第一章：引论

操作系统是运行在内核态的软件，为程序猿提供资源集抽象以及管理硬件

1.1.2

主要任务：记录那个程序在用什么资源，管理资源分配，评估使用代价，调节冲突

1.3.1

1.操作系统必须知道所有的寄存器，以便中断时保存进度

2.用户程序在用户态运行时，仅允许执行至灵级的一个子集，一般不能调用 IO 和内存保护指令

3.陷阱：

a. 用于执行系统调用

b. 多数由硬件引起，用于警告异常

4.超线程：无并行处理，线程切换纳秒级

1.3.2 存储器

1. 寄存器（和 CPU 一样快）-》高速缓存（多级缓存）-》主存（RAM ROM EEROM 闪存）

1.3.3

上下文切换：多道程序系统中从一个程序切换到另一个程序

1.3.5

1. 设备驱动程序：控制 IO 设备，与控制器对话并收发命令
2. 设备存储器：映射到操作空间
   A．优点：不需要特定 IO 指令
   B．缺点：占地址空间（8088）
3. 实现输入输出的方法：
   A． 忙等待：设备驱动循环检查 IO
   B． 操作完成时中断
   C． 使用特殊的直接存储器访问芯片 DMA

1.3.6

1. USB：通用串行总线，键盘鼠标等慢速设备

1.3.7 启动

1. 加电-》BIOS 检查硬件-》BIOS 查询启动设备（设备第一扇区用启动签名才可以作为启动设备）-》硬盘第一区（MBR），分区表，超级块等

1.5.1 进程

1. 本质：正在执行的程序的实例，地址空间（core image 进程可读写，有数据和堆栈）。
2. 相关：资源集（寄存器，报警，文件清单等）
3. 容许运行一个程序所需要所有信息的容器
4. UID 与 GID

1.5.3

1. IO 设备的分类：
   A．块设备：硬盘，可随机读取
   B．字符特殊文件：键盘鼠标

2．管道：虚文件，连接进程

1.6 系统调用

1. 用户程序与操作系统交互：处理抽象
2. 能进入内核的过程调用

用户态切换到核心态三种方法：中断，异常，系统调用

3.TRAP 指令：副作用切换到内核态

1.7.3 微内核

1. 高可靠性，把操作系统划分成小的，定义良好的模块，只有微内核运行在内核，其他是普通用户程序
2. 设备驱动：崩溃不会导致系统死机
3. 机制与策略分离

第二章：进程与线程

2.1 进程模型

1. 多道程序设计：CPU 在多个程序之间快速切换
2. UNIX: 开始是相同，之后不同。Windows：一直不同。
3. 进程退出的原因：
    1. 正常退出；
    2. 出错退出；（异常处理）
    3. 严重错误；（非法指令，引用错误内存，除零错误）
    4. 被杀死
4. 进程层次

    Windows: 没有层次的概念，所有进程地位相同

    Linux：进程及进程的子女们组成进程组
5. 进程的三种状态：
    1. 运行态（实际占用 CPU）
    2. 就绪态（可运行）
    3. 阻塞态（等待外部事件）
6. 进程表：储存进程状态（程序计数器，堆栈指针，内存分配状况，打开的文件状态。账号等）
7. 中断向量：与每一个 IO 类关联
    1. 中断发生时，中断硬件程序将进程表中的重要数据压入堆栈，计算机跳到中断向量的地址
    2. 汇编语言设置新的堆栈（无法用 C 语言这类高级语言来描述）
8. 多道程序设计
    1. 假设一个进程等待 IO 与停留在 CPU 的时间比为 p，n 个进程时，CPU 使用率为

使用率 = 1 – p^n

2.2 线程（http://www.cnblogs.com/way_testlife/archive/2011/04/16/2018312.html 进程与线程）

1. 定义：传统操作系统中，每个进程有一个地址空间和一个控制线程
2. 线程将应用程序分解成可以并行运行的多个顺序线程
3. 使用多线程的原因：
    1. 并行实体共享同一个地址空间和所有可用数据的能力
    2. 线程更轻量级，所以他们比进程更快创建和撤销
    3. 同时需要大量 IO 和 CPU 计算时，多线程允许多个活动彼此重叠进行，从而加快执行速度
    4. 多核系统中，多线程可以真正实现并行
    5. 例子：多线程/单线程 web 服务器

1. 第三种设计（有限状态机： 并行，非阻塞系统调用，【中断】）：唯一的线程对请求进行考察，如果需要 IO，则启动一个非阻塞 IO，服务器在表格里记录当前请求，然后处理下一个事项。

6. 线程模型
    1. 进程：集中程序运行的相关资源（地址空间，全局变量等）
    2. 线程：程序计数器，寄存器，堆栈，共享的地址空间，多个线程的执行能力。
7. 线程之间没有保护：
    1. 不可能
    2. 不需要（线程之间是合作关系）
8. 每个线程都有自己的堆栈
9. thread_yield：不同于进程，线程无法使用时钟中断强制线程让出 CPU
10. 线程引入的问题
    1. fork 系统调用是否应该复制子线程
    2.共享文件冲突

4 线程实现

1. 用户空间实现
    1. 每个进程需要有其专门的线程表，由运行时系统管理
    2. 优点
        1. 可以在不支持线程的操作系统上实现多线程
        2. 线程切换速度快（调用运行时系统的过程，不需要刷新和上下文切换）
        3. 允许每个进程有自己定制的调度算法
        4. 有较好的拓展性（内核线程需要固定的表格空间和堆栈空间）
    3. 缺点
        1. 某个线程进行阻塞调用会引起所有其他线程阻塞
            1. 使用非阻塞系统调用
            2. 阻塞提前通知（select 系统调用）
        2. 页面故障阻塞其他线程
        3. 除非线程放弃 CPU，否则其他线程（包括调度线程）无法运行（没有时钟中断）
            1. 运行时系统也给与时钟中断：不好，不可能而且开销大
2. 内核线程
    1. 内核有记录所有线程的线程表
    2. 使用环保方法回收线程
    3. 在线程级别使用调度算法：如果线程的操作比较多，会带来很大的开销
    4. 信号：是发给进程的。当多个线程注册时，会出问题
3. 混合实现
    1. 使用内核级线程，将用户级线程与某些或全部内核线程多路复用（很灵活）
4. 调度机制
    1. 内核给每个进程安排一定数量的虚拟处理器并且让运行时系统分到线程上
    2. 进程被阻塞后，内核通知运行时系统（upcall）
    3. 根据中断决定是否继续
5. 弹出式线程
    1. 一个消息的到达导致系统创造新的线程处理消息-》弹出式线程
    2. 优点：没有历史，创建迅速

6. 重写单线程代码
    1. 私有的全局变量
    2. 可重入的库
        1. 重写整个库
        2. 为每个过程提供 wrapper，标志该库正在被使用中
    3. 信号：内核不知道用户级线程，因此不容易将信号发给正确的线程
    4. 堆栈管理：内核不了解线程，无法自动增长，可能会造成线程堆栈出错。

2.3 进程间通信

1. 三个基础问题
    1. 进程如何把信息传递给另一个·
    2. 确保两个或多个进程在关键活动中不会交叉
    3. 进程执行的正确顺序
2. 竞争条件：两个或多个进程读写共享数据，最后结果取决于进程执行的精确时序。
3. 临界区：多个进程中访问共享区域的程序段
    1. 互斥：不能同时多个进程使用共享变量或者文件
    2. 临界区解决方案四个条件
        1. 任何两个进程不能同时处于临界区
        2. 不应对 CPU 的数量和速度有任何的假设
        3. 临界区外的程序不应阻塞其他程序
        4. 不能使程序无限期等待进入临界区
    3. 解决方案（基于忙等待：进程如不能进入互斥区，则会一直原地等待）

        缺点：
        1. 可能导致优先级反转问题
        2. 浪费 CPU
        3. 用户级线程会一直忙等待，从而没用办法让拥有锁的线程运行

        1. 屏蔽中断：进程进入临界区之后屏蔽所有中断（CPU 不会切换）

        评价：不好的方案
        1. 不能把屏蔽中断的权力交给用户进程（不打开中断则系统会终止）
        2. 多处理器时不能解决互斥

        2. 锁变量：共享锁，程序在进入临界区之前检查锁的值

        评价：无法解决临界区问题

        3. 严格轮换法

```
while (TRUE) {
    while (turn != 0)        /* 循环 */
    critical_region();
    turn = 1;
    noncritical_region();
}
```

```
while (TRUE) {
    while (turn != 1)        /* 循环 */
    critical_region();
    turn = 0;
    noncritical_region();
}
```

        评价：
        1. 可以解决，但为忙等待。只有有理由认为等待时间很短的情况下才使用忙等待（锁被称为自旋锁）
        2. 在一个进程比另一个进程慢很多的情况下，不好（违反条件三）

        4. Peterson 解法

        评价：可以解决，满足四大条件

```
#define FALSE  0
#define TRUE   1
#define N      2                                    /* 进程数量 */

int turn;                                           /* 现在轮到谁？ */
int interested[N];                                  /* 所有值初始化为0（FALSE） */

void enter_region(int process);                     /* 进程是0或1 */
{
     int other;                                     /* 其他进程号 */

     other = 1 - process;                           /* 另一方进程*/
     interested[process] = TRUE;                    /* 表明所感兴趣的 */
     turn = process;                                /* 设置标志 */
     while (turn == process && interested[other] == TRUE); /*空语句 */
}

void leave_region(int process)                      /* 进程：谁离开？ */
{
     interested[process] = FALSE;                   /* 表示离开临界区 */
}
```

5. TSL 指令（硬件解法）

TSL RX LOCK：测试并加锁，把 LOCK 值读到 RX 并在 LOCK 上存入 1。原子操作可以阻止所有处理器访问 LOCK（屏蔽中断只能屏蔽本地处理器）

4. 睡眠——唤醒方案（进程无法进入临界区时会阻塞）
   1. 原语：生产者——消费者问题：一个发给未睡眠进程的信号丢失了
   2. 解决方案
      1. 唤醒等待位：唤醒时生产者置 1，消费者睡眠前检查该位，若为 1，则清除该位并继续保持清醒（不好，多进程时需多个等待位）
      2. 信号量（semaphore）：检查数值，修改变量等应为原子操作
         1. 在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。
         2. 实现方法：操作系统在执行以上事务时屏蔽中断
         3. 另一种用途：互斥锁
      3. 互斥锁：只需要一个二进制位

```
mutex_lock:
     TSL REGISTER,MUTEX
     CMP REGISTER,#0
     JZE ok
     CALL thread_yield
     JMP mutex_lock
ok:  RET


mutex_unlock:
     MOVE MUTEX,#0
     RET
```

与忙等待差异：在未获得锁时，调用另外的线程

4. 如何共享锁？
   a) 共享数据结构可以存放在内核并且只能用系统调用访问

b) 让进程与其他的进程共享部分地址空间

5. 条件变量
    1. 允许线程因为某些未达到的原因而阻塞
    2. 允许被阻塞而等待的线程原子性进行
    3. 经常与互斥量一起使用：让一个线程锁住一个互斥量，当他不能获得期待结果时等待一个条件变量。有另外一个线程发信号唤醒。
    4. 条件变量不会存在于内存（发出后丢失）

5. 管程：一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据
    1. 任何一个时刻管程里只能有一个活动的进程
    2. 第二个进程将被挂起直到活跃进程离开
    3. 如果一个条件变量上有若干程序在等待，则执行 signal 操作以后，系统只能从中任选一个恢复
    4. 实现：JAVA synchronized 关键字
    5. 劣势
        1. 操作系统是 C 写的，没有管程的概念
        2. 分布式系统无法避免

6. 消息传递
    1. 使用系统调用 send 和 receive
    2. 潜在问题：消息丢失，消息认证，消息传递速率低
    3. 编址方法
        1. 为每个进程分配一个唯一的地址
        2. 引入新的数据结构：信箱。消息发往信箱。（解决消费者生产者问题）

7. 屏障：在每个阶段结尾安放屏障，当一个进程达到屏障时，它被屏障阻拦，直到所有进程都达到屏障

2．4 调度

1. 定义：多个进程就绪是，CPU 选择下一个将要执行的进程的方法。
2. 进程行为：IO 密集型（web 服务器）和 CPU 密集型（象棋软件）。越来越多的软件倾向于 IO 密集型，应该多运行这类进程以保持 CPU 使用率
3. 何时调度：
    1. 创建新进程时，应该先调度父进程还是子进程
    2. 进程退出时
    3. 进程阻塞时（IO，信号量等）
    4. IO 中断发生时
4. 调度算法定义:
    1. 抢占式：挑选一个进程，并且让进程运行某个固定时段的最大值，之后挂起
    2. 非抢占式：挑选一个进程，让进程运行直到其阻塞或者自动释放 CPU
5. 调度算法目标（所有系统）：
    1. 公平：每个进程公平的 CPU 份额
    2. 策略强制执行：所宣布的策略执行
    3. 平衡：系统所有部分都忙碌
6. 调度算法分类：
    1. 批处理：处理周期性作业，广泛的商业应用
        1. 调度算法目标：

