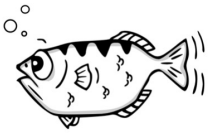
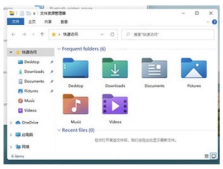
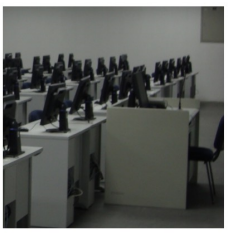
操作系统笔记

	目标纯粹 必须管理一台计算机	目标中立 管理计算机硬件就行	目标混乱 管理什么都行
对象纯粹 必须服务二进制代码	 Windows/Linux/vxWorks 当然是操作系统	 固件也是操作系统	 gdb也是操作系统
对象中立 服务对象是程序就行	 浏览器/微信/支付宝/JVM 都是操作系统	 Hadoop也是操作系统	 资源管理器也是操作系统
对象混乱 服务谁都可以	 机箱是操作系统	 机房也可以算操作系统	 校长为什么不是操作系统?

一、概述

1. 开机：CPU -> 执行BIOS内的“很小的自举装入程序“ -> 找到引导块并装入内存 -> 执行引导块中的“完整的自举装入程序” -> 读入FAT表的块 -> 读入根目录的块 -> 然后就可以创建文件或者运行程序了。
2. 计算机系统（Computer System）自下而上分为：硬件（hardware）操作系统（Operating System）应用程序（application）用户（User）
3. 操作系统的主要功能是什么?
    i. 处理机管理（processor management）（包括进程控制、进程同步、进程通信、调度）
    ii. 存储器管理（Memory）（内存分配（静态/动态/连续/非连续分配）、地址映射（逻辑地址-->物理地址）、内存扩充（虚拟存储技术））
    iii. 设备管理（device）（缓冲管理、设备分配、设备处理（启动设备、中断设备））
    iv. 文件管理（file）（文件存储空间管理、目录管理、文件读/写保护
4. 操作系统的特征（characteristic）
    i. 并发 concurrence：可以在同一时间间隔处理多个进程，需要硬件支持（并发会产生一种并行的错觉）

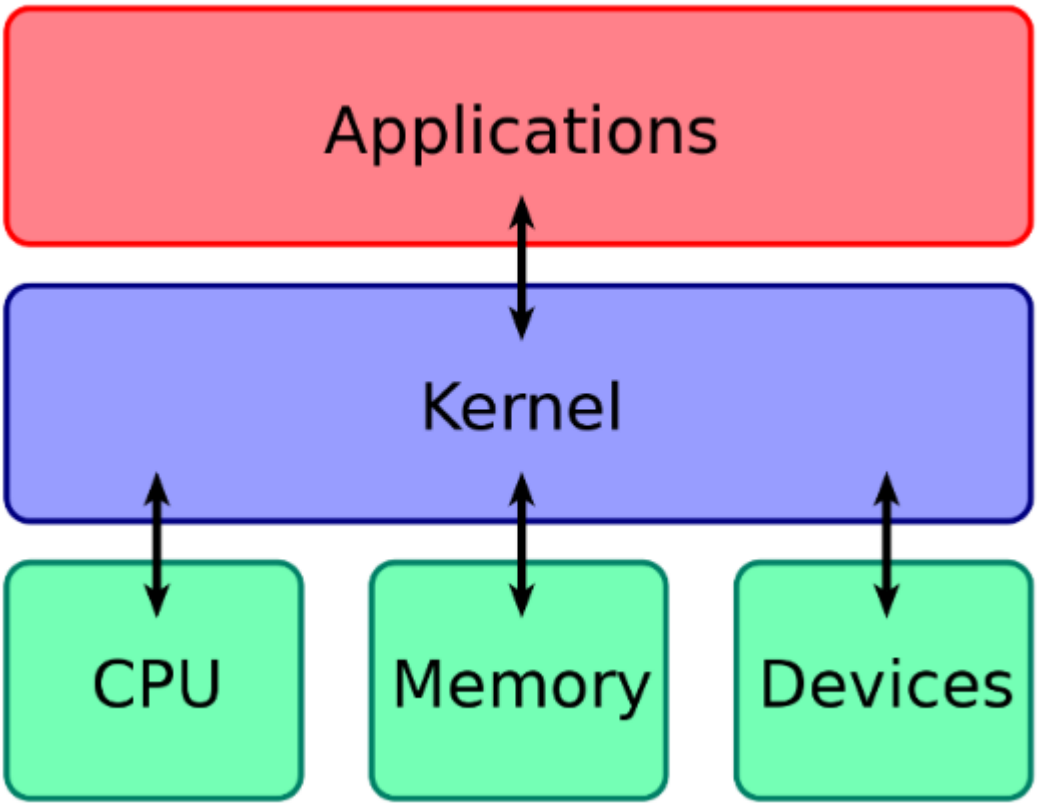
ii. 共享 sharing：资源可被多个并发执行的进程使用
iii. 虚拟 virtual：将物理实体映射成为多个虚拟设备
iv. 异步Asynchronism：[æˈsɪŋkrəˌnɪzəm] 进程执行走走停停，每次进程执行速度可能不同，但OS需保证进程每次执行结果相同

5. **OS/Computer developing process**
   i. 无操作系统（人工操作方式）：用户独占、CPU等待人工；
      a. **1940s** ENIAC 逻辑门：真空电子管 存储器采用延迟线(delay lines) IO采用打孔纸带
      b. 能跑起来程序已经很牛逼了，程序直接使用指令操作硬件，无需画蛇添足的程序进行管理
   ii. 单道批处理：内存只保存一道作业，多用户排队共享计算机
      a. **1950s** 逻辑门：晶体管；内存：磁芯；IO速度严重比cpu慢，中断机制诞生；Fortran诞生
      b. 批处理系统 = 程序自动切换（换卡片/程序）+ 提供库函数API
   iii. 多道批处理：同时将多个程序载入内存，且可以灵活调度
      a. **1960s** 出现集成电路、总线；内存更大更快 -> 可以同时载入多个程序到内存，而无需换卡了；更丰富的IO设备；现代os诞生: Multics(MIT1965)
      b. 有了process的概念；丰富了进程管理的API
      c. 进程执行IO时，可以将cpu让给另一进程
         a. 虚拟存储使得多个进程之间进行地址隔离，防止一个程序的bug干掉整个系统
      d. 基于中断机制（eg时钟中断），os的调度策略进行程序定时切换
   iv. **1970s** 基本具备了现在能干的所有事情：CISC指令集, PC, IO, 中断, 异常, 网络, PASCAL(1970), C(1972), Apple...
      a. UNIX(1969): 管道 grep socket procfs
         a. BSD(1977) GNU(1983) MacOS(1984) Windows(1985) Linux(1991) Debian(1996) Ubuntu(2004) iOS(2007) Android(2008) win10(2015)
            - > Linux is a kernel of OS, while GNU/Linux(Linux kernel + GNU project) is the whole OS.
            - > GNU include GCC(GNU Compiler Collection, for C), shell(Bash), etc, but do not have a kernel.
   v. **今天**的os：空前复杂, cpu, memory, io device, 应用需求更复杂
   vi. ~~分时：及时接收、及时处理，交互性~~
   vii. ~~实时：实时控制、实时信息处理~~

6. 特权指令Privileged instructions：IO指令、置中断指令 ----核心态（管态、内核态）kernel mode（操作系统在控制CPU）
   非特权指令：访管trap指令----用户态（目态） User mode（普通应用控制CPU）
7. 中断Interruption过程：关中断--保存断点--引出中断服务程序----保存现场--开中断--执行中断服务程序--关中断--恢复现场--开中断--中断返回
8. 系统调用：

i. 运行在用户态的程序向os请求需要更高权限运行的服务，系统调用提供用户程序和os之间的接口。
ii. 系统调用由os核心提供，运行在核心态，而普通函数的调用由函数库或用户自己提供，运行在用户态。
iii. 凡是与资源有关的操作都必须通过系统调用方式向os提出请求。

