Cheatsheet

一、进程和线程

1. 进程和线程的各种状态转换
    1. 新生状态 (new): 表示一个进程刚刚被创建出来, 还未完成初始化, 不能被调度运行. 经过初始化后, 进程进入预备状态.
    2. 预备状态 (ready): 该状态表示进程可以被调度执行, 但还未被调度器选择. 在被调度器选择执行后, 进程进入运行状态.
    3. 运行状态 (running): 该状态表示进程正在 CPU 上运行. 当一个进程执行一段时间后, 调度器可以选择中断它并放回调度队列, 进而进入预备状态. 如果进程需要等待一些外部事件, 例如某个 I/O 请求的完成, 就可以放弃 CPU 进入阻塞状态. 当进程运行结束, 它就会进入终止状态.
    4. 阻塞状态 (blocked): 该状态表示该进程需要等待外部时间, 例如某个 I/O 的完成, 暂时无法被调度. 当该进程等待的外部事件完成后, 就会进入预备状态.
    5. 终止状态 (terminated): 该状态表示进程已经完成了执行, 且不会再被调度.
2. 系统调用
    1. 系统调用是一种特殊的异常, 是操作系统为用户程序提供服务的一种手段. 内核实现系统调用是以一个软中断的形式, 即陷阱指令, 如 i386 的 `int 0x80` 指令实现的.
    2. 系统调用是操作系统提供给用户访问内核空间的特殊接口, 其对应的服务例程属于系统程序, 在内核态运行; API 函数是应用程序接口, 为应用程序开发者提供便携的功能支持. 区别: 系统调用必然访问内核态, 而 API 函数强调的是如何用接口来获得所需服务, 部分 API 函数可以在用户态运行. 联系: 一个 API 函数根据是否需要访问内核态, 可能不需要或需要一个到多个系统调用来实现特定功能.
3. 进程控制块 (PCB)
    1. 在内核中, 每个进程都通过一个数据结构来保存它相关的状态, 如它的进程标识符 PID、进程状态、虚拟内存状态、打开的文件等, 这个数据结构称为进程控制块 PCB.
4. 多道程序设计
    1. 上下文切换
        1. 将当前处理器的寄存器上下文保存到当前进程的系统级上下文的现场信息中;
        2. 将新进程系统级上下文中的现场信息作为新的寄存器上下文恢复到处理器的各个寄存器中;
        3. 将控制转移到新进程执行.
    2. CPU 利用率: 用于真实计算的 CPU 时间比例
        1. 假设等待 I/O 时间与停留内存时间之比为 p, 则 n 个独立进程的 CPU 利用率为 $1-p^n$.

5. 线程控制块 (TCB)

二、调度

1. 调度的指标
    1. 吞吐: 系统每小时完成的作业数量.
    2. 周转时间: 从一个批处理作业提交时刻开始, 直到该作业完成时刻为止的统计平均时间.
    3. CPU 利用率: CPU 利用率常常用于对批处理系统的度量.
    4. 响应时间: 对于交互式系统来说很重要, 即从发出命令到得到响应之间的时间.
    5. 均衡性: 用户对一件事情需要的时间的固有看法.
    6. 截止时间: 实时系统必须要满足截止时间.
    7. 可预测性: 涉及多媒体的实时系统, 人的耳朵和眼睛十分灵敏, 所以进程调度必须是高度可预测和有规律的.
2. 批处理系统的调度
    1. 先来先服务 (FCFS, FIFO): 当新作业进入, 排到队尾; 当进程被堵塞, 就接着运行队头任务; 当阻塞进程变为就绪时, 进入队尾.
    2. 最短作业优先 (SJF)
    3. 最短剩余时间优先 (SRTN)
3. 交互式系统中的调度
    1. 轮转调度 (RR)
    2. 优先级调度 (PS)
    3. 多级反馈队列 (MLFQ)
        1. 短任务拥有更高的优先级. MLFQ 会为每个任务设置任务的最大运行时间, 如果超过了最大时间, 就会将该任务的优先级减一.
        2. 低优先级的任务采用更长的时间片.
        3. 定时地将所有任务地优先级提升到最高, 保证不会有饥饿.
        4. Game the system: 进程在时间片用完之前, 调用一个无用的 I/O 操作, 主动放弃 CPU, 从而保持在高优先级, 占据更多的 CPU 时间. 我们可以追踪记录进程在一个很大的时间周期 (几个时间片) 中运行的总时间, 然后给每个优先级赋予一个最大的 CPU 总时间.
4. 实时系统的调度
    1. 准入控制: 假设有 m 个任务, 其中第 i 个任务的运行时间记为 C_i, 周期记为 T_i, 任务在单位时间内对 CPU 的利用率则为 C_i/T_i, 则总 CPU 利用率 $U = \sum_{i=1}^{m} C_i/T_i \leq 1$.
    2. 单调速率调度 (RM)
        1. 速率指的是任务的到达速率, 它是任务周期的倒数, 即 $1/T$.
        2. 需要预测任务的周期 T, 并且通过周期静态地为每个任务分配一个优先级, 任务的周期越短, 则优先级越高, RM 策略还支持抢占调度, 高优先级任务可以强制低优先级任务执行.