1. 吞吐量 throughout：系统每小时完成的作业数量
2. 周转时间 turnaround time:从一个批处理作业提交时刻开始直到作业完成时刻位置的平均时间
3. CPU 利用率

2. 具体调度算法：
    1. 先来先服务（first-come）
        优点：易于理解，便于实现
        缺点：同时运行 IO 密集型和 CPU 密集型程序时效率低下
    2. 最短作业优先（不可抢占）
        1. 计算公式：$T = \frac{na+(n-1)b+(n-2)c+\cdots+z}{n}$ 其中 $a < b < c \ldots . < z$
        2. 只有所有作业都可同时运行且运行时间可以预知的情况下才能使用
    3. 最短剩余时间优先：最短作业优先的抢占版
        1. 新作业到达时，整个时间与当前进程的时间做比较，运行剩余时间较少的作业

2. 交互式系统
    1. 调度算法目标：
        1. 响应时间：发出指令到相应的时间
        2. 均衡性：满足用户的期望
    2. 具体调度算法
        1. 轮转调度：最古老，最简单，最公平，使用最广
            1. 每个进程被分配时间片，时间片结束时切换进程
            2. 时间片
                1. 过长：对短命令的交互请求时间变长
                2. 过短：过多进程切换，降低 CPU 效率
        2. 优先级调度：每个进程赋予优先级，优先级高的进程先执行
            1. 使 IO 进程良好运行的算法：进程优先级为 1/f，其中 f 是进程在上一时间片中实际运行的比例（IO 进程会得到较高优先级）
            2. 多优先级队列：低优先级可能出现饥饿现象
        3. 多级队列
            1. 实现方法：最高优先级的进程获得 1 个时间片，之后优先级下降并在下次运行时获得两倍的时间片
        4. 最短进程优先
            1. 根据过去进程运行时间预测并执行估计运行时间
            2. 老化算法：当前测量值和先前估计值进行加权平均（1/2 比较好，右移一位即可）
        5. 保证调度：每个进程获得 1/n 的时间
        6. 彩票调度：向进程提供各种系统资源的彩票，某些进程可以协作获得更多彩票。不错的方法
        7. 公平分享调度：每个用户获得等量的 CPU 时间

3. 实时系统
    1. 调度算法目标：
        1. 满足截止时间：避免丢失数据

2. 可预测性：在多媒体系统中避免品质降低

2. 分类

硬实时：必须满足绝对的截止时间

软实时：虽然不希望偶尔错失，但是可以容忍

2. 周期性时间公式

M 个周期时间，事件 i 以周期 Pi 发生，需要用 Ci 的 CPU 时间，那么这个系统可调度的条件是

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \leq 1$$

7. 调度机制与策略：将调度机制与调度策略分离

例子：系统使用优先级调度，但是把赋予优先级的行为委托给进程。

8. 线程调度

1. 用户级线程：内核不知道线程，用户进程调用专门定制的线程调度程序
2. 内核级线程：时间片。

2.5 经典 IPC(Inter-Process Communication)问题

1. 哲学家进餐

1. 随机事件解法：可能因为不可靠的随机数字而失败

```
#define N           5                   /* 哲学家数目 */
#define LEFT        (i+N-1)%N           /* i 的左邻居编号 */
#define RIGHT       (i+1)%N             /* i 的右邻居编号 */
#define THINKING    0                   /* 哲学家在思考 */
#define HUNGRY      1                   /* 哲学家试图拿起叉子 */
#define EATING      2                   /* 哲学家进餐 */
typedef int semaphore;                  /* 信号量是一种特殊的整型数据 */
int state[N];                           /* 数组用来跟踪记录每位哲学家的状态 */
semaphore mutex = 1;                    /* 临界区的互斥 */
semaphore s[N];                         /* 每个哲学家一个信号量 */

void philosopher(int i)                 /* i：哲学家编号，从0到N-1 */
{
    while (TRUE) {                      /* 无限循环 */
    }
}

void take_forks(int i)                  /* i：哲学家编号，从0到N-1 */
{
    down(&mutex);                       /* 进入临界区 */
    state[i] = HUNGRY;                  /* 记录哲学家i处于饥饿的状态 */
    test(i);                            /* 尝试获取2把叉子 */
    up(&mutex);                         /* 离开临界区 */
    down(&s[i]);                        /* 如果得不到需要的叉子则阻塞 */
}

void put_forks(i)                       /* i：哲学家编号，从0到N-1 */
{
    down(&mutex);                       /* 进入临界区 */
    state[i] = THINKING;                /* 哲学家已经就餐完毕 */
    test(LEFT);                         /* 检查左边的邻居现在可以吃吗 */
    test(RIGHT);                        /* 检查右边的邻居现在可以吃吗 */
    up(&mutex);                         /* 离开临界区 */
}

void test(i)                            /* i：哲学家编号，从0到N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
```

2. 读者-写者问题：不能让写者无法写入数据

第三章 存储管理

3.1 无存储器抽象：程序直接访问物理内存

1. 不能在内存里【同时】运行多个程序
2. 实现并行可采用多线程编程：不现实
3. 运行多道程序的方法
    1. 交换：保证一个时刻只有一个程序在内存里
    2. 特殊硬件：防止进程相互干扰
        缺点：不能直接使用绝对物理地址，装载器需要一定方法分别地址和常数
4. 一些问题
    1. 用户系统如果可以寻址内存的每个字节，就有可能破坏操作系统
    2. 多道程序运行非常困难

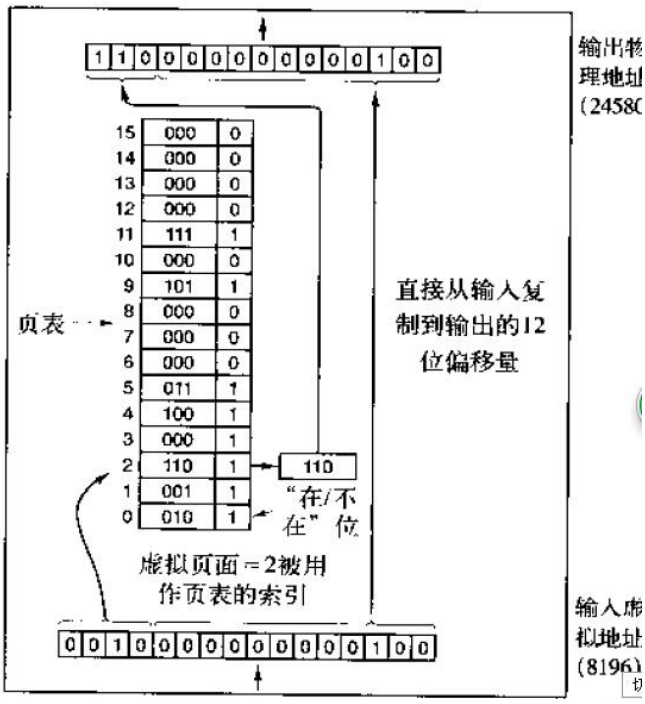
3.2 地址空间抽象

1. 概念：一个进程可用于寻址内存的一套地址集合，一般独立与其他进程的地址空间，除某些情况下需要共享
2. 解决方法
    1. 动态重定位：基址寄存器和界限寄存器
        缺点：，每次访问都需要加法和比较运算
    2. 交换技术：将一个进程完整调入内存，使其运行一段时间后再存回硬盘
        1. 换入后使用硬件重定位
        2. 产生空洞，需要内存紧缩
        3. 现代程序需要内存较多，但是硬盘速度慢，交换时间长
        4. 注意：进程大小如果增长，需要辅助手段。
            解决方案：移入时分配额外内存，再次交换出去时不交换额外的。
    3. 空闲内存管理
        1. 位图存储：每个字需要 1 位位图
            优点：空间少
            缺点：分配内存时需要在位图里寻找指定长度的 0 串，费时。
        2. 链表管理：维护一个记录已分配内存段和空闲内存段的链表，每个链表的节点或包含一个进程，或者两进程之间的空闲区
            优点：进程终止或者被换出是链表的更新非常直接
            具体分配方法：
                1. 首次适配 first fit：沿列表搜索直到找到一个足够大的空间
                2. 下次适配 next fit：首次适配之后，从上次结束的地方开始搜索。（性能略低于首次适配）
                3. 最佳适配 best fit：搜索整个链表，找出能容纳进程的最小空闲区。（速度较慢，会产生大量无用的小空闲区）
                4. 最差适配 worst fit：分配最大的空闲区（也不好）
                5. 为进程和空闲区保存单独链表
                    a) 优点：提高分配算法速度
                    b) 缺点：增加内存复杂度和内存释放速度变慢

6. 快速适配 quick fit：为常用大小的空闲去维护单独的链表

3.3 虚拟内存

1. 基本思想：每个程序拥有自己的地址空间，这个空间被分割成许多块，每一块称作一页 page，page 被映射到物理内存，但是只有某些页真正在内存中。操作系统调用不存在页面的时候，引发缺页中断
2. 分页技术
    1. 有程序产生的地址被称为虚拟地址，它们被称为虚拟地址空间。地址空间里的单元成为页面，在物理内存对应的单元称为页框 page frame。
    2. 是用虚拟内存时，地址不是送到内存总线，而被送到内存管理单元(memory management unit, MMU)MMU 将虚拟地址映射成物理地址。内存对 MMU 一无所知
    例子：16 位虚拟地址对应 15 位物理地址，输入的虚拟地址被分为 4 位页面和 12 位偏移量，在页表中查询，之后组成 15 位实际地址



3. 页表
    1. 本质：一个把虚拟页面映射成页框的函数
    2. 页表项的结构
        1. 页框号
        2. 在/不在位
        3. 保护位
        4. 访问位和修改位
        5. 禁止高速缓存位：对映射到设备寄存器的页面非常重要
    3. 不在内存的页面的硬盘地址不是页表的一部分：缺页中断时，该页面的磁盘地址等信息保存在操作系统的内部软件上
4. 加速分页
    1. 主要问题
        1. 虚拟地址到物理地址的映射必须非常快
        2. 如果虚拟地址很大，页表也会很大
    2. 极端解决方案

1. 页表全部在寄存器里
   优点：简单，映射过程中不需要访问内存
   缺点：页表很大时代价高昂，每一次上下文切换都必须装载整个页表，性能低
2. 页表全在内存里
   优点：上下文切换快
   缺点：每条指令执行都需要访问内存

3. 现实解决方法
   1. 转换检测缓冲区 translation lookaside buffer，TLB
      1. 本质：把虚拟地址直接映射成物理地址的小型硬件
      2. 使用：传入的虚拟页号并行匹配
      3. 未命中：MMU 未检查到有效匹配项，则进行页表查询，从 TLB 淘汰一个表项，并用新页表项代替
      4. 软失效：页面不在 TLB 但是在内存：更新 TLB，几纳秒
         硬失效：页面不在内存：缺页中断，磁盘 IO，几毫秒
   4. 大内存页表
      1. 多级页表：32 位虚拟地址划分成页表 1，页表 2，偏移量三部分
         原因：避免全部页表一直在内存（一般程序只有少量的正文段，数据段和堆栈段，中间是大量空洞，访问空洞时强制缺页中断并发信号）
      2. 倒排页表（inverted page table，64 位机器）：每一个页框对应一个页面
         优点：节省空间
         缺点：从虚拟地址到物理地址的转换困难（必须搜索整个表项来寻找（进程， 虚拟页面）对）
         解决方案：建立一张散列表，用虚拟地址来散列，相同 hash 值的表链在一起

3.4 页面置换算法：缺页中断时，操作系统必须淘汰一个旧页面/web 服务器淘汰一个高速缓存项

1. 最优页面置换算法：用 X 条指令后才会用到来标记页面，置换 X 最大的页面
   优点：已知最好算法
   缺点：不可能实现
2. 最近未使用页面置换算法（not resently used，NRU）：启动进程时，系统把页面的访问位(R)和修改位(M)置零，R 位被定期清零，以区别最近是否访问过。页面分为 4 类：
   1. 没有被访问和修改
   2. 没有被访问，已修改
   3. 被访问，没有修改
   4. 访问修改

   NRU 算法在类标号最小的非空集里挑选一个页面淘汰
   优点：简单易实现
   缺点：性能不是最好的
3. 先进先出页面置换算法（FIFO）：淘汰最老页面，可能会淘汰很有用的，不单纯使用
4. 第二次机会页面置换算法（second chance）：检查最老页面的 R 位，如果是 0，置换之，否则清零并把该页面放到链表尾端。
   本质：找寻一个最近的时钟间隔以来没有被访问过的页面
5. 时钟页面置换算法（clock）：第二次机会页面的时钟版