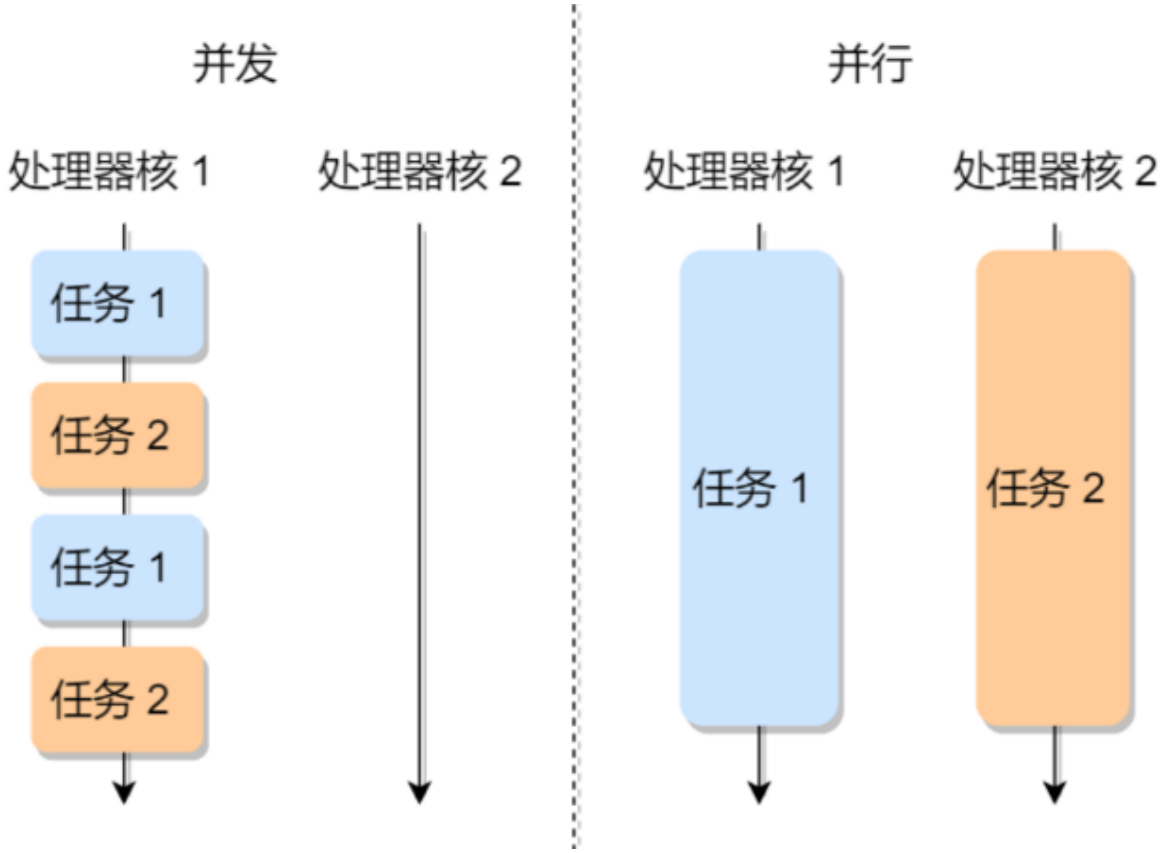
OS结构



1. 内核：计算机是由各种外部硬件设备组成的，比如内存、cpu、硬盘等，如果每个应用都要和这些硬件设备对接通信协议，那这样太累了，所以这个中间人就由内核来负责，让内核作为应用连接硬件设备的桥梁，应用程序只需关心与内核交互，不用关心硬件的细节。
2. 内核的基本能力：
    i. 进程调度：管理进程、线程，决定哪个进程、线程使用 CPU；
    ii. 内存管理：决定内存的分配和回收；
    iii. 硬件通信：管理硬件设备，为进程与硬件设备之间提供通信能力；
    iv. 提供系统调用：如果应用程序要运行更高权限运行的服务，那么就需要有系统调用，它是用户程序与操作系统之间的接口。
3. 内核空间vs.用户空间
    i. 内核空间：只给内核程序访问；用户空间：专门给应用程序访问；
    ii. 用户空间的代码只能访问一个局部的内存空间，而内核空间的代码可以访问所有内存空间；
    iii. 当进程/线程运行在内核空间时就处于内核态，而进程/线程运行在用户空间时则处于用户态。

iv. 应用程序要进入内核空间，需要通过系统调用：当应用程序使用系统调用时，会产生一个中断。发生中断后，CPU会中断当前在执行的用户程序，转而跳转到中断处理程序，也就是开始执行内核程序。内核处理完后，主动触发中断，把CPU执行权限交回给用户程序，回到用户态继续工作。

4. Linux内核设计理念：
   i. MultiTasK
      a. 并发concurrency：对于单核 CPU 时，可以让每个任务执行一小段时间，时间到就切换另外一个任务，从宏观角度看，一段时间内执行了多个任务。（虽然任务看起来是同时执行的，但实际上是通过快速地在任务之间切换来实现的，每个任务在任意时刻只能执行一部分）（**并发会产生一种在并行的错觉**）
      b. 并行parallelism：对于多核 CPU 时，多个任务可以同时被不同核心的 CPU 同时执行。
      c.



   ii. 宏内核：宏内核的特征是系统内核的所有模块，比如进程调度、内存管理、文件系统、设备驱动等，都运行在内核态。（Linux）
      a. 微内核架构的内核只保留最基本的能力，比如进程调度、虚拟机内存、中断等，把一些应用放到了用户空间，比如驱动程序、文件系统等。（稳定、可靠；但频繁切换到内核态，损耗性能；eg鸿蒙）
   iii. ELF 可执行文件链接格式
   iv. SMP 对称多处理
5. windows内核
   i. multiTask
   ii. SMP 对称多处理

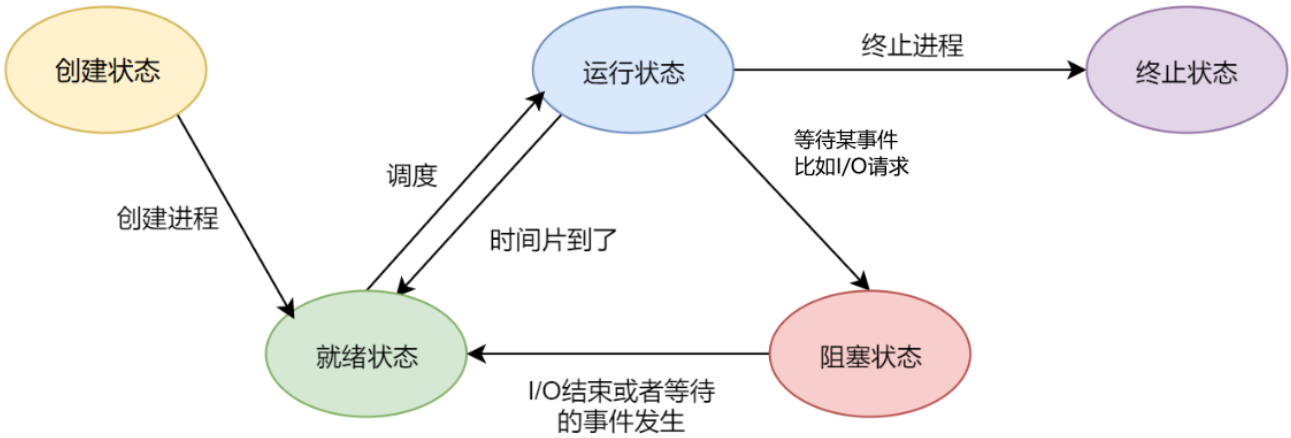
iii. 混合型内核
iv. PE 可移植执行文件：扩展名.exe, .dll, .sys；所以windows和linux的可执行文件不能互相运行

二、进程管理

进程 线程 处理机调度

first of all: code只是存储在硬盘里的静态文件，编译后会生成二进制可执行文件，当我们运行该可执行文件，它会被装在到内存，接着cpu执行程序内的每一条指令，此时，这个运行中的程序，我们称之为process。

