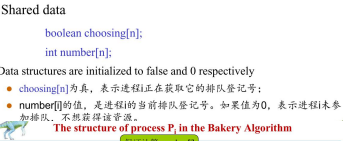


Meets all three requirements, solves the critical-section problem for two processes

Bakery Algorithm (面包房算法)

- Algorithm of solving the Critical section for n processes
 - Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
 - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
 - The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5,...
- order: (a,b) -- (ticket #, process id #)
- $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
 - max $(a_0, ..., a_{n-1})$ is a number, k such that $k \geq a_i$ for $i = 0, ..., n - 1$



```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ..., number [n - 1]) + 1;
    choosing[i] = false;
    for (j = 0; j < n; j++)
    { while (choosing[j]);
      while ((number[j] != 0) && (number[j],j) < (number[i],i));
    }
    critical section
    number[i] = 0;
    remainder section
  } while (1);
```

硬件同步

单处理机很简单：临界区禁止中断，这样就OK了。

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}

while(1){
    do {
        key = TRUE;
        while (key == TRUE)
            Swap(&lock,&key);
        critical section
        lock = false;
        while (true);
        remainder section
    }
    {
        boolean temp = *a;
        *a = "b";
        *b = temp;
    }
}
```

抽象出两个硬件实现的原子操作：赋值和交换，然后来解决临界区，testandset的共享变量是lock 初始 false，swap也是一样，但是多了局部变量key不是共享的。

硬件方法优点：进程数随意，简单，支持多个临界区；缺点：无法让权等待，可能饥饿，可能死锁。会引起忙等

信号量 semaphore

两个操作 wait(P)和 signal(V)。信号量分为计数信号量，值域不受限制，二值信号量，只能是01 所以也叫互斥锁(mutex locks)。为了等待资源进行无限循环是忙等，通过对信号量的修改增加了 block(run->wait)和 wakeup(wait->ready)来避免忙了忙等。wait(semaphore *S) { S->value--;

if (S->value < 0) { add this process to S->list; block(); } } signal(semaphore *S) { S->value++; if (S->value <= 0) { remove a process P from S->list; wakeup(P); } }具有忙等的信号量值非负，但是这种实现可以为负，负数的绝对值代表等待该信号量的进程数，0代表无资源可用。Wait和 signal 成对出现，互斥操作就在同一进程出现，同步操作在不同进程。连续的 wait 顺序是需要注意的，但是连续的 signal 无所谓。同步 wait 和互斥 wait 相邻时，要先同步 wait，使用不当会死锁，实现复杂

优先级倒置(priority inversion) 当优先级较低的进程持有较高优先级进程所需的锁定时的调度问题

Bounded-Buffer Problem 有限缓冲区 生产者

消费者问题

是很多相互合作进程的抽象。算法：设置 N 个缓冲项；信号量 mutex 初始化为 1，用来保证对缓冲项访问的互斥要求；信号量 full 初始化为 0，表示满缓冲项的个数。信号量 empty 初始化为 N 表示空缓冲项的个数。生产者：do { ... produce an item in nextp ...wait(empty); wait(mutex); ...add nextp to buffer ...signal(mutex); signal(full);} while (1); 消费者：do {wait(full); wait(mutex); ...remove an item from buffer to nextc ...signal(mutex); signal(empty); ...consume the item in nextc ...;} while (1);

Readers-Writers Problem

数据库读写的抽象。第一读写问题：允许多个读者同时读，但是只有一个写者，也就是没有读者会因为写者在等待而等待其他读者的完成，写者可能饿死。第二读写问题：写者就绪后，写者就立即开始写操作，也就是说写者等待时，不允许新读者进行操作，读者可能饿死。共享数据有访问的数据、mutex 初始 1,保证更新 readcount时互斥，wrt 初始 1,为读写公用，供写者作为互斥信号量，被第一个进入临界区和最后一个离开临界区的读者使用，其他读者不适用。Readcount 初始 0，用来跟踪多少进程正在读。写进程：do {wait (wrt) ;// writing is performed signal (wrt) ;} while (TRUE); 读进程：do {wait(mutex); readcount++; if (readcount == 1) wait(wrt);signal(mutex); ...reading is performed ...wait(mutex); readcount--; if (readcount == 0) signal(wrt);signal(mutex); } while (TRUE);

Dining-Philosophers Problem 哲学家进餐

典型的同步问题，问题描述：N 个哲学家坐在圆桌，每个哲学家和邻居共享一根筷子；哲学家吃饭要用身边的两只筷子一起吃；邻居不允许同时吃饭；哲学家只会思考或者吃饭。共享数据：数据集 /一碗米饭；共享变量 chopstick[5]初始为 1；第 i 个哲学家进程：do {wait(chopstick[i]) wait(chopstick[(i+1) % 5]) ... eat ...signal(chopstick[i]); signal(chopstick[(i+1) % 5]); ... think ...} while (1);(这个解决方案可以保证没有 2 个哲学家同时使用 1 个筷子，但是很显然会导致死锁，如果 5 个哲学家同时饥饿，同时拿起左手筷子，就死锁了。

一些其他的可能解决：最多只允许 4 个哲学家坐在桌上/临界区内必须拿起两根筷子/使用非对称的解决方法：奇数先拿左手，偶数先拿右手。这些额外限制都能防止死锁。

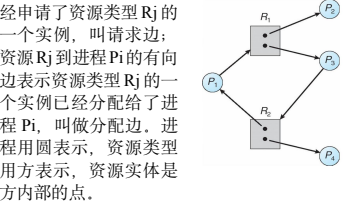
Deadlock 死锁

死锁指多个进程因竞争共享资源而造成的一种僵局，若无外力作用，这些进程都将永远不能再向前推进。进程按以下顺序使用资源：申请 使用 释放。申请和释放为系统调用。

四个必要条件：

互斥：至少有一个资源处于共享模式，一次只能有一个进程使用该资源 占有并等待(hold and wait)：一个进程必须至少占有一个资源并且等待另一个资源，且该资源被其他进程占有不可抢占：资源不能被抢占，只能在进程使用完成后释放 循环等待：进程间循环等待资源，A 等 B 占的 B 等 C 占的 C 等 A 占的。

资源分配图，由点 V 和边 E 组成，V 被分为两部分：系统活动进程的集合 系统所有资源类型的集合。进程 Pi 到资源 Rj 的有向边记为 Pi->Rj，表示进程 Pi 已经申请了资源类型 Rj 的一个实例，叫请求边；



资源 Rj 到进程 Pi 的有向边表示资源类型 Rj 的一个实例已经分配给了进程 Pi，叫做分配边。进程用圆表示，资源类型用方表示，资源实体是方内部的点。如果分配图无环->没有进程死锁，如果有环，那么可能死锁。如果每个资源恰好只有一个实例，有环则必死锁。如果环所在的资源类型是只有一个实例的，则必死锁。如果每个资源有多个实例，有环不一定死锁。P4 可能释放 R2 的实例，这个资源分配给 P3，这样就打破了死锁。

死锁处理

保证系统不进入死锁：预防 避免 prevention avoidance；允许进入死锁但是需要恢复：检测 接触 detection recovery。U L W 三个系统都忽略问题假装没有死锁，是鸵鸟方法。

死锁预防

通过限制请求的方式来预防死锁。互斥：对于非共享资源必须互斥，例如一台打印机不能被多个进程共享，因此要互斥，而共享资源不需要互斥，也不睡导致死锁，类似只读文件。

占有并等待：必须保证：一个进程申请一个资源时不能占有其他资源。进程在执行前就要申请并分配资源，是资源静态预分配的方法；缺点：低资源利用率、可能饥饿。

非抢占：如果一个进程战友资源并且已申请了另一个不能立即分配的资源，那么它现已分配的资源都可以被抢占，也就是被隐式释放了。抢占资源分配到进程所等待的资源的连表上。进程需要获取到原有的资源和申请的新资源后才能运行。循环等待：按照资源请求递增的方式分配资源，通过资源的有序申请破坏了循环等待条件。

死锁避免

前面的方法虽然避免了死锁，但是降低了吞吐率，我们可以通过获取一些额外的事先信息从而避免死锁 prior information。最简单和最有效的模型要求每个进程声明它可能需要的每种类型的资源的最大数量。死锁避免算法动态检查资源分配状态，确保永远不会出现循环等待。资源分配状态可用和已分配资源的数量以及进程的 maximum demand 需求。

安全状态：对于所有进程，如果存在一个安全序列，那么系统就处于安全状态。对于进程序

列 P1,P2,...,Pn，如果对于每一个 Pi,Pi 仍然可以以申请的资源数小于当前可用的资源加上所有进程 Pi(i-j)所占有的资源，那么这一序列是安全序列。这种情况下，进程 Pi 的资源即使不能立即可用，那么 Pi 可等待知道所有 Pj 释放其资源，当它们完成时 Pi 就可以运行，Pi 运行结束后，Pi+1 就可以获得到所需的资源，如此进行。

安全状态->没有死锁；不安全状态->可能有死锁；避免->保证系统永远不进入不安全状态。

资源分配图，单实体资源类型避免算法：

引入一种新边 claim edge 需求边，Pi->Rj 表示进程 Pi 在未来可能请求资源 Rj，用虚线表示。当进程真正请求资源时，用请求边覆盖掉需求边。当资源被分配给进程后，用 assignment edge 分配边来覆盖掉请求边，当资源被释放后，分配边恢复为需求边。系统必须事先说明需求边。算法：假设进程 Pi 申请资源 Rj，只有在需求边 Pi->Rj 变成 分配边 Rj->Pi 而不会导致资源分配图形成环时，才允许申请。用该算法循环检测，如果没有环存在，那么资源分配会使系统继续安全状态，否则就会不安全，Pi 就要等待。

银行家，多实体资源类型避免算法：

每个进程实现说明最大需求：进程请求资源时可能会等待；进程拿到资源后必须在有限时间内释放它们。

数据结构：N 进程数，m 资源类型的种类数；Available：长度为 m 的向量，表示每种资源的现有实例数量，available[j]=k 表示 j 型资源还有 k 个；Max：n*m 的矩阵，定义每个进程的最大需求，max[i][j]=k 表示进程 Pi 最多可以申请 k 个 Rj 型资源；Allocation：n*m 的矩阵，表示每个进程所分配的各种资源类型的实例数，allocation[i][j]=k 表示已经为 Pi 分配了 k 个 Rj 型实例；Need：n*m 矩阵，表示每个进程还需要的剩余的资源，need[i][j]=k 表示进程 Pi 还可以继续申请 k 个 Rj 型的实例。Need=max-allocation。

安全状态检测算法：

