宇宙最全系列 | 操作系统面试题v1.0

作者：迹寒

写在前面

操作系统（包括计算机组成）是大部分开发岗位必考的内容，无论你做应用开发，内核开发，数据库开发，分布式系统开发，都离不开之。平时经常用的Linux，Windows，Mac或多或少也应该了解一些。

以往大家都是看一题，背一题，面试官问什么，我就复习什么，但这样是效率低下的。因为你没有在脑海中建立知识逻辑树，也就是「**试题背后的逻辑联系**」，所以迹寒将按照「**由浅入深，循序渐进**」的方式去帮助大家准备面试，具体而言：

- **基本概念**：并发与并行，虚拟化，局部性原理，原子操作等；
- **线程与进程**：包括进程与线程关系，进程地址空间，进程间通信，状态转换，调度等；
- **内核态**：内核态与用户态，内核态陷入，中断异常；
- **并发和锁**：同步和互斥，常见的锁，死锁产生的条件和预防；
- **内存管理**：分段和分页，内存交换，OOM，逻辑地址转换等；
- **Linux**：常见命令，软链接和硬链接，什么时候seg fault；
- **网络IO模型**：见专题；
- **其他**：作业和会话，ASCII和utf-8区别；
- **外部设备**：磁盘空间分配，RAID技术；
- **算法**：生产者消费者，读者写者，哲学家就餐，银行家算法。。。

面试复习过程一定要有**侧重点，全背=全不背。**对于常考的点（❤表示），一定在**理解**的基础上记忆，做到对答如流，这样面试官就会认为你基础很扎实。（理解的方法有：敲代码，看源码，看博客等）

赛道已铺好，只待尔努力！加油你就是offer收割机！

一 基本概念

一些操作系统的基本概念。

01 ❤并发和并行的区别。

并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。

并行需要硬件支持，如多流水线、多核处理器或者分布式计算系统。

操作系统通过引入进程和线程，使得程序能够并发运行。

02 ❤你知道局部性原理吗？各自有什么特点？

分为**时间局部性**和**空间局部性**。

时间局部性：程序执行代码以后，在一定时间后再次执行；如果数据被访问过，在一定时间后再次被访问；比如循环语句。

空间局部性：指访问的数据在内存中是连续存放的，比如数组。

03 ❤虚拟技术是什么？有哪些?

虚拟技术是把一个物理实体转换为多个逻辑实体的技术。

包括**时分复用**和**空分复用**技术两个大类。

时分复用：多个程序或者用户想访问同一资源。比如时间片轮转，多道程序设计。

空分复用，最典型的就是虚拟内存。

04 ❤什么是原子操作?

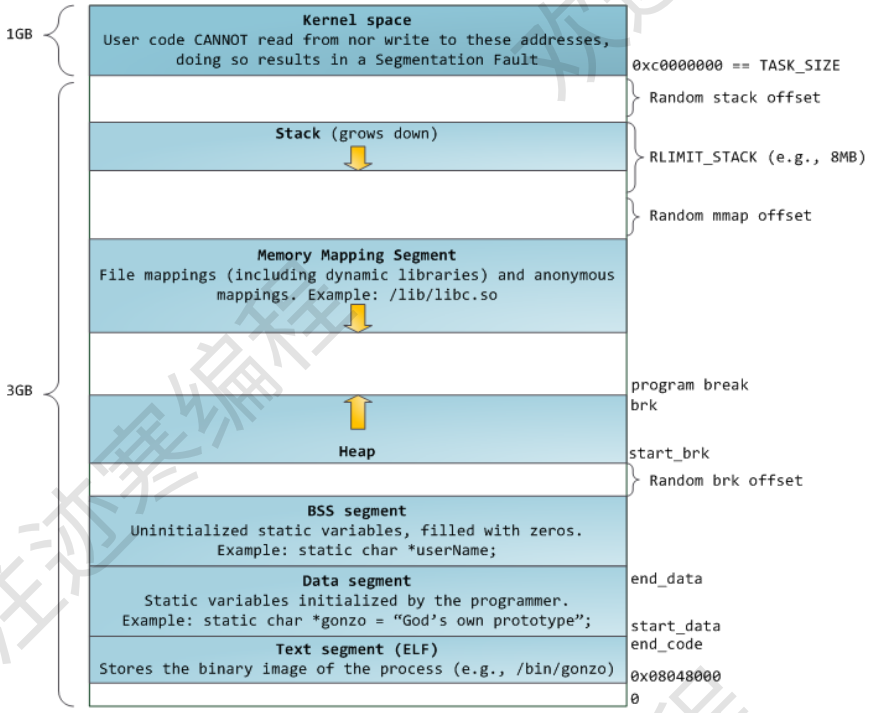
原子操作就是: 不可中断的一个或者一系列操作, 也就是不会被线程调度机制打断的操作, 运行期间不会有任何的上下文切换(context switch)，原子操作可以避免昂贵的上下锁开销。

二 线程和进程

线程与进程的相关考点。

01 ❤进程地址空间

虚拟内存为每个进程创造了一种独占系统内存空间的假象，通常分为用户空间和内核空间，一般用户空间包括代码段，数据段，BSS段，堆区，映射区和栈区。



进程地址空间

02 ❤进程、线程和协程之间的联系

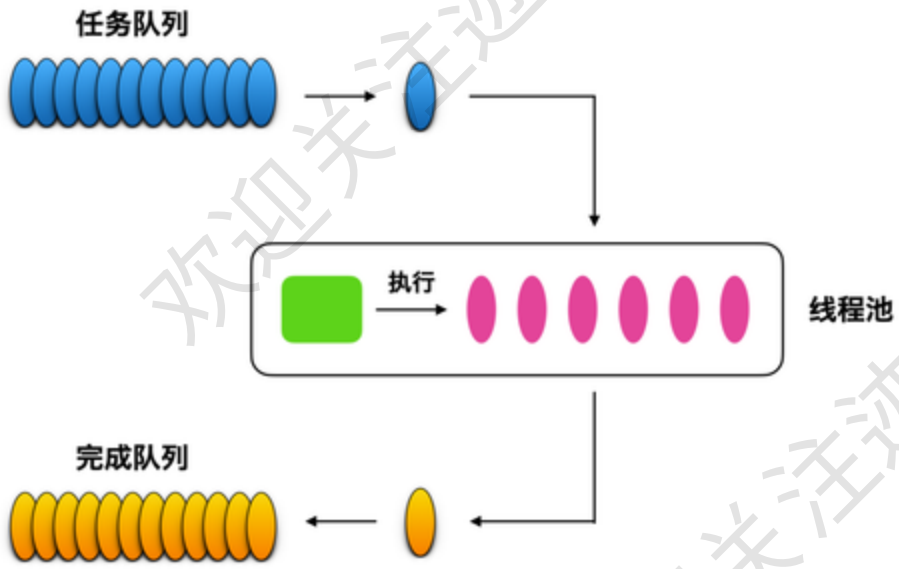
	进程	线程	协程
定义	资源调度的基本单位	CPU调度的基本单位	用户态轻量级线程，线程内部调度的基本单位
切换情况	进程CPU环境（栈，寄存器，页表和文件句柄等）的保存以及新调度进程环境的设置。	保存和设置程序计数器，少量寄存器和栈的内容	先将寄存器的上下文和栈保存，等切换回来再进行恢复
切换过程	用户态–》内核态–》用户态	用户态–》内核态–》用户态	用户态
调用栈	内核栈	内核栈	用户栈
并发性	不同进程之间实现并发，各自占用CPU时间片运行	一个进程内部多个线程并发执行	同一时间只能一个协程，而其它的协程处于休眠状态，适合对任务分时处理
系统开销	需要切换虚拟地址空间，切换内核栈和硬件上下文，系统开销很大。	切换时只需保存和设置少量寄存器内容，有一定的开销	直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，开销最小
通信	进程之间的通信需要借助操作系统	线程之间可以直接读写进程数据段，来进行通信	共享内存，消息队列
占用内存	依据所调用的资源大小	固定不变，由编译器决定	初始一般较小，可以自动扩展

03 一个进程可以创建多少线程，和什么有关?

