这些是需要背诵的操作系统的content

A 线程、进程和同步

这三个东西是经常考的进程线程八股文，需要多背一背。

等到晚上回去以后，再了解一下项目经验。

一.进程与线程的概念与区别?

进程是系统进行资源调度和分配的基本单位，实现了操作系统的并发。

线程是进程的子任务，是CPU调度和分派的基本单位，用于保证程序的实时性，实现进程内部的并发。

线程共享的资源：共享代码段（代码和常量），数据段（全局变量和静态变量），拓展段（堆），每个线程拥有自己独立的栈，运行时段，用来存放所有的局部变量和临时变量。这也就是共享同一地址空间的意义（动态内存、映射文件、目标代码等）

二者的区别：进程系统开销较大，需要分配内存空间、IO设备等。进程切换时，涉及到进程CPU环境的保存和新的CPU环境的设置。线程切换只需要保存少量寄存器的内容，并不涉及存储管理。进程切换的开销远大于线程切换。

多线程通信简单，多进程通信IPC有很多复杂的机制，具体的我也了解的不是特别清楚。

二.进程间的通信方式

管道、系统IPC（消息队列、信号量、信号、共享内存）、套接字SOCKET通信。

1、管道

无名管道和命名管道。无名用于有亲缘关系的进程通信，命名管道FIFO用于无亲缘关系的进程通信。

普通管道PIPE:半双工，具有固定的读写端，仅用于父进程和子进程通信，特殊的文件，但是不属于文件系统，只存在于内存中。

命名管道FIFO：无关进程之间交互数据，拥有路径名，特殊的设备文件形式存在于文件系统中。

2、系统IPC

（1）消息队列MQ Message Queue

消息链接表，存放在内核中，有一个标识符队列ID来标记。克服了信号传递信息少，管道只能写入无格式字节流的问题，具有写权限的进程可以向消息队列添加某种格式的信息，具有读权限的进程可以从队列中读取信息。

面向记录，独立于发送和接收进程，不一定按照先进先出的次序读取，可以读取各种不同信息。

消息还可以具有优先级。

（2）信号量 semaphore

计数器，用于实现进程间的互斥和同步，不用于存储通信的信息。

可以用来控制多个进程对共享资源的访问。

基于操作系统的PV操作，访问时用P加锁，离开时用V解锁。

（3）共享内存

多个进程访问同一块内存空间，是最快的IPC直接对内存进行读取。

信号量+共享内存经常一起使用，这个和多线程的同步互斥有一定的想师资处。

（4）信号

较为复杂，用来通知进程某个事件已经发生？其实是有很多机制在这里面的。

（5）套接字

socket多进程通信，这个算是另一种形式的内容。

三、线程间的通信方式

临界区：多线程串行化访问同一个公共资源，速度快，适合数据访问。

互斥量：互斥机制，拥有互斥对象的线程才有访问公共资源的权限。

信号量：

条件变量？通知操作的方式保持多线程同步。

自旋锁：任何时刻只能有一个线程访问对象，倘若失败，会在原地自选，也就是无穷while循环，在某种情况下会极大地提升效率，加锁时间过长，会极大地浪费CPU资源。

四、有了进程，为何还要线程?

一个进程的开销是一个线程开销的30倍左右。

多线程的程序结构也有很多的好处。

多线程：线程切换代价小，适用于IO密集型工作场景。

多进程：适用于CPU密集型，适用于多台主机分布的模型，也是适用的。

单核机器上写多线程程序，仍然需要线程锁，防止共享资源被非法修改了。

五、互斥锁和读写锁

这二者有什么关系呢？暂时是不太清楚的。

互斥锁mutex：任意时刻有且仅有一个线程访问该对象

读写锁rwlock：读锁和写锁，读时，允许多个线程同时读。写时，只允许一个线程写，其他都会睡眠，直到被唤醒。用于读取数据的频率远大于写数据的场合。

linux的四种锁:互斥锁、读写锁、自旋锁、RCU（read-copy-update），在修改数据时，会对生成的副本进行修改，修改完成以后，再把老数据update一下。

六、死锁

两个或两个以上的进程在执行时，因竞争资源造成互相等待的现象。

死锁发生的四个必要条件：

互斥条件：进程对所分配的资源不允许别的进程访问，被一个进程已经占有。

请求保持条件：进程获取一定的资源后，又对别的资源请求，由于锁此时请求被阻塞，该进程不会释放资源。

不可剥夺条件：在未完成使用之前，不可被剥夺

环路等待条件：存在一个进程、资源之间的环形链

解决死锁的方法：资源一次性分配？可剥夺资源？资源有序分配法（按照请求编号有序的请求资源）

七、线程保持上下文，SP、PC、EAX寄存器的作用

SP：堆栈指针，指向当前栈的栈顶地址

PC：程序计数器，指向下一条要执行的指令

EAX：累加寄存器?