1. 设 work 和 finish 分别是长度为 m 和 n 的向量，初始化：work=available，finish[i]=false；2.寻找 i 满足 finish[i]=false 且 need[i]<=work，如果 i 不存在跳到第四步；3.work=work+allocation[i],finish[i]=true，返回第二步；4.如果所有的 finish 都是 true，那么系统处于安全状态。

算法需要 m*n*n 的操作数量级确定系统状态

资源请求算法：

Request[i]为 Pi 的请求向量，如果 request[i][j]=k 那么进程 Pi 需要资源类型 Rj 的数量为 k。当进程 Pi 请求资源时，动作如下：1. 如果 request[i]<=need[i]跳到第二步，否则出错，因为进程 Pi 已经超过了其最大需求。2.如果 request[i]<=available 跳到第三步，否则 Pi 必须等待，因为没有可用资源

3.假定系统可以分配给进程 Pi 请求的资源，进行下面的操作：Avaible=avaible-request[i];allocation[i]=allocati on[i]+request[i];need[i]=need[i]-request[i];如果产生的资源分配状态是安全的，那么交易完成且进程 Pi 可以分配到资源，如果新状态不安全，那么进程 Pi 必须等待 Request[i]并且恢复到原有的资源分配状态。

死锁检测

允许系统进入死锁状态的话，那么系统就需要提供检测算法和恢复算法。

等待图，单实体资源类型检测算法：

等待图是资源分配图的变形，节点都是进程，Pi->Pj 表示 Pi 在等待 Pj 释放 Pi 所需的资源。当且仅当等待图中有一个环，系统死锁，检测环的算法需要 n*n，n 为点数。

多实体资源类型检测算法：

数据结构：Available，allocation 是一样的，request：n*m 的矩阵，表示当前各进程的资源请求状况，request[i][j]=k 表示 Pi 正在请求 k 个资源 Rj。算法：1.设 work 和 finish 分别是长度为 m 和 n 的向量，初始化：work=available，如果 allocation[i]非 0，finish[i]=false 否则初始化为 true；

2.寻找 i 满足 finish[i]=false 且 request[i]<=work，如果 i 不存在跳到第四步；3.work=work+allocation[i],finish[i]=true，返回第二步；

4.如果某个 finish 是 false，那么系统处于死锁状态，且对应下标的进程 Pi 死锁。算法需要 m*n*n 的操作数量级确定系统状态

死锁检测算法的应用

检测算法的调用时刻及频率取决于：死锁发生频率以及思索发生时受影响的进程数。如果经常发生死锁，那么就要经常调用检测。如果在不确定的时间调用检测算法，资源图可能有很多环，通常不能确定哪些造成了死锁

死锁恢复

检测到死锁后的措施：通知管理员 系统自己恢复。打破死锁的两种方法：抢占资源 进程终止。

进程终止

两种方法来恢复死锁：终止所有死锁进程 一次终止一个进程直到不死锁。许多因素都影响终止进程的选择：优先级 进程已经计算了多久，还要多久完成 进程使用了哪些类型的资源 进程还需要多少资源 多少进程需要被终止 进程是交互的还是批处理的

抢占资源

抢占资源需要处理三个问题：选择一个牺牲品 victim：要代价最小化 回滚：回退到安全状态，但是很难，一般需要完全终止进程重新执行 饥饿：保证资源不会总是从同一个进程中被抢占。常见方法是为代价因素加上回滚次数。

Main Memory 主存

层次存储中主存 cache 寄存器为 volatile 易失的，逻辑地址/虚地址/相对地址：由 CPU 生成，首地址为 0，逻辑地址无法在内存中读取信息。物理地址/实地址/绝对地址：内存中存储单元的地址，可以直接寻址。

物理地址中的逻辑地址空间是通过一对基址寄存器和界限地址寄存器控制的 base and limit register。如果基址寄存器为 300040，界限寄存器为 120900，那么程序的合法访问从 300040 到 420910(含)的所有地址。

地址绑定的三种情况：

编译时间：如果编译时就知道进程在内存中的地址，那么就可以生成绝对代码 absolute code，装载时间：编译时不知道在哪，那么编译器生成可重定位代码 relocatable code。执行时间：如果进程在执行时就可以移动到另一个内存段，需要硬件支持也就是 base and limit

目前绝大多数都是采用这种。

Memory-Management Unit (MMU)

就是将虚拟地址映射到物理地址的硬件设备。在 MMU 中，base 寄存器叫做重定位寄存器。用户进程送到内存前，都要加上重定位寄存器的值，PA=relocation reg+LA，用户程序只能处理 LA，永远看不到真正的 PA。

Dynamic Loading (动态加载)

进程大小受到物理内存大小的限制，为了有更好的空间使用率，采用动态加载，一个子程序只有在调用时才被加载，所有子程序都可以重定位的形式存在磁盘上，需要的时候装入内存中。OS 不需要特别支持，是程序设计做的事。当需要大量的代码来执行一些不常发生的事时很有用，如错误处理。

Dynamic Linking (动态链接)

将链接延迟到运行时，DDL。动态链接需要一个存根 stub，它是一小段代码，用来指出如何定位库程序。

Swapping (交换技术)

进程可以暂时从内存中交换到备份存储 backing store 上，当需要再次执行时再调回。需要动态重定位 dynamic relocate 备份存储：是快速硬盘，而可以容纳所有用户的所有内存映像，并为这些内存映像提供直接访问，如 Linux 交换区 windows 的交换文件 pagefile.sys。Roll out roll in：如果有一个更高优先级的进程需要服务，内存交换出低优先级的进程以便装入和执行高优先级进程，高执行完后低再交换回内存继续执行。

交换时间中的主要部分是转移时间 transfer time。总转移时间与所交换的内存大小成正比。系统维护一个就绪的可立即运行的进程队列，并在磁盘上有内存映像。

Contiguous Allocation (连续分配)

内存通常分为两个区域：一个驻留 resident 操作系统，一个用于用户进程。由于中断向量一般位于第内存，所以 OS 也放在低内存。重定位寄存器用于保护各个用户进程以及 OS 的代码和数据不被修改。Base 是 PA 的最小值；limit 包含了 LA 的范围，每个 LA 不能超过 Limit。MMU 地址映射是动态的。Multiple-partition allocation：分区式管理将内存划分为多个连续区域叫做分区，每个分区放一个进程。有固定分区和动态分区两种。

动态分区：

动态划分内存，在程序装入内存时切出一个连续的区域 hole 分配给进程，分区大小恰好符合需要。操作系统需要维护一个表，记录哪些内存可用哪些已用。从一组可用的 hole 选择一个空闲 hole 的常用算法 first best worst-fit 三种。分别是分配第一个足够大的/分配最小的是够大的/分配最大的。First 和 best 在时间和空间利用率都比 worst 好，还有一个 next-fit 是每次都从上次查找结束的位置开始找，找到第一个足够大的。

碎片 fragmentation

first 和 best 都存在外部碎片的问题。外碎片指所有的总可用内存可以满足请求，但是并不连续。外碎片可以通过紧凑 compaction 拼接 defragmentation 减少。重定位是动态并且在执行时间完成可以进行紧凑操作 重新排列内存来将碎片拼成一个大块，但是拼接的开销很大。内碎片是进程内部无法使用的内存，这是由于零头和块大小造成的，比如块大小 8B，进程有

9B, 那么不得不给他 16B 的内存, 就出现了 7B 的内碎片。

分页存储管理

分页允许进程的 PA 空间非连续; 将物理内存分为固定大小的块, 叫做帧 frame/物理块/页框, 将逻辑内存也分为同样大小的块叫做页 page, Linux Win(x86)是 4KB,

OS 需要跟踪所有空闲帧, 叫帧表。

运行一个 n 页的程序就需要找到 n 个空帧然后装载进去。

OS 需要维护一个页表来进行 LA 到 PA 的转换, 分页技术避免了外碎片, 只有内碎片存在。

物理地址 逻辑地址 页表项数计算:

页表项数和内存大小相关 2^m * B 的内存大小

对应需要 m 项的页表, 物理地址长度为 m,

逻辑地址和页表大小及虚存空间有关: 虚存大小 2^m * B 那么逻辑地址长度为 m, 页表大小

2ⁿ * B, 则页偏移位数 n, 页号位数 m-n,

地址映射过程: 逻辑页号拼上 offset 经过页表

查到物理页号, 然后得到物理真号拼上 offset,

然后进入到内存中找 frame。

页表的实现

页表放在内存中, PTBR page-table base reg 指

向页表, 切换页表只需要改变这个寄存器就可

以 PRLR page-table length register 说明页表长

度, 这样的模式下每次数据/指令访问都需要两

次内存访问, 一次查页表一次查数据/指令。为

了加速这个过程, 引入了特殊的转换表缓冲

区 TLB, 是一种硬件 cache, 部分 TLB 维护了 ASID

addressspace identifier, 用来唯一地标识进程,

为进程提供空间保护。

Effective Access Time 有效访问时间 EAT

Associative lookup=1 查 TLB 表的时间

Memory access time=t2 内存访问时间

α TLB 命中率

那么 EAT=(t1+t2)* α + (t1+t2+t2)* α

也就是查 TLBmiss 后, 需要进内存查一次页表,

再去取一次数据, 命中就直接取数据。

保护 protection

内存保护通过与每个帧关联的保护位实现。

Valid bit 存在页表中的每一个条目上。

Shared code 共享代码: 如果代码是可重入即只

读代码 reentrant code 或者是纯代码 pure code,

可以共享, 共享代码在各个进程中的逻辑地址

空间相同。然后每个进程再花较小的空间保存

私有代码和数据即可。

分级页表 Hierarchical page table

由于现代计算机逻辑地址空间很大, 导致页表

会很大, 而且页表还要连续, 所以不现实。因

此要将页表划分变小, 简单的实现方法: 两页

分页算法, 就是将页表再分页。就是将页号部

分再划分为页偏移和页码。下面是寻址模式:



P1 是用来访问外页表的索引, p2 是外页表的

页偏移, 然后 d 是内页表的偏移。对于一个 32

位的 LA, 一般 10 位外 10 位内 12 位偏移

某计算机采用二级页表的分页存储管理方式, 按字节编址, 页大小为 2¹⁰ 字节, 页表项大小为 2 字节, 逻辑地址结构为:

页目序号	页号	页内偏移量
------	----	-------

逻辑地址空间大小为 2¹⁶ 页, 则表示整个逻辑地址空间的页目录表中包含表项的个数至少是

A. 64 B. 128 C. 256 D. 512

页大小为 2¹⁰ B, 页表项大小为 2B, 采用二级

页表, 一页可存放 2⁹ 个页表项, 逻辑地址空间

大小为 2¹⁶ 页, 要使表示整个逻辑地址空间的

页目录表中包含的个数最少, 则需要

2^{16/2}=2⁸=128 个页面保存页表项, 即页目录表

中包含的个数最少为 128。

哈希页表

超过 32 位 LA 地址空间时, 一般采用哈希页表,

将虚页号的哈希值存到哈希表里, 哈希表的每

一项都是链表, 链着哈希值相同的页号。然后

在查表时用虚页号与链表中的每个元素进行

比较从而查物理表号

反向页表

对于每个真正的内存帧才会有一个条目。每个

条目包含保存在真正内存未知的页的虚地址

及拥有该页的进程信息, 因此整个系统只有

一个叶匙, 对每个物理内存的帧也只是一条相

应的条目。拿时间换空间, 需要为页表条目中

添加一个地址空间标识符 ASID。

分段 Segmentation

分页无法避免的是用户视角的内存和物理内

存的分离。分段管理支持用户视角的内存管理

方案, LA 空间是由一组段组成的, 每个段都

有其名称和长度, 地址指定了段名称和段内偏

移。因此 LA 通过有序对

<segment-number, offset> 构成。

段表将用户定义的二维地址映射成一维, 每一

个条目包含 base 和 limit, STBR segment table

base reg 指向内存中段表的位置, STLR 一个

程序使用的段长度, 用户使用的有序对中的

segment-number 必须小于 STLR, 同样有 valid

位, 还有读写执行的权限设置, 也可以进行

code share, 内存分配是动态存储分配问题。

Virtual Memory 虚存

虚存将用户的路基存储和物理存储分开; LA