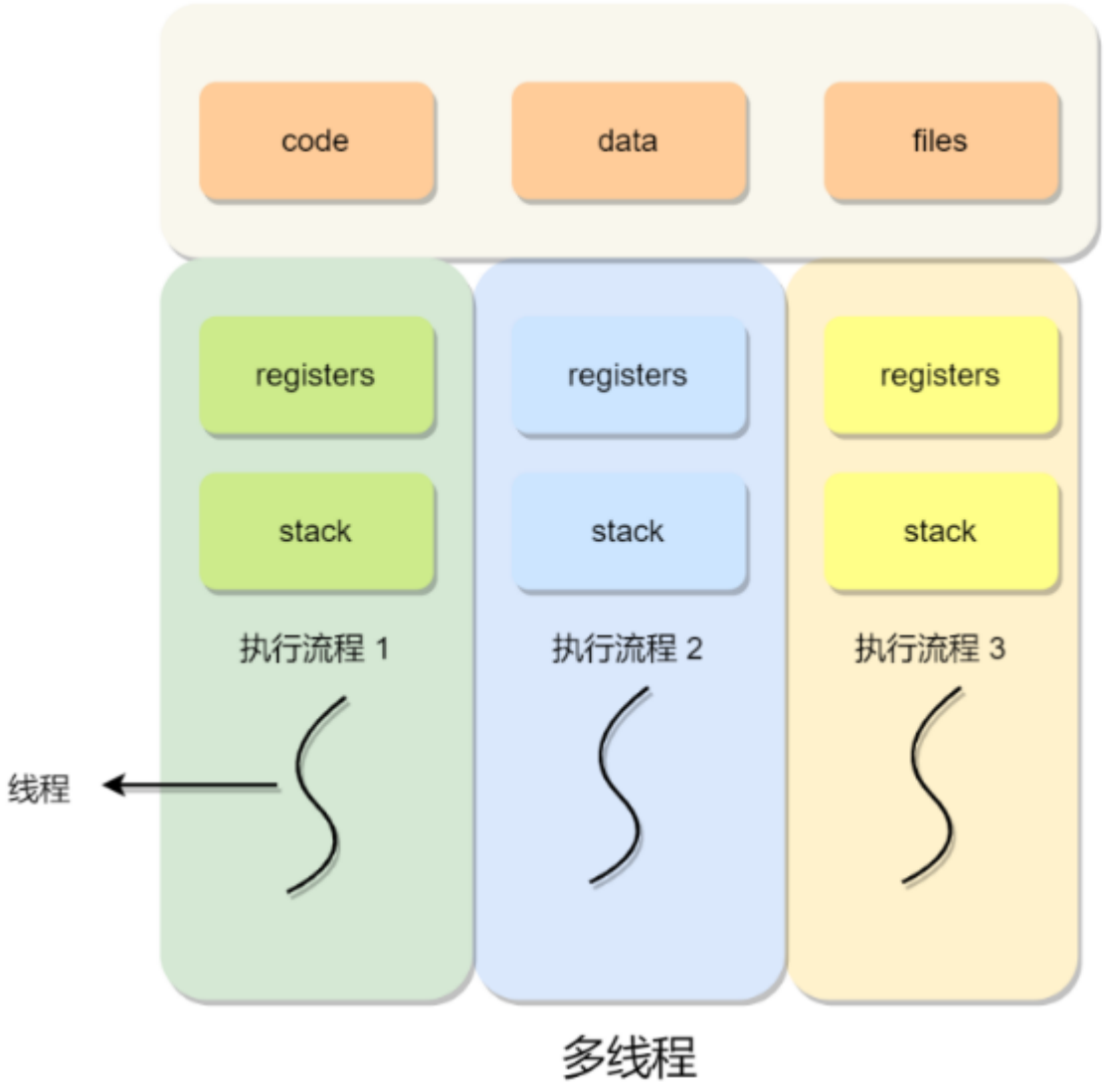
1. 进程(实体)组成：程序段program segment；数据段data segment；进程控制块process control block PCB
2. PCB：用来描述进程的数据结构，是进程存在的唯一标识。
    i. PCB组成：
        a. 进程描述信息(pid, uid)
        b. 进程控制信息(进程状态, 优先级)
        c. 资源分配清单内 (内存地址空间，虚拟地址空间，文件列表，io设备信息)
        d. CPU相关信息 (cpu内寄存器的值)
    ii. PCB是如何组织的?
        a. 将相同状态的进程PCB通过**链表**连接在一起形成“队列”，比如就绪队列、阻塞队列； （这里的“队列”不是queue，不遵循FIFO，应该就是单链表）
3. . **进程的五种状态**：创建态new；就绪态ready；运行态running；阻塞态blocked；终止态exit
    i.



    ii. 运行态->结束态：当进程已经运行完成或出错时，会被操作系统作结束状态处理；
    iii. 其实还有一个挂起状态suspend，挂起状态不释放cpu，是一种主动行为; 阻塞状态释放cpu，被动行为; 分为两种：
        a. 阻塞挂起状态：进程在外存并等待某个事件的出现；
        b. 就绪挂起状态：进程在外存，但只要进入内存，即刻立刻运行；

iv. 进程各状态的控制：见p152 xiaolincoding

4. **CPU上下文切换**是：把前一个任务的CPU上下文（CPU寄存器和PC）存到系统内核，然后加载新任务的上下文到这些寄存器和PC上，然后跳转到PC所指的新位置，执行新任务。根据任务的不同，可以分为进程上下文切换、线程上下文切换和中断上下文切换。
5. **进程上下文切换**：各进程之间是共享CPU的，在不同的时候进程之间需要切换，让不同的进程可以在CPU执行，那么一个进程切换到另一个进程运行，称为进程的上下文切换。(进程切换发生在内核态)
   i. 保存处理机上下文，包括PC和其他CPU寄存器
   ii. 更新PCB信息（什么意思？？）
   iii. 把进程的PCB移动到相应的队列，如就绪队列。
   iv. 选择另一个进程执行，并更新其PCB。
   v. 更新内存管理的数据结构
   vi. 回复处理机上下文
6. 线程Thread：同一个进程内多个线程之间可以共享代码段、数据段、打开的文件等资源，但每个线程各自都有一套独立 的寄存器和栈，这样可以确保线程的控制流是相对独立的。
   i.



多线程

ii. 线程控制块TCB：包括PC，栈指针，寄存器；用户级TCB由线程库函数维护，内核级TCB由os内核维护。

7. **进程和线程的区别?**

线程被称作轻量级进程，在进程中包含线程。进程有独立的内存空间，不同进程间不能直接共享其他进程资源，同一个进程内的线程共享进程内存空间；相比进程，线程切换对系统开销更小一些；进程是资源分配的最小单位，线程是程序执行的最小单位.

> 操作系统的任务调度对象是线程，而进程只是给线程提供了虚拟内存、全局变量等资源。

i. 调度。线程是CPU调度的基本单位。
ii. 资源。进程是资源分配的基本单位，线程不独立拥有资源，但可以访问隶属进程的资源。
iii. 并发性。引入线程就是为了获得更好的并发性。
iv. 开销。进程的创建、撤销和切换都要有资源的分配和回收，开销远大于线程的切换。
v. 地址空间和其它资源。进程的地址空间都是独立的，而线程可以共享隶属进程的地址空间和资源。
vi. 通信方面。进程之间通信需要用进程通信手段实现，而线程间通信可以直接读写数据段。

biaodianfu
https://www.biaodianfu.com

+ 关注

4626 人赞同了该回答

看了一遍排在前面的答案，类似”**进程是资源分配的最小单位，线程是CPU调度的最小单位**“这样的回答感觉太抽象，都不太容易让人理解。

做个简单的比喻：进程=火车，线程=车厢

- 线程在进程下行进（单纯的车厢无法运行）
- 一个进程可以包含多个线程（一辆火车可以有多个车厢）
- 不同进程间数据很难共享（一辆火车上的乘客很难换到另外一辆火车，比如站点换乘）
- 同一进程下不同线程间数据很易共享（A车厢换到B车厢很容易）
- 进程要比线程消耗更多的计算机资源（采用多列火车相比多个车厢更耗资源）
- 进程间不会相互影响，一个线程挂掉将导致整个进程挂掉（一列火车不会影响到另外一列火车，但是如果一列火车上中间的一节车厢着火了，将影响到所有车厢）
- 进程可以拓展到多机，进程最多适合多核（不同火车可以开在多个轨道上，同一火车的车厢不能在行进的不同的轨道上）
- 进程使用的内存地址可以上锁，即一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。（比如火车上的洗手间）－”互斥锁”
- 进程使用的内存地址可以限定使用量（比如火车上的餐厅，最多只允许多少人进入，如果满了需要在门口等，等有人出来了才能进去）－“信号量”

编辑于 2018-06-20 11:16

▲ 赞同 4626 ▼ 97 条评论 分享 收藏 喜欢 ···

8. 线程的上下文切换：
    i. 当进程只有一个线程时，可以认为进程就等于线程；
    ii. 当进程拥有多个线程时，这些线程会共享相同的虚拟内存和全局变量等资源，这些资源在上下文切换时是不需要修改的；
    iii. 另外线程也有自己的私有数据，比如栈和寄存器。
    iv. 所以现成的上下文切换：
        a. 如果两个线程不属于同一进程，那切换过程等同于进程上下文切换。
        b. 如果属于同一进程，由于虚拟内存是共享的，所以只需要切换线程的私有数据、寄存器等不共享的数据。故而线程的上下文切换开销比较小。
9. 线程间的同步与通信类型有哪些?
    i. 互斥锁mutex；
    ii. 条件变量；
    iii. 信号量机制
10. 线程的实现方式有哪几种?
    i. 内核级线程kernel thread：在内核中实现的线程（由os内核管理）
        a. 在一个进程中，如果某内核线程发起系统调用而阻塞，不影响其他内核线程的运行。
        b. 线程的创建、终止和切换都是通过系统调用的方式来进行，因此对于系统来说，系统开销比较大。
    ii. 用户级线程user thread：在用户空间实现的线程
        a. 由用户态的线程库实现和管理，整个线程管理和调度，os内核是不直接参与的。无需用户态和内核态的切换，速度快。
        b. 一个线程发起了系统调用而阻塞，那进程中所包含的用户线程都不能执行了。
    iii. 轻量级进程light weight process：内核支持的用户线程，
11. 管程是什么?
    由一组数据及对这组数据操作的定义组成的模块。同一时间只能有一个进程使用管程，即管程是互斥使用的，进程释放管程后需唤醒申请管程资源的等待队列上的进程。进程只有通过进入管程并使用管程内部的操作才能访问其中数据
12. 处理机的三级调度：
    - 高级调度（作业调度）：把作业从外存中取出，给它分配内存和其它资源，让它称为一个进程，是其能具备竞争处理机的条件。它是主存和辅存之间的调度。
    - 中级调度（内存调度）：作用是提高内存利用率。将那些不能运行的进程挂起到外存，如果他们已经具备运行条件，有稍微有些空闲，由中级调度决定外存上的进程重新调入内存，并修改为就绪态。
    - 进程调度（进程调度）：是操作系统最基本的调度。按照某种方法从就绪队列中选取一个进程，将处理机分配给它。频率最高。
13. 不能进行进程调度和切换：中断；原子操作；进程在内核临界区。
14. **处理机/CPU调度**dispatch/scheduling方法：