八、一个线程死掉会导致进程也死掉崩溃

九、fork和vfork的区别

fork创建新进程，之后调用exec()载入二进制映像，替换当前进程映像。

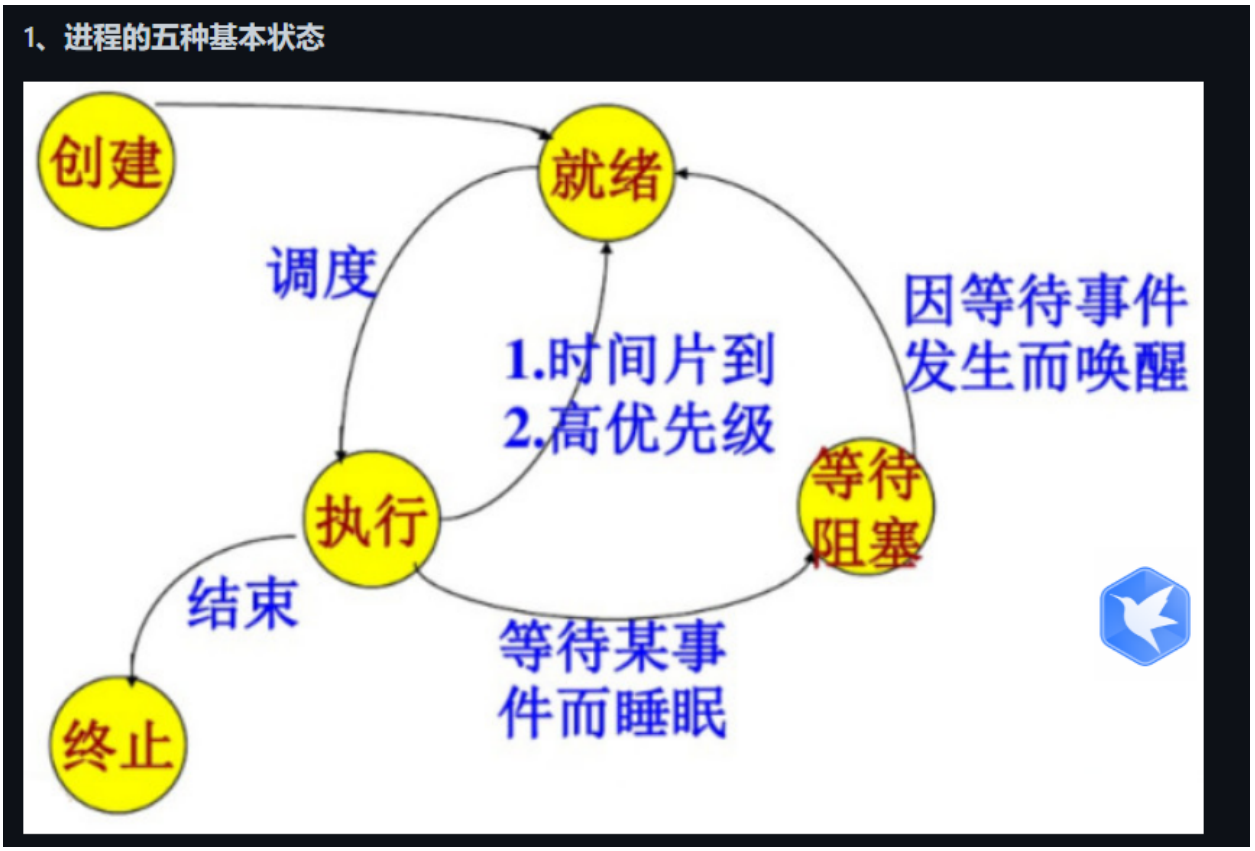
内部数据结构会全部赋值，现在则是写时复制。

vfork() fork之后，立即执行exec会造成地址空间的浪费？为什么?