    3. 在所有静态优先级的实时调度策略中, RM 策略是最优的.
    4. RM 策略需要调度 N 个任务时, 最坏情况下的 CPU 利用率为 $N * (2^{1/N} - 1)$. 两个任务约为 83%, 无限多个任务时约为 69%.
3. 最早截止期限优先 (EDF)

三、内存管理

1. 内存管理: 管理所有和内存相关的操作和保存在主存中的资源, 使得多个进程能够使用主存和资源.
    1. 记录所有被用到的内存;
    2. 动态分配内存;
    3. 权限管理和内存保护;
    4. 回收不需要的内存;
    5. 最大化内存使用率和系统处理能力.
2. 地址空间: 就像进程概念创造了一类抽象的 CPU 以运行程序一样, 地址空间为程序创造了一种抽象的内存. 地址空间时一个进程可用于寻址内存的一套地址集合.每个进程都有一个自己的地址空间, 并且这个地址空间独立于其他进程的地址空间.
3. 动态重定位: 使用基址寄存器和界限寄存器将每个进程的地址空间映射到物理内存的不同部分.
4. 连续内存分配:
    1. 首次适配 (first fit): 沿着链表搜索, 直到找到一个空闲区.
    2. 最佳适配 (best fit): 搜索整个链表, 找出能够容纳进程的最小空闲区.
    3. 最差适配 (worst fit): 总是分配最大的可用空闲区.
    4. 快速适配 (quick fit): 为常用的空闲区维护单独的链表, 类似按照大小对空闲区链表排序, 以便提高最佳适配算法的速度.
5. 内部碎片和外部碎片
    1. 内部碎片: 分页时一个页面过大, 正文段、数据段和堆栈区很可能不会恰好装满整数个页面, 平均的情况下, 最后一个页面有一半是空的, 多余的空间就被浪费掉了, 这种浪费称为内部碎片.
    2. 外部碎片: 与页相比, 段是不定长的, 多次替代和调换后, 就会形成空闲区, 这种现象称为外部碎片. 这种现象可以通过内存紧缩来解决.
6. 虚拟地址: 由程序产生的地址产生的地址称为虚拟地址, 它们共同构成了一个虚拟地址空间. 没开启虚拟内存时, 虚拟地址就是物理地址; 开启了虚拟内存时, 虚拟地址不是直接被送到内存总线, 而是被送到内存管理单元 (MMU), 由 MMU 负责把虚拟地址映射为物理内存地址.
7. 转换检测缓冲区/快表 (TLB):
    1. 通常在 MMU 中, 包含少量的表项, 一般不会超过 256 个, 每个表项包括有效位、虚拟页面号、修改位、保护位和页框号.
    2. 工作时, 硬件将虚拟页号与 TLB 中所有表项并行比较并匹配, 如果发现了有效的表项, 则就不用访问页表. 如果 MMU 检测到没有有效匹配项, 则会查找页表, 然后从

TLB 中淘汰一个表项.
3. TLB 一致性: 一旦上下文切换, PTBR 寄存器发生变化, 就要冲刷一次 TLB.