- 先来先服务（First Come First Served, FCFS）(非抢占式)：对长作业有利，适合cpu繁忙型作业的系统，不适合io繁忙型作业的系统。
- 短作业优先（Shortest Job First, SJF）:对长作业不利
- 高响应比优先（Highest Response Ratio Next, HRRN）:权衡了长作业和短作业
  - 响应比优先权 $= \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$
- 时间片轮转（Round Robin, RR）：最古老、最简单、最公平、适用范围最广的算法。(假设所有进程同等重要)
  - 如果时间片设置过短，那么就会造成大量的上下文切换，增大了系统开销。
  - 如果过长，就退化成 FCFS 算法了。
- 最高优先级调度（Highest Priority First, HPF）：从就绪队列选择优先级最高的进程。
- 多级反馈队列（Multilevel Feedback Queue）：是RR和HPF的综合和发展
  - 「多级」表示有多个队列，每个队列优先级从高到低，同时**优先级越高时间片越短**。
  - 「反馈」表示如果有新的进程加入优先级高的队列时，立刻停止当前正在运行的进程，转而去运行优先级高的队列；

15. **页面置换算法?**
    最佳置换算法OPT 先进先出置换算法FIFO 最近最久未使用算法LRU
    时钟算法LOCK 改进型时钟算法
16. **磁盘调度算法?**
    先来先服务FCFS 最短寻道时间优先SSTF 扫描算法SCAN 循环扫描算法C-SCAN
17. **面向过程与面向对象的区别：（问题保留）**
    面向过程让计算机有步骤地顺序做一件事，是过程化思维，使用面线过程语言开发大型项目，软件复用和维护存在很大问题，模块之间的耦合严重。面向对象相对面向过程更适合解决较大的问题，可以拆解问题复杂度，对现实事物进行抽象并映射为开发对象，更接近人的思维。

    举例：大象装进冰箱。
    面向过程：打开冰箱 --> 存储大象 --> 关上冰箱
    对于面向过程思想，强调的是过程（动作）.语言：C。
    面向对象：冰箱打开-->冰箱存储大象 -->冰箱关上
    对于面向对象思想，强调的是对象（实体）。语言：C++、Java、C#
    特点：1. 面向对象就是一种常见的思想，符合人们的思考习惯。
    i. 面向对象的出现，将复杂的问题简单化。
    ii. 面向对象的出现，让曾经在过程中的执行者，变成了对象的指挥者。

进程间通信

每个**进程的用户地址空间都是独立**的，一般而言是不能互相访问的，但**内核空间是每个进程都共享**的，所以进程之间要通信**必须通过内核**。(见p197 小林总结 还不错)

1. **管道**pipeline：linux中的|就是管道，将前一个命令的输出作为后一个命令的输入，单向（半双工），如果需要双向通信需要创建两个管道；
    i. **管道就是内核里面的一串缓存。**从管道的一段写入的数据，实际上是缓存在内核中的，另一端读取，也就是从内核中读取这段数据。
    ii. |是匿名管道，用完了就销毁了；匿名管道的通信范围是存在父子关系的进程。（因为匿名管道没有实体文件，只能通过fork来复制父进程的文件描述符fd）详见p187
    iii. 还有一种命名管道(FIFO)， `makefifo mypipe -> echo "helle" > mypipe -> cat mypipe`
        a. 命名管道的读取和写入操作都是阻塞的。如果没有数据可读，读取进程将被阻塞，直到有数据可用
        b. 命名管道可以在不相关的进程建通信，因为他有fd可以随意使用；
2. **消息队列**：用于解决管道不适合进程间频繁交换数据（通信效率低）的问题，克服了管道通信的数据是无格式的字节流的问题；消息队列是保存在内核中的消息链表；
    i. 消息队列不适合比较大数据的传输；
    ii. 消息队列通信过程中，**存在用户态与内核态之间的数据拷贝开销**，因为进程写入数据到内核中的消息队列时，会发生从用户态拷贝数据到内核态的过程，反之亦然。
3. **共享内存**：拿出一段虚拟地址空间，将其映射到**相同**的物理地址，这样就无需拷贝了。
    i. 解决了消息队列切换状态的问题。
    ii. 问题：多个进程**同时**修改同一共享内存可能引起**冲突**，所以保护机制：**信号量**上场。
4. **信号量**：实现**进程间**的互斥与同步，本质上是一个**整形计数器。**（保证多个进程或线程能够按照一定的顺序访问共享资源。）见P190 xiaolin
5. **信号**：信号是一种在软件层面上用于进程间通信或者内核向进程通知事件发生的机制。（与信号量完全不同）
6. **Socket**：前面提到的管道、消息队列、共享内存、信号量和信号都是在同一台主机上进行进程间通信，那要想跨网络与不同主机上的进程之间通信，就需要 Socket通信了。
7. 进程通信的方式(old version)
    i. 低级通信方式：PV操作（信号量机制）。
        a. P：wait(S)原语，申请S资源
        b. V：signal(S)原语，释放S资源
    ii. 高级通信方式：以较高效率传输大量数据的通信方式
        a. 共享存储（使用共享空间）
        b. 消息传递（进程间以格式化的消息进行数据交换，底层通过发送消息和接收消息两个原语实现）
        c. 管道通信（两个进程中间存在一个特殊的管道文件，特点：半双工通信）

多线程同步

我们知道进程是资源分配的基本单位，线程是调度的基本单位。线程会共享进程的资源，比如代码段、堆空间、数据段和打开的文件等。此外，每个线程也会有自己独立的栈和寄存器。

那么如何让多个线程在共享资源的时候不造成混乱呢?

临界区critical section：访问共享资源的代码段，一定不能给多线程同时执行。

互斥mutual exclusion / mutex：间接制约关系，当一个线程(进程)进入临界区，其他的必须被阻止进入临界区（等待）。

> 互斥并非仅针对多线程，多进程竞争共享资源的时候也可以使用互斥避免混乱。

我们知道在多线程里，每个线程并不一定是顺序执行的，它们基本是以各自独立的、不可预知的速度向前推进，但有时候我们又希望多个线程能密切合作，以实现一个共同的任务。

同步Synchronism：直接制约关系，两个线程(进程)在一些关键点上可能需要互相等待与互通小计（在时间上有先后关系）。

> 同步好比是操作A应该在操作B之前执行；互斥好比是操作A和操作B不可同时执行。
> 同步的四个准则：空闲让进，忙则等待，让权等待，有限等待

进程/线程互斥：通过锁或者信号量实现
进程/线程同步：通过信号量实现

锁

任何想进入临界区的线程，必须先执行**加锁**操作。若加锁操作顺利通过，则线程可进入临界区；在完成对临界资源的访问后再执行**解锁**操作，以释放该临界资源；（有点回溯/栈的思想）

锁可以分为**忙等待锁**和**无忙等待锁**

- 忙等待锁（自旋锁spin lock）：在使用忙等待锁时，线程会反复检查锁是否可用，如果锁被其他线程持有，当前线程就会一直循环检查直到锁可用为止。这种方式会占用大量的处理器时间，因为线程会持续执行循环检查的操作。
- 无忙等待锁：将线程在无法获取到锁时放入等待队列，然后将cpu让给其他线程，直到锁可用时再唤醒线程。

信号量

1.信号量semaphore是os提供的协调共享资源访问的方法；信号量表示共享资源的数量，本质上是一个整形计数器。

2.什么是PV操作?
P操作：将信号量sem减1，如果此时sem<0，表明资源已被占用，进程/线程需要阻塞等待；如果

sem>=0，表明还有可用资源，进程/现成可继续执行。
V操作：将信号量sem加1，如果此时sem<=0, 表明当前有处于阻塞中的进程/线程，会将该进程唤醒；如果sem>0, 表明没有阻塞中的进程。

```
// 信号量数据结构
type struct sem_t{
    int sem;      // 资源个数
    queue_t *q;   // 等待队列
} sem_t;

// 初始化信号量
void init(sem_t *s, int sem)
{
    s->sem = sem;
    queue_init(s->q);
}

// P 操作
void P(sem_t *s)
{
    s->sem--;
    if(s->sem < 0)
    {
        1. 保留调用线程 CPU 现场;
        2. 将该线程的 TCB 插入到 s 的等待队列;
        3. 设置该线程为等待状态;
        4. 执行调度程序;
    }
}

// V 操作
void V(sem_t *s)
{
    s->sem++;
    if(s->sem <= 0)
    {
        1. 移出 s 等待队列首元素;
        2. 将该线程的 TCB 插入就绪队列;
        3. 设置该线程为「就绪」状态;
    }
}
```

