OS-Review

edited by KG3

2023.07.02

Chapter 1 - OS Introduction

- 操作系统功能
  - 系统角度：1.操作系统是一个控制程序：控制程序的执行，以避免错误和对计算机的不当使用。 2.系统是一个资源分配器：系统管理所有的资源，在冲突的资源需求下做出公平和高效的资源使用决策。
  - 用户角度：1.为个人电脑用户提供便捷的使用和出色的表现，隐藏底层的资源分配问题。 2.为共享计算机的用户最大化资源利用，使各个使用共享计算机的用户的需求尽可能满足。 3.为使用共享资源的用户提供服务，使得用户之间可以交换他们拥有的资源。 4.为移动计算机用户优化可用性和电池寿命。
- 操作系统的定义

  操作系统是一组控制和管理计算机软硬件资源、合理地对各类作业进行调度以及方便用户的程序的集合。
- Multiprogramming 多道程序设计

  multi-programming是指操作系统组织调度所要执行的多个任务以便CPU总在执行某个任务，所要执行的任务被预先放在任务池中，当某个任务暂时阻塞时（例如执行I/O操作时），系统可以从任务池中取出其它任务继续执行，其设计目的是使CPU持续工作使它得到充分的利用。
- Multitasking 多重任务处理

multi-tasking是指通过通过分时使用CPU使CPU在多个任务中迅速切换，其设计目的是使用户看来这些任务 是同时进行的，从而使多个用户能够同时运行多个任务。

- Interrupt Driven Mechanism 中断驱动机制
  - 硬件中断：设备。
  - 软件中断（例外或陷阱）：软件error、请求操作系统服务、进程出错（e.g.,无限循环）。
- Dual-mode 双模式
  - Dual mode让操作系统可以选择工作在用户模式或内核模式，通过硬件提供的Mode bit来指定模式。被指定为特权的指令只能运行在内核模式。执行系统调用时从用户模式切换到内核模式，调用结束后再切换回用户模式。
  - 采用Dual Mode可以让一些特权指令不能被用户程序执行，以保护操作系统和其它系统组件。
- System Calls 系统调用

  系统调用是进程与操作系统内核之间的程序接口，系统调用为用户程序提供了请求操作系统执行任务的方法。
- Program != Process 程序 != 进程
  - 进程是程序的一个执行实例。多个进程可以执行同一个程序代码。一个进程也不一定只执行一个程序的代码。
  - 进程是active的，有自己的local states!

Chapter 2 - Operating System Structures

- 详细阐述系统调用与API的关系

API是应用程序接口，提供可供程序员使用的函数。与系统调用相比，系统调用更加底层，与操作系统内核交互，在API中可以间接执行系统调用，用户程序调用API，API执行系统调用，系统调用将结果返回给API，API再返回给用户程序。使用API与直接使用系统调用相比具有更高的可移植性。

- 系统调用参数传递（3种）
  - 最简单的：通过寄存器传递。
  - Table-based：参数存储在内存里的block或table，然后block或table的地址被当作参数存储在寄存器中被传递。
  - Stack-based：参数先被程序入栈，然后被OS调出栈。
- 系统调用类型（6种主要种类）
  - Process control 进程控制

    end( ), abort( ), create_precess( )...
  - File manipulation 文件操作

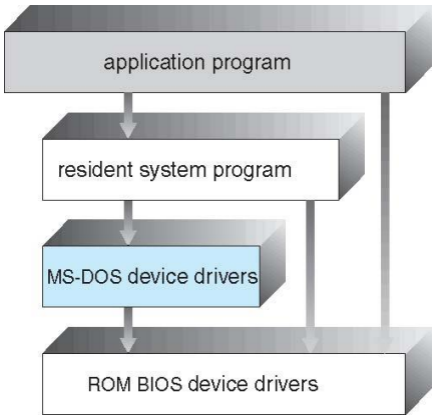
    create file, delete file...
  - Device manipulation 设备操作

    request device, release device...
  - Information maintenance 信息维护

    get/set time or data...
  - Communications 通信

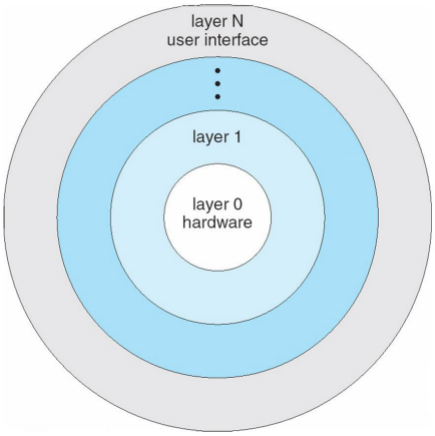
    两种模式：信息传递模式、共享内存模式
  - Protection 保护

    get_permission(), set_permission()...
- OS结构
  - Simple structure 简单结构– MS-DOS
    - 没有well-defined的结构；
    - 没有被划分为模块；
    - 接口和功能的层次没有很好的分开：应用程序也可以访问基础的I/O，易受错误程序的影响，受限于硬件。

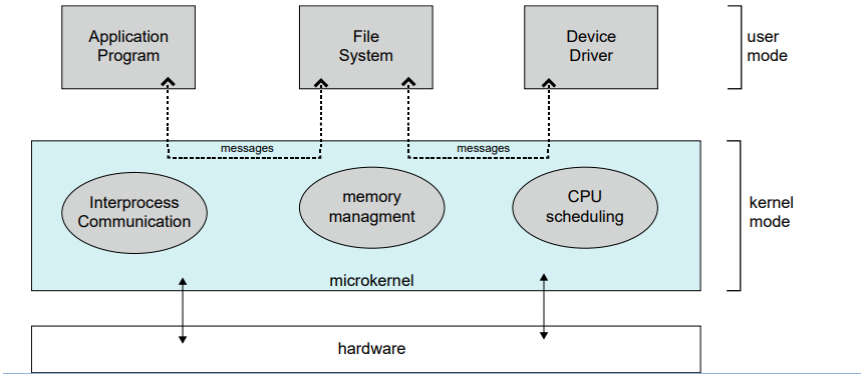
    

  - Monolithic Structure 单片结构-- UNIX
    - 由系统程序和内核两部分构成；
    - 将所有功能组合在一个层级；
    - 优点是性能有优势，缺点是难以实现。

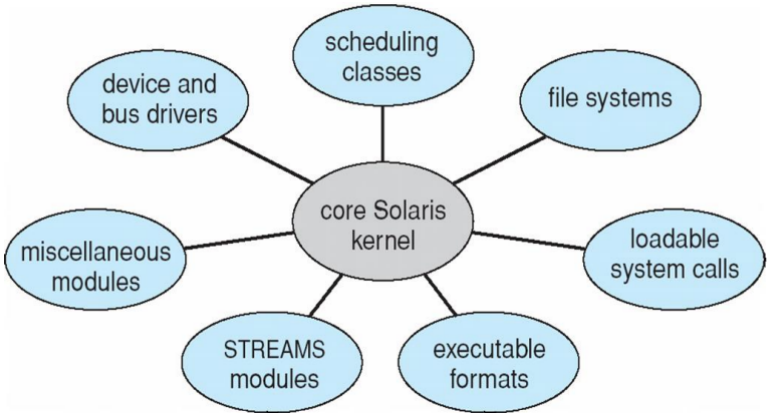
- Layered Approach 分层方法
  - 操作系统被分成一系列的层次，从硬件层逐层过渡到用户层。
  - 优点是容易构建和调试并在高层隐藏了低层实现。
  - 缺点是层次难以划分，会面临效率问题。



- Microkernel System Structure 微内核系统结构
  - 将尽可能多的内容从内核移到用户空间，提供最小的进程、内存管理与通信功能，提供客户端程序与服务之间的通信功能。
  - 优点是易于拓展，便于将操作系统移植到新的架构，更加灵活和安全。
  - 缺点是需要额外的用户空间到内核空间的通信开销。



- Modules 模块化结构
  - 操作系统有一系列的核心组件，可以通过模块的形式链接到不同的服务。
  - 优点是易于拓展，便于将操作系统移植到新的架构，更加灵活和安全
  - 缺点是需要额外的用户空间到内核空间的通信开销。



- Hybrid Systems 混合系统

- OS设计与实现
  - 首要问题：定义目标（user+system）和规范
  - 机制（mechanism）与策略（policy）分离
    - 机制决定如何做，策略决定做什么。
    - 举例：1.用于CPU保护的计时器机制，对应策略是计时器应该被设置为多长时间，在改变时间时不需要改变保护的机制。 2.在任务调度中的优先机制，对应的策略是I/O 密集型程序的优先级高于 CPU 密集型程序，在改变优先级的策略时不需要改变优先机制。
    - 机制与策略分离的设计原则的好处是拥有最大的灵活性，可以在改变策略的情况下不影响机制。
  - OS使用多种语言实现
    - 优点是代码编写更快，更为紧凑，更容易理解和调试。操作系统更容易移植到其他硬件。
    - 缺点是速度的降低和存储的增加。

Chapter 3 part1 - Process Concepts & Operations (from Programmer's Perspective)

- 什么是程序(program)?

  程序就是一段代码。或者说，程序是一个可执行文件。

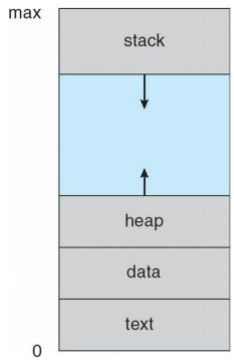
- Pre-processor 预处理机
  - 编译器进行语法检查和分析，如果没有语法错误，则生成中间代码，例如汇编代码。
  - 汇编器把.s生成为.o
  - 链接器把所有的目标文件和库链接在一起。
  - 如何编译多文件?
    - step1：准备所有的源文件。必须有且只有一个文件包含main函数。
    - step2：将它们逐一编译成目标代码。
    - step3：所有目标代码一起构建程序。
- 什么是进程(process)?
  - 进程是在执行中的程序。
  - 内存中的进程
    - Text section 文本段

      程序代码
    - Data section 数据段

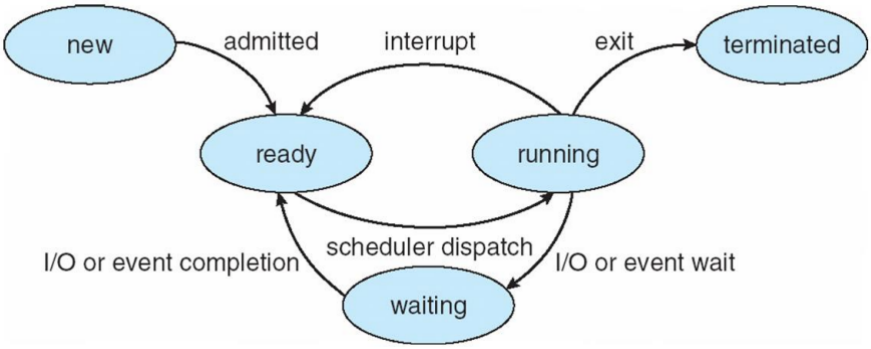
      全局变量
    - Stack 栈

      临时变量（函数参数，返回地址，局部变量）
    - Heap 堆

      动态分配内存
    - PC&Contents of Register 程序计数器和寄存器内容



- 进程状态
  - new 新的：进程正在创建
  - running 运行：指令正在执行
  - waiting 等待：进程等待发生某个事件（如I/O完成或收到信号）
  - ready 就绪：进程等待分配处理器
  - terminated 终止：进程已经完成执行