一个虚拟内存空间一般为**4G（32位）**，默认情况线程栈大小为**1M**，理论上只能创建4096个线程。一个进程可以创建的线程数由**可用虚拟空间**和**线程栈**的大小共同决定，只要虚拟空间足够，那么新线程的建立就会成功。如果需要创建超过2K以上的线程，减小你线程栈的大小就可以实现了，虽然在一般情况下，你不需要那么多的线程。过多的线程将会导致大量的时间浪费在线程切换上，给程序运行效率带来负面影响

04 什么是线程池技术。

线程池技术提出是为了解决线程创建销毁的开销问题。线程池采用预创建的技术，在应用程序启动之后，将立即创建一定数量的线程(N1)，放入空闲队列中。这些线程都是处于阻塞（Suspended）状态，不消耗CPU，但占用较小的内存空间。当任务到来后，缓冲池选择一个空闲线程，把任务传入此线程中运行。当N1个线程都在处理任务后，缓冲池自动创建一定数量的新线程，用于处理更多的任务。在任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程都一直处于暂停状态，线程池自动销毁一部分线程，回收系统资源。

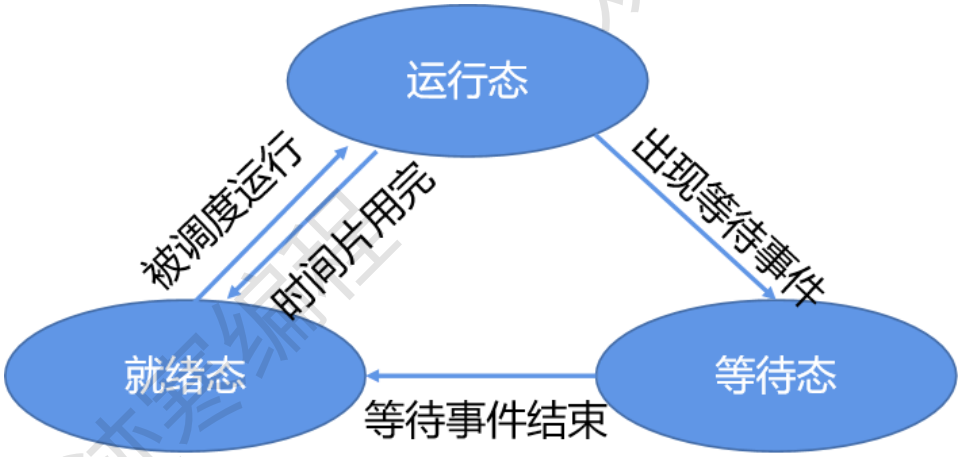


线程池示意图

05 ❤进程状态的切换你知道多少?

- 就绪状态Ready: 等待调度
- 运行状态Running

- 等待状态Waiting：等待资源，也称阻塞状态



进程状态机

06 ❤关于进程调度算法你了解多少?

- **先来先服务**(FCFS)

非抢占式的调度算法，按照请求的顺序进行调度，类似于队列。

- **短业务优先**(SJF)

非抢占式的调度算法，按照运行时间最短的业务优先。

- **最短剩余时间优先**(SRTF)

SJF的抢占式版本，如果新的进程的运行时间比当前进程剩余执行时间短，则发生抢占，挂起当前进程。

- **时间片轮转**

所有进程按照FCFS排成一个队列，每次调度时将时间片分给队首进程。当时间片用完，就由计时器发出时间中断，调度程序便停止该进程执行，并把它送到队尾，同时继续执行下一个时间片。

时间片过小，会导致调度频繁，时间片过大，又不能保证实时性。

- **优先级调度**

为每个进程分配一个优先级，按照优先级进行调度。

- **多级反馈队列**

如果一个进程需要100个时间片，那么按照时间片轮转算法，需要调度100次。多级队列是设置了多个队列，每个队列的时间片大小都不相同，进程在第一个队列没执行完就会被放入第二个队列，每个队列的时间片大小不相等，通常按照1,2,4,8...这样的比例。

07 ❤ Linux下进程通信的方式

- **管道通信 pipe**
- **有名管道（FIFO文件）** 有名管道以FIFO文件的形式存在于文件系统中，这样即使没有亲缘关系的进程也可以进行通信。有名管道也是半双工。
- **无名管道（内存文件）** 无名管道只能半双工通信，而且只能在拥有“亲缘关系”的进程进行，比如父子进程。
- **信号量semaphore**

信号量是一个计数器，通常用来控制不同进程对临界区资源访问的互斥和同步。

- **共享内存 shared memory**

共享内存就是**映射一段能被其他进程所访问的内存**，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的IPC(进程间通信)方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与信号量，配合使用来实现进程间的同步和通信。或者叫“内存映射”。

- **消息队列 message queue ，MQ**

消息队列是有消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

消息队列可以独立于读写进程存在，从而避免了FIFO中同步管道打开和关闭可能产生的困难。

避免了FIFO同步阻塞问题，不需要进程提供自己的同步方法。

读进程可以根据消息类型进行接受，而不像FIFO默认选择接受。

- **套接字socket**

用于不同机器间进程通信，在本地也可作为两个进程通信的方式。

- **信号signal**

用于通知接收进程某个事件已经发生，比如按下ctrl + C就是信号SIGKILL。Linux 下通过kill –l 可以看到全部信号。

08 讲一下线程之间的通信方式?

Linux下主要有四种线程通信方式：

1. 信号
2. 条件变量：使用通知的方式解锁，与互斥锁配合使用
3. 锁：互斥锁、读写锁、自旋锁
4. 信号量

09 ❤守护进程、僵尸进程和孤儿进程

- **守护进程**

指在**后台运行**，没有控制终端与之相连的进程，周期性的执行某一个任务。Linux大多数服务就是通过daemon进行，如web的http和crontab等。

- **孤儿进程**

如果**父进程先于子进程退出**，则子进程变为孤儿进程。此后孤儿进程将被init（进程号为1）收养，完成状态收集和退出工作。

- **僵尸进程**

如果**子进程先退出，而父进程并没有调用wait()或者waitpid()系统调用取得子进程的终止状态**，子进程将变为僵尸进程。

设置僵尸进程的**目的**是维护子进程的信息，以便父进程在以后某个时间点获取，包括进程pid，终止状态等。所以当终止子进程的父进程调用wait或waitpid时就可以得到这些信息。如果一个进程终止，而该进程有子进程处于僵尸状态，那么它的所有僵尸子进程的父进程ID将被重置为1（init进程）。继承这些子进程的init进程将清理它们（也就是说init进程将wait它们，从而去除它们的僵尸状态）。

10 如何避免僵尸进程?

- 通过signal(SIGCHLD, SIG_IGN)通知内核对子进程的结束不关心，由内核回收。如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的。
- 父进程调用wait/waitpid等函数等待子进程结束，如果尚无子进程退出wait会导致父进程阻塞。waitpid可以通过传递WNOHANG使父进程不阻塞立即返回。
- 如果父进程很忙可以用signal注册信号处理函数，在信号处理函数调用wait/waitpid等待子进程退出。
- 通过两次调用fork。父进程首先调用fork创建一个子进程然后waitpid等待子进程退出，子进程再fork一个孙进程后退出。这样子进程退出后会被父进程等待回收，而对于孙子进程其父进程已经退出所以孙进程成为一个孤儿进程，孤儿进程由init进程接管，孙进程结束后，init会等待回收。

第一种方法忽略SIGCHLD信号，这常用于并发服务器的性能的一个技巧因为并发服务器常常fork很多子进程，子进程终结之后需要服务器进程去wait清理资源。如果将此信号的处理方式设为忽略，可让内核把僵尸子进程转交给init进程去处理，省去了大量僵尸进程占用系统资源。

11 关于父进程创建子进程你了解多少?

父进程通过fork创建子进程后，除了pid不一样，其它全都一样。

父子进程共享数据，并不是说它们对同一块数据操作，子进程通过**写时复制RCU**技术从父进程拷贝数据并操作。

如果子进程需要运行自己的代码，可以调用execv()来重新加载。

子进程终止时会向父进程发送SIGCHLD信号，告知父进程回收自己，但该信号的默认处理动作为忽略，需要捕捉处理实现子进程的回收。

三 内核态

内核态和用户态，天生一对。

01 内核态和用户态有什么区别?