8. 有效访问时间 (Effective access time): $\mathrm{EAT} = (\varepsilon + t)\alpha + (\varepsilon + 2t)(1 - \alpha)$
   1. 命中率 (α): 页号在 TLB 中找到的次数的百分比.
   2. 内存访问时间 (t).
   3. TLB 搜索时间 (ε).
9. 分段
   1. 段表结构: 每个进程都有段表, 其中每一项均包含
      1. 段基址 (Segment Base): 段的开始物理地址.
      2. 段限制 (Segment Limit): 段的空间大小.
      3. 有效位 (Validation Bit): 段是否有效.
      4. 权限: 段的读写和执行权限.
   2. 分段地址转换
      1. 段表基址寄存器 (STBR): 指向段表的其实位置.
      2. 段表长度寄存器 (STLR): 指示了段表的段的数目.
      3. 物理地址 = 逻辑地址 + 段基址.
10. 分页
    1. 页框: 虚拟地址空间按照固定大小划分为被称为页面的若干单元, 在物理内存中对应的单元称为页框.
    2. 页表项: 不同计算机的页表项大小可能不一样, 但是 32 位是一个常用的大小. 常包括页框号、保护位、修改位 (脏位)、访问位和禁用高速缓存位. 修改位和访问位对页置换算法很有用, 禁用高速缓存位对内存映射 I/O 很有用.
    3. 多级页表: 32 位的虚拟地址常常被分为 10 位的 PT1 域、10 位的 PT2 域和 12 位的偏移量, 这样能够解决页表过大的问题.
    4. 倒排页表: 内存中的每个页框对应一个表项, 而不是每个虚拟页面对应一个表项. 但是这样会导致虚拟地址到物理地址会很困难, 解决这种困难的方法是 TLB 和散列表.
11. 虚拟内存
    1. 请求调页: 当应用程序申请分配内存时, 操作系统可以选择将新分配的虚拟页标记为已分配但未映射到物理内存的状态, 而不必为这个虚拟页分配对应的物理页. 当应用程序访问这个虚拟页的时候, 会触发缺页异常, 这时候才真正为其分配物理页.
    2. 缺页异常
       1. 硬件陷入内核, 在堆栈中保存 PC.
       2. 启动一个汇编代码历程保存通用寄存器和其他易失信息.
       3. 发现缺页中断, 尝试就找需要哪个虚拟页.
       4. 检查虚拟页地址是否有效, 然后检查是否有空闲页框, 如果没有, 则执行页面置换算法来找一个页面来淘汰.
       5. 如果选择的页框是脏的, 则安排该页写回磁盘, 并由于 I/O 操作而发生一次进程调度.

    6. 一旦页框干净, 则查找所需页面在磁盘上的地址, 通过 I/O 操作装入内存, 同样发生进程调度.
    7. 磁盘中断发生时, 表示该页已经载入, 页表也可以更新, 以反映它的位置.
    8. 恢复发生缺页中断以前的状态, PC 重新指向这条指令.
    9. 调度引发缺页中断的进程.
    10. 恢复寄存器和其他状态信息, 返回用户空间继续执行.
12. 页面置换算法
    1. 最优 (Optimal): 通过模拟执行代码以了解所有页面的使用顺序, 总是替换掉最晚将被使用的页面.
    2. 最近未使用 (NRU): 读位 R 定时被置 0, 并且有一个写位 W, 然后根据 RW 将页面分为 00 到 11 这四类, 然后随机地从类编号最小地非空类中挑选一个页面淘汰.
    3. 先进先出 (FIFO): 最早被载入的页面最早被淘汰.
    4. 二次机会: 检查最老页面的 R 位, 如果 R 位是 0, 则立刻置换掉; 如果 R 位是 1, 则清零, 然后放到链表尾端, 就好像刚装入一样, 然后继续搜索.
    5. 时钟算法: 用循环链表来实现二次机会中的链表, 以减少将页面放入链表尾端的时间.
    6. 最近最少用 (LRU): 置换未使用时间最长的页面. 一个简单的实现方式是硬件有一个计算已经执行了多少条指令的计数器 C, 每次访问内存后, 将当前的 C 值保存到当前页面对应页表项中. 一旦发生缺页中断, 就淘汰掉值最小的一个页面.
    7. 最不常用 (NFU): 一种模拟 LRU 的算法, 将每一个页面与一个软件计数器相关联, 计数器初值为 0. 每次时钟中断时, 将页面的 R 值加到计数器中. 然而, 要加入老化机制, 在 R 位被加入前, 先将计数器右移一位, 然后将 R 位加到计数器最左端的位, 而不是最右端的位. 发生缺页中断时, 将置换计数器最小的页面.
    8. 缺页中断率 (Page Fault Frequency) 和页框分配
        1. 页替换有两种策略分类, 一是全局页替换, 考虑所有的进程; 二是局部页替换, 只考虑当前进程.
        2. 为了使用全局页替换算法, 必须保证每个进程都有一定的页面, 根据进程大小按比例为其分配页面也是可能的, 但是需要动态管理内存分配, 其中一种办法就是 PFF 算法. 它指出了何时增加或减少分配给某一个进程的页面.
        3. 测量缺页中断率的方法是直截了当的: 计算每秒的缺页中断数, 也可能取前几秒的平均. 如果中断率低于某个值, 进程就失去页框; 如果中断率高于某个值, 进程就得到页框.
13. 工作集
    1. 工作集: 一个进程当前正在使用的页面的集合称为它的工作集. 它是随着时间变化而发生变化的. 定义上, 工作集就是最近 k 次内存访问所使用过的页面的集合.
    2. 颠簸: 如果每执行几条指令程序就发生一次缺页中断, 那么就称这个程序发生了颠簸.
    3. 访问局部性: 在进程运行的任何阶段, 它都只访问较少的一部分页面.
    4. 工作集置换算法: 当发生页中断时, 淘汰一个不在工作集中的页面.