任何时刻任何处理器，只能有一个进程正在运行。其他许多进程在就绪或等待。

- 进程控制块（PCB）
  - 进程状态
  - 程序计数器：进程将要执行的下个指令的地址
  - CPU寄存器：累加器、索引寄存器等
  - CPU调度信息：进程优先级、调度队列的指针等
  - 内存管理信息：页表、段表等
  - I/O状态信息：分配给进程的I/O列表等
  - 记账信息：CPU时间、实际使用时间等

- 进程操作
  - 进程识别
    - getpid()：返回调用进程的PID
  - 进程创建
    - fork()
    - 父进程/子进程
    - 第一个进程：**init**

      在内核启动时创建，第一个任务是创建更多进程。
    - 子进程可能变成孤儿，可以通过re-parent讲init进程作为孤儿进程的父进程。

- 父进程和子进程的关系

  资源共享：父子共享所有资源or子进程共享父进程的资源子集or父子不共享任何资源

  执行：父子并发执行or父进程等待子进程终止

  地址空间：子进程是父进程的副本（具有与父进程同样的数据和程序）or子进程加载另一个新的程序

- 子进程拷贝如下参数：

Cloned items	Descriptions
Program code [File & Memory]	They are sharing the same piece of code.
Memory	Including local variables, global variables, and dynamically allocated memory.
Opened files [Kernel's internal]	If the parent has opened a file "A", then the child will also have file "A" opened automatically.
Program counter [CPU register]	**That's why they both execute from the same line of code after fork() returns.**

子进程不拷贝如下参数：（注意，这些参数全都处于内核中）

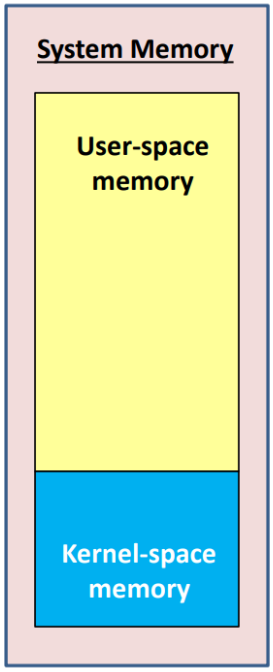
Distinct items	Parent	Child
Return value of `fork()`	PID of the child process.	0
PID	Unchanged.	Different, not necessarily be "Parent PID + 1"
Parent process	Unchanged.	Doesn't have the same parent as that of the parent process.
Running time	Cumulated.	Just created, so should be 0.

  - 进程执行
    - exec()
    - 调用exec()后，进程改变了当前正在执行的代码，并且不会返回到初始代码。
    - 进程扔掉了很多东西，比如：Memory（局部变量，全局变量，动态分配内存），寄存器值（PC）。
    - 进程不会改变：PID、进程间的关系、运行的时间
  - fork()+ exec*() + wait() = system()
    - 利用wait()来保证父子间的执行顺序
    - 当父进程的某一个子进程由运行到终止时，wait()返回
    - 如果根本没有子进程，或者没有正在运行的子进程，wait()不会挂起父进程
    - waitpid()

Chapter 3 part2 - Process Operations(from Kernel's Perspective)

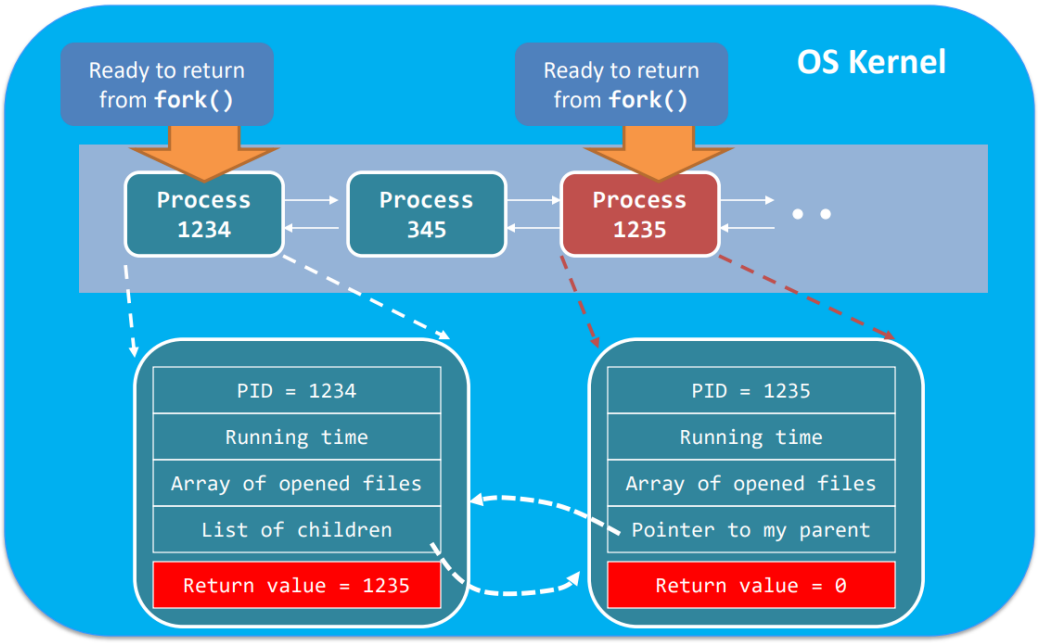
- 内核空间 vs 用户空间

	Kernel-space memory	User-space memory
Storing what	Kernel data structure Kernel code Device drivers	Process' memory. Program code of the process
Accessed by whom	Kernel code	User program code + kernel code

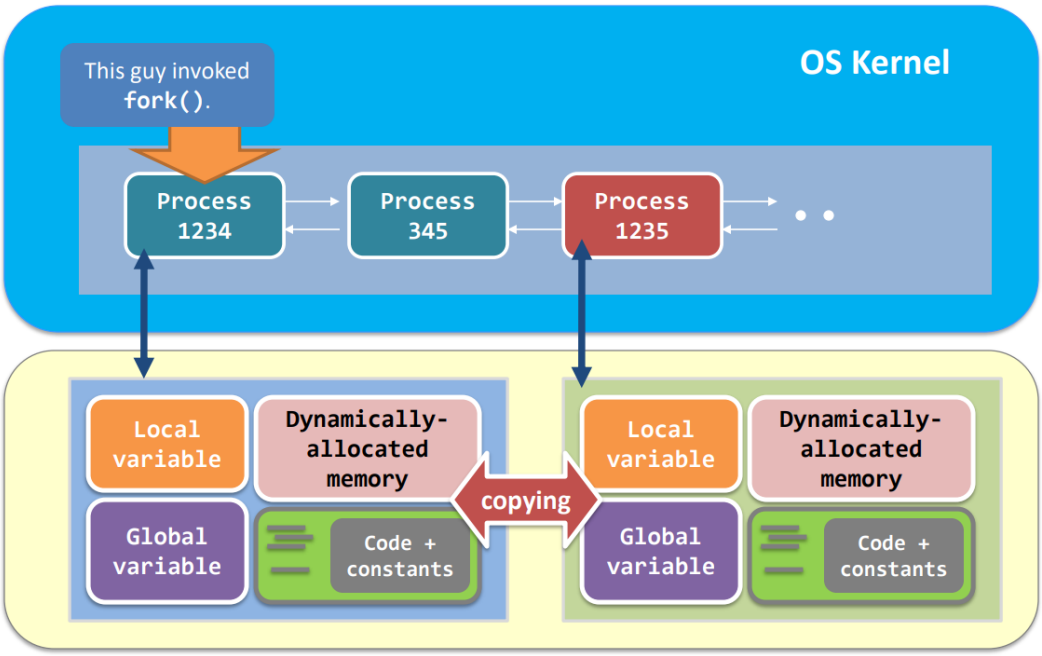


The kernel is invincible!