内核态可以使用所有指令（包括特权指令和非特权指令）而用户态只能使用非特权指令。权限从高到低分为R0,R1,R2,R3。R0对应内核态，R3对应用户态。Linux系统仅采用ring 0 和 ring 3 这2个权限。

02 ❤什么时候操作系统会陷入内核态?

系统调用、异常、设备中断。

03 什么是系统调用?

计算机的各种硬件资源是有限的，为了更好的管理这些资源，用户进程是不允许直接操作的，所有对这些资源的访问都必须由操作系统控制。为此操作系统为用户态运行的进程与硬件设备之间进行交互提供了一组接口，这组接口就是所谓的**系统调用**。

Linux的系统调用可以在 `/usr/include/x86_64-linux-gnu/asm/unistd_32.h` 中找到。

04 ❤中断和异常有什么区别？外中断和内中断有什么区别?

异常也称**内中断**，陷入（trap）指**CPU内部处理事件**，是由于CPU执行特定指令时出现的非法情况。 比如除数为0，或者地址越界，虚存系统的缺页。对异常的处理依赖于程序运行的现场，并且不能被屏蔽，一旦出现要立即处理。

中断通常称为**外中断**，来自于**CPU执行指令以外的事件**，比如设备I/O发出的中断，硬件故障中断。中断可以被屏蔽，也就是CPU可以选择不立即响应中断。

软中断是执行中断指令产生的，而硬中断是由外设引发的。

四 并发（同步&互斥）

并发之威力不可小觑。

01 操作系统的临界区知道吗?

每个进程访问临界资源的那段代码被称为“临界区”，临界区每次只允许一个进程进入。所以必须使用互斥量来进行同步和互斥。

02 操作系统并发源有哪些?

并发源有三种：

1. **中断处理**，当进程在访问某个临界资源的时候发生了中断，随后进入中断处理程序，如果在中断处理程序中，也访问了该临界资源。虽然不是严格意义上的并发，但是也会造成了对该资源的竞态；

2. **内核态抢占**：当进程在访问某个临界资源的时候进入了高优先级的进程，如果该进程也访问了同一临界资源，那么就会造成进程与进程之间的并发。

3. **多处理器的并发**：多处理器系统上的进程与进程之间是严格意义上的并发，每个处理器都可以独自调度运行一个进程，在同一时刻有多个进程在同时运行 。

03 Linux 下线程同步机制?

禁用中断

对于单处理器不可抢占系统来说，系统并发源主要是中断处理。因此在进行临界资源访问时，进行禁用/使能中断即可以达到消除异步并发源的目的。

自旋锁

自旋锁是当一个进程发现它申请的资源被锁住，会周期地不断的尝试申请获得锁。自旋锁可以保护共享数据。有普通自旋锁，读写自旋锁和顺序自旋锁。

信号量

可以对进程进行同步或者对线程进行同步，分为普通信号量，互斥信号量和读写信号量。

互斥锁mutex

Linux内核针对count=1的信号量重新定义了一个新的数据结构struct mutex,一般都称为互斥锁。内核根据使用场景的不同，把用于信号量的down和up操作在struct mutex上做了优化与扩展，专门用于这种新的数据类型。

读写锁

允许多个读进程同时访问临界区，但是对于写进程只能有一个。

RCU

RCU概念：RCU全称是Read-Copy-Update(读/写-复制-更新),是linux内核中提供的一种免锁的同步机制。详细原理建议参考：rcu 机制简介

04 ❤进程同步的方法

- **临界区**(critical area)

对临界区资源访问的那段代码称为临界区；

- **同步和互斥**

同步：多个进程因为合作有相互制约关系，有特定的执行顺序。例如生产者-消费者，哲学家进餐问题。

互斥：多个进程同一时刻只有一个能进入临界区。

- **信号量**

信号量是整型变量，有两种操作，V和P对应于加减操作：

- 当信号量大于0，调用P进行减一，当信号量等于0，进程睡眠
- 当信号量小于0，调用V进行加1，

原语P（或Wait()），V（或Signal()）均为原子操作，不可分割，通常会屏蔽中断。

如果信号量只能等于0，1，那么就是互斥量。分别表示解锁和上锁。

- **管程**

使用信号量需要很多控制代码，管程就把这些控制代码独立出来。

管程一次只允许一个进程使用管程，它是信号量的一种改进手段，管程引用了**条件变量**以及相关操作：wait()和signal()来实现同步操作，wait()适用于将进程阻塞，把管程让出来给另一个进程使用，而signal()则是唤醒阻塞的进程来使用管程。

05 条件变量是什么？和锁有什么区别?

条件变量(conditonal variable)是在多线程程序中用来实现“**等待→唤醒**”逻辑的常用的方法。条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待“条件变量的条件成立”而挂起；另一个线程使“条件成立”而发出信号。条件变量的使用总是和一个互斥锁结合在一起。

06 ❤介绍几种典型的锁?

- 读写锁

也称多读单写，适合某个变量或资源需要同时读，单次写，且读写互斥（写优先于读）的场景。

- 互斥锁

同一时间只有一个进程能获得互斥锁，其它进程只有等待。由于互斥锁涉及到线程状态的切换，即从运行状态变为阻塞状态，因此需要由操作系统管理，涉及到进程的上下文切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁。

- 自旋锁

自旋锁是当一个进程发现它申请的资源被锁住，会**周期地不断的尝试申请获得锁**。这种循环等待的策略就是自旋锁spinlock。自旋锁可以避免进程上下文切换的开销。但是长时间上锁，自旋锁就很消耗性能，阻止了其它线程的运行和调度。一般会设置一个自旋时间，到达一定时间后让自旋锁自动释放。

07 ❤乐观锁和悲观锁

所谓悲观锁就是以悲观的方式处理一切数据冲突。它以一种预防的方式先锁数据再释放锁。但是**每次都需要上锁和释放锁**，所以性能不高。

乐观锁则是对数据访问不加锁，认为别人不会修改数据。**只是在更新的时候看数据是否被修改；如果数据被修改则放弃操作**。

08 乐观锁实现方式

乐观锁主要实现方式有两种：**CAS**和**版本号机制**。

CAS：Compare and Swap

CAS操作包括了3个操作数：

- 需要读写的内存位置(V)
- 进行比较的预期值(A)
- 拟写入的新值(B)

CAS逻辑如下，如果V==A，则将该位置的值更新为B，否则不进行任何操作；许多CAS操作是自旋的

如果操作不成功，则一直重试，直到操作成功。

CAS是CPU支持的原子操作，其原子性是在硬件层面上保证的。

版本号机制

基本思路是在数据字段添加一个版本号。每次修改数据时，只有版本号一致才能修改，同时版本号加1。

优缺点和适用场景

乐观锁和悲观锁分别适用于不同的场景：

1. 与悲观锁相比，乐观锁**只能保证单个变量的原子性**，当涉及到多个变量时，CAS无能为力 。再比如版本号机制，如果query是针对表1而update是针对表2，很难通过版本号来实现乐观锁。
2. 考虑数据竞争的激烈程度，如果数据竞争不激烈，则乐观锁更有优势。否则悲观锁更有优势。

09 ❤死锁产生的条件?