四、同步

1. 竞争条件: 两个或多个进程读写某些共享数据, 而最后的结果取决于进程运行的精确时序, 称为竞争条件.
2. 临界区: 我们把共享内存进行访问时的程序片段称作临界区.
    1. 安全性: 任何两个进程不能同时处于其临界区.
    2. 不应该对 CPU 速度和数量做任何假设.
    3. 临界区外运行的进程不得阻塞其他进程.
    4. 活性: 不得使进程无期限等待进入临界区.
3. 互斥
    1. 忙等待: 连续测试一个变量直到某个值出现为止, 称为忙等待. 用于忙等待的锁, 称为自旋锁.
    2. 皮特森算法: 只需要两个全局变量 `turn` 和 `interested` 来分别指示轮到的进程 (其实感觉更应该说是轮到谁等待) 和对临界区感兴趣的进程.

        ```
        void enter_region(int process) {
            int other = 1 - process;
            interested[process] = TRUE;
            turn = process;
            while (turn == process && interested[other] == TRUE);
        }

        void leave_region(int process) {
            interested[process] = FALSE;
        }
        ```

    3. TSL 指令: 测试并加锁, `TSL RX, LOCK` 可以原子地将 `LOCK` 读到 `RX` 寄存器中, 并在 `LOCK` 处写一个非零值. 类似地还有原子地交换两个位置内容的指令 XCHG.
    4. 优先级反转: 由于低优先级进程 L 占据了临界区, 导致高优先级进程 H 一直在临界区外忙等待, 让 L 不能被调度, 也就不能出临界区. 这种情况有时候称为优先级反转.
4. 条件变量
    1. 生产者消费者问题:

        ```
        pthread_mutex_lock(&mutex);
        while (buffer ?= 0) pthread_cond_wait(&cond, &mutex);
        buffer = ?;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
        ```

5. 信号量
    1. P (down): 对一个信号量尝试减 1, 如果等于零 0, 则无法减 1, 只能将进程睡眠, 此时 down 操作仍未结束, 会等待有一个进程加 1 后继续.
    2. V (up): 对一个信号量加 1, 如果有进程在这个信号量上睡眠, 则会随机唤醒一个进程让其继续 down 操作.

3. 通过信号量实现互斥锁和条件变量: 设定信号量初值为 `1` 即可实现加锁; 设定信号量初值为 `0` 即可实现条件变量.
4. 通过互斥锁和条件变量实现信号量:

```
void P(semaphore *s) {
    mutex_lock(&(s->mutex));
    while (s->val == 0) {
        wait(&(s->cond), &(s->mutex));
    }
    s->val--;
    mutex_unlock(&(s->mutex));
}
void V(semaphore *s) {
    mutex_lock(&(s->mutex));
    s->val++;
    signal(&(s->cond));
    mutex_unlock(&(s->mutex));
}
```

5. 生产者消费者问题: 使用 `full`, `empty` 和 `mutex` 三个信号量来解决. `full` 初值为 0, `empty` 初值为缓冲区中槽数量, `mutex` 初值为 1.

```
// producer
down(&empty);
down(&mutex);
insert_item(item);
up(&mutex);
up(&full);
// consumer is similar
```

6. 管程
    1. 定义: 一个管程是一个由过程、变量以及数据结构等组成的一个集合, 且任一时刻管程中只能有一个活跃进程. (可以简单理解为由编译器管理的互斥量或二元信号量).
    2. 自身阻塞: 引入条件变量和两个相关操作 `wait` 和 `signal`. 而 `signal` 后对进程的处理, 则引出了三种不同的语义.
    3. Hoare: 让新唤醒进程运行, 挂起发信号进程. 因此需要三个队列 `signal_queue`, `wait_queue` 和 `ready_queue`, 因此是实现起来最复杂的.
    4. Hansen: 执行 `signal` 的进程必须立刻退出管程, 即 `signal` 是管程的最后一条语句. 因为 Hansen 语义为简单, 只需要 `wait_queue` 和 `ready_queue`.
    5. Mesa: 让发信号进程继续运行, 直到发信号进程退出管程后, 才允许等待的进程开始运行. 类似 Hansen 语义, 只不过没有了 `signal` 是末尾语句的限制, 也只需要 `wait_queue` 和 `ready_queue`, 因此被广泛地实现. 但是因此也要注意一个问题, 线程被唤醒时, 必须检查 `condition` 是否满足, 也就是要使用 `while` 而不是 `if`.
7. 生产者消费者问题: 同上
8. 读者写者问题