3.如何利用信号量实现临界区的互斥访问?
只需要对每一类共享资源设置一个初值为 1 的信号量 s ，表示该临界资源未被占用。然后后续在申请该

资源之前进行 `P(s)` 操作，之后进行 `V(s)` 操作进行释放。
对于两个并发线程，互斥信号量的值仅取1、0和-1:
1: 表示没有线程进入临界区
0: 表示有一个线程进入临界区
-1: 表示一个线程进入临界区，另一个线程等待进入

4.如何利用信号量实现事件同步?
设置初值为 `0` 的信号量 `s` ，然后申请事件的时候P(s)，事件准备好的时候会V(s);

5.生产者-消费者问题
一个初始为空大小为n的缓冲区，
只有缓冲区没满时生产者才可以把消息放入缓冲区，否则等待； -> 同步关系 `fullBuffers=0`
只有缓冲区非空时消费者才可以从中取出消息，否则等待。 -> 同步关系 `emptyBuffers=n`
缓冲区是临界资源，任何时候只允许一个生产者或消费者进入。 -> 互斥关系 -> `mutex=1`

code比较简单;

6.哲学家进餐问题
5个哲学家和5根筷子，如果让一名哲学家拿到左右两根筷子而不造成死锁或者饥饿现象。

> 「哲学家进餐问题」对于互斥访问有限的竞争问题（如 I/O 设备）一类的建模过程十分有用
> 思维精髓是：不能贪心，要考虑后续后果。如果每人都拿一根那就死锁了。

多种方法:

- 最多允许4位同时进餐;
- 仅当一名哲学家左右两边的筷子都可拿时才拿;
- 对哲学家顺序编号，奇数号哲学家先拿左边筷子，再拿右边筷子。偶数哲学家刚好相反;

7.读者-写者问题
读者只会读取数据，不会修改数据，而写者即可以读也可以修改数据。
描述:
「读-读」允许：同一时刻，允许多个读者同时读
「读-写」互斥：没有写者时读者才能读，没有读者时写者才能写
「写-写」互斥：没有其他写者时，写者才能写
解法略 参考王道os

死锁

2. **死锁deadlock**：多个进程因竞争资源而造成的一种僵局，没有外力作用，都无法继续执行。
   原因：对互斥资源分配不当；进程推进顺序不当。

3. 死锁的四个必要条件：互斥条件(Mutual exclusion) 请求与保持条件(Hold and wait) 非剥夺条件(No pre-emption) 循环等待条件(Circular wait)
4. 处理产生死锁的办法有哪些?
    i. 预防死锁（破坏产生死锁的必要条件）
    ii. 避免死锁（银行家算法）；
    iii. 检测死锁（资源分配图）；
    iv. 解除死锁（剥夺资源 or 撤销进程）.
5. 死锁deadlock与饥饿hunger的区别?
    i. 都是资源分配问题
    ii. 死锁是等待永远不会释放的资源，而饥饿申请的资源会被释放，只是永远不会分配给自己
    iii. 一旦产生死锁，则死锁进程必然是多个，而饥饿进程可以只有一个
    iv. 饥饿的进程可能处于就绪状态，而死锁进程一定是阻塞进程

三、内存管理

(一）基础：将多个进程保存到内存中

1. 内存管理memory management功能：内存空间的分配与回收、地址转换（eg VA->PA）、扩充内存空间（虚拟存储技术逻辑上扩充）、存储保护（利用界地址寄存器保护os不受用户进程影响）。
2. **C语言运行过程/源程序->目标程序**:
   源程序(hello.c)->可执行文件(hello)
    i. 预处理阶段（预处理器cpp）：对#开头的命令进行处理，eg #include <stdio.h>将.h文件插入程序。输出文件hello.i;
    ii. 编译阶段（编译器ccl）：**翻译**生成汇编语言源程序hello.s；
    iii. 汇编阶段（汇编器as）：将hello.s**翻译**成机器指令，打包成一个可重定位目标文件hello.o；**目标文件都从0开始编址，都是独立的逻辑地址/相对地址**
    iv. 链接阶段（链接器ld）：将多个目标文件和所需库函数**打包**为一个可执行目标文件hello.exe；**可执行模块具备完整的逻辑地址**
    v. 装入：由装入程序把exe文件装入内存运行；**将逻辑地址转为物理地址（地址重定位），装入到实际的物理地址**
3. 内存保护的方式：
    i. 上下限寄存器法：存放用户作业在主存中的上下限地址。
    ii. 重定位寄存器（基址寄存器）和界地址寄存器（限长寄存器）组合法。
        a. 界地址寄存器含有逻辑地址的最大地址，是用来“比较”的；重定位寄存器含有物理地址最小值，是用来“加”的。
4. 逻辑地址virtual 与 物理地址physical：

i. **逻辑地址（虚地址/相对地址/人的视角）**：编译后每个目标模块都从0开始编制，叫做相对地址或逻辑地址。
ii. **物理地址（实地址/机器视角）**：是地址转换后的最终地址。装入程序将代码装入内存，必须将逻辑地址变成物理地址，称为**地址重定位**。
    a. os如何管理VA和PA之间的关系？内存分段和内存分页。

1. 有哪些**连续**分配内存管理方式?
    i. 单一连续分配：将内存分成系统区（低地址）和用户区。优点：简单、无外部碎片；缺点：适用于**单道程序**（整个用户只存放一道程序），有**内部碎片**，内存利用率非常低。
        a. 内部碎片：**已分配**给某进程的内存空间大于该进程实际需要的空间。
        b. 外部碎片：内存中已分配的分区外的存储空间越来越多的碎片难以利用。一般都是进程退出后留下的小块空闲块。
    ii. 固定分区分配：将内存空间划分成大小固定的分区，每个分区只装入一个作业。分区大小可以相等也可以不等。优点：可以多道程序; 缺点:分区固定不变，程序不能太大，否则放不进去，有**内部碎片**；内存利用率低。
    iii. 动态分区分配：不预先划分分区，而是等进程装入内存时再动态建立分区。优点是：可以使分区大小刚好合适。缺点是：会产生较小的**外部碎片**分配不出去。
        a. 动态分区分配的四种算法：
            a. 首次适应 first fit：空闲分区按地址递增链接。
            b. 最佳适应 best fit：空闲分区按容量递增链接。
            c. 最坏适应 worst fit：按容量递减。会留下很多外部碎片。
            d. 邻近适应 next fit：从首次适应基础上，在上次查找结束位置继续查找。
2. **非连续**分配管理：将一个程序分散地装入不相邻的内存分区（当然需要额外空间存储各区域的索引作为代价）
    i. 按照分区大小是否固定分为：分页存储管理方式和分段存储管理。

> 分页存储管理方式根据作业时是否把作业的所有页面都装如内存才可运行分为：基本分页 + 请求分页(虚拟内存)。

3. 页式内存管理中的**页表**是什么，多级页表呢?
    i. 页表：进程页号 -> 物理内存块号（mapping）
        a. 页表项 = 进程页号 + **内存物理块号**
            a. 所以页表项中的内存物理块号 + 逻辑地址中的页内偏移量共同组成物理地址；
        b. 一个进程对应一张页表。（一个进程对应一个自己的虚拟地址空间）
    ii. 页表一定要覆盖全部虚拟地址空间，不分级的页表就需要有 100 多万个页表项来映射，而二级分 页则只需要 1024 个页表项（此时一级页表覆盖到了全部虚拟地址空间，二级页表在需要时创建）
4. 描述**页式存储管理**。

i. 把主存空间物理上划分为大小相等且固定的块（页框/页帧）；与固定分区相似，不会产生外部碎片，但这里的块相比于分区要小得多。只有在为进程最后一个不完整的页申请主存块的时候才会产生大概半个块的内部碎片，洒洒水啦。
ii. 每个进程逻辑上以块（页/页面）为单位划分，在执行时以块为单位申请主存中的块空间。
iii. 逻辑地址Logical address = 进程页号 + **页内偏移量**；(地址结构决定了虚拟内存的寻址空间大小)
iv. 进程的页映射到物理内存的块

逻辑地址 -> 物理地址(核心在于**先找到页表项PA**，然后concat(表项的物理块号, LA的页内偏移量)即可):

i. 根据页面大小L，计算出逻辑地址LA的页号(LA/L)和页内偏移量(LA%L)。
ii. 由逻辑地址中的页号与PTR中的页表长度对比，若大，则越界，发生中断。
    a. 页表寄存器PTR中存放页表内存始址 + 页表长度（页表项个数）
iii. 若页号合法：页表项地址 = 页表始址 + 页号 * 页表项长度（页表项(页地址)占用存储空间）。
iv. 物理地址PA = 页表项中块号 + 逻辑地址中的偏移量。