定义：死锁是指两个或两个以上的**进程**在执行过程中，由于**竞争资源**或者由于**彼此通信**而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

产生条件：

1. **互斥条件**，指两个进程无法共享使用资源，如果一个在使用，另一个必须等待这个使用完。
2. **不可剥夺条件**，指一旦进程使用资源，就无法抢占式的获得资源。只能等使用进程释放。
3. **请求和保持条件**，进程拥有一种资源，但又申请另外的资源，且在申请成功之前不会释放当前进程占用的资源。
4. **循环等待条件**，存在资源的循环等待链，资源链上每个进程都得不到资源。

解决方案：

死锁预防：通过设置某些限制条件，破坏死锁产生的条件；

死锁的避免：系统对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源；如果分配后系统可能发生死锁，则不予分配，否则予以分配。这是一种保证系统不进入死锁状态的动态策略。

死锁检测和解除：

先检测：这种方法并不须事先采取任何限制性措施，也不必检查系统是否已经进入不安全区，此方法允许系统在运行过程中发生死锁。但可通过系统所设置的检测机构，及时地检测出死锁的发生，并精确地确定与死锁有关的进程和资源。检测方法包括定时检测、效率低时检测、进程等待时检测等。

然后解除死锁：采取适当措施，从系统中将已发生的死锁清除掉。恢复策略包括：

- 抢占式的恢复
- 利用回滚恢复
- kill进程恢复

这是与检测死锁相配套的一种措施。当检测到系统中已发生死锁时，须将进程从死锁状态中解脱出来。常用的实施方法是撤销或挂起一些进程，以便回收一些资源，再将这些资源分配给已处于阻塞状态的进程，使之转为就绪状态，以继续运行。死锁的检测和解除措施，有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。

死锁消极解法：因为解决死锁的代价很高，如果不采用任何措施（鸵鸟策略）会获得更高的性能，大多数操作系统都采用这种策略。

10 如何避免死锁?

操作系统里面讲到，破坏死锁产生的四个条件中的一个就可以（互斥、不可剥夺、循环等待、请求和保持），这里需要展开来说：

- 加锁的时候使用try_lock，如果获取不到锁则释放自身的所有的锁；
- 使用mutex加锁的时候按照地址从小到大进行顺序加锁；
- 将线程锁设置为 `PTHREAD_MUTEX_ERRCHECK` ，死锁会返回错误，不过效率较低。

五 内存管理

内存管理，合理分配资源，全面脱贫，共同富裕。

01 ❤分段和分页的区别?

- 段是**逻辑信息**的单位，一般是为了满足用户需要而制定；页是**物理信息**的单位，目的是减小外部内存碎片，提高内存利用率。
- 分段没有内部碎片，有外部碎片；分页则刚好相反。
- 页的大小固定而且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，页的大小是固定的。段的长度不固定，决定于用户所编写的程序。操作系统必须为每个进程维护一个段表，以说明每个段的加载地址和长度。

02 ❤什么是内部碎片和外部碎片?

内部碎片：**已经被分配出去**（能明确指出是哪个进程），但无法被利用的空间；一般常见于固定分配方式。

外部碎片：**还没有被分配出去**（不属于任何进程），但由于太小了，无法分配给新进程的内存空闲区域；一般见于动态分配方式。

03 ❤内存交换是什么?

把处于**等待**状态（或在CPU调度原则下被剥夺运行权利）的程序从内存移到外存，把内存空间腾出来，这一过程又叫换出。把准备好竞争CPU运行的程序从外存移到内存，这一过程又称为换入。**中级调度**（策略）就是采用交换技术。交换主要用于现代计算机。

04 ❤如何消除内存碎片?

对于外部碎片可用采用**紧凑技术**消除，就是操作系统定期对进程内存空间进行移动。

解决内部碎片的方案是**内存交换**。通过内存中程序交换到磁盘上，然后从磁盘读回到内存，重新整理空间。

05 ❤缺页异常(page fault)是怎么产生的?

CPU并不会直接和内存打交道，而是通过MMU（memory manage unit），将逻辑地址转换为物理地址。如果MMU没有找到对应的数据则会产生缺页异常，一般分为**硬缺页异常**和**软缺页异常**：

- 硬缺页异常：由于页表被交换到外部存储设备（如磁盘），物理内存中没有对应的页，系统进入内核态，从外部设备读取数据到物理内存中；
- 软缺页异常：物理内存是存在页帧的，只不过可能是其它进程调入的，发出缺页异常的进程不知道，此时MMU只需建立映射即可，一般出现在多进程共享内存的场景。
- 非法缺页异常：指进程访问的内存地址不在它的虚拟地址空间范围内，属于越界访问，会报segment fault。

参考：一切皆是映射：浅谈操作系统内核的缺页异常（Page Fault）

06 什么是major page fault 和 minor page fault?

major page fault也称为hard page fault, 指需要访问的内存不在虚拟地址空间，也不在物理内存中，需要从慢速设备载入。从swap回到物理内存也是hard page fault。

minor page fault也称为soft page fault, 指需要访问的内存不在虚拟地址空间，但是在物理内存中，只需要MMU建立物理内存和虚拟地址空间的映射关系即可。（通常是多个进程访问同一个共享内存中的数据，可能某些进程还没有建立起映射关系，所以访问时会出现soft page fault）

invalid fault也称为segment fault, 指进程需要访问的内存地址不在它的虚拟地址空间范围内，属于越界访问，内核会报segment fault错误。

page fault带来的性能问题–阿里云开发者社区.

07 什么是OOM，为什么会出现OOM?

oom是指系统已经没有足够的内存给进程使用，即能free的都已经free了，能swap out的也已经swap out了，再也不能挤出物理内存的情况。

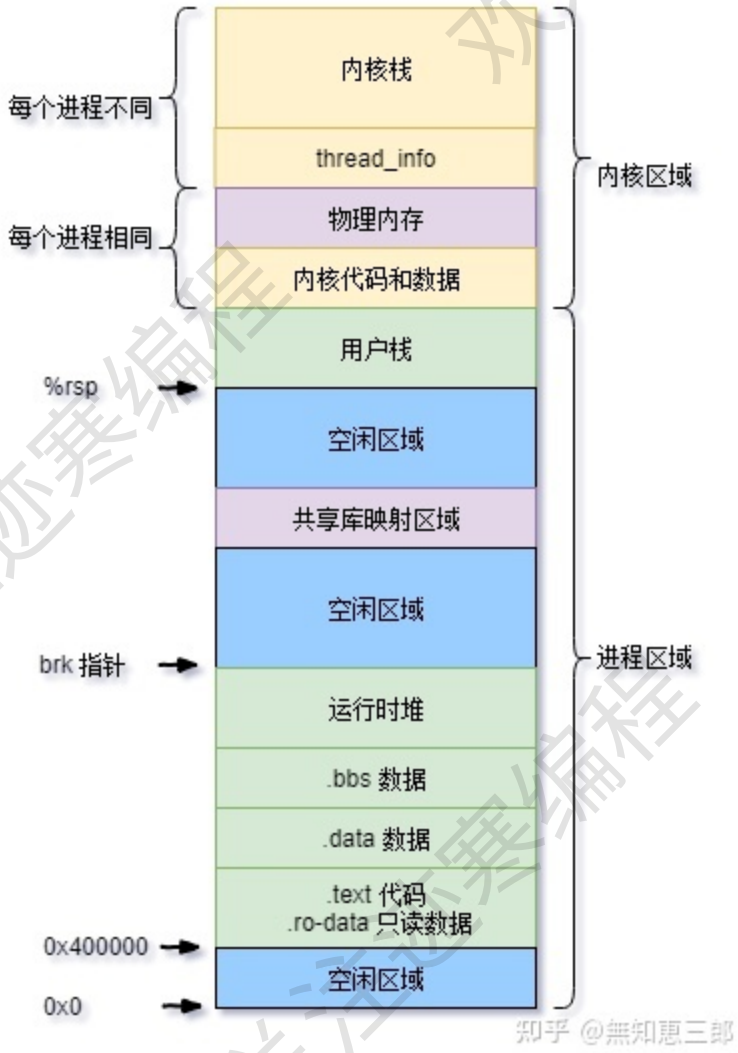
如果遇到这种情况就会发生OOM，表示系统内存以及不足，Linux会挑选并KILL一些进程，来释放内存。

参考：如果在线上遇到了OOM，该如何解决?

08 虚拟内存和物理内存的区别?

虚拟内存是为了解决物理内存不足而提出的概念，它利用磁盘空间虚拟出一段逻辑内存，被称为“交换空间”。

进程有了虚拟内存后，每个进程都认为自己拥有4G的内存空间。但实际上，虚拟内存对应的实际物理内存，可能只对应的分配了一点点的物理内存。



虚拟内存空间

09 什么是内存池技术?

通常人们使用new/delete来分配和释放空间，但这样会有一些额外的开销，比如需要“动态分区分配”，和加锁以实现互斥。为了避免这种开销，出现了内存池技术。核心思想是通过**系统内存分配预先申请一些适当大小的内存块组成链表。**每次需要的时候，从链表头部取出，用完再放回去。一般会按照内存块大小设计一系列链表。

10 ❤动态分区分配算法有哪些，可以分别说说吗?

