

## 1 操作系统概论

### 2 存储管理

- 2.1 存储管理基础
- 2.2 基本存储管理方法
- 2.3 可变分区存储管理
- 2.4 内存扩充技术
- 2.5 纯分页的存储管理
- 2.6 请求分页系统
- 2.7 段式存储管理
- 2.8 段页式存储管理
- 2.9 Linux存储管理

### 3 进程管理

- 3.1 进程概述
- 3.2 进程控制块
- 3.3 调度
- 3.4 UNIX系统的进程调度
- 3.5 进程的控制
- 3.6 进程的创建和映像改换
- 3.7 线程

### 4 进程通信

- 4.1 进程的同步与互斥
- 4.2 进程间互斥控制方法
- 4.3 信号量和semWait、semSignal操作
- 4.4 信号量的应用
- 4.5 进程间的数据通信
- 4.6 软中断和信号机构
- 4.7 死锁

### 5 设备管理

- 5.1 概述
- 5.2 操作系统与中断处理
- 5.3 操作系统与时钟系统
- 5.4 操作系统对I/O操作的控制
- 5.5 设备管理的数据结构
- 5.6 磁盘调度
- 5.7 UNIX系统V的设备管理
- 5.8 设备分配

### 6 文件系统

- 6.1 概述
- 6.2 文件目录
- 6.3 文件存储资源分配
- 6.4 文件的系统调用
- 6.5 文件的标准子例程
- 6.6 UNIX文件系统的内部结构
- 6.7 管道文件和管道通信

# 1 操作系统概论

操作系统——用以控制和管理计算机系统资源，方便用户使用的程序和数据结构的集合

脱机输入/输出——具体的输入/输出不需要在主计算机上进行，而是使用磁带作为输入/输出的中介，极大提高计算机的输入/输出速度

批处理系统——一个程序运行完毕，OS自动从输入磁带上读入下一个作业，直到整批作业全部处理完毕

### OS技术转折点：

- 缓冲技术
  - 脱机、批处理没有完全解决CPU与外部设备速度的匹配问题
  - CPU和外设之间设置缓冲区，CPU与外设并行工作，减少互相等待时间
- 中断技术
  - 具有缓冲的输入输出导致CPU需要轮询外设状态
  - 只要I/O设备一旦完成了输入/输出操作，它会自动向CPU发出中断信号，解决了CPU和外设的协调
- DMA技术
  - CPU响应中断和处理数据的耗时可能大于数据到达的时间间隔，从而导致数据丢失
  - 一旦收到DMA发来的中断请求，CPU在设置了缓冲区、指针、计数器后，DMA不再需要CPU干预，在内存和设备之间整块传输数据。消除了每次只能传输一个数据的限制，传输效率提高；传输由DMA设备完成，不需要CPU参与
- SPOOLING(假脱机)
  - 受限于IO的作业，输出缓冲满，而输入空；受限于CPU的作业，输入缓冲区满，而输出空
  - 将磁盘模拟为输入/输出设备，以磁盘为几乎无限大的缓冲区来解决上述问题
  - 技术特点：
    - 从对低速I/O设备进行的I/O操作变为对输入井或输出井(实际是磁盘)的操作，提高了I/O速度，缓和了CPU与低速I/O设备速度不匹配的矛盾
    - 设备并没有分配给任何进程，在输入井或输出井中，分配给进程的是一存储区和建立一张I/O请求表
    - 实现了虚拟设备功能，多个进程同时使用一独享设备，而对每一进程而言，都认为自己独占这一设备，不过是逻辑上的设备
  - 组成部分：输入井和输出井、输入缓冲区和输出缓冲区、输入进程和输出进程、I/O请求队列
  - SPOOLING和内存缓冲技术的区别：
    - 内存缓冲技术：使用内存缓冲，使作业的IO与本身的计算重叠地进行
    - SPOOLING技术：使用磁盘缓冲，使多个作业的IO与计算重叠地进行
  - 真脱机与假脱机的区别和联系：
    - 联系：利用缓冲，缓和了CPU与低速I/O设备速度不匹配的矛盾
    - 区别：假脱机使得独占设备变成可共享设备；OS可根据系统当前状况从缓冲池的作业中挑选下一个作业，而不用按顺序运行作业，使作业调度成为可能，提高了利用率
- 多道程序设计：
  - 以上技术不能使CPU和I/O设备保持忙碌状态
  - OS将多个作业放在作业缓冲池，一个作业的运行受阻需等待时可调度另一个作业运行，使CPU保持忙碌

现代OS类型：

- 分时系统：多路性、独立性、交互性、及时性
- 实时系统：限定时间内快速响应
- 微机操作系统：单用户系统(资源独享)、用于工作站的系统、用于服务器的系统
- 多处理机系统：提高系统并行性，主从式、对称式
- 分布式操作系统：多计算机系统，地理位置不同，软硬件资源不同；具有一个统一的操作系统，分配子任务，调度，管理资源，对用户透明
- 网络操作系统：网络操作系统不是一个集中、统一的操作系统，基本是在各种各样自治的计算机原有操作系统基础上加上具有各种网络访问功能的模块
- 嵌入式操作系统
- 智能卡操作系统：单片微机系统

作业——请求计算机完成的一个完整的处理任务，可包括几个程序的相继执行，作业中相对独立的部分为作业步

进程——程序在一个数据集合上的运行活动，它是系统进行资源分配和调度的一个可并发执行的独立单位

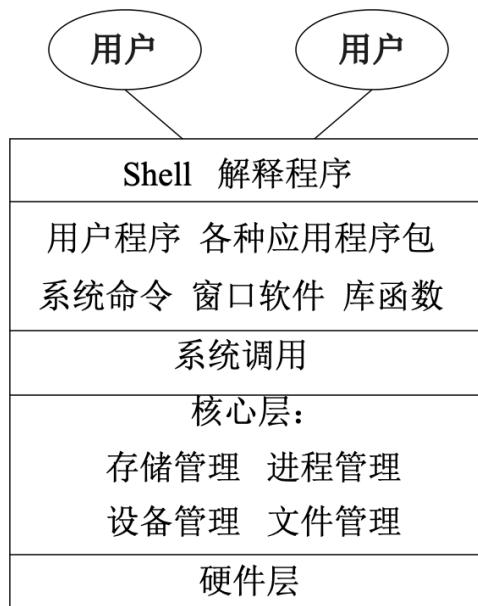
#### OS特征：

- 并发：在一段时间内，多个程序同时在运行，而并行是同一时刻多个运行，只有多CPU才能实现
- 共享：多个用户和程序共享系统的软硬件资源，互斥共享（打印机，写数据），同时共享（磁盘，只读数据）
- 虚拟：为裸机提供高级抽象服务，虚拟出功能更强大的虚拟服务（分时系统、SPOOLING）
- 不确定性：同样一个数据集的同一个程序在同样的计算机环境下运行，其执行顺序和所需时间都不相同，不是指结果

#### OS功能：

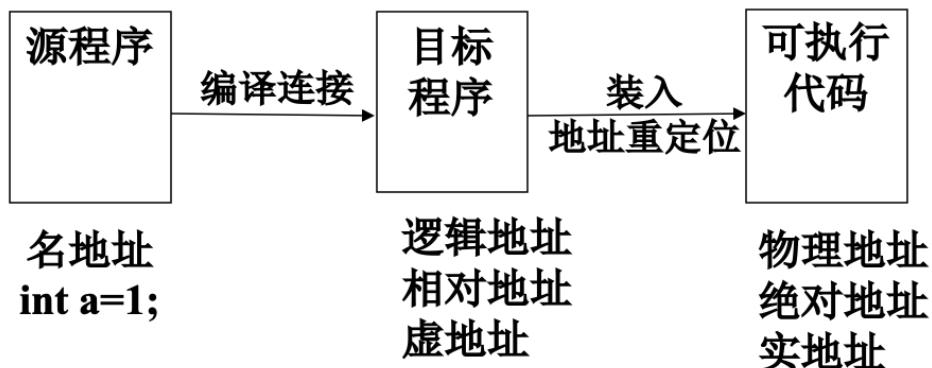
- CPU管理：作业和进程调度、进程控制、进程通信
- 存储管理：内存分配、地址映射、内存保护、内存扩充
- 设备管理：缓冲区管理、设备分配、设备驱动、设备无关性
- 文件管理：文件存储空间的管理、文件操作的一般管理、目录管理、文件的读写管理和存取控制
- 用户接口：命令界面、程序界面（系统调用）、图形界面

UNIX系统基本结构：



## 2 存储管理

## 2.1 存储管理基础



地址定位方式：

- 固定定位方式：直接指定程序在执行时访问的实际存储器地址
- 静态重定位方式：目标代码中的地址是以0为起始地址的相对地址，当需要执行时，根据作业在本次分配到的内存起始地址将可执行目标代码装到指定内存地址中，并修改所有有关地址部分的值——对每一个逻辑地址的值加上内存区首地址（基址）值
  - 无需硬件地址变换机构支持
  - 必须连续存储区域，难扩展、难移动、难共享
- 动态重定位方式：程序在装入内存时不必修改程序的逻辑地址值，程序执行期间在访问内存之前，再将逻辑地址转换成物理地址——先将逻辑地址送入虚地址寄存器VR，再将BR和VR中的值相加后送入地址寄存器MR，并按MR中的值访问内存
  - $(MR) = (BR) + (VR)$
  - 运行期间可换入换出、无需连续、可移动、去碎片

## 2.2 基本存储管理方法

单一连续区存储管理：

- 单用户系统，一个作业在运行时独占整个用户区
- 固定定位/静态重定位

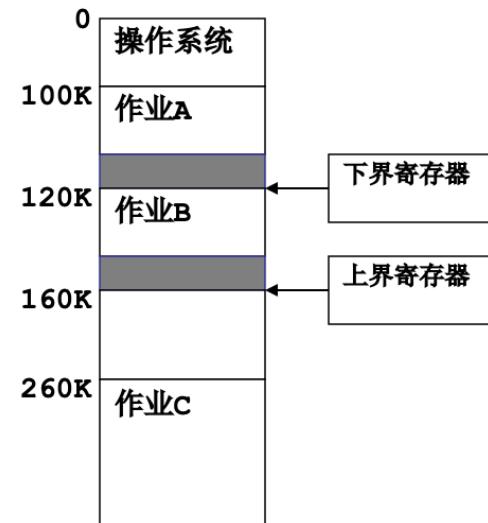


固定分区存储管理：

- 在系统初始化时就把存储空间划分成若干个分区，这些分区的大小可以不同，以支持不同的作业对内存大

- 小需求的不同
- 需要建立分区说明表
- “内零头”浪费
- 静态重定位/动态重定位

分区号	大小	地址	状态
1	20K	100K	已分配
2	40K	120K	已分配
3	100K	160K	未分配
4	200K	260K	已分配

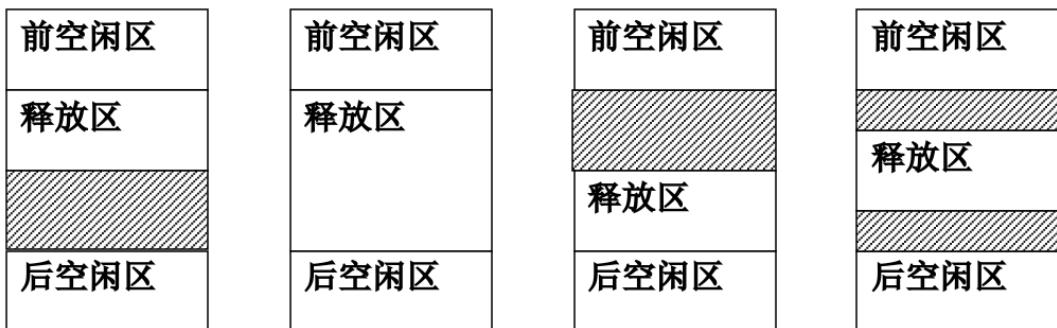


## 2.3 可变分区存储管理

等到作业运行需要内存时就向系统申请，从空闲的内存区中“挖”一块出来，其大小等于作业所需内存大小，这样就不会产生“内零头”，但有“外零头”

首次适应法：

- 分配算法：从表的起始端的低地址部分开始查找，当第一次找到大于或等于申请大小的空闲区时，就按所需大小分配给作业
- 回收算法：



- 利用低地址的空闲区，尽可能保留高地址的较大空闲区，以满足大作业
- 空闲区按地址排序

循环首次适应法：

- 把空闲表设计成顺序结构或双向链接结构的循环队列，设置一个起始查找指针，指向循环队列中的一个空闲区表项
- 分配时总是从起始查找指针所指的表项开始查找，第一次找到满足要求的空闲区时，就分配所需大小的空闲区，修改表项，并调整起始查找指针，使其指向队列中被分配的后面的那块空闲区
- 空闲区按地址排序

最佳适应算法：

- 在所有大于或等于要求分配的空闲分区长度中挑选一个最小的分区
- 顺序结构：空闲区按地址排序
- 链接结构：空闲区按由小到大的非递减次序排序，释放算法效率低

最差适应法：

- 分割的空闲存储区是所有空闲分区中的最大的一块
- 链接结构：由大到小的次序排序，一次查找就行，分配快
- 顺序结构没优点

扫描次数：最佳(2+2) 最差(1+2) 首次(1+1) 循环(1+1)

小碎片产生可能性：最佳 > 首次 > 循环 > 最差

后继大作业分配成功可能性：最佳 > 首次 > 循环 > 最差

多重分区：

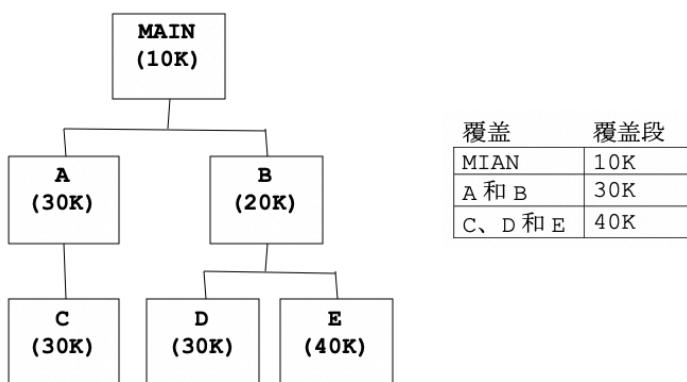
- 在作业的运行过程中可以申请附加的分区，以装入一个或多个子程序，新申请的空闲分区也无需与作业的现有分区相邻接
- 提高外部碎片的利用率，也便于多个作业共享分区的代码和数据
- 动态重定位

## 2.4 内存扩充技术

借助大容量的辅存在逻辑上实现内存的扩充

覆盖：

- 用于一个作业的内部
- 将一个大程序按程序的逻辑结构划分成若干个程序（或数据）段，并将不会同时执行、从而就不必同时装入内存的程序段分在一组内，该组称为覆盖段
- 这个覆盖段可分配到同一个称为覆盖区的存储区域
- 对程序员不透明



交换技术：

- 用于不同的作业
- 当作业的时间片用完或因等待某一事件而不能继续运行时，系统就挑选下一个作业进入主存运行
- 可只将原作业的一部分保存到辅存中，只要释放的主存空间刚好能够装入下一个运行作业
- 对程序员透明

虚拟存储器：

- 作业即使不全部装入主存也能正确运行
- **局部性原理**: 从运行的时间角度来分析, 在一段时间内作业一般不会执行到所有程序段的指令和存取绝大部分数据, 它往往相对集中地访问某些区域中的指令和数据
- 在主存中可只装入最近经常要访问的某些区域的指令和数据, 剩余部分就暂时不必装入, 等到以后要访问到它们时再调入内存
- 程序地址空间受到两个条件的限制:
  1. 指令地址字长度的限制
  2. 存放程序指令和数据的外存储器(交换区) + 内存 大小

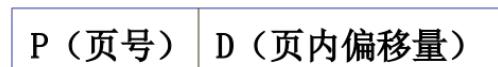
## 2.5 纯分页的存储管理

基本思想:

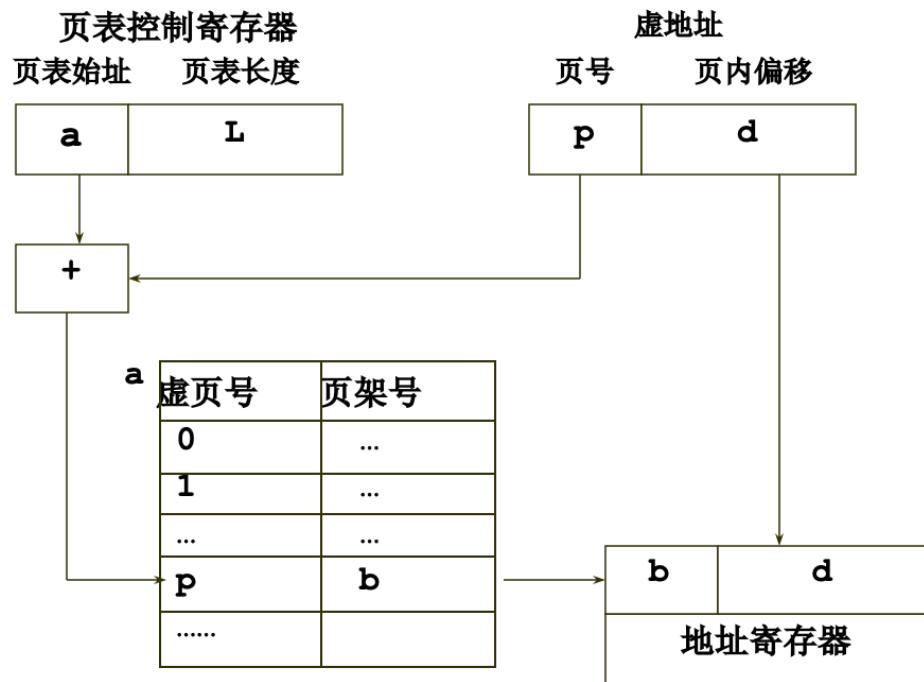
- 将一个作业用户空间划分为较小单位, 分散驻留到内存碎片中, 提高利用率
- 把作业的虚拟地址空间划分成若干长度相等的页, 也称虚页, 每一个作业的虚页都从0开始编号
- 主存也划分成若干与虚页长度相等的页架, 也称页框或实页, 主存的页架也从0开始编号
- 程序装入时, 每一个虚页装到主存中的一个页架中, 这些页架可以是不连续的
- 纯页式(静态页式)存储管理、请求页式(动态页式)存储管理

**地址变换:**

- 地址结构:

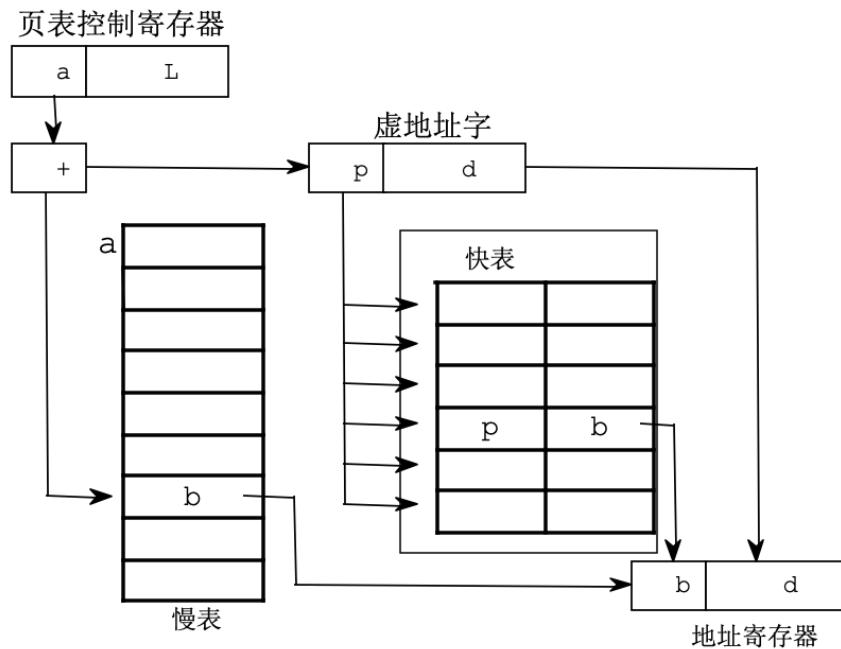


- 虚拟地址字的页内偏移部分占据低 $n$ 个二进制位, 使 $2^n$ 刚好等于页的大小
- 程序的虚地址空间是连续的, 但程序的虚页可以分配到主存中不连续的空闲页架中
- 故分页系统需要在主存中开辟一个页表区域来建立每一个作业的虚页号到内存的页架号之间的映射关系
- 系统还需要建立一个总页表来记录各个作业页表的始址和长度, 并将当前运行进程的页表始址和长度存放
- 在页表控制寄存器中



联想存储器和快表：

- 快表——一组快速寄存器，用来存放当前运行作业的页表表项，以加速地址变换过程
- 快表由联想寄存器构成，联想寄存器是一种按内容进行并行查找的一组快速寄存器，输入p，快速输出b
- 现运行作业的页表不能全部装入到联想寄存器中，故一般只装入当前经常要访问的那些页表表项



空闲内存页的管理：

- 在页式管理系统中需要一张表记录主存中每个页的使用情况和当前空闲页的总数
- 由于主存页总数很多，且页的大小相等，故可用位图描述各页的状态

## 2.6 请求分页系统

请求分页的基本原理：

- 每一个运行作业，只装入当前运行需要的一部分页面集合，该集合又称“工作集”
- 当作业运行时需要访问其它不在主存中的虚页时，硬件产生“缺页中断”
- 如主存资源紧张，可在原先装入主存的页面中选择一个或多个页，将其换出到辅存中，再把所需的页调入主存