5. 介绍一下快表TLB:
    i. 由于页式存储管理在地址变换时，需要两次内存访问：访问页表（确定所需要的数据或者指令的PA）+ 取数据/指令；故设置快表TLB，充当页表的cache，一个专门的高速缓冲存储器。与之相对的主存中的页表称作慢表。
    ii. 引入TLB的地址变换:
        a. 首先将页号与快表的所有页号进行比较，如果命中，直接取出物理块号。即可与LA的页内偏移量形成PA；（仅一次访存）
        b. 若未命中，还是去访问主存中的页表，**同时将其存入快表。**
            a. 一般快表命中率可达90%；又是局部性原理咯。
6. 介绍多级页表:
    i. 当逻辑地址空间很大时，页表长度会大大增加。需要一块比较大的连续物理空间存储，这不好。
    ii. 二级页表时 逻辑地址 = 一级页号 + 二级页号 + 页内偏移量
    iii. 缺点就是增加了一次访存时间。
7. 描述**段式存储管理。**
    i. LA = 段号 + OFF
    ii. 段表项 = 段号(**不占空间(why)**) + 段长 + 本段在主存中始址
    这种管理方式考虑到了程序员的感受，以满足方便编程、信息保护和共享、动态增长及动态链接等要求。它按用户进程中的自然段划分逻辑空间，每个段从0开始编址，并分配连续的地址空间。段内连续，段之间可以不连续。

逻辑地址->物理地址（与页式类似）

- 段表项地址 = 段表起始地址 + 段号 × 段表项长度
- 物理地址 PA = 页表项中本段起始地址 + LA中段内偏移量

（二）虚拟内存管理

（所以虚拟地址在虚拟内存之前就出现了？那么va的本意是？）
为了在多进程环境下，使得进程之间的内存地址不受影响，相互隔离，于是操作系统就为每个进程独立分配一套虚拟地址空间，每个程序只关心自己的虚拟地址就可以，实际上大家的虚拟地址都是一样的，但分布到物理地址内存是不一样的。作为程序，也不用关心物理地址的事情。
既然有了虚拟地址空间，那必然要把虚拟地址「映射」到物理地址，这个事情通常由操作系统来维护。
那么对于虚拟地址与物理地址的映射关系，可以有分段和分页的方式，同时两者结合都是可以的。

1. 为什么要引入虚拟内存?
    i. 主要目的是为了扩展计算机的内存空间，使得可以运行更大的程序或处理更多的数据。
    ii. 为每个进程提供了一个独立的虚拟内存空间，使得进程以为自己独占全部内存资源。避免内存冲突。
2. 我们在上面介绍了各种内存管理策略，目的在于同时将多个进程保存在内存中，实现多道程序设计。但他们存在一定的缺陷（总而言之，许多在程序运行中不用或暂时不用的程序或数据占据了大量内存，造成浪费）：
    i. 一次性：作业必须一次性全部装入内存后才可开始运行。
        a. 作业很大内存不够就跑不了；
        b. 作业很多的话，不够同时运行多个作业了。
    ii. 驻留性：作业被装入内存后会一直驻留在内存中，直至运行结束。
3. **虚拟存储器**：基于**局部性原理**，将程序的一部分装入内存，而将其与部分留在外存，就可以启动运行程序。在执行过程中，程序要访问的信息不在内存，由操作系统将所需的部分调入内存执行。操作系统将暂时不用的内容换出到外存上，空闲空间存放从外存换入的信息。这样操作系统就**好像**为用户提供了一个比实际内存大得多的存储器。
    i. 之所以叫做虚拟存储器：是因为os提供了部分装入、请求调用和置换功能后，给用户的感觉好像是存在一个比实际物理内存大得多的存储器。
    ii. 虚拟存储器的大小由计算机的地址结构决定，而非内存和外存的求和。虚存实际容量≤内存+外存；虚存最大容量≤计算机的地址位数能容纳的最大容量。
    iii. 虚拟存储器与前面的内存策略不同，有三个主要特征
        a. 多次性：无需在作业运行时一次性全部装入内存
        b. 对换性：无需再作业运行时一直常驻内存，允许作业进行换进和换出
        c. 虚拟性：从**逻辑上**扩充了内存的容量，使得用户看到的内存容量远大于实际内存
4. 虚拟存储技术所需硬件支持：页表(段表)机制、缺页中断机构、地址变换机构、一定容量的内存和外存。
5. 虚拟内存的三种实现方式与前面对应，称作：请求分页存储管理、请求分段存储管理和请求段页式存储管理
6. 介绍一下请求分页存储管理
    i. 请求分页存储管理 = 基本分页存储管理 + 请求调页功能 + 页面置换功能

    ii. 页表项 = 进程页号 + 内存物理块号 + 状态位(是否已调入内存) + 访问字段(访问次数或时间，供置换算法) + 修改位(页面调入内存后是否修改) + 外存地址(供调页使用)
    iii. 地址变换机构：先查快表，若命中直接形成PA，否则 -> 查慢表，若命中形成PA，否则 -> 发生中断，请求调页。
7. 页面置换算法（决定换入哪页，换出哪页，输入io操作）：
    i. 最佳置换算法 optimal replacement algorithm：被淘汰页面是之后最长时间不访问的页面。**理想**情况，无法实现。
    ii. 先进先出置换算法 FIFO：会产生belady异常
        a. Belady异常：分配物理块数增大，缺页次数不减反增。
    iii. 最近最久未使用 least recently used（LRU）：最近没用，未来用的概率不大。
        a. 是堆栈类算法，因为他要看看之前的东西，需要寄存器和栈的硬件支持。性能接近于opt算法，但是开销较大。
    iv. 时钟置换算法clock：性能接近LRU，开销小一些。
8. 驻留集resident set：分配给进程的物理页框数。
9. 工作集working set：某时间间隔内，进程要访问的页面集合。
10. 抖动/颠簸page jitter：刚刚换出的页面又要换入主存。

总结

为了避免不同的进程操控到同一块物理内存地址，遂引入虚拟地址进行抽象和隔离，每个进程都有一块独立的虚拟内存空间，不同进程的虚拟空间会映射到不同的物理内存上（地址变换机构借助页表实现这件事情，即使相同的逻辑地址也可以映射到不同的物理地址），由于总的虚拟内存空间肯定比物理内存大的多（而且我们需要实现多道批程序设计，不能让每一个进程完全装在到内存中），我们会为进程按需分配内存；还有当物理内存不足的时候，我们会进行页面置换。

对于虚拟内存中，为了减少页表空间占用，引入多级页表；为了加速地址变换，引入快表。

四、文件管理

1. 文件控制块FCB：存放控制文件的信息。FCB的集合就是文件目录。
2. 文件的逻辑结构：
    i. 无结构文件（流式文件）： eg 源程序文件 目标代码文件
    ii. 有结构文件（记录式文件）： 顺序文件、索引文件、索引顺序文件
3. 硬链接和软链接方式的比较
    i. 硬链接：（基于索引节点的共享方式）这种共享方式中，文件的物理地址和其它的文件属性不再放入目录项中，而是放在索引结点中。在文件目录中只设置文件名和指向索引结点的指针。索引结点中有计数器表示指向这个索引结点的用户目录项的个数。当删除时将索引结点的计数器减一，然后删除对应的用户目录项。当计数器为零时，就将文件彻底删除。
        a. 优点：实现了异名共享。
        b. 缺点：文件拥有者不可删除正在共享的文件。

ii. 软连接：（利用符号链实现文件共享）只有文件的拥有者才拥有只想起索引节点的指针，而共享该文件的其他用户只有该文件的路径名。
    a. 优点：拥有者可删除
    b. 缺点：其他用户开销大

4. 文件的分配方式（对非空闲磁盘块的管理）
    i. 连续分配：FCB中记录起始块号和数量（长度），支持随机访问。
    ii. 链接分配：离散分配，消除了外部碎片，提高了利用率。隐式连接分配每个块都有next指针，显式链接分配把 指向各物理块的指针 显式地存在文件分配表FAT中。
    iii. 索引分配：索引表，FCB中包括索引块的地址。支持随机访问。(多层索引与混合索引)
5. 对空闲磁盘块的管理
   空闲表法、空闲链表法、位示图法和成组链接法
6. 一次磁盘读写时间 = 寻道时间（磁盘调度算法） + 逆转延迟时间（交替编号、错位命名） + 传输时间
7. 磁盘调度算法：
    i. FCFS：公平
    ii. 最短寻找时间优先算法（Shortest Seek Time First,SSTF）：眼前最优未必总体最优，“饥饿”现象
    iii. SCAN算法/电梯算法：磁头移动到最外侧磁道才往内移动。利于端头一侧。
    iv. CSCAN：规定磁头单向移动，使个位置磁道的响应频率平均。

五、IO系统

每个IO设备都有相应的设备控制器（相当于一个小cpu），设备控制器中有一些寄存器，分别是状态寄存器、命令寄存器和数据寄存器，CPU通过他们和设备打交道。
通过写入这些寄存器，os可以命令设备发送接收数据、开启或关闭等操作。
通过读取这些寄存器，os可以了解设备的状态。