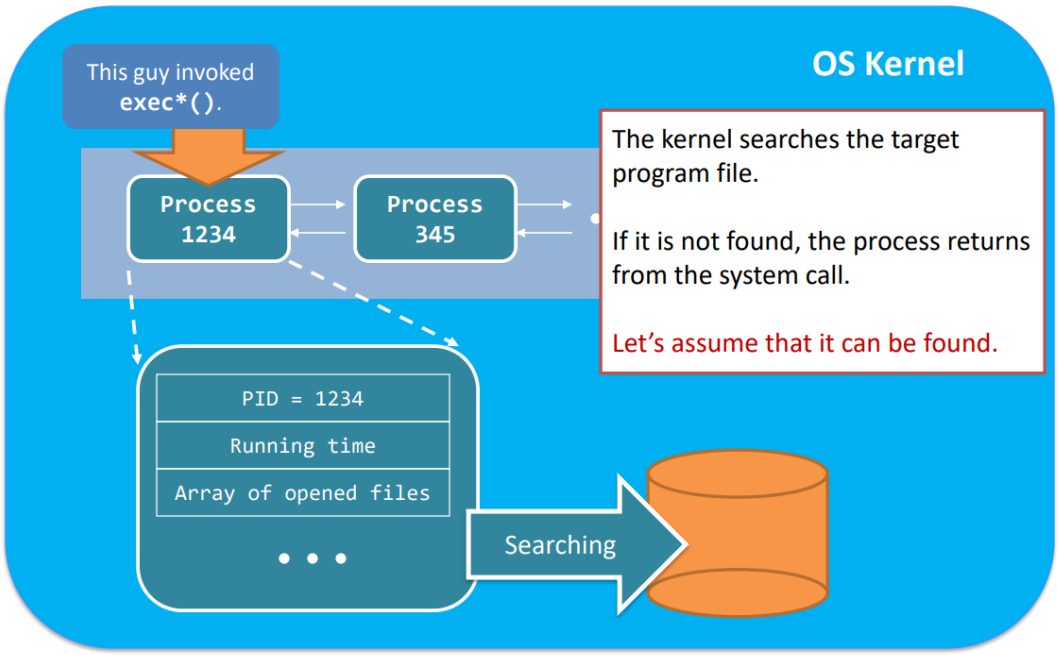
- 总运行时间=用户时间+系统时间
- fork()
  - 内核空间



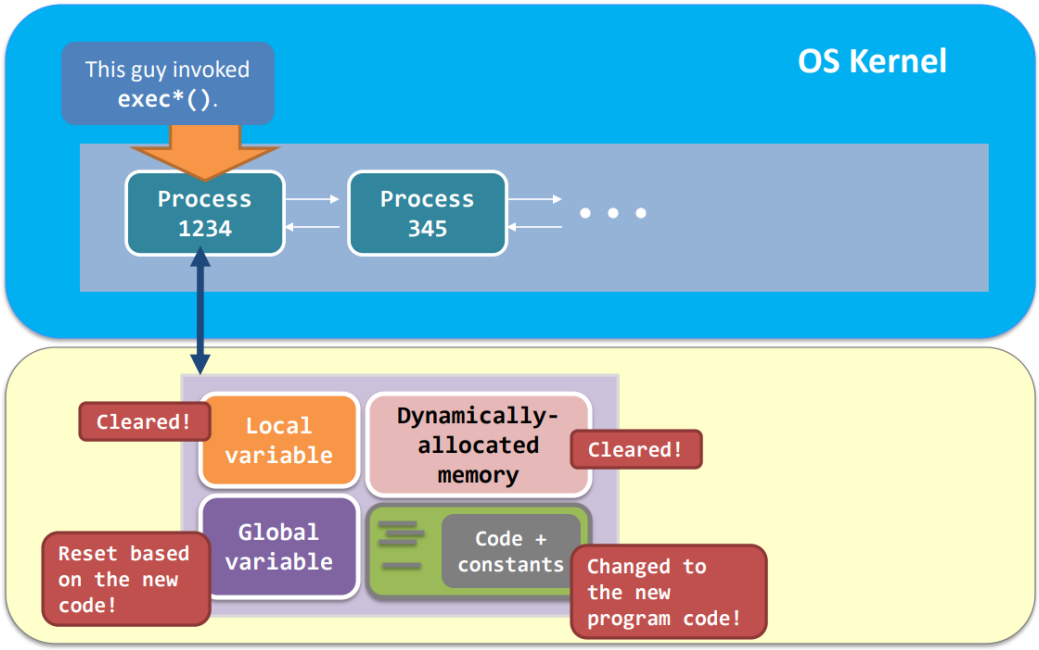
  - 用户空间



- exec()
  - 内核空间



  - 用户空间

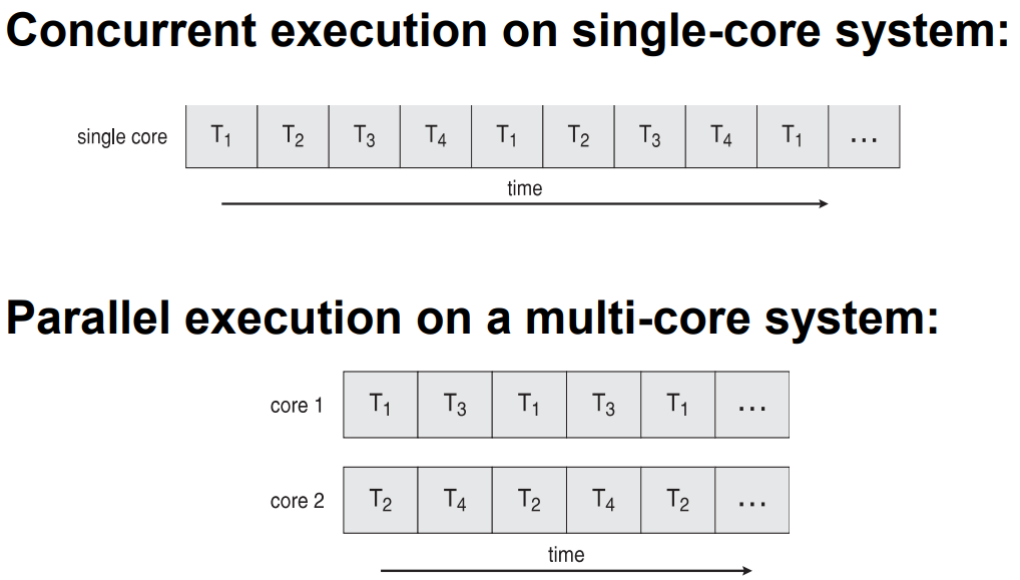


- wait() & exit()
  - 子进程视角下调用exit()
    - 内核释放所有已分配的空间
    - 然后，内核删除用户空间内存中有关相关进程的所有内容，包括程序代码和已分配的内存。
    - 在进程表中保留子进程条目(终止状态)
    - 这个子进程现在被称为僵尸进程。它在内核空间内存中的存储空间保持在最小。子进程有PID和进程结构
    - 内核将子进程的终止通知子进程的父进程。通知是一个被称为SIGCHLD的信号
    - 无法杀死僵尸进程，因为它已经dead。
  - 父进程视角下调用wait()
    - 内核给父进程设置一个信号处理例程(它是一个函数指针)。该信号处理例程将在SIGCHLD到来时执行。
    - 内核将父进程设置为睡眠状态。
    - 当SIGCHLD出现时，将调用信号处理例程
    - SIGCHLD的默认处理：接受并删除SIGCHLD;销毁向她发送信号的子进程。
    - 然后删除信号处理程序，也就是说，父进程再次忽略SIGCHLD。它返回到先前执行的代码，返回到用户空间
    - 最后，wait()系统调用的返回值是终止子进程的PID。
  - Wait()在子进程已经终止后被调用
    - 子进程已经被终止(成为僵尸)，SIGCHLD也已经被发送到父进程
    - 与情形1类似，内核设置信号处理例程...然而，wait()系统调用发现SIGCHLD信号已经存在。因此，会立即采取默认操作。
  - 如果父进程没有调用wait()就终止?

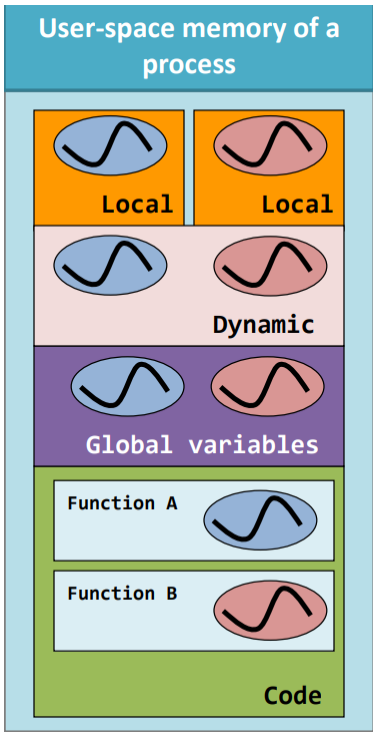
    子进程会成为孤儿，Init是新的父进程，它定期调用wait()

Chapter 4 - Threads

- 多线程
  - 动机
    - 应用视角：大多数软件应用程序都是多线程的，每个应用程序都被实现为具有多个控制线程的一个进程；
    - 系统视角：现代计算机通常是多核的，但是，每个处理器一次只能运行一个进程，CPU未被充分利用，所以为了提高效率，给每个核心分配一个任务，实现真正的并行性(而不仅仅是单核系统上的交错并发)。
  - 并发 vs 并行

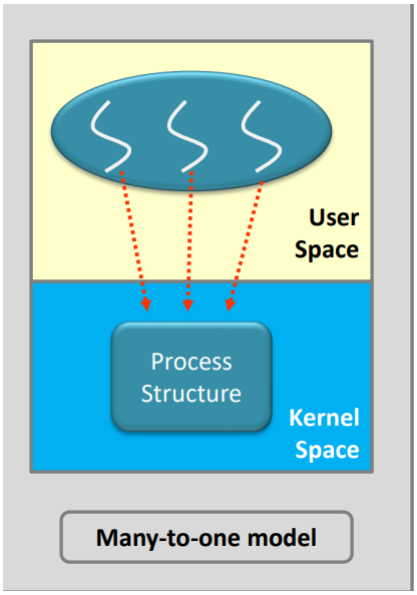


  - 内部构成
    - 所有线程共享相同的代码。一个线程从一个特定的函数开始，称之为线程函数。线程函数可以调用其他函数或系统调用，但是，线程永远不能返回到线程函数的调用者。
    - 所有线程共享相同的全局变量区和相同的动态分配内存。所有线程都可以读写这两个区域。
    - 每个线程都有自己的局部变量的内存范围。因此，栈是每个线程的私有区域。



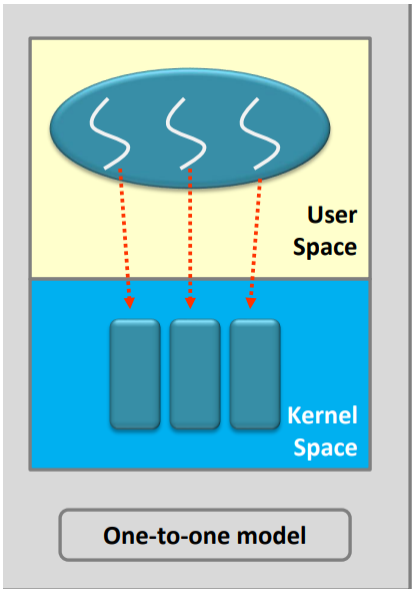
- 多线程的优点
  - 响应性和多任务处理能力：多线程设计允许应用程序同时执行并行任务，对于交互式应用程序尤其重要；
  - 易于资源共享：进程通过共享内存或消息传递共享资源，这必须由程序员显式地安排；
  - 经济：为创建进程分配内存和资源的成本很高，比创建线程慢几十倍，进程之间的上下文切换也很昂贵，速度要慢好几倍；
  - 可伸缩性：线程可能在不同的核上并行运行。

- 线程模型
  - 多对一模型
    - 多个用户线程映射到一个内核线程。
    - 优点：易于内核的实现
    - 缺点：如果一个线程执行阻塞系统调用，那么整个进程将会阻塞。因此无法利用多个处理核！

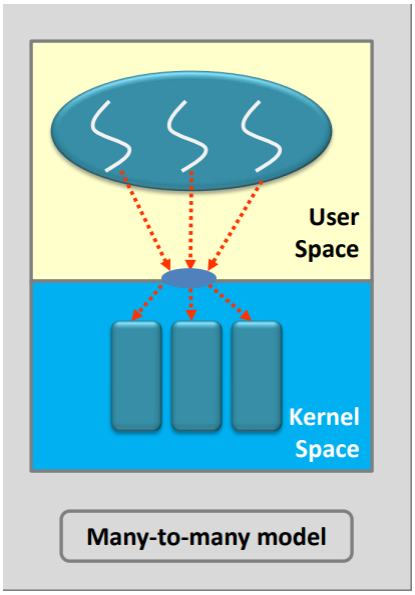


  - 一对一模型
    - 一个用户线程映射到一个内核线程

        - 优点：调用阻塞系统调用只阻塞那些调用线程，因此有高并发性
        - 缺点：创建一个用户线程就要创建一个内核线程，这个开销会影响应用程序的性能；不能创建太多线程，因为它受到内核内存大小的限制