作业的虚页可能驻在主存中，也可能驻在辅存中，因此在页表中要扩充表项的内容：

虚页号	内存页架号	辅存地址	状态
-----	-------	------	----

页面淘汰：

- 最好的策略：淘汰那些今后不会再被访问或最长的时间里不会被访问的页
- **页面抖动**：刚刚将某一页淘汰，随后又要访问该页，从而使系统花费大量的时间用于页面在主存和辅存之间频繁的调入调出，大大降低了作业运行效率
- **最优淘汰算法 (OPT)**：
  - 程序执行时访问页的页号序列构成了页面流
  - 淘汰那些从当前时刻起在页面流中不再出现的页，如没有这类页，则淘汰一个在页面流中最晚出现的
  - 淘汰算法评价标准
- **先进先出淘汰算法 (FIFO)**：
  - 总是淘汰最早调入主存的页面，可采用一个先进先出的队列实现
  - 在作业按顺序访问地址空间的情况下才是理想的，否则效率不高
- **最近最少使用淘汰算法 (LRU)**：
  - 比较最近一段时间里对各页面的访问频率，淘汰访问频率最低的页面
  - 实际上很多系统将其实现为淘汰“最近一段时间内最久没有访问”过的页，依据为局部性原理
  - 可以设置一个从中间可以出队的非严格意义上的队列，队列中的每一个单元对应于一个内存页，每次访问一个页面时将对应单元从队列中抽出，重新排到队尾去，淘汰的页从队列头取
  - 减少算法时间复杂性的方法：
    1. 用硬件实现队列结构和操作
    2. 不必在每条访存指令都执行队列操作，可以一定时间间隔（如时钟中断）执行一次
    3. 矩阵法，k行置1，k列置0，淘汰二进制值最小的一行
- **最近未使用淘汰算法 (NUR) (Clock算法)**：
  - 淘汰最近一段时间内未曾访问过的某一页，低开销的近似于LRU的淘汰算法
  - 不仅能考虑最近未曾访问过的页，还能优先挑选页面数据未曾修改过的页，这样可减少将淘汰页写回辅存的开销
  - 这种算法要为每一项增设两个硬件位——访问位和修改位。当该页未访问过或没修改过时对应位为0，反之为1
  - 初始时所有页的访问位和修改位都清0。当访问某页时，该页的访问位置1，当某页的数据被修改后，将该页的访问位和修改位都置1
  - 系统将主存中每个使用页组织成一个循环队列，并设置一个循环检测指针
  - 当需要淘汰一页时，从指针开始循环考察各页的使用情况：
    - 若两位均为0，选择该页淘汰

- 如果访问位为1，则清0
- 如果访问位为0，且修改位为1，则将修改位清0，同时更新辅存对应页
- 最新读过的页在第一轮循环检测中不会被选中，最近写过的页在第一、二轮循环检测中不会被选中
- 另一种策略：
  - 不在检测时清0，而是在某个时刻将所有访问位清零，则使用状态存在4种状态

序号	访问位	修改位
0	0	0
1	0	1
2	1	0
3	1	1

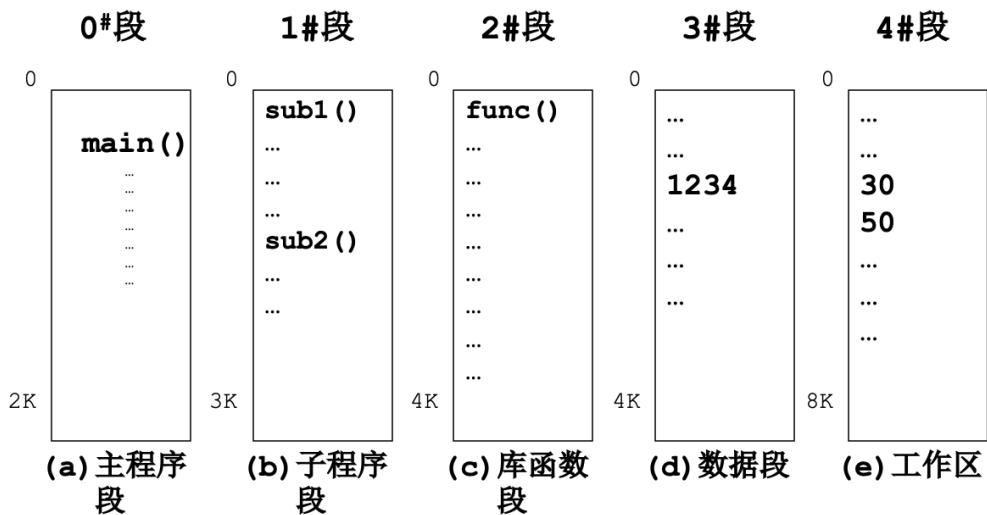
- 淘汰策略：淘汰序号最小的一个页，访问位为0而修改位为1的页是自最后那次清0以来还未被访问过的

全局淘汰算法、局部淘汰算法

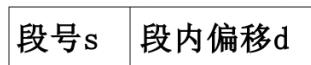
## 2.7 段式存储管理

页式存储管理，作业地址空间是连续一维的，各作业间难以共享代码和数据

段式存储管理，用户可以根据逻辑结构将程序分成若干段，每一段的虚地址空间各自都从0开始编址，因此整个作业的虚地址空间是二维的。主存分配以段为单位，每一个段要分配一个连续的主存分区，各个段之间不必相邻



在段式存储系统中，指令的虚地址字分为段号和段内偏移两个部分：



段式管理通过一个段表进行地址变换：

段号	段长	内存基地址	外存基地址	状态
0	2	.....	.....	LOAD
1	3	.....	.....	LOAD
2	4	.....	.....	LOAD
3	4	.....	.....	LOAD
4	8	.....	.....	LOAD

系统还要建立一张所有作业的段表起始地址和长度的表

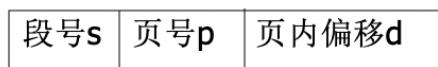
## 2.8 段页式存储管理

页式管理优点：大容量的虚拟存储器、没有页外碎片、无需紧凑内存、有效利用主存、对用户透明

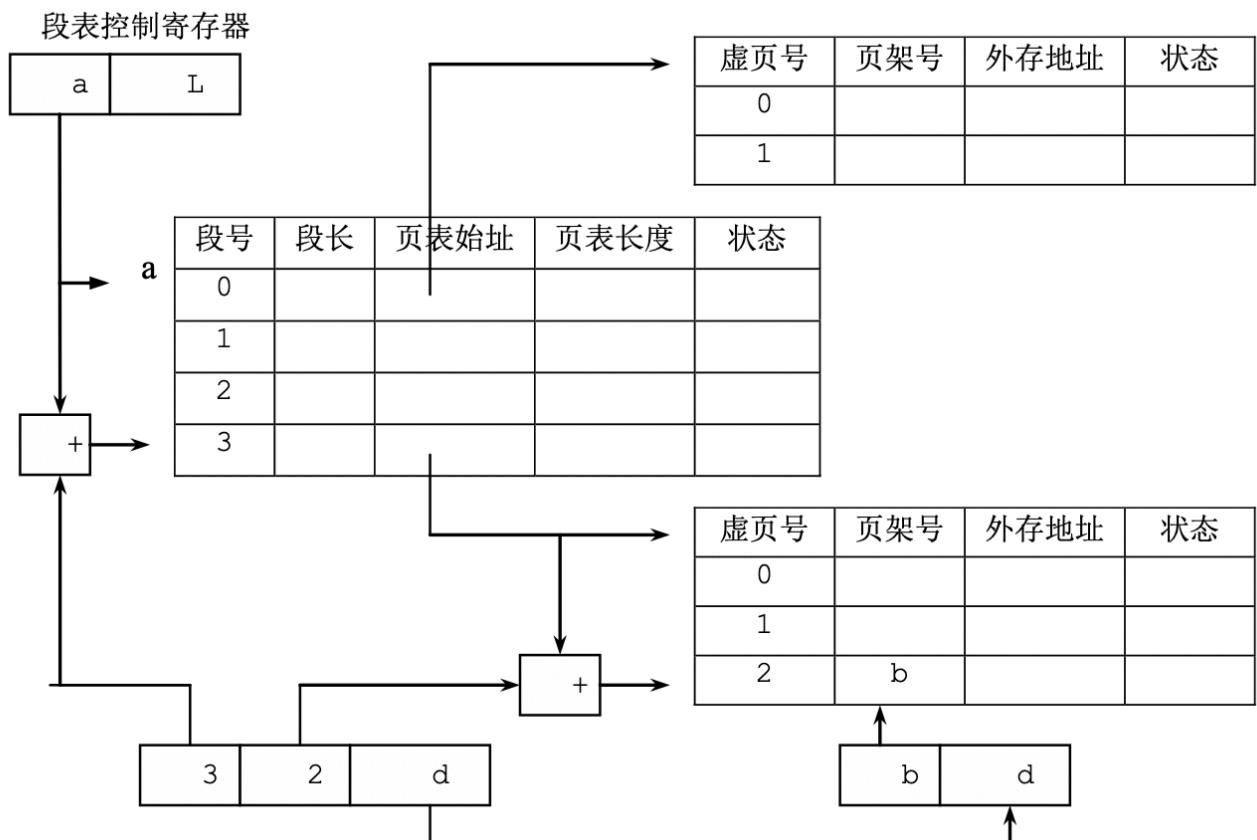
段式管理优点：便于模块化设计、允许分段动态扩展、动态链接、分段的共享和段地址的保护

段页式存储管理的基本思想：将面向用户的程序地址空间分为段，系统为每一段分配和管理实存时再分页

段页式存储管理系统的逻辑地址可分成三个部分：



系统为每一个作业建立一张段表，再为每一段建立一张页表



## 2.9 Linux存储管理

“按需分页”的段页式虚拟存储管理：

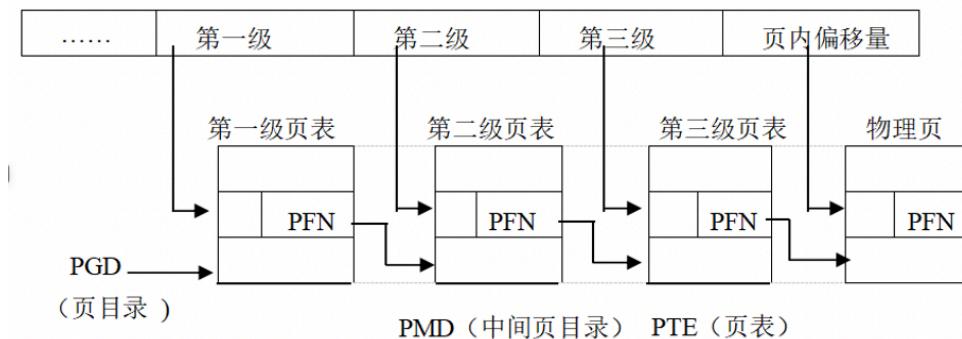
- Linux采用了虚拟存储技术，它使得系统能够同时运行更多的比实际内存能容纳的进程
- Linux的内存管理采用“按需分页”技术实现段式存储管理
- 系统运行时，需要的内容以页面为单位调入内存，暂不执行的则仍留于外存交换区。这样就涉及到页面的管理及页面的换入换出操作

高速缓冲：

- 缓冲区缓存：包括从块设备中读入的和将要写到块设备中的信息
- 页缓存：对于磁盘映像文件，Linux每次装入一页文件，并将读取的页面存储到页缓存中
- 交换缓存：只保存被修改过的页面到交换文件中。如果某页面被写入到交换文件中后未被修改过，则此页面下次从内存中换出时就无需再写入交换文件中，丢弃即可
- 硬件缓存：CPU中有一个常用的硬件高速缓存，它保存页表的内容

Linux中的多级页表：

- 采用三级页表来描述虚拟地址空间和虚拟内存空间的映射关系



Buddy:

1 Mbyte Block		1 M					
请求 100 Kb (A)		A	128K	256K		512K	
请求 240 Kb (B)		A	128K	B		512K	
请求 64 Kb (C)		A	C	64K	B		512K
请求 256 Kb (D)		A	C	64K	B	D	256K
释放 B		A	C	64K	256K	D	256K
释放 A		128K	C	64K	256K	D	256K
请求 75 Kb (E)		E	C	64K	256K	D	256K
释放 C		E	128K	256K		D	256K
释放 E		512K			D	256K	
释放 D		1M					

有哪些地方用到了局部性原理？

### 3 进程管理

## 3.1 进程概述

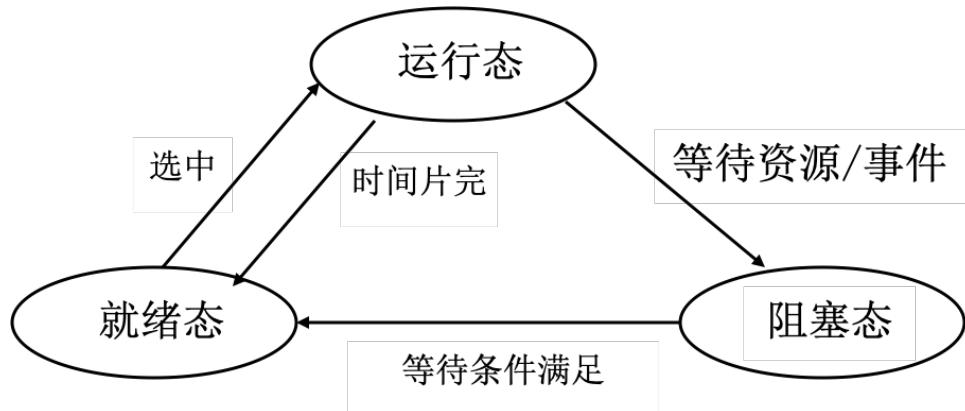
进程：程序处于一个执行环境中在一个数据集上的运行过程，它是系统进行资源分配和调度的一个可并发执行的独立单位

进程是程序的一次动态执行活动，而程序是进程运行的静态描述文本

进程上下文执行环境——进程映像，进程存储器映像：

- 进程控制块PCB，进程基本控制块(常驻内存)，进程扩充控制块(进程不执行时放至交换区)
- 共享正文段(不可修改的代码和常数)(进程执行的程序)
- 数据区(进程执行时用到的数据、非共享程序、全局变量、静态变量)
- 工作区(核心栈、用户栈)

进程的状态及其变化：



- SSLEEP：睡眠状态
- SRUN：执行或就绪状态

## 3.2 进程控制块

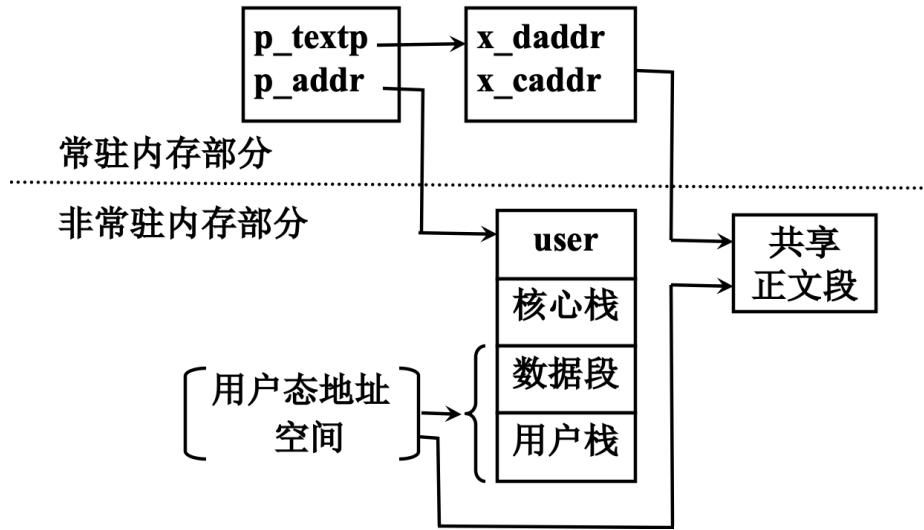
PCB：进程的标识信息、进程的状态、进程的特征、进程的位置和大小、现场保护区、进程通信信息、资源信息、进程间联系、运行管理信息

UNIX中

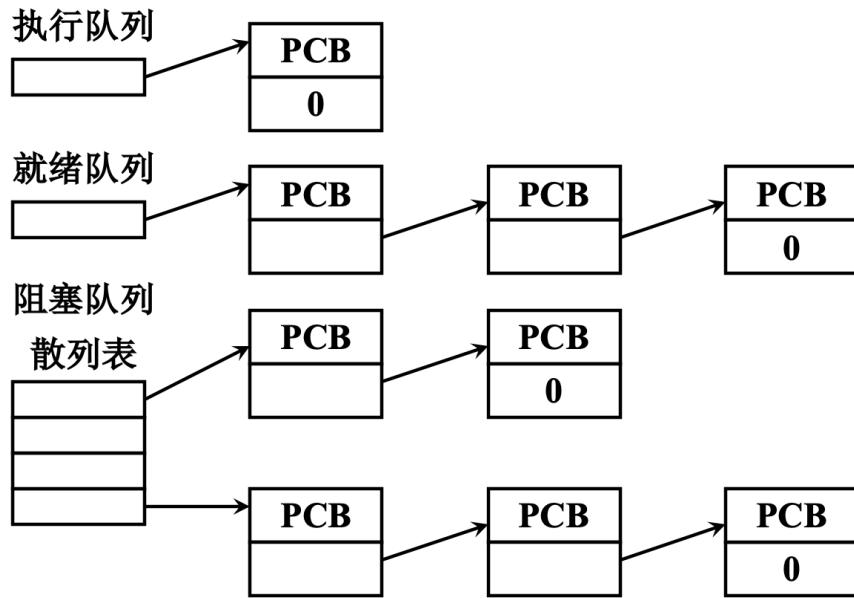
- 常驻内存的PCB部分——proc结构
- 非常驻内存的PCB部分——user结构

系统分配一个控制块text结构，以便于多个进程共享一个可执行程序和常数段(共享正文段)

进程映像的基本结构：



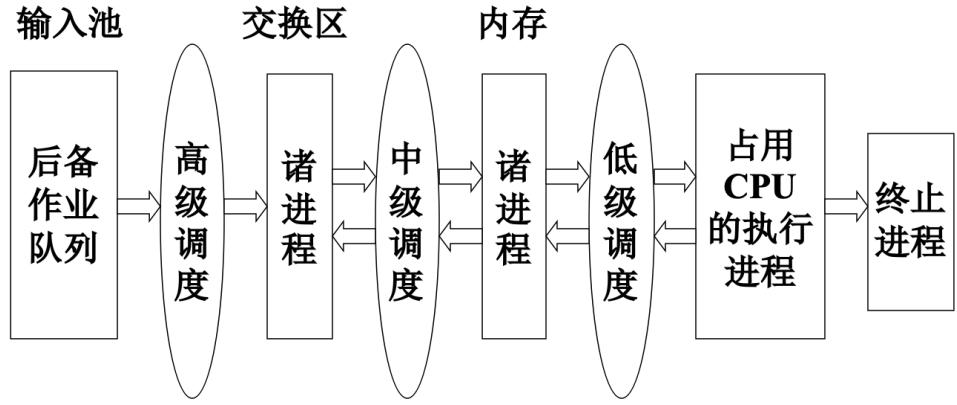
多队列的PCB组织结构：



### 3.3 调度

**三级调度：**

- 高级调度：又称作业调度、长程调度，它决定处于输入池中的哪个后备作业可以调入主系统，成为一个或一组就绪进程
- 中级调度：又称对换调度、中程调度，它决定处于交换区中的就绪进程中哪一个可以调入内存，以便直接参与对CPU的竞争。在内存资源紧张时，将内存中处于阻塞状态的进程调至交换区
- 低级调度：又称进程调度或处理机调度，它决定驻在内存中的哪一个就绪进程可以占用CPU



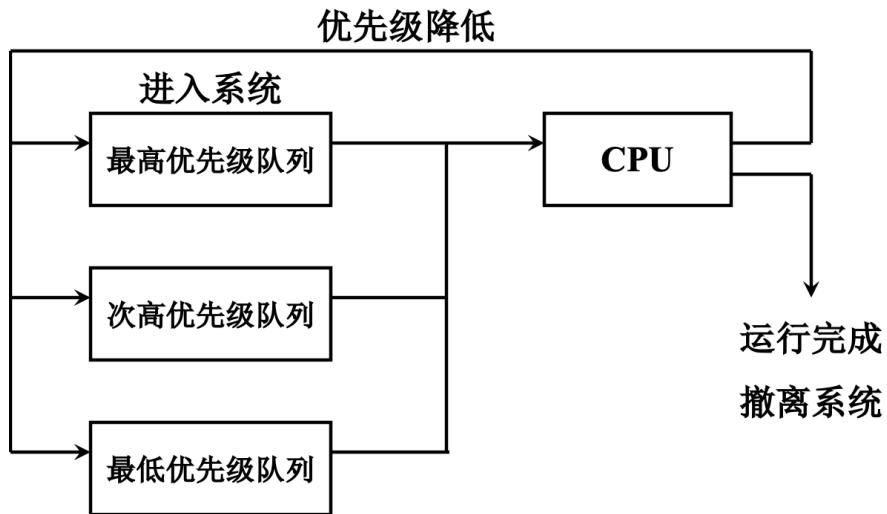
进程切换的两种方式：不可剥夺方式(除非自己放弃)、可剥夺方式

### 调度算法：

- 先来先服务(FIFO)调度算法
  - 按照进程到达就绪队列的时间次序分配处理机，不可抢占式
  - 一个大进程运行时会使后到的小进程等待很长时间，增加了进程平均等待时间
  - 对于I/O繁忙的进程，每进行一次I/O都要等待其它进程一个运行周期结束后才能再次获得处理机，故大大延长了该类作业运行的总时间，也不能有效利用各种外部设备资源
- 时间片轮转法
  - 按进程到达的时间排在一个FIFO就绪队列中，每次选择队首的进程占用处理机并运行一段称为“时间片”的固定时间间隔
  - 适合于交互式分时系统
  - 时间片大小——交互性与切换开销的平衡
  - 可在系统中可设置时间片大小不同的n个队列
  - 小时间片队列为空时，再去调度长时间片队列
- 优先级调度算法
  - 静态优先级法
    - 在进程创建时就赋予一个优先数，在进程运行期间该优先数保持不变
    - 系统进程应当赋予比用户进程高的优先级
    - 短作业的进程可以赋予较高的优先级
    - I/O繁忙的进程应当优先获得CPU
    - 根据用户作业的申请，调整进程的优先级
    - 静态优先权法较适合于实时系统，其优先级可根据事件的紧迫程度事先设定
  - 动态优先级法
    - 反映进程在运行过程中不同阶段的优先级变化情况
    - 进程占用CPU时间越长，就可降低其优先级；反之一个进程在就绪队列中等待的时间越长，就可升高其优先级
    - 也可根据进程在运行阶段占用的系统资源，如内存、外部设备的数量和变化来改变优先级
- 多级反馈队列调度算法
  - 越往下优先级越低，时间片越长
  - 一个新进程进入系统中时，被置于优先级最高的第一个就绪队列尾
  - 当某优先级队列中的程序被选中运行：
    - 若时间片内任务完成，离开系统
    - 若没有用完时间片，因资源等待转变为阻塞状态。待等待完成转为就绪态时，仍回原来的优先级队列尾部

### 级队列

- 若时间片用完还未完成任务，则降低优先级，进入下一优先级就绪队列尾部
- 当一个进程运行时，更高优先级队列中来了新进程，则高优先级进程抢占当前运行进程的处理机。
- 被抢占进程仍在原就绪队列