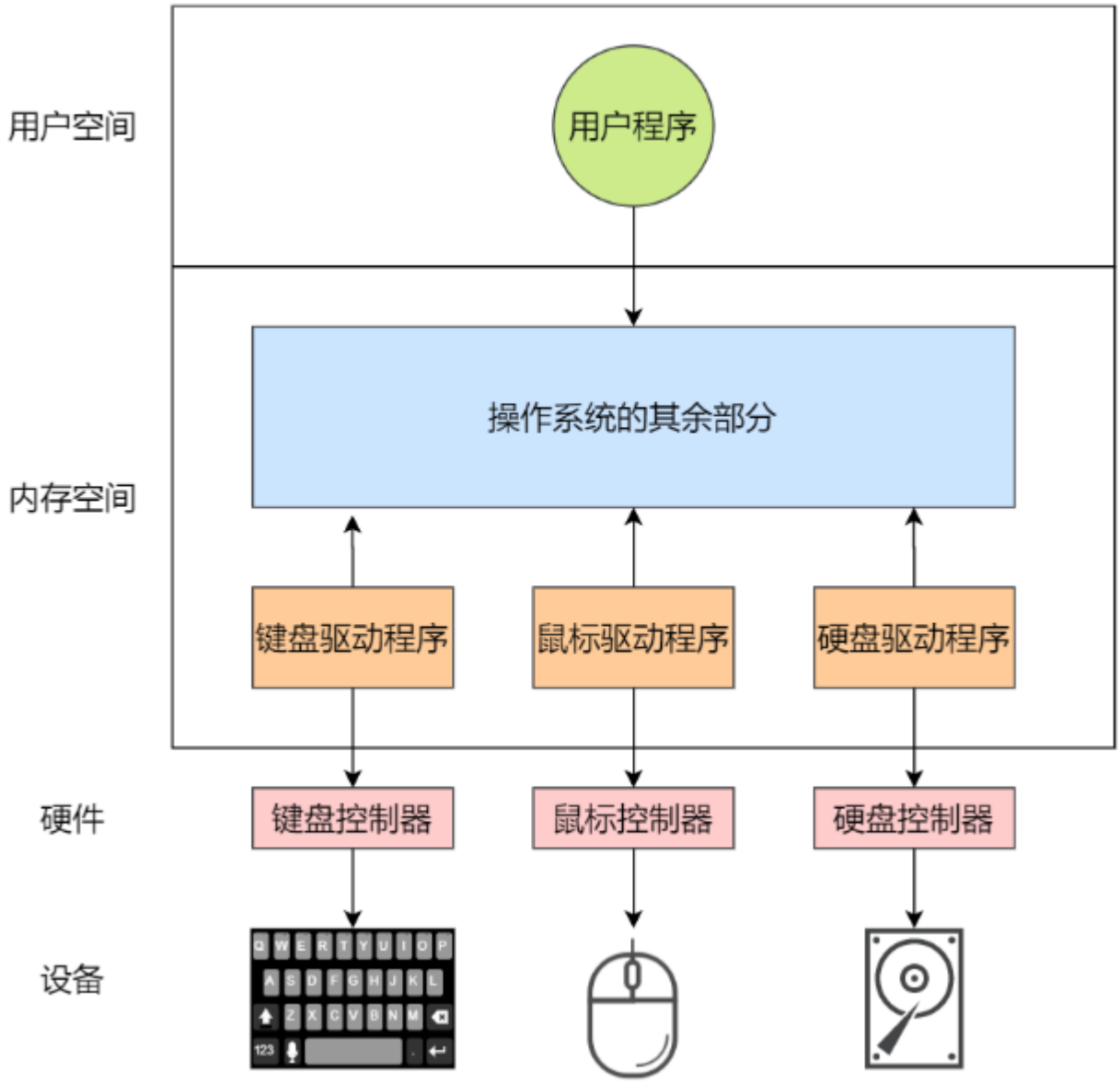
此外，对于块设备（比如磁盘）对应的设备控制器中还有数据缓冲区。

我们知道每种设备都有一个设备控制器（小CPU），它可以自己处理一些事情，但问题是，当CPU给设备发送了一个指令，让设备控制器去读设备的数据，它读完的时候，要怎么通知CPU呢?

1. IO控制方式：
    i. 直接控制方式/程序查询方式（轮询等待）：CPU每读取一个字，就要对外设状态进行轮询检查。CPU和IO只能串行工作，cpu利用率低。 由于CPU的高速性和IO设备的低速性，致使CPU大部分时间都处于等到IO设备的循环，造成CPU资源的极大浪费。
        a. 让cpu一直查询控制器中寄存器的状态；很蠢

ii. **中断驱动**方式。CPU在向IO设备发出读命令后，可以转去做其它的事情，等到IO设备数据就绪，由IO设备主动发出中断请求打断CPU。这样是CPU和设备都可以尽量忙起来。
   a. 频繁中断(如磁盘这种频繁读写设备)对cpu并不友好
   b. 依然需要cpu亲自参与数据搬运过程
iii. **DMA**(Direct Memory Access)方式。DMA方式基本思想是，在主存和IO设备之间直接开辟数据通路，彻底解放CPU，使得设备可以不依靠cpu的情况下自行把输入放到主存。其特点是基本单位是数据块，所传送的数据是从设备直接送入内存的。仅仅在一个或多个数据块传输开始或结束时才需要CPU的干预，这个数据块的传输是在DMA控制器的控制下完成的。

2. 设备驱动程序：虽然设备控制器**屏蔽**了设备的众多细节，但每种设备的控制器的寄存器、缓冲区等使用模式都是不同的， 所以为了**屏蔽**「设备控制器」的差异，引入了设备驱动程序；



> 设备控制器不属于os范畴，它是属于硬件，而设备驱动程序属于os的一部分，os内核代码可以像本地调用代码一样使用设备驱动程序的接口
> 不同的设备控制器虽然功能不同，但是设备驱动程序会提供统一的接口给os，允许不同的设备驱动

> 程序以相同的方式接入os
> 设备驱动程序中包含中断处理程序

3.linux存储系统io的层次

- 文件系统层：包括虚拟文件系统和其他文件系统的具体实现，它向上为应用程序统一提供了标准的文件访问接口，向下会通过通用块层来存储和管理磁盘数据。
- 通用块层：包括块设备的I/O队列和I/O调度器，它会对文件系统的I/O请求进行排队，再通过I/O调度器，选择一个I/O发给下一层的设备层。
- 设备层：包括硬件设备、**设备控制器和驱动程序**，负责最终物理设备的I/O操作。

