共享空间进行写/读操作实现进程之间的信息交换。在对共享空间进行写/读操作时，需要使用同步互斥工具(如P操作、V操作)对共享空间的写/读进行控制。
    - 共享存储又分为两种：低级方式的共享是基于数据结构的共享；高级方式的共享则是基于存储区的共享。操作系统只负责为通信进程提供可共享使用的存储空间和同步互斥工具，而数据交换则由用户自己安排读/写指令完成。
2. 消息传递
    - 直接通信方式：发送进程直接将消息发送给接收进程，并将它挂在接收进程的消息缓冲队列上，接收进程从消息缓冲队列中取得消息
    - 间接通信方式：发送进程将消息发送到某个中间实体，接收进程从中间实体取得消息。这种中间实体一般称为信箱。该通信方式广泛应用于计算机网络中。
3. 管道通信
    - 管道通信允许两个进程按生产者-消费者方式进行通信,只要管道不满，写进程就能向管道的一端写入数据；只要管道非空，读进程就能从管道的一端读出数据。
    - 管道机制必须提供三方面的协调能力：①互斥，指当一个进程对管道进行读/写操作时，其他进程必须等待。②同步，指写进程向管道写入一定数量的数据后，写进程阻塞，直到读进程取走数据后再将它唤醒；读进程将管道中的数据取空后，读进程阻塞，直到写进程将数据写入管道后才将其唤醒。③确定对方的存在。
    - 普通管道只允许单向通信，若要实现双向通信就要定义两个管道。

线程和多线程模型

引入线程的目的：提高资源利用率、更好地使多道程序并发执行、减小程序在并发执行时所付出的时空开销、提高操作系统的并发性能。

1. 线程的基本概念
    - 最直接的理解就是轻量级进程，引入线程后它变为CPU的基本执行单元
2. 线程与进程的比较
    - 调度：引入线程后，线程是调度的基本单位，线程切换的代价远低于进程；
    - 并发性：一个进程、不同进程中的多个线程可以并发执行；
    - 拥有资源：进程是拥有资源的基本单位；
    - 独立性：进程拥有独立的地址空间和资源，线程共享同一进程的地址空间和资源；
    - 系统开销：线程开销更小；

- 支持多处理器系统：对于传统单线程进程，不管有多少个CPU,进程只能运行在一个CPU上。对于多线程进程，可将进程中的多个线程分配到多个CPU上执行。

3. 线程的属性
   - 线程是一个轻型实体，它不拥有系统资源，但每个线程都应有一个唯一的标识符和一个线程控制块，线程控制块记录线程执行的寄存器和栈等现场状态
   - 不同的线程可以执行相同的程序
   - 同一进程中的各个线程共享该进程所拥有的资源
   - 线程是CPU的独立调度单位，多个线程是可以并发执行的
   - 一个线程被创建后，便开始了它的生命周期，直至终止。线程在生命周期内会经历阻塞态、就绪态和运行态等各种状态变化
4. 线程的状态与转换

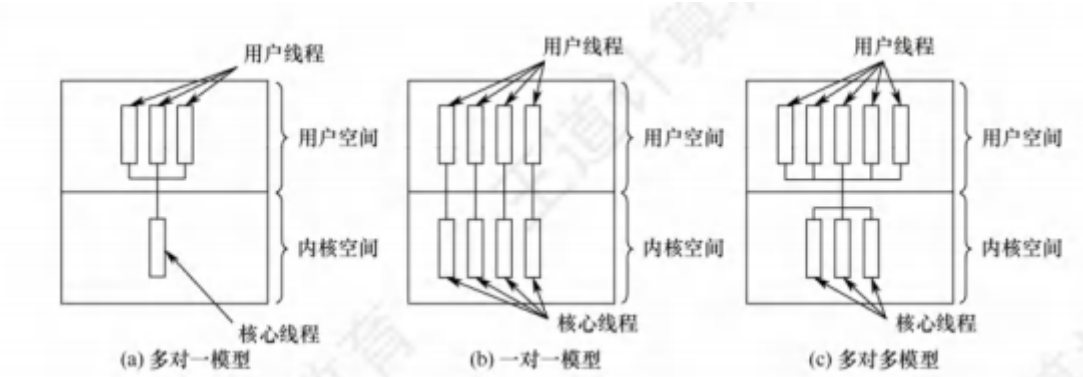
   三种基本状态：就绪态、阻塞态、运行态

   状态的转换与进程一样
5. 线程的组织与控制
   a. 线程控制块：线程标识符、寄存器标识符、运行状态、优先级、存储区
   b. 线程创建：用户程序启动时，通常仅有一个称为初始化线程的线程正在执行，其主要功能是用于创建新线程。在创建新线程时，需要利用一个线程创建函数，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小、线程优先级等。线程创建函数执行完后，将返回一个线程标识符。
   c. 线程终止：通常，线程被终止后并不立即释放它所占有的资源，只有当进程中的其他线程执行了分离函数后，被终止线程才与资源分离，此时的资源才能被其他线程利用。被终止但尚未释放资源的线程仍可被其他线程调用，以使被终止线程重新恢复运行。
6. 线程的实现方式
   a. 用户级线程（一对多）
      - 用户级线程被线程库在用户态管理
      - 内核意识不到用户线程的存在，应用程序通过使用线程设计成多线程程序
      - 调度仍然以进程为单位进行，但是线程切换不需要切换到内核空间，节省了模式切换的开销
      - 调度算法上，线程和进程可以使用不同的调用算法
      - 系统调用时一个线程的阻塞会导致同一个进程的所有线程同时阻塞
      - 不能发挥多CPU的优势，每次运行在一个CPU上的还是同一个进程，进程里面仍然只有一个线程能运行

b. 内核级线程（一对一）
   - 此时线程处于内核态，被进程所管理
   - 能发挥多CPU的优势，内核同时调度一个进程中的多个线程并行执行
   - 一个线程阻塞不影响其他线程，提高了系统的运行速度和效率
   - 但是进程的切换会转到核心态，导致系统开销较大

c. 组合方式（多对多）
   - 结合了以上两种方式的优点

7. 多线程模型



CPU调度

调度的概念

1. 调度的时机

   应该进行进程切换和调度的时机：
   - 创建新进程后
   - 进程正常结束或异常终止
   - IO请求
   - IO设备完成后

   不能进行进程切换和调度的时机：
   - 处理中断的过程
   - 需要完全屏蔽终端字的原子操作过程

2. 调度的方式
   - 非抢占方式
   - 抢占方式

3. 闲逛进程

没有其他进程可以调度的时候就会运行闲逛进程

4. 两种线程的调度
    - 用户级线程调度：由于内核并不知道线程的存在，所以内核还是和以前一样，选择一个进程，并给予时间控制。由进程中的调度程序决定哪个线程运行。
    - 内核级线程调度：内核选择一个特定线程运行，通常不用考虑该线程属于哪个进程。对被选择的线程赋予一个时间片，如果超过了时间片，就会强制挂起该线程。

调度的目标

调度的目标是为了优化以下几个量：

1. CPU利用率

$$\text{CPU的利用率} = \frac{\text{CPU有效工作时间}}{\text{CPU有效工作时间}+\text{CPU空闲等待时间}}$$

2. 系统吞吐量
3. 周转时间 = 作业完成时间 - 作业提交时间
4. 平均周转时间 = （n个作业的周转时间）/ n
5. 等待时间 = CPU处于等待的时间之和
6. 响应时间 = 用户提交作业到系统首次产生响应的时间

进程切换

1. 上下文切换
    - 挂起一个进程，将CPU上下文保存到PCB,包括程序计数器和其他寄存器
    - 将进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列
    - 选择另一个进程执行，并更新其PCB
    - 恢复新进程的CPU上下文
    - 跳转到新进程PCB中的程序计数器所指向的位置执行

典型的调度算法

1. 先来先服务调度算法（FCFS）
    - 属于不可剥夺算法
    - 算法简单但效率低
    - 对长作业有利，对短作业不利
    - 有利于CPU繁忙型作业，不利于IO繁忙型作业