所谓动态分区分配，是针对堆上内存分配而言的。

- **首次适应算法**

每次都从**低地址**开始查找，找到第一个满足大小空闲分区。空闲分区以地址递增的顺序排列，每次分配内存时顺序查找空闲分区链，找到能满足要求的第一个空闲分区。缺点是剩余区域容易产生内存碎片。

- **最佳适应算法**

每次优先使用小空闲区。我们将空闲区域按照其大小递增连接成一个链表，每次从头开始找到第一个满足要求的空闲区。

缺点：每次都利用小分区，会产生越来越多的内存碎片。

- **最坏适应算法**

每次优先使用大的空闲区。这样可以减小内存碎片，但当大进程进来的时候，没有内存可用了。

- **邻近适应算法**

每次从上一次分配区域的下一个分区开始搜索，同时将链表组织成循环链表。但它常常导致内存的末尾空间分裂成小碎片。

总结：首次适应算法，效果最好速度最快。但会产生内存碎片。

11 ❤讲一下从逻辑地址到物理地址的过程

Linux下，进程并不是直接访问物理内存，而是通过内存管理单元(MMU)来访问内存资源。如下图所示。

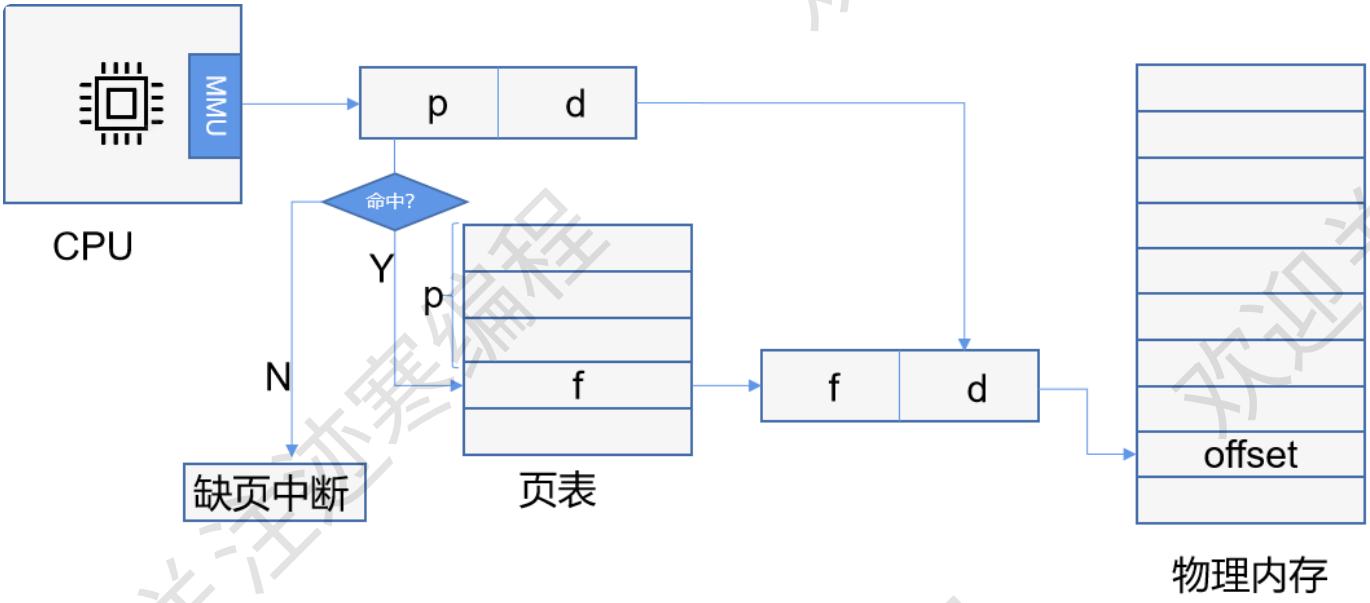
可以通过**页表**将逻辑地址转换为物理地址。

1. 根据逻辑地址计算出，页号和页内偏移量，
2. 判断页号是否存在于页表，若不存在报“缺页异常”错误。
3. 查询页表找到页号对应的块号
4. 通过内存块号和页内偏移量得到物理地址

注意：页面大小通常是2的整数次幂。

逻辑地址=页号+页内地址

物理地址=块号+块内地址

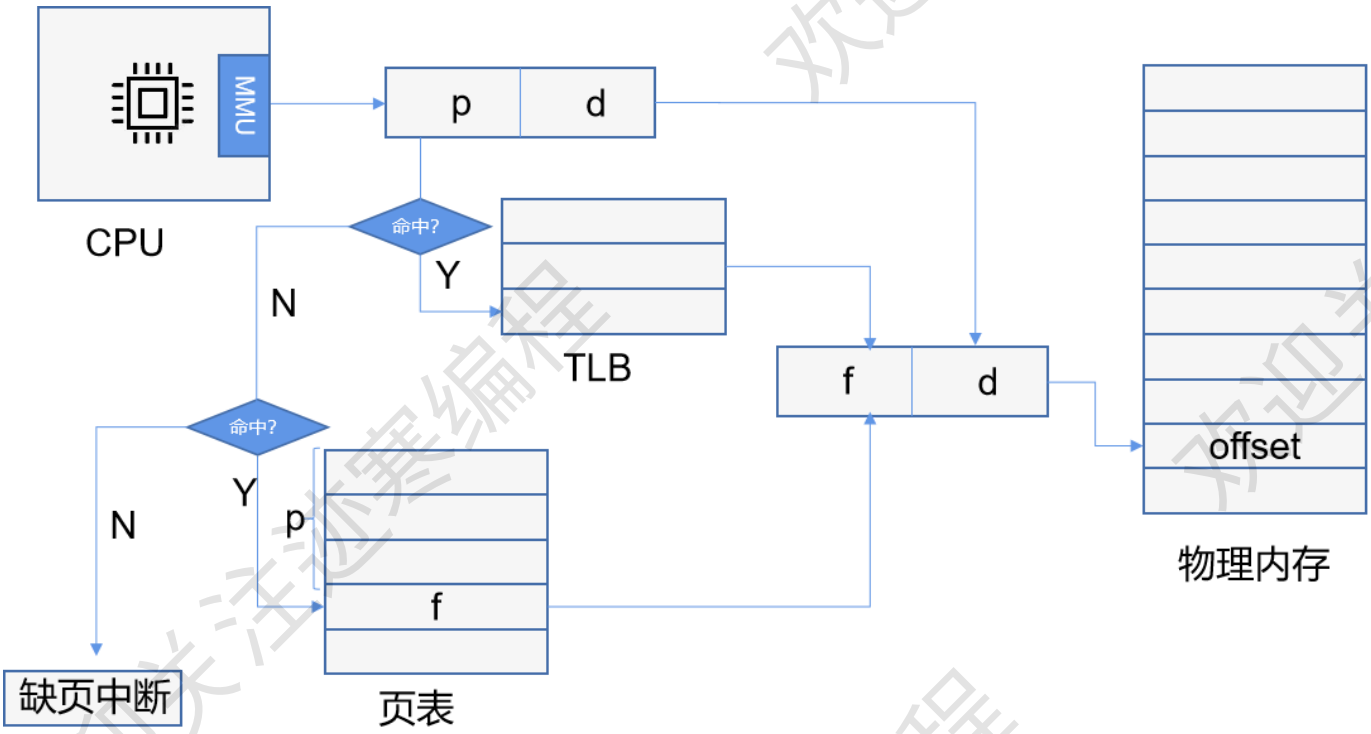


逻辑地址转换过程

12 如果系统中具有快表后，那么地址的转换过程变成什么样了？

快表是一种特殊的寄存器，正式名称translation lookaside buffer, TLB。页表只保存了**一小部分**页表中的条目，相当于页表的缓存。

由于查询快表的速度比查询页表的速度快很多，因此只要快表命中，就可以节省很多时间。 因为局部性原理，一般来说快表的命中率可以达到90%以上。



逻辑地址到物理地址转换（含快表）

13 缺页中断和缺页异常的区别?