6. 最近最少使用页面置换算法(LRU, Least Recently Used)

实现方法：需要在内存里维护所有页面的链表，最近最多的使用的页面在表头，最近最少的页面在表尾

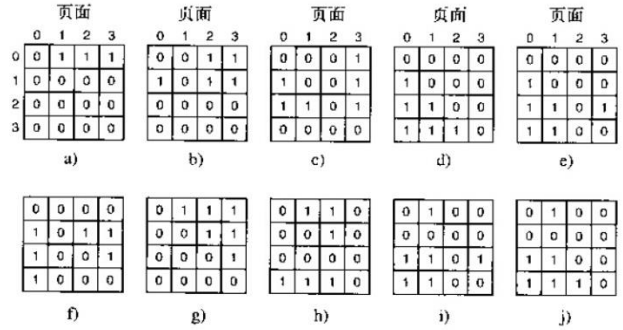
缺点：每次访问内存都必须更新整个链表，费时

特殊硬件实现方法：

1. 64 位计数器 C，每个页表都有一个计数器，每次执行完以后加 1，缺页中断时置换计数器最小的页面
2. N*N 矩阵。初值为 0，访问页框 k 时，硬件首先把 k 行的位都设置成，再把 k 列的位都设置成 0。任何时刻二进制数值最小的行对应页面会被置换

0 1 2 3 2 1 0 3 2 3

访问页面0后的状态如图3-17a所示，访问页1后的状态如图3-17b所示，以此类推。



7. 软件模拟的 LRU

1. 最不常用算法（NFU）：每个页面与软件计数器相连，每次时钟中断是操作系统扫描所有页面并将 R 位加到计数器上，置换计数器值最小的页面

缺点：记忆太久，操作系统可能会置换有用的页面

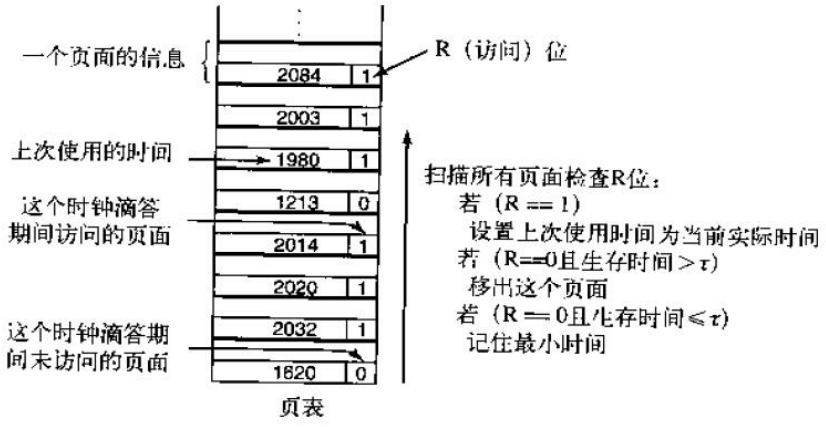
2. 老化算法 aging：在 R 位加入之前，计数器右移一位（相当于过去访问次数*1/2），然后把 R 加到计数器最左边。置换计数器值最小的页面。

缺点：不能确定时钟滴答中哪个页面被先访问

计数器只有有限位数，限制了其对以往页面的记录（实际中一般够用）

8. 工作集页面置换算法

1. 定义：一个进程当前正在使用的页面的集合称为工作集
2. 分页系统会设法跟踪工作集，在进程运行之前就预先调页
3. 大多数进程不会均匀访问地址空间，而是一小部分页面
4. 缺页中断时，淘汰不在工作集的页面。
5. 近似：过去的 t 秒实际运行时间中进程所访问的页面的集合



6. 算法：

9. 工作集时钟页面置换算法：工作集算法的时钟版本

10.小结：最好的算法是老化算法和工作集时钟算法

算　法	注　释
最优算法	不可实现，但可用作基准
NRU（最近未使用）算法	LRU的很粗糙的近似
FIFO（先进先出）算法	可能抛弃重要页面
第二次机会算法	比FIFO有大的改善
时钟算法	现实的
LRU（最近最少使用）算法	很优秀，但很难实现
NFU（最不经常使用）算法	LRU的相对粗略的近似
老化算法	非常近似LRU的有效算法
工作集算法	实现起来开销很大
工作集时钟算法	好的有效算法

3.5 分页系统的设计问题

1. 局部分配策略与全局分配策略
    1. 局部算法可以有效地为每个进程分配固定的内存片段

        例子：工作集, FIFO, LRU
    2. 全局算法在可运行进程之间动态分配页面，通常工作更好。

        例子：FIFO, LRU

        1. 为进程分配相等的份额：不好，对大进程不公平，应该给每个进程分配最小的页框数

        2. PFF：监测缺页率，保证每个进程的 PFF 在可控范围内。
2. 负载控制：换成进程时考虑其特性（IO 密集或者 CPU 密集）
3. 页面大小

    小页面：更多页面更大页表，更多缺页中断，内部碎片少，分配时效率高

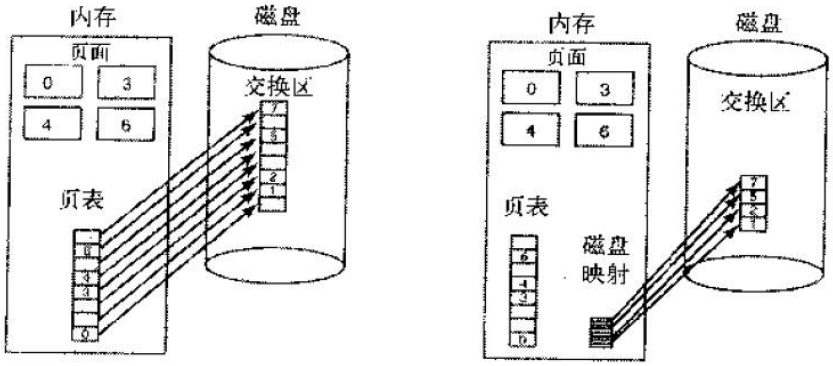
    大页面：反之

    计算公式：$P = \sqrt{2 * s * e}$ P页面，s 进程平均大小，e 每个页表项大小
4. 分离的指令空间和数据空间：独立分页
5. 共享页面
    1. 共享 I 空间页面
    2. 共享页面时，换出进程不应换出共享页面：使用专门的数据结构记录共享页面
    3. 共享数据：写时复制
6. 共享库
    1. windows：DLL 文件，连接器没有加载函数，而是一小段可以在运行时绑定函数的存根里程 stub routine
    2. 优点：共享库中一个函数被修正：不需要重新编译程序
    3. 问题：编译共享库时，不产生使用绝对地址的指令
7. 内存映射文件：进程发起一个系统调用，把一个文件映射到虚拟地址的一部分。

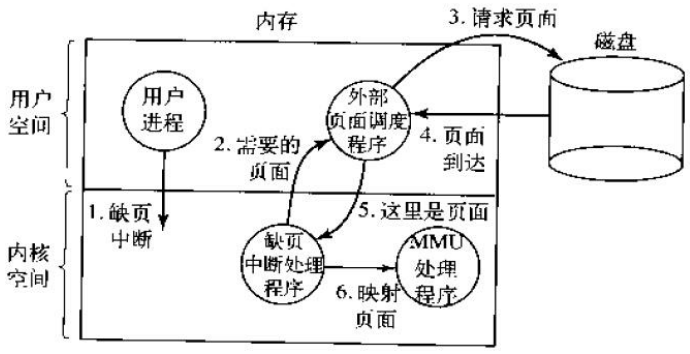
    映射共享页面时不会实际读入页面，而在其访问页面时才会每次一页的读入。
8. 清除策略
    1. 分页守护进程：保证有足够空闲页框供给
    2. 实现方法：双指针时钟，前指针由分页守护进程控制
9. 虚拟内存接口：允许程序员控制内存映射
    1. 可以允许两个或多个进程共享一部分内存
    2. 实现高性能的消息传递系统：发送进程清除映射，接收进程建立映射

3.6 分页系统的实现

1. 上下文切换：重置 MMU，刷新 TLB，装载页表
2. 缺页中断处理
    1. 陷入内核，保存程序计数器
    2. 调用汇编例程保存其他寄存器
    3. 操作系统查看寄存器或者分析指令判断所缺页面
    4. 操作系统检查虚拟地址是否有效，若是则启动页面置换算法更换页面
    5. 如果页面被修改过，启动 IO，标记该页面，上下文切换
    6. 启动 IO 装载缺失页面
    7. 页表更新，恢复堆栈和寄存器，重启被中断进程
3. 指令备份：使用隐藏的内部寄存器，在每条指令执行之前，把程序计数器的内容复制过来
4. 锁定内存页面：锁住正在进行 IO 的页面或者在内核缓冲区完成所有 IO
5. 后备存储
    1. UNIX：swap 交换区：没有文件系统
    2. 进程启动前必须初始化交换区
    3. 实现方法
        1. 将整个进程复制到交换区：进程可能增长
        2. 直到换出时才分配交换区：内存中每个页面都要记录磁盘地址



    3. 交换分区：windows 使用大文件
    4. 空间不足时抛弃程序正文或者共享库
6. 策略与机制分离：更多的模块化代码和更好的适应性，更多的陷入内核
    1. 三部分：底层 MMU 处理程序，内核的缺页中断处理程序，用户空间的外部页面调度程序



    2. 问题：外部调度程序无权访问页面的 R 位和 M 位
        1. 把相应数据映射到或者传递给外部调度程序

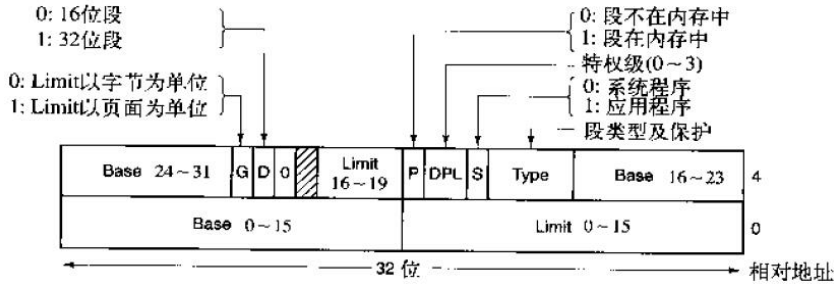
2. 把页面置换算法放在内核

3.7 分段：在机器上提供多个独立的地址空间，长度从 0 到某个整数，可以动态改变

1. 段是逻辑实体，不会同时包含不同类型的内容，不是定长的
2. 1 维地址中，修改过程会影响其他无关地址的起始地址
3. 比较

考查点	分页	分段
需要程序员了解正在使用这种技术吗?	否	是
存在多少线性地址空间?	1	许多
整个地址空间可以超出物理存储器的大小吗?	是	是
过程和数据可以被区分并分别被保护吗?	否	是
其大小浮动的表可以很容易提供吗?	否	是
用户间过程的共享方便吗?	否	是
为什么发明这种技术?	为了得到大的线性地址空间而不必购买更大的物理存储器	为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护

4. 纯分段的实现：可能会有碎片，需要内存紧缩
5. 分页与分段结合
    1. 虚拟地址分为段号和段内地址，其中段内地址里有页号和页内偏移量
    2. 先找段---找页面---找虚拟地址
    3. 实例：奔腾处理器
        1. 虚拟内存的核心：局部描述符表 LDT 和全局描述符表 GDT



        2. 利用选择子可以同时实现纯分页/分段和混合模式
        3. 保护环：越级调用

第四章 文件系统

4.1 文件

1. 拓展名：UNIX 任意，WINDOWS 注册
2. 二进制文件：魔数，放在二进制文件头
3. 文件存取：顺序存取（磁带），随机存取（硬盘，seek）

4. 文件操作：open：把文件属性和磁盘地址装进内存，方便后续的快速调用（fd）

4.2 目录

1. 软连接：目录的 data block 加一条记录，新生成一个文件（i 节点，data block 里是被链接目录的路径），可以链接目录，可以跨文件系统（转储时需要注意）
2. 硬链接：目录的 data block 加一条记录，指向现有文件 i 节点。只能链接文件，不能跨文件系统