## 3.4 UNIX系统的进程调度

UNIX的切换调度策略：

- 采用动态优先权算法：在一个适当的时机，选择一个优先权最高，也即优先数（p\_pri）最小的就绪进程，使其占用处理机
- 进程可以处于两种运行状态，即用户态和核心态
- 系统对在用户态下运行的进程，可以根据优先数的大小，对它们进行切换调度
- 对在核心态下运行的进程，一般不会对它们进行切换调度，即UNIX核心本质上是不可重入的

优先数的计算：

- 对用户态的进程，核心在适当时机用下列公式计算进程的优先数：
  - $p\_pri = p\_cpu/2 + p\_nice + PUSER + NZERO$
- $p\_cpu$ 是进程占用处理机的量度，每次时钟中断，使当前执行进程的 $p\_cpu$ 加1，但最多加到80；每1秒钟使所有就绪状态进程的 $p\_cpu$ 衰减一半
- 形成一个负反馈的过程，使用户态的诸进程能比较均衡地使用处理机
- $p\_nice$ 是用户设置的进程优先数偏置值，普通用户只能降优先级，超级用户可以升优先级
- $PUSER$ 和 $NZERO$ 是两个常量，用于分界不同类型的优先数

优先数的设置：

- 在核心态下运行的进程，不会被强迫剥夺处理机。只有当它因等待系统资源等原因进入睡眠状态时，系统才分配给其一个与事件相关的优先数。只有当它被唤醒之后，才以设置的优先数参与竞争处理机
- 核心态进程根据被设置优先数的大小，可分为可中断和不可中断两类优先级

UNIX进程优先级的排列：

- $NZERO$  (25) ——可中断和不可中断的分界常数
- $PUSER$  (60) ——核心态与用户态的分界常数

进程切换调度的时机：

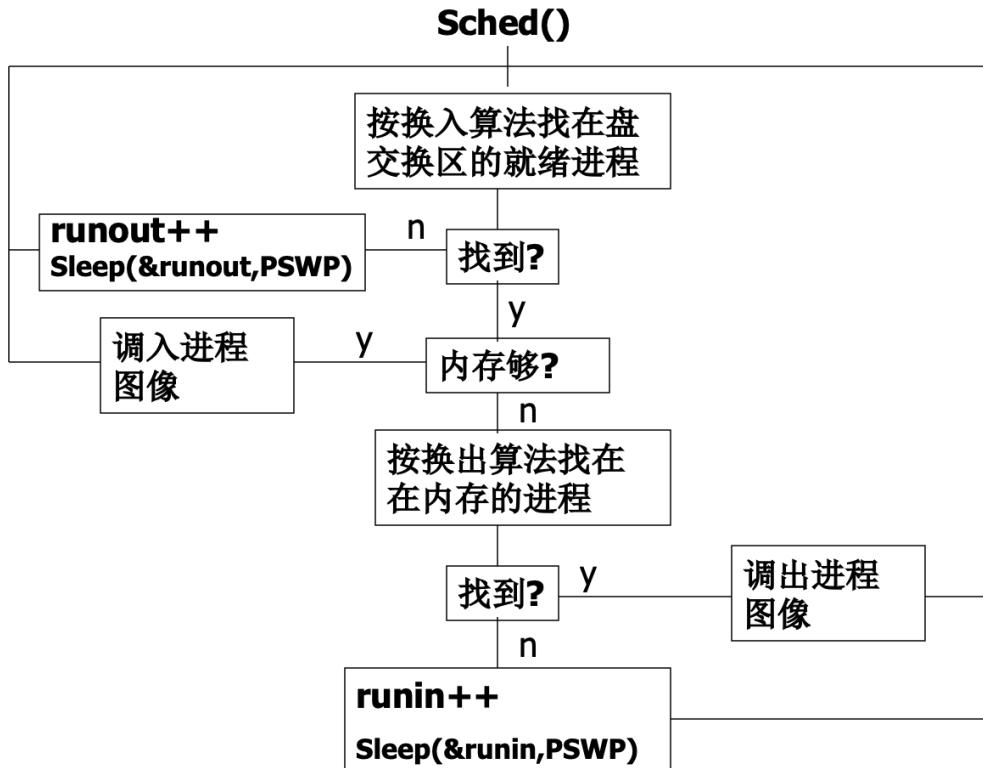
- 进程自愿放弃处理机而引起切换调度
- 系统发现可能更适合占用处理机的进程，设置了强迫调度标志runrun

切换调度程序：

- 实施进程切换调度的程序是swtch，其主要任务：
  - 保存现运行进程的现场信息
  - 在就绪队列中选择一个在内存且优先数 p\_pri 最小的进程，使其占用处理机，如找不到这样的进程，机器就空转等待
  - 为新选中的进程恢复现场

进程的对换调度：

- 在磁盘开辟一块空间——进程映像的交换区，作为内存的逻辑扩充
- 0号进程的任务是按照换入换出算法在内存和盘交换区之间传输进程的映像
- 实现进程映像的函数是sched()
- 换入算法：
  - 找在盘交换区的就绪进程，按它们在外存驻留时间p\_time从长到短的次序逐个换入内存，直至全部调入或内存无足够空闲区为止
  - 当0号进程将交换区中的就绪进程全部调入内存后，它就暂时无事可干，将全局标志变量runout置位，进入睡眠状态
  - 换入过程中如发现内存无足够空间，则要将内存中的进程换出，以便为要换入的进程腾出空间
- 换出算法：
  - 先将部分映像已换出的进程从内存中完全换出，然后考虑处于睡眠或被跟踪的进程，最后是在内存驻留时间最长的进程，包括就绪状态的进程，但p\_flag中包括SSYS或SLOCK的进程不能换出
  - 考虑换出最后一类进程时，要求换入进程在交换区驻留时间应大于3秒，换出进程在内存驻留时间应大于2秒
  - 当找不到合适的换出进程时，0号进程就将全局标志runin置位，进入睡眠状态
  - runin标志置位表示盘交换区有就绪的进程要调入内存，但内存没空，且无合适的进程可换出
- 换入是主动的，是目的；换出是被动的，是权宜之计
- 当系统中有一个处于盘交换区的进程被唤醒后，若runout已被设置，则要唤醒0号进程，以使它可以继续执行调入操作
- 如一个进程在进入睡眠状态时，要对runin进行测试，如该标志已被设备，则要唤醒0号进程，使它能够执行调出操作，以便为盘交换区要进入内存的进程腾出空间



## 3.5 进程的控制

进程阻塞的原因：

- 请求系统服务或请求分配资源时得不到满足
- 同步等待I/O的完成，由中断处理程序解除该进程的阻塞状态
- 进程间互相配合共同完成一个任务时，同步等待
- 一进程在等待某一进程发消息
- 进程将自己挂起或进程将自己某个子进程挂起
  - 挂起是对进程进行控制的手段，如原进程处于就绪状态，则转变为静止就绪状态；如原进程处于阻塞状态，就转变为静止阻塞状态

进程睡眠：

- UNIX中阻塞状态称为睡眠状态
- 核心中，进程通过调用sleep程序自愿地进入睡眠状态，其调用形式是：
  - sleep(caddr\_t chan, int disp);
  - chan是睡眠原因，实际上是与睡眠事件相关的变量或数据结构的地址
  - disp的低7位是根据睡眠原因的缓急程度对睡眠进程设置的优先数，被唤醒后参与竞争处理机时起作用

定时睡眠：

- sleep(chan, disp)是内核函数，用户不能直接调用，UNIX向用户另外提供了一个系统调用：
  - sleep(seconds)

进程唤醒：

- 当引起进程阻塞的原因消除后，核心就可将阻塞进程唤醒，唤醒的方法是根据睡眠原因先算出对应的阻塞队列
- 不同的原因可能挂在同一个队列中，在对应的hash队列中还要核查睡眠原因
- 可能有多个进程因同一原因而阻塞，一般将这些进程全部唤醒，将它们排到就绪队列中，使其有被调度的资格
- 同一原因而唤醒的进程将先竞争处理机，从而可获得有关的资源
- 调用形式：
  - wakeup(chan)
- 唤醒睡眠进程时，不区别进程是处在内存还是盘交换区

进程终止：

- 一个进程在完成了任务后，可用系统调用exit(int status)终止自己
  - status左移8位后作为传送给父进程的参数
- 进程调用exit时，关闭所有的打开文件，释放共享正文段、本进程的数据段、用户栈和核心栈的存储空间，暂时保留proc结构和盘交换区的user结构副本，进程的状态改为SZOMB状态
- 进程终止时如有父进程因等待子进程的终止而处于睡眠状态，就唤醒父进程，最后调用swtch程序重新调度
- 父进程对进入SZOMB的子进程作善后处理后，释放该子进程的一切资源，使其生命期最后被终止

父进程等待子进程终止：

- 创建子进程的父进程可以通过系统调用wait等待它的一个子进程终止：
  - pid = wait(&status)
- 通过返回值pid可获得终止进程的进程标识数
- status的高位部分为子进程传给父进程的参数，低8位部分为核心设置的系统调用状态码
- 如父进程在调用wait时，子进程已先期终止了，那么对子进程作善后处理后，立即返回

## 3.6 进程的创建和映像改换

---

创建进程的主要步骤：

- 为新进程分配唯一的进程标识数，接着为新进程分配进程控制块的空间，在进程表中加入新项
- 为新进程分配进程各部分映像所需的内存空间
- 初始化进程控制块，根据进程的性质或缺省值初始化进程的控制信息，运行状态一般初始化为就绪状态
- 子进程复制父进程扩充控制块，子进程将共享父进程的全部打开文件、信号处理方式等
- 子进程复制了父进程的数据区、核心栈和用户栈，父子进程程序执行的当前位置、状态、数据区、变量的当前值都是相同的，但随着父、子进程的各自独立执行，子进程的映像将会与父进程有明显的差异

fork：

- 在执行系统调用fork后，父进程得到的返回值是所创建子进程的标识数，而子进程的返回值为0

```

main()
{
    int pid;
    printf("Before fork\n");
    pid=fork();
    if (pid) {
        printf ("Parent process: PID= %d\n", getpid());
        printf ("Produced child's PID= %d\n", pid);
    } else {
        printf ("Child process: PID= %d\n", getpid());
    }
    printf("Parent or child process: PID= %d\n",getpid());
}

```

进程映像的改换：

- fork之前的图像已由父进程执行过了，子进程是不会再从头开始执行的，因而这部分图像的存在对子进程是毫无意义的；fork之后父进程所执行的图像部分的存在对子进程来说也是一个浪费，父进程也有相似的问题
- 进程映像改换的exec系列的系统调用：
  - 用一个可执行文件中的程序和数据取代当前正在运行的程序和数据，从而使主调进程的映像改换成新的映像
  - 尽管新执行的程序与原进程的执行程序完全不同，但该调用并不形成新进程，因为原进程的proc结构和user结构并不改换，其进程标识数p\_pid与主调进程相同，与父进程的关系也没有改变
  - execl系统调用：
    - 用于装入一个带路径的可执行文件，用新程序覆盖老程序，然后运行这个新程序，老进程
    - 一般情况下此系统调用不返回到主调程序，仅当调用出错时（如不存在指定的可执行文件），系统才返回错误码
    - execl(pathname, cmdname, arg1, arg2, ..., (char\*)0);
  - execv系统调用：
    - execv(pathname, argv);
  - execl和execv对应的另两个系统调用是execlp和execvp，第一参数不必带路径

系统调用fork和execl的使用示例：

```

main()
{
    int i, pid, status = 1;
    printf("Before fork call.\n");
    while((i=fork( ))== -1);
    if (i) /* 父进程 */
        printf("It is the parent process.\n");
        pid =wait(&status);
        printf("Child process %d, status =%d \n", pid, status);
    } else /* 子进程 */
        printf("It is the child process.\n");

```

```

        execl("/bin/ls", "ls", "-l", (char*)0); /* 映象改换 */
        printf("execl error.\n");           /* 映象改换失败 */
        exit (2);
    }
    printf ("Parent process finish. \n");
}

```

## 3.7 线程

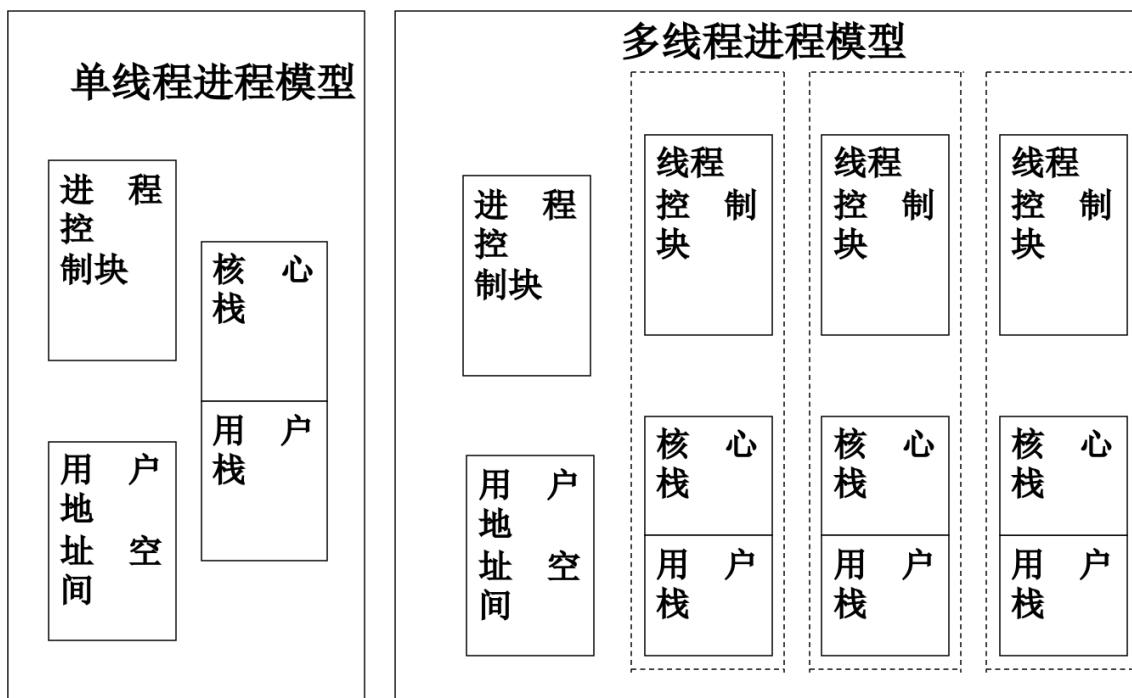
进程包含下列两个特征：

- 资源拥有单位：进程的虚址空间用于驻留进程的映像，进程可以分配到主存和控制其它的系统资源
- 调度单位：进程是可以并发执行的独立单元，是操作系统进行调度的一个实体

线程：一种新的调度和执行的实体单元，原先的资源拥有单位通常还是称为进程或任务

多线程：操作系统支持在单个进程中执行多个线程的能力

- 进程被定义为保护单位和资源分配单位
- 每一个线程具有如下特征：
  - 线程的执行状态（运行、就绪等）
  - 线程上下文环境，可以把线程看成是进程内一个独立的程序计数器的运转
  - 执行栈
  - 存取所属进程内的主存和其它资源，在本进程的范围内与所有线程共享这些资源



- 优点：
  - 提高了系统性能，在一个现存的进程中创建或终止一个线程的时间很小
  - 在同一个进程内部两个线程的切换开销比进程之间的切换开销小得多
  - 服务器使用线程来提高效率，当一个文件服务请求到达时，由于服务器能处理很多请求，在一个短时间内，很多线程将被产生和销毁

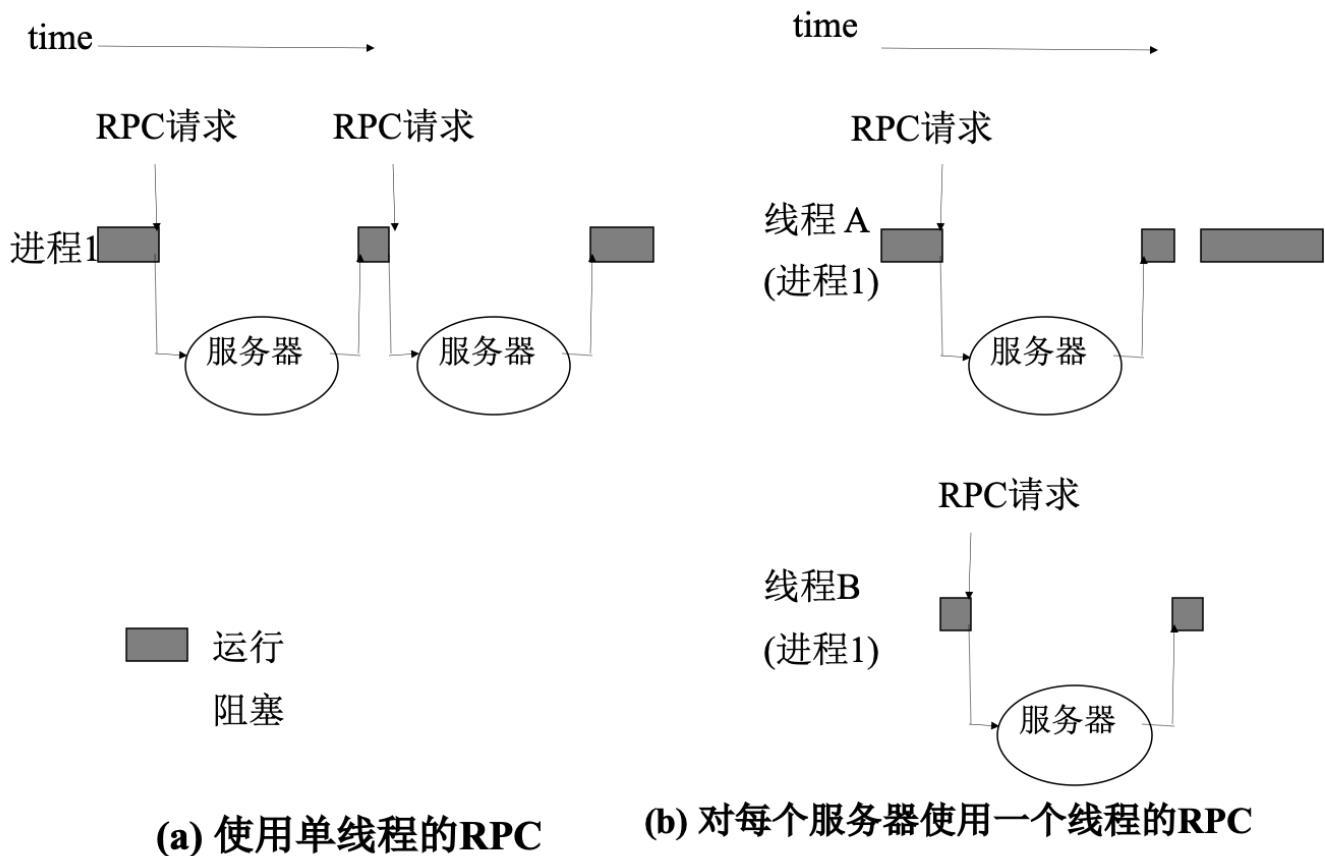
- 线程能有效增加不同执行程序之间的通信效率，同一进程中线程共享内存

线程的状态：运行、就绪、阻塞（一般没有挂起）

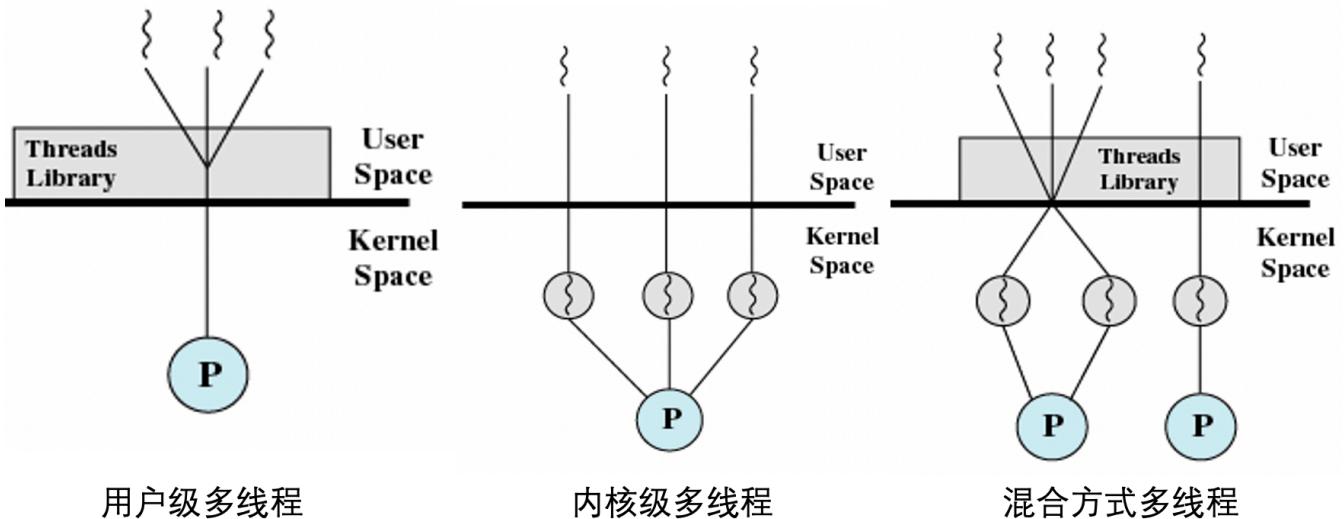
线程的操作：创建、阻塞、解除阻塞、终止

线程的并发：

- 执行两个对不同的主机的远程过程调用（RPCs）以获得的组合的结果
- 在单线程程序，这两个结果是顺序获得的，故程序要依次等待每一个服务器的响应
- 如用各自的线程执行RPC以获得调用结果的方法重写程序，性能就获得实质性的提高
- 程序是并行地等待两个主机的响应结果



主要的线程实现方式：



- **用户级多线程:**
  - 管理线程的所有工作由应用程序完成，内核感觉不到这类线程
  - 应用程序使用线程库进行多线程程序设计
  - 优点:
    - 线程切换无需内核级特权，减少模式切换开销
    - 调度程序能够面向特定的应用系统
    - 能运行于任何OS
- **内核级多线程:**
  - 管理线程的所有工作由内核完成
  - 优点:
    - 内核可以同时调度一个进程中的多个线程在多处理机上运行
    - 如果进程中的一个线程被阻塞，内核可调度同一进程中的其它线程
    - 内核中的子程序本身也可设计为多线程
- **混合方式多线程:**
  - 线程的创建完全在用户空间进行
  - 一个应用程序内大多数的调度和线程同步在用户空间进行
  - 同一个应用程序中的多线程能够在多处理器上并行运行
  - 一个阻塞的系统调用不需要阻塞整个进程