2. 短作业优先调度算法（SJF）
    - 对长作业不利
    - 不一定能真正做到短作业优先调度，因为作业长短是估计出来的
    - 拥有最短平均等待时间和平均周转时间
    - 如果允许抢占，就变成最短剩余时间调度算法
3. 高响应比优先调度算法
    - 响应比 =（等待时间 + 要求服务时间）/ 要求服务时间
    - 每次要调度进程时将进程的响应比进行计算并排序，越高越有机会被调度
4. 优先级调度算法
    - 非抢占式优先级调度算法
    - 抢占式优先级调度算法
    - 静态优先级
    - 动态优先级
    - 优先级设置：系统进程>用户进程；交互性进程>非交互性进程
5. 时间片轮转调度算法（RR）
    - 适用于分时系统
    - 系统将所有的就绪进程按FCFS策略排成一个就绪队列
    - 系统可设置每隔一定的时间(如30ms)便产生一次时钟中断，激活调度程序进行调度，将CPU分配给就绪队列的队首进程，并令其执行一个时间片
    - 在执行完一个时间片后，即使进程并未运行完成，它也必须释放出(被剥夺)CPU给就绪队列的新队首进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行
6. 多级队列调度算法
    - 该算法在系统中设置多个就绪队列，将不同类型或性质的进程固定分配到不同的就绪队列
    - 每个队列可实施不同的调度算法
    - 同一队列中的进程可以设置不同的优先级，不同的队列本身也可以设置不同的优先级
7. 多级反馈队列调度算法
    - 设置多个就绪队列，并为每个队列赋予不同的优先级，队列的优先级逐个降低
    - 在优先级越高的队列中每个进程的时间片越小

- 每个队列都采用FCFS算法。新进程进入内存后，首先将它放入第1级队列的末尾，按FCFS原则等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统。若它在一个时间片结束时尚未完成，调度程序将其转入第2级队列的末尾等待调度；若它在第2级队列中运行一个时间片后仍未完成，再将它放入第3级队列，以此类推。当进程最后被降到第n级队列后，在第n级队列中便采用时间片轮转方式运行
- 按队列优先级调度，仅当第1～i-1级队列均为空时，才会调度第i级队列中的进程运行。若CPU正在执行第i级队列中的某个进程时，又有新进程进入任何一个优先级较高的队列，此时须立即将正在运行的进程放回到第i级队列的末尾，而将CPU分配给新到的高优先级进程

	先来先服务	短作业优先	高响应比优先	时间片轮转	多级反馈队列
能否可抢占	否	可以	可以	可以	队列内算法不一定
优点	公平，实现简单	平均等待时间、平均周转时间最优	兼顾长短作业	兼顾长短作业	兼顾长短作业，有较好的响应时间，可行性强
缺点	不利于短作业	长作业会饥饿，估计时间不易确定	计算响应比的开销大	平均等待时间较长，上下文切换浪费时间	最复杂
适用于	无	批处理系统	无	分时系统	相当通用

同步与互斥

同步与互斥的基本概念

- 临界资源：一次仅允许一个进程使用的资源
- 同步：为完成某种任务而建立的两个或多个进程，这些进程因为需要协调它们的运行次序而等待
- 互斥：当一个进程进入临界区使用临界资源时，另一个进程必须等待
- 实现互斥的准则：空闲让进、忙则等待、优先等待、让权等待

实现临界区互斥的基本方法

1. 软件实现方法
   - Peterson算法

```
进程 P0:                                进程 P1:
flag[0]=true;                           flag[1]=true;                   //进入区
turn=1;                                 turn=0;                         //进入区
while(flag[1]&&turn==1);                while(flag[0]&&turn==0);        //进入区
critical section;                       critical section;               //临界区
flag[0]=false;                          flag[1]=false;                  //退出区
remainder section;                      remainder section;              //剩余区
```

2. 硬件实现方法
   - 中断屏蔽方法

        - 当一个进程正在执行临界区代码时进行关中断
        - 限制CPU进行交替执行程序的能力
        - 关中断权限不应该赋予用户态进程
        - 不适用于多处理器系统，没有防止其他CPU执行临界区代码的能力
    - 硬件指令方法
        - TAS指令：读出标志后将标志设为True并返回旧值，当且仅当旧值为False表示能获得锁
        - Swap指令：交换两个字节的内容，当且仅当将False换成True表示能获得锁

互斥锁

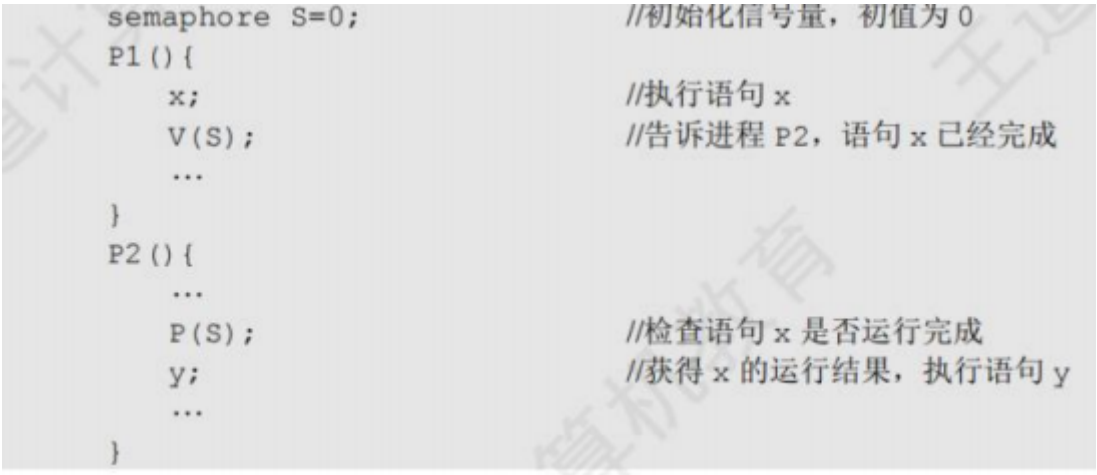
这里仅讨论自旋锁：进程在进入临界区时调用acquire()获得锁，但是如果不能获得锁就会陷入循环忙等；在退出临界区时通过release()归还锁。

信号量

- P操作：相当于wait()，消耗
- V操作：相当于signal()，产出
- 整形信号量：只要信号量S≤0就会一直等待直到signal产出资源
- 记录型信号量：记录型信号量机制是一种不存在“忙等”现象的进程同步机制。除了需要一个用于代表资源数目的整型变量value外，再增加一个进程链表L,用于链接所有等待该资源的进程。记录型信号量得名于采用了记录型的数据结构
- 利用信号量实现互斥：

```
semaphore S=1;                    //初始化信号量，初值为1
P1(){
    ...
    P(S);                         //准备访问临界资源，加锁
    进程P1的临界区;
    V(S);                         //访问结束，解锁
    ...
}
P2(){
    ...
    P(S);                         //准备访问临界资源，加锁
    进程P2的临界区;
    V(S);                         //访问结束，解锁
    ...
}
```

- 利用信号量实现同步：



- 利用信号量实现前驱关系：“前驱操作”之后，对相应的同步信号量执行V操作，在“后继操作”之前，对相应的同步信号量执行P操作。

经典同步问题

- 生产者-消费者问题：一组生产者进程和一组消费者进程共享一个初始为空、大小为n的缓冲区。只有当缓冲区不满时，生产者才能将消息放入缓冲区；否则必须阻塞，等待消费者从缓冲区中取出消息后将其唤醒。只有当缓冲区不空时，消费者才能从缓冲区中取出消息；否则必须等待，等待生产者将消息放入缓冲区后将其唤醒。由于缓冲区是临界资源，因此必须互斥访问

```
semaphore mutex = 1; // 缓冲区互斥信号量
semaphore empty = n; // 缓冲区空闲位置数量
semaphore A = 0;     // 缓冲区A物品数量
semaphore B = 0;     // 缓冲区B物品数量

cobegin
    Process
    producer1()
{
    while (1)
    {
        生产一个A物品;
        P(empty); // 占用缓冲区一个空位
        P(mutex);
        往缓冲区中放入一个A物品;
        V(mutex);
        V(A); // 缓冲区A物品数量加一
    }
}
Process producer2()
{
    while (1)
    {
```

```
        生产一个B物品;
        P(empty); // 占用缓冲区一个空位
        P(mutex);
        往缓冲区中放入一个B物品;
        V(mutex);
        V(B); // 缓冲区B物品数量加一
    }
}
Process consumer1()
{
    while (1)
    {
        P(A); // 请求缓冲区一个A物品
        P(mutex);
        从缓冲区中取出一个A物品;
        V(mutex);
        V(empty); // 缓冲区空位数量加一
        消费一个A物品;
    }
}
Process consumer2()
{
    while (1)
    {
        P(B); // 请求缓冲区一个B物品
        P(mutex);
        从缓冲区中取出一个B物品;
        V(mutex);
        V(empty); // 缓冲区空位数量加一
        消费一个B物品;
    }
}
coend
```