    - 多对多模型
        - 多路复用多个用户级线程到同样数量或更少数量的内核线程。
        - 优点：可以创建尽可能多的线程，也具有高度的并发性

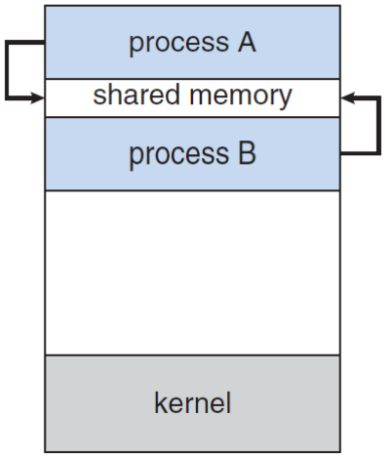


- 线程编程
    - 同步线程 vs 异步线程
        - 异步线程：父线程在创建子线程后继续执行，父线程和子线程并发执行，每个线程独立运行，数据共享少；
        - 同步线程：Fork-join策略:父线程等待子线程终止，数据共享多
    - 线程创建
        - pthread_create()：线程创建时设置被创建线程的线程函数
        - pthread_join()：与wait() and waitpid()功能类似
    - 参数传递
    - 多线程
    - 返回值

        - pthread_exit()：终止时，一个指向全局变量或一块动态分配的内存的指针被返回给主线程。
- 隐式线程
    - 定义：将线程的创建和管理从程序员转移到编译器和运行时库，这种策略称为隐式线程
    - 线程池
        - 在进程开始时创建一定数量的线程，并加到池中以等待工作。
        - 必要时唤醒线程；
        - 完成后返回池；
        - 如果池中没有可用线程，则等待直到一个线程空闲。
    - OpenMP
        - 识别并行区域——可以并行运行的代码块，应用程序开发人员在并行区域插入编译指令
        - 当OpenMP遇到该指令时，它会创建与系统处理核数量一样多的线程
- fork()和exec()的语义
- 信号处理
- 线程撤销
- 线程本地存储

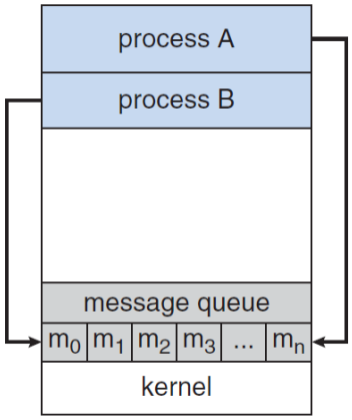
Chapter 5 part1 - Process Communication

- IPC：进程间通信
- 共享内存系统
    - 建立共享内存区域，对共享区域进行读写
    - 访问被视为例行内存访问



    - 有界/无界缓冲区，关注有界缓冲区
        - 当生产者想入队一个但buffer满时，需要挂起生产者，等待消费者出队一个，然后唤醒生产者。
        - 当消费者想出队一个但buffer空时，需要挂起消费者，等待生产者入队一个，然后唤醒消费者
- 消息传递系统
    - 交换信息
    - 需要内核干预

- 在分布式系统中更容易实现



- 命名
  - 直接通信：需要通信的每个进程必须明确指定通信的接收者或发送者。

    缺点是：有限模块化
  - 间接通信：通过邮箱（端口）来发送和接收信息。

    问题1：当多个进程与一个邮箱相关联时，谁接收消息?

    策略：允许一个链路最多与两个进程关联；允许一次最多一个进程执行操作receive；允许系统随意选择一个进程以便接受消息。

    问题2：谁拥有邮箱?

    进程(所有权可能被传递)；操作系统(需要一个方法来创建，发送/接收，删除)
- 同步
  - 阻塞（阻塞被认为是同步的)

    阻塞发送：发送方被阻塞，直到消息被接收进程或邮箱所接收

    阻塞接收：接收方被阻塞，直到有消息可用
  - 非阻塞（非阻塞被认为是异步的）

    非阻塞发送：发送者发送消息并恢复

    非阻塞接收：接收方接收有效的信息或空消息
- 缓存
  - 零容量：链路中不能有任何消息处于等待，因此发送者应阻塞，直到接收者接收到消息。
  - 有限容量
  - 无限容量

- 管道
  - 普通管道
    - 两个进程按照标准的生产者-消费者方式进行通信
    - 生产者向管道的一段写（写入端）
    - 消费者从管道的另一端读（读出端）
  - 命名管道
    - 不需要父子关系(进程必须驻留在同一机器上)
    - 多个进程可以使用命名管道进行通信(可能有几个生产者)
    - 在被显式删除之前持续存在

        - 通信是双向的(仍然是半双工)

chap 5 part2 - Process Synchronization

- 竞争条件

  多个进程并发访问和操作同一数据并且执行结果与特定访问顺序有关。

- 互斥
  - 抢占式内核和非抢占式内核
    - 允许(不允许)进程在内核模式下运行时被抢占。
    - 抢占式内核反应更迅速，更适合实时编程
  - 定义临界区是一种解决方案。
  - 进入和退出的实现-要求
    - `互斥：不能同时有两个进程处于临界区
    - `速度：每个进程以一定的速度运行，但是不对进程的相对速度和CPU数量做出任何假定
    - `进步：不处于临界区的进程不能阻塞其它进程进入临界区
    - `有限等待：进程不能一直等待进入临界区，等待次数或时间应有限
  - 方案#1 - 禁止中断
    - 方法：当进程在临界区内时禁用上下文切换。
    - 影响：当一个进程处于它的临界区时，没有其他进程能够运行。
    - 在多处理器环境中不可行。性能问题(可能牺牲并发性)
  - 方案#2 - 互斥锁
    - 方法：进程必须在进入临界区之前获得锁，并在退出临界区时释放锁
    - 问题：可能造成忙等，耗费cpu资源（spinlock）。
  - 方案#3 - 严格轮转
    - 方法：借助"turn"，轮流进入临界区
    - 问题：可能造成忙等。交替顺序太严格了，一个进程无法连续两次进入临界区。
  - 方案#4 - Peterson解决方案
    - 方法：进程会像“绅士”一样:如果你想进入临界区，我会让你先进入临界区

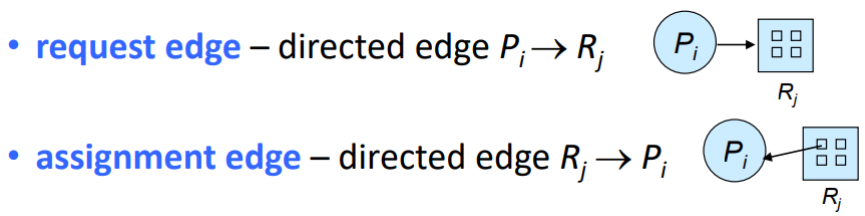
Shared object: “turn” & “interested[2]”

```
1  int turn;                            /* who can enter critical section */
2  int interested[2] = {FALSE,FALSE};   /* wants to enter critical section*/
3
4  void enter_region( int process ) {   /* process is 0 or 1 */
5    int other;                         /* number of the other process */
6    other = 1-process;                 /* other is 1 or 0 */
7    interested[process] = TRUE;        /* want to enter critical section */
8    turn = other;
9    while ( turn == other &&
               interested[other] == TRUE )
10     ;    /* busy waiting */
11 }
12
13 void leave_region( int process ) {    /* process: who is leaving */
14   interested[process] = FALSE;    /* I just left critical region */
15 }
```

Entry

Exit

    - 问题：忙等（浪费CPU资源）；优先级反转：低优先级的在临界区时，高优先级的会忙等
  - 最终方案：信号量
    - 两种：计数型信号量和二进制型信号量
    - 希望使用资源，执行down()来减少计数；释放资源，执行up()来增加计数；当计数变为0时，阻塞希望使用的进程
    - 问题1：忙等。解决方法：阻塞进程而不是忙等待(将进程放入等待队列中)
    - 问题2：原子性。解决方法：使中断无效。只有一个进程可以调用“disable_interrupt()”。之后的进程将被阻塞，直到调用" enable_interrupt() "。
  - 原子操作
    - 原子操作的所有指令都没有完成，或者原子操作的所有指令都完成了。
- 死锁
  - 定义：在多道程序环境中，多个进程可以竞争有限数量的资源。当一个进程申请资源时，如果这时没有可用资源，那么这个进程进入等待状态，如果所申请的资源被其它也在等待的进程占有，那么该等待进程可能再也无法改变状态，这种情况称为死锁。
  - 要求
    - 互斥：一个资源一次只能被一个进程所使用
    - 占有并等待：一个进程应至少占有一个资源并在等待被其它进程所占有的资源
    - 非抢占：资源只能在进程执行结束后被进程自愿释放
    - 循环等待：有一组等待进程{P0，P1，……Pn}，P0等待的资源为P1占有，P1等待的资源为P2占有，……，Pn-1等待的资源为Pn占有，Pn等待的资源为P0占有
  - 资源分配图



  - 死锁检测
    - case1：每种资源只有一个。若有环，则有死锁；若无环，则无死锁。

    - case2：每种资源有多个。银行家算法

      安全状态：存在一个运行顺序，保证所有进程的需求都能得到满足

      不安全状态：不存在任何顺序来保证需求（这不是死锁）
    - 鸵鸟算法：忽略问题并假装死锁从未发生过(手动停止运行并重新启动)
- 经典的IPC问题
  - 生产者-消费者问题（有界缓冲问题）
    - 可能无法使用管道：多个生产者vs多个消费者，且他们之间没有父子关系
    - 缓冲区是一个共享对象。互斥是必要的。how？一个二进制信号量用作临界区的入口和出口
    - 因为缓冲区的大小是有限的，所以需要进程间的协调。how？两个信号量用作计数器来监视缓冲区的状态。需要两个信号量，因为两种挂起的条件不同。
    - 信号量的先后顺序很重要，否则可能会产生死锁

Producer function

```
1  void producer(void) {
2      int item;
3
4      while(TRUE) {
5          item = produce_item();
6          down(&empty);
7          down(&mutex);
8          insert_item(item);
9          up(&mutex);
10         up(&full);
11     }
12 }
```

Consumer Function

```
1  void consumer(void) {
2      int item;
3
4      while(TRUE) {
5          down(&full);
6          down(&mutex);
7          item = remove_item();
8          up(&mutex);
9          up(&empty);
10         consume_item(item);
11     }
12 }
```

  - 哲学家就餐问题
    - final solution

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Main function

```
1  void philosopher(int i) {
2      think();
3      take(i);
4      eat();
5      put(i);
6  }
```

Section entry

```
1  void take(int i) {
2      down(&mutex);
3      state[i] = HUNGRY;
4      test(i);
5      up(&mutex);
6      down(&s[i]);
7  }
```