## 4 进程通信

### 4.1 进程的同步与互斥

同步——两个或两个以上的进程要协作完成一个任务，它们之间就要互相配合，需要在某些动作之间进行同步

互斥——一般发生在两个或两个以上的进程竞争某些同时只能被一个进程使用的资源的情况下

临界资源——在一段时间内只能允许一个进程访问的资源

临界段或互斥段——进程执行的访问临界资源的程序段

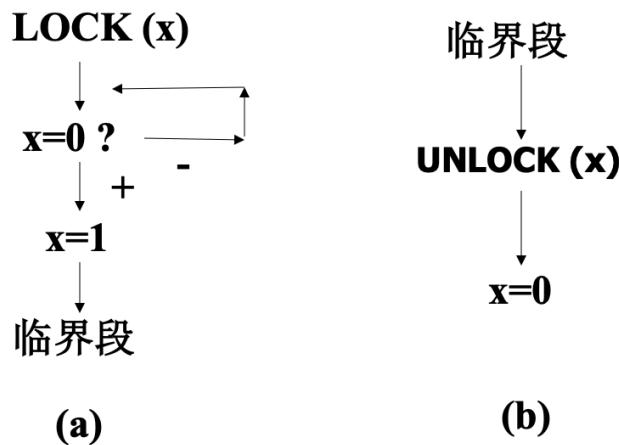
临界段的调度原则：

- 在所有共享相同资源或对象的临界段中，每次只能允许一个进程进入
- 一个进程在非临界段中的暂停运行不能影响其它进程
- 一个进程如需要进入临界段，不能发生无限延迟的情况，即既不会死锁，也不会饥饿
- 无进程在临界段时，必须让任何希望进入该程序段的进程无延迟地进入
- 一个进程只能在临界段内停留有限的时间
- 对于相关进程的运行速度和处理机的数量不做假设

## 4.2 进程间互斥控制方法

锁可以用于控制临界段的互斥执行：

- $x = 0$ : 锁的打开状态
- $x = 1$ : 锁的关闭状态



锁的关闭操作LOCK包括测试和关闭两个操作步骤，这两个操作步骤涉及临界资源x，故这段程序也是临界段

- 不安全，两重锁无法彻底解决问题

彻底的解决方法：

- 测试并设置指令test&set
  - 专门的锁操作指令test&set，该指令首先测试锁变量的值，如为1，则重复执行本指令；如为0，则立即将锁变量的值置为1
  - test&set是一条完整的指令，而在一条指令的执行中间是不会被中断的，故保证了锁的测试和关闭操作的连续性
- 交换指令exchange

```

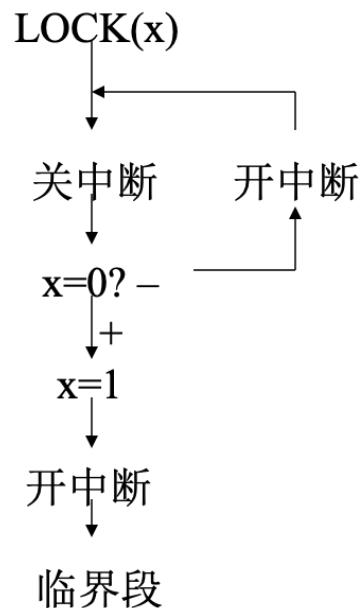
check:    MOV AL, 1           ; 置测试单元寄存器AL的值为1
          LOCK XCHG X, AL   ; 在本指令的执行时封锁总线，交换锁变量x与测试单元的值
          TEST AL, AL        ; 测试AL的值
          JNZ check         ; 如AL非0，即原锁处于关闭状态，跳转至check，重复测试过程
          临界段             ; 锁变量x已置为1

```

- 开、关中断法
  - 用硬件锁，当一个进程进入临界段，只要不自行挂起，就一直执行
  - 只能用于单CPU系统、长时间关闭中断影响系统处理紧迫事件、禁止了其它无关的进程进入不同的临界段



- 用硬件锁锁软件锁，用软件锁锁临界段



## 4.3 信号量和semWait、semSignal操作

前述锁操作缺点：

- 对于不能立即进入临界段的进程，需循环忙等
- 饥饿问题，一个离开，多个竞争
- 死锁问题，P1在临界段，执行期间中断，优先级更高的P2占用CPU，访问相同的临界资源

两个或两个以上的进程能用简单的信号工具实现互相协作

一个进程被强制停留在特定的地方，直至它收到了特定的信号才能继续执行下去

为了通过信号量s传递信号，一个进程要执行semSignal(s)原语，V操作

为了使信号量接收一个信号，进程要执行semWait(s)原语，P操作

如果相应的信号还未发送过来，执行semWait(s)的进程就被挂起，直至接收到信号才能恢复执行

信号量定义成具有整型值，并能对其施加以下三种操作的变量，除了这三种操作之外的任何操作都不能测试和处理信号量的值：

- 初始化操作，信号量能初始化为非负的值
- semWait操作，能减小信号量的值，如结果值为负，执行semWait操作的进程就被封锁
- semSignal操作，能增加信号量的值，如果结果值非正，那么原先因执行semWait操作而阻塞的进程被解除阻塞

PV操作为原子操作，不能被中断

```

typedef struct semaphore {
    int      value;
    Queue   queue;
} Semaphore;
Semaphore s;

void  semWait(s)
semaphore s ;
{
    if  ( --s.value < 0 ) {
        将进程置入等待队列queue中;
        封锁进程;
        转进程调度程序;
    }
}

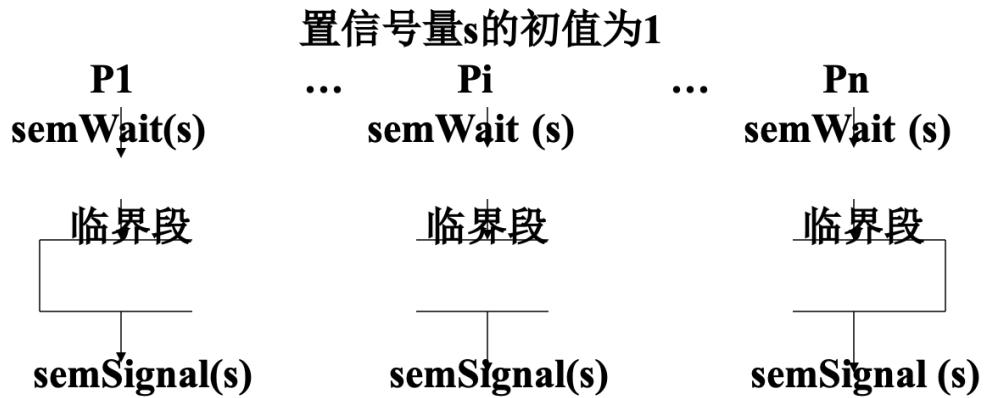
void  semSignal(s)
semaphore s;
{
    if  ( ++s.value <= 0 ) {
        从等待队列queue中移出一进程;
        将该进程置入就绪队列中;
    }
}

```

## 4.4 信号量的应用

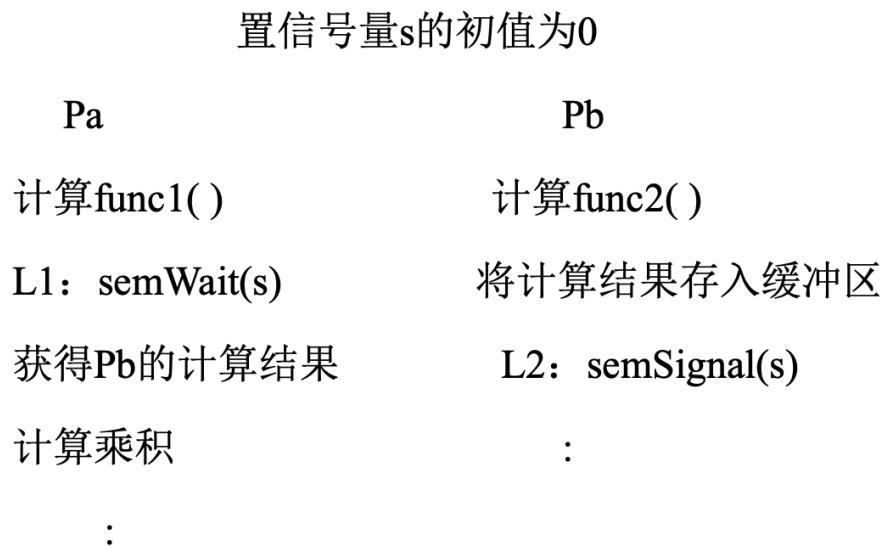
利用信号量实现互斥：

- 为一临界段设置一个信号量s，初值为1
- 进程进入临界段前对s执行semWait操作，若s=0，则允许进入临界段；若s<0，则进程阻塞
- 进程退出临界段，对s执行semSignal操作，使等待队列中的一个挂起进程转为就绪态，使它可以进入临界段
- 如有n个并发进程涉及一个临界段，则s的取值为i， $-(n-1) \leq i \leq -1$ ，表示当前有|i|个进程被阻塞



阻塞/唤醒协议：

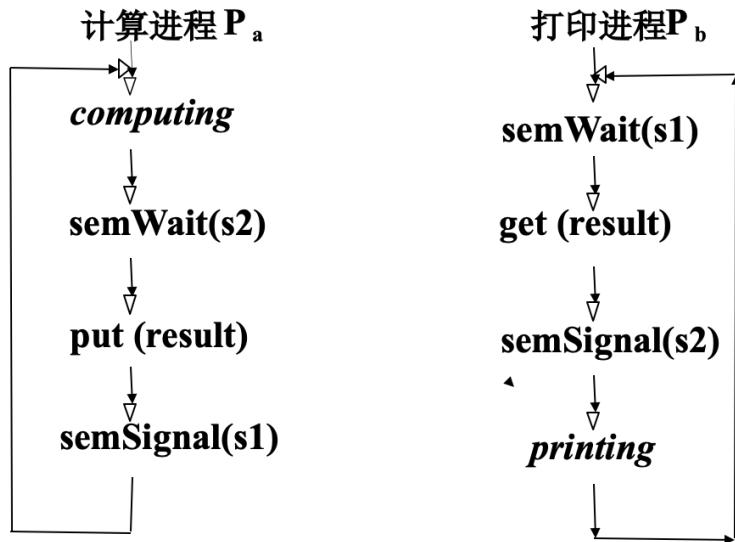
- 当进程需要等待相关的协同进程完成某个协作任务，可以对初值为0的s执行semWait操作，以使自己在此阻塞
- 当协同进程完成了协作任务，应执行semSignal操作，以使等待进程可以继续运行
- a等b， b不用等a——半同步



两个进程间的同步：

- 设有两个进程Pa和Pb：进程Pa每次执行一个计算，并将结果存入一个缓冲区；进程Pb则从缓冲区中取出结果，并将结果打印出来
- 为了实现计算进程和打印进程之间的相互同步，就需要设置2个信号量S1和S2
- S1表示缓冲区中是否已存入进程Pa的计算结果，也即通知进程Pb是否可取出缓冲区中的信息并送往打印机
  - 0: Pa没存入新的计算结果
  - 1: Pa已存入新的计算结果
- S2表示缓冲区中的结果是否已被进程Pb取去，也即通知进程Pa是否可将新的计算结果再存入缓冲区
  - 0: Pb没取走缓冲区中的数据，缓冲区满
  - 1: Pb已取走缓冲区中的数据，缓冲区空
- 信号量的初值可设置为：
  - S1为0：缓冲区还未存入数据

- S2为1：缓冲空闲（相当于Pb已取走数据）



互斥和同步对信号量操作方法的差异：

- 互斥实现是进程对同一信号量进行semWait、semSignal 操作，一个进程在成功地对信号量执行了semWait操作后进入临界段，并在退出临界段后，由该进程本身对这信号量执行semSignal操作，表示没有进程处于临界段，可让其他进程进入
- 同步的实现由一个进程Pa对一个信号量进行semWait操作后，只能由另一个进程Pb对同一个信号量进行semSignal操作，使Pa能继续前进，在这种情况下，进程Pa要同步等待Pb。如进程Pb也要同步等待Pa，则要设置另一个信号量

### 生产者和消费者问题：

- 信号量的非负值表示可供分配的资源数目或可存放数据的缓冲区数目
- 生产者和消费者问题是通过有限的缓冲区将一群生产者P1, P2...Pk和一群消费者C1,C2...Cm联系起来。可设信号量
  - buffers的初值为n，表示空闲的缓冲区数目
  - products的初值为0，表示已存入缓冲区的产品数目
- 生产者和消费者问题的一般过程为：
  - 生产者在执行semWait(buffers)之后，只要buffers≥0(还有空闲的缓冲区)，就可将产品送入
  - 消费者在执行semWait(products)后，只要products≥0(产品还未取完)，就可以从缓冲区中取走产品
  - 否则，生产者或消费者进程就被阻塞

```

producers:
while ( ) {
    produce next product;
    semWait (buffers);
    semWait (mutex);
    Put product into buffers;
    semSignal (mutex);
    semSignal (products);
}
  
```

```

customers:
  
```

```

while ( ) {
    semWait (products);
    semWait (mutex);
    get product from buffers;
    semSignal (mutex);
    semSignal (buffers)
    consume product;
}

```

### Example

假设有3台打印机供多个进程共享打印。打印机编号分别为1, 2, 3。请使用信号量P, V操作实现申请和释放打印机的函数 (用类C语言) : int AllocPrinter(), void FreePrinter(int index)。要求如下:

(1) int AllocPrinter() : 分配打印机函数。返回值为打印机的编号。如果没有空闲的打印机, 则应该申请进程阻塞。

(2) void FreePrinter(int index): 释放打印机函数。输入参数index为打印机编号。

(3) 详细说明信号量的含义和初始值。

```

bool print[3] = {0};
int value = 3;
int mutex = 1;

int AllocPrinter(){
    int temp;
    P(value);
    P(mutex);
    if(!print[0]){
        print[0] = 1;
        temp = 1;
    } else if(!print[1]){
        print[1] = 1;
        temp = 2;
    } else{
        print[2] = 1;
        temp = 3;
    }
    V(mutex);
    return temp;
}

void FreePrinter(int index){
    P(mutex);
    print[index] = 0;
    V(mutex);
    V(value);
}

```

## 4.5 进程间的数据通信

前述操作只是决定进程是否继续执行下去，是低级进程通信

消息通信：

- 基本思想是由系统的消息通信机构统一管理一组空闲的消息缓冲区
- 一个进程要向另一个进程发送消息，先要向系统申请一个缓冲区，填写了消息正文和其它有关消息的特征、控制信息后，通过消息通信机构将该消息送到接收进程的消息队列中
- 接收进程在一个适当时机从消息队列中移出一个消息，读取所有的信息后，再释放消息缓冲区
- 一个消息缓冲区的数据结构中除了要包括消息的正文外，一般还要包括其他有关的控制信息
  - send\_pid：发送进程标识
  - type：消息类型
  - size：消息长度
  - next\_ptr：下一个消息的指针
  - text []：消息正文
- 发送进程在发送消息之前，先要在进程自己的内存空间中开辟一个发送缓冲区，将消息正文及有关控制信息填入其中，再调用发送消息的系统调用msgsnd(sm\_ptr)，其中参数sm\_ptr指向进程的发送缓冲区始址
- 接收进程在接收消息之前，也先要在进程自己的内存空间中开辟一个接收缓冲区，再调用接收消息的系统调用msgrecv(rm\_ptr)，其中参数rm\_ptr指向接收缓冲区始址

共享存储区：

- 信息通信效率不高
- 共享存储区机制可以把内存中的一个区域连入多个进程的虚地址空间
- 一个进程可以分配多个共享存储区
- 使用共享存储区进行通信时进程间的互斥或同步要靠其它机构解决

管道通信：

- 一种信息流缓冲机构，用于连接发送进程和接收进程，以实现它们之间的数据通信
- 以先入先出（FIFO）的方式组织数据的传输
- 在发送进程和接收进程之间能传递任意大的信息，但在实现时所开的缓冲区大小是有限的
- 当管道写满时，发送进程就被阻塞，只有当接收进程从管道中读出一部分或全部信息后，发送进程才能继续向管道写信息
- 反之也一样，当接收进程读空管道时，就要等待发送进程继续将信息写入管道
- 在UNIX中，管道是以文件为基础，再适当考虑其特殊要求而实现的通信机构

## 4.6 软中断和信号机构

信号的概念：

- 一组软中断信号，由于模拟硬件中断
- 信号由事件引起，事件的来源可以有异常、终端中断、通知、报警
- 信号是一取值为1~19 (MAX\_SIGS) 的某个整数，可以在进程之间传送，用于通知进程发生了某种异常事件，需要执行事先安排好的动作
- 信号与中断的不同：
  - 中断有优先级，信号没有
  - 中断程序运行于内核态，信号处理程序不一定

- 中断响应一般是及时的，而信号需要进程在运行中的某几个时机主动询问和做出响应，延时长
- 信号的产生：

- 在UNIX中，主要在以下几种情况下向进程发送软中断信号：
  - 在用户态运行时产生了各种软、硬件故障
  - 用户通过键盘按键向与该终端有关的进程发信号
  - 进程之间通过系统调用kill传递信号
  - 进程写管道时发生异常
  - 父进程对子进程进行跟踪
- 信号机构将发给进程的信号存放在该进程proc结构的p\_sig项
- 进程收到信号后并不立即进行处理，只有当进程从核心态即将返回到用户态时，即系统调用、陷入或中断返回时才检查p\_sig项，并处理与信号对应的事件
- 如一个进程处于较低优先级的睡眠状态，那么系统将唤醒该进程，使其转入就绪状态，并在被调度程序选中，转入执行状态时执行信号处理程序

信号类型：

- 9——SIGKILL——终止进程——无条件终止进程

信号的处理方式：

- 每一个进程的user结构中有一个长度为20的数组signal[20]，以信号类型作为该数组的下标索引，元素的值决定了对应信号的处理方式。信号的处理方式有三类：
  - 若数组元素值为0，则执行信号机构定义的缺省动作
  - 若数组元素值为1（或奇数），则忽略该信号，不执行任何动作
  - 若数组元素值为偶数（函数入口地址），则作为对应信号处理程序的指针
- 一个进程在创建时，继承了父进程所有的信号处理方式，即其signal[NSIG]各元素的值与父进程完全相同。但此后除了SIGKIL外，信号表中定义的信号处理方式都可以用系统调用signal (sig, func) 设置或修改

信号的传送：

- 利用信号实施进程间通信的主要方式是使用系统调用kill (pid, sig)，其功能是将信号sig传送给由参数pid限定的进程。当：
  - pid为正值时，对应于一个有效的进程标识数，该信号就发送给这个唯一的进程
  - pid为0时，将信号发送给受同一终端控制的所有进程
  - pid为-1时，将信号发送给与发送进程用户标识数相同的所有进程
  - pid < -1时，将信号发送给组标识数为pid的绝对值的所有进程

```
#include <sys/types.h>
#include <signal.h>

int main ( )
{
    int status;
    pid_t pid;
    void func ( );
    signal (SIGUSR1,func); /* 预置信号处理程序 */
```

```

if (pid=fork ()) {
    printf ("Parent: will send signal.\n");
    kill (pid, SIGUSR1);           /* 发送信号 */
    wait (& status);             /* 等待子进程停止 */
    printf ("status=%d: Parent finish:\n", status); 10
} else {
    sleep (10);                  /* 等待接受信号 */
    printf ("Child: signal is received.\n"); 8
    exit (0);
}
}

void func ()
{
    printf ("It is signal processing function.\n");
}

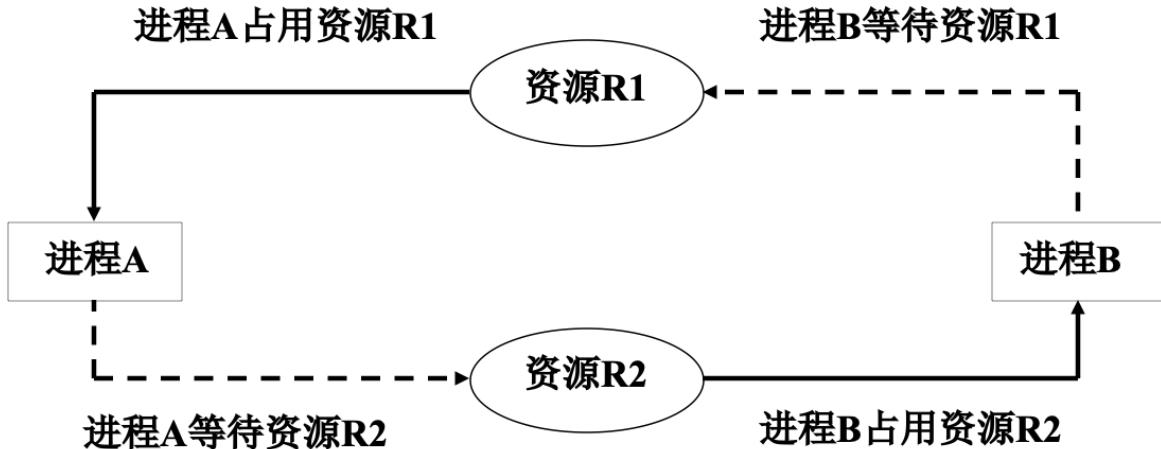
```

- 在程序的开始部分用系统调用设置信号16的处理方式为执行func程序，在父进程用fork创建子进程后，子进程继承了对信号的处理方式。父进程向子进程发送信号后，如子进程处于低优先权睡眠，则将其唤醒。子进程被唤醒后，检查是否收到信号，发现已收到信号，就执行该信号（SIGUSR1）所对应的处理程序func ()。执行完毕后返回，继续执行余下程序段

## 4.7 死锁

产生死锁的原因：

- 当两个进程各占了对方所要的一个资源，就会形成死锁



- 系统资源可分为两类：
  - 可重复使用的永久性资源
    - 处理机、主存、暂存、I/O通道、打印机以及文件、数据库等
    - 可重用的资源又可分为可剥夺的资源（处理机）和不可剥夺的资源（打印机）
    - 涉及到可重用资源的死锁例子：
      - 一个进程占用了打印机，又要申请磁带机，另一个进程占用了磁带机，又要申请打印机，每个进程都占用并保持了一个资源，并等待对方所占用的资源时就发生了死锁

- 会被消耗的临时性资源
  - 中断、信号、消息、I/O缓冲区中的信息等

### 产生死锁的条件：

- 同时具备下列三个静态的必要条件时，才有可能产生死锁：
  - 互斥执行：每次只能允许一个进程占有和使用一个资源，其他申请该资源的进程被阻塞
  - 保持并等待：当进程等待分配给它新的资源时，保持占有已分配的资源
  - 不可剥夺：不能强迫移去进程占有的未使用完的资源
- 上述这三个条件是产生死锁的必要条件，但即使存在全部这三个条件也不一定会发生死锁。要产生死锁必须存在第四个动态条件：
  - 循环等待：存在一个闭合的进程——资源链，以致每一个进程至少占有链中下一个进程所需要的一个资源
    - 实际隐含了前3个条件