4.3 文件系统的实现

1. 磁盘 0 号扇区：MBR，引导计算机启动·，之后分区表，之后 superblock
2. 文件的实现
    1. 连续分配：CD-ROM，DVD
        优势：实现简单，记录第一块的磁盘地址和块数即可；读操作性能好
        劣势：磁盘上会出现大量碎片；创建文件时必须知道文件大小
    2. 链表分配：每个块的第一个字作为下一块的指针
        优势：没有内部碎片
        劣势：随机读取非常慢，必须先读之前的 n-1 块；块的大小不再是 2 的整数次幂，降低系统运行效率
    3. 内存中采用表的链表分配（FAT，每个磁盘的指针字放在内存里）
        优势：加速随机存储
        劣势：必须把整个表放在内存里，对大硬盘不合适
    4. i 节点（UNIX）：只有对应文件打开的时候，i 节点才在内存中
        优势：占用内存较少
        劣势：需要磁盘块个数会超过 i 节点大小：i 节点可以指向更多的 i 节点
3. 目录的实现
    1. UNIX:文件属性在 i 节点里。Windows：文件属性在目录里
    2. 长文件名
        1. 长度限制：浪费大量目录空间
        2. 目录项固定部分：长度，数据项等。下一个进来的文件不一定符合空隙
        一个目录项可能分布在多个页面上，读取时会有页面故障
        3. 目录项固定长度，文件名在目录后的堆中
            优点：移走文件后，总有新文件可以加进来；文件名不需要从字边界开始
            缺点：管理堆+页面故障仍然存在
    3. 查找文件名：线性查找或者使用散列表
        1. 散列表：查找迅速，管理复杂（可以加入高速缓存）
4. 共享文件
5. 日志结构文件系统 LFS
    1. 基础思想：将整个磁盘化成一个日志，每隔一段时间，缓存在内存中所有未进行的写操作都被放到一个独立的段中，并写到日志末尾（难以找到 I 节点）
    2. 清理线程：定期扫描日志进行磁盘压缩
6. 日志文件系统：NTFS，ext3
    1. 基本思想：保存记录文件系统下一步要做什么，防止崩溃导致的不一致
    2. 基本实现：日志文件系统先写一个日志，列出要进行的动作。然后进行操作，成功之后擦出日志。若系统崩溃，日志系统会重新开始行动。
    3. 写入日志的操作必须是幂等的：只要有必要，可以重复多次，不会带来破坏
    4. 原子事务

7. 虚拟文件系统：建立初衷 NFS
    1. 基本思想：抽象出所有文件系统都共有的部分，并且将这部分代码放在单独一层，该层调用底层的实际文件系统来具体管理数据
    2. 装新文件系统时，必须向 VFS 提供需要的函数地址的列表
    3. v 节点：VFS 在 fdt 中为调用进程创建一个入口，并指向一个新的 v 节点。之后 VFS 返回文件描述符。读文件时，VFS 读 v 节点，然后功能指针，最后实际文件系统的入口函数

4.4 文件系统的管理和优化

1. 磁盘空间管理
    1. 块大小
        大块：浪费大量磁盘空间（低空间利用率）
        小块：多次寻道和旋转延迟，降低性能（低数据率）
        不存在合理的平衡方案。磁盘空间越来越大，应使用大块
    2. 记录空闲块
        1. 磁盘块链表，每个块含 255 个空闲块块号和指向下一个记录块的地址
        优化：记录连续空块
        问题：指针块满或者空的时候，小型读写操作会引起大量 IO
        解决：拆分满了的指针快，保证内存里有一个半满的指针块
        2. 位图：磁盘块会比较紧密的聚集在一起，可以调进内存
    3. 磁盘配额：配额表
2. 文件系统备份
    1. 问题：
        1. 备份整个文件系统还是只备份部分文件
        2. 增量储备？
        3. 压缩：单个坏点可能导致解压缩失败
        4. 活动系统的文件备份
        5. 其他非技术性问题
    2. 方案
        1. 物理转储：复制磁盘块，万无一失，简单快速
            1. 未使用的磁盘块无需备份
            2. 坏块不应转储
            3. 不能增量存储，不能恢复特定文件
        2. 逻辑转储：恢复一个或几个目录
            1. 转储通向修改过文件或目录的路径上的所有目录
                1. 为了能把转储的文件和目录整体恢复到新文件系统中
                2. 可以对单个文件进行增量恢复
            2. 算法实例
                第一阶段：检查所有目录项，标记所有的目录和每一个修改过的文件
                第二阶段：扫描目录树，若某个目录里没有修改过的文件或者目录，去除它
                第三阶段，转储所有被标记的目录和文件
            3. 问题
                1. 空闲块列表需要从 0 开始构造
                2. UNIX 文件包含空洞，不应转储

3. 特殊文件不应转储

3. 文件系统一致性

1. 块的一致性

1. 方法：构造两张表，计数器都初始化为 0，第一个表的计数器跟踪该块在文件中的出现次数，第二个跟踪该块在空闲表的出现个数。若一致，则对任意一块，其必只在一张表里的计数为 1，一张为 0

2. 不一致的解决方法

1. 都是 0：加回空闲表

2. 空闲表出现多次：重新建立空闲表

3. 文件中出现多次：分配一空闲块，复制该块内容，将其插入到文件中

2. 文件的一致性

1. 方法：构造一张表，计数器都初始化为 0，计数器跟踪该文件在目录中的出现次数，对比其 i 节点个数。若文件系统一致，则统计数据也应一致

2. 不一致的解决方法：把 i 节点连接计数设置正确

4. 文件系统的性能

1. 高速缓存：减少磁盘访问速度

1. 常用管理算法：检查全部读请求，查看是否有相应块在高速缓存里

2. 散列：加快寻找

3. 与分页相比，访问不频繁，所以可以精确实现 LRU（可能导致关键块没有被及时写回磁盘）

4. Windows：高速缓存被修改的块被立即写回内存（通写高速缓存）

5. UNIX：每 30s 写回一次

2. 块提前读：需要用到之前提前写进高速缓存，提高命中率

1. 只适用于顺序存储文件（可以跟踪文件行为判定）

3. 减少磁盘臂运动

1. 块簇技术：用连续块簇做分配单位，但读取和高速缓存时仍使用块，寻道次数减半

2. 在磁盘中间储存 i 节点：平均寻道时间减半（没看懂 point 在哪里。。。。）

5. 磁盘碎片整理

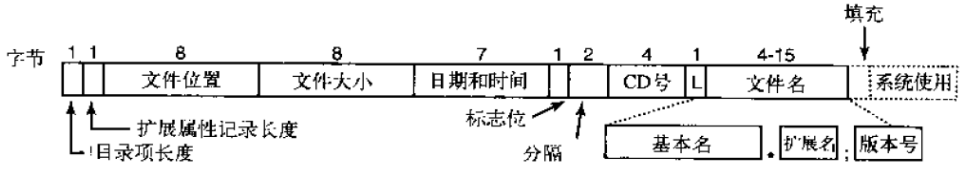
1. windows：把文件碎片整合，放在一个连续的空间以提高读取效率。

2. linux：不需要

4.5 文件系统实例

1. CD-ROM：不需要记录空闲块，不能删除

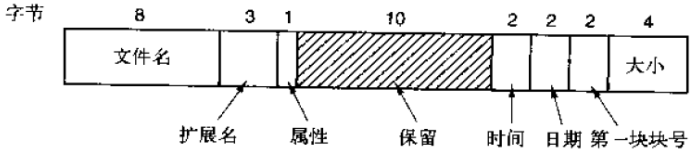
1. ISO9660：16 块前导，可放引导信息等。第 17 块存放基本卷描述符，不能使用大写字母，数字等



2. Rock Ridge 拓展(UNIX)：NM 拓展，允许无限长名字，权限拓展域等

3. Joliet 拓展(Windows)：长文件名等

2. MS-DOS 文件系统：FAT



块大小	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

1. 后面的数字代表磁盘地址位数（簇大小）

3. UNIX V7:
    1. UNIX 的 i 节点含文件大小，三个时间，所有者，所在组，保护信息，计数等信息
    2. 大文件使用多个连接块

第五章：输入/输出

5.1 I/O 硬件原理

1. IO 设备
    1. 块设备：传输一块为单位，每个块可独立读写，可寻址
    2. 字符设备：以字符为单位，不可寻址
2. 设备控制器
    1. 低层次接口，控制设备
3. 内存映射 IO
    1. IO 端口，形成 IO 空间，操作系统使用特殊指令访问

        优缺点把下面反过来

    2. 把所有控制寄存器映射到内存空间，每一个寄存器有唯一内存地址，并且保证不会有内存被分配

        优点：

        1. 不需要特殊的 IO 指令来读写设备寄存器（C/C++没有实现 IN 或 OUT 的指令）
        2. 不需要特殊的保护机制阻止用户进程执行 IO（只需要操作系统不把映射的地址空间交给用户）
        3. 可以引用内存空间的每一条指令都可以用在控制寄存器（TEST 等）

        缺点：

        1. 硬件必须针对某个页面具有选择性禁用高速缓存的能力
        2. 所有内存和 IO 都必须检测所有的引用

        对于单独内存总线的解决方法：

        1. 全部引用发到内存，响应失败再给 IO
        2. 内存总线放置探查设备，放过潜在指向所关注 IO 设备的地质

3. 在 PCI 桥芯片中过滤地址（pentium 的设计）

3. 工作原理：CPU 把地址放置在总线上，由内存或者 IO 设备来响应

4. 直接存储器存取 DMA：独立于 CPU 访问系统总线

1. 结构：一个内存地址寄存器，一个字节计数寄存器，多个控制寄存器

2. 没有 DMA 时的运作：控制器从磁盘把数据读入内部缓冲区，计算校验和，之后给操作系统中断，让操作系统把数据读入内存

1. 内部缓冲使校验错误更早被发现

2. 一旦硬盘开始工作，读入的数据是固定速率。块被放入内部缓冲区时，DMA 启动之前不需要访问主线，简化控制器设计

3. DMA 运作

1. CPU 对 DMA 控制器进行编程

2. DMA 在总线上向磁盘发起读请求

3. 磁盘控制器把数据写到内存

4. 磁盘控制器发控制信号给 DMA 控制器，DMA 决定是否继续读

5. 读结束，DMA 发中断给 CPU，CPU 从内存读数据

4. DMA 可以一次处理多路信号

5. 周期窃取：DMA 从 CPU 处偷走一个总线周期以读取一个字

6. 突发模式：DMA 发起一连串传送，阻塞 CPU 和其他设备

优点：效率高

缺点：可能阻塞很久

7. DMA 可以把数据复制到自己的缓冲区，再复制到内存

优点：更加灵活，可以实现内存到内存

缺点：需要两个总线周期

8. 弊端：DMA 比 CPU 慢，可能降低效率

5. 中断

1. 中断控制器置起信号，并在地址线上放置数字（指向中断向量），之后重开中断控制器并保存程序计数器和其他寄存器

1. 保存在内部寄存器：中断控制器之后无法获得应答，直到所有信息被读出可能丢失中断或数据

2. 保存在堆栈（大部分的做法）

1. 用户堆栈：堆栈指针不合法，缺页错误等（此时无法进行缺页中断）

2. 系统堆栈：涉及到 MMU,TLB 等，浪费时间

2. 中断向量：新的中断表格的索引，开始中断服务例程

3. 精确中断与不精确中断

1. 原因：现代 CPU 的流水线和并行

2. 精确中断：中断时将机器留在一个明确状态

1. 程序计数器保存在已知位置

2. 之前的所有指令都执行完毕，之后的指令都没有开始

3. PC 所指向的指令状态已知

3. 不能满足要求的是不精确中断：需要记录大量内部状态，中断响应和恢复很慢

4. 解决方法

1. 某些类型的中断是精确的，或者用户可以强制精确

缺点：日志，影子副本等操作降低性能

2. 使用精确中断：牺牲芯片性能

5.2 IO 软件性能

1. IO 软件目标
    1. 设备独立性：命令执行不需要指定设备
    2. 统一命名：所有文件都是用路径名寻址
    3. 错误处理：尽可能在硬件层面解决
    4. 同步阻塞驱动与异步中断驱动：操作系统挂起 IO 程序使物理异步 IO 变成同步
    5. 缓冲：大量复制，影响性能
    6. 共享设备与独立设备：死锁
2. 程序控制 IO：操作系统轮询 IO 设备，忙等待（嵌入式系统中较好）
    优点：简单
    缺点：低效
3. 中断驱动 IO：等待 IO 设备时调用其他例程
    缺点：中断发生在每个 IO 操作上，浪费 CPU 时间
4. DMA 的 IO
    优点：中断只发生在每次缓冲区操作
    缺点：DMA 慢于 CPU，可能降低效率