避免了地址空间的按页复制。

由于写时复制的策略，地址空间会被直接交换复制，不会重复复制父进程的地址空间了。

十、进程的状态转换图



创建-》就绪态-》执行态-》等待阻塞-》终止态

交换技术：可能出现全部进程阻塞等待IO，由于IO处理速度慢，此时CPU就会空闲，没有就绪进程来执行。

以上问题有两种解决手段：

交换技术：换出一部分阻塞进程到外存（硬盘？），腾出内存空间

虚拟存储技术：每个进程只装入一小部分程序和数据

在交换技术上，将内存暂时不能运行的进程，或者不用的数据和程序，换出到外存（交换区？），腾出足够的内存空间，之后把能够运行的进程换入内存，从而出现挂起状态。

当进程被交换到外存时，进程状态就成为了挂起状态。

活动阻塞：进程在内存被阻塞 静止阻塞：进程在外存，被阻塞

活动就绪和静止就绪。

十一、协程

微线程，看上去也是子程序。

在执行过程中，内部可中断，转而执行别的子程序，之后再返回来执行程序。

相比于多线程，协程有极高的执行效率。

子程序切换由程序自身控制，没有线程切换的开销。

不需要锁机制，只有一个线程。

操作的一般手段：在协程上利用多核CPU：

多进程+协程 充分利用多核，发挥协程的高效率，最大程度上提高性能。

十二、僵尸进程

父进程利用wait或者waitpid来获得子进程的终止状态。

父进程退出，倘若它的子进程仍在运行，将成为孤儿进程，会被init进程收养，并且完成状态收集的过程。

僵尸进程：一个进程使用fork创建子进程，子进程退出，父进程没有wait，那么这些进程描述符仍然存在于系统，这些进程就叫僵尸进程。

这些资源会一直被占有，如果大量产生，将可能导致系统无法产生新的进程。

解决手段：

外部：kill 发送SIGTERM SIGKILL信号消灭主进程，它的子进程会成为孤儿进程。

内部：子进程退出时，向父进程发送SIGCHILD信号，父进程处理这个信号。

fork两次?