### 死锁的预防：

- 间接方法：避免前三个必要条件中的某一个
- 直接方法：避免发生第四个条件
- 互斥执行：一般来说，此条件不能排除，如果存取一个资源需要互斥执行，那么操作系统就要支持互斥执行
- 保持和等待：能预防，只要进程一次申请它所需要的所有的资源，在所有的需要同时满足以前，阻塞自己
- 不可剥夺：有几种方法可预防这个条件：
  - 如占有某些资源的进程不能获得进一步的资源，该进程必须释放原先所占有的资源；如果需要，以后再申请这些资源
  - 如果一个进程需要申请当前正被其它进程占用的资源，操作系统就要求后者释放它所占用的这类资源，这种预防死锁的方法只能用在后申请资源的进程优先级较高的情况下
  - 只有当资源的状态容易保存和便于以后恢复的情况下，这种方法才是实际可行的。处理机就是这类资源的例子，如剥夺像打印机那样的资源，就会使输出变得杂乱无章。但借助Spooling技术可将独享设备改为虚拟的共享设备，就能破坏本条件，预防死锁
- 循环等待：采用有序资源使用法可以防止循环等待条件
  - 如果一个进程已经分配了类型R的资源，那么以后它只能申请在资源顺序表中排在R后面的资源类型
  - 经常使用的普通资源低序号，数量少的贵重资源高序号

1	2	3	4	5
数 / 模 转 换 器	磁 带 机	打 印 机	光 刻 机	绘 图 仪

### 死锁的避免：

- 死锁预防中，要对资源的申请加以限制，以预防四个死锁条件中的至少一个，降低了资源使用效率和进程执行速度
- 死锁避免的方法允许三个死锁的必要条件都存在，但要动态地进行审慎的判断，以保证运行不会到达死锁这一点上
- 避免死锁主要有以下两个判断和处理时机：
  - 进程启动时判断：
    - 如果对资源的要求会导致死锁，就不启动有关进程
    - 假定的是最坏情况，即所有进程都同时需要最大数量的资源
  - 资源分配时判断：如果对资源的分配会导致死锁，就暂不允许进一步为进程分配资源
  - 银行家算法：
    - 分配资源时，申请者要把同类资源的最大需求量告诉系统，如系统现存的可用资源数能满足申请者剩余需求量时，就满足当前的部分或全部申请，否则就推迟分配
    - 这样至少保证有申请者能得到所需的全部资源，可执行到结束，然后释放资源供别的申请者使用
    - 如果系统保证申请者在有限的时间内能获得所需的全部资源，则称系统处于安全状态，否则称系统处于不安全状态，并有可能引起死锁。银行家算法是在能确保系统处于安全状态时才把资源分配给申请者
    - 例：有8个资源供三个进程共享，它们的最大需求数分别为6、4、7。在某一时刻，资源的分配情况如下所示

	最大需求	当前占有	还要申请
P0	6	2	4
P1	4	2	2
P2	7	1	6
系统剩余额			3

- 这时，系统处于安全状态，因为剩余的资源可以先供进程P1使用，P1运行结束后将释放所占全部资源，这样系统剩余资源数变为5，又可保证P0的全部申请得到满足。等到P0归还所占资源后，就可满足P2的申请。如此系统存在着一个安全的资源分配序列
- 但在上述的状态中，如果P0要申请2个（而不是1个）资源，系统就不能立即分配给它，而要推迟到一个适当的时机再实施分配过程，否则系统资源的分配情况将变为：

	最大需求	当前占有	还要申请
P0	6	4	2
P1	4	2	2
P2	7	1	6
系统剩余额			1

- 这种状态是不安全的，因为剩余的资源数已不能满足任何一个进程还要申请的资源数，如此就可能形成死锁

死锁的检测：

- 死锁的预防策略是非常保守的，它是靠限制对资源的存取及进程的并发执行程度来实施的

- 与其相反，死锁检测策略不减少对资源的存取或限制进程的并发运行。使用死锁检测，只要可能，就将所申请的资源分配给进程。操作系统定期地执行检查算法，以判断是否存在条件4的循环等待链
- 死锁的检测可在每当进程申请资源时进行，或可适当减少检测的频度，以减少检测的开销

死锁的解除：

- 强迫撤销所有的死锁进程
- 将每一个死锁进程退回到一些以前定义的“检查站”，再启动进程，这需要系统支持进程的回退和重启动机制
- 逐个撤销死锁进程，直至死锁不存在，终止死锁进程的次序应当基于最小代价的标准，每终止一个进程后就调用死锁检测算法，以判定死锁是否还存在
- 相继地剥夺进程所占的资源，直至死锁不再存在，同样，剥夺资源的次序应基于成本方面的考虑，被剥夺资源的进程必需回退到获得该资源之前的某个执行点上

## 5 设备管理

### 5.1 概述

除CPU、主存储器以外的设备的管理

I/O系统：设备及其接口线路、控制部件、管理软件

有关外设的驱动、控制、分配等技术问题都统一由设备管理程序负责

设备的分类：

- 功能：输入设备、输出设备、存储设备、供电设备、网络设备等
- 数据组织方式：块设备和字符设备
- 管理模式：物理设备和逻辑设备
- 资源属性：独占设备、共享设备和虚拟设备

I/O设备控制与驱动：

- I/O设备的控制和驱动技术包括了硬件控制驱动技术和驱动软件。前者是I/O设备厂商设计建立的与设备密切相关的技术。后者涉及系统所有I/O处理的软件。I/O驱动软件是操作系统的一部分
- 随着操作系统的发展，使I/O驱动软件成为一种带有标准接口的可选型的软件，操作系统内核中只保留与设备无关的那部分软件，而将与设备有关的驱动软件作为一种可装卸的程序，可以按照系统配置的需求进行配置
- 操作系统中的I/O驱动软件一般分为几个层次，如中断处理程序、设备驱动程序、操作系统I/O原语和用户级软件

设备管理的设计要求和任务：

- 为用户提供方便、统一的设备使用界面
- 提高外部设备利用率，尽量提高并行程度
- 实现程序与设备的无关性
- 系统与设备间的协调主要是速度上的协调，通常要解决快速的处理器与慢速的I/O设备之间的操作匹配的问题，在操作系统中采用缓冲区的方式来缓解这个矛盾，设备管理要实现这些缓冲区的建立、分配、释放与回收

### 5.2 操作系统与中断处理

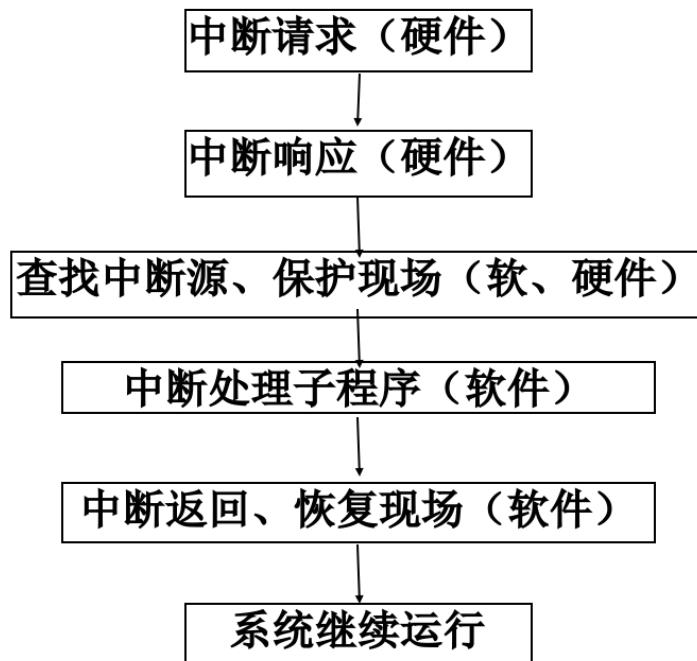
中断是OS与硬件的界面

中断分为外部中断与内部中断(多程序陷入)

中断与陷阱的区别：

- 外部中断，就是通常所说的中断(interrupt)。对于执行的系统来说，这种中断发生完全是“异步”的，根本无法预测到此类中断会在什么时候发生。因此，CPU(或者软件)对于此类外部中断完全是“被动”的。不过，软件可以通过关中断的形式来关闭对中断的响应，把它“反映情况”的途径掐断
- 由软件产生的中断则不同，它是由专设的指令，如Intel X86的“INT n”，在程序中有意地产生，所以是主动的，“同步”的。只要CPU一执行一条int指令，就知道在开始执行下一条指令之前一定要先进入中断服务程序，这种主动的中断称为“陷阱”
- 还有一种与中断相似的机制称之为“异常”(exception)，一般也是异步的，多半是由于“不小心”犯了规才发生的。例如，当在程序中发出一条除法指令div，而除数为零时就会发生一次异常。这多半是由于不小心，而不是故意的，所以这也是被动的
- 总结一下，中断和异常有个比较大的共同点就是“不可预知性”，所以是被迫的；而陷阱有“有意为之”的含义。实际上，cpu优先权不能屏蔽陷阱，所以说，实际上陷阱就是一种不可屏蔽中断

中断的处理过程：



中断的类型：I/O中断、时钟中断、系统请求中断、报警中断、程序错误中断、机器错误中断

中断机构处理外部设备的I/O中断

陷入机构处理指令的陷入(自陷)和由于软、硬件故障或错误造成的陷入

系统调用是UNIX操作系统面向用户的程序的界面。在汇编级上，系统调用使用trap指令

系统调用：

- 与进程管理和控制有关的系统调用：fork、exit、wait、signal、kill、semget、semop、semctl
- 与文件系统有关的系统调用：creat、open、close、read、write
- 远程进程通信：socket、connect
- 其它系统调用 times

中断的响应和实现过程：

- 每当执行完一条指令，检测有无中断请求
- 不同的中断类型规定了不同的优先级，中断嵌套
- 根据中断源找出相应的中断处理程序入口地址，以便转去执行，中断矢量存放中断处理程序的地址
- 保护现场，指令断点，运行参数和条件，现运行程序状态寄存器PS的内容以及累加器或通用寄存器的内容和标记
- 中断屏蔽

中断处理程序和驱动程序：

- UNIX把设备作为一种虚拟的文件对待，每个设备有一个像文件名那样的名字，可以对它像一个文件那样存取
- 在UNIX系统中，将设备分成两类：块设备和字符设备。核心与驱动程序的接口是由块设备开关表和字符设备开关表描述的

中断的返回与恢复：

- UNIX的中断处理都是在核心态下进行的
- 如果中断前处理器状态为核心态，则在执行完设备处理子程序后就恢复现场，然后用中断返回指令回到中断前状态，继续执行被中断的操作系统程序
- 如果中断前为用户态，则在执行完设备处理子程序后，先要检查标志runrun是否设置

## 5.3 操作系统与时钟系统

在计算机系统中可分为三类时钟：系统时钟、日历时钟、实时时钟

- 系统时钟主要用于控制系统处理器执行指令的速率
- 日历时钟产生一个精确的时间计数，程序对此进行转换，给出与日历相符的日期和时间
- 实时时钟每秒提供若干个时钟中断，提醒处理器有重要的事情要做

UNIX时钟管理的主要任务是调整动态优先数和换入换出时间条件，并为一些外设管理提供服务

每隔20ms处理的工作如下：

- 重新设置时钟初值20ms
- 计算当前进程在用户态或核心态下的累计运行时间，将u.u\_utime++或u.u\_stime++
- 当前运行进程p\_cpu加1
- 处理延时启动终端打印机的工作，clock若发现延迟时间到了，则重新启动对应的输出驱动程序

每秒一次处理的工作如下：

- 日历时钟变量time加1
- 所有进程的内存或对换区的驻留时间p\_time++
- 所有目前未运行进程的p\_cpu除以2

## 5.4 操作系统对I/O操作的控制

I/O设备的资源分配：

- I/O地址：进行正确的I/O地址设置
- I/O中断请求：争用剩余的中断请求号，也会产生冲突
- DMA数据传输通道：对争用同一个DMA通道的I/O设备需要进行协调和重新配置

- I/O缓冲区：这个系统资源也是I/O设备争用的

I/O通道技术：

- I/O通道是一种硬件设施，带有专用处理器的、有很强I/O处理功能的智能部件
- 可以独立地完成系统处理器交付的I/O操作任务，通道具有自己专门的指令集，即通道指令
- 通道执行来自处理器的通道程序，完成后只需向系统处理器发出中断，请求结束
- 字节多路通道：主要用于连接大量低、中速、以字节作为传输单位的I/O设备
- 选择通道：主要用以支持高速设备（如磁盘），每次只对一个设备进行数据传输
- 成组多路通道：以分时方式同时执行几道通道程序，每条通道指令可以传送一组数据

I/O缓冲技术：

- 提高中央处理器与外设的并行程度
- 可以采用硬件缓冲和软件缓冲两种方式
- 软件缓冲是借助操作系统的管理，采用内存中的一个或者多个区域作为缓冲区
- 缓冲区的数量可根据不同的系统和操作来确定，常用的缓冲技术有三种：双缓冲、环形缓冲和缓冲池

设备的驱动：

- 为了将设备的硬件复杂性与用户隔离，也为了建立一种通用的I/O接口，OS采用设备驱动程序来完成设备的驱动

## 5.5 设备管理的数据结构

设备控制表（DCT）：

- 提供若干高级I/O系统调用，用这些抽象的I/O操作把用户与复杂的I/O设备操作隔离，隐藏设备操作的细节，有利于编写与设备无关的程序
- 要完成抽象到实际的映射，通常采用称为设备控制表(DCT)的数据结构来完成。它记录每一个抽象设备描述、对应的实际设备地址、所使用的设备驱动程序等参数

设备开关表：

- 针对各类设备不同的物理特性，系统为它们各自设置了一套子程序，它们包括打开、关闭和启动子程序
- 系统为每类设备又设置了一数据结构，存放这些程序的入口地址，该数据结构称为设备开关

## 5.6 磁盘调度

磁盘系统硬件：磁盘驱动器、磁盘控制器

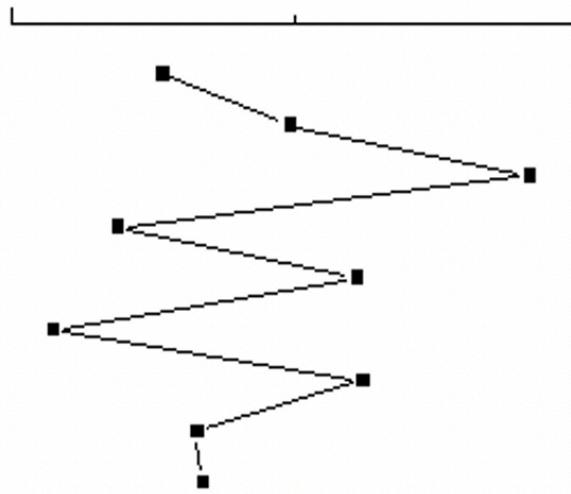
磁盘服务的总时间：

- 寻道延迟（最大延迟）：移动读写磁头到对应磁道的时间
- 旋转延迟（等待时间）：等待扇区旋转到磁头下面的时间
- 数据传输延迟（最小延迟）：从（向）扇区上读（写）二进制数据

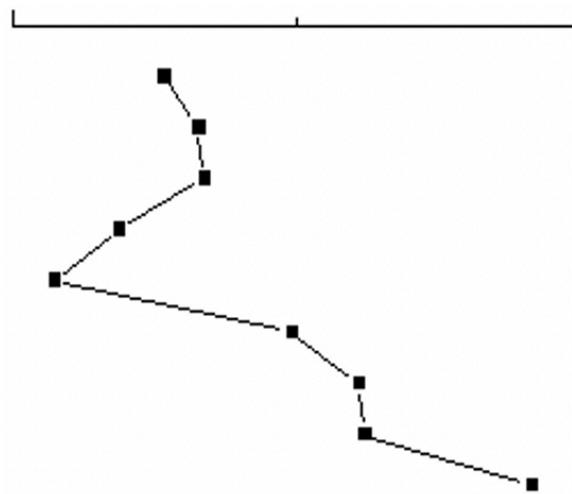
对磁盘的请求一般有以下内容：输入或输出、盘地址（驱动器、柱面、面号、扇区）、内存地址、传送长度

磁盘调度算法：

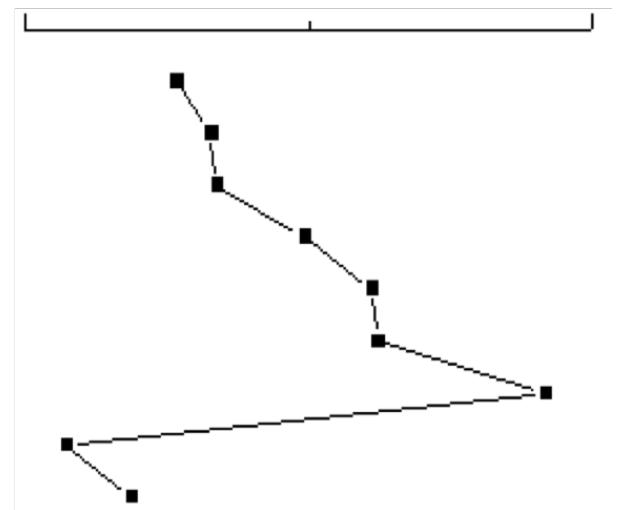
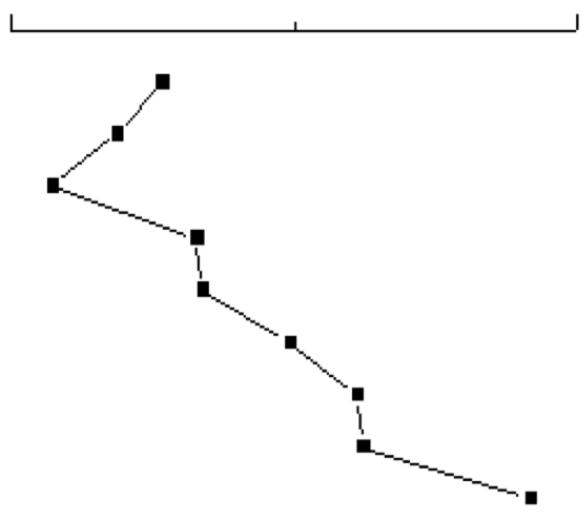
- 先来先服务调度（FCFS）



- 最短寻道时间优先法 (SSTF)



- 扫描法 (SCAN和C-SCAN)



## 5.7 UNIX系统V的设备管理

UNIX系统中包含2类设备：块设备、字符设备

用户通过文件系统与设备接口，对设备的使用类似于对文件的存取

文件系统与设备驱动程序之间的接口是设备开关表：块设备开关表、字符设备开关表

UNIX系统采用多重缓冲技术来平滑和加快文件信息在内存与磁盘之间的传输，缓冲管理模块处在文件系统和块设备驱动模块之间

缓冲控制块buf:

- 在系统初启时，核心根据内存大小和系统性能要求分配若干缓冲区
- 一个缓冲区由两部分组成：存放数据的内存区（一般称为缓冲区）和一个缓冲控制块
- 缓冲区和缓冲控制块是一一对应的，系统通过缓冲控制块实现对缓冲区的管理
- b\_flags反映缓冲区的使用情况和I/O方式，如忙或闲、数据有效性、“延迟写”、正在读/写、等待缓冲区空闲等
- 从buf的组成可见，它不仅包含了与使用缓冲区有关的信息，也记录了I/O请求及其执行结果。所以一般而言，buf既是缓存控制块，同时又可以是针对该缓存进行的I/O请求块
- 为管理方便，系统还设置了自由缓存队列控制块bfreelist和进程图像传送控制块swbuf。这两个块结构与buf结构相同，但只用部分项，其余则弃之不用

块设备表：

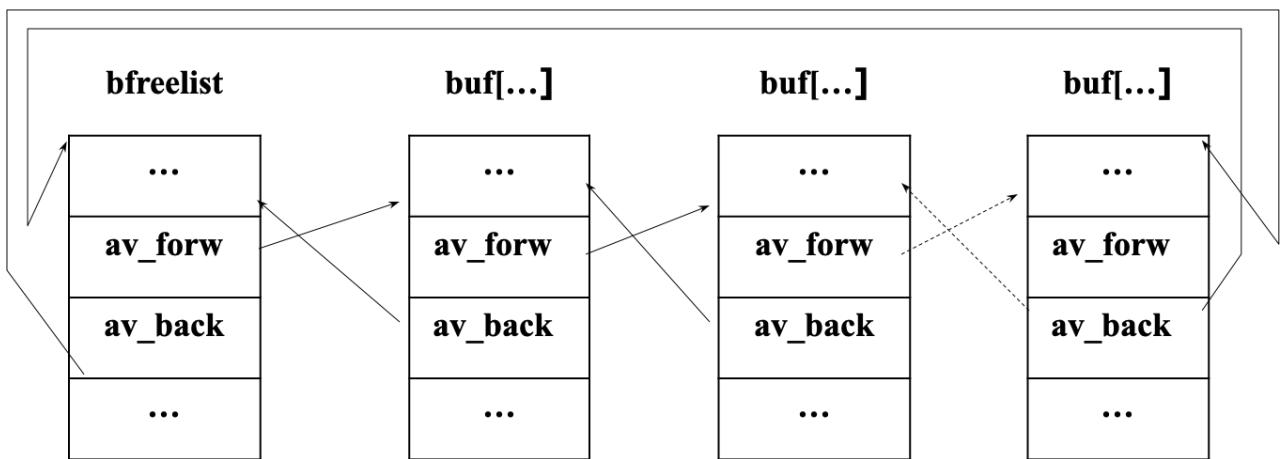
- 为了管理块设备，UNIX为每个块设备控制器设置了块设备表

块设备开关表：

- UNIX为各类块设备设置块设备开关表，存放设备管理程序的入口地址

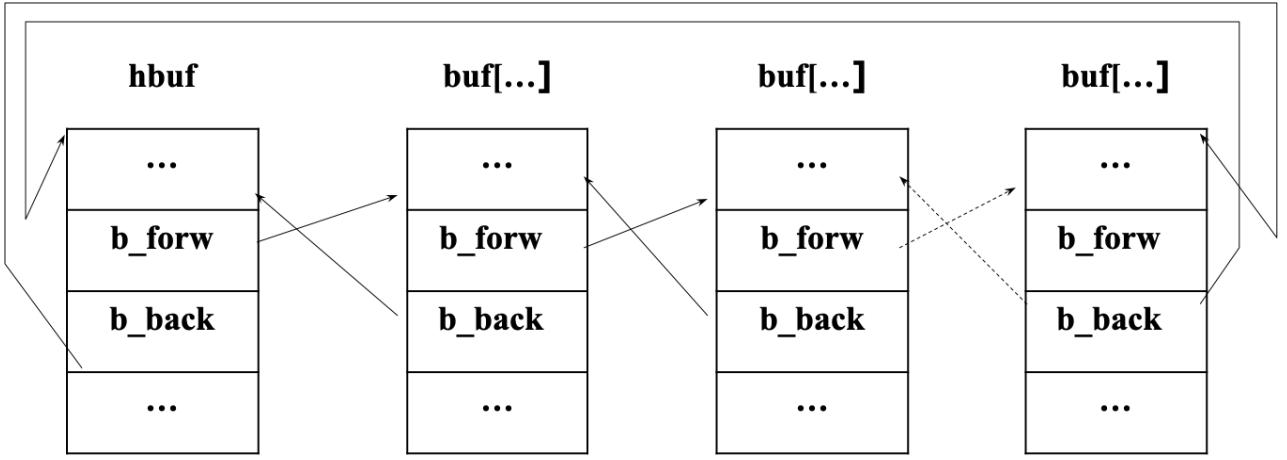
多种缓冲区管理队列：

- 系统设置了多种队列对所有缓冲区进行管理，缓冲区管理队列实际上是缓存控制块buf队列
- 自由buf队列
  - 系统把空闲缓冲区的buf组成空闲buf队列，即自由buf队列
  - 这个队列是双向链结构，队首块为bfreelist，bfreelist和自由buf通过av\_forw和av\_back作为双向指针
  - 采用FIFO管理，回收缓冲区时相应buf插到队尾，分配时从队首取