5.3 IO 软件的层次

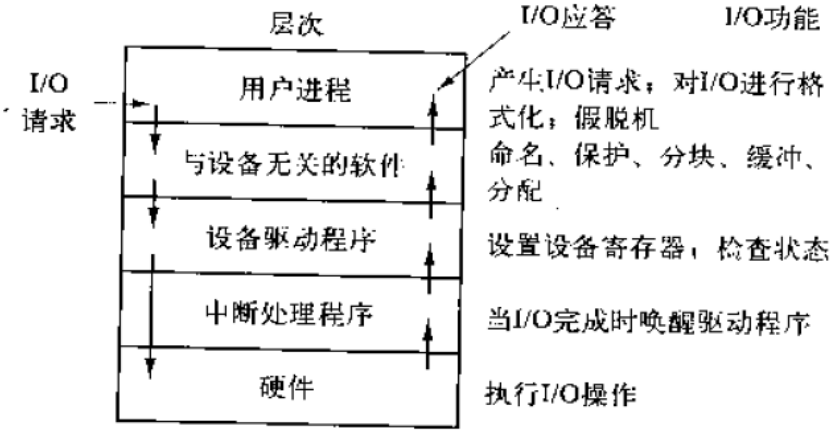
1. 中断处理程序
    1. 应该被深隐藏：将启动一个 IO 操作的驱动程序阻塞，直到 IO 完成且产生中断
        手段有：信号量上执行 down，条件变量 wait，信息上 receive
    2. 具体执行：需要设置页表，MMU，TLB 等

1) 保存没有被中断硬件保存的所有寄存器（包括PSW）。
2) 为中断服务过程设置上下文，可能包括设置TLB、MMU和页表。
3) 为中断服务过程设置堆栈。
4) 应答中断控制器，如果不存在集中的中断控制器，则再次开放中断。
5) 将寄存器从它们被保存的地方（可能是某个堆栈）复制到进程表中。
6) 运行中断服务过程，从发出中断的设备控制器的寄存器中提取信息。
7) 选择下一次运行哪个进程，如果中断导致某个被阻塞的高优先级进程变为就绪，则可能选择它运行。
8) 为下一次要运行的进程设置MMU上下文，也许还需要设置某个TLB。
9) 装入新进程的寄存器，包括其PSW。
10) 开始运行新进程。

2. 设备驱动程序：每个连接到计算机的 IO 设备都需要特定代码来驱动
    1. 通常必须是操作系统内核的一部分。（也可以不是：隔离内核与操作系统使内核不受干扰）
    2. 大多数操作系统都定义了一个所有设备都必须支持的标准接口
    3. 功能：初始化，电源管理，接受上方与设备无关的软件发出的抽象读写请求并执行之
    4. 具体实现
        1. 启动时检查输入参数
        2. 检查设备是否在用
        3. 将命令写到控制寄存器
        4. 命令发出后
            1. 驱动程序阻塞自身之后中断到来解除阻塞
            2、 无延迟，不需要阻塞

5. 操作完成后检查错误
5. 驱动程序必须是可重入的
6. 热插拔：内核重新分配资源，撤除旧资源，适当位置填入新资源

3. 与设备无关的 IO 软件
    1. 设备驱动程序的统一接口：是所有 IO 和驱动设备看起来或多或少是相同的
        1. 实现：对于每一种设备，操作系统定义一组驱动程序必须支持的函数。通过表格调用这些函数
        2. 设备保护
    2. 缓冲
        1. 在用户空间里设置一个缓冲区
            优点：效率高
            缺点：缓冲区可能被调出分页；如果钉住，可用页面池会减小
        2. 在内核空间和用户空间里都设置一个缓冲区
        内核缓冲区填满的的时候，将包含用户缓冲区的页面调入内存
            优点：效率更高
            缺点：调入内存时无法缓冲数据
        3. 在用户空间设置两个缓冲区：解决所有问题
        4. 循环缓冲区
        5. 缺点：多次复制会降低效率
    3. 错误报告：许多错误是设备特定的并且必须由适当的驱动程序来处理
        1. 编程错误：返回错误代码给调用者
        2. 实际 IO 错误：由驱动程序决定做什么
    4. 分配与释放专有设备：某些设备在任意给定的时刻只能由一个进程使用
        1. 要求进程在代表设备的特殊文件直接执行 open 操作：如果不可用，open 会失败
        2. 更复杂的方法：对请求和释放特殊设备有特殊的机制：试图得到不可用的设备可以将将被阻塞并放到一个特殊队列里
    5. 与设备无关的块大小：应该由与设备无关的软件隐藏并提供同一块大小
4. 用户空间的 IO 软件：某些独立于内核的程序
    1. 系统调用通常是由库过程实现
    2. 假脱机：多道程序处理独占 IO 设备的方法
        守护进程与假脱机目录：进程打印文件时，首先要生成整个文件，并将其放在假脱机目录下，最后由守护进程打印



5.4 盘

1. 盘的硬件

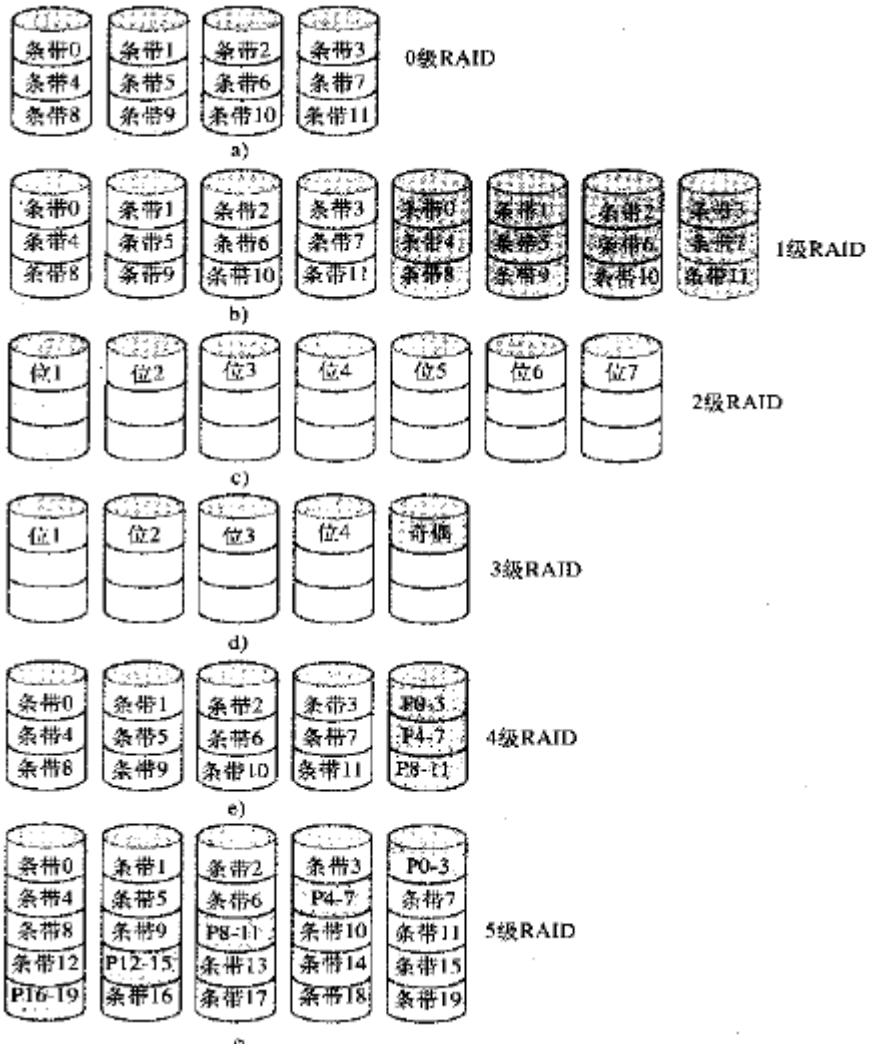
1. 磁盘：IDE 与 SATA

1. 重叠寻道：控制器能否同时控制两个或多个驱动器寻道

2. 隐藏细节：软件工作时仿佛存在 x 柱面，y 磁头和 z 扇区，由控制器将请求实际硬件

3. 逻辑块寻址：磁盘扇区从 0 开始编号，不管磁盘的几何规格

2. RAID：独立磁盘冗余阵列



1. 基本思想：将一个装满了磁盘的盒子安装到计算机，用 RAID 控制器替换磁盘控制卡，将数据复制到整个 RAID 上以获得更好的性能和可靠性（并行操作）

2. 0 级 RAID：将连续的条带以轮转方式写到全部驱动器上

RAID 控制器会把命令解析成单独的命令，以正确的顺序将命令对应四块磁盘中的一块且并行操作，最后在内存里正确装配数据

优点：性能杰出，简单明了

缺点：1. 对于每次请求一个扇区的操作系统，效率低

2. 可靠性比单块硬盘差

3. 1 级 RAID：同 0 级原理一样，但是复制所有磁盘

优点：读性能高了一倍；出色的容错性和恢复速度

缺点：写操作性能下降

4. 2 级 RAID：以字节为单位

优点：巨大的数据率

缺点：所有控制器旋转同步，单位时间内校验汉明码等

5. 3 级 RAID：2 级 RAID 的简化版，使用奇偶检验驱动器为每个数据字建立奇偶检验

优点：巨大的数据率，奇偶校验码可以用来改正错误

缺点：所有控制器旋转同步

6. 4 级 RAID：使用条带，用独立的奇偶校验驱动器

优点：损失的保护：崩溃的驱动器数据可以通过异或回复

缺点：1. 对于微小更新性能较差：必须读取旧数据并计算更新校验值

2. 奇偶驱动器负担沉重

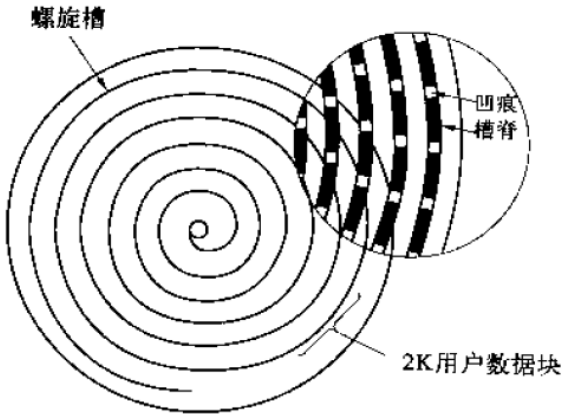
7. 5 级 RAID：将奇偶检验条带分散到每一个驱动器

优点：降低奇偶驱动器负担

缺点：回复崩溃驱动器非常复杂

3. CD-ROM：更高的记录密度

1. 使用凹陷/凸起的过渡来记录 1，平缓记录 0。数据写在连续螺旋里



2. 黄皮书：CD-ROM 的标准

主要是数据格式化与纠错能力

3. 寻道：CD-ROM 的寻道非常困难：软件计算一个近似位置，激光头在四周寻找前导区

4. 绿皮书：补充了图形以及在相同扇区保存交错音频视频和数据的能力

对多媒体 CD-ROM 非常重要

4. CD-R：可刻录 CD

1. 激光光束遇到染料时破坏其化学键

2. 橘皮书：允许 CD-R 逐渐增长式写入

3. 防止盗版复制

1. 将 CD-R 上所有文件的长度记录为几 G 字节

2. 在挑选出来的扇区故意使用错误的 ECC，期望 CD 复制程序会改正错误

3. 使用非标准间隙

5. CD-RW：可重写 CD：三种激光使合金处于三种形态

6. DVD：数字通用光盘

1. 与 CD 的差别：容量提高 7 倍
    1. 更小的凹痕
    2. 更密的螺旋
    3. 红色激光（需要第二个激光器才能同时读取 CD 和 DVD）
2. 下一代 DVD：HD DVD Vs Blu-ray DVD（蓝光 DVD 获胜，2009）

2. 磁盘格式化
    1. 硬盘由一叠铝，合金或者玻璃组成
    2. 低级格式化
        1. 扇区由前导码，数据和 ECC 组成
        2. 前导码
            1. 以一定的位模式开始
            2. 包含柱面，扇区号和其他信息
        3. 柱面斜进 cylinder skew：改善性能，一次读取多个磁道
            取决于驱动器的几何规模
        4. 结果：磁盘容量减小，取决于前导码，扇区间隙，ECC 大小和备用扇区数量
        5. 单交错：为了给控制器时间以便完成校验及将数据从缓冲区复制到内存
        5. 双交错：单交错的进化
    3. 高级格式化：对分区执行，设置一个前导块，空闲存储管理，根目录和空文件系统。还要将一个代码放置在分区表项以指示文件系统。

3. 磁盘臂调度算法：默认实际硬盘的虚拟几何规格和实际相同
    1. 读写时间影响因素
        1. 寻道时间（主导地位）
        2. 旋转延迟
        3. 实际数据传输时间
    2. 寻道时间优化算法
        1. 最短寻道优先 SSF：处理与磁头最近的请求
            优点：效率高
            缺点：两边的区域请求得到的服务较差，不公平
        2. 电梯算法：磁盘臂向一个方向移动并处理沿途请求，到达尽头后转向
            优点：公平，且对于任意一组请求，移动长度的上界都是柱面数的两倍
        3. 电梯算法改：处理完最高编号的请求后磁盘臂移动到最低编号的请求并继续向上
        4. 如果软件可以检查磁头下方的扇区号：驱动程序发出请求读写下一次要通过磁头的扇区
    3. 旋转时间优化算法：未完成的请求用扇区号排序
    4. 多驱动器优化算法：为每个驱动器维护一个请求表，空闲下来的驱动器立刻发寻道请求，当传输结束时检查是否有驱动器在正确柱面上
        1. 在：开始读写
        2. 不在：发出新的寻道命令
    4. 整体优化：一次读出多个扇区并缓存
        硬盘缓冲区：独立于操作系统，通常保存没有实际请求的块