空间可以大于 PA 空间; 允许 PA 空间被多个

进程共享。

局部性原理: 时间: 指令的一次执行和下次

执行 数据的一次访问和下次访问都集中在

一个较短时期内; 空间: 当前指令和邻近的指

令 当前数据和邻近的数据都集中在一个小区域

内。虚存是是具有请求调入功能和置换功能,

能仅把进程的一部分装 入内存便可运行进

程的存储管理系统, 它能从逻辑上对内存容

量进行扩充的一种虚拟的存储系统。

按需调页 Demand Paging

指在需要时才调入相应的页的技术。采用 lazy

swapper 的方式, 除非需要页面, 否则不进行

任何页面置换。

页错误 Page fault

非法地址访问和不在主存或无效的页都会

page fault, Page fault rate 等于 1 不代表 every

page is a page fault。

更完整的页表项 请求分页中

虚拟页号 物理帧号 状态位 P(存在位 页是

EAT=(1-p)*memory access time + p*page fault time

Page fault time 包括 page fault overhead, swap

page out, swap page in, restart overhead 等

为了计算 EAT, 必须知道需要花多少时间处

理 page fault, page fault 会引起以下动作的产

生: 1.陷入 trap 到 OS

2.保存用户 reg 和进程状态

3.确定中断是否为 page fault

4.检查页引用是否合法并确定所在磁盘位置

5.从磁盘读页到内存的空闲帧(包含磁盘队

列中的等待 磁盘的寻到 旋转延迟 磁盘的传

输延迟)

6.在等待过程中的 CPU 调度

7.IO 中断

8.保存其他用户寄存器和进程状态 (如果进

行了 6)

9.确定中断是否来自磁盘

10.修正页表和其他相关表, 所需页已经在内

存中

11.等待 CPU 再次分配给本进程

12.恢复用户寄存器、进程状态和新页表, 重

新执行。

其中的三个主要 page fault 时间是缺页中

断服务时间 缺页读入时间和重启时间

写时复制 copy-on-write

COW copy on write 允许父子进程开始时共

享同一页面, 在某个进程要修改共享页时, 它

才会拷贝一份该页面进行写。

COW 加快了进程创建速度。当确定一个页

采用 COW 时, 这些空闲页在进程栈或堆

拓展时可用于分配 或用于管理 COW 页。OS

此案用按需填 0 zero fill on demand 按需填

零需要在分配前填 0.Win linux solaris 都用了

COW

页面置换

寻找一些内存中没有使用的页换出去。内存

的过度分配 over-allocation 会导致 page fault

调页后发现所有页都在使用。

使用 dirty/modity 位来减少页传输的开销,

只有脏页才需要写回硬盘。

基本页面置换过程:

1.查找所需页在磁盘上的位置。2.查找空闲

帧, 如果有直接使用; 如果没有就用置换算法

选择一个 victim, 并将 victim 的内容写回硬

盘, 改变页表和帧表。3.将所需页读入新的

空闲帧, 改变页表和帧表。4.重启用户进

页面置换算法

采用最小页错误率的置换算法。

评估方法:

针对特定的内存引用序列, 运行算法, 计算

出页错误数。引用序列叫做引用串 reference

string。注意两个事实: 给定页大小, 只

需要关心页码, 不用管完整地址; 紧跟页 p

后面为对页 p 的引用不会引起页错误。

First-In-First-Out Algorithm (FIFO, 先进先

出算法)

最简单的页面置换算法, 必须置换一页时,

选择最旧的。不需要记录时间, 只需要 FIFO

队列来管理页即可。15 次缺页

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 0 1 1 1 2 2 4 4 4 4 0 1 0 1 2 2 2 2 2 2 3 3 3 3 0 1 1 2 7 7 0 1

page frames

FIFO 会出现可用帧越多, 错误数越大的问题,

这种结果叫 Belady's Anomaly Belady 异常;

Optimal Page Replacement OPT 最佳页面置换

OPT 时所有算法中页错误率最低, 且绝对没

有 Belady 异常。置换最长不会使用的页,

或者说选择未来不再使用/离当前最远位置

上出现的页置换。这个使用时长看下次该

页号出现的距离即可。就是向未来看

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 0 1 1 1 2 2 4 4 4 4 0 1 0 1 2 2 2 2 2 2 3 3 3 3 0 1 1 2 7 7 0 1

page frames

Least Recently Used LRU 最近最久使用

LRU 选择内存中最久没有引用的页面, 考

虑的是局部性原理, 性能最接近 OPT, 但是

需要记录页面的使用时间, 硬件开销太大,

就是向后看的算法。

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 0 1 1 1 2 2 4 4 4 4 0 1 0 1 2 2 2 2 2 2 3 3 3 3 0 1 1 2 7 7 0 1

page frames

LRU 算法如何获取多长时间没引用? 两种方

法:

计数器 counter: 每一个页表条目都有一

个 counter, 每次被引用, 就把时钟信息复

制到 counter。当置换时, 置换时间最小

的页, 最近越使用, clock 越大。

栈实现: 维护一个页码栈, 栈由双向链表

实现。引用页面时将该页面移动到顶部,

需要改变 6 个指针。替换时直接替换栈底

部就是 LRU 页。reference string

4 7 0 7 1 0 1 2 1 2 2 7 1 2

2 7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

stack before a stack after b

7 1 2 0 1 4

需要一个帧时, 指针向前移动知道找到一个

引用位 0 的页, 在其向前移动的过程中, 它

会清除引用位。最坏情况下所有帧都会被

给二次机会, 他就会清除所有引用位后再

选择页进行置换, 此时二次机会=FIFO。

reference bit

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 0 1 1 1 2 2 4 4 4 4 0 1 0 1 2 2 2 2 2 2 3 3 3 3 0 1 1 2 7 7 0 1

page frames

reference bit

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 0 1 1 1 2 2 4 4 4 4 0 1 0 1 2 2 2 2 2 2 3 3 3 3 0 1 1 2 7 7 0 1

page frames

Enhanced Second chance 改进 clock 增强二

次机会

通过将引用位和脏位作为有序对来考虑,

可以改进二次机会算法。两个位有四种可

能: (0,0)无引用无修改, 置换的最佳页

(0,1)无引用有修改, 置换前需要写回脏

页 (1,0)有引用无修改, 很可能会继续用

(1,1)有引用有修改, 很可能会继续用且

置换前须要写回脏页

淘汰次序 (0,0)> (0,1)> (1,0)> (1,1)

当页面需要被置换时, 使用时钟算法。置

换 (0,0) 的页, 在进行置换前可能要多次

搜索循环队列。改进的点在于给未引用

但是修改了的页更高优先级, 降低了 IO

数。Macintosh 使用

Counting 基于计数的置换算法

为每个页保存一个用于记录引用次数的

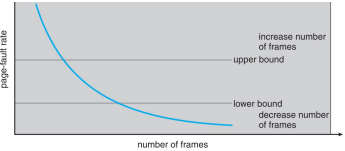
计数器, 具体方案有两种:

Least frequently used LFU: 置换计数最

小的。但是有问题: 一个页可能一开始

狂用, 但是后来不用了, 他的计数可能很

大,



Memory-Mapped Files 内存映射文件

使用虚存技术来讲文件 IO 作为普通文件访问的技术。

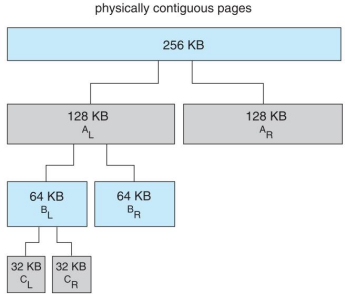
开始的文件访问按照普通按需请求调度，会出现页错误。这样，一页大小的部分文件从文件系统中读入物理页，以后的文件访问就可以按照通常的内存访问来处理，这样就可以用内存操作文件，而非 read write 等系统调用，简化了文件访问和使用。多个进城可以允许将同一文件映射到各自的虚存中，达到数据共享的目的。

Allocating Kernel Memory 内核内存分配

与对待用户内存不同：内核内存从空闲内存池中获取，两个原因：1.内核需要为不同大小的数据结构分配内存。2.一些内核内存需要连续。

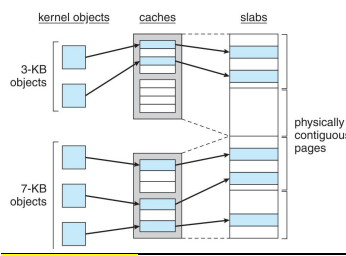
Buddy 系统

从物理上连续的大小固定的段上进行分配。内存分配按 2 得幂的大小来分配：请求大小必须是 2 的幂；如果不是，那么调整到下一个更大的 2 得幂；当需要比可用的更小的分配时，当前块分成两个下一个较低幂的段，继续这一过程直到适当大小的块可用。Buddy 系统的优点是可以通过合并快速形成更大的段。明显缺点是由于调整到下一个 2 的幂容易产生内存碎片。



Slab 分配

为了解决 Buddy 碎片损失的问题，slab 是由一个或多个物理上连续的页组成的。Cache 包含一个或者多个 slab，每个内核数据结构都有一个 cache，每个 cache 都含有内核数据结构的对象实例。当 cache 被创建时，起初包括若干标记为空闲的对象，对象的数量和 slab 大小有关，12KB 的 slab(包含三个连续的页)可以存储 6 个 2KB 的对象，当需要内核数据结构的对象时，可以直接从 cache 上取，并将该对象标记为使用 used，Slab 首先从部分空闲的 slab 中分配，如果没有则从全空的 slab 进行分配。如果没有，从物理连续页上分配新的 slab，把他赋给一个 cache，再从 slab 分配空间。Slab 优点：没有碎片引起的内存浪费；内存请求可以快速满足。



预调页 prepaging

为了减少冷启动时大量的页错误。同时将所有需要的页一起调入内存，但是如果预调页没有被用到，那么 IO 就被浪费了。假设 s 页被预调到内存，其中 a 部分被用到了。问题在于节省的 s*a 个页错误的成本是大于还是小于其他 s*(1-a)不必要的预调页开销。如果 a 接近于 0，调页失败，a 接近 1，调页成功。

页大小

页大小必须考虑到：碎片、页表大小、IO 开销以及局部性

TLB 范围 TLB reach

TLB 范围指通过 TLB 可以访问到的内存量。TLB Reach=TLB size * Page Size。理想情况下，每个进程的 WS 应该位于 TLB 中，否则就会有不通过 TLB 调页导致的大量 IO 增大页大小的话：可能会导致不需要大页表的进程带来的内存碎片

提供多种页大小的支持：那么 TLB 无法硬件化，性能降低。IO 互锁。IO 互锁指页面必定有时被所在内存中。必须锁住用于从设备复制文件的页，以便通过页面置换驱逐。

File System Interface 文件系统接口

文件是存储某种介质上的（如磁盘、光盘、SSD 等）并具 有文件名的一组相关信息的集合

文件属性

名称 标识符(唯一标识该文件的数字) 类型 位置 大小 保护 时间日期 用户标识 所有的文件信息都保存在目录结构中，而目录结构保存在外存上。

文件操作

文件是 ADT 抽象数据类型，其操作有：创建 读写 文件内重定位 删除 截短 truncate Open(Fi) 在硬盘上寻找文件目录块并且移动到内存中 Close(Fi)将内存中的目录结构移动到磁盘上。

打开文件

每个打开文件都有以下信息：文件指针：跟踪上次读写位置作为当前文件位置指针 文件打开计数器 file-open count：跟踪文件打开和关闭的数量，在最后关闭时，计数器为 0，系统可以移除该条目。

文件磁盘位置 disk location of file：用于定位磁盘上文件位置的信息

访问权限：访问模式信息

文件内部结构 File Structure

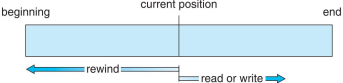
None 字 字节的序列 流文件结构 Simple record structure 记录文件结构：lines, fixed length, variable length Complex Structures：formatted document, relocatable load file