- 设备缓冲区队列

- 连接所有各类设备使用过的缓冲区，这也是一个双向队列，buf中的b\_forw和b\_back分别为该队列的前指针和后指针，头部为hbuf，共64个队列
  - 一个缓冲区被分配用于读、写某个设备的字符块时，其相应的buf就进入该设备的设备buf队列，并一直保留在该设备buf队列中，除非被移作它用
  - 系统V与第六版有所不同，不是每个块设备一个缓冲队列，同一个块设备的缓冲区可以分布在不同的散列队列，不同设备的缓冲分布均匀，加快缓冲区的搜索速度



- 空设备队列 (NODEV队列)

- NODEV队列是一个特殊的设备buf队列。当系统需要缓存，但它不与特定的设备字符块相关联时，将分配到的缓存控制块buf送入NODEV队列。其队列控制块也是bfreelist，用的指针是b\_forw和b\_back
- 在UNIX中有两种情况将buf送入NODEV队列
  - 在进程执行一个目标程序的开始阶段，它用缓存存放传向该目标程序的参数
  - 用缓存存放文件系统的资源管理块
- 在系统初启时，所有空闲缓冲区的buf既在自由buf队列，又在NODEV的设备buf队列中

- 设备I/O请求队列

- 每个块设备有一个设备I/O请求队列，单向连接，头部为iobuf, b\_actf和b\_actl分别指向队首和队尾

- 四个队列所属缓冲区之间的关系总结：

- 任何一个缓冲区在稳定的状态下，同时属于二个队列
- 在自由buf队列中缓冲区可能同时属于NODEV队列（未分配给指定设备），也可能属于设备buf队列（已经分配给某设备用过，因I/O结束而释放）
- 在设备buf队列中的缓冲区，可能挂在自由buf队列（已用过被释放），也可能挂在设备I/O请求队列（正在使用）
- 在NODEV队列中的缓冲区如已用过被释放，肯定在自由buf队列中
- 在设备I/O请求队列中的缓冲区则肯定属于设备buf队列

缓冲区管理算法：

- 最久未使用算法 LRU
- 一个缓存刚分配用于读写某一块设备，buf的b\_flags含有B\_BUSY标志。它一定位于相应设备buf队列，不在自由队列
- 一旦读写完成，就释放缓存，清B\_BUSY标志。送入自由队列尾，但仍留在原设备队列
- 特点和好处：
  - 一个缓存既在设备队列，又在自由队列，只要还要重复使用原设备队列中该缓存内容，就只要简单地将其从自由队列抽出即可，避免了重复I/O
  - 有必要可将缓存重新分配它用，将它从自由队列首和原设备队列同时抽出，送入新的设备队列。操作完成后仍留在新的设备队列并送入自由队列
- 为使一个已释放地缓存尽可能长地保持原先内容，以增加重复使用可能，将其送入自由队列尾，分配作它用缓存从自由队列首取

- 当一个buf在自由队列中移动时，只要原设备队列又重新使用它，立即将其从自由队列中间抽取，使用完毕，再次送入自由队列尾（LRU算法）
- 对于写，如一块未写满，在b\_flags中设置B\_DELWRI标志，推迟写，再清B\_BUSY，释放至自由队列尾
- 当设置B\_DELWRI标志的缓存排到自由队列首时，不能立即淘汰，它作重新分配处理，而是提出I/O，将其内容复制到块设备后，再次释放到自由队列尾（也有释放到自由队列首），也清B\_DELWRI标志

缓冲区的分配和释放：

- 当进程想从特定盘块上读取数据或打算把数据写到特定盘块上时，核心要查看该块是否已在缓冲池中
- 如果未在，则为该块分配一个空闲的缓冲区
- 搜索和分配缓冲区的工作由getblk程序完成
- 当核心用完缓冲区后，要将其释放，加入到自由队列中，使用函数brelse

UNIX系统对块设备的主要使用方式：

- 文件系统使用，块设备管理和文件系统之间的界面，使用缓存技术
- 进程映像在内存和盘交换区之间进行传送时使用，不通过缓存

读盘块：

- 字符块输入是指从盘上读字符块，两种方式
  - 基本读入方式
    - 读字符块的基本工作过程
    - 从块设备上用同步方式将一个指定的字符块读入缓存，使用函数bread()
    - 输出一般异步
  - 预读操作
    - 提高CPU和块设备工作并行程度所采用的技术，使用程序breada
    - 当一个进程顺序读取文件时，为加快它的前进速度，提高CPU和块设备工作的并行程度
    - 核心检查第一块是否在缓冲区中，如不在，则调用磁盘驱动程序读该块。如第二块不在缓冲区中，核心指示磁盘驱动程序异步读它。然后进程睡眠，等待第一块I/O完成
    - 该进程被唤醒后就返回第一块的buf，而不管第二块是否读完。以后，当第二块读完后，产生盘I/O中断，由中断处理程序识别异步读完成，并释放相应的buf

## 5.8 设备分配

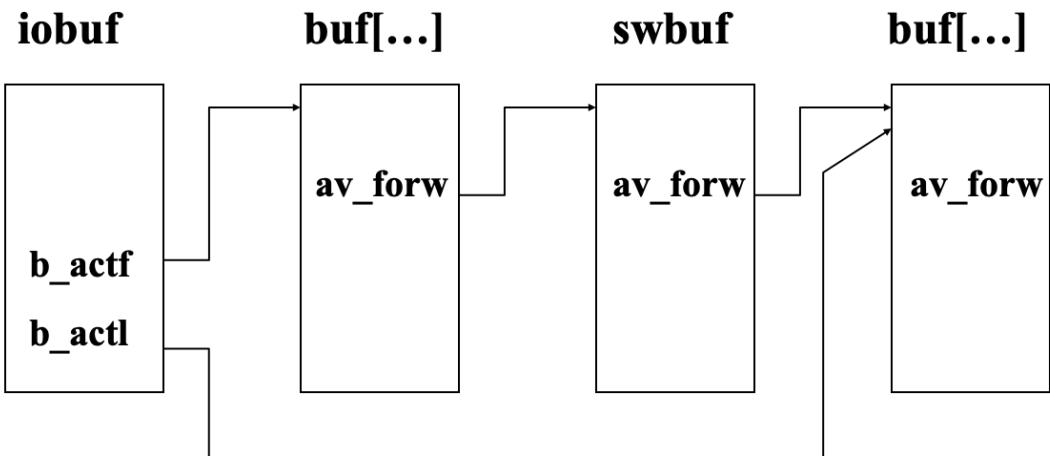
---

设备分配方式：

- 静态分配：作业运行前系统分配全部，利用率低，无死锁
- 动态分配：进程运行中分配，利用率高，可能死锁

设备分配的原则：

- 先请求先服务和按请求I/O的进程的优先级决定
- 要使用设备时必须提供进行I/O操作的有关信息，指出执行I/O的逻辑设备名（如设备号）、操作类型、传送数据的数目、信息源或目的地址等
- 存放进行I/O操作的信息的结构称为I/O请求块。如在UNIX系统中，系统的I/O请求块的内容是包含在缓冲区控制块buf中
- 在请求I/O时，首先请求分配缓冲区，然后把与操作有关的信息写到buf中，并把这个buf挂到请求设备的I/O请求队列中



类别	层次	说明	磁 盘 请 求	数据有效性	大 IO 数据量传输能 力	小 IO 请求率
条带化	0	非冗余	N	低于单个磁盘	很高	读和写都很高
镜像	1	被镜像	2N,3 N 等	高 于 RAID2,3,4,5; 低 于 RAID6	读时高于单个磁盘; 写时与单个盘相近	读时最快可以为单个 磁盘的两倍; 写时与 单个磁盘相近
并行 访问	2	通过汉明 码实现冗 余	N+m	明显高于单个磁 盘 ; 高 于 RAID3,4,5	所有方案中最高	大概是单个磁盘的两 倍
	3	交错位奇 偶校验	N+1	明显高于单个磁 盘 ; 相 当 于 RAID2,4,5	所有方案中最高	大概是单个磁盘的两 倍
独立 访问	4	交错块奇 偶校验	N+1	明显高于单个磁 盘 ; 相 当 于 RAID2,3,5	读时与 RAID0 相近; 写时明显慢于单个磁 盘	读时与 RAID0 相近; 写时显著慢于单个磁 盘
	5	交错块分 布奇偶校 验	N+1	明显高于单个磁 盘 ; 相 当 于 RAID2,3,4	读时与 RAID0 相近; 写时慢于单个磁盘	读时与 RAID0 相近; 写时通常慢于单个磁 盘
	6	交错块双 重分布奇 偶校验	N+2	所有方案中最高 的	读时与 RAID0 相近; 写时慢于 RAID5	读时与 RAID0 相近; 写时显著慢于 RAID5

## 6 文件系统

### 6.1 概述

字段：数据的基本单位，又可称为域或数据项。不可分隔的字段含有一个简单的值，如姓名、日期等。字段的特征可由长度和数据类型表示。字段可以是固定长度的或可变长度的

记录：能被某些应用程序处理的相关字段的集合。例如，雇员记录可包括姓名、社会保险号、工种、雇用日期等。记录可以是固定长度或可变长度的

文件：相同记录的集合，可以用名字来引用，并可以产生和删除。存取控制通常施加到文件这一级。在某些复杂的系统中，存取控制能施加到记录级，甚至字段级

数据库：相关数据的集合。数据库的基本特征是在数据元素之间存在明显的关系

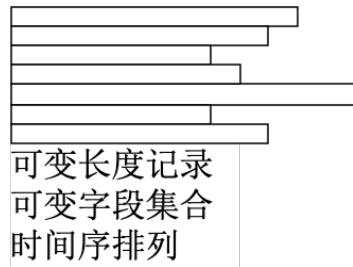
文件管理系统：为用户和用户程序在使用文件时提供服务的系统软件集合。一般用户或应用程序存取文件时必须通过文件管理程序，文件管理系统避免了用户或程序设计员必须为每种应用都开发特定目的软件

在选择文件组织方法时，有几个重要的参考标准：存取快速、更新容易、节省存储单元、管理简单、可靠性

文件组织方法：

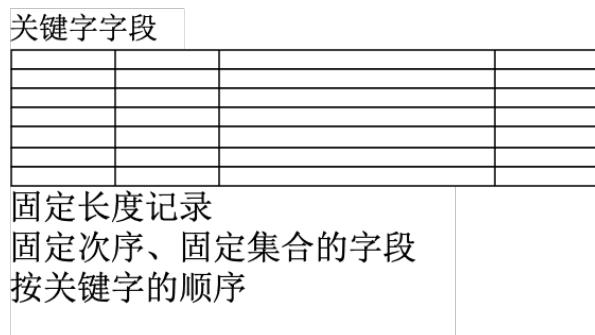
- 堆文件

- 数据根据到达时间的顺序收集起来，每一个记录包括一堆集中到达的数据
- 堆的目的只是简单地聚集大量的数据，并存储它
- 当收集数据时需先存储再处理或数据不容易组织时，就可用到堆
- 穷尽搜索文件



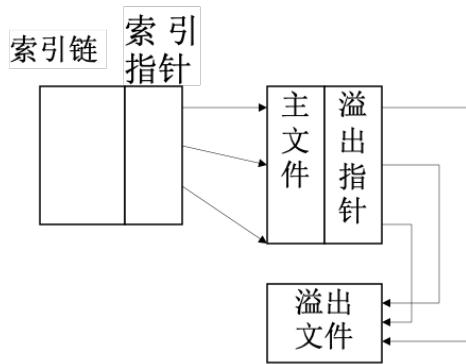
- 顺序文件

- 使用固定格式的记录，所有记录的长度相等，含有相同个数、特定次序的固定长度字段
- 由于每一个字段的长度和位置是已知的，所以仅仅需要存储字段的值
- 每个字段的字段名和长度归于文件结构的属性
- 顺序文件的另一种组织方法是采用链接表



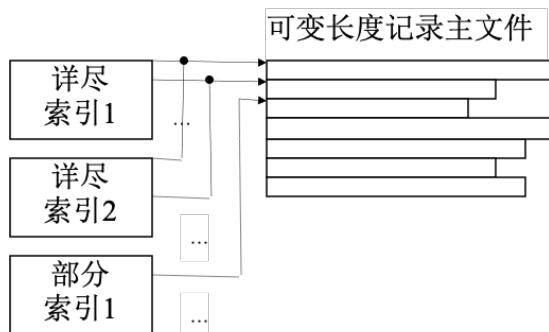
- 索引顺序文件

- 维护了顺序文件的关键字特征：记录以关键字字段的顺序组织
- 增加了两个新的特征：支持随机存取的文件索引和溢出文件
- 在索引文件中的每个记录包括两个字段：
  - 与主文件关键字相同的关键字字段
  - 指向主文件的指针



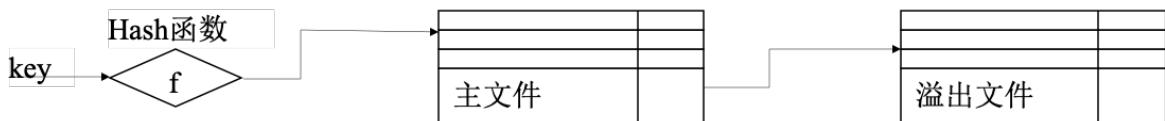
- 索引文件

- 需要使用多索引的结构
- 详尽的索引为主文件中的每一个记录建立一个入口项
- 部分索引对感兴趣字段的记录建立入口项
- 索引本身组织成一个顺序文件以便容易搜索



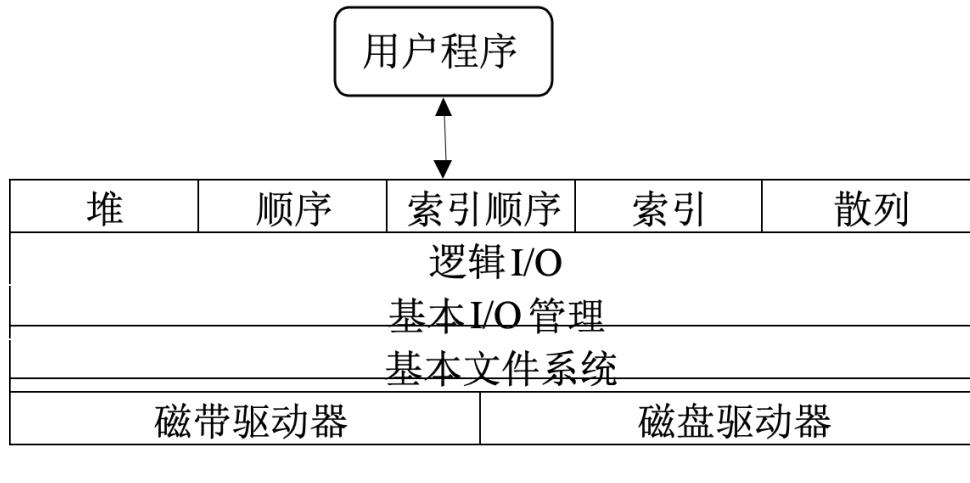
- 散列文件

- 直接存取的散列文件使用对关键字值的散列。散列文件经常用于需要快速存取的场合
- 文件包括固定长度的记录，应用程序一般一次存取一个记录
- 目录、调度表和名字表等都可使用散列文件



文件系统结构：

- 设备驱动程序负责对设备的启动I/O操作，处理I/O请求完成后的工作
- 上一层是基本文件系统，或称物理I/O层，这一层是与计算机系统外部环境的主要接口
- 基本I/O管理主要负责所有文件I/O的初始化和结束工作。在这一层，维护处理设备I/O、调度和文件的控制结构。I/O缓冲区的分配及外存的定位工作也在这一层完成
- 逻辑I/O使用户和应用程序能够存取记录
- 最接近用户的文件系统层是存取方法层，其提供了用户与文件系统和存储数据的设备之间的标准接口



## 6.2 文件目录

目录的内容：

- 目录含有文件的信息，包括文件的属性、位置和属主等
- 目录本身也是文件，各种文件管理例行程序要通过操作系统存取目录文件
- 用户不能直接存取目录，即使是以只读的方式
- 目录提供了为用户和应用程序所知的文件名和文件实体本身的映射
- 基本信息：文件名、文件类型、文件组织
- 地址信息：起始地址、已使用大小
- 存取控制信息：文件主、允许的操作
- 使用信息：创建日期、读时间、修改时间

目录的结构：

- 对目录操作的类型：
  - 搜索：搜索目录以找到对应于文件的目录项
  - 创建文件：将目录项加到目录中
  - 删除文件：从目录中移去一个目录项
  - 列目录：可以要求列出全部或部分目录内容
- 使用两级目录：在这种情况下，系统有一个主目录，并为每一个用户设置一个目录
- 层次结构或称树型结构：系统中有一个主目录，在它之下有一些用户目录，每一个用户目录也同样允许有子目录和文件

## 6.3 文件存储资源分配

预分配和动态分配：

- 预分配的策略需要在创建文件时说明文件的最大长度，困难而且浪费
- 动态分配在需要时才给文件分配空间

分区大小：

- 连续的空间提高了操作性能
- 具有大量的小分区增加了管理存储分配信息表的大小
- 具有固定大小的分区（如，以块为单位）可简化存储空间的算法

- 具有可变大小或固定的小尺寸的分区可以减少存储空间的浪费

两个主要的选择方案：

- 可变长度、连续的大分区：性能较好，可变长度避免浪费存储空间，文件分配表也较小
- 块：小而固定长度的分区，灵活性大；但存储分配需要较大的表或复杂的结构；存储块即需即分配，但块间就不一定相

可变长度分区需要考虑空闲区的碎片问题：

- 首次适应法：选择第一个足够大小的连续空闲块组
- 最佳适应法：在所有满足大小要求的空闲块组中选择最小的一组
- 循环首次适应法：选择离该文件的前次分配物理位置最近的满足大小要求的空闲块组

空闲存储空间管理：

- 位表：每一二进制位对应一个磁盘块。二进制位为0表示一个空闲块，位值为1表示该块已被使用。位表所具有的优点是查找一个或一组连续的空闲块比较方便。位表已是尽可能地小，因此可以全部放入主存中
- 空闲分区链：空闲分区可以用指针连在一起，还需要一个空闲分区的长度信息
- 索引：索引方法将空闲存储区当作文件一样处理，像文件分配那样使用索引表。在索引表上的每一项对应磁盘上的每一个空闲分区

## 6.4 文件的系统调用

文件的创建：

```
fd=creat(pathname,mode);
int fd, mode;
char *pathname;
 pathname是要创建文件的带路径的文件名
 mode是以二进制的位值为该文件设置的存取控制权限
 fd中存放文件创建成功后系统返回的整数值
```

文件的打开：

```
fd=open (pathname,flags);
int fd, flags;
char *pathname;
 flags表示本次打开后要对文件进行的操作
```

文件的关闭：

```
close (fd) ;
```

文件的联接、解除联接和删除：

为名为name1的文件再起一个新的名，可使用系统调用：

```
link ("name1", "name2");
```

要取消某文件的一个文件名，则可用系统调用：

```
unlink (pathname);
```

利用link和unlink调用可以改变文件的名：

```
link ("name1", "name2");
```

```
unlink ("name1");
```

相当于UNIX命令：mv name1 name2

unlink调用仅仅使其联接数减去1。当文件仅剩下一个名时，unlink调用取消这一文件名后再将该文件从系统中删

文件的读写：

```
n=read (fd, buf, nbytes);  
n=write (fd, buf, nbytes);  
int fd, n;  
unsigned nbytes;  
char *buf;
```

fd为先前已打开文件的描述字

nbyte是欲读、写的字节数

对于读，buf是读出的文件信息应送至的目标区始址；对于写，buf是要写入文件的信息源区首址

n是调用返回值，表示实际读出和或实际写入的字节数。n的值可能不等于nbytes的值。如不相等，对于读，0≤n<nbytes时，表示已读到了文件尾，n=-1时，表示读操作出错。对于写，n≠nbyte或n=-1时都表示写出错

3个特殊文件：

- 一般在进程创建后就已处于打开的状态，它们是：
  - 0号——标准读文件代表键盘输入
  - 1号——标准写文件代表屏幕显示
  - 2号——标准错误输出文件，是程序的运行时的出错信息，也写至屏幕上

调整文件读写位置lseek：

- read和write系统调用提供了文件顺序读、写功能，但如要从文件中的指定位置起读或写，就要使用改变文件读写指针的系统调用lseek

```
lseek (fd, offset, whence);  
int fd, whence;  
long offset;
```

offset和whence配合起来决定调整到文件中的一个新位置。offset为字节偏移值，whence为参考点

创建任何类型文件mknod：

- 目录、有名管道和特别文件只能用mknod创建
- 普通用户可以用mknod创建一个有名管道，但只有超级用户可以用mknod创建文件、目录和特别文件

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod (pathname, mode, device);
char * pathname; /*文件名*/
int mode, device; /*文件的属性和权限模式, 设备号*/
```

其它的文件系统调用：

- 改变文件的权限chmod
- 改变当前目录chdir
- 复制文件描述字dup
- 查询文件的状态stat与fstat：前一个调用所指文件是以UNIX的路径名形式给出，因此能返回文件系统中存在的文件状态，后一个调用是以文件描述字的形式给出，故只能用于已打开的文件。后一种调用执行的速度要比前一种快
- 文件控制fcntl
- 设置文件权限屏蔽字：umask调用可以设置或改变进程所创建文件的屏蔽字

## 6.5 文件的标准子例程

标准I/O的概念：

- 文件基本操作的系统调用提供的只是字节序列方式的最基本的功能，不能完成任何格式转化。为了提供功能更强和使用更方便的输入和输出操作，UNIX系统提供了标准I/O库
- 标准I/O库主要通过用户态空间的自动缓冲机构以及数据类型转化和格式化的输入输出，提供了效率高、功能强和可移植的文件访问或字符串处理功能
- 在UNIX的标准I/O库中，通过一个FILE类型结构建立与打开文件的联系，称为流（stream）

流文件的打开：