4. 错误处理
    1. 坏块处理
        1. 控制器处理

1. 坏块映射到新块
2. 坏块映射到下一块，所有之后的数据块向后顺移
优点：一次读写可以读出整个扇区
缺点：扇区包含数据时开销较大：重写前导码和所有数据

2. 操作系统处理：必须有坏扇区列表，或者自己测试硬盘
1. 创建一个包含坏扇区列表的文件
2. 对应用软件隐藏坏块

2. 机械臂故障：驱动程序发出重校准命令，让磁盘臂向最外面移动
其他：控制器在芯片置起一个引脚，使自身当前动作清除并复位
1. 需要均匀位留时不能校准：AV 盘，不会重新校准

5. 稳定存储器：得到写命令时，磁盘要么正确写数据，要么什么都不写，保持完整
1. 模型假设
1. 在磁盘写一个块时，操作要么正确，要么错误，错误可以在随后的读操作里通过检查 ECC 发现
2. 一个被正确写入的扇区可能会自发变坏
3. CPU 可能出故障，导致写操作崩溃

2. 实现：使用一对完全相同的硬盘，对应的块一起工作
1. 稳定写：先写在驱动器 1 上，然后读回检测正确（失败 N 次后使用备用块）。再在驱动器 2 上写，直到都成功
2. 稳定读：先在驱动器 1 上读，如果 ECC 错误 n 次，失败则读驱动器 2
3. 崩溃恢复：程序扫描两个硬盘对应的快
1. ECC 都正确，内容一样：什么都不做
2. ECC 都正确，内容不同：用驱动器 1 的覆盖 2 的
3. ECC 有一个是错误的：用正确的块覆盖错误的块

3. 优化：在稳定写期间跟踪被写的块
1. 用 CMOS 非易失性存储器存储块号
2. 把块号使用稳定写保存到一个特殊区域

5.5 时钟：维护时间，并防止进程垄断 CPU
1. 时钟硬件：有晶体振荡器、计数器和存储寄存器组成
1. 操作模式
1. 一次完成：将存储寄存器的值复制到计数器，倒数到 0 后发出中断
2. 方波模式：中断后存储寄存器的值自动复制，循环进行

2. 优点：软件控制频率
3. 备用电路防止丢失时间（UTC 协调世界时）

2. 时钟驱动程序
1. 任务列表
1. 维护日时间
1. 使用 64 位计数器
2. 使用秒维护的日时间
3. 对时钟滴答计数，相对于系统开启时间，实际时间=日时间+计数

2. 防止进程超时运行：时钟将从时间片数值递减，到 0 后调度其他进程
3. 给 CPU 运行时间记账
1. 进程启动时，系统启动另外一个辅助时钟计时
2. 在全局变量中维护一个指针指向当前正在运行的表项，每一个时钟滴

答让当前进程表项的计数器加 1

缺点：多次发生中断的进程尽管工作不多，仍然要付出整个滴答

4. 处理用户进程提出的 alarm 系统调用
    1. 例子：网络传包，特定时间无 ACK 则重发
    2. 实现
        1. 为每个请求设置单独的时钟
        2. 维护一张表，记录所有未完成的计时器信号时刻以及下一个信号的时刻
        3. 在链表里把所有未完成的请求按时间顺序链接
5. 为系统本身的各个部分提供监视器
    1. 例子：调用硬盘时如果硬盘没有旋转，则开启电机并设置一个时间足够长的监视定时器，以便在速度合适后中断
6. 完成概要剖析，监视和信息收集

3. 时钟软件：避免时钟中断的开销
    1. 实现：内核运行时在返回用户态之前，它都应检查软件是否到期。若到期则执行被调度的事件而无需切换内核态，因为系统已在内核态
    2. 使用场合
        1. 系统调用
        2. TLB 未命中
        3. 页面故障
        4. IO 中断
        5. CPU 变成空闲

5.6 用户界面：鼠标、键盘和显示器

1. 输入软件
    1. 键盘：包含一个微处理器，键被按下/释放时候发出中断并由驱动程序收集数据
        1. 键盘软件
            1. 非规范模式：驱动程序接受输入并原封不动地向上层传递
            2. 规范模式：驱动程序处理所有行内编辑，只把矫正后的行传给上级
        2. 回显：在显示器上显示输入的字符
        3. 特殊处理：折行，制表符和回车（UNIX），回车换行（Windows）

字 符	POSIX名	注 释
CTRL-H	ERASE	退格一个字符
CTRL-U	KILL	擦除正在键入的整行
CTRL-V	LNEXT	按字面意义解释下一个字符
CTRL-S	STOP	停止输出
CTRL-Q	START	开始输出
DEL	INTR	中断进程（SIGINT）
CTRL-\	QUIT	强制核心转储（SIGQUIT）
CTRL-D	EOF	文件结尾
CTRL-M	CR	回车（不可修改的）
CTRL-J	NL	换行（不可修改的）

    2. 鼠标
        1. 橡皮球鼠标：通过橡皮球滚动定位
        2. 光学鼠标

3. 鼠标步 mickey：最小的移动单位

2. 输出软件：

1. 文本窗口

转义序列：移动光标并在光标处插入/删除字符的命令集

2. X 窗口系统：非常好的移植性、灵活性和拓展性，运行在用户空间

1. X 客户：运行程序，收发命令
2. X 服务器：收集键盘鼠标数据并将输出写在键盘上（必须运行在本机）
3. X 是一个窗口系统而不是 GUI：编写窗口需要 Xlib
4. 窗口管理不是 X 系统的一部分
5. 高度事件驱动
6. 资源：保存一定信息的数据结构，可共享，生命周期短。

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                          /* 服务器标识符 */
    Window win;                            /* 窗口标识符 */
    GC gc;                                 /* 图形上下文标识符 */
    XEvent event;                          /* 用于存储一个事件 */
    int running = 1;

    disp = XOpenDisplay("display_name');   /* 连接到X服务器 */
    win = XCreateSimpleWindow(disp, ... ); /* 为新窗口分配内存 */
    XSetStandardProperties(disp, ...);     /* 向窗口管理器宣布窗口 */
    gc = XCreateGC(disp, win, 0, 0);       /* 创建图形上下文 */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);                 /* 显示窗口，发送Expose事件 */

    while (running) {
        XNextEvent(disp, &event);          /* 获得下一个事件 */
        switch (event.type) {
            case Expose:      ...;  break; /* 重绘窗口 */
            case ButtonPress: ...;  break; /* 处理鼠标点击 */
            case Keypress:    ...;  break; /* 处理键盘输入 */
        }
    }

    XFreeGC(disp, gc);                     /* 释放图形上下文 */
    XDestroyWindow(disp, win);             /* 回收窗口的内存空间 */
    XCloseDisplay(disp);                   /* 拆卸网络连接 */
}
```

3. GUI：WIMP 窗口、图表、菜单和指向设备

1. 输出送往特殊的电路板：图形适配器
2. 面向消息：键盘或鼠标的输入被系统捕获并转化成消息，送到正在被访问的窗口所属于的程序
3. 两种调用程序的方法
    1. 发送消息到窗口
    2. 投递消息到消息队列
4. 总结：windows 程序创建窗口，每个窗口有一个类对象，与每个程序相关联

的是一个消息队列和一组处理过程，最终程序的行为由到来的事件驱动

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass;              /* 本窗口的类对象 */
    MSG msg;                        /* 进入的消息存放在这里 */
    HWND hwnd;                      /* 窗口对象的句柄（指针） */

    /* 初始化wndclass*/
    wndclass.lpfnWndProc = WndProc;          /* 指示调用哪个过程 */
    wndclass.lpszClassName = "Program name";  /* 标题条的文本 */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);  /* 装载程序图标 */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);    /* 装载鼠标光标 */

    RegisterClass(&wndclass);       /* 向Windows注册wndclass */
    hwnd = CreateWindow ( ... )     /* 为窗口分配存储 */
    ShowWindow(hwnd, iCmdShow); /* 在屏幕上显示窗口 */
    UpdateWindow(hwnd);             /* 指示窗口绘制自身 */

    while (GetMessage(&msg, NULL, 0, 0)) {   /* 从队列中获取消息 */
        TranslateMessage(&msg);   /* 转换消息 */
        DispatchMessage(&msg);    /* 将msg发送给适当的过程 */
    }
    return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
   /* 这里是声明 */

    switch (message) {
        case WM_CREATE:    ... ;  return ... ;   /* 创建窗口 */
        case WM_PAINT:     ... ;  return ... ;   /* 重绘窗口的内容 */
        case WM_DESTROY:   ... ;  return ... ;   /* 销毁窗口 */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam));/* 默认 */
}
```

4. 位图
    1. windows 元文件：聚集一组对 GDI 过程的调用，描述一个复杂的图画
    适用于 windows 程序之间传输图画
    2. DIB：解决位图不能跨设备缩放等问题
5. 字体：TrueType 字体
    1. 不是位图而是轮廓，每个点都是相当于原点，所以容易进行缩放：只需要每个坐标乘以比例因子
    2. 栅格化：以任何的期望率把字体转化成位图

5.7 瘦客户机：高性能的云端运算

1. 分布式机器的不便之处
    1. 必须维护大容量的硬盘和复杂的软件
    2. 定期备份
    3. 不容易资源共享
2. 基本思想：从客户端剥离一切智能和软件，只将其作为显示器
3. 五条命令
    1. Copy：显示器从视频 RAM 的一个部分移动数据到另一个部分
    2. sfill：单一像素值填充一个区域

3. pfill：复制一个模式到某区域
4. bitmap：由前景色和后景色的区域

5.8 电源管理：关闭不用的组件或者是应用程序减少耗能

1. 硬件问题：将硬件设置成多种状态——工作，睡眠，休眠和关闭，每一个状态都比前一个耗能少，但是唤醒时间和耗能多
2. 操作系统问题
    1. 显示器：背光照明
        解决方法：显示器有若干区域组成，能够独立开启关闭
    2. 硬盘：维持高速旋转
        1. 一段时间不用后关闭：重新启动硬盘耗能较多
        2. 在 RAM 里维护大容量的高速缓存
        3. 操作系统发送硬盘状态消息给程序，使它们自由决定写操作时间
    3. CPU
        1. 降低电压：慢速运行更有效率
    4. 内存
        1. 刷新或者关闭高速缓存
        2. 将主存写进磁盘，然后关闭主存：较长加载时间
    5. 无线通信
        解决方法：计算机关闭无线设备时发消息给基站，让基站缓存信息
    6. 热量管理：风扇
        解决方法：操作系统监视温度，达到阈值后启动风扇
    7. 电池管理
        解决方法：操作系统持续监控电池，并改变其工作参数
    8. 应用程序接口
        解决方法：Windows 使用 ACPI 高级电源管理接口给驱动程序发送命令，减少他们的能耗
3. 应用程序：退化用户体验
    例子：彩色视频使用黑白等

第六章 死锁

6.1 资源：需要排他使用的对象

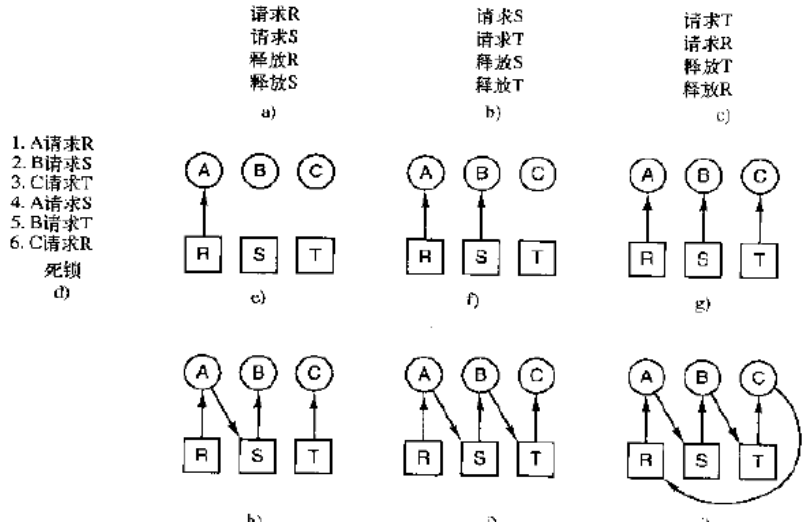
1. 分类
    1. 可抢占资源：可以从拥有它的进程中抢占而不会产生任何副作用：存储器
    2. 不可抢占资源：不引起计算错误的前提下不能抢占它
2. 假设：如果某个进程请求资源，它将休眠
3. 用户管理资源的方法：为每个资源配置一个信号量：使用 down 获取资源，使用资源，用 up 释放