Section exit

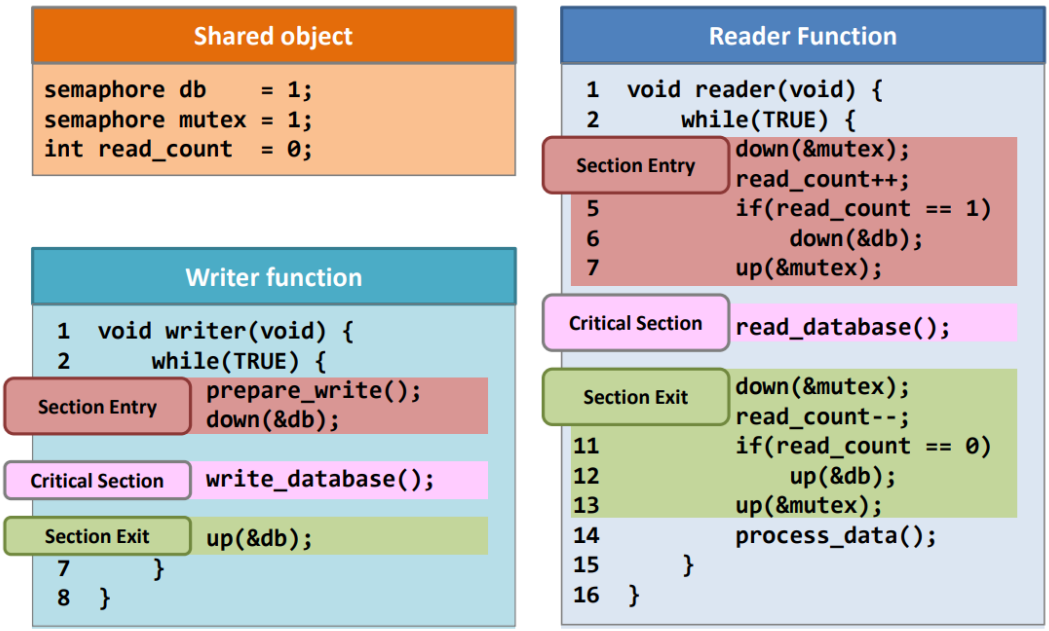
```
1  void put(int i) {
2      down(&mutex);
3      state[i] = THINKING;
4      test(LEFT);
5      test(RIGHT);
6      up(&mutex);
7  }
```

I will explain the code later.