```
include <stdio.h>
FILE *fp;
fp=fopen ( pathname, type);
```

char \* pathname, \*type;  
如果打开成功，把它与一个流联系起来，并返回标识该流的FILE结构的指针。如fopen失败，返回NULL。参数type可取下列的基本值：

r 打开文件用于只读  
w 建立文件或把已存在文件截为空文件，用于只写  
a 打开文件用于文件尾的追加写

在上述参数后各加上一个“+”，表示修改模式，即可读可写

与0#，1#，2#三个标准的打开文件描述字相联系，标准I/O库提供了三个不需要打开的流，并用下列的FILE指针标识：

stdin	标准输入流
stdout	标准输出流
stderr	错误输出流

流文件的关闭：

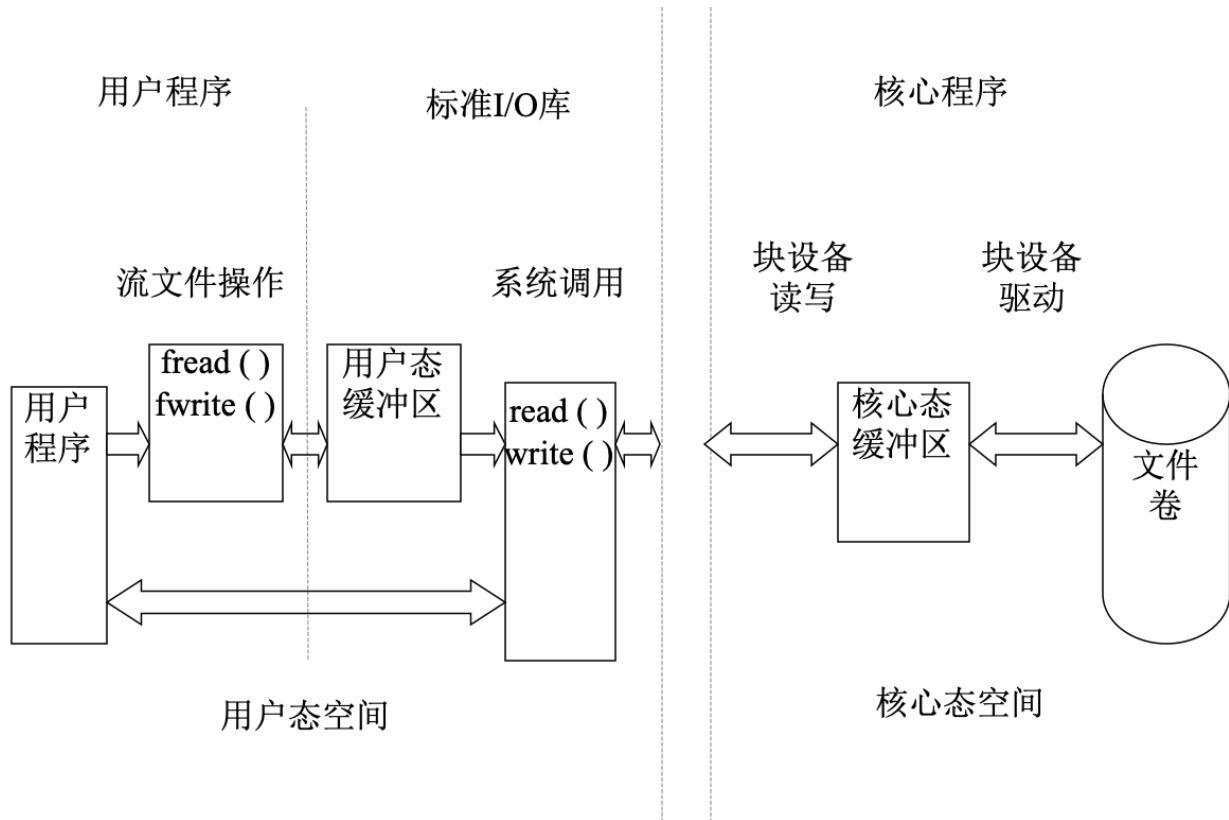
```
fclose (fp) ;
```

该命令刷新与文件相关的缓冲区，关闭已打开的文件并释放FILE结构

流文件的读写：

```
n=fread (buf, size, nitems, fp) ;
n=fwrite (buf, size, nitems, fp) ;
int n, size, nitems;
char *buf;
FILE *fp;
```

- 流文件操作用1024次用户态函数的调用和返回的开销代替了直接使用系统调用时1024次用户态与核心态之间的切换
- 当不是以整块的方法输入或输出数据时，使用流文件操作比使用系统调用效率高很多



调整和获取流文件的读写位置：

```
fseek (fp, offset, whence);
rewind (fp);
position = ftell (fp);
FILE *fp;
long offset, position;
int whence;
```

格式输入与输出：

- 除前面介绍的不经格式转化的二进制数据输入输出外，标准I/O库还提供了能进行格式转化的I/O例行程序

- 用户可以直接观察存储在文件中或显示在屏幕上的数据，而且这类操作是与机器特征无关的。如果用户要从键盘上输入数据，那么带格式转化的输入一般是唯一的方法

```
fscanf (fp, format [, arg1, arg2, ... argn] );
fprintf (fp, format [, arg1, arg2, ... argn] );
FILE *fp;
char *format;
```

如果要从标准输入中读入，或将数据写至标准输出上，则可写成：

```
fscanf (stdin, format [, arg1, arg2, ... argn] );
fprintf (stdout, format [, arg1, arg2, ... argn] );
```

流的单字符I/O操作：

- 如果每次只要从流文件中读入一个字符或向流文件中输出一个字符，可使用下面两个例行程序

```
c = fgetc (fp)      和
fputc (c, fp)
char c;
FILE *fp;
```

- 要从标准输入中读进一个字符，或把一个字符写至标准输出上，则可使用下面两个例行程序

- 由于使用了用户态空间的缓冲，故执行这些单字符I/O操作大大减少了对应的系统调用的次数，也就是减少了用户态与核心态切换的开销

```
c = fgetc (stdin);      和
fputc (c, stdout);
也可写成如下的形式：
c = getc (stdin);
putc (c, stdout);
另外两种更加方便的书写形式是：
c = getchar();
putchar(c);
```

- 返回一个字符

- ungetc可以将一个字符放回一个流中
- 在执行过任何流的输入操作后，只允许对每个文件退回刚读出的一个字符，但一般只用于在单字符输入操作之后

```
ungetc(c, fp);
char c;
FILE *fp;
```

行的输入与输出

- 从流文件中读入一行或向流文件中写入一行：

```
retstring = fgets(buf, size, fp);
fputs(string, fp);
char *buf, *retstring, *string;
int size;
FILE *fp;
```

- 从标准输入上读入一行或向标准输出上写一行信息：

```
retstring = gets (buf) ;
puts (string);
char *buf, *retstring, *string;
```

程序的执行：

- 类似于系统调用exec，例行程序库向程序员提供了在一个程序中执行另一个独立程序的函数调用，其中最基本的是system，其格式为：

```
retval = system (command);
char *command;
int retval;
```

## 6.6 UNIX文件系统的内部结构

索引节点：

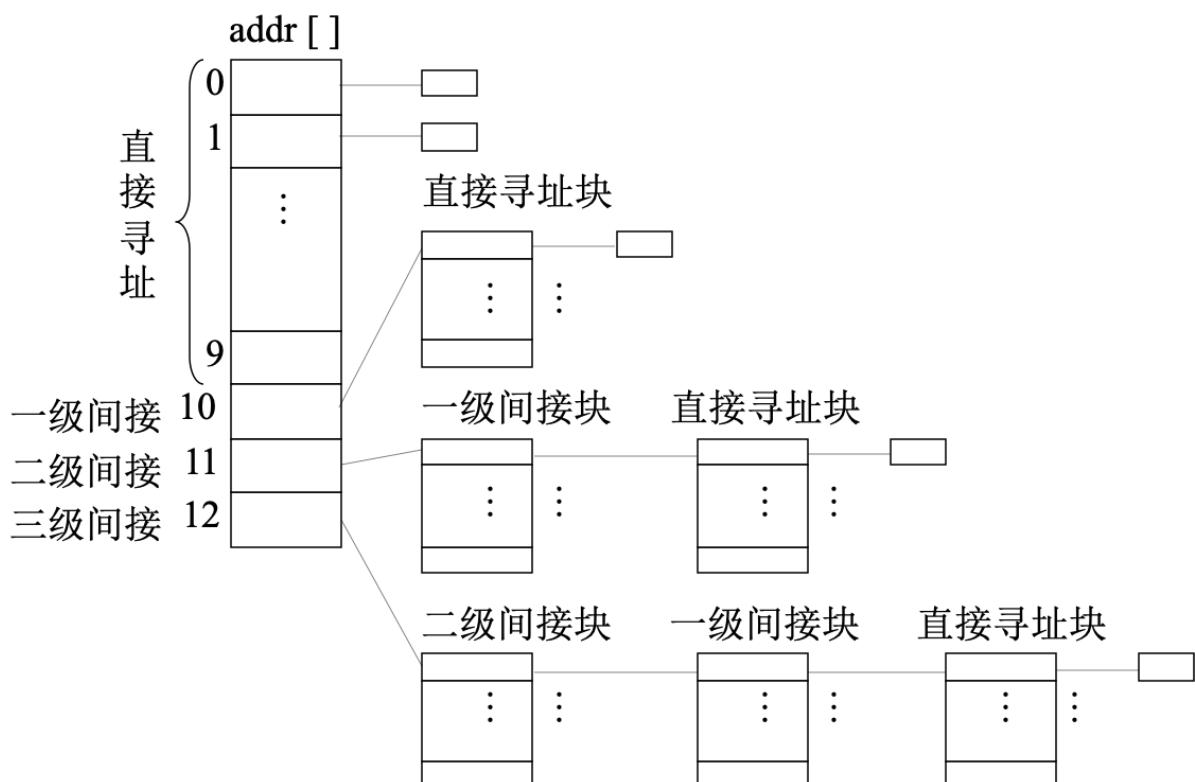
- 文件所有的控制信息构成了文件的索引节点，所有的I节点是集中存放在磁盘上的I节点区，故又称为磁盘I节点

```
struct dinode {
    ushort      di_mode;          文件控制模式
    short       di_nlink;         文件的链接数
    ushort      di_uid;           文件主用户标识数
    ushort      di_gid;           文件主同组用户组标识数
    off_t       di_size;          文件长度，以字节为单位
    char        di_addr[40];       地址索引表，存放文件的盘块号
    time_t      di_atime;         文件最近一次访问时间
    time_t      di_mtime;         文件最近一次修改时间
    time_t      di_ctime;         文件创建时间
}
```

- di\_mode的各个二进制位或它们的组合分别定义了文件的类型、执行时置文件主或组用户标识符、常驻盘交换区和文件的存取控制数
- 字符数组di\_addr[40]中每三个字节组成一个单元，以记录文件的盘块号，构成了13个表项的地址索引表，将其设置成40字节大小是为了使索引节点的大小为64字节，以便在一个盘块中正好放满整数个I节点

文件索引结构：

- di\_addr共有13个表项，记录了文件中所有盘块地址
- 假定这些表项都直接存放文件的一个盘块号，每个盘块大小为1KB，那么文件的最大长度是13KB，显然不够，因此UNIX将这13个表项分成4种寻址方式
- 直接寻址：
  - UNIX系统中文件大小在8KB以下的占85%，为了提高对绝大多数文件的访问速度，宜采用直接寻址方式
- 一级间接寻址：
  - 对于长度大于10个盘块的文件，其前10个盘块还是采用直接寻址方式，后面的盘块部分采用一级间接寻址方式
  - 即在地址索引表的第11个表项登记的不是文件的物理盘块号，而是一个索引块的地址，该索引块也是按3字节组成一个表项
  - 每个表项存放文件第10个逻辑块以后的物理盘块的地址采用这种索引方式的文件大小为  $[1024/3]=341$ KB，与直寻址方式相加，文件最大长度为351KB
- 二级间接寻址：
  - 对于长度超过前两种寻址方式所能寻址的文件，超过部分则采用二级间接寻址方式
  - 在地址索引表的第12个表项中登记了一个具有341个表项的间接索引块地址，间接索引块的每一个表项又各登记了一个具有341个表项的索引块地址，在这级索引块中的表项才有存放文件的物理盘块地址
  - 单用二级间接索引就可访问 $341 \times 341$ 个文件物理块
- 三级间接寻址：
  - 对于长度超过前三种寻址方式所能寻址的文件，超过部分采用三级间接索引，其寻址原理与上类似。单用三级间接寻址方式所能访问的物理盘块数为 $341 \times 341 \times 341$ 块
- 以上四种寻址方式所能访问的文件最大长度为  $(10+341+341 \times 341+341 \times 341 \times 341)$  KB，即近40G
- 文件的长度还要受I节点中记录文件长度的成员di\_size的取值限制，di\_size为32位的无符号长整型，这样文件的最大长度为 $2^{32}$  byte = 4G

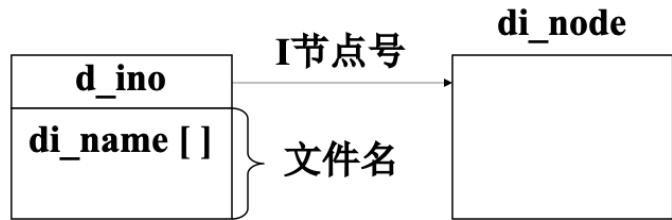


目录结构：

- 目录项结构：

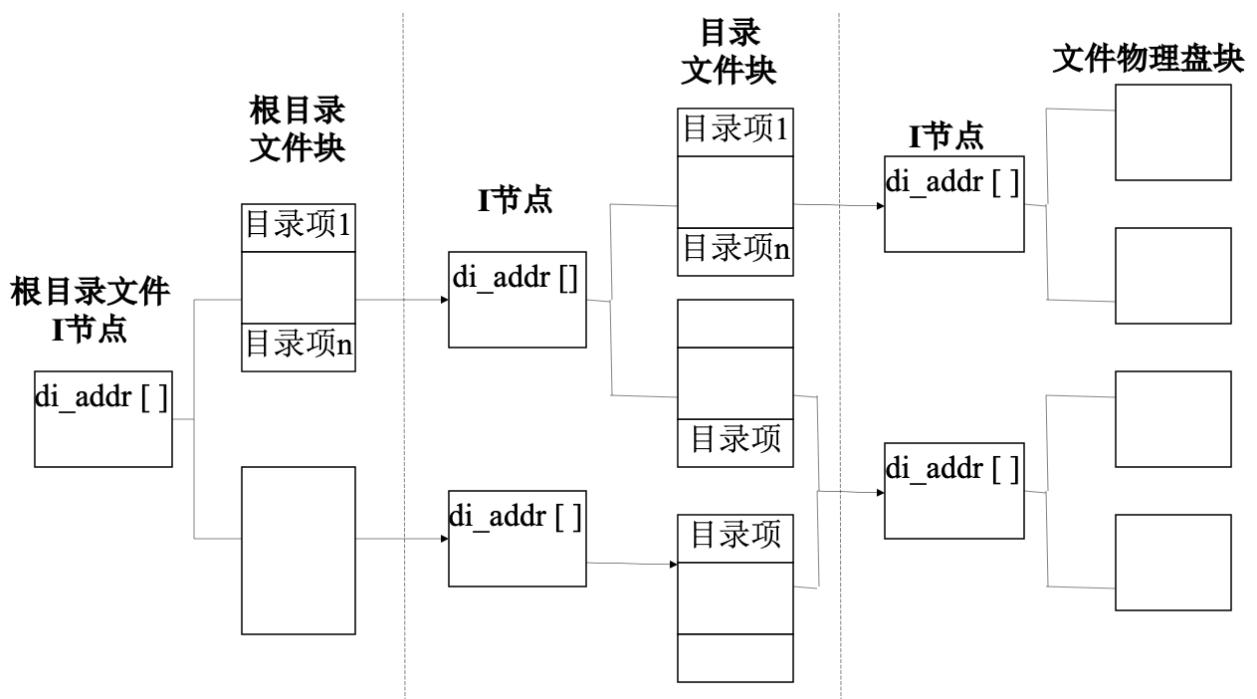
- 每个目录项由1个2字节的I节点号和14字节的文件名分量组成，一些UNIX系统的文件名分量可扩展

```
struct direct {  
    ino_t      d_ino;          /* 对应文件的I节点 */  
    char       d_name [14];    /* 文件名 */  
};
```



- 目录结构：

- UNIX将同一目录下的所有文件的目录项数据组成一个目录文件，一个目录文件可由一至多盘块组成，每个盘块可存放64个目录项
- 目录文件中的前两个目录项的文件名比较特殊，它们为“.”和“..”，分别表示当前目录和父目录，它们的I节点号就是当前目录文件的I节点号和上一级目录文件的I节点号，这便于用户在目录系统中移动位置
- 根目录文件的I节点是存放在磁盘I节点区的固定位置上（由全局变量rootdir指示），故根目录文件不需要有目录项，它是无名的
- 从根开始，根目录的下级目录的名字存放在根目录文件的目录项中，并各有一个指针指示下级目录文件的I节点。如此一级一级扩展下去，直至最底层的数据文件或空的目录文件，构成了整个UNIX的树型目录结构



- 目录中的勾连：

- 在UNIX同一个文件系统中，同一层次或不同层次的目录项可指向同一个文件的I节点，这就是目录结构的勾连
- 目录的勾连可由系统调用link建立，这为文件的共享提供了方便，这也是UNIX为什么要将文件的目录项与I节点分开存储和管理的主要原因之一
- 目录搜索：
  - 对于绝对路径名，如/etc/passwd，系统先得到根I节点，再根据其中的地址索引表将根目录文件的盘块内容读入内存，并逐项比较，看其中是否有一个目录项的文件名与根的下一层路径名分量，如etc相匹配
  - 找到匹配项后，根据该项的I节点号再得到下层目录文件的I节点。通过该I节点的地址索引表，读入etc目录文件的物理盘块。在该盘块中再逐项比较，直至找到文件名为passwd的目录项并获得passwd文件的I节点和读取passwd文件的各个物理盘块内容
  - 如用户给出的是相对路径名，则除了起始搜索点是当前目录项的I节点外，其余并无不同之处

打开文件结构：

- 访问一个文件时，系统要从根目录或当前目录出发，循序读取和搜索各级目录文件磁盘I节点，索引结构等，找到文件物理块号后再存取文件数据。由于文件系统十分庞大，这个操作比较费时
- 要访问一个文件，最关键的是要取得该文件的I节点，由它可进一步获得文件的所有控制信息及数据块。为了提高打开文件后对文件访问速度，当读取该文件的I节点后，应当在对整个文件的访问期间内，在内存中保存该I节点的副本，并能通过一个方便快速的途径存取它，这就要建立打开文件管理机构
- 打开文件的管理机构包括三部分，它们是内存索引节点、系统打开文件控制块和进程打开文件表
- **内存索引节点：**
  - 磁盘I节点某些信息在文件打开的期间是不需访问和处理的，例如文件的最近访问、修改的时间等，要等关闭文件时，才根据当前的时间重新更新对应的记录项
  - 另一方面，为了便于检索内存I节点和在关闭时将I节点写入磁盘原先的位置，又要增加一部分信息

```

struct inode {
    struct inod      *i_forw;      内存I节点的散列队列双向循环勾连指针
    struct inod      *i_back;
    char             i_flag;       状态标志，如锁标志、修改标志等
    cnt_t            i_count;     引用计数，表示该文件打开了几次
    dev_t            i_dev;        文件所在的设备号
    ino_t            i_number;    对应的磁盘索引节点号
    struct {
        union {
            daddr_t   i_a[13];    常规文件或目录文件的地址索引表
            short     i_f[26]      管道文件的地址索引表
        } i_p;
        daddr_t     i_l         最近一次读入的文件逻辑块，用于预读
    } i_blk;
}

```

- **打开文件控制块：**

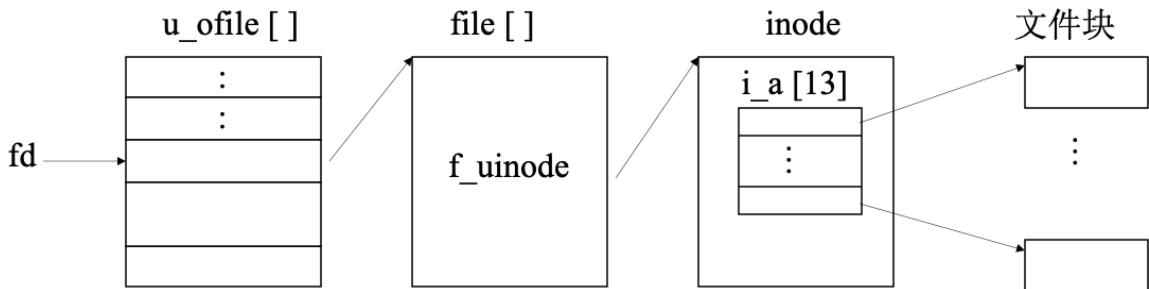
- I节点不能反映文件的动态特征，因此系统定义了打开文件控制块结构
- 一个文件可以被不同进程以不同的操作要求同时打开，而且对文件操作的当前指针也不同