6.2 死锁概述

1. 规范定义：一个进程集合中的每个进程都在等待只能有该进程集合中的其他进程才能引发的事件，该进程集合就是死锁的
2. 资源死锁：等待的时间是释放其他进程所占有的资源
3. 死锁的四个必要条件：死锁时四个条件一定满足
    1. 互斥条件：每个资源要么已经被分配，要么可用

2. 占有和等待条件：已经得到资源的进程可以再申请其它资源
3. 不可抢占条件：已经分配的资源只能被显式释放而不能被抢占
4. 环路等待条件：死锁发生时一定有进程环路出现，其中每个进程都在等待下一个进程所占有的资源

4. 死锁建模：有向图表示资源被占用/请求资源，环路表示死锁



5. 处理死锁的算法
    1. 鸵鸟算法：忽略死锁问题

    如果解决死锁的代价超过了收益，忽略它是明智的选择
    2. 死锁检测和恢复
        1. 每种类型一个资源的死锁检测：只需要一个有向图里寻找环路的算法
        2. 每种类型多个资源的死锁检测：

如果有多种相同的资源存在，就需要采用另一种方法来检测死锁。现在我们提供一种基于矩阵的算法来检测从P_1到P_n这n个进程中的死锁。假设资源的类型数为m，E_1代表资源类型1，E_2代表资源类型2，E_i代表资源类型$i(1 \leqslant i \leqslant m)$。$E$是现有资源向量（existing resource vector），代表每种已存在的资源总数。比如，如果资源类型1代表磁带机，那么E_1=2就表示系统有两台磁带机。

在任意时刻，某些资源已被分配所以不可用。假设A是可用资源向量（available resource vector），那么A_i表示当前可供使用的资源数（即没有被分配的资源）。如果仅有的两台磁带机都已经分配出去了，那么A_1的值为0。

现在我们需要两个数组：C代表当前分配矩阵（current allocation matrix），R代表请求矩阵（request matrix）。C的第i行代表P_i当前所持有的每一种类型资源的资源数。所以，C_{ij}代表进程i所持有的资源j的数量。同理，R_{ij}代表P_i所需要的资源j的数量。这四种数据结构如图6-6所示。

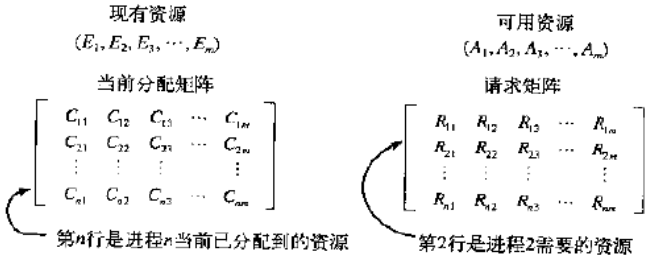


图6-6 死锁检测算法所需的四种数据结构

这四种数据结构之间有一个重要的恒等式。具体地说，某种资源要么已分配要么可用。这个结论意味着：

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

---

1. 算法描述

1. 寻找一个没有标记的进程 P（标记后表示能执行，不会死锁），且 R 矩阵该行的向量小于 A：寻找有资源请求且可被当前资源满足的进程
2. 找到：把 C 矩阵的 i 行向量加到 A 中，标记该进程，跳到 1
3. 找不到：结束

没有标记过的进程都是死锁的

2. 检查时间
    1. 有资源请求时

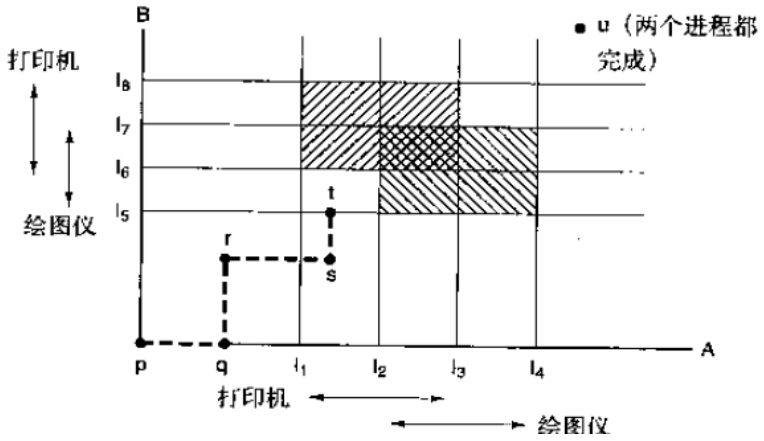
        优点：发现早

        缺点：占用 CPU 时间
    2. 每隔 K 分钟检查一次
    3. 当 CPU 使用率低于阈值的时候检查：死锁时进程无法运行

3. 从死锁中恢复
    1. 抢占恢复：临时转移资源，取决于进程的特性
    2. 回滚恢复：使用检查点记录进程状态，死锁时回滚到最近的检查点，并把资源分配给一个死锁进程
    3. 杀死进程
        1. 杀死集合中的进程，直到死锁消失
        2. 杀死集合外的进程，把其持有的资源送给死锁集合

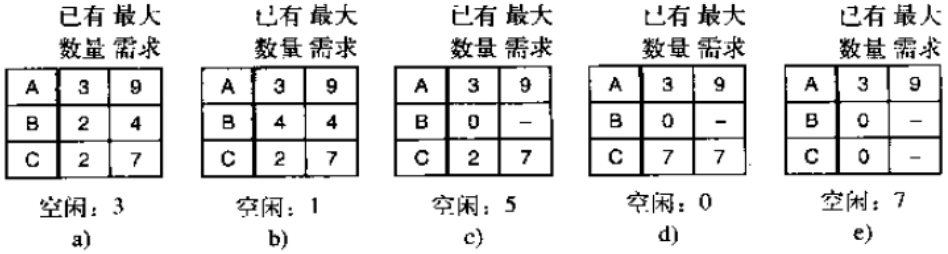
            注意：有可能的话最好杀死可以重头再来并且没有副作用的进程

3. 死锁避免
    1. 资源轨迹图



    2. 安全状态与不安全状态

        1. 安全：即使所有当前进程都突然请求最大需求资源，仍然存在某种调度算法使得每一个进程都能运行完毕。



2. 不安全：不能保证所有的进程都能运行完，但不是死锁

3. 单个资源的银行家算法

对每一个请求进行检查，看满足它是否会达到安全状态。是，则满足；不是则推迟满足这一请求

4. 多个资源的银行家算法

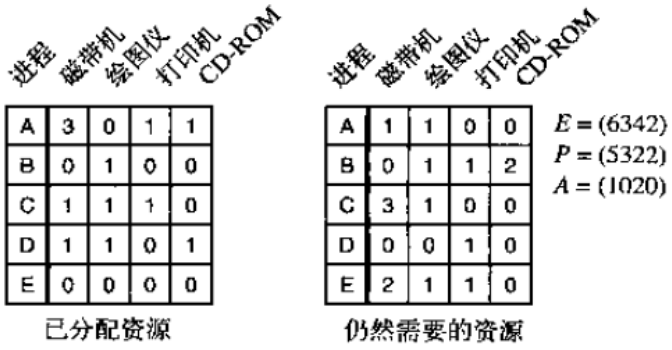


图6-12 多个资源的银行家算法

1. 检查右边矩阵中是否有一行，其没有被满足的资源数小于或等于 A
2. 存在：满足这一行的请求，标记进程，资源加到向量 A 上
3. 不存在：要么所有进程被标记，要么死锁

5. 评价：该算法虽然很有意义但是缺乏使用价值，因为很少有进程在运行前就知道所需资源最大值，而且进程数不固定，可用的资源也会变得不可用

4. 破坏死锁的条件

1. 破坏互斥条件。如果资源不被一个进程独占，则不会死锁：假脱机
2. 破坏占有和等待条件：禁止已持有资源的进程在等待其他进程
    1. 规定所有进程在开始执行时请求所需的全部资源

    缺点：1. 很多进程只有在运行时才知道需要的资源数量

    2. 资源利用率低，请求后的资源可能被闲置
    2. 请求新资源时，先释放老资源，再请求所需的全部资源
3. 破坏不可抢占条件：不是所有的资源都能做到虚拟化
4. 破坏环路等待条件
    1. 保证每个进程只能占有一个资源，请求新资源时必须释放旧的（不可接受）
    2. 将所有资源标号，进程可在任何时刻提出请求，但是所有请求必须按升序标号来/不能请求比当前占有资源更低编号的资源，不能改变顺序。

6.3 其他问题

1. 两阶段加锁：第一阶段，进程对所有记录加锁；第二阶段，更新记录并释放锁

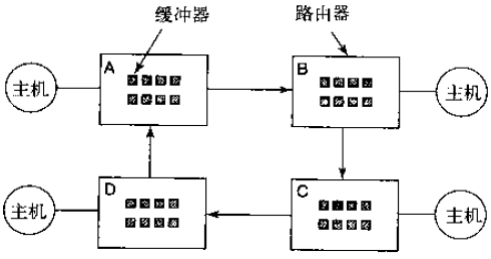
注意：对于一个进程缺少某个资源就重置它，是不可接受的

2. 通信死锁：

1. 进程 A 向 B 发出通信消息，消息丢失，两方都阻塞。

解决方法：超时重发

2. 路由器互相发送环路，没有可用的缓冲区：死锁



3. 活锁：忙等待的死锁，虽然进程在运行，但是只能空耗 CPU 时间
4. 评价：比起限制所有用户去使用资源，大多数人更愿意接受一次偶然的死锁
5. 饥饿：某些调度算法使优先级低的进程无法被服务

第七章 多媒体操作系统

7.1. 多媒体技术简介
1. 多媒体特点
    1. 极高的数据率：需要进行压缩
    2. 实时回放
2. 影响因素
    1. 颤动 jitter：传输率的变化，必须精确控制
    2. 服务质量：平均带宽，可用峰值带宽，最大和最小延迟，位丢失概率
        保证方法：预先为每一个新到的客户预留资源（进入控制算法）

7.2 多媒体文件
1. 数字电影：由一个视频文件，多个音频文件（语言），多个包含字幕的文本文件
    1. DVD：32 种语言的字幕
    2. 文件系统需要跟踪每个多媒体文件的多个子文件
        1. 用新的数据结构列出每个子文件
        2. 保持子文件同步的某种办法
2. 视频编码
    * 运动平滑性：每秒图像数。 闪烁：每秒刷新屏幕的次数
    1. 模拟视频
        1. 黑白电视：摄像机用一个电子束对图像进行横向扫描并缓慢向下移动，记录电子束经过处光的强度，扫描终点后电子束折回，称为一帧（逐行扫描）
            1. 场：针对每秒 25 帧会让人感觉闪动
                方法：不是从上到下，而先显示奇数帧，再显示偶数帧，半帧称为场
            2. 每秒 50 场无闪动：隔行扫描
        2. 彩色视频：三色光各使用一个电子束
            兼容黑白电视：RGB 信号线组合成亮度（人类更敏感）和两个色度信号
    2. 数字视频
        1. 帧的序列，每一帧由像素组成
        2. 消除闪烁：使用逐行扫描，每一帧刷新 2-3 遍
            现实应用：帧率为每秒 25 帧，但是计算机把每帧绘制两次
    3. 音频编码：奈奎斯特法则，以 2f 为频率采样

7.3 视频压缩
1. 编码和解码算法的不对称性：多媒体文件只需要编码一次，但需要解码多次
    1. 速度和硬件不对称：可以接受缓慢而昂贵的编码+快速的解码
    2. 不必须是 100%可逆：所有多媒体系统都是有损的
2. JPEG 标准：静态照片编码方式（对于 800*640 而言）
    1. 块预制：有损。用色度和亮度构造矩阵，压缩色度矩阵（人对色度不敏感），发图像分块（亮度矩阵 4800 块，色度矩阵 1200 块）
    2. 对所有矩阵使用 DCT 离散余线变化：有损，超越数和浮点数的舍入
    3. 量化：有损。去除不重要的 DCT 参数

4. 将每一块的左上角原点值以它与前一块中相应元素相插的量减小
5. 将矩阵元素线性化并对得到的列表进行行程长度编码
6. 使用 huffman 编码对列表中的数字编码以进行传输

总结：解码时间与编码基本等同

3. MEPG 标准：动态图像编码
    1. 空间冗余：每帧使用 JPEG 编码
    2. 时间冗余：互相连续的帧几乎都是相同的
    3. MEPG 实现
        1. 原理：I 帧，B 帧和 P 帧
        2. I 帧：使用 JPEG 编码的静态图像，全分辨率的亮度和半分辨率的色度，每秒 1 个或者两个 i 帧