Extremely important helper function

```
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

  - 读者-作者问题
    - 实际上是一个并发数据库问题
    - 规则：1.读者读时，其他读者也可以读；2.读者读时，作者不可以写；3.作者写时，读者不可以读



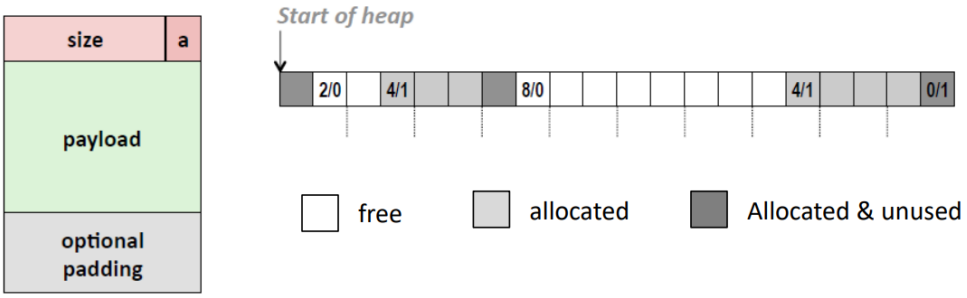
Chapter 6 - Process Scheduling

- 上下文切换
  - 先备份旧进程所有的寄存器值，然后load新进程的context
  - 上下文切换具有开销
- 抢占调度vs非抢占调度
- 性能衡量指标
  - CPU利用率
  - 吞吐量：单位之间完成的进程数量
  - 周转时间：从进程提交到进程完成的时间
  - 等待时间：进程在等待队列里的时间
  - 响应时间：从进程提交到第一次响应的时间
- 调度算法
  - FIFO
    - FIFO调度对输入顺序很敏感。
  - SJF
    - 非抢占式vs非抢占式
    - 等待时间和周转时间的减少是以增加上下文切换次数为代价的。
    - SJF可以证明是最优的，因为它提供了最小的平均等待时间
    - 挑战:如何知道下一个CPU请求的长度?
  - RR
    - 时间量
    - RR算法在所有方面都不好，为什么我们还需要它？--在RR算法下，进程的响应性很好
  - 优先级调度
    - 问题：无限阻塞或饥饿
    - 解决方案：老化(逐渐增加等待进程的优先级)

    - 多级队列调度
        - 进程在提交给系统时被分配一个固定的优先级。
        - 优先选择优先级高的进程允许
        - 每个优先级内部有自己的调度算法
        - 问题：低优先级的可能会无限阻塞或饥饿。解决方法：动态优先级
- 实时CPU调度
    - 单调速率调度
        - 抢占的、静态优先级
        - 进程的优先级与其周期成反比
    - 最早截止期限优先调度
        - 根据截止期限动态分配优先级
        - 截止期限越早，优先级越高

Chapter 7 part1 - Memory Management from a Programmer's Perspective

- 地址空间
    - 每个进程都有自己的地址空间，它可以驻留在物理内存的任何部分
- 程序代码&常量
    - 常量存储在代码段。
    - 代码和常量都是只读的。
- 数据段&BSS
    - 静态变量被视为与全局变量相同
    - 数据
        - 包含初始化的全局变量和静态变量。
        - 数据段已经分配了所需的空间。
    - BSS
        - 包含未初始化的全局变量和静态变量。
        - BSS只是一堆符号。空间尚未分配。一旦进程开始执行，空间才会分配。
        - 在32位Linux系统上，用户空间的寻址空间大约是3GB。
- 栈
    - 包含：所有的局部变量，所有函数参数，程序参数，和环境变量
    - 如果无限递归
        - 堆栈溢出，程序终止
        - 解决方案：尽量减少参数的数量、尽量减少局部变量的数量、尽量减少调用次数、使用全局变量
- 堆
    - 只有当你请求内存时，才会分配内存。
    - 当malloc()遇到空闲块时
        - 隐式空闲链表

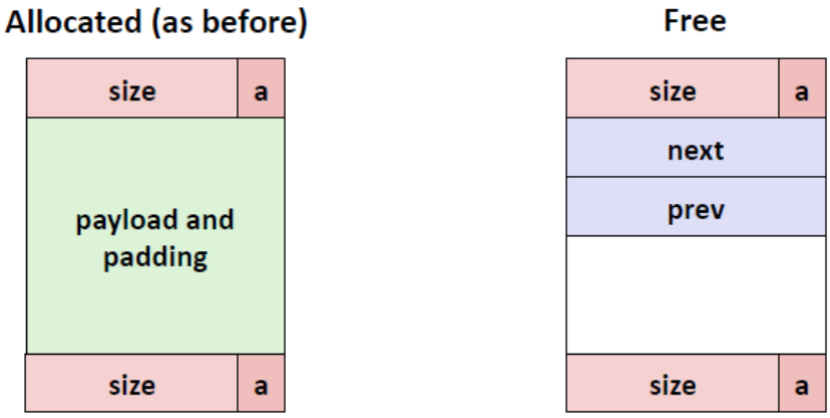


连续分配：可能需要线性时间搜索

首次适应；循环首次适应：从上次搜寻结束的位置开始继续寻找；最佳适应；最坏适应；

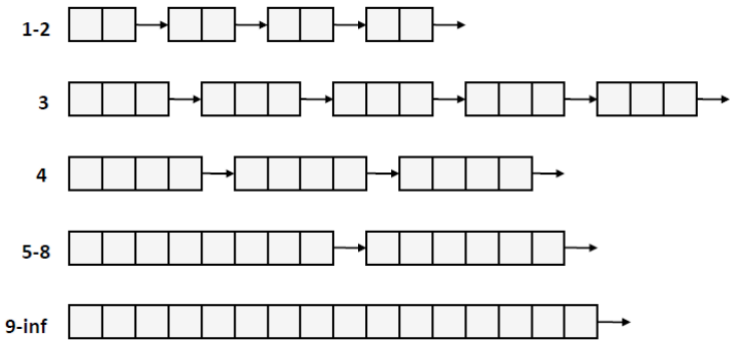
合并（4种情况）

- 显示空闲链表

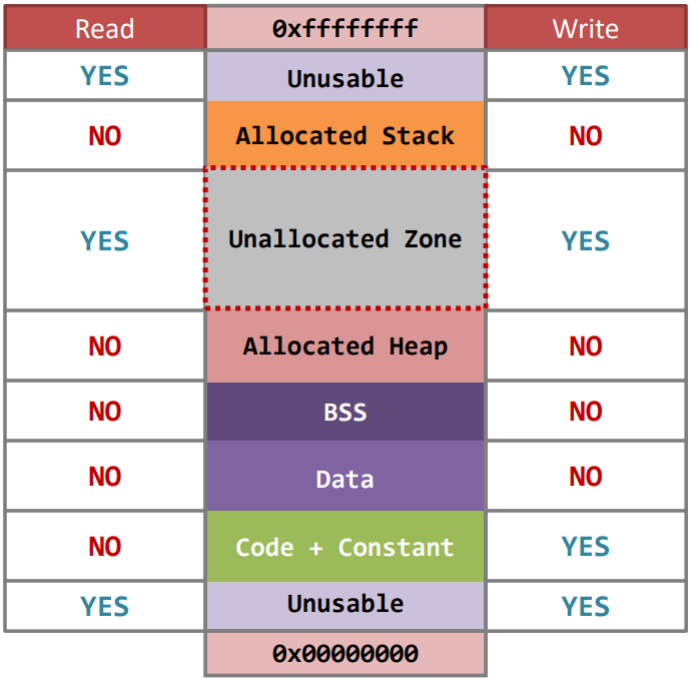


只跟踪空闲块(后进先出或地址顺序)

- 分离空闲链表



- 段错误



Chapter 7 part2 - Memory Management from the Kernel's Perspective: Virtual Memory Support

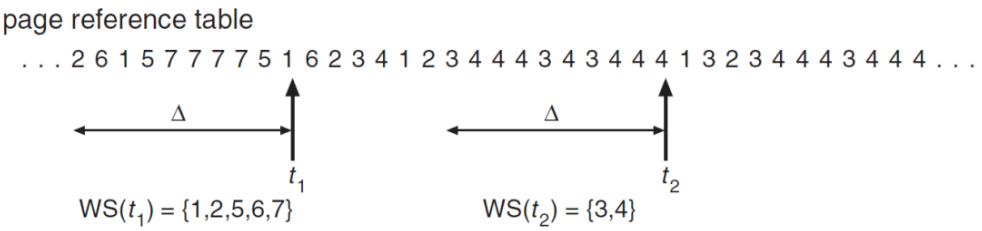
- 虚拟内存的好处
  - 不同的进程使用相同的虚拟地址，它们可能被转换成不同的物理地址。
  - 可以实现内存共享。
  - 可以实现内存增长。
- MMU实现
  - paging
    - paging将内存划分为称为页的固定块。
    - MMU中的查找表现在称为页表。
    - 每个进程有自己的页表
  - 页表结构
    - 正常的页表太大
    - 可以采取两级页表/多级页表
  - TLB
- 请求调页
  - 系统只显示内存已分配：分配了虚拟地址空间，但尚未分配页。
  - 直到访问它，它才真正被分配。
  - 如果进程试图访问那些尚未实际分配的虚拟页时，会发生缺页错误。
  - 缺页错误处理例程
    - 若物理页未满，找到物理页分配即可
    - 若物理页已满，则找到受害页，进行替换，替换出去的到交换区
    - 交换区通常是一个预留在永久存储设备中的空间。交换区大小至少应该与物理内存的大小相同。
  - 写时拷贝

        - Copy-on-write技术允许父进程和子进程在fork()被调用后共享页面。
        - 只有当其中一个进程想要写入共享页时，才会复制和修改新的分离页。
    - 性能

$$(1-p) \times ma \ + \ p \times page\ fault\ time$$

- ma: memory access time (10-200ns)
- p: prob. of a page fault
- $page\ fault\ time$: ms

- 页面替换算法
    - 最优页面置换算法
        - 置换最长时间不会被使用到的页面
        - 问题：不幸的是，你永远不知道未来.....
    - FIFO
        - 当必须置换页面时，选择最旧的页面
    - LRU
        - 置换最长时间没有使用的页面
    - 第二次机会算法（近似LRU）
        - 循环队列，找引用位为0的页面
        - 若页面的引用位为1，则给它第二次机会，并把它的引用位置0
    - Belady's anomaly
        - 对某些页面置换算法，随着分配帧数量的增加，缺页错误率可能会增加。
        - 堆栈算法不会，例如LRU
        - 当分配帧为n时在内存中的页面，当n变为n+1时这些页面也会在 内存中
- 帧分配
    - 抖动
        - 如果一个进程被分配的帧不够，就会频繁产生页错误，然后引起页面调度，这种高度的页面调度活动称为抖动。
        - 解决方法：使用工作集策略估计所需帧数。