可以通过插入适当的控制字符，用第一种方法模拟最后两个 这些模式由 OS 和程序所决定。

访问方法

Sequential access 顺序访问

文件信息按顺序，一个记录接着一个记录处理。访问模式最操汗能够用，编辑器和编译器用这种方式。读操作读取文件下一文件部分，并自动前移文件指针，跟踪 IO 位置。写操作向文件尾部增加内容，相应文件指针到新文件结尾。顺序访问基于文件的磁带模型，也适用于随机访问设备。可以重新设置指针到开始位置或者向前向后跳过记录。No read after last write。



Direct access 直接访问

文件由固定长度的逻辑记录组成，允许程序按任意顺序进行快速读写。直接访问是基于文件的磁盘模型。文件可作为块或记录的编号序列。读写顺序没有限制。可以立即访问大量信息，DB 常用。

文件操作必须经过修改从而能将块号作为参数，有读 n 操作，而不是读下一个；写 n 操作：定位到 n；要实现读 n 只需要定位 n 再读下一个即可。注意 n 是相对块号，相对于文件开始的索引号。

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp ; cp = cp + 1;

Indexed block access 索引顺序访问访问

目录结构

目录是包含所有文件信息节点的集合。目录结构和文件在磁盘上。

磁盘结构

磁盘可以装多种文件系统，分区或片 minidisk slice。

目录操作

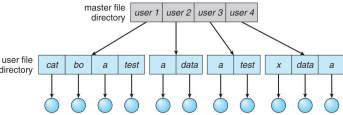
搜索文件 创建文件 删除文件 遍历 list 目录 重命名文件 遍历 traverse 文件系统

单级目录

所有文件包含在同一目录中，一个文件系统提供给所有用户。由于所有文件在同一级，不能有重名，此外存在着分组问题

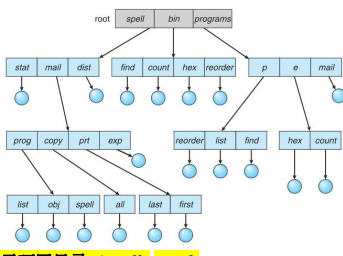
二级目录

为每个用户创建独立目录。每个用户都有自己的用户文件目录 user file directory UFD。不同用户可以有同名文件，搜索效率高，但是没有分组能力。



树形目录

将目录二级目录拓展即可。搜索高校 有分组能力。



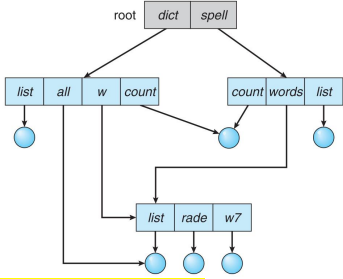
无环图目录 Acyclic graph

树形结构禁止共享文件和目录。无环图允许目录含有共享子目录和文件。

实现文件盒目录共享，UNIX 采用创建一个叫做链接的新目录条目。链接实际上是另一个文件的指针。连接通过使用路径名定位真正文件。注意，无环图目录倒置一个文件可以有多个绝对路径名。不同文件名可能表示同一文件，出现了别名问题。

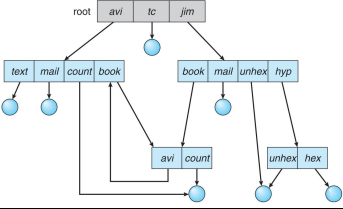
对于采用符号链接实现共享的系统，删除链接并不影响源文件，如果文件本身被删除，链接也被删除。

删除的另一方法是保留文件知道删除其全部引用，所以为文件引入了计数。删除一次链接或者条目就计数-1，到 0 时完全删除文件，UNIX 的硬链接采用这种方法，在 inode 中保留一个引用计数。通过禁止对目录的多重引用，可以维护无环图结构。



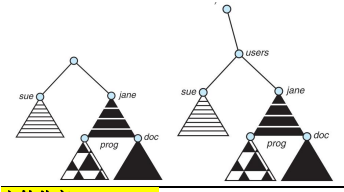
普通图目录 General graph

采用这种目录必须确保没有环，仅允许向文件链接，不允许目录，需要 GC。每次加入链接都要执行环检测算法。



文件系统挂载 mount

文件系统在访问前必须挂载。一个为挂载的文件系统会在挂载点 mount point 挂载。左图是未安装的卷，右图的 users 为挂载点。



文件共享 file sharing

多用户系统的文件共享很有用。文件共享需要通过一定的保护机制实现：在分布式系统，文件通过网络访问；网络文件系统 NFS 是常见的分布式文件共享方法。NFS 是 UNIX 文件共享协议 CIFS 是 WIN 的协议。

保护 Protection

访问类型：读 写 执行 追加 append 删除 列表清单 list

访问控制列表 access-control list ACL

三种用户类型：拥有者 owner access 组 group access 其他 public access

在 UNIX 里，一个类型有 rwx 三个权限，所以一个文件需要 3*3=9 位说明文件访问权限。

File System Implementation 文件系统实现

文件系统：是操作系统中以文件方式管理计算机软件资源的软件和被管理的文件和数据结构（如目录和索引表等）的集合。文件系统储存在二级存储中，磁盘。

文件控制块 file control block：包含文件的树形如拥有者、权限、文件内容的位置。设备驱动控制物理设备。

分层设计的文件结构

应用程序：发出文件请求者。逻辑文件系统：管理元数据：文件系统的所有结构数据，而不包括文件的实际顺序；根据给定符号文件名来管理目录结构；逻辑文件系统通过 FCB 来维护文件结构。

文件组织模块知道文件及其逻辑块和物理块，包括空闲空间管理器。基本文件系统：向合适的设备驱动程序发送一般命令就可 对磁盘上的物理块进行读写

IO 控制：由设备驱动程序和中断处理程序组成，实现内存与磁盘之间的信息转移

文件系统实现

On-Dist FS structure

磁盘上，文件系统可能包括：如何启动所存储的 OS、总块数、空闲块数目和位置，目录结构及各个具体文件。

磁盘结构包括：每个卷的引导控制块 boot control block 包括从该卷引导操作系统所需的信息；每个卷的卷控制块 volume control block 包括卷的详细信息；目录结构来组织文件；每个文件的 FCB。

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

In-Memory FS structure

In-mrmory partition table 分区表 in-memory directory structure 目录结构 system-wide open-file table 系统打开文件表 per-process open-file table 进程打开文件表

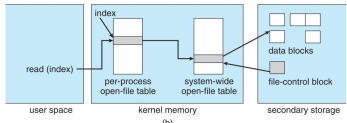
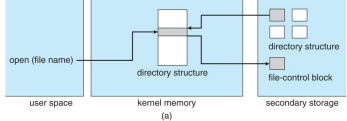


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

虚拟文件系统 VFS

VFS 提供面向对象的方法实现文件系统。允许将相同的系统调用接口（API）用于不同类型的文件系统。

目录实现

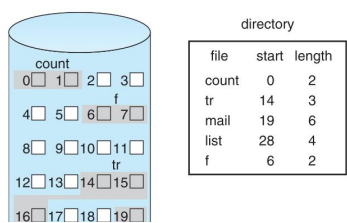
线性列表 linear list：使用储存文件名和数据块指针的线性表。哈希表：线性表与哈希结构，哈希表根据文件名得到一个值返回一个指向线性表中元素的指针。

分配方法 Allocation Method

常见的主要磁盘空间分配方法：连续、链接和索引。

连续分配 Contiguous Allocation

每个文件在磁盘上占有一组连续的块。优点：访问很容易，只需要起始块位置和块长度就可以读取。支持岁寄存器 random access。但是浪费空间，存在动态存储分配问题。First and best 表现差不多，first 时间快很多。存在外碎片问题，此外文件大小不可增长。逻辑到物理的映射：LA/512 分为两部分商 Q 和余数 R，Block to be accessed = Q + starting address Displacement into block = R，LA 是存取文件逻辑地址，512 是块大小



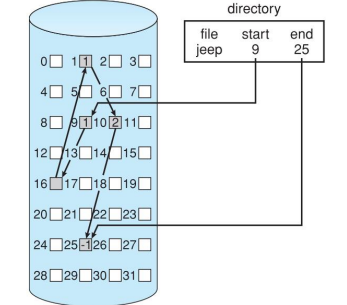
变种：基于长度的系统。利于 Veritas FS 采用。解决了文件大小无法增长的问题，增加了另一个叫做 extent 的连续空间给空间不够的文件，然后与原文件块之间有个指针。一个文件可以

有多个 extent。

Linked Allocation 链接分配

解决了连续分配的所有问题。每个文件都是磁盘块的链表。访问起来只需要一个起始地址。没有空间管理问题，不会浪费空间，但是不支持 random access。

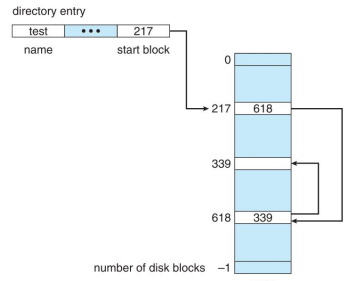
地址映射：LA/(512-1)得到商 Q 和余 R，Block to be accessed is the Qth block in the linked chain of blocks representing the file. Displacement into block = R + 1。因为每个索引块的末尾节点是用来链接下一个索引块的，不链数据块，所以要 512-1



FAT File allocation table 文件系统

磁盘空间分配用于 MS-DOS 和 OS/2。FAT32 引导区记录被扩展为包括重要数据结构的备份，根目录为一个普通的簇链，其目录项可以放在文件区任何地方。原本的链接分配有问题，指针在每个块中都会占空间，可靠性也不高，任何指针丢失都会导致文件其余部分丢失。FAT 采用单独的磁盘区保存链接。

计算机系统启动时，首先执行的是 BIOS 引导程序，完成自检，并加载主引导记录和分区表，然后执行主引导记录，由它引导 激活分区引导记录，再执行分区引导记录，加载操作系统，最后执行操作系统，配置系统。FAT32 目录结构：FAT 的每个目录项为 32 个字节；FAT32 长文件名的目录项由几个 32B 表项组成。，用一个表项存放短文件名和其他属性（包括簇号、文件大小，最后修改时间和最后修改日期、创建时间、创建日期和最后存取日期），短文件名的属性是 0x20。用连续若干个表项存放长文件名，每个表项存放 13 个字符（使用 Unicode 编码，每个字符占用 2 个字节。）长文件名的表项首字节的二进制制低 5 位值，分别为 00001、00010、00011、……，表示它们的次序，左起第 2 位为 1（也就是在低 5 位基础上加 40H）表示该表项是最后一项。最后项存放 13 个字符位置多余时，先用 2 个字节 0 表示结束，再用 FFH 填充。长文件名的属性是 0x0F。长文件名项的第 13、27、28 字节为 0x00，第 14 字节为短文件名校验和。



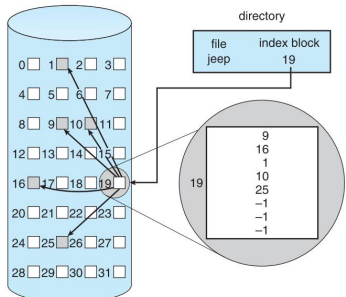
NTFS 文件系统

每个分区都有 MFT master file table，MFT 由一个个 MFT 项（也称为文件记录）组成，每个 MFT 项占用 1024 字节的空间。MFT 前 16 个记录用来存放元数据文件的信息，它们占有固定的位置。每个 MFT 项的前部几十个字节有着固定的头结构，用来描述本 MFT 项的相关信息。后面的字节存放着文件属性等。每个文件或目录的信息都包含在 MFT 中，每个文件或目录至少有一个 MFT 项。

Indexed Allocation 索引分配

索引分配把所有指针放在一起，通过索引块解决这个问题。每个文件都有索引块，是一个磁盘块地址的数组。当首次写入第 i 块时，先从空闲空间管理器获得一块，再将其地址写到索引块中的第 i 个条目。对于小文件，大部分索引块被浪费。如果索引块太小，可以多层索引，然后互相连接。访问需要索引表，支持 randomaccess，动态访问没有外碎片，但是有索引开销。LA/512 得商 Q 和余数 R，Q = displacement into index table R = displacement into block

链接索引 Linked scheme，把索引块链接起来，LA/(512*511)得到商 Q1 和余数 R1，商 Q1 时索引表的块号，R1/512 得到商 Q2 和余数 R2，Q2 = displacement into block of index table，R2 = displacement into block of file。如果是二级索引，那么 LA/(512*512)得到 Q1 和 R1，Q1displacement into outer-index，R1/512 得到 Q2 和 R2，Q2 = displacement into block of index table，R2displacement into block of file:



索引计算

连续、链接组织的最大文件大小可以大到整个磁盘文件分区。考虑每块大小 4KB，块地址 4B。一级索引：一个索引块可以存 4KB/4B=1K 个索引地址，每个索引地址直接引到文件块，所以最大 1K*4KB=4MB。二级索引：一个索引块可以再继续连接到索引块，因此有 1K*1K*4KB=4GB 的最大文件。

采用 Linux 分配方案，Linux 中共有 15 个指针在 inode 中，前面 12 个直接指向文件块，因此有 48KB 可以直接访问，其他三个指针指向间接块，第一个间接块指针是指向以及间接块，第二个是二级间接块，第三个是三级间接块。因此最大文件的大小为：12*4KB+1K*4KB+1K*1K*4KB+1K*1K*1K*4KB=48KB+4MB+4GM+4TB