- 读者-写者问题

  a. 读者优先：有读者和写者两组并发进程，共享一个文件，当两个或以上的读进程同时访问共享数据时不会产生副作用，但若某个写进程和其他进程(读进程或写进程)同时访问共享数据时则可能导致数据不一致的错误。因此要求：①允许多个读者可以同时对文件执行读操作；②只允许一个写者往文件中写信息；③任意一个写者在完成写操作之前不允许其他读者或写者工作；④写者执行写操作前，应让已有的读者和写者全部退出

```
int readcount = 0;  semaphore x = 1, wsem = 1;

```

```
void reader()
{
    while (1)
    {
        P(x);
        readcount++;
        Ⅰ if (readcount == 1) P(wsem); // 第一个读者会执行此if语句，随后的会跳过P(wsem)直接进入临界区
        Ⅱ V(x);
        READ;
        P(x);
        readcount--;
        Ⅲ if (readcount == 0) V(wsem); // 最后退出的读者执行此语句，释放临界区权限
        Ⅳ V(x);
    }
}

void writer()
{
    while (1)
    {
        P(wsem);
        WRITE;
        V(wsem);
    }
}

```

b. 写者优先：当有读进程正在读共享文件时，有写进程请求访问，这时应禁止后续读进程的请求，等到已在共享文件的读进程执行完毕，立即让写进程执行，只有在无写进程执行的情况下才允许读进程再次运行

```
int readcount = 0, writecount = 0;
semaphore x=l, y= 1, wsem=1, rsem=l;

void reader( ) {
   while (1) {
      P(rsem);
       P(x);
        readcount++;
        if (readcount ==1) P(wsem);
       V(x);
      V(rsem);
     READ;
```

```
      P(x);
       readcount--;
       if (readcount ==0) V(wsem);
      V(x);
    }
 }
void writer( ) {
  while (1) {
    P(y);
     writecount++;
     if (writecount ==1) P(rsem);
    V(y);
    P(wsem);
     WRITE;
    V(wsem);
    P(y);
     writecount--;
     if (writecount==0) V(rsem);
    V(y);
  }
}
```

- 哲学家进餐问题：

```
semaphore cs[5] = {1, 1, 1, 1, 1} // 每根筷子设置为互斥量
Pi()
{
    while
    {

        if (i % 2 == 1)
        {
            P(cs[i]);           // 互斥取左筷子
            p(cs[(i + 1) % 5]); // 互斥取右筷子
        }
        else if (i % 2 == 0)
        {
            p(cs[(i + 1) % 5]); // 互斥取右筷子
            P(cs[i]);           // 互斥取左筷子
        }
        吃饭... v(cs[i]); // V原语，放下筷子不必考虑先后
        v(cs[(i + 1) % 5]);
        思考...
```

```
    }
}
```

- 吸烟者问题：

```
// 设0，1，2 分别代表烟草，纸和火柴
semaphore S0 = 1; // 互斥信号量，表示供应者是否可以在桌子上放东西
semaphore S1 = 0; // 表示第一个吸烟者所需要的资源
semaphore S2 = 0; // 表示第二个吸烟者所需要的资源
semaphore S3 = 0; // 表示第三个吸烟者所需要的资源
int i = 0, j = 0; // 表示供应者放的两种资源

cobegin
    process businessman
{ // 供应者
    i = RAND() % 3;
    j = RAND() % 3;
    while (i == j)
    { // 随机计算出需要放到桌子上的两样东西
        i = RAND() % 3;
        j = RAND() % 3;
    }
    P(S0);                                  // 如果桌子上没有东西了，则独占桌子的使用权
    Put_items[i] _on_table;                 // 放东西
    Put_items[j] _on_table;                 // 继续放东西
    if ((i = 0 &&j = 1) || (i = 1 &&j = 0)) // 表示供应者提供的是烟草和纸
        V(S3);                              // 第三个吸烟者拥有火柴，烟草和纸是TA需要的，因此唤醒TA
    else if ((i = 1 &&j = 2) || (i = 2 &&j = 1))
        V(S1);
    else
        V(S2)
}

process consumer_k(k = 1, 2, 3)
{
    while (true)
    {
        P(S[k]); // 申请自己需要的资源
        take_one_item_from_table;
        take_one_item_from_table;
        V(S0); // 提醒供应者放东西
        make_cigarette_and_smoking;
```

```
38      }
39  }
40  coend
41
```

管程

- 管程的定义:利用共享数据结构抽象地表示系统中的共享资源，而将对该数据结构实施的操作定义为一组过程,这个代表共享资源的数据结构，以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序称为管程(monitor)
- 条件变量:如果一个进程进入管程后被阻塞，若进程不释放管程,其他进程将无法进入管程。阻塞的原因就被定义为条件变量。

```
monitor Demo{
    共享数据结构 S;
    condition x;                    //定义一个条件变量 x
    init_code(){ ... }
    take_away(){
        if(S<=0) x.wait();          //资源不够，在条件变量 x 上阻塞等待
        资源足够，分配资源，做一系列相应处理;
    }
    give_back(){
        归还资源，做一系列相应处理;
        if(有进程在等待) x.signal();//唤醒一个阻塞进程
    }
}
```

死锁

死锁的概念

- 死锁的定义：多个进程因竞争资源造成的使得各个进程都被阻塞的情况
- 饥饿：进程在信号量内无穷等待的情况。原因是资源分配策略不合理
  - 死锁一定阻塞，但是饥饿不一定
- 死锁原因
  - 系统资源竞争
  - 进程推进顺序非法
- 死锁的必要条件
  - 互斥条件

    - 不可剥夺条件
    - 请求并保持条件
    - 循环等待条件
- 死锁的处理策略
    - 死锁预防（保守）
    - 死锁避免
    - 死锁检测（宽松）

死锁预防

1. 破坏互斥条件
2. 破坏不可剥夺条件
3. 破坏请求并保持条件
4. 破坏循环等待条件

死锁避免

- 银行家算法
    - 若Max - Allocation ⩽ Available则表明可以分配，最后将Allocation回收，继续判断直到没有进程或无法运行位为止

死锁检测和解除

- 资源分配图（死锁定理）
    - 如果能消去资源分配图中的所有边，即能被完全简化，则不会死锁
    - 否则产生死锁
- 死锁解除的方法
    - 资源剥夺法：挂起死锁进程
    - 撤销进程法：强制撤销部分进程
    - 进程回退法：让进程回退到足以回避死锁的地步

内存管理

内存管理概念

内存管理的基本原理和要求

- 程序的链接与装入

  用户源程序变成可执行程序需要经历以下步骤：

  a. 编译：将用户源代码编译成若干目标模块

  b. 链接：将编译后的模块以及所需的库函数链接在一起

     i. 静态链接：程序运行前进行链接，以后不再拆开

     ii. 装入时动态链接：边装入内存边链接

     iii. 运行时动态链接：运行时需要某模块时才进行链接

  c. 装入：将模块装入内存运行

     i. 绝对装入：只适用于单道程序环境， 编译程序产生绝对地址的目标代码

     ii. 可重定位装入：程序中使用的指令和数据地址都是相对于始址的

     iii. 动态运行时装入：装入程序将装入模块装入内存后，地址的转换推迟到程序真正要执行时才进行

- 逻辑地址 & 物理地址
  - 操作系统通过内存管理部件（MMU）将进程使用的逻辑地址转换为物理地址
- 进程的内存映像

  进程映像的要素
  - 代码段
  - 数据段
  - 进程控制块
  - 堆
  - 栈
- 内存保护
  - 采用界地址寄存器：存放最大最小逻辑地址
  - 采用重定位寄存器：存放物理起始地址
- 内存共享
  - 例如多个用户使用同一份程序或数据
- 内存分配与回收、
  - 连续分配
    - 单一连续分配
    - 固定分区分配
    - 动态分区分配