            周期性出现的重要性：
            1. 观众收看是随机的，如果所有的帧都依赖第一帧，那么错过第一帧就不能再对视频解码
            2. 任何一帧接受错误都会导致视频无法再解码
            3. 快进或倒带时解码器不得不计算每一帧
        3. P 帧：基于宏块思想，对帧间差进行编码。搜索上一帧中的宏块进行编码
            1. 宏块：亮度 16*16，色度 8*8 的矩阵
            2. MPEG 没有规定如何搜索，搜索多远和计算匹配好坏

            实现方式：在前一帧的当前位置及所有在 x 方向偏移 m，y 方向偏移 n 的位置搜索宏块，计算最高得分。
            3. 找到宏块后，针对差值进行编码，再用 JPEG 编码
        4. B 帧：同 P 帧基本一致，但是同时基于过去和未来的一帧
        5. 为了对 B 帧编码，必须在内存里使用充足的缓冲

7.4 音频压缩
1. 波形压缩：信号通过傅里叶变换成频率分量，对每一个振幅用简短的二进制编码
2. 感知编码：寻找人类听觉系统的特点。结果在示波器上不相同，但是听起来没有区别
    1. 频段屏蔽：一个频段内响亮的声音遮掩另一频段中柔和的声音
    2. 暂时屏蔽：响亮声音撤去后，仍有一段时间耳朵无法接受到柔和声音
    3. 原理：人们没必要对功率在可听阈值（也就是听不到）的声音编码
    4. 应用：MP3 编码，对声音做傅里叶变换得到频率，之后传递不被屏蔽掉的频率

7.5 多媒体进程调度
1. 调度同质进程：服务器支持固定数量的电影，所有电影都有相同的帧率，视频分辨率和其他参数
    1. 实现：对每一部电影存在一个线程，其工作是每次从磁盘中读取电影的一帧并传送给用户。使用轮转调度并有定时机制，确保每一进程以恰当的频率运行
    2. 主控时钟：每秒滴答适当的次数，所有进程以相同的次序运行。

    评价：只要进程足够少，所有工作都能在一帧内完成。
2. 一般实时调度：
    1. 现实： 用户数目和帧大小不断变化，不同的电影会有不同的分辨率
    2. 模型：多个进程竞争 CPU，每个进程有自己的工作量和最终时限
    3. 进程可以抢占。传输缓冲区在很少的几个突发中被填满，最终时限到来之前有完全满的缓冲区
    4. 静态算法：预先分配固定的优先级，否则是动态算法。

3. 速率单调调度 RMS：适用于可抢占的周期性进程的经典静态实时调度算法
    1. 前提条件
        1. 每个周期性进程必须在周期内完成
        2. 没有进程依赖于其他进程
        3. 每一次进程在突发中都需要相同的 CPU 时间量
        4. 任何非周期性进程都没有最终开销
        5. 进程抢占即可发生而没有系统开销（是系统建模更容易）
    2. 实现：RMS 分配进程一个固定的优先级，优先级等于进程触发事件的频率，与进程的速率成正比。调度程序总运行优先级最高的进程，必要时可以抢占当前进程
    3. 只能工作在 CPU 利用率不太高的时候，最大利用率不少于 ln2。3 进程时利用率为 0.808，保证算法可运行
4. 最早最终实现优先调度 EDF: 动态算法
    1. 不要求进程是周期的，也不要求相同的 CPU 突发时间。
    2. 实现：只要一个进程需要 CPU 时间，就宣布它的到来和最终时限。EDF 维持一个按进程最终时限排序的进程列表，EDF 运行第一个（最终最早时限）进程。新进程就绪时，系统检查其时限是否在当前进程之前，若是则抢占当前进程。
    3. 平局解决：当前进程继续运行，不承担切换的代价
    4. 只要 CPU 利用率小于 100%，对于任意一组进程总保证可以工作

7.6 多媒体文件系统范型

1. 困境：
    1. 用户必须以精确的时间间隔进行 read 调用
    2. 服务器必须能够没有延迟的提供数据
2. 工作方式：推送式服务器
    1. 用户发起 start 系统调用，指定要读的文件和其他参数
    2. 视频服务器以固定速率发出帧
    3. 用户进程处理到来的帧
3. VCR 控制功能：暂停，快进和倒带
    1. 暂停：用户发信息给视频服务器要求停止，视频服务器记录要送出的下一帧
    注意：被占用的资源可能导致浪费
    2. 倒带/快进：每 k 帧显示一帧
        1. 压缩：每一帧大小不同，不能简单计算。音频压缩独立于视频
        2. 速度：以 10 倍速度把数据拉出磁盘。或者以 10 倍速度把数据传给用户。
        3. 解决方法：预先规划，使用特别的快进/快倒文件
            缺点：1. 需要额外的磁盘空间
                2. 快进/倒带只能针对特别的速度
                3. 切换时需要额外的算法
4. 近似视频点播：通知用户，电影只在某些特定的时刻播放。（不完全随点播开始）
    1. 另一个模型：人们在需要的时候预定电影，没有观众的数据流不会被传输
5. 具有 VCR 功能的近似视频点播
    1. 缓冲区策略：
        1. 前 T 分钟：显示后保存下来
        2. 后 T 分钟：从前一个数据流读入并保存
        3. 在新一帧被读入后，在缓冲区的终点添加一帧，起点丢弃一帧，当前的帧成为播放点（位于缓冲区中间）

2. 快进/快倒策略
    1. 缓冲区以内：直接处理
    2. 离开区间：开启私有的数据流服务用户，再用最近的数据流填充缓冲区

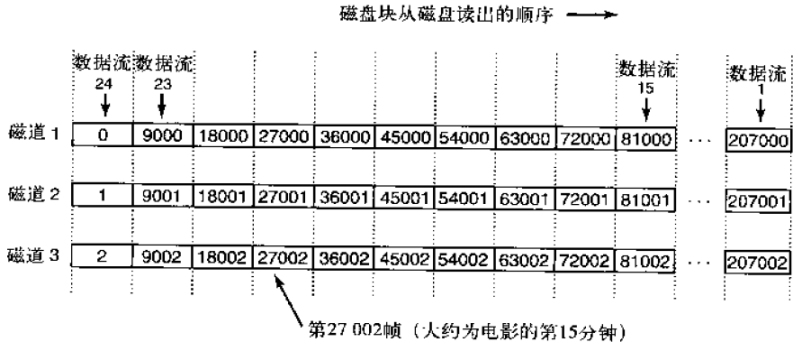
7.7 文件存放：多媒体文件大，只写一次读多次，顺序访问，回放需满足严格条件

1. 单个磁盘上存放文件
    1. 要求：数据以必要的速度流出，没有颤动
    2. 实现方法
        1. 使用连续的文件
        2. 以帧为单位存储：每帧之后是音频和字母，每次读出一帧，传送需要的

        评价：消除了所有的寻道。无法随机访问，且需要额外的缓冲区
2. 代替的文件组织策略
    1. 小块模型：恒定时间长度
        1. 基本思想：每一部电影有一个帧索引序列，指向帧开始
        2. 读操作：在帧索引中找到 k 索引，在一次磁盘操作中整体读入
    2. 大块模型：恒定数据长度
        1. 基本思想：在每一块中放入多个帧，使用块索引。每个块不一定有整数帧
        2. 实现方法：
            1. 填不满时，保持剩余部分空闲
                1. 磁盘管理复杂，需要找大小合适的连续磁盘：孔洞列表
            2. 填不满时，将帧分裂开

            评价：需要在磁盘空间和寻道之间权衡
    3. 区别：
        1. 小块模型需要大量 RAM，浪费空间小。大块模型需要少量 RAM，缺点与实现方式有关
        2. 缓冲策略：
            1. 小块使用简单的双缓冲，每个缓冲区必须能装下最大的 I 帧
            2. 大块更加复杂，需要比提供的磁盘块稍微大一点的循环缓冲区。使他达到阈值时，还有空间能够容纳另一个完整的磁盘块
        3. 磁盘性能：大磁盘块可以全速运转
        4. 快进：只有大块空闲方法可能实现只显示 I 帧的快进

        代替手段：特殊快进文件，需要能够在快进文件中定位帧
            1. 小块模式：检索帧索引
            2. 大块模式：二分搜索块
    4. 应该尽可能把所有的块和帧放在一个狭窄的区域内，加速寻道
3. 近似视频点播的文件存放
    1. 假设需要 24 个视频流：由 24 个帧组成的集合串作为一个记录写入磁盘



2. 视频服务器使用一次寻道就可以满足 24 个数据流的需要
3. 缓冲策略：双缓冲，一个向外传递数据，一个装载新数据

注意：缓冲区大小装下第二大的磁道比较明智

4. 在单个磁盘上储存多个文件
    1. Zipf 定律：

因而，前三部电影的命中率分别是$C/1$、$C/2$和$C/3$，其中C的计算要使全部项的和为1。换句话说如果有N部电影，那么

$$C/1+C/2+C/3+C/4+\cdots+C/N=1$$

    2. 管风琴算法：最流行的电影放在磁盘中央，第二和第三流行的在两边，以此类推
5. 在多个磁盘上存放文件
    1. 不用 RAID：RAID 控制器瓶颈/以性能为代价换取可靠性
    2. 磁盘园：只是数目很多的磁盘
        1. 组织简介，一块磁盘损毁不影响其他磁盘
        2. 负载不平衡，可以手工移动电影
    3. 条带磁盘园
        1. 所有电影从第一块磁盘开始：负担不均衡。
        2. 改进：交错起始磁盘/随机条带磁盘
        3. 按块分条带：系统可以发出多个块请求并被并行处理
        4. 在多少磁盘上分条带
            1. 宽条带：每部电影在所有的磁盘上分条带
            2. 窄条带：磁盘被分成小的组，每部电影限制在组内
            3. 评价：前者平衡负载做得好，后者没有单点故障的问题

7.8 高速缓存：一个块不太可能被使用两次，通常的高速缓存技术不适用

1. 块高速缓存
    1. 利用多媒体系统的可预测性
    2. 一部电影被多人观看时，可被标记为可高速缓存的。

    或者：可以使两部电影同步。
        1. 改变播放者的帧率合并播放流
        2. 包含广告
2. 文件高速缓存：使用磁盘做高速缓存
    1. 大部分视频服务器维护着一个请求最频繁的电影的高速缓存
    2. 或者可以保存每一部电影的最初几分钟，放映时从 DVD 导入剩余部分到磁盘

7.9 多媒体磁盘臂调度算法

1. 静态磁盘调度：可预测性
    1. 假设：存在 10 个用户，观看不同电影，电影分辨率，帧率和其它特性一致
    2. 磁盘臂对所有请求排序，用优化的顺序处理
    3. 增加了可以同时传送的电影书，回环的富裕时间可用来服务非实时请求
    4. 可以用更多视频流换取稀少的错过最终实现
    5. 双缓冲策略：在磁盘空间与 IO 次数上权衡
2. 动态磁盘调度
    1. 读请求需要指定磁盘块和最终时限
    2. scan-EDF 算法：把最终时限接近的请求收集分成若干批，以柱面的顺序处理
    3. 接纳新客户算法
        1. 计算平均资源，与当前剩余资源对比

2. 结合客户想要观看的资源处理

第 8 章 多处理器系统

1. 多处理器系统的局限
    1. 让时钟走得更快，但是已经到了极限
    2. 散热
2. 多处理器系统分类：归根结底是位串
    1. 共享存储器多处理器：每个 CPU 都能平等访问每个处理器
    2. 消息传递型多计算机：许多 CPU-存储器通过某种高速互联网络通信。
    3. 关于分布式系统：所有计算机系统都通过一个广域网连接
3. 多处理器：CPU 可对存储器写入某个字，再读出，并得到一个不同的字，因为被另一个 CPU 修改。（一个 CPU 向存储器写入数据而另一个读取数据）
    1. 多处理器硬件
        1. 基于总线的 UMA 统一存储器访问多处理器体系结构
            1. 基于单总线：读内存的时候先检查总线是否忙，是则等待。
               缺点：CPU 数量较多时对总线的争夺无法管理
            2. 添加高速缓存：
                1. 以字节块为基础
                2. 高速缓存块被标记成读写（不能在其他高速缓存里存在）或者只读（可以存在在其他高速缓存）
                3. 高速缓存一致性协议
            3. 高速缓存加本地私有存储器（通过私有总线访问）
                1. 编译器需要把所有程序的代码，字符串，常量等放进私有存储器
                2. 共享存储器只用于可写的共享变量
                3. 需要编译器的积极配合
        2. 使用交叉开关的 UMA 统一存储器访问多处理器体系结构