1. 读写锁: 多个读者可以同时进入临界区; 但是如果有一个写者在临界区内, 则必须保证临界区内没有其他读者和写者.
2. 偏向问题: 在某一时刻, 已经有一些读者在临界区中. 此时有一个写者和一个读者同时申请进入临界区, 那么应该先让写者进入还是读者进入临界区. 优先读者可能导致写者饿死, 优先写者可能会减少读者并行性而减少效率. 我们也可以使用一个队列来依次服务, 这时则称为相对公平.
3. 读者偏向: 实现起来比较简单, 只需要使用两个锁 `reader_lock` 和 `writer_lock`, 维持一个读者计数 `reader_cnt`, 并且在第一个读者进入时锁上写者锁, 在最后一个读者退出时解锁写者锁即可.
4. 写者偏向: 相对复杂, 需要 `reader_cnt`, `has_writer`, `lock`, `reader_cond` 和 `writer_cond`. 读者需要根据 `has_writer` 来使用 `writer_cond` 阻塞自身, 写者需要在仍有读者时用 `reader_cond` 来阻塞自身.
5. 相对公平: 更加复杂了, 通过信号量 `Line` 来保持 FIFO 队列的唤醒顺序, 并且使用各种各样的变量来实现同步, 这里就省略吧, 反正也记不住.

9. Read-Copy-Update (RCU)
    1. RCU 主要针对的数据对象是链表, 目的是提高遍历读取数据的效率, 为了达到目的使用 RCU 机制读取数据的时候不对链表进行耗时的加锁操作. 这样在同一时间可以有多个线程同时读取该链表, 并且允许一个线程对链表进行修改 (修改的时候, 需要加锁).
    2. 保证读取链表的完整性. 新增或者删除一个节点, 不至于导致遍历一个链表从中间断开. 但是 RCU 并不保证一定能读到新增的节点或者不读到要被删除的节点.
    3. 订阅-发布机制: 在读取过程中, 另外一个线程插入了一个新节点, 而读线程读到了这个节点, 那么需要保证读到的这个节点是完整的. 这里涉及到了发布-订阅机制 (Publish-Subscribe Mechanism).

        ```
        // 发布
        rcu_assign_pointer(&NodeA->Next, NodeC);
        // 订阅
        read = rcu_dereference(&Node->Next);
        ```

    4. 宽限期: 在读取过程中, 另外一个线程删除了一个节点. 删除线程可以把这个节点从链表中移除, 但它不能直接销毁这个节点, 必须等到所有的读取线程读取完成以后, 才进行销毁操作. RCU 中把这个过程称为宽限期(Grace period).

        ```
        // 用来保持一个读者的 RCU 临界区
        rcu_read_lock()
        rcu_read_unlock()

        // RCU 的核心所在, 它挂起写线程, 等待读者都退出后释放老的数据
        synchronize_rcu()
        ```

10. 哲学家就餐问题
    1. 死锁: 同时拿起左叉, 但是不放下, 就会导致死锁.

2. 饿死: 同时拿起左叉, 同时放下, 同时又拿起右叉, 同时又放下, 循环进行, 就会导致饿死.
3. 最简单的解决方法: 破除依赖, 让一个哲学家与众不同, 他先拿右叉, 而不是像其他人那样拿起左叉.
4. 一个相对复杂的解决方法:

```
void take_forks(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);  // 可能阻塞, 等待邻居主动递叉子
}
void put_forks(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
void test(i) {
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);  // 拿到叉子了, 解除阻塞
    }
}
```

五、死锁

1. 死锁四个必要条件
    1. 互斥: 每个资源要么分配给了一个进程, 要么是可用的.
    2. 持有并等待: 已经得到了某个资源的进程可以再次申请新资源.
    3. 不可抢占: 已经分配给一个进程的资源不能强制性地被抢占.
    4. 环路等待: 死锁发生时, 一定有两个或两个以上的进程组成了一条环路, 环路中的每个进程都在等待下一个进程占有的资源.
2. 资源分配图
    1. 每种类型一个资源: 用圆形表示进程, 用方形表示资源. 从资源到进程的箭头代表进程占有一个资源; 从进程到资源的箭头代表进程请求一个资源.
    2. 每种类型多个资源: 用圆形表示进程, 用方形表示资源, 方块里面的点数代表资源数.
3. 死锁检测
    1. 每种类型一个资源: 用环路检测算法, 即 DFS, 依次将每一个节点作为一棵树的根节点, 进行深度优先搜索.
    2. 每种类型多个资源: 基于资源分配矩阵, 请求矩阵和可用向量的算法检测. (和银行家算法基本就是一模一样?)
        1. 寻找一个没有标记的进程 P_i, 对于它而言 $R_i \leq A$.