空闲空间管理

位图 bit vector: 空闲置 0 占有置 1，块数计算: (number of bits per word)*(number of 0-value words)+offset of first 1bit 位向量所需空间的计算: disk size/block size 便于查找连续文件。

采用链表管理：把所有的空闲块链接起来，将指向第一个空闲块的指针保存在磁盘的特殊位置并且还存在内存中，但是 IO 效率很低，因为需要遍历。对空闲链表的改进是将 n 个空闲块的地址存到第一个空闲块中，这样可以快速找到大量空闲块的地址。

还有计数的方法：不记录 n 个空闲块的地址，而是记录第一各空闲块和紧跟着的空闲块的数量 n。页面缓冲 page buffer 将文件数据作为页而不是磁盘块缓冲起来到盘存。

恢复 Recovery

一致性检查：将目录结构数据与磁盘数据块比较，并且纠正发现的不一致。用系统程序将磁盘数据备份到另一个设备。然后从该设备恢复。

日志结构的文件系统

日志文件系统记录文件系统的更新为事务。事务会被写到日志里。事务一旦写入日志就是已经 commit 了，否则文件系统还没更新。

Mass storage system 大容量存储

磁盘的 0 扇区是最外面的第一个磁道的第一个扇区，逻辑块时最小传出单位 512B

磁盘调度

Seek time 寻道时间，磁头移动到包含目标扇区的柱面的时间。旋转延迟 rrotational latency: 旋转到目标扇区的时间。传输时间 transfer time: 数据传输时间 磁盘带宽是传递的总字节数初一服务请求到传递结束的总时间。平均旋转延迟=一圈/2

FCFS 先来先服务：算法公平，但不是最快。SSTF 最短寻道时间优先：处理靠近当前磁头位置的请求，本质上和 SJF 一样，所以有可能请求会永远无服务，时间也不是最优。SCAN: 从磁盘一端到另一端，移过的柱面进行服务。到达另一端时改变移动方向，继续处理，也叫作电梯算法。

CSCAN 磁头从一端移动到另一端，到了另一端就马上返回到磁盘开始，返回路径不服务。LOOK: 磁头从一端到另一端，到达另一端最远的服务就不继续走了，开始折返服务。CLOOK: 磁头从一端到另一端，到达另一端最远服务就立即返回到磁盘开始的第一个服务，返回路径不服务。

调度算法选择：SSTF 一般来说比较好 SCAN SSCAN 对于高负荷磁盘表现更好 表现依赖于请求类型和数量：磁盘请求又依赖于

于文件分配策略：磁盘调度算法应该模块化，可以随时更换自由选择。SSTF 或者 LOOK 都是很棒的默认算法。

磁盘管理

低级格式/物理格式化：将磁盘划分为扇区才能进行读写。逻辑格式化：创建文件系统。要是用一个磁盘保存文件，OS 需要这么几步：首先分区，然后逻辑格式化，也就是创建文件系统；为了提升效率然后将块集中到一起成为簇 cluster。

一般 bootstrap 存在 ROM 里。系统启动顺序：ROM 中的代码(simple bootstrap)boot block 里的代码(full bootstrap)也就是 boot loader 如 Grub LILO 然后是整个 OS 内核

RAID

0: 无冗余 1: 镜像 2: 纠错码 3: 奇偶校验 1 个盘 -r 添加条带化 Striping 5: 校验盘分散到各个盘 6: P+Q 冗余，差错纠正码

三级存储 Tertiary storage device

Low cost is the defining characteristic of tertiary storage. Generally, tertiary storage is built using removable media Common examples of removable media are floppy disks and CD-ROMs; other types are available

Swap space 三级存储 Tertiary storage device 虚存使用硬盘空间作为主存。两种形式：普通文件系统: win 都是 pagefile.sys 独立硬盘分区 linux solaris 都是 swap 分区。还有一种方法：创建在 raw 的磁盘分区上，这种速度最快。

性能:

Sustained bandwidth 大传输的平均速率 字节/时间。Effective bandwidth IO 时间下的平均速率。前者是数据真正流动时的速率，后者是驱动器能够提供的能力，一般驱动器带宽指前者。

IO system

IO 方式

轮询 polling 中断 CPU 硬件有一条中断请求线 IRL，IO 设备触发，需要 IO 时就申请中断。DMA direct memory access 对于需要进行大量 IO 的设备，为了避免程序控制 IO 即 PIO，将一部分任务西方给了 DMA 控制器，在 DMA 开始传输时，主机向内存中写入 DMA 命令块。然后 CPU 在写入后继续干别的，DMA 去自己操作内存总线，然后就可以向内存进行传输。

IO 应用接口

实现统一的 IO 接口，设备驱动提供了 API 来操控 IO 设备 设备分成很多种：Character-stream or block 字符流或者块设备 Sequential or random-access 顺序或随机访问设备 Synchronous or a Synchronous 同步或异步 Sharable or dedicated 共享或独占设备 Speed of operation Operating System jim 操作速度（快速、中速、慢速）read-write, read only, or write only 读写、只读、只写设备

块设备和字符设备 块设备：包括硬盘，一般有读写 seek 的命令，对其进行 raw 原始 IO 或者文件系统访问。内存映射文件访问也 OK 字符设备：键盘鼠标串口，命令是 get put。库函数提供具有缓冲和编辑功能的访问访问。阻塞 IO 和非阻塞 IO 阻塞 IO 进程挂起直到 IO 完成，很容易使用和理解，

但是不能满足某些需求 非阻塞 IO: IO 调用立刻返回。用户接口就是，接收鼠标键盘输入，还要在屏幕上输出，放视频也是，从磁盘读帧然后显示。异步: IO 与进程同时运行。非阻塞和异步的区别: 非阻塞的 read 会马上返回，虽然可能读取的数据没有达到要求的，或者就没读到。异步 read 一定要完整执行完

实验

2. 命令功能：用来压缩和解压文件。tar 本身不具有压缩功能。他是调用压缩功能实现的 3. 命令参数：必要参数有如下： -A 新增压缩文件到已存在的压缩 -B 设置区块大小 -c 建立新的压缩文件-d 记录文件的差别 -r 添加文件到已经压缩的文件 -u 添加改变了和现有的文件到已经存在的压缩文件 -x 从压缩的文件中提取文件 -t 显示压缩文件的内容 -z 支持 gzip 解压文件 -j 支持 bzip2 解压文件 -Z 支持 compress 解压文件 -v 显示操作过程 -l 文件系统边界设置 -k 保留原有文件不覆盖 -m 保留文件不被覆盖 -W 确认压缩文件的正确性 可选参数如下： -b 设置区块数目 -C 切换到指定目录 -f 指定压缩文件 --help 显示帮助信息 --version 显示版本信息 4. 常见解压/压缩命令 tar 解包: tar xvf FileName.tar 打包: tar cvf FileName.tar DirName (注: tar 是打包，不是压缩!)

-gz 解压 1: gunzip FileName.gz 解压 2: gzip -d FileName.gz 压缩: gzip FileName.tar.gz 和 .tgz 解压: tar zxvf FileName.tar.gz 压缩: tar zcvf FileName.tar.gz DirName .bz2 解压 1: bzip2 -d FileName.bz2 解压 2: bunzip2 FileName.bz2 压缩: bzip2 -z FileName .tar.bz2 解压: tar jxvf FileName.tar.bz2 压缩: tar jcvf FileName.tar.bz2 DirName .bz 解压 1: bzip2 -d FileName.bz 解压 2: bunzip2 FileName.bz .tar.bz 解压: tar jxvf FileName.tar.bz .Z 解压: uncompress FileName.Z 压缩: compress FileName .tar.Z 解压: tar Zxvf FileName.tar.Z 压缩: tar Zcvf FileName.tar.Z DirName .zip 解压: unzip FileName.zip 压缩: zip FileName.zip DirName .rar 解压: rar x FileName.rar 压缩: rar a FileName.rar DirName 5. 使用实例 实例 1: 将文件全部打包成 tar 包 命令: tar -cvf log.tar log2012.log

tar -zcvf log.tar.gz log2012.log tar -jcvf log.tar.bz2 log2012.log Task_struct 结构定义 include/linux/sched.h, line 701 volatile long state 描述进程状态的成员 unsigned long flags 反应进程的状态信息 mm 进程所拥有的虚存信息，内核线程为 NULL struct fs_struct *fs //进程的可执行映象所在的文件系统 struct files_struct *files //进程打开的文件

VFS

VFS 并不是一种实际的文件系统，ext2 等物理文件系统是存在于外存空间的，而 VFS 仅存在于内存。文件系统的源代码可以在 linux/fs 中找到。

超级块对象 superblock :存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统超级块或控制块。

索引节点对象 inode object : 存储某个文件的信息，通常对应磁盘文件系统的文件控制块 目录项对象 dentry object : dentry 对象主要是描述一个目录项，是路径的组成部分。文件对象 file object: 存储一个打开文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在与内存 struct super_block { s_list: 指向了超级块表中前一个超级块和后一个超级块的指针。 s_dev: 超级块所在的设备的描述符。 s_blocksize 和 s_blocksize_bits: 指定了磁盘文件系统的块的大小。 s_dirty: 超级块的“脏”位。 s_maxbytes: 文件最大的大小。 s_type: 指向文件系统的类型的指针。 s_op : 指向超级块操作的指针，指向 super_operations 结构的指针，super_operations 中包含着一系列的操作函数指针，即这些操作函数的入口地址

s_root: 指向目录的 dentry 项。 s_dirt: 表示“脏”（内容被修改了，但尚未被刷新到磁盘上）的 inode 节点的链表，分别指向前一个节点和后一个节点。 s_fs_info: 指向各个文件系统私有数据，一般是各文件系统对应的超级块信息。以 ext2 文件系统为例，当 ext2 文件系统的超级块装入到内存，即装入到 super_block 的时候，会调用 ext2_fill_super()函数，在这个函数中填写 ext2 对应的 ext2_sb_info，然后挂在这个指针上。

物理文件系统的 inode 在外存中并且是长期存在的，VFS 的 inode 对象在内存中，它仅在需要时才建立，不再需要时撤销。物理文件系统的 inode 是静态的，而 VFS 的 inode 是一种动态结构

VFS 的 inode 与某个文件的对应关系是通过设备号 i_dev 与 inode 号 i_ino 建立的，它们唯一地指定了某个设备上的一个文件或目录。

VFS 的 inode 是物理设备上的文件或目录的 inode 在内存中的统一映像。这些特有信息是各种文件系统的 inode 在内存中的映像。如 EXT2 的 ext2_inode_info 结构。

i_lock 表示该 inode 被锁定，禁止对它的访问。i_flock 表示该 inode 对应的文件被锁定。i_flock 是个指向 file_lock 结构链表的指针，该链表指出了一系列被锁定的文件。

VFS 的 inode 组成一个双向链表，全局变量

first_inode 指向链表的表头。在这个链表中，空闲的 inode 总是从表头加入，而占用的 inode 总是从表尾加入。

系统还设置了一些管理 inode 对象的全局变量，如：