缺页中断：在请求分页的过程中，如果访问的页面不再内存中，会产生一次缺页中断，在外存中找到所缺的一页将其调入内存。

缺页异常：由以下几种情况：

1. 请求地址不在虚拟地址空间中
2. 请求地址在虚拟地址空间中，但没有访问权限
3. 接上一条，没有与物理地址建立映射关系，比如fork等系统调用时并没有映射物理页，写数据→缺页异常→写时拷贝
4. 映射关系建立了，但在交换分区中

14 ❤你说一说页面置换算法有哪些?

缺页异常的处理过程，操作系统立即阻塞该进程，并将硬盘里对应的页换入内存，然后使该进程就绪，如果内存已经满了，没有空地方了，那就找一个页覆盖，至于具体覆盖的哪个页，就需要看操作系统的页面置换算法是怎么设计的了。

- **最佳置换算法**（OPT）

每次淘汰的页面是以后最长未使用的页面。虽然这个算法理论上可以得到最低的缺页率，但在工程上无法实现，因为无法预测之后访问页面的顺序。它常常作为其它页面置换算法的参照。

- **先进先出**（FIFO）

每次淘汰的页面是最早进入内存的页面。FIFO性能较差，因为最先进入内存的页面也有可能访问最频繁。采用FIFO算法时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多但缺页率反而提高的异常现象。

- **最久未被使用置换算法**LRU

least recently used。该算法记录每个页面自上次被访问后所经历的时间片的数量，当发生缺页中断时，淘汰最久未被访问的页面。需要专门的硬件支持，开销比较大。

- **时钟置换算法CLOCK**

为了解决LRU硬件开销大的问题，提出了CLOCK算法。它为每个页面设置一个访问位，初始化为0，再将内存中的页面都通过链表构成一个循环队列，CLOCK算法像时钟一样扫描这个队列。当某一页被访问时，将其访问位置1。当需要淘汰一个页面，只需从头检查各节点的访问位，如果是1，则将它置0；否则换出该页面。

若某一时刻所有的访问位为1，那么将所有访问位置0。因此置换一个页面最多经历两轮扫描。

此外还有时钟置换算法的改进。事实上，如果被淘汰的页面没有被修改过,就不需要执行I/O操作写回外存。只有被淘汰的页面被修改过时，才需要写回外存。

15 ❤你说一下缓存置换算法有哪些?

这里的缓存是一种思想，并不局限于CPU，内存或者磁盘。包括LRU, LFU, FIFO, ARC, 2Q。

- 最久未被使用算法LRU

least recently used。每次淘汰最久未被使用的页面。需要专门的硬件，开销较大。

- 最不经常使用置换算法LFU

least frequently used。LFU将页面按照访问的频次进行排序，优先淘汰访问频次低的。如果频次最低的页面不止一种。我们要选择淘汰最老的页面。如果一个页面开始有很高的访问频率但后面很少访问，该算法就不够灵活。

- 先进先出FIFO

是一种绝对公平的方式，但容易导致效率降低。因为先进入队列的请求可能经常被访问，这样做会导致经常被访问的页面被置换到内存或磁盘上，很快就发生缺页中断，降低效率。

- 自适应缓存替换算法ARC

Adaptive replacement cache。该算法结合了LRU和LFU的优势。当访问的数据趋于访问最近的内容，会更多的访问LRU List，这样会增加LRU的空间；当数据趋向于访问最频繁的内容，会更多地命中LFU。

- 2Q

two queues。 两个缓存队列，一个是FIFO另一个是LRU。当页面第一次被访问时，加入到FIFO队列中，当数据第二次被访问，将页面从FIFO移到LRU，同时按照各自的方式淘汰页面。

16 ❤什么是抖动?

指**页面的频繁调度**现象，如刚进内存的页面被调出去。产生抖动的原因是分配给进程的物理块不够。

六 Linux

为什么不是Windows？答：因为Windows不开源。

01 Linux的进程创建顺序。

Linux创建进程的过程为：0号进程 → 1号内核进程 → 1号用户进程(init进程) → getty进程 → shell进程 → 命令行执行进程。所以说用户命令行执行的程序全都是shell的子进程。

02 ❤讲一下静态链接和动态链接的区别？有什么特点？

静态链接：**在运行之前进行链接**。将一个库或多个库的目标文件链接到可执行文件中。相关的后缀为.a(linux),.lib(windows).

动态链接：**把链接这个过程推迟到运行时候再执行**，由操作系统加载库。相关的后缀为.so(linux),.dll(windows).

静态链接优点：

- 代码装载**速度快**，
- 只需保证开发者电脑中有正确的.lib文件就行，不需要考虑用户机上有无.lib文件

缺点：生成的可执行**文件体积大**，容易造成空间浪费。

动态链接的优点：

- 生成的可执行**文件体积更小**
- 独立性强，代码耦合度小，适合大规模软件开发
- 动态链接文件和可执行文件独立，极大降低代码维护的难度

缺点：客户机**必须有动态链接文件**才能运行，此外速度比静态链接慢。

实际中，动态链接用的更多。

03 ❤软链接和硬链接有何区别?

软链接又叫**符号链接**，这个文件包含了另一个文件的路径名。可以是任意文件或目录，可以链接不同文件系统的文件。可以理解为“快捷方式”。

硬链接就是一个文件的一个或多个文件名。把文件名和计算机文件系统使用的节点号链接起来。因此我们可以用多个文件名与同一个文件进行链接，这些文件名可以在同一目录或不同目录。

删除一个硬链接文件并不影响其他有相同 inode 号的文件。

删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软连接被称为死链接（即 dangling link，若被指向路径文件被重新创建，死链接可恢复为正常的软链接）。

相关操作：

`ln f1 f2` #创建从f1→f2硬链接

`ln -s f1 f2` #创建f1→f2软链接

`ls -al` #查看所有链接

04 linux下如何查看一个进程的所有线程?

可以采用ps命令ps – T –p <pid>，也可以用top，top –H –p <pid>

05 pkill和kill还有killall有什么区别。

kill是杀掉单个进程，killall是杀掉所有同名进程，pkill是杀掉一类进程或者某个用户的所有进程。

06 linux下如何查看文件夹的大小?

du –h –d 1， h表示转化为适合人读的格式，–d表示深度.

07 ❤linux中如何查看指定端口是否开放

1. `lsof –i :<port_num>`
2. `netstat` 命令(结合grep命令):

```bash
netstat -aptn // 显示所有端口
netstat -ntpl // 显示TCP端口
netstat -nupl // 显示UDP端口
netstat -ano|findstr 8080 // 显示8080端口占用情况
```

08 如何让进程在后台运行?

在Linux中，如果要让进程在后台运行，一般情况下，我们在命令后面加上&即可，实际上，这样是将命令放入到一个作业队列中了：

```bash
$ ./test.sh &

$ jobs -l
[1]+ 17208 Running ./test.sh &
```

对于已经在前台执行的命令，也可以重新放到后台执行，首先按ctrl+z暂停已经运行的进程，然后使用bg命令将停止的作业放到后台运行：

```bash
$ ./test.sh
[1]+ Stopped ./test.sh

$ bg %1
[1]+ ./test.sh &

$ jobs -l
[1]+ 22794 Running ./test.sh &
```

但 是如上方到后台执行的进程，其父进程还是当前终端shell的进程，而一旦父进程退出，则会发送hangup信号给所有子进程，子进程收到hangup以 后也会退出。如果我们要在退出shell的时候继续运行进程，则需要使用nohup忽略hangup信号，或者setsid将将父进程设为init进程 (进程号为1)

```bash
$ nohup ./test.sh &
$ setsid ./test.sh &
```

参考资料： Linux中如何让进程(或正在运行的程序)到后台运行?