```

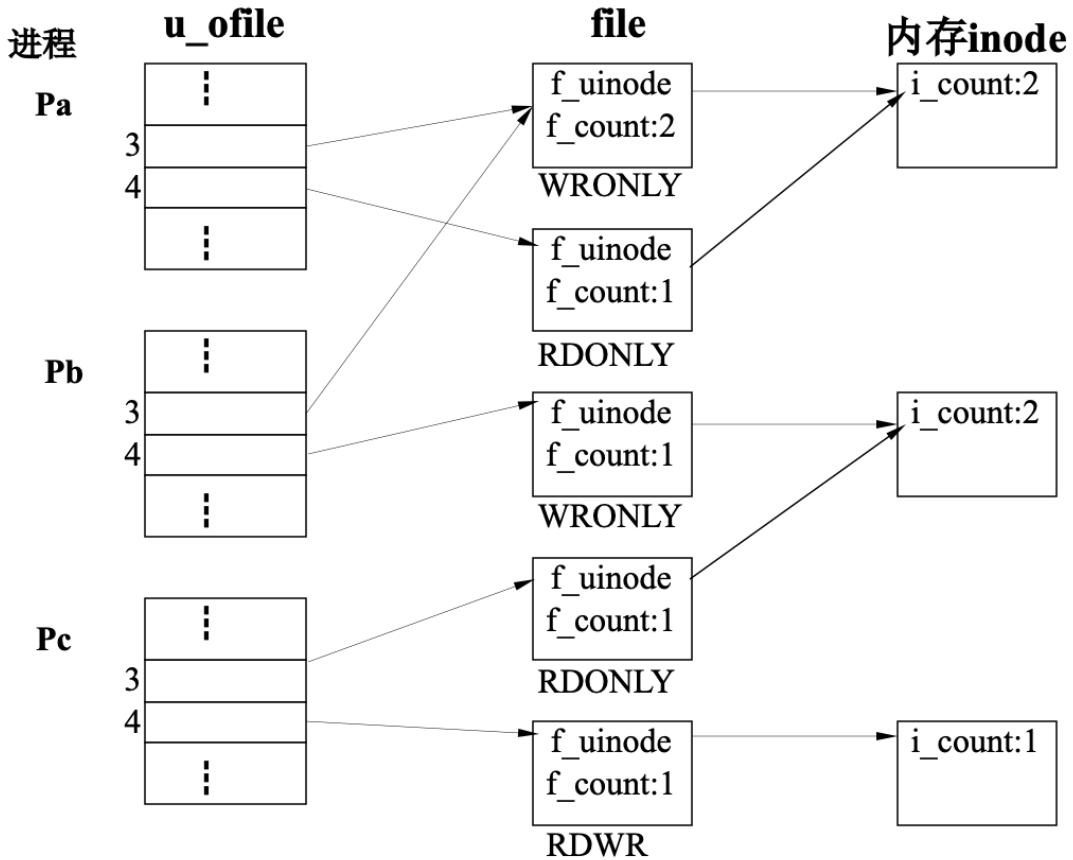
struct file {
    char f_flag;           操作方式，如写、读、追加写等
    cnt_t f_count;         共享该file结构的进程数
    union {
        struct inode *f_uinode;   指向内存I节点
        struct file  *f_unext;   空闲file的链接指针
    } f_up;
    union {
        off_t f_off;          读写位置指针
    } f_un;
};


```

- 每一表项在某一时刻只可处于2种状态：空闲状态或分配状态
- **进程打开文件表：**
  - 进程打开文件表或称打开文件描述字表是进程扩充控制块user结构中一个指针数组struct file \*u\_ofile[NOFILE]
  - 进程在打开文件时按下标序由低至高顺次使用该数组中的某一空闲项，在该表项中填入打开文件控制块file结构变量的地址，打开文件描述字的值就是该空闲项的下标值
  - 程序以打开文件描述字fd的值，索引指针数组u\_ofile，从对应的表项中可获得打开文件控制块地址，进一步可得到内存I节点的文件的控制信息及数据块



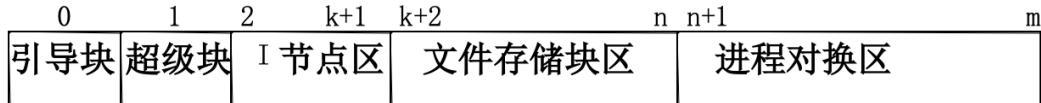
- 进程打开一个文件时，系统要为其分配一个空闲内存inode，将磁盘I节点的主要内容复制到其中
- 接着再分配一个空闲打开文件控制结构file，并使f\_uinode指针指向内存inode
- 最后再在进程user结构的u\_ofile中找到一个空闲项，在其中记下file结构的地址，并使数组索引值作为打开文件描述字返回给用户



- 上图表示了不同共享方式打开文件结构的各部分之间联系。进程Pa以写方式打开了3#文件后建立了`u_ofile`到`file`及`file`到内存`inode`的联系路径，此时`file`结构中的`f_count`值及`inode`结构中的`i_count`的值都为1
- 此后，进程Pa创建了子进程Pb，子进程Pb user结构的`u_ofile`的值与其父进程完全相同，故共享父进程Pa的所有打开文件，包括0#，1#，2#和3#文件。对于3#文件，子进程Pb与父进程Pa指向同一个`file`结构，故子进程对共享文件的操作方式也与父进程相同，不过`file`结构中的`f_count`的值变成了2，表示有两个进程打开文件的表项联系了同一`file`，但此时内存`inode`结构中的`i_count`值还是1
- 父进程在创建了子进程后，又以不同方式（只读方式）打开原先已打开的文件。系统将为其新分配一个`file`结构，使`file`结构中的`f_uinode`指向同一个内存`inode`（此时内存`inode`的`i_count`值变成2），并在进程打开文件表中下一可用项指向该`file`结构
- 此后，子进程Pb也独立打开了另一个文件，建立了`u_ofile`到该文件内存`inode`的联系
- 另一个独立的进程Pc则打开了进程Pb已打开的文件，且又独自打开了另一个文件

## 文件系统存储资源管理

- 文件系统存储机构的总体安排
  - 在UNIX系统中，文件以块为单位存放在磁盘中
  - 一个文件系统（或称文件卷）对应与一台逻辑设备。现在磁盘容量很大，一个物理磁盘可以划分成几个段，段内的存储空间是连续的，每个段各组成一台逻辑设备，这样一台磁盘设备就可以驻存几个文件系统
  - 如将磁盘的存储块线性展开，在一个文件系统的存储空间的安排如下图所示，图中各部分所占的块数由系统配置决定



- 引导块和进程对换区：

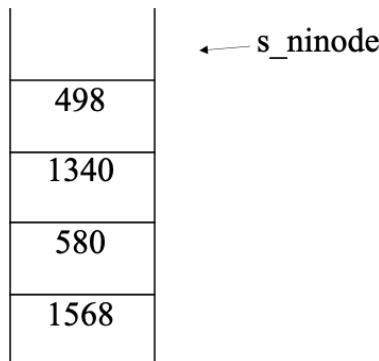
- 在块号为0的引导块中包含操作系统的自举程序，它既不属于文件系统，也不属于操作系统的一部分，因此，并非每个文件系统的这一块都有内容
- 进程对换区用于保存换出内存的进程映像和系统管理员设定的一些常驻盘对换区的执行程序。它构成一个顺序的线性存储空间，作为内存的扩充。进程映像的虚地址空间可以全部或部分地映射在对换区内
- 由于存取这部分存储块的内容不通过文件系统的管理机构，因此与内存交换数据的速度较快。同样，并非每个文件卷都包括这个盘区

- 文件系统超级块：

- 包含文件系统各部分所占的盘块总数、超级块直接管理的空闲I节点数和空闲I节点索引表、超级块直接管理的空闲盘块数和空闲盘块索引表、文件系统的类型和时间和状态信息等

- 空闲I节点的管理：

- UNIX使filsys采用栈方式管理，最多直接管理NINOD个空闲inode编号，并以s\_ninode作为栈指针，s\_ninode的值表示了当前filsys直接管理的空闲I节点数
- 当需要创建一个文件时，系统就要通过核心函数ialloc为新文件分配一个空闲的inode。如s\_ninode不为0，则通过语句ino=fp→s\_inode[--fp→s\_ninode]获得一个空闲inode号（fp指向filsys结构）



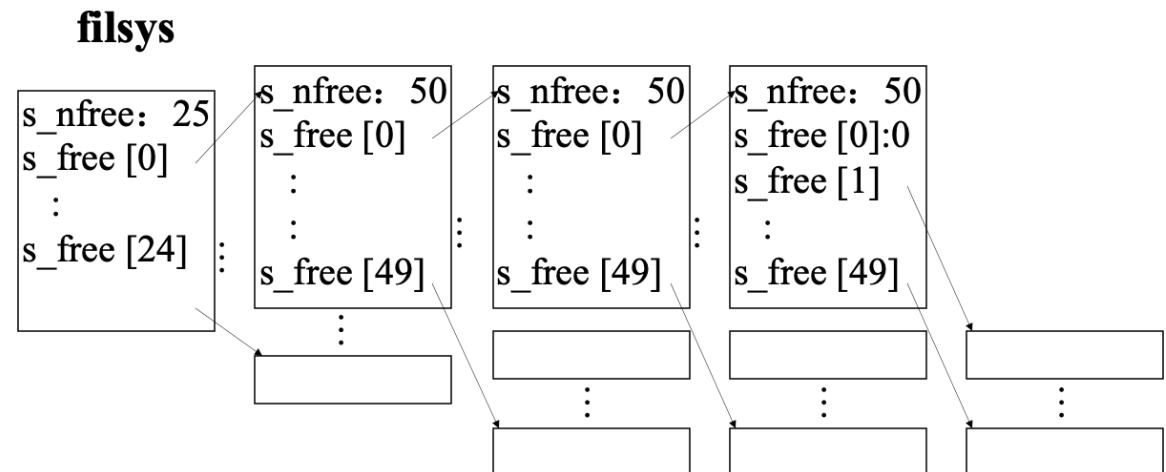
s\_inode [ ]

- 如表空，即s\_ninode为0，就要封锁超级块，从系统记住的磁盘I节点区编号最小的空闲I节点起，向后搜索I节点区，直至装满空闲I节点索引表s\_inode或搜索完了整个I节点区为止
- 搜索I节点区需要读盘操作，每读一个盘块最多可找到16个空闲I节点
- 当删去一个文件时，系统就要将对应的I节点释放，使其成为空闲。实现释放算法的核心函数是ifree
- 如空闲I节点表未满，把释放I节点的编号送入表中
- 如表已满，任其散布在磁盘I节点区，但如果其编号值小于I节点区搜索起点的I节点编号值，则重新调整搜索起点的位置值
- 在文件的创建和删除相对比较平衡时，I节点的分配和释放可望基本都在直接管理表s\_inode中进行，如此，对磁盘I节点区搜索的频度是相当低的

- 空闲文件存储块管理：

- 对空闲文件存储块的直接管理部分采用了与空闲I节点的管理相似的方法，由s\_free[NICFREE]最多直接管理NICFREE(50)个空闲存储块
- 由于文件存储块的数量比磁盘节点区的盘块大得多，且是非结构形式的，故不能采用搜索磁盘文件存储块区的方式获得空闲块。为此，系统在初始化时就将所有的空闲文件存储块组织成如下图所示

的分组链式结构



- 在分组链式管理结构中，由`s_nfree`记住当前登记的空闲块数。由`s_free[0]`指示的最后一个直接管理的空闲块又间接管理了下一组50个空闲块，以此类推
- 分配算法的程序是`alloc`，其主要过程是：如果`s_nfree`值不为0，则通过下面语句：`bno = fp→s_free[0]--fp→s_nfree;`；获得一个空闲存储块号
- 但当这是`s_free[0]`登记的最后一个空闲块时，由于该空闲块还间接管理了下一组空闲存储块，故在将它分配给文件之前，还要将其中的管理数据复制到超级块中，使间接管理转化为直接管理，这样超级块又登记了50个直接管理的空闲存储块了
- 释放算法：
  - 释放文件存储块时，将释放块号登记在`s_nfree`所指示的`s_free`表项中。但如发现表已满，不能再直接登记释放块时，就要将`filsys`直接管理的表项内容复制到释放块，使直接管理转变成间接管理，再将释放块号写入`s_free[0]`中，置`s_nfree`为1
  - 如此，`filsys`只直接管理了一个空闲块，但分组链则增加了一个组。当然，在分配和释放过程中还要对超级块进行判锁和上锁操作

## 6.7 管道文件和管道通信

管道通信的概念：

```
pr file.c | lp
```

- 这一命令使shell启动分页进程pr和打印进程lp，“|”告诉shell建立一个管道，把pr的标准输出连接到lp的标准输入。这一命令与下列命令序列的结果相同：

```
pr file.c > tmpfile
lp < tmpfile
rm tmpfile
```

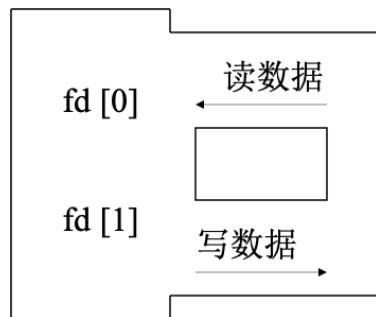
- 但后面的命令序列要等pr命令执行结束后再启动lp命令，而且tmpfile可能占用很大的存储空间，而前一个管道命令在pr进程执行期间，lp进程就将pr的输出作为其输入，将数据源源不断地送往打印机

管道文件：

- 在UNIX中，一个管道线就是连接前后两个进程的打开文件，前一个进程可以向该文件中顺序地写入数据，后一个进程可以从该文件中顺序地读出数据
- 数据的写入和读出以先进先出的方式进行，并由系统自动地处理两个进程间的调度、同步和数据缓冲，这类文件就称为管道（pipe）文件，简称管道
- pipe文件不是由用户命名的普通文件，它是使用系统调用pipe创建的，只能在与创建pipe的进程同一进程族内传递数据的打开文件。创建pipe文件的方式是：

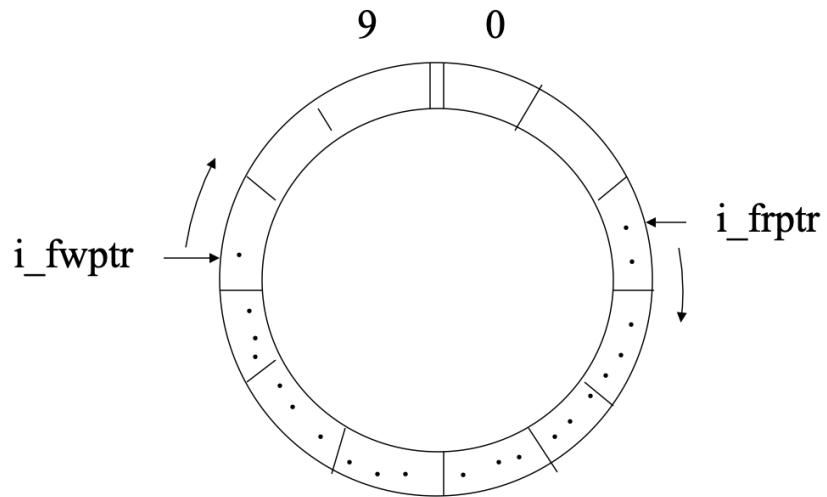
```
int fd[2], retv;
retv=pipe(fd);
```

- 管道是在管道设备的文件系统中建立起来的。在创建管道时，核心首先为其在管道设备文件系统中分配一个磁盘I节点和一个相应的内存I节点，再分配两个分别用于读打开和写打开的file结构，这两个file结构的f\_uinode指针指向同一个内存I节点
- 接下来在进程打开表u\_ofile中分配两个文件描述字表项，一个指向读打开file结构，一个指向写打开file结构，其中，在fd[0]中返回的是读端文件描述字，在fd[1]中返回的是写端文件描述字
- 在创建好管道以后，进程就可以使用与普通文件一样的读和写的系统调用，向管道的写端fd[1]送入数据，写入的数据可以从管道的读端fd[0]顺序读出

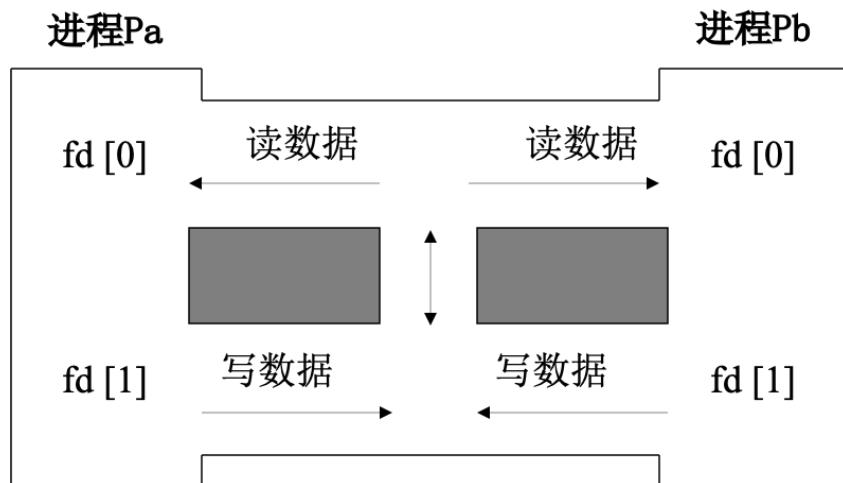


#### 管道的读写和关闭：

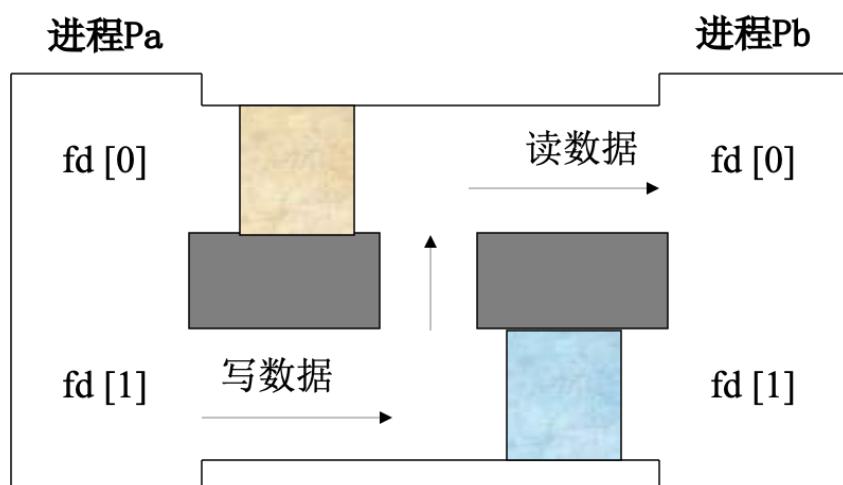
- 管道文件的最大长度为10个存储块，逻辑上构成一个线性空间
- 系统将这些存储块组织成尾部和头部“粘合”在一起的循环队列的存储区，并设同样也是循环移动的读（出队）、写（进队）两个指针，并在读写管道中的数据时由系统自动修改读写指针的值
- 初始化时读写指针都置为0，以后应当先向管道中写入数据，并自动移动写指针和增加管道文件的长度。只有当管道中存有数据后才能读管道、移动读指针和减小文件的长度
- 在对管道文件进行读写时，循环移动的读或写指针都不能“跑”到对方的前面。当读指针追上了写指针时，表示管道空，读进程只能睡眠等待，直到写进程再次向管道中送入数据后才唤醒等待读的进程
- 另一方面，当写指针追上读指针时表示管道满，写进程就要睡眠，直至读进程从管道中读出一部分信息后才将写进程唤醒



- 进程在生成管道文件后，为了使pipe机构用于进程间通信，一般总要接着创建一个或多个子进程



- 为了避免混乱，一个管道最好只为两个进程共有，而且一个进程只用其写端，另一个进程只用其读端，于是就要分别关闭自己的一个读端或一个写端



```

main ( )
{
    char *msg = "A message form parent. "
    int chan [2];
    char buf [100];

```

```

pipe (chan);
if (fork ( )) {
    close (chan [0]); /* 父进程关闭管道读端 */
    write (chan [1], msg, strlen (msg)+1); /* 写管道 */
    close (chan [1]);
} else {
    close (chan [1]);
    read (chan [0], buf, sizeof (buf)); /* 子进程读管道 */
    printf ("Child Proedss: % s\n", buf);
    close (chan [0]);
}
}

```

有名管道：

- 只有与生成管道文件的进程属同一簇的进程之间才能利用无名管道pipe机构进行通信，其它的进程甚至感觉不到该管道的存在
- 有名管道则像普通文件一样有其目录项，在文件系统中能长久地存在，任何有访问权限的用户都可以通过路径名来打开它，进而存取其中数据，因此无关的进程就可以通过有名管道进行通信
- 尽管有名管道的结构像文件，但它的功能像管道。数据按先进先出的次序写入或读出，核心保证读写FIFOs的操作是原子级的。核心也同样提供对有名管道文件的lseek操作
- 有名管道的I节点与无名管道一样，除了10个直接地址索引项外，剩余的索引项改成了读写指针和读写进程计数等
- 创建有名管道：
  - 目录、特别文件和有名管道文件只能用mknod创建。但是，创建普通文件、目录和特别文件的mknod调用只对超级用户开放，而所有的用户都能用mknod创建有名管道文件

```

#include <sys/types.h>
#include <sys/stat.h>
int mknod(pathname, mode, device);
char *pathname; /* 有名管道文件名 */
int mode, device; /* 存取权限等，设备号 */

```

- 有名管道的打开：
  - 在创建了有名管道后，任何进程都可以用与打开文件相同的系统调用打开一个有名管道文件，如：fd=open("fifos",O\_WRONLY);
  - mknod只是创建有名管道文件，但并不打开它。通常，当为读而打开一个FIFOs时，进程就会进入等待状态，直至另一进程为写而打开同一个FIFOs。反之为写而打开时，进程也要等待读打开
- 有名管道的读写、关闭与删除：
  - 有名管道的读、写、关闭调用与无名管道一样，在关闭了一个有名管道后，它所使用的磁盘空间全部释放，这点与普通文件不一样，但有名管道本身并没有消失也即它所占的目录项和磁盘I节点还在。要删除一个有名管道，可利用与删除一个文件相同的调用：unlink(pathname);
- 有名管道进行通信的例子：
  - 下面是两个独立的程序使用有名管道进行通信的例子

- 第一个程序创建一个所有用户都有读写许可的有名管道fifo，并以写方式打开它，此后循环地向其中写入数据，第二个程序以读方式打开同一个有名管道，并从中读数据

```
#include <stdio.h>      // NULL 的说明
#include<sys/stat.h>   // S_IFIFO的说明
#include <fcntl.h>     // O_WRONLY 的说明
main( )
{
    int fd;
    char buf [256];
    mknod ("fifo", S_IFIFO|0666,0);
    fd = open ("fifo",O_WRONLY);
    while (gets(buf)!=NULL)
        write (fd, buf, strlen(buf)+1);
    close (fd);
}

#include <fcntl.h> // O_RDONLY的说明
main ( )
{
    int fd, n;
    char buf [256];
    fd = open ("fifo", O_RDONLY | O_NDELAY);
    while ((n = read (fd, buf, sizeof(buf)))!= -1)
        if (n == 0)
            sleep (NAPTIME);           /* may do something else */
        else
            puts (buf);
    close (fd);
}
```