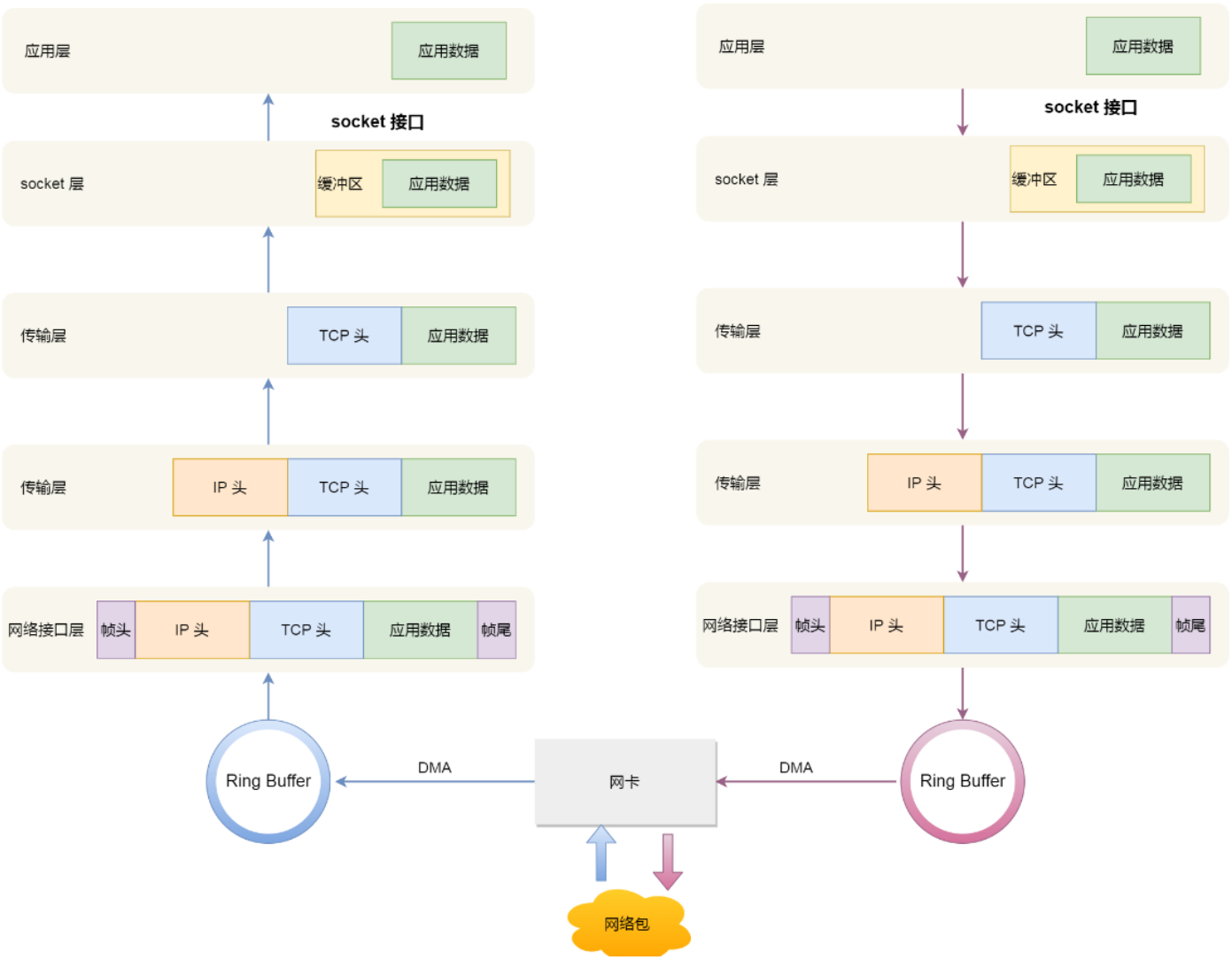
4.当我们在键盘上敲击字母"A"时，操作系统内部会发生以下过程：

- 键盘控制器检测到键盘输入事件,并将对应的扫描码放入键盘控制器的寄存器中。
- 键盘控制器通过总线向CPU发送中断请求
- cpu收到中断请求后，os会保存被中断进程的cpu上下文，然后调用键盘的**中断处理程序**
- 中断处理程序会根据扫描码查找对应的ASCII字符码,并将其放入**读缓冲区队列**
- 显示设备驱动程序会定期从读缓冲区队列读取数据到写缓冲区队列, 并将其写入到显示设备的控制器的寄存器/数据缓冲区，最后将其显示到屏幕上。
- 显示出结果后，回复被中断进程的上下文。

网络系统

1. linux接收网络包流程
    i. 当网卡NIC收到一个网络包时，会通过DMA这种io控制方式，将网络包放入一个环形缓冲区（ring buffer）；
        a. ring buffer是系统分配的一块内存空间
        b. io控制方式是将数据从io设备传到内存
    ii. 网卡发起**硬件中断**，执行网卡硬件中断处理函数，然后暂时屏蔽中断，唤醒**软中断**来轮询处理数据，直到没有新数据时才恢复中断，这样就可以一次中断处理多个网络包。
    iii. 软中断是怎么处理网络包的?
        a. 从ring buffer拷贝数据到内核structured sk_buff缓冲区，从而可以把这个网络包交给网络协议栈进行逐层处理了。
        b. 网络接口层：检查报文合法性，非法丢弃；合法则分辨出上层协议(ipv4 or v6?)，去掉帧头帧尾，交给网络层
        c. 网络层：判断网络包是否需要转发，如不需要，从ip头中分辨出上层协议(UDP or TCP?)，去掉ip头，交给传输层
        d. 传输层：取出TCP/UDP头，根据ip和端口找到对应的socket，将包内的应用数据拷贝到socket的接收缓冲区

e. 应用层：调用socket接口，从内核中的socket接收缓冲区读取数据到应用层



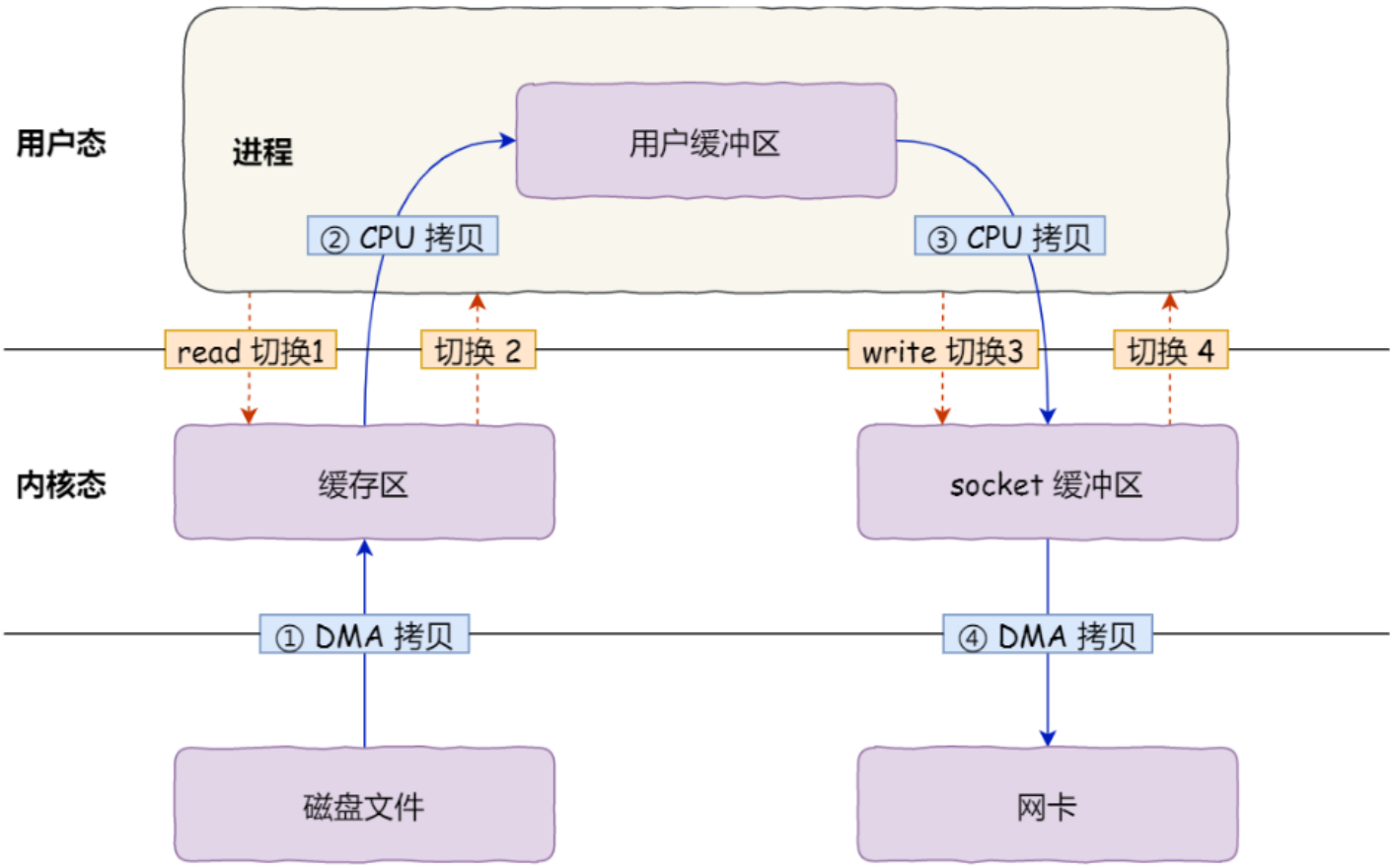
2.linux发送网络包流程

- 应用程序调用socket发送数据包的接口（系统调用），**从用户态陷入内核态的socket层**，socket层会将应用数据拷贝到socket发送缓冲区
- 网络协议栈从socket发送缓冲区取出数据包，从上往下进行逐层封装
  - 如使用TCP，在传输层增加TCP包头，然后交给网络层，网络层会增加IP头，然后通过**查询路由表确认下一跳的IP**，并按照MTU大小进行**分片**
  - 分片后的网络包被送到网络接口层，通过**ARP协议获得下一跳的MAC地址**，然后增加帧头和帧尾，放到ring buffer中
- 触发软中断通知网卡驱动程序，网卡驱动程序通过DMA从发包队列中读取网络包到NIC的队列，然后发送。

3.传统的糟糕的文件传输方式（从磁盘取文件读取文件，给到网卡）
使用以下两个系统调用:

```
read(file, tmp_file, len);
write(socket, tmp_file, len);
```

具体操作如图所示



问题：
2次系统调用发生了4次用户态和内核态之间的切换，需要进行上下文切换，造成消耗。
发生4次数据拷贝，两次DMA拷贝，两次CPU拷贝

这种简单又传统的文件传输方式，**存在冗余的上文切换和数据拷贝，在高并发系统里是非常糟糕的**，多了很多不必要的开销，会严重影响系统性能。

所以**如何优化**呢?
a.想要减少上下文切换次数，就要减少系统调用（之所以要发生上下文切换，是因为用户空间没有权限操作磁盘或网卡，这些操作设备的过程都需要交由操作系统内核来完成）
b.如何减少数据拷贝的次数？砍掉用户缓冲区（因为在文件传输任务中，用户空间并不会对数据再加工）

4.如何实现**零拷贝**