09 如何知道在Linux/Windows平台下栈空间的大小。

Linux下由操作系统决定，可以通过 `ulimit -a` 来查看，通过 `ulimit -s` 来设置，默认8MiB。

Windows下一般由编译器决定，一般为1MiB。

10 ❤说说常用的Linux命令

1. cd命令：用于切换当前目录
2. ls命令：查看当前文件与目录
3. grep命令：该命令常用于分析一行的信息，若当中有我们所需要的信息，就将该行显示出来，该命令通常与管道命令一起使用，用于对一些命令的输出进行筛选加工。
4. cp命令：复制命令
5. mv命令：移动文件或文件夹命令
6. rm命令：删除文件或文件夹命令
7. ps命令：查看进程情况
8. kill命令：向进程发送终止信号
9. tar/unzip命令：对文件进行打包，调用gzip或bzip对文件进行压缩或解压
10. cat命令：查看文件内容，与less、more功能相似
11. top命令：可以查看操作系统的信息，如进程、CPU占用率、内存信息等
12. pwd命令：命令用于显示工作目录。
13. wget命令：下载一个链接。
14. apt/yum：下载应用程序包。

11 什么是作业？它和进程有什么关系？

shell分前后台来控制的不是进程而是作业（job）或者进程组（PG）。一个前台作业可以由多个进程组成，一个后台作业也可能由多个进程组成。shell可以运行一个前台作业和多个后台作业。

12 为什么只能运行一个前台作业？

我们在前台新启动了一个作业，那么shell就被放到后台，因此shell就无法继续接收指令并解析运行。

作业和进程组的区别：如果作业中某个进程创建了子进程，那么这个子进程是不属于作业的。一旦作业结束，shell就会把自己提到前台，如果子进程还没有终止，那么它将自动变为后台进程组。

13 什么是会话?

会话一个或多个进程组的集合。一个会话可以有一个控制终端。在Xshell或WinSCp打开一个窗口就是新建一个会话。

14 ❤写一个大型工程，什么时候会出现segment fault？如何检测?

出现segmentation fault一般有以下原因：

1. 访问不存在的地址。
2. 访问受保护的地址。
3. 访问只读的地址。
4. 栈溢出。

可以使用 `AddressSanitizer` 和Valgrind等工具检测内存泄漏和越界。

15 接上题，如何debug?

1. –g / gdb

在gcc/g++编译的时候加上–g参数可以使得生成的二进制文件中加入可以用于gdb调试的有用信息。gdb还可以attach到正在运行的程序，再用bt查看调用栈。

2. dmesg

可以在应用程序crash掉，显示内核中保存的信息。

3. nm

使用nm命令列出二进制文件中的符号表，包括符号地址、符号类型、符号名等，这样可以帮助定位在哪里发生了段错误。

4. ld

使用ldd命令查看二进制程序的共享链接库依赖，包括库的名称、起始地址，这样可以确定段错误到底是发生在了自己的程序中还是依赖的共享库中。

5. core文件

利用gdb <executable> <core_file> 来调试，当core很大的时候，该方法不可取。

6. objdump

使用objdump生成二进制的相关信息，重定向到文件中。

七 网络IO模型

超全的网络IO面试题【持续更新中】

不知何时起，面试题总少不了那几道网络IO题目。在高并发、百万连接的业务背景下，网络IO的重要性...

知乎专栏

八 其他

01 ❤ASCII，Unicode，UTF-8, UTF-16的区别?

ASCII 只有127个字符，表示英文字母的大小写、数字和一些符号。

Unicode就是将这些语言统一到一套编码格式中，通常两个字节表示一个字符，而ASCII是一个字节表示一个字符。

UTF-8 是unicode的“**可变长编码**”版本， UTF-8编码将Unicode字符按数字大小编码为1-6个字节，英文字母被编码成一个字节，常用汉字被编码成三个字节。UTF-16,任何字符都用两个字节来存储，处理起来比较快。

02 ❤大端和小端知道吗?

- 大端模式(big-endian)，高字节放在内存低的地址。
- 小端模式(little-endian)，高字节放在内存高的地址。

区别和联系：

(1) 在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码

(2) 用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件。

(3) 浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器.

03 ❤什么是文件描述符?

在Linux操作系统中，一切皆文件。为了将应用程序和文件对应，文件描述符(file discriptor, fd)应运而生。例如**0表示标准输入，1表示标准输出，2表示标准错误**。fd是非负整数。实际上它是一个索引值，指向内核为每个进程所维护的该进程打开的文件记录表。

九 外部设备

01 常见的磁盘调度算法

影响读写一个磁盘块的时间因素有：

1. 寻道时间（磁头移动到目标磁道）
2. 旋转时间（主轴转动盘面，使磁头移动到合适的扇区上）
3. 数据传输时间

其中寻道时间最长。

- FCFS

按照磁盘请求的顺序进行调度。优点是公平简单，但平均寻道时间长。

- 最短寻道时间优先SSTF

优先调度与当前磁头所在磁道距离最近的磁道。虽然平均寻道时间比较低，但是不够公平。如果新到达的磁道请求总是比一个在等待的磁道请求近，那么在等待的磁道请求会一直等待下去，也就是出现饥饿现象。

- 电梯扫描算法SCAN

磁头总是向一个方向移动，直到该方向没有请求。

扫描算法的平均响应时间比最短寻找楼层时间优先算法长，但是响应时间方差比最短寻找楼层时间优先算法小，从统计学角度来讲，扫描算法要比最短寻找楼层时间优先算法稳定。

02 磁盘空间分配的方式

连续分配、链接分配和索引分配

连续分配：每个文件在磁盘中占有连续的块，这样便于顺序访问和直接访问，磁盘寻道时间短。但这样需要事先知道文件分配空间的大小。

链接分配：每个文件是磁盘块的链表，可以充分利用磁盘空闲空间。但寻道时间较长，不支持直接访问。此外链表的指针也会占用空间。

索引分配：将指针聚集在一起，组成一个索引块。索引分配支持直接访问，并且没有外部碎片问题，因为磁盘的任何空闲块可以满足更多空间的请求。然而，索引分配确实浪费空间。索引块指针的开销通常大于链接分配的指针开销。考虑一下常见情况，即一个文件只有一块或两块。采用链接分配，每块只浪费一个指针的空间。采用索引分配，即使只有一个或两个指针是非空的，也必须分配一个完整的索引块。

03 RAID技术

RAID 0： 将数据按照条带化进行组织。

RAID1：将数据镜像复制一份。

RAID1+0：将数据镜像并按照条带化组织。

RAID0+1：将数据按照条带化组织并对条带镜像。

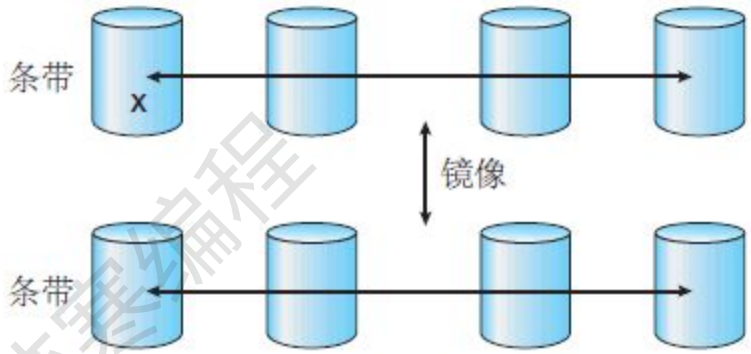
RAID2：带海明校验。

RAID3：带奇偶校验，使用单块校验盘。

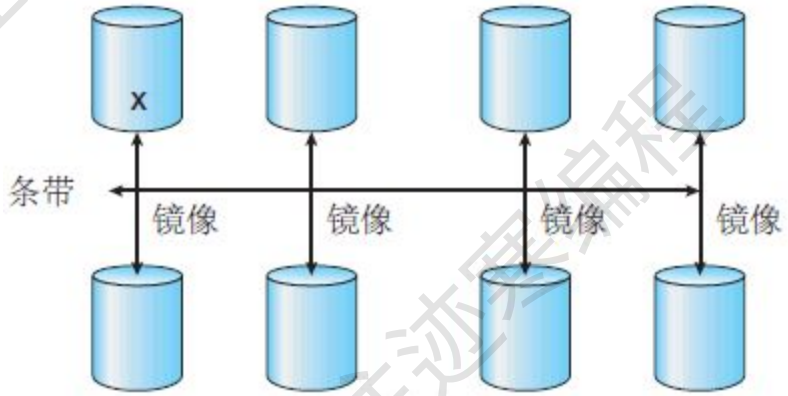
RAID4：它对数据访问是按磁盘来的。RAID3一次一横条，而RAID4一次一竖条。

RAID5：奇偶校验码存在于所有磁盘上，常用round-robin布局。

RAID6：有两种校验码，能容2错。



a) 单个磁盘故障的 RAID 0 + 1



b) 单个磁盘故障的 RAID 1 + 0

RAID技术存在的问题：

RAID并不总是保证数据对操作系统和使用者是可用的。

例如，文件指针可能是错的，或文件结构内的指针可能是错的。如果没有正确恢复，则不完整的写入会导致数据损坏。一些其他进程也会偶然写出文件系统的结构。RAID 防范物理媒介错误，但不是其他硬件和软件错误。与软件和硬件错误一样，系统数据潜在危险也有许多。

十 操作系统算法

01 如何使用信号量实现生产者–消费者模型?