```
max_inodes 给定了 inode 的最大数量，
nr_inodes 表示当前使用的 inode 数量，
nr_free_inodes 表示空闲的 inode 数量。
```

Inode operations

create: 只适用于目录 inode，当 VFS 需要在“inode”里面创建一个文件（文件名在 dentry 里面给出）的时候被调用。VFS 必须已经检查过文件名在这个目录里面不存在。

lookup: 用于检查一个文件（文件名在 dentry 里面给出）是否在一个 inode 目录里面。

link: 在 inode 所给出的目录里面创建一个从第一个参数 dentry 文件到第三个参数 dentry 文件的硬链接（hard link）。

unlink: 从 inode 目录里面删除 dentry 所代表的文件。

symlink: 用于在 inode 目录里面创建软链接（soft link）。

mkdir: 用于在 inode 目录里面创建子目录。

rmdir: 用于在 inode 目录里面删除子目录。

mknod: 用于在 inode 目录里面创建设备文件。

rename: 把第一个和第二个参数（inode，dentry）所定位的文件改名为第三个和第四个参数所定位的文件。

readlink: 读取一个软链接所指向的文件名。

follow_link: VFS 调用这个函数跟踪一个软链接到它所指向的 inode。

put_link: VFS 调用这个函数释放 follow_link 分配的一些资源。

truncate: VFS 调用这个函数改变一个文件的大小。

permission: VFS 调用这个函数得到对一个文件的访问权限。

setattr: VFS 调用这个函数设置一个文件的属性。比如 chmod 系统调用就是调用这个函数。

getattr: 查看一个文件的属性。比如 stat 系统调用就是调用这个函数。

setxattr: 设置一个文件的某项特殊属性。详细情况请查看 setxattr 系统调用帮助。

getxattr: 查看一个文件的某项特殊属性。详细情况请查看 getxattr 系统调用帮助。

listxattr: 查看一个文件的所有特殊属性。详细情况请查看 listxattr 系统调用帮助。

removexattr: 删除一个文件的特殊属性。详细情况请查看 removexattr 系统调用帮助。

目录项对象 dentry object

每个文件除了有一个索引节点 inode 数据结构外，还有一个目录项 dentry 数据结构。每个 dentry 代表路径中的一个特定部分。如：/、bin、vi 都属于目录项对象。目录项也可包括安装点，如：/mnt/cdrom/foo、/mnt、cdrom、foo 都属于目录项对象。inode 结构代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统，其 inode 结构在磁盘上就有对应的映像。一个索引节点对象可能对多个目录项对象。目录项对象作用是帮助实现文件的快速定位，还起到缓冲作用