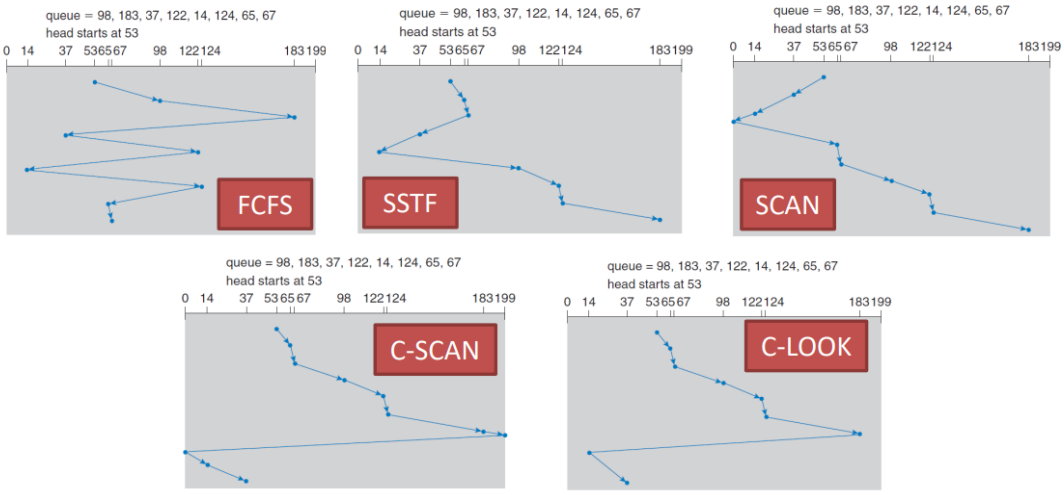


Chapter 8 - Mass Storage

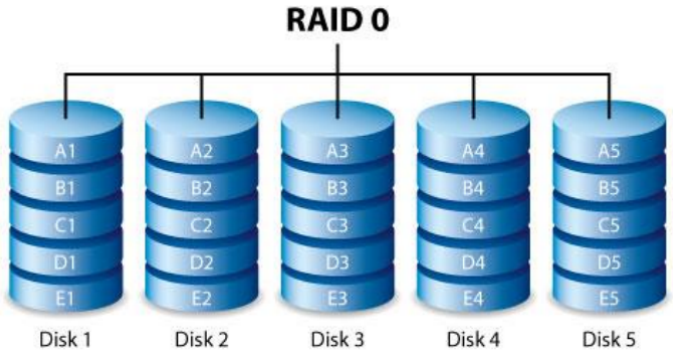
- 硬盘结构-物理视角
    - 柱面
    - 磁道
    - 扇区

    - 访问=寻道+旋转
        - 寻道时间=把磁头移动到指定柱面
        - 旋转等待时间
    - 恒定线速度
        - 每个磁道的比特密度均匀，外磁道可容纳更多扇区
        - 可调转速，保持数据传输率不变
    - 恒定叫速度
        - 恒定旋转速度
        - 内部磁道到外部磁道的比特率不断降低
- 如何使用磁盘
    - 地址映射
    - 格式化
    - 坏块管理
- 磁盘格式化
    - 步骤1:低级格式化/物理格式化
        - 划分扇区，使磁盘控制器可以读/写
        - 为每个扇区用特殊的数据结构填充磁盘
        - 在工厂完成，用于测试和初始化。这一步也可以设置扇区大小
    - 步骤2:发货后如何使用磁盘保存文件?
        - 选择1:文件系统
        - 选择2:原始磁盘，不用FS
- 坏块管理
    - 维护坏块列表(在低级格式化期间初始化)并保留一定数量的备用扇区
    - 扇区保留/转发:逻辑上将坏扇区替换为一个备用扇区
    - 扇区滑动:重新映射到下一个扇区(需要移动数据)
- 磁盘调度
    - FCFS
        - 本质公平，但不能提供最快的服务
    - SSTF
        - 选择最接近当前磁头位置的请求
        - 可能造成饥饿
    - SCAN
        - 电梯算法
        - 如果请求密度最大的是磁盘的另一端，那么它们需要等待很长时间
    - C-SCAN
        - 到达一端后立即返回另一端，旨在提供更统一的等待时间
        - 不需要移动整个磁盘的宽度，而只需要到达最后的请求
    - C-LOOK

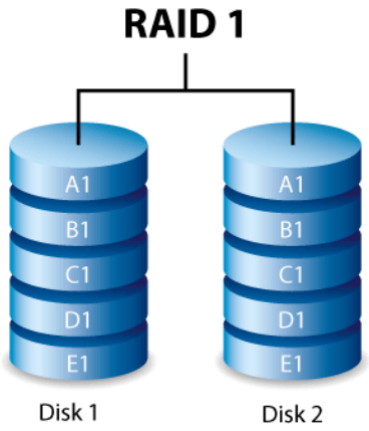
- 在继续朝既定方向前进之前，先寻找一个请求
- 磁头运动比SCAN/C-SCAN少



- 固态磁盘
- RAID（磁盘冗余阵列）
  - RAID0
    - 块分条，没有冗余
    - 提供更高的数据传输速率
    - 不能提高可靠性。一旦硬盘出现故障，可能导致数据丢失。

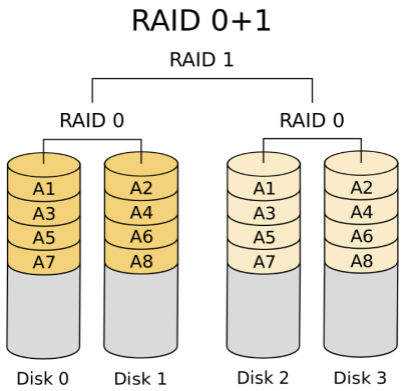


  - RAID1
    - 磁盘镜像
    - 数据的两个副本保存在两个物理磁盘上，数据总是相同的。
    - 存储成本高

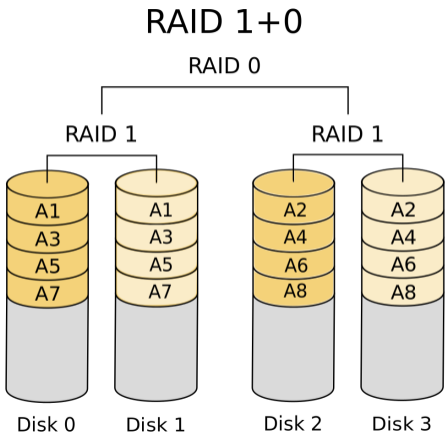


  - RAID 0+1

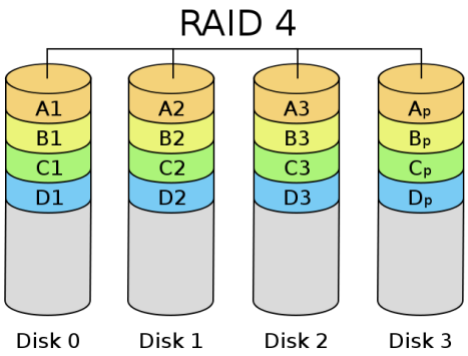
- 首先数据镜像，然后数据分条
- RAID0提供性能，RAID1提供可靠性



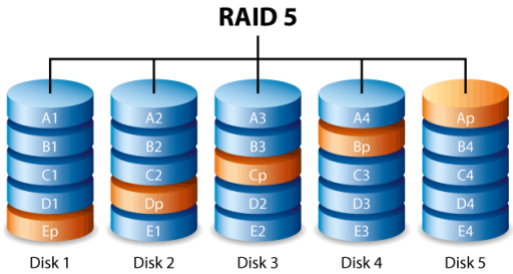
- RAID 1+0
  - 首先数据分条，然后数据镜像



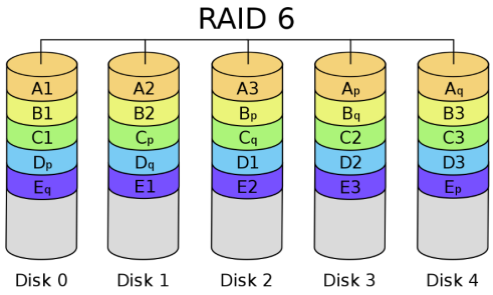
- RAID4
  - 奇偶校验生成:每个奇偶校验块为对应数据盘的异或值。有专用奇偶校验盘。
  - 问题：磁盘带宽未被充分利用。在正常模式下，校验盘无法被访问。校验盘可能成为瓶颈。



- RAID5
  - 均匀奇偶校验分布
  - 兼备良好的性能、良好的容错能力、高容量、存储效率



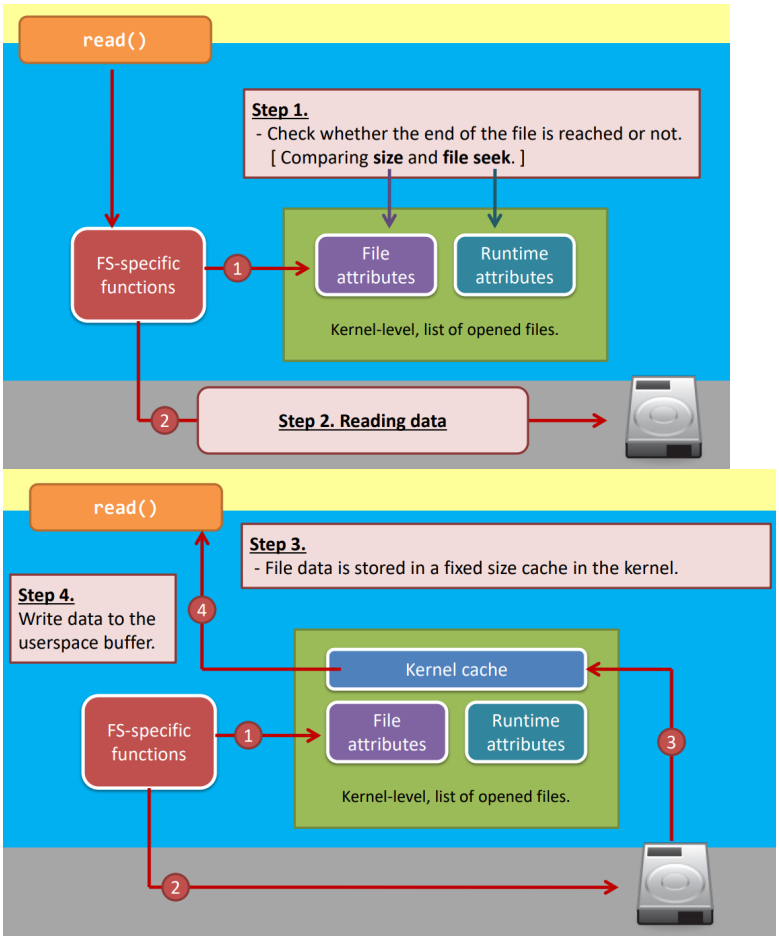
- RAID6
  - 通过维护两个校验来防止两个磁盘故障



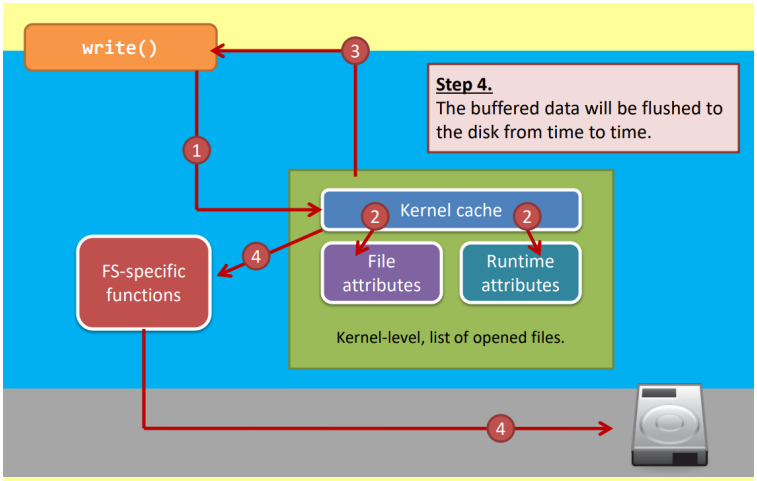
Chapter 9 part1 - File Systems From Programmer Perspective

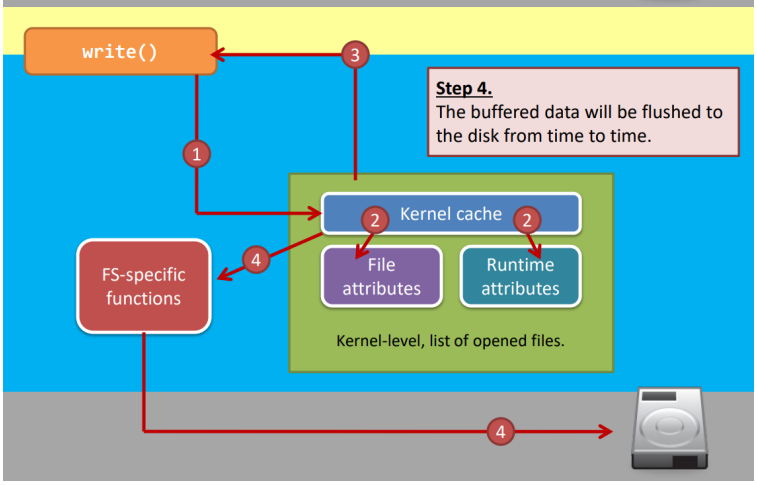
- FS介绍
  - OS != FS
    - 一个OS可以支持多个FS
    - 一个FS可以被多个OS读取
  - 存储设备 != FS
    - FS必须存储在设备上。但是，设备可能包含也可能不包含FS。某些存储设备可以承载多个FS。
    - 存储设备只是一个虚拟容器。它不知道也不需要知道什么FS存储在其中。操作系统指示存储设备如何存储数据。
- 文件
  - 什么是文件?
    - 操作系统提供的存储信息的统一逻辑视图。
    - 操作系统角度:文件是一个逻辑存储单元(一系列逻辑记录)，是一种抽象的数据类型
    - 用户视角:最小的逻辑辅助存储分配
  - 文件属性
  - 文件权限
    - 文件所有者/文件组/其他人的读/写/执行（3 bits each）
- 路径名vs文件名
  - 文件名只在它所在的目录中是唯一的
- 如何打开文件
  - 1.进程向操作系统提供路径名。
  - 2.操作系统在磁盘中查找目标文件的文件属性。
  - 3.磁盘返回文件属性。

    - 4.然后，操作系统将这些属性关联到一个数字，这个数字称为文件描述符。
    - 5.操作系统将文件描述符返回给进程。
    - 打开文件只涉及文件的路径名和属性，而不是文件内容!
    - 即使一个文件是由两个不同的进程打开的，但内核使用一个结构来维护它!
    - 文件描述符只是每个进程定位其打开的文件的数组索引。
- 如何读已打开文件
    - 1.该进程向操作系统提供一个文件描述符。
    - 2.操作系统读取文件属性，并根据存储的属性定位需要的数据。
    - 3.磁盘返回所需的数据。文件数据存放在内核中固定大小的缓存中。
    - 4.操作系统用数据填充进程提供的缓冲区。将数据写入用户空间缓冲区
- read()系统调用



- write()系统调用



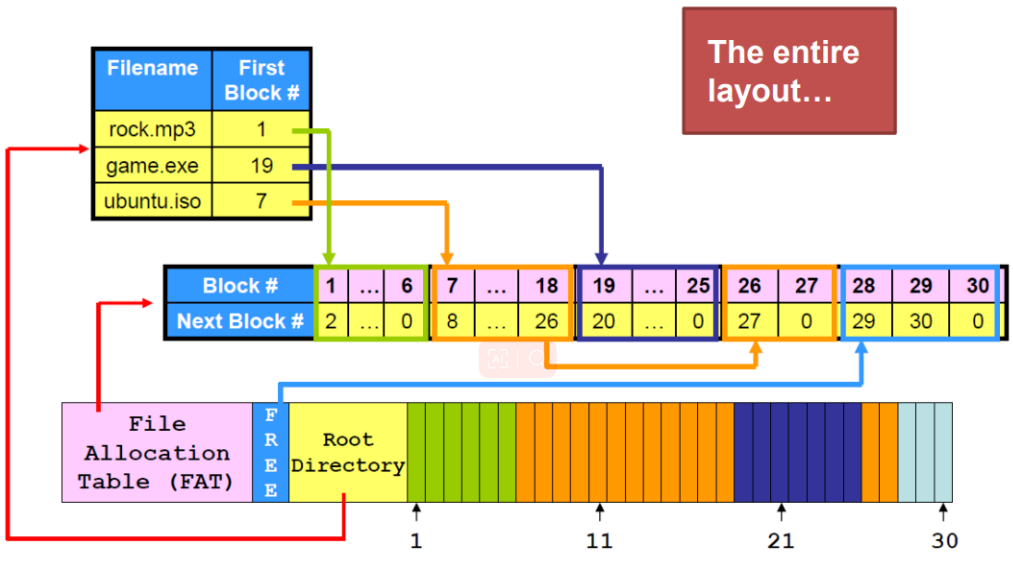


- 目录
  - 目录也是文件
  - 目录遍历过程
    - 步骤(1)：假设进程要打开文件“/bin/ls”。然后，该进程向操作系统提供唯一的路径名" /bin/ls "。
    - 步骤(2)：操作系统获取根目录“/”下的目录文件。
    - 步骤(3)：磁盘返回目录文件。
    - 步骤(4)：操作系统在目录文件中查找名称为“bin”的文件。
    - 步骤(5)：如果找到，在操作系统中检索目录文件“/bin”，使用“bin”的文件属性信息。
    - 步骤(6)：操作系统在目录文件“bin”中查找名称为“ls”的文件。如果找到了，那么操作系统就知道找到了“/bin/ls”文件，并开始打开该文件的过程“/ bin / ls
- 文件创建/删除
  - 文件创建==更新目录文件
  - 删除文件的过程与创建过程相反

Chapter 9 part2 - File System Layout