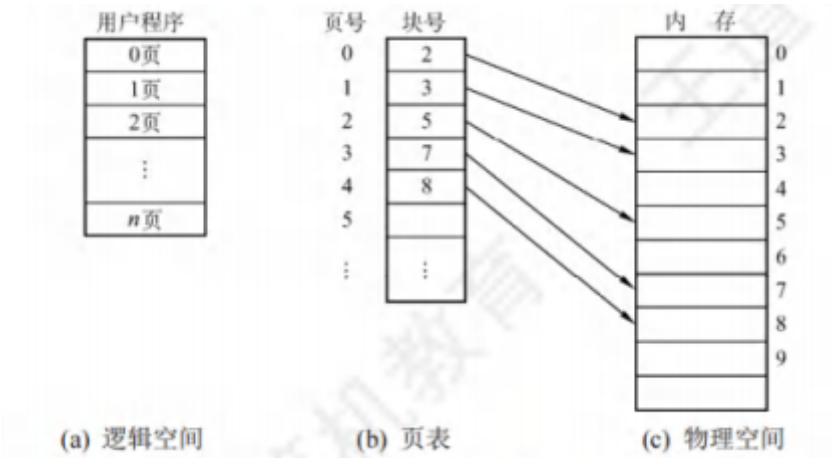
- 离散分配
  - 页式存储管理
  - 段式存储管理

连续分配管理方式

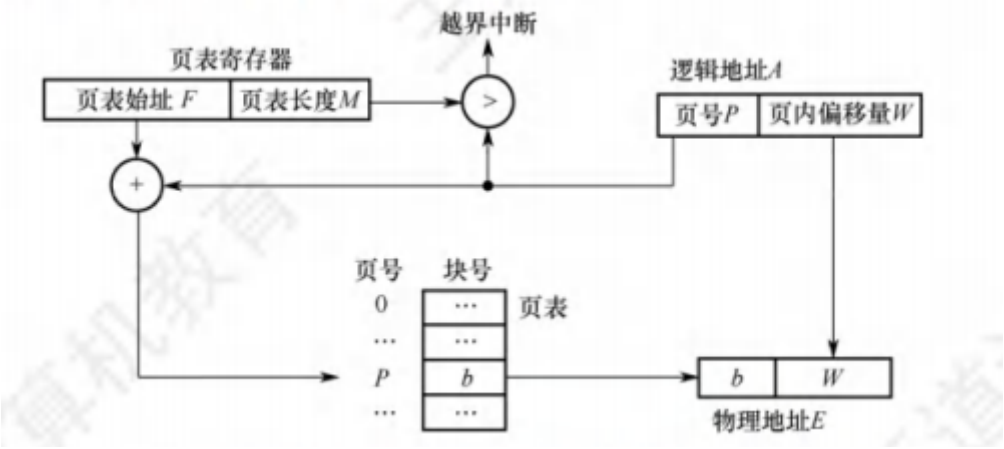
1. 单一连续分配
   - 用户独占整个用户区
   - 无外部碎片，有内部碎片
2. 固定分区分配
   - 将用户区划分为若干固定大小的分区，每个分区只装入一道作业
   - 无外部碎片，有内部碎片
3. 动态分区分配
   - 装入内存时根据作业的实际需要动态地分配内存
   - 内存分区分配算法
     i. 首次适应算法：找到第一个可用的分区即可，通常表现最佳
     ii. 邻近适应算法：从上次查找结束的位置继续查找可用分区
     iii. 最佳适应算法：找最小的能放的下的空闲分区
     iv. 最坏适应算法：找最大的空闲分区
   - 有外部碎片，无内部碎片

基本分页存储管理

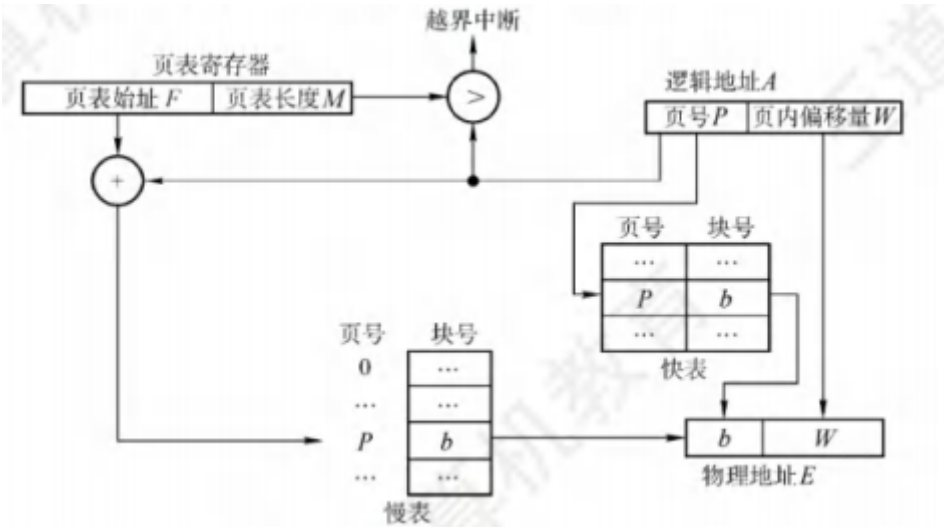
- 地址结构：页号 + 页内偏移量
- 页表



- 基本地址变换机构



- 快表



- 多级页表：用一张索引表记录每个页表的存放位置，而且只需要将当前需要的部分页表调入内存，其他页表仍然位于磁盘，需要时再调入（虚拟内存思想）就可以解决页表占用空间过大的问题

基本分段存储管理

- 地址结构：段号 + 段内偏移量
- 段表：由各个段表项构成：段表项的结构为：段号 + 段长 + 本段在主存的始址
- 地址变换机构：段表寄存器，存放了段表的始址和段表的长度，只要将段号加上段表始址就可以去段表所在地址查询实际物理地址

段页式存储管理

- 地址结构：段号 + 页号 + 页内偏移量
- 为每个进程建立一张段表，每个段有一张页表：进程的段表只有一个，页表可以有多个

虚拟内存管理

虚拟内存的基本概念

- 局部性原理
  - 时间局部性
  - 空间局部性
- 虚拟存储器的定义和特征
  - 定义：将外存所需信息调入内存称为**请求调页**；内存空间不够时需要将暂时不用的信息换到外存成为**页面置换**。二者结合，系统相当于提供了逻辑空间更大的虚拟存储器
  - 特征
    - 多次性：作业可分多次调入内存
    - 对换性：作业无需常驻内存，必要时可以换出
    - 虚拟性：逻辑上扩充内存容量
- 虚拟内存技术的实现方式
  - 请求分页存储管理
  - 请求分段存储管理
  - 请求段页式存储管理

请求分页管理方式

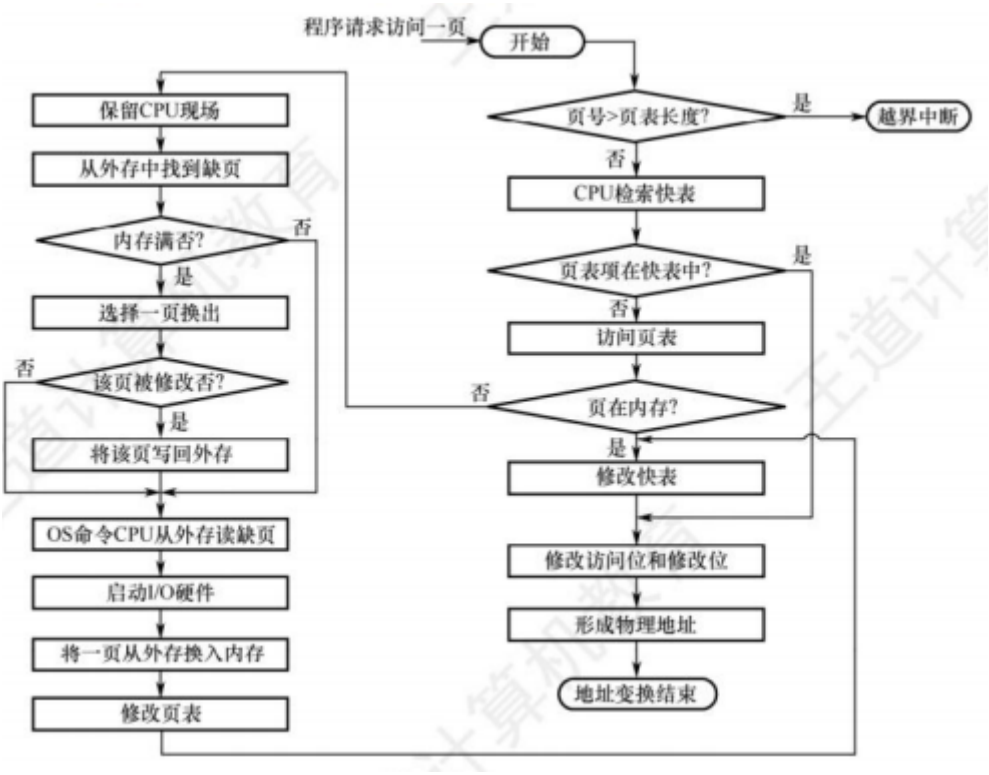
1. 页表机制

   为了实现请求调页功能，需要知道每个页面是否已调入内存；若没有调入内存，还需要直到页面在外存的位置，所以页表项还要增加如**状态位、访问字段、修改位、外存地址**等信息。