B、文件与内存

这里的信息大多是简单的机制，看不懂的再跳出来看一看喽。

一、虚拟内存

不同的程序在运行时，他看到自己独自占有了4G的内存。所有进程共享一个物理内存。

进程初始化时，只会创建虚拟内存的布局，初始化进程控制表中内存相关的链表。不会立刻把数据拷贝到物理内存中。只有当运行时，触发缺页异常，才会进行数据的拷贝。

二、程序的内存结构

这个只是程序的所有内容

分为BSS段（未初始化区，用于存放未初始化的全局变量和静态变量）、数据段（已初始化的）、代码段（程序的执行代码区域）

text和data已经固定，BSS段由链接器获取内存

在可执行时，一般还会有堆栈区，咱们用来存储东西。

栈区专门存储函数参数、局部变量等信息。

堆区专门存储动态分配的内存，malloc和free函数的操作位置。

三、缺页中断

当malloc时，可能出现物理内存没有加载页的情况，于是称为缺页中断。

四、缺页置换算法

先进先出（FIFO）算法

最近最少使用（LRU）算法，这里的LRU算法是非常经典的。

最近最少使用？每次访问的数据会移动到栈顶，倘若满了，会淘汰最后一个。

用特殊的栈来保存各个页面号。

LRU算法还是比较好理解的。

使用哪个，就把哪个提到最前面。

五、虚拟内存和物理内存映射

1.段式管理

段式内存管理单元，包含段标识符和段内偏移

2.页式管理

线性地址被划分为固定长度的组，叫做页。 4KB作为一个页来划分。

多级页表管理内存映射，还是比较ok的。

六、文件句柄

linux默认最大文件句柄1024，也就是文件标识符，最多可以同时打开的文件数量为1024。

七、结构体对齐、字节对齐

硬链接：一个文件的多个文件名

把文件名和计算机系统使用的文件节点连接起来。

inode号：文件系统的文件编号，这个信息在这里是固定的，还算是比较好理解的。

十一、大小端字节与主机/网络字节序

主机字节序，CPU字节序，不是由操作系统决定，是由CPU指令集决定

大端字节序：高序字节存储在低位地址

小端字节序：高位字节存储在高位地址

区分方法就是开头的字节是大字节还是小字节？这是一种存储的方式而已。

一个简单的区分手段：

32 位整数 `0x12345678` 是从起始位置为 `0x00` 的地址开始存放，则：

内存地址	0x00	0x01	0x02	0x03
大端	12	34	56	78
小端	78	56	34	12

其实高位字节序符合人类的阅读习惯。

高位指的是数字的高位，也就是数字一开头的各种信息，在这里是确定的。

各种处理器的字节序：

x86 小端序

ARM 默认小端序

网络字节序：大端序，默认是大端的排列方式。

这一点总是恒定的。

十二、静态变量初始化时间

静态变量存在数据段和BSS段，C语言在代码执行之前初始化，C++规定静态对象当且仅当初次用到时构造。

十三、内存溢出和内存泄漏

内存泄漏：由于疏忽，程序未能释放掉不再适用的内存，失去了对该段内存的控制，因而造成了内存的浪费。

未将基类的析构函数设置为虚函数，当基类指针指向子类对象时，不会自动调用子类的析构函数。于是，子类的资源没有被正确释放，由此带来的内存泄漏，即为内存泄漏的一种情况的。

十三、操作系统如何设置page cache

也就是TLS内存加速包？是另一种缓存，具体的实现方式不是现在要考虑的。

C、用户/内核态

用户态和内核态的特权级别不同。内核态拥有较高的特权优先级。

运行在用户态的程序不能访问操作系统的内核数据结构和程序。

二者转换的方式：系统调用、异常和中断

二、系统调用

system call，又叫做系统呼叫。用户态程序向内核请求更高权限运行的服务。提供了用户程序和操作系统之间的接口。

设备IO、进程通信：运行在内核态下面

危险的、权限很高的指令被包装为系统调用，这些资源都由操作系统控制，唯一的操作入口就是系统调用。

open、write、fork、execve等均是有名的系统调用。

三、内核态和用户态的相互转换

系统调用、异常（段错误、缺页异常）、外围设备中断（硬盘IO读写完成以后，需要中断来处理一下）

至于这个具体的中断流程是不需要特别考虑的，总而言之就是把寄存器内容存起来，执行完中断以后，再把各个寄存器的值拉回来。这个过程是非常好理解的，简单的破坏后再恢复现场即可。

D、编译的流程

编译我们曾经学过编译原理，但是具体的流程还没有了解清楚的。

一、编译流程

1.预编译：删除所有#define，处理条件编译，处理#include指令

2.编译：将预编译以后的文件进行词法分析、语法分析、语义分析、优化、生成目标代码，生成汇编语言。

3.汇编：将汇编代码编程机器码文件，产生.*o文件或者*.obj文件，这个是明确的。

4.链接：将不同的汇编后的目标文件进行链接，从而形成一个可执行程序。

链接是将汇编好的源文件链接起来，形成可执行程序的过程。

静态链接和动态链接，这是非常容易理解的。

E、IO

这里主要涉及5种IO模型

阻塞IO：非阻塞IO（每隔一段时间检查一下）：信号驱动IO(SIGIO信号)：IO复用、多路转接IO

多线程、线程池、IO多路复用?

提前建好线程池，用生产者消费者模型，创建任务队列，有任务就挂在任务队列上，同时开启对应的一个线程。

如果没有的话，就让所有线程睡眠。

实现线程池的方法：?

设置生产者消费者队列，作为临界资源。

初始化n个线程，并且运行起来，加锁去队列获取任务运行。

当任务队列为空，所有线程阻塞。

当生产者队列来了一个任务后，先对队列加锁，把任务挂到队列上，然后用条件变量去通知阻塞中的某一个线程。

F、其他

一、微内核和宏内核

宏内核:除了最基本的进程、线程管理、内存管理，将文件系统、驱动、网络协议集成在内核里面。

微内核：内核仅有基本的调度、内存管理。 驱动、文件系统由用户态的守护进程实现。

二、linux中的Timer定时器机制

可以直接忽略这里的定时器机制，稍微有点复杂。

大致把这些操作系统的知识看了一小遍，看得人头晕眼花。