```
struct dentry * d_parent; /* 父目录的目录项
struct list_head d_hash; /* 目录项形成的哈希表 */
struct list_head d_lru; /* 未使用的 LRU 链表 */
struct list_head d_child; /* 父目录的子目录所形成的链表 */
struct list_head d_subdirs; /* 该目录的子目录所形成的链表 */
struct list_head d_alias; /* 索引节点别名的链表 */
int d_mounted; /* 目录项的安装点 */
struct qstr d_name; /* 目录项名（可快速查找） */
struct dentry_operations *d_op; /* 操作目录项的函数 */
struct super_block *d_sb; /* 目录项树的根（即文件的超级块） */
unsigned long d_vfs_flags;
void *d_fsdata; /* 具体文件系统的数据 */
unsigned char d_iname[DNAME_INLINE_LEN]; /* 短文件名 */
.....
VFS 的 dentry cache 与 inode cache

为了加速对经常使用的目录的访问，VFS 文件系统维护着一个目录项的缓存。为了加快文件的查找速度 VFS 文件系统维护一个 inode 节点的缓存以加速对所有装配的文件系统的访问。用 hash 表将缓存对象组织起来。



File 对象



文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。该对象由相应的 open() 系统调用创建，由 close() 系统调用销毁。



```
struct file {
 struct list_head f_list; /*file 结构链表*/
 struct dentry *f_dentry; /*指向与文件对象关联的 dentry 对象*/
 struct vfsmount *f_vfsmnt; /* 文件相应的 vfsmount 结构*/
 struct file_operations *f_op; /*文件对象的操作集合*/
 atomic_t f_count; /*文件打开的引用计数*/
 unsigned int f_flags; /*使用 open () 时设定的标志*/
 mode_t f_mode; /*文件读写权限*/
 loff_t f_pos; /*对文件读写操作的当前位置*/
 struct fown_struct f_owner;

};
file_operations
lseek: 用于移动文件内部偏移量。
read: 读文件。aio_read: 异步读，被 io_submit 和其他的异步 IO 函数调用。
write: 写文件。
aio_write: 异步写，被 io_submit 和其他的异步 IO 函数调用。
readdir: 当 VFS 需要读目录内容的时候调用这个函数。
poll: 当一个进程想检查一个文件是否有内容可读写的时候，VFS 调用这个函数：一般来说，
```


```

```
struct dentry * d_parent; /* 父目录的目录项
struct list_head d_hash; /* 目录项形成的哈希表 */
struct list_head d_lru; /* 未使用的 LRU 链表 */
struct list_head d_child; /* 父目录的子目录所形成的链表 */
struct list_head d_subdirs; /* 该目录的子目录所形成的链表 */
struct list_head d_alias; /* 索引节点别名的链表 */
int d_mounted; /* 目录项的安装点 */
struct qstr d_name; /* 目录项名（可快速查找） */
struct dentry_operations *d_op; /* 操作目录项的函数 */
struct super_block *d_sb; /* 目录项树的根（即文件的超级块） */
unsigned long d_vfs_flags;
void *d_fsdata; /* 具体文件系统的数据 */
unsigned char d_iname[DNAME_INLINE_LEN]; /* 短文件名 */
.....
VFS 的 dentry cache 与 inode cache

为了加速对经常使用的目录的访问，VFS 文件系统维护着一个目录项的缓存。为了加快文件的查找速度 VFS 文件系统维护一个 inode 节点的缓存以加速对所有装配的文件系统的访问。用 hash 表将缓存对象组织起来。



File 对象



文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。该对象由相应的 open() 系统调用创建，由 close() 系统调用销毁。



```
struct file {
 struct list_head f_list; /*file 结构链表*/
 struct dentry *f_dentry; /*指向与文件对象关联的 dentry 对象*/
 struct vfsmount *f_vfsmnt; /* 文件相应的 vfsmount 结构*/
 struct file_operations *f_op; /*文件对象的操作集合*/
 atomic_t f_count; /*文件打开的引用计数*/
 unsigned int f_flags; /*使用 open () 时设定的标志*/
 mode_t f_mode; /*文件读写权限*/
 loff_t f_pos; /*对文件读写操作的当前位置*/
 struct fown_struct f_owner;

};
file_operations
lseek: 用于移动文件内部偏移量。
read: 读文件。aio_read: 异步读，被 io_submit 和其他的异步 IO 函数调用。
write: 写文件。
aio_write: 异步写，被 io_submit 和其他的异步 IO 函数调用。
readdir: 当 VFS 需要读目录内容的时候调用这个函数。
poll: 当一个进程想检查一个文件是否有内容可读写的时候，VFS 调用这个函数：一般来说，
```


```

调用这个函数之后进程进入睡眠，直到文件中有内容读写就绪时被唤醒。详情请参考 select 和 poll 系统调用。

ioctl: 被系统调用 ioctl 调用。

unlocked_ioctl: 被系统调用 ioctl 调用；不需要 BKL（内核锁）的文件系统应该使用这个函数，而不是上面那个 ioctl。

compat_ioctl: 被系统调用 ioctl 调用；当在 64 位内核上使用 32 位系统调用时候使用这个 ioctl 函数。

mmap: 被系统调用 mmap 调用。

open: 通过创建一个新的文件对象而打开一个文件，并把它链接到相应的索引节点对象。

flush: 被系统调用 close 调用，把一个文件内容写回磁盘。

release: 当对一个打开文件的最后引用关闭的时候，VFS 调用这个函数释放文件。

fsync: 被系统调用 fsync 调用。

fsasync: 当对一个文件启用异步读写（非阻塞读写）的时候，被系统调用 fcntl 调用。

lock: fcntl 系统调用使用命令 F_GETLK, F_SETLK 和 F_SETLKW 的时候，调用这个函数。

编译内核:

make clean 删除大多数的编译生成文件，但是会保留内核的配置文件.config，还有足够的编译支持来建立扩展模块

make mrproper 删除所有的编译生成文件，还有内核配置文件，再加上各种备份文件

为了与正在运行的操作系统内核的运行环境匹配，可以先把当前已配置好的文件复制到当前目录下，新的文件名为.config 文件：

```
cp /boot/config-`uname -r` .config
make distclean mrproper 删除的文件，加上编辑备份文件和一些补丁文件。
apt-get install kernel-package
libncurses5-dev fakeroot wget bzip2
//安装工具包
make config 是有问必答的方式，每个内核选项它都会问你，要不要、模块，选错了一个就必须从头再来一遍；
make menuconfig 提供一个基于文本的图形界面，它依赖于 ncurses5 这个包，键盘操作，可以修改选项，一般推荐用这个；
make xconfig 需要你有一个 x window system 支持，就是说你要在 KDE、GNOME 之类的 X 桌面环境下才可用，好处是支持鼠标，坏处是 X 本身占用系统周期，而且 X 环境容易引起编译器的不稳定
make -j4 启动 4 个线程（双核）来编译内核文件生成 o 等中间文件
内核文件 bzImage 的位置在 /usr/src/linux/arch/i386/boot 目录下。
make modules_install 安装模块
make install 使用命令 make install 将 bzImage 和 System.map 拷贝到/boot 目录下。这样，Linux 在系统引导后从/boot 目录下读取内核映像到内存中
添加系统调用号 在系统调用表中修改或添加相应项
添加系统调用:
system_call()函数实现了系统调用中断处理程序；

1.它首先把系统调用号和该异常处理程序用到的所有 CPU 寄存器保存到相应的栈中，SAVE_ALL


```

2.把当前进程 task_struct (thread_info) 结构的地址存放在 ebx 中

3.对用户态进程传递来的系统调用号进行有效性检查。若调用号大于或等于 NR_syscalls，系统调用处理程序终止。（sys_call_table）

4.若系统调用号无效，函数就把-ENOSYS 值存放在栈中 eax 寄存器所在的单元，再跳到 ret_from_sys_call()

5.根据 eax 中所包含的系统调用号调用对应的特定服务例程

实验修改的主要有 3 处地方：

```
for (p = &init_task; (p = next_task(p)) != &init_task;)
//遍历进程
p->comm //comm 类型为 char[16],代表进程名
p->pid //当亲进程号
p->state //当前进程的状态
-1 unrunnable, 0 runnable, >0 stopped
p->parent //指向父进程 task_struct 的地址
添加文件系统:
文件系统:
普通文件（文件名不超过 255）
目录文件
字符设备文件和块设备文件:
fd0 (for floppy drive 0)
hda (for harddisk a)
lp0 (for line printer 0)
tty (for teletype terminal)
管道(FIFO)文件 链接文件 socket 文件
文件系统分三大类：基于磁盘的文件系统，如 ext2/ext3/ext4、VFAT、NTFS 等，网络文件系统，如 NFS 等。特殊文件系统，如 proc 文件系统、devfs、sysfs (/sys) 等。
Linux 以 ext2/ext3 做为基本的文件系统，所以它的虚拟文件系统 VFS 中也设置了 inode 结构。物理文件系统的 inode 在外存中并且是长期存在的，VFS 的 inode 对象在内存中，它仅在需要时才建立，不再需要时撤消。物理文件系统的 inode 是静态的，而 VFS 的 inode 是一种动态结构。
dd: 用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换
命令语法：dd [选项]
if = 输入文件（或设备名称）
of = 输出文件（或设备名称）
bs = bytes 同时设置读/写缓冲区的字节数（等于设置 ibs 和 obs）
count=blocks 只拷贝输入的 blocks 块
conv = ucase 把字母由小写转换为大写
conv = lcase 把字母由大写转换为小写。
例：dd if=/dev/zero of=myfs bs=1M count=1 /dev/zero: 零设备 "0"
/dev/loop: loopback device (回环设备、或虚拟设备) 是指用文件来模拟块设备
实验报告内容
#mount -t myext2 -o loop /fs.new /mnt
#cd /mnt
#mknod myfifo p
mknod: `myfifo': Operation not permitted
第一行命令：将 fs.new mount 到/mnt 目录下。
第二行命令：进入/mnt 目录，也就是进入 fs.new 这个 myext2 文件系统。
第三行命令：执行创建一个名为 myfifo 的命名管道的命令。
第四、五行是执行结果：第四行是我们添加的
```

myext2_mknod 函数的 printk 的结果:第五行是返回错误号 EPERM 结果给 shell，shell 捕捉到这个错误后打出的出错信息。