2. 缺页中断机构

   请求页面不存在时就要调入内存，此时需要缺页中断机构进行处理。

3. 地址变换机构



页框分配

1. 驻留集大小：即给一个进程分配的页框的集合
    - 驻留集越小，驻留在内存中的进程就越多，可以提高多道程序的并发度，但分配给每个进程的页框太少，会导致缺页率较高，CPU需耗费大量时间来处理缺页
    - 驻留集越大，当分配给进程的页框超过某个数目时，再为进程增加页框对缺页率的改善是不明显的，反而只能是浪费内存空间，还会导致多道程序并发度的下降
2. 内存分配策略
    - 固定分配局部置换：每个进程具有固定数量的物理块。如果运行中发生缺页，只能从已分配页面选出一页换出。
    - 可变分配全局置换：每个进程运行期间可适当增减物理块。如果运行中发生缺页，可以从空闲物理块队列中取出一块。
    - 可变分配局部置换：每个进程运行期间可适当增减物理块。如果运行中发生缺页，只能从已分配页面选出一页换出。
3. 物理块调入算法
    - 平均分配算法
    - 按比例分配算法：按进程大小比例分配
    - 优先权分配算法：按进程优先级分配
4. 调入页面的时机
    - 预调页：运行期
    - 请求调页：运行时

5. 从何处调入页面：对换区

页面置换算法

1. 最佳（OPT）置换算法：选择最长时间内不再被访问的页面进行换出
    - 该算法无法实现
2. 先进先出（FIFO）置换算法：淘汰最早进入内存的页面
    - 没有利用局部性原理，性能较差
    - 只有该算法会出现 **Belady异常**
    - **Belady异常**：分配的页面数增多，但缺页次数不减反增
3. 最近最久未使用（LRU）置换算法：选择最近最长时间未使用的页面进行换出
    - LRU 是堆栈类算法，不可能出现 Belady 异常
4. 时钟（CLOCK）置换算法：
    - 为每个页面设置一位访问位，当某页首次被装入或被访问时，其访问位被置为1
    - 内存中的页面链接成一个循环队列，并有一个替换指针与之相关联
    - 某一页被替换：该指针被设置指向被替换页面的下一页
    - 淘汰一页：若为 0 就选择该页换出；若为1则将它置为 0，暂不换出，顺序检查下一个页面
    - 当检查到队列中的最后一个页面时，若其访问位仍为 1 则返回到队首去循环检查、
    - 改进CLOCK算法：增加修改位，修改过的页面替换代价更大，优先考虑未访问、其次考虑未修改将页面换出

抖动和工作集

- 抖动：刚换出（入）的页面马上又要换入（出）内存，根本原因：为进程分配的物理块太少
- 工作集：某段时间间隔内进程要访问的页面集合

内存映射文件

- 内存映射文件是操作系统向应用程序提供的一个系统调用，在磁盘文件和进程的虚拟地址空间之间建立映射关系

文件管理

文件系统基础

文件的基本概念

1. 文件的定义
    - 文件的结构：数据项+记录+文件
2. 文件的属性：
    - 名称+类型+创建者+所有者+位置+大小+保护+创建时间
3. 文件的分类
    - 按性质和用途：系统文件、用户文件、库文件
    - 按数据形式：源文件、目标文件、可执行文件
    - 按存取控制属性分类：可执行文件、只读文件、读写文件
    - 按组织形式和处理方式分类：普通文件、目录文件、特殊文件

文件控制块和索引节点

1. 文件控制块（FCB）

   新建一个文件就要建立一个FCB
    - 基本信息：文件名、物理位置、逻辑结构、物理结构等
    - 存取控制信息：存取权限
    - 使用信息：建立时间、修改时间
2. 索引节点（i 节点）
    - 检索目录时，文件的其他描述信息不会用到，只需要检查文件目录的目录项，目录项由文件名和对应的索引节点号组成
    - 磁盘索引节点
        - 文件主标识符：拥有该文件的个人或小组的标识符
        - 文件类型：包括普通文件、目录文件或特别文件
        - 文件存取权限：各类用户对该文件的存取权限
        - 文件物理地址
        - 文件长度：指以字节为单位的文件长度
        - 文件链接计数：在本文件系统中所有指向该文件的文件名的指针计数
        - 文件存取时间：本文件最近被进程存取、修改的时间及索引节点最近被修改的时间
    - 内存索引节点
        - 索引节点号：用于标识内存索引节点
        - 状态：指示 i 节点是否上锁或被修改
        - 访问计数：每当有一进程要访问此 i 节点时，计数加 1；访问结束减 1

        - 逻辑设备号：文件所属文件系统的逻辑设备号
        - 链接指针：设置分别指向空闲链表和散列队列的指针

文件的操作

1. 文件的基本操作
    - 创建文件
    - 删除文件
    - 读文件
    - 写文件
2. 文件的打开与关闭
    - 用户首次对某文件发出操作请求时须先利用系统调用open将该文件打开
    - 系统维护打开文件表：系统检索到指定文件的目录项后将该目录项从外存复制到内存中的打开文件表的一个表目中，并将该表目的索引号(也称文件描述符)返回给用户
    - 当用户再次对该文件发出操作请求时可通过文件描述符在打开文件表中查找到文件信息，节省检索开销
    - 当文件不再使用时可利用系统调用close关闭它，系统将会从打开文件表中删除这一表目
    - 每个打开的文件应该具有：文件指针、文件打开计数、文件磁盘位置、访问权限

文件保护

1. 访问类型

   无非是增删改查的权限
2. 访问控制
    - 为每个文件添加访问控制列表（ACL），规定每个用户的允许的访问类型
    - 口令：用户建立文件时提供口令并告知可以共享该文件的其他用户，但口令在系统内部，不够安全
    - 密码：用户对文件加密，访问时需要提供密钥，保密性强

文件的逻辑结构

1. 无结构文件（流式文件）
    - 长度以字节为单位
    - 文件本身没有结构，只能穷举搜索
2. 有结构文件

- 定长记录

  所有记录长度相同

- 变长记录

  i. 顺序文件

  性能较差

  ii. 索引文件

  查找可以利用索引表，加快速度，但是要配置索引表导致存储开销大

  iii. 索引顺序文件

  索引项对应的一个组内的关键字不一定有序，但不同组的关键字集合对应的索引有序

  iv. 散列文件

  会引起冲突

文件的物理结构

1. 连续分配
   - 逻辑连续的文件也在连续物理块
   - 支持顺序、直接访问，速度都很快
   - 产生外部碎片、很难动态增长、、、、不方便插入和删除
2. 链接分配

   a. 隐式链接

   每个磁盘块的最后会指明下一个磁盘块的位置

   b. 显式链接

   用文件分配表（FAT）中的表项指明每个盘块号对应的下一块标号，-1 表示没有下一块了

3. 索引分配

   a. 单级索引分配方式

   为每个文件分配一个索引表，每个盘块占 4KB，不同的盘块号表示不同的盘块，一个文件就可以被分配盘块号数量 * 4KB 大小的空间

   b. 多级索引分配方式

   用多级索引可以表示更多空间，可惜这可能会导致因为索引节点表示的空间太大而出现内部碎片

   c. 混合索引分配方式

结合以上两种，可以比较灵活地表示文件分配的空间

目录

目录的基本概念

FCB 的有序集合就是文件目录，一个 FCB 就是一个文件目录项。

目录结构

1. 单级目录结构
   - 在整个文件系统中只建立一张目录表，每个文件占用一个目录项
   - 查找速度慢
   - 不允许重名
2. 两级目录结构
   - 将文件目录分为主文件目录和用户文件目录，第一层是用户文件目录
   - 只解决了不同用户之间的重名问题
3. 树形目录结构
   - Trie 树
4. 无环图目录结构
   - 在保证拓扑有序的情况下允许不同子树的文件进行引用，但是要维护共享计数器