        2. 如果找到了 P_i, 则将 C_i 加到 A 中, 标记该进程, 转到第 1 步.
        3. 如果没找到 P_i, 则算法终止. 所有没有标记过的进程都是死锁进程.
4. 死锁避免
    1. 安全状态: 存在一个分配序列使得所有的进程都能完成.
    2. 不安全状态: 不存在一个分配序列使得所有的进程都能完成. 理论上来说不安全状态不是死锁, 因为依然有进程能运行一段时间. 但是实际上应该说有部分程序有死锁, 而另一部分程序没有死锁.
    3. 银行家算法: 对每一个请求进行检查, 检查如果满足这一请求是否会达到安全状态. 若是, 则满足该请求; 若不是, 则推迟对这一请求的满足.
        1. 寻找一个没有标记的进程 P_i, 对于它而言 $R_i \leq A$.
        2. 如果找到了 P_i, 则将 C_i 加到 A 中, 标记该进程, 转到第 1 步.
        3. 如果没找到 P_i, 则算法终止. 如果所有进程都被标记过, 也就是没有死锁, 则这个状态是安全的.
5. 死锁预防: 破坏四个条件中随便一个.

六、文件系统

1. 文件系统的功能
    1. 三个基本要求
        1. 能够存储大量信息.
        2. 使用信息的进程终止时, 信息依然存在.
        3. 能使多个进程并发地访问有关信息.
    2. 命名: 命名文件和目录
        1. 翻译名字并找到对应磁盘块偏移.
    3. 文件访问: 打开、读取、写入和其他操作.
    4. 磁盘管理: 公平且有效地使用磁盘.
        1. 给文件分配空间.
        2. 追踪记录空闲空间.
        3. 快速访问文件.
    5. 权限管理: 保证不同用户对文件有相应的访问权限.
    6. 可靠性: 不能损失文件数据.
2. 文件
    1. 文件的命名和属性
        1. 文件命名: 一个重要的文件系统抽象. 可以使得用户无需了解信息在磁盘上是如何存储和使用的. 具体的命名规则与具体的系统有关.
        2. 文件类型: 常见的有普通文件、目录、字符特殊文件和块特殊文件. 每个操作系统都必须至少识别一种文件类型: 它自己的可执行文件类型. 普通文件一般可以通过文件拓展名和魔数来进一步划分和识别类型.
        3. 文件属性: 操作系统一般还会保存其他和文件有关的信息, 它们被称为文件属性, 有时候也被称为元数据. 例如文件创建日期和时间, 文件大小等. 还有一些

文件属性和文件保护相关.

2. 文件的顺序访问模式和随机访问模式
    1. 顺序访问: 进程可以从头到尾地按顺序读取文件的所有内容, 但是不能跳过一些内容, 也不能不按顺序读取. 这种访问模式对磁带这类存储介质很友好.
    2. 随机访问: 可以不按顺序地读取文件中的字节或记录, 或者按照关键字而不是位置来访问记录.
3. 与文件相关的操作: 常见的有 `create`, `delete`, `open`, `close`, `read`, `write`, `append`, `seek`, `get/set attributes`, `rename`.
4. inode
5. 打开文件表 (Open-File Table)
    1. 为了减少解析文件名 (在目录中搜索文件名和检查文件权限) 的开销, 操作系统会在进程中维护一个文件描述符表 (FDT), 包含了进程打开的所有文件. 同时操作系统还会有一个系统级的打开文件表 (OFT) 和一个系统级的 inode 表.
    2. 打开文件后, 会返回一个文件描述符 (file descriptor), 作为打开文件表的索引.
    3. `fd_flags` 保存在进程的 FDT 中, `file_offset` 和 `status_flags` 保存在系统级的 OFT 中 (`fork` 之后的进程有着相同的 `file_offset`), 文件类型等信息保存在 inode 表中.
    4. 不同的进程可能同时打开了同一个文件, 然后 OFT 中就会出现两条记录, 但是都对应同一个 inode 表项.

3. 目录
    1. 目录的层级结构
    2. 硬链接和符号链接
        1. 硬链接: 链接技术允许多个目录中出现同一个文件, 即多个目录项指向同一个 inode. 这种类型的链接增加了 inode 节点计数器的计数, 有时候称为硬链接. 相当于是 inode 节点的别名.
            1. 一般不能硬链接到一个目录. 否则被硬链接到的目录会存在多个父目录, 则此时 `..` 不唯一.
        2. 软链接 (符号链接): 一个文件名指向另一个文件的小文件. 相当于是文件名的别名. 符号链接自由度很高, 但是可能会导致文件系统效率低下.
4. 文件和目录的角色: 两个关键的抽象.
    1. 文件: 可以读写的字节序列.
    2. 目录: 一系列文件和目录的列表.
5. 文件系统布局
    1. 多数磁盘划分为一个或多个分区, 每个分区有一个独立的文件系统, 这样的分区也被称为卷.
    2. 磁盘的 0 号扇区称为主引导记录 (MBR), 用以引导计算机. 在 MBR 结尾是分区表, 该表给出了每个分区的起始和结束地址. 表中的一个分区被标记为活动分区, 活动分区的第一个块被称为引导块 (boot block), MBR 会执行它, 然后装载该分区中的操作系统. 为了统一起见, 所有的分区都从一个引导块开始, 即使它不包含一个操作系统也是如此.