```
#!/bin/bash
/sbin/losetup -d /dev/loop2
/sbin/losetup /dev/loop2 $1
/sbin/mkfs.ext2 /dev/loop2
dd if=/dev/loop2 of=/tmpfs bs=1k count=2
./changeMN $1 /tmpfs
dd if=/fs.new of=/dev/loop2
/sbin/losetup -d /dev/loop2
rm -f /tmpfs
第一行 表明是 shell 程序.第三行 如果有程序用了 /dev/loop2 了，就将它释放。第四行 用 losetup 将第一个参数代表的文件装到 /dev/loop2 上 第五行 用 mkfs.ext2 格式化 /dev/loop2。也就是用 ext2 文件系统格式格式化我们的文件系统。第六行 将文件系统的头 2K 字节的内容取出来，复制到 tmpfs 文件里面。第七行 调用程序 changeMN 读取 tmpfs，复制到 fs.new，并且将 fs.new 的 magic number 改成 0x6666 第八行 再将 2K 字节的内容写回去。第九行 把我们的文件系统从 loop2 中卸下来。第十行 将临时文件删除。
```

习题

Suppose a system had 12 resources, 3 processes P0, P1 and P2.

	max	current	need
P0	10	5	5
P1	4	2	2
P2	9	3	6

Currently there are two resources available. This system is in an unsafe state as process P1 could complete, thereby freeing a total of four resources. But we cannot guarantee that processes P0 and P2 can complete. However, it is possible that a process may release resources before requesting any further. For example, process P2 could release a resource, thereby increasing the total number of resources to five. This allows process P0 to complete,which would free a total of nine resources, thereby allowing process P2 to complete as well.

A system has 3 concurrent processes, each of which requires 4 items of resource R. What is the minimum number of resource R in order to avoid the deadlock. Answer: 10

The system design the structure File Control Block (FCB) to manage the files. Commonly, File control block is created on disk when the open system call is invoked.

Which kind of swap space is fastest?

A raw partition

2. 文件 F 由 200 条记录组成。记录从 1 开始编号。用户打开文件后，欲将内存中的一条记录插入文件 F 中，作为第 30 条记录，请回答下列问题，并说明理由。

(1) 若文件系统为顺序分配方式，每个存储块放一条记录，文件 F 的存储区块尾均有足够空闲的存储块时，则最少需要多少次访问多少存储块？ F 的文件控制区中是否有哪些信息？

(2) 若文件系统为链式分配方式，每个存储块放的一条记录和一个指针时，则需从网上还须再读多少块？访问多少存储块？若每个存储块大小为 15K，其中 4 个字节存放指针，则该系统文件的最大长度是多少？

(3) 若文件系统为索引分配方式，每个存储块放的一条记录和一个索引指针，则需从网上还须再读多少块？访问多少存储块？若每个存储块大小为 15K，其中 4 个字节存放指针，则该系统文件的最大长度是多少？

(4) 采用直接方式需要访问多少块？2 块存储块，然后将新记录的存储块插入链中即可，则需再读入 1 次指针，然后应读第 29 块的链接地址再读回链又一次指针，一共就需 29+1+1=31 次。

4 个字节指控制地址范围是 2³²，(1K=4096) =4096*32

所以此系统该文件的最大长度为 2³²*(1K=4096) =4096*32

Q: 一个文件系统中有一个 20MB 大文件和一个 20KB 小文件，当分别采用连续、链接、链接索引、二级索引和 LINUX 分配方案时，每块大小为 4096B，每块地址用 4B 表示。问：

(1) 各文件系统管理的最大文件是多少？

(2) 每种方案对大、小两文件各需要多少专用块来记录文件的物理地址(说明各块的用途)？

(3) 如需要读大文件前面第 5.5KB 的信息和后面第 (16M + 5.5KB) 的信息，则每个方案各需要多少次盘 I/O 操作？

A: (1) 连续分配：理论上是不受限制，可大到整个磁盘文件区。

隐式链接：由于块的地址为 4 字节，所以能表示的最多块数为 232=4G，而每个盘中存放文件大小为 4092 字节。链接分配可管理的最大文件为：4G × 4092B=16368GB

链接索引：由于块的地址为 4 字节，所以最多的链接索引块数为 232=4G，而每个索引块有 1023 个文件块地址的指针，盘块大小为 4KB。假设最多有 n 个索引块，则 1023 × n=n+232，算出 n=222，链接索引分配可管理的最大文件为：4M10234KB=16368GB

二级索引：由于盘块大小为 4KB，每个地址用 4B 表示，一个盘块可存 1K 个索引表目。

二级索引可管理的最大文件容量为 4KB × 1K × 1K = 4GB。

LINUX 混合分配：LINUX 的直接地址指针有 12 个，还有一个一级索引，一个二级索引，一个三级索引。因此可管理的最大文件为 48KB + 4MB+4GB + 4TB。

(2) 连续分配：对大小两个文件都只需在文件控制块 FCB 中设二项，一是首块物理块号，另一是文件总块数，不需专用块来记录文件的物理地址。

隐式链接：对大小两个文件都只需在文件控制块 FCB 中设二项，一是首块物理块号，另一是末块物理块号；同时在文件的每个物理块中设置存放下一个块号的指针。

一级索引：对 20KB 小文件只有 5 个物理块大小，所以只需一个专用物理块来作索引块，用来保存文件的各个物理块地址。对于 20MB 大文件共有 5K 个物理块，由于链接索引的每个索引块只能保存 (1K – 1) 个文件物理块地址（另有一个表目存放下一个索引块指针），所以它需要 6 块专用物理块来作链接索引块，用于保存文件各个的物理地址。

二级索引：对大小文件都固定要用二级索引，对 20KB 小文件，用一个物理块作一级索引，用另一块作二级索引，共用二块专用物理块作索引块，对于 20MB 大文件，用一块作一级索引，用 5 块作二级索引，共用六块专用物理块作索引块。

LINUX 的混合分配：对 20KB 小文件只需在文件控制块 FCB 的 i_addr[15]中使用前 5 个表目存放文件的物理块号，不需专用物理块，对 20MB 大文件，FCB 的 i_addr[15]中使用前 12 个表目存放大文件前 12 块物理块块号 (48K)，用一级索引块一块保存大文件接着的 1K 块块号 (4M)，剩下还有不到 16M，还要用二级索引存大文件以后的块号，二级索引使用一级索引 1 块，二级索引 4 块 (因为 4KB × 1K × 4=16M)。总共也需要 6 块专用物理块来存放文件物理地址。

(3) 连续分配：为读大文件前面和后面信息都需先计算信息在文件中相对块数，前面信息相对逻辑块号为 5.5K / 4K=1 (从 0 开始编号)，后面信息相对逻辑块号为 (16M + 5.5K) /4K=4097。

再计算物理块号 = 文件首块号 + 相对逻辑块号, 最后化一次盘 I/O 操作读出该块信息。

链接分配: 为读大文件前面 5.5KB 的信息, 只需先读一次文件头块得到信息所在块的块号, 再读一次第 1 号逻辑块得到所需信息, 共 2 次。而读大文件 16MB + 5.5KB 处的信息, 逻辑块号为 $(16M + 5.5K) / 4092 = 4107$, 要先把该信息所在块前面块顺序读出, 共化费 4107 次盘 I / O 操作, 才能得到信息所在块的块号, 最后化一次 I/O 操作读出该块信息。所以总共需要 4108 次盘 I / O 才能读取 (16MB+5.5KB) 处信息。

链接索引: 为读大文件前面 5.5KB 处的信息, 只需先读一次第一个索引块得到信息所在块的块号, 再读一次第 1 号逻辑块得到所需信息, 共化费 2 次盘 I / O 操作。为读大文件后面 16MB+5.5KB 处的信息, $(16MB+5.5KB)/(4KB \times 1023) = 4$, 需要先化 5 次盘 I / O 操作依次读出各索引块, 才能得到信息所在块的块号, 再化一次盘 I/O 操作读出该块信息。共化费 6 次盘 I / O 操作。

二级索引: 为读大文件前面和后面信息的操作相同, 首先进行一次盘 I / O 读第一级索引块, 然后根据它的相对逻辑块号计算应该读第二级索引的那块, 第一级索引块表目号=相对逻辑块号 / 1K, 对文件前面信息 1 / 1K = 0, 对文件后面信息 4097 / 1K = 4, 第二次根据第一级索引块的相应表目内容又化一次盘 I / O 读第二级索引块, 得到信息所在块块号, 再化一次盘 I / O 读出信息所在盘块, 这样读取大文件前面或后面处信息都只需要 3 次盘 I / O 操作。

LINUX 混合分配: 为读大文件前面 5.5KB 处信息, 先根据它的相对逻辑块号, 在内存文件控制块 FCB 的 i_addr 第二个表目中读取信息所在块块号, 而只化费一次盘 I / O 操作即可读出该块信息。为读大文件后在 (16MB + 5.5KB) 信息, 先根据它的相对逻辑块号判断要读的信息是在二级索引管理范围内, 先根据 i_addr 内容化一次盘 I / O 操作读出第一级索引块, 再计算信息所在块的索引块号在第一级索引块的表目号为 $(4097-12-1024) / 1024 = 2$, 根据第一级索引块第 3 个表目内容再化费一次盘 I / O 操作, 读出第二级索引块, 就可以得到信息所在块块号, 最后化一次盘 I / O 读出信息所在盘块, 这样总共需要 3 次盘 I / O 操作才能读出文件后面的信息。