目录的操作

1. 搜索
2. 创建文件
3. 删除文件
4. 创建目录
5. 删除目录
6. 移动目录
7. 显示目录
8. 修改目录

目录实现

1. 线性列表
2. 哈希表

文件共享

1. 基于索引节点的共享方式（硬链接）
    - 文件目录存放指向索引节点的指针，用引用计数表示文件被共享的次数
    - 索引节点可以连接到真正的文件上
    - 创建文件时被引次数自动为 1
    - 引用次数大于 1 时不能删除文件
2. 利用符号链实现文件共享（软链接）
    - 除了文件的创建者以外，索引节点指向的文件只是一个链接文件，这个文件才指向真正的文件
    - 文件的所有者可以删除文件，无需介意引用次数，但可能有悬空指针，没有副作用
    - 空间和时间上比硬链接稍差

文件系统

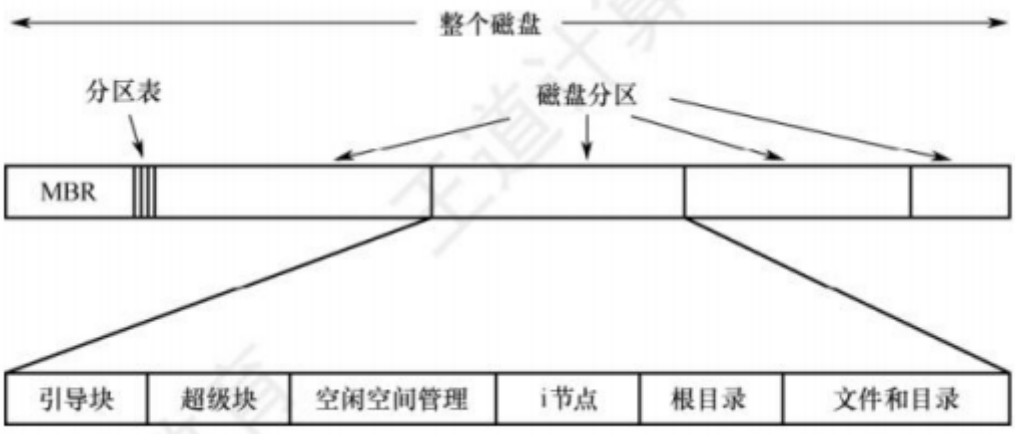
文件系统结构

按照自底向上层次分，文件系统包括：

1. IO 控制层
2. 基本文件系统
3. 文件组织模块
4. 逻辑文件系统

文件系统布局

1. 文件系统在磁盘中的结构



- 主引导记录（MBR）：确定活动分区，读入引导块
- 引导块：启动分区中的操作系统
- 超级块：包含文件系统的所有关键信息，元数据
- 空闲块：用位示图或者指针链接形式给出

2. 文件系统在内存中的结构
   - 安装表：每个已安装的文件系统分区的有关信息
   - 目录结构的缓存：最近访问目录的信息
   - 整个系统的打开文件表：每个打开文件的 FCB 副本、打开计数及其他信息
   - 每个进程的打开文件表：包含进程的开文件的文件描述符

外存空闲空间管理

1. 空闲表法：
   - 为外存的每一块空闲区记录空闲盘块号以及空闲块数信息，构成一张表
   - 有较高的盘块分配速度，减少访问磁盘IO频率
2. 空闲链表法
   a. 空闲盘块链
      - 将磁盘上的所有空闲空间以盘块为单位拉成一条链
      - 分配和回收盘块的过程简单
      - 为文件分配盘块时可能要重复操作多次，效率很低，空闲盘块可能会过长

   b. 空闲盘区链
      - 将磁盘上的所有空闲盘区拉成一条链
      - 分配和回收的效率高，但是分配和回收的过程比较复杂
3. 位示图法
   - 用一个二进制位表示一个盘块是否正在被使用
   - 容易在位示图中找到一个或一组相邻接的空闲盘块
   - 位示图很小，可以保存在内存中节省磁盘的开销
4. 成组链接法
   - 将空闲盘块分成若干组，每组的第一个盘块记录下一组的空闲盘块总数和空闲盘块号，即第一块是索引块
   - 分配：按照栈顶指针不断分配空闲盘块，如果遇到了第一块说明一组的盘块已经用完，此时使用第一块寻找下一组的位置继续分配
   - 回收：若栈满则开辟新的一组

虚拟文件系统（VFS）

当用户程序访问文件时，通过 VFS 提供的统一调用函数操作不同文件系统的文件，而无需考虑具体的文件系统和实际的存储介质

1. 超级块
   - 表示一个已安装(或称挂载)的特定文件系统
   - 超级块对象对应于磁盘上特定扇区的文件系统超级块，用于存储已安装文件系统的元信息
   - 操作方法主要有分配 inode 、销毁 inode 、读 inode 、写 inode 等
2. 索引节点
   - 表示一个特定的文件
   - 索引节点和文件是一对一的关系：只有当文件被访问时才在内存中创建索引节点对象
   - 每个索引节点对象都会复制磁盘索引节点包含的一些数据
   - 提供许多操作函数：创建新索引节点、创建硬链接、创建新目录
3. 目录项
   - 目录项对象是一个路径的组成部分，包含指向关联索引节点的指针、指向父目录和指向子目录的指针
   - 目录项对象在磁盘上没有对应的数据结构，而是VFS在遍历路径的过程中将它们逐个解析成目录项对象的
4. 文件
   - 表示一个与进程相关的已打开文件
   - 文件对象和物理文件的关系类似于进程和程序的关系
   - 文件对象仅是进程视角上代表已打开的文件，它反过来指向其索引节点
   - 文件对象包含与该文件相关联的目录项对象，包含该文件的文件系统、文件指针等，还包含在该文件对象上的一系列操作函数

I/O管理

I/O 管理概述

I/O 设备

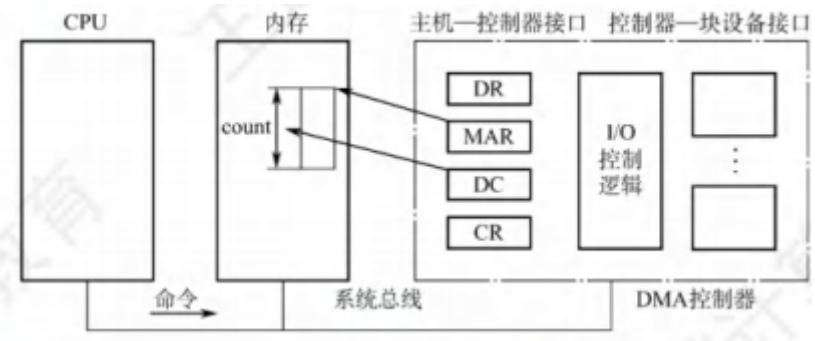
1. 设备的分类
   - 按信息交换的单位分类：

        i. 块设备
        ii. 字符设备
    - 按设备的传输速率分类：
        i. 低速设备：每秒几字节到数百字节
        ii. 中速设备：每秒数千字节到数万字节
        iii. 高速设备：每秒数百千字节到千兆字节
    - 按设备的使用特性分类：
        i. 存储设备
        ii. 输入/输出设备
    - 按设备的共享属性分类：
        i. 独占设备
        ii. 共享设备
        iii. 虚拟设备：通过 SPOOLing 技术将独占设备改造为共享设备
2. I/O 接口
    - 设备控制器与 CPU 的接口
    - 设备控制器与设备的接口
    - I/O 逻辑
    - 设备控制器的功能：接受和识别命令、数据交换、标识和报告设备的状态
3. I/O 接口的类型
    - 按数据传送方式：分为并行接口和串行接口
    - 按主机访问 I/O 设备的控制方式：分为程序查询接口、中断接口、DMA 接口
    - 按功能选择的灵活性：分为可编程接口、不可编程接口
4. I/O 端口
    a. 寄存器：
        - 数据寄存器：用于缓存从设备送来的输入数据或从 CPU 送来的输出数据
        - 状态寄存器：保存设备的执行结果或状态信息以供CPU读取
        - 控制寄存器：由CPU写入以便启动命令或更改设备模式
    b. 编址方式：
        - 独立编址：为每个端口分配一个 I/O 端口号，与主存地址空间独立，使得 I/O 译码更简单，但增加了控制的复杂性