- 试验1.0-连续分配
  - 就像写书一样，从目录开始，称之为“根目录”
  - 但是，能在短时间内找到已分配的空间和空闲空间吗?
    - 需要O(n)次搜索，其中n是文件的总数。
  - 文件删除很容易!空间取消分配与更新根目录相同。
  - 缺点

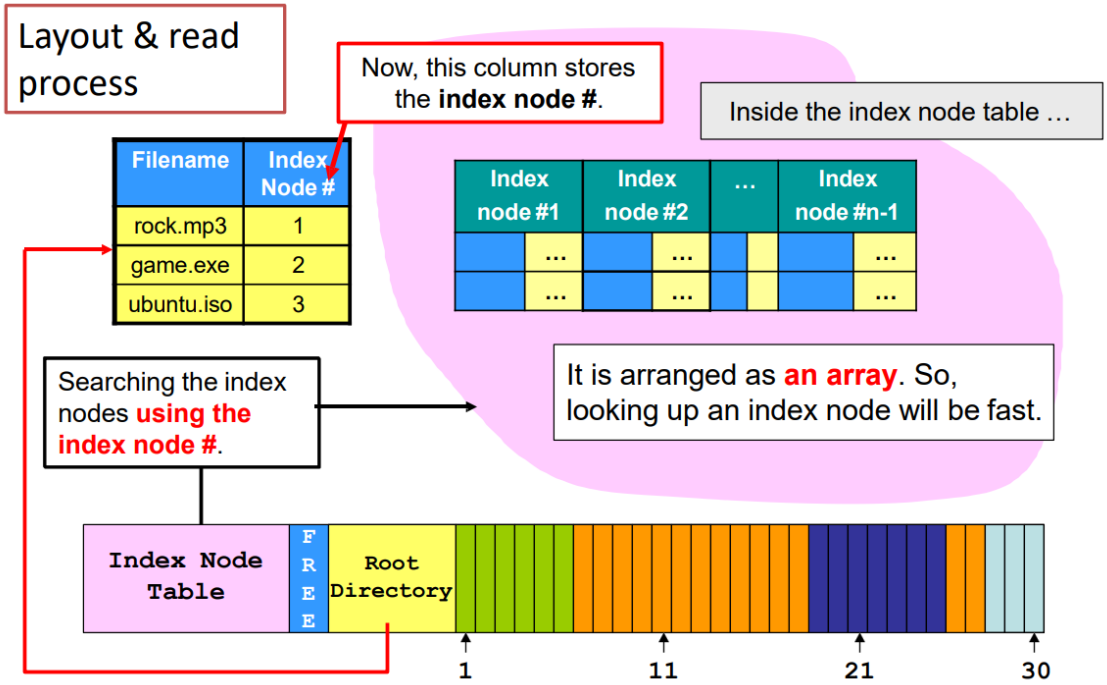
        - 我们有足够的空间，但是没有孔洞可以满足我的要求，即外部碎片。碎片整理过程可能会有所帮助。
        - 增长问题……没有空间让文件增长
    - 用途
        - 只读存储设备
- 试验2.0-链表分配
    - 实现方式
        - 步骤(1):将存储设备切成大小相等的块
        - 步骤(2):将新文件逐块填充到空白区域。
        - 步骤3:根目录...变得奇怪/复杂。
- 试验2.1-链表分配升级
    - 实现方式
        - 从每个块中借用4个字节
        - 将下一个块的块号写入每个块的这4个字节。
        - 注意，我们需要存储在根目录中的文件大小，因为文件的最后一块可能没有被完全填充。
    - 空闲空间管理
        - 将空闲块维护为链表
    - 优点
        - 解决外部碎片化问题。
        - 文件可以自由增长和收缩。
        - 自由块管理很容易实现。
    - 缺点
        - 随机访问性能问题。
        - 内部碎片。文件并不总是块大小的倍数。文件的最后一块可能没有被完全填满。
- 试验2.2-FAT
    - 文件分配表



    - 可以通过在内核中保留FAT的缓存版本来缓解随机访问问题

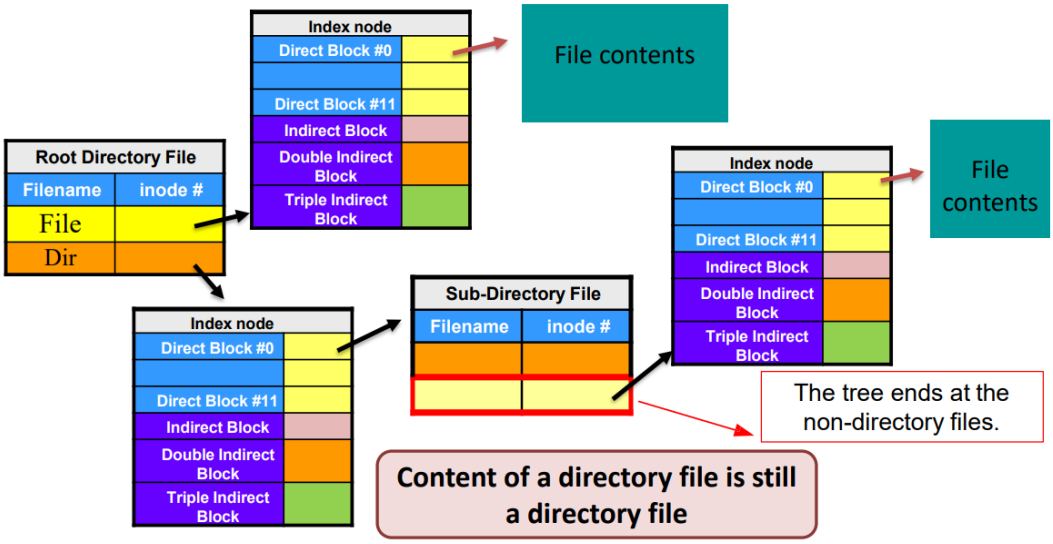
- FAT中的一个块被命名为一个簇

- 试验3.0-索引分配
  - 文件的数据块的位置存储在inode中
  - 挑战/问题
    - index管理
    - 采用固定大小的inode
  - 两种块
    - 数据块：存储文件数据
    - 间接块：存储一个块地址数组。一个地址既可以指向一个数据块，也可以指向另一个间接块。然而，在一个块中，所有的地址要么指向间接块，要么指向数据块。



  - 开销
    - 索引分配使用更多的存储空间，额外的东西是间接块。

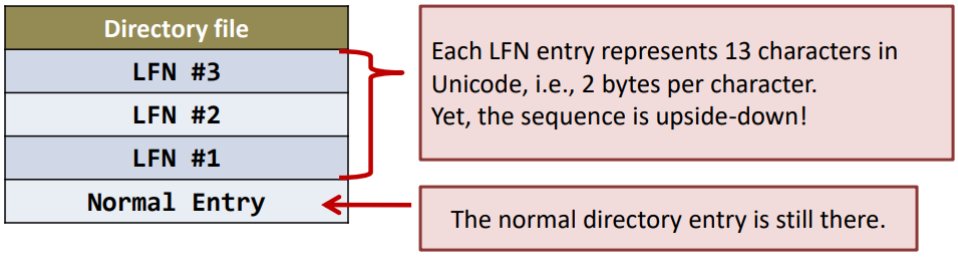
- 子目录
  - 以索引分配为例



- 文件系统信息
  - 例子
    - 一个块有多大?
    - 有多少已分配的块?
    - 有多少空闲的块?
    - 根目录在哪里?
    - 分配信息在哪里，例如FAT和inode表?
    - 分配信息有多大?
  - 它是一组重要的、FS特有的数据
    - 我们能在内核代码中硬编码这些信息吗?不! !因为不同的存储设备有不同的需求
    - 解决方案:解决方法是将这些信息保存在设备上。

Chapter 10 - Details of FAT32

- 目录遍历
  - 步骤(1)读取从cluster# 2开始的根目录的目录文件。
  - 步骤(2)读取目录文件“C:\windows”从cluster#123开始。
- 目录项
  - 目录项只是一个结构，一共32字节。
  - 目录项存储了文件的相关信息，比如文件名，文件属性，文件起始簇地址
- 大端对齐vs小端对齐
- LFN目录项
  - LFN：长文件名

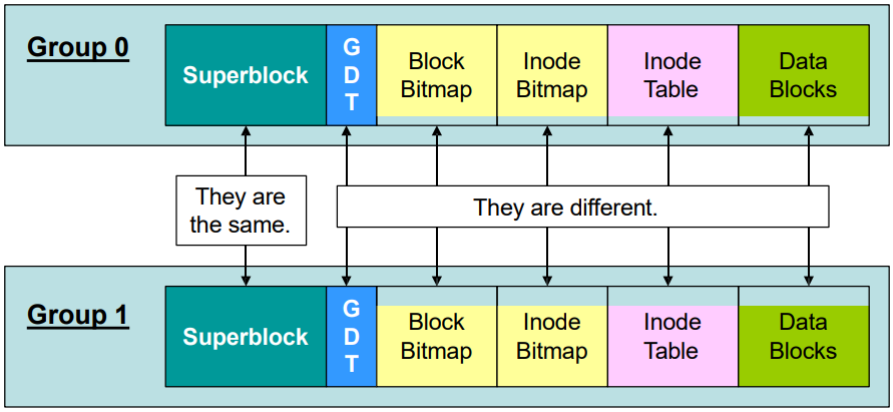


- 文件操作
  - 读文件
    - 先从文件起始簇读
    - 然后利用FAT表读下一个簇
    - 最后，由于FAT已标记为“EOF”，我们已到达最后一个簇。文件大小有助于确定从最后一个簇读取多少字节。
  - 写文件
    - 定位最后一个簇。
    - 开始向非满簇写入数据。
    - 通过FSINFO分配下一个簇。
    - 更新FAT和FSINFO。

        - 当写入完成时，更新文件大小。
    - 删除文件
        - 取消分配所有涉及的块。更新FSINFO和FATS。
        - 将目录条目的第一个字节更改为0xE5。LFN条目也得到相同的处理。这就是删除的结束。
        - 没有真正的删除。“删除的数据”一直存在，直到重新使用取消分配的簇。
    - 恢复被删除的文件
        - 拔下电源插头将阻止目标簇被覆盖

Chapter 10 part2 - Details of Ext2/3 File System

- Ext2/3文件系统遵循索引分配
- 特殊布局
    - 目的
        - 可靠性：超级块有很多副本，这就增加了FS的可靠性。
        - 性能：使inode和文件内容紧密地放在一起。特定组中的inode通常引用同一组中的数据块。



    - 组内布局
        - 超级块：存储FS特别数据
        - 块组描述符表：它存储块位图、索引节点位图和索引节点表的起始块编号。自由块数量，自由索引节点数量等...
        - 块位图：表示块是否被分配的位字符串。
        - 索引位图：表示索引节点是否被分配的位字符串。
- Inode结构
- 目录结构
- 链接文件
    - 硬链接
        - 是指向现有文件（inode）的一个目录条目。未创建新的文件内容
        - 从概念上讲，这会创建一个具有两个路径名的文件。
        - 特殊硬链接：“.”和".."
        - unlink()：链接数为0时，数据块和inode会被取消分配
    - 符号链接

        - 符号链接是一个文件，有新的inode
        - 存储目标路径名
- 内核缓冲缓存

  三种

Page Cache	It buffers the data blocks of an opened file.
Directory entry (dcache) cache	Directory entry is stored in the kernel.
Inode cache	The content of an inode is stored in the kernel temporary.

    - LRU算法管理
- 日志
    - 在更新磁盘时，在覆盖现有结构之前，首先写下一个小注释，描述您将要做的事情
    - 数据日志
        - 写日志：写入事务的内容(包括TxB、元数据和数据)
        - 日志提交：元数据和数据(包括TxE)
        - 检查点：将更新的内容写入磁盘上的位置
    - 元数据日志
        - 在将元数据写入日志的同时并行将数据写入磁盘
- VFS（虚拟文件系统）
    - 提供一种面向对象的方式来实现文件系统。定义了一个清晰的VFS接口，将文件的通用操作和具体实现分开。

Chapter 11 - I/O System

- 轮询
- 中断
- DMA