3. 在引导块之后, 磁盘分区的布局是随着文件系统不同而变化的. 不过一般都有一个超级块, 包含了文件系统的所有关键参数, 例如用以识别文件系统类型的魔数、文件系统中块的数量和其他重要的管理信息.
4. 一般来说都是: `[S][i][d][...inodes...][...data...]`, 即超级块 + inode 管理 + data 管理 + inode 块 + data 块.

6. 文件存储的实现
    1. Contiguous: 将每个文件作为一连串的数据块存储在磁盘上. 优点是效率高, 缺点是容易形成磁盘碎片, 所以一般用于一次性的存储介质, 例如各种光学光盘.
    2. Linked List: 为每个文件构造磁盘块链表, 也就是让每个块的第一个字作为指向下一块的指针.
    3. FAT: 在 Linked List 的基础上, 取出每个磁盘块的指针字, 把它们放在内存的一个表中, 内存中的这样一个表格被称为文件分配表 (FAT).
    4. Indexed (inode 多级索引): 给每个文件赋予一个称为 inode 的数据结构, 其中列出了文件属性和文件块的磁盘地址. 然后使用多级的索引来增加存储的数据块索引数目.
        1. 例如有 15 个指针.
            1. 前 12 个指针的数据块大小为 12 * 4KB = 48KB.
            2. 后 3 个指针是间接块.
                1. 一级间接块: 1K * 4KB = 4MB.
                2. 二级间接块: 1K * 1K * 4KB = 4GB.
                3. 一级间接块: 1K * 1K * 1K * 4KB = 4TB.
7. 目录的实现
    1. 目录项中包含的信息: 操作系统利用路径名找到相应的目录项, 目录项提供了查找文件磁盘所需的信息.
        1. Contiguous: 整个文件的磁盘地址.
        2. Linked List 或 FAT: 第一个数据块的编号.
        3. Indexed: inode 号.
    2. 文件属性的存储位置: 保存在目录项中, 或者保存在 inode 块中. 后者的优点更多.
    3. 不同长度文件名的处理
        1. 最简单的方案: 给定一个长度限制, 例如 255 个字符. 但是这样也会浪费大量的目录空间.
        2. 在目录项中保存可变长字符串, 并且进行基础的对齐. 缺点是在删除文件后会产生可变长度的空隙.
        3. 让目录项有固定的大小, 然后将字符串保存在堆中. 优点是不仅目录项长度固定, 字符串也不用对齐了. 缺点是仍然要管理堆中的字符串.
    4. 在目录中查找文件
        1. 线性地从头到尾搜索.
            1. 缺点是对于长目录来说会很花时间.
        2. 对每个目录进行哈希处理, 构造哈希表.
            1. 用链表处理冲突.

        2. 可以更快地找到文件, 但是管理起来会更复杂.
    3. 不用哈希表, 而是用 B-tree.
    4. 也可以对查找的结果进行缓存.
8. 磁盘空闲空间管理
    1. 不同数据块大小对文件系统的影响
        1. 过大: 会浪费许多磁盘空间来存储小文件.
        2. 过小: 会浪费很多时间来读一个大文件, 因为它被分成了很多小块.
    2. 记录空闲块
        1. Bit Map
            1. 查找连续的块比较简单.
            2. 需要额外的存储空间. 块大小为 4KB, 磁盘空间大小为 1TB, 则需要 256MB 的空间来存储.
        2. Free List
            1. 因为是使用空闲块来存储下一个空闲块的指针, 所以几乎不需要额外的空间.
            2. 分配会较为困难.
                1. 获取连续的空间并不容易.
                2. 分配空闲块需要额外的磁盘操作.
                3. 分配多个块时需要遍历链表.
            3. 计数策略
                1. 记录连续块的起始块位置和块的个数.
                2. 在碎片较多时效率低下.
            4. 组策略
                1. 每个块都记录尽可能多的磁盘块位置.
                2. 不再需要遍历链表.
                3. 缺点是使用的空间比 Bit Map 还多得多.
9. 文件系统的性能
    1. 缓存
    2. 快速文件系统
        1. 让文件系统的结构和分配策略具有 "磁盘意识".
        2. FFS 将磁盘划分为一些分组, 称为柱面组.
        3. 将文件数据块分配到和 inode 相同的分组, 将同一目录的所有文件, 放在所在目录的柱面组中.
10. 文件系统一致性
    1. 崩溃场景: 理论上应该一次性地写入数据块、inode 和位图, 但是实际上一般不可能, 就使得崩溃时只完成了部分任务, 导致文件损坏.
    2. 文件系统一致性检查 (fsck):
        1. 超级块: 检查超级块是否合理, 例如确保文件系统大小大于分配的块数.
        2. 空闲块: 扫描 inode 以生成一个正确版本的 inode bitmap, 然后进行比对, 信任新生成的 bitmap.