- 统一编址：将主存地址空间分出一部分给 I/O 端口进行编址，更加灵活和方便，但是占用了部分主存空间

I/O 控制方式

1. 程序直接控制方式
   - CPU 对 I/O 设备的控制采取轮询
   - 易于实现
   - CPU 利用率相当低下
2. 中断驱动方式
   - 允许 I/O 设备打断 CPU 并请求服务
   - 相比前一种方式提高了效率，解放了 CPU
   - 但是设备和内存之间的数据狡猾你都必须经过 CPU 中的寄存器
   - 还存在设备的处理单位与 CPU 不相符的情况
3. DMA 方式



   - CR：命令/状态寄存器
   - MAR：内存地址寄存器
   - DR：数据寄存器
   - DC：数据寄存器
   - 在 I/O 设备和内存之间开辟直接的数据交换通路
   - 基本传送单位是数据块，不再是字节
   - 仅在传送一个或多个数据块的开始和结束时才需要 CPU 干预
   - DMA 方式的工作过程是：CPU 接收到设备的 DMA 请求时，它向 DMA 控制器发出一条命令，同时设置 MAR 和 DC 初值，启动 DMA控制器，然后继续其他工作。之后 CPU 就将 I/O 控制权交给 DMA 控制器，由 DMA 控制器负责数据传送。DMA 控制器直接与内存交互，每次传送一个字，这个过程不需要 CPU 参与。整个数据传送结束后，DMA 控制器向 CPU 发送一个中断信号。因此只有在传送开始和结束时才需要 CPU 的参与

4. 通道控制方式
    - 设置通道后 CPU 只需向通道发送一条 I/O 指令，指明通道程序在内存中的位置和要访问的 I/O 设备，通道收到该指令后，执行通道程序，完成规定的 I/O 任务后，向 CPU 发出中断请求。通道方式可以实现 CPU 、通道和 I/O 设备三者的并行工作，从而更有效地提高整个系统的资源利用率
    - 通道与一般处理机的区别：通道指令的类型单一，没有自己的内存，通道所执行的通道程序是放在主机的内存中的，也就是说通道与CPU共享内存
    - 通道与DMA方式的区别：DMA方式需要CPU来控制传输的数据块大小、传输的内存位置，而通道方式中这些信息是由通道控制的。另外，每个DMA控制器对应一台设备与内存传递数据，而一个通道可以控制多台设备与内存的数据交换

I/O 软件层次结构

1. 用户层软件
2. 设备独立性软件
3. 设备驱动程序
4. 中断处理程序

应用程序的 I/O 接口

1. I/O 接口分类
    - 字符设备接口
    - 块设备接口
    - 网络设备接口
2. 阻塞 I/O 和非阻塞 I/O
    - 阻塞 I/O：指当用户进程调用 I/O 操作时，进程就被阻塞，并移到阻塞队列，I/O 操作完成后，进程才被唤醒，移到就绪队列。当进程恢复执行时，它收到系统调用的返回值，并继续处理数据
        - 优点：操作简单，实现难度低，适合并发量小的应用开发
        - 缺点：I/O 执行阶段进程会一直阻塞下去

- 非阻塞 I/O：指当用户进程调用I/O操作时，不阻塞该进程，但进程需要不断询问I/O操作是否完成，在I/O执行阶段，进程还可以做其他事情
  - 优点：进程在等待I/O期间不会阻塞，可以做其他事情，适合并发量大的应用开发
  - 缺点：轮询方式询问I/O结果，会占用CPU的时间

设备独立性软件

设备独立性软件

- 也称与设备无关的软件，是I/O系统的最高层软件，它的下层是设备驱动程序，其界限因操作系统和设备的不同而有所差异。比如，一些本应由设备独立性软件实现的功能，也可能放在设备驱动程序中实现。这样的差异主要是出于对操作系统、设备独立性软件和设备驱动程序运行效率等多方面因素的权衡
- 总体而言，设备独立性软件包括执行所有设备公有操作的软件

高速缓存与缓冲区

1. 磁盘高速缓存
   - 逻辑上属于磁盘，物理上属于驻留在内存中的盘块
2. 缓冲区

   假定从设备将一块数据输入到缓冲区的时间为 T，操作系统将该缓冲区中的数据传送到工作区的时间为 M，而 CPU 对这一块数据进行处理的时间为 C。

   注意，必须等缓冲区充满后才能从缓冲区中取出数据。

   a. 单缓冲
      - 当 T > C 时：平均处理一块数据的时间是 T + M
      - 当 T < C 时：平均处理一块数据的时间是 C + M
      - 合并以上两种情况得 Max(C, T) + M

   b. 双缓冲
      - 当 T > C + M 时：平均处理一块数据的时间是 T
      - 当 T < C + M 时：平均处理一块数据的时间是 C + M
      - 合并以上两种情况得 Max(C + M, T)

   c. 循环缓冲

   d. 缓冲池
3. 高速缓存与缓冲区的对比

		高速缓存	缓冲区
相同点		都介于高速设备和低速设备之间	
区别	存放数据	存放的是低速设备上的某些数据的复制数据，即高速缓存上有的，低速设备上面必然有	存放的是低速设备传递给高速设备的数据（或相反），而这些数据在低速设备（或高速设备）上却不一定有备份，这些数据再从缓冲区传送到高速设备（或低速设备）
	目的	高速缓存存放的是高速设备经常要访问的数据，若高速设备要访问的数据不在高速缓存中，则高速设备就需要访问低速设备	高速设备和低速设备的通信都要经过缓冲区，高速设备永远不会直接去访问低速设备

设备分配与回收

1. 设备分配的数据结构
    a. 设备控制表（DCT）：存储设备的各个属性
    b. 控制器控制表
    c. 通道控制表
    d. 系统设备表
2. 设备分配时应考虑的因素
    a. 设备的固有属性：独占/共享/虚拟
    b. 设备分配算法
    c. 设备分配中的安全性
3. 设备分配的步骤
    a. 分配设备
    b. 分配控制器
    c. 分配通道
4. 逻辑设备名到物理设备名的映射
    a. 利用逻辑设备表（LUT）
    b. LUT 包含：逻辑设备名、物理设备名、设备驱动程序的入口地址

SPOOLing 技术

- **将低速 I/O 设备上的数据传送到高速磁盘上，或进行反过程**
- **当CPU需要输入数据时可以直接从磁盘中读取数据，需要输出数据时可以先将数据输出到磁盘上**
- **牺牲了空间，换取了高速的 CPU 能够暂时存放数据的机会**

设备驱动程序接口

- 设备驱动程序是I/O系统的上层与设备控制器之间的通信程序

- 其主要任务是接收上层应用发来的抽象I/O请求，如 read 或 write 命令，将它们转换为具体要求后发送给设备控制器，进而使其启动设备去执行任务
- 反之，它也将设备控制器发来的信号传送给上层应用