问题描述：生产者往缓冲区填入数据，消费者从缓冲区取出数据。同时若缓冲区已满，则无法继续生产；缓冲区为空，则无法继续消费。

伪代码如下：

Bash | 复制代码

```
typedef int semaphore;
semaphore mutex = 1;//生产–消费是互斥过程
semaphore limit = N;//缓冲区上限
semaphore buff = 0;//缓冲区初始化为0
void producer(){
    while(true){
        P(limit)//A
        P(mutex)//B
        produce()
        V(mutex)
        V(buff)
    }
}

void customer()
{
    while(true){

        P(buff)
        P(mutex)
        consume()
        V(mutex)
        V(limit)

    }
}
```

上述的代码A，B两行是否能互换，为什么?

不能互换，否则容易生成死锁，当缓冲区满了之后，如果P(mutex)在前面，那么生产者仍能进入临界区，但此时需要满足P(limit)也就是缓冲区不满，但消费者不能进入临界区消费，导致生产者和消费者互相等待，造成死锁。

02 如何解决读者写者问题?

读者写者问题又称为多读单写问题。多个读进程可以并发，但读和写互斥，只有写进程完成之后才能读。这意味着最后一个读进程要解除锁。

一个数据变量count用于记录同时读操作的数目，count_mutex用于对count加锁。

Bash | 复制代码

```
#define reader N//the number of readers
#define writer 1//the number of writer
int count = 0;//the variable for writing
semaphore RD_mutex = 1;
semaphore WR_mutex = 1;
void read(){
    P(RD_mutex);
    count++;
    if(count == 1){ P(WR_mutex); }//第一个读者对写加锁
    V(RD_mutex);
    _read()...
    P(RD_mutex);
    count--;
    if(count == 0){ V(WR_mutex); }//最后一个读者释放写锁
    V(RD_mutex);

}
void write(){
    P(WR_mutex);
    count ++;
    V(WR_mutex);
}
```

03 如何解决哲学家进餐问题?

若干哲学家围绕一个圆形饭桌就座，哲学家只有两种活动：吃饭和思考。吃饭的时候需要同时拿起叉子和勺子才能进食。且相邻的哲学家共用一个叉子或勺子。

思路： 为了避免死锁产生，我们设置两个条件：

1. 相邻的哲学家不能同时吃饭
2. 必须同时拿起左右手的餐具才能进餐

有三种策略可供参考

1. 最多只允许N-1个哲学家同时进餐

```Bash
#define N 5//哲学家的数量
semaphore tableware[N];//第i个餐具的信号量
#define thinking 0
#define eating 1
semaphore limit = 4;//最多允许同时进餐的人数
void philosopher(int i){
    think();
    P(limit)
    P(tableware[i%N]);//请求左手的餐具
    P(tableware[(i+1)%N]);//请求右手的餐具
    eat()
    V(tableware[i%N]);//释放左手的餐具
    V(tableware[(i+1)%N]);//释放右手的餐具
    V(limit)
}
```

2. 仅当左右餐具都可以用时才进餐

```Bash
#define N 5//哲学家的数量
semaphore tableware[N];//第i个餐具的信号量
#define thinking 0
#define eating 1
semaphore mutex = 1;//保护信号量
void philosopher(int i){
    think();
    P(mutex);
    P(tableware[i%N]);//请求左手的餐具
    P(tableware[(i+1)%N]);//请求右手的餐具
    V(mutex);
    eat()
    V(tableware[i%N]);//释放左手的餐具
    V(tableware[(i+1)%N]);//释放右手的餐具
}
```

3. 奇数哲学家总是先拿左手餐具再拿右手餐具，偶数哲学家则刚好相反

```
#define N 5//哲学家的数量
semaphore tableware[N];//第i个餐具的信号量
#define thinking 0
#define eating 1
semaphore limit = 4;//最多允许同时进餐的人数
void philosopher(int i){
    if(i & 1)
    {
        think();
        P(tableware[(i+1)%N]);//请求右手的餐具
        P(tableware[i%N]);//请求左手的餐具
        eat()
        V(tableware[(i+1)%N]);//释放右手的餐具
        V(tableware[i%N]);//释放左手的餐具
    }
    else{
        think();
        P(tableware[i%N]);//请求左手的餐具
        P(tableware[(i+1)%N]);//请求右手的餐具
        eat()
        V(tableware[i%N]);//释放左手的餐具
        V(tableware[(i+1)%N]);//释放右手的餐具
    }
}
```

04 银行家算法

银行家算法由Dijkstra提出，它的目的是保证系统动态分配资源后是资源安全的，不会产生死锁。

比如我们现在有一组进程按照顺序，{P0,P1,P2,....Pn}执行。如果每个进程都能顺利执行，那么称此时系统的状态为安全状态，否则称为不安全状态。

银行家算法规定了如下的数据结构：

- 可用资源向量Available表示系统中各类资源的当前可用数目；
- 分配矩阵Allocate，每个进程对资源的当前占有量；

- 需求矩阵Need，它记录了每个进程当前对各类资源的申请量，等于最大需求矩阵与分配矩阵之差；
- 请求向量request，它记录了某个进程当前对各类资源的申请量，是银行家算法的入口参数；

银行家算法描述如下：

1. 如果request_i > Need_i，则进程P出错；
2. 如果request_i>available_i,则进程P阻塞；
3. 系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值

```bash
Available[j] = Available[j] - Requesti[j];
Allocation[i,j] = Allocation[i,j] + Requesti[j];
Need[i,j] = Need[i,j] - Requesti[j];
```

4. 系统进行安全性检查，检查此次资源分配后系统是否安全，若安全才完成本次分配；否则本次分配作废，让P_i进程等待。 这个安全性检查其实就是判断是否进程安全的过程，如果找到就返回true。 我们定义：

- 工作向量work，记录系统中各类只要当前可用数目，它是available的替身；
- 进程可完成标志向量finish;
  - i. 初始化 `work = available, finish=[false]`
  - ii. 若按照进程编号找到一个可以加入安全序列的进程，则假设该进程不久将完成任务归还资源 `work = work + allocation, finish_i = true`
  - iii. 否则， 若所有进程的可完成标志finish为真，则返回逻辑真，表示安全，否则返回false。

总结

操作系统类问题繁多而复杂，但**有规律可循，**以进程，线程，协程出发就引入了**并发**和**同步**的概念，此外还有通信的概念。操作系统需要做各种资源调度，比如**内存**和**IO资源，** 因此有了**内存分配**和管理，**虚拟内存**，**内存交换**等技术。对于IO资源，操作系统可以使用**IO多路复用技术**提高并发能力。此外由于进程之间的资源竞争可能出现**死锁**问题，如何避免和解决。

国外的操作系统课程

MIT 大名鼎鼎的6.828

清华大学的OS课程 ucore,视频在学堂在线和bilibili均有

清华大学用Rust从零开始写OS(rCore-Tutorial-v3)，代码，文档

南京大学 ICS PA

NJU ICS PA Bilibili

NJU OS

上海交通大学 操作系统 (陈海波、夏虞斌)

- 上海交通大学 SE315
- 视频课程（好大学在线）
- 对应教材 《现代操作系统——原理与实现》
- 配套 Lab

CMU CSAPP 对应的课程 15213

CMU 15410/605

Gate Lectures OS