3. inode 状态: 检查每个 inode 是否存在损害, 如果 inode 字段存在问题, 则将清除该 inode.
4. inode 链接: 验证每个 inode 的链接数, 如果不匹配, 修正 inode 中的计数; 如果已分配的 inode 没找到任何目录引用它, 则会将其移动到 lost + found 目录.
5. 重复: 检查是否有两个 inode 引用同一个数据块. 如果有, 要么清除其中一个, 要么复制一个新的块用以指向.
6. 坏块: 检查是否有超出分区范围的指针, 然后将其删除.
7. 目录检查: 确保 `.` 和 `..` 是前面的条目, 并且每个目录条目的 inode 已分配, 并确保整个层次结构中没有目录被引用超过一次.

3. 日志文件系统
    1. LFS 永远不会覆写现有数据, 而是始终按顺序写入空闲位置.
    2. 用户写入数据时, 不仅是数据被写入磁盘, 还有其他需要更新的元数据 inode 和一段 inode map 也写入磁盘.
    3. LFS 在磁盘有一个固定的检查点区域 (checkpoint region, CR). 它将指向最新的 inode map.

11. 虚拟文件系统: 通过上层的 POSIX 接口和下层的 VFS 接口实现虚拟文件系统. 为每个打开的文件维护一个 v 节点, 然后通过 VFS 接口的功能指针调用具体文件系统的功能.

七、设备管理

1. 设备控制器: I/O 设备一般由机械部件和电子部件两部分组成. 电子部件被称作设备控制器或适配器.
2. 设备类型
    1. 块设备: 把信息存储固定在固定大小的块中, 每个块都有自己的地址.
    2. 字符设备: 以字符为单位发送或接受一个字符串, 而不考虑任何块结构, 而且也是不可寻址的.
3. 设备驱动程序: 每个连接到计算机上的 I/O 设备都需要某些设备特定的代码来对其进行控制, 这样的代码称为设备驱动程序.
4. 端口映射 I/O: 每个控制器有几个寄存器来与 CPU 通信, 如果让每个控制寄存器分配一个 I/O 端口号, 就形成了 I/O 端口空间.
5. 内存映射 I/O: 将所有控制寄存器映射到内存空间中. 每个控制寄存器被分配唯一的一个内存地址, 并且不会有内存被分配这一地址.
6. 直接存储器存取 (DMA):
    1. 使用步骤
        1. CPU 通过设置 DMA 控制器的寄存器对它进行编程, 即地址、计数和控制这三类寄存器.
        2. DMA 控制器在总线发出一个读请求到磁盘控制器而发起 DMA 传送.
        3. 在另一个标准总线周期将数据传送到内存.
        4. 磁盘控制器在总线发出一个应答信号到 DMA 控制器.

    5. DMA 控制器步增要使用的地址, 并且步减字节计数. 如果计数仍大于 0, 则跳回第 2 步.
    6. 计数归零后, 中断 CPU 让 CPU 知道传送现在完成了.
2. 总线操作
    1. 周期窃取: 总线以每次一字模式操作, DMA 控制器请求传送一字并得到这个字, CPU 如果想使用总线, 必须等待.
    2. 突发模式: 总线以每次一块模式操作, DMA 控制器通知设备获得总线, 发起一连串的传送, 然后释放总线. 缺点是 CPU 和其他设备可能阻塞相当长的周期.
    3. 飞跃模式: DMA 通知设备控制器直接将数据传送到主存.

7. 廉价磁盘冗余阵列
    1. RAID 0: 将磁盘分为一个个条带, 每个条带 k 个扇区.
    2. RAID 1: 在 RAID 0 的基础上加入相同数目的备份磁盘.
    3. RAID 2: 在字或字节的基础上, 构造汉明码, 然后将每一位分散存储到不同磁盘上.
    4. RAID 3: 不用汉明码, 而是用奇偶校验位, 这样一定能纠正一个磁盘出错的情况.
    5. RAID 4: 在条带的基础上使用奇偶校验位.
    6. RAID 5: 以循环方式在所有驱动器上均匀分布奇偶校验位.
    7. RAID 6: 使用了额外的奇偶块, 可以检查出两个磁盘同时出错.
8. 磁盘臂调度算法
    1. 读写一个磁盘块所需时间: 寻道时间 + 旋转延迟 + 实际数据传输时间. 一般来说, 寻道时间占主导地位.
    2. 先来先服务 (FCFS): 通过队列记录先后顺序.
    3. 最短寻道优先 (SSF): 总是处理与磁头距离最近的请求以使寻道时间最小化.
    4. 电梯算法: 始终保持按照一个方向移动, 直到那个方向上没有请求为止.