磁盘和固态硬盘

磁盘

- 磁盘安装在磁盘驱动器中
- 磁盘地址用 **柱面号・盘面号・扇区号** 表示

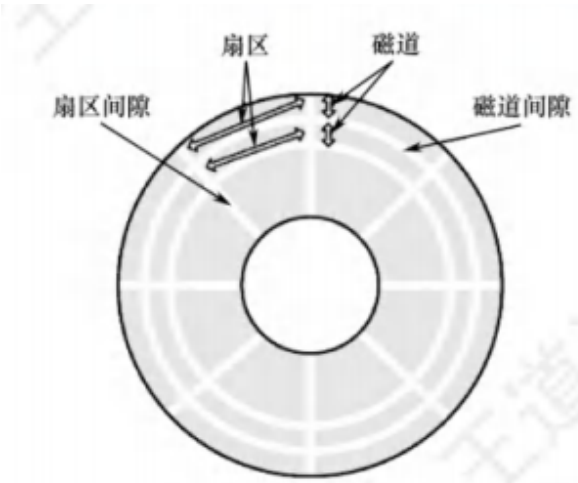


图 5.15　磁盘盘片

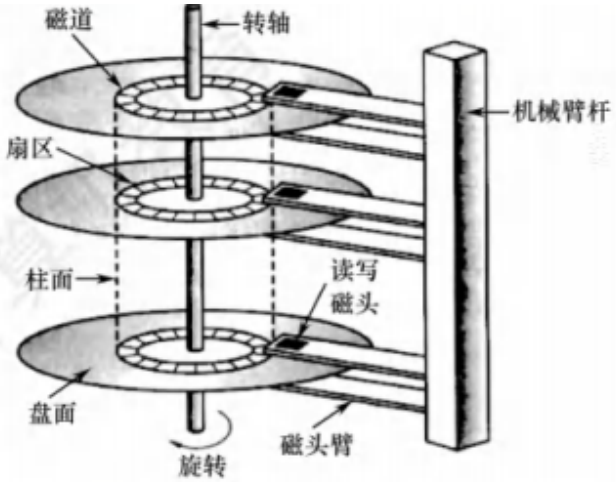


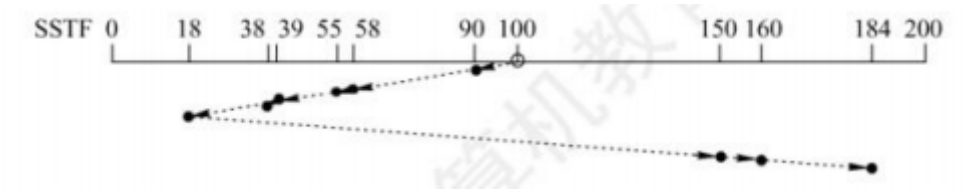
图 5.16　磁盘的结构

磁盘的管理

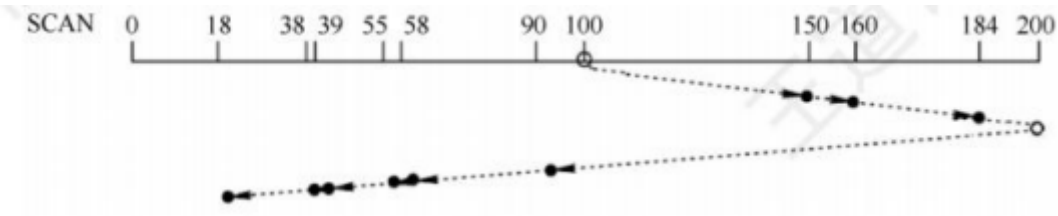
1. 磁盘初始化
    a. 物理格式化：将磁盘分为若干个扇区
    b. 逻辑格式化：向磁盘写入文件系统
2. 分区
3. 引导块
    - 计算机启动时需要运行一个初始化程序（自举程序）,它初始化CPU、寄存器、设备控制器和内存等，接着启动操作系统。为此，自举程序找到磁盘上的操作系统内核，将它加载到内存，并转到起始地址，从而开始操作系统的运行
    - Windows系统将引导代码存储在磁盘的第 0 号扇区，它称为主引导记录（MBR）。引导首先运行 ROM 中的代码，这个代码指示系统从 MBR 中读取引导代码。除了包含引导代码， MBR 还包含一个磁盘分区表和一个标志（以指示从哪个分区引导系统）
4. 坏块

磁盘调度算法

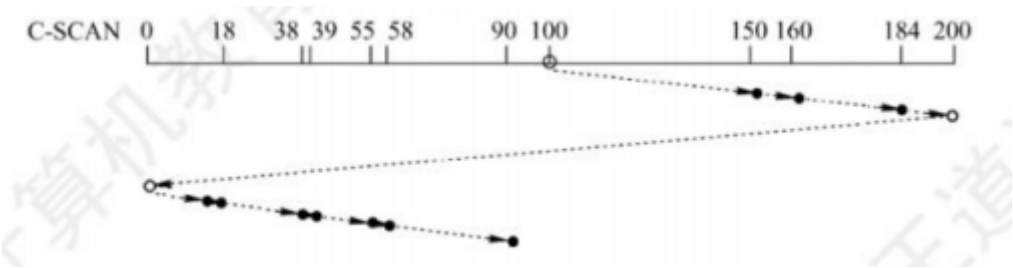
1. 磁盘的存取时间 = 寻道时间 + 旋转时间 + 传输时间
    a. 寻道时间：m 为常数，n 为跨越的磁道数， s 为启动磁头臂的时间，结果为 m * n + s
    b. 旋转延迟时间：1 / 2r，r 为转速
    c. 传输时间：b 为传输字节数，r 为转速，N 为一个磁道上的字节数，结果为 b / rN
2. 磁盘调度算法
    a. 先来先服务算法
    b. 最短寻道时间优先算法（SSTF）



    c. 扫描（SCAN）算法：移到最边缘时才向内移动



    d. 循环扫描（C-SCAN）算法：移到边缘后，返回时直接回到起始端



    e. 如果不是移动到边缘而是移动到最远的一个请求点后就可以返回，对 SCAN 和 C-SCAN 就分别优化为 LOOK 和 C-LOOK

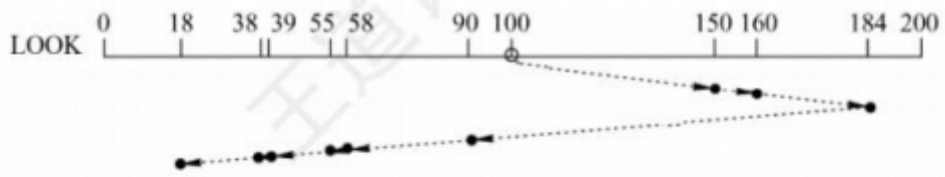


图 5.22　LOOK 磁盘调度算法

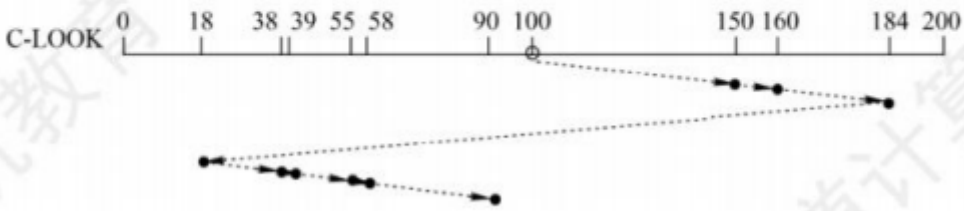


图 5.23　C-LOOK 磁盘调度算法

表 5.2　四种磁盘调度算法的优缺点

	优　点	缺　点
FCFS 算法	公平、简单	平均寻道距离大，仅应用在磁盘 I/O 较少的场合
SSTF 算法	性能比“先来先服务”好	不能保证平均寻道时间最短，可能出现“饥饿”现象
SCAN 算法	寻道性能较好，可避免“饥饿”现象	不利于远离磁头一端的访问请求
C-SCAN 算法	消除了对两端磁道请求的不公平	—

3. 提高磁盘 I/O 速度的方法
    a. 采用磁盘高速缓存
    b. 调整磁盘请求顺序
    c. 提前读
    d. 延迟写
    e. 优化物理块的分布
    f. 虚拟盘：用内存空间仿真磁盘
    g. 采用磁盘阵列 RAID

固态硬盘

1. 固态硬盘的特性
    - 固态硬盘(Solid State Disk,SSD)是一种基于闪存技术的存储器
    - 一个SSD由一个或多个闪存芯片和闪存翻译层组成
    - 闪存芯片替代传统磁盘中的机械驱动器，而闪存翻译层将来自CPU的逻辑块读写请求翻译成对底层物理设备的读/写控制信号，因此闪存翻译层相当于扮演了磁盘控制器的角色
    - 随机写很慢，有两个原因：
        - 首先，擦除块比较慢，通常比访问页高一个数量级

        - 其次，如果写操作试图修改一个包含已有数据的页 P，那么这个块中所有含有用数据的页都必须被复制到一个新（擦除过的）块中，然后才能进行对页 P_1 的写操作
    - 比起传统磁盘，SSD 有很多优点：
        - 它由半导体存储器构成，没有移动的部件，因此随机访问速度比机械磁盘要快很多
        - 没有任何机械噪声和震动，能耗更低、抗震性好、安全性高等。
2. 磨损均衡
    a. 动态磨损均衡：写入数据时自动选择较新的闪存块
    b. 静态磨损均衡：就算没有数据写入，也会检测并自动进行数据分配，让老的闪存块承担无需写数据的存储任务，同时让新的闪存块腾出空间，进行更多的读写操作