操作系统的启动流程

1. CPU初始化，找到第一条指令

   CS寄存器内容左移四位加上IP，计算出第一条指令的位置，即存放BIOS的位置。

2. 进入BIOS，进行BOIS的初始化（CPU要进行处理的第一个“可执行程序”）

   BIOS中包含了CPU的相关信息、设备启动顺序信息、硬盘信息、内存信息、时钟信息、PnP特性等等。在此之后，计算机心里就有谱了，知道应该去读取哪个硬件设备了。

3. 读取主引导扇区MBR

   主引导扇区中存放着预启动信息、分区信息

4. 跳到活动分区的引导扇区，执行加载程序，开始加载操作系统到内存中。

   加载程序的细化
   我们说加载程序它首先不是直接去加载你的内核，而是去文件系统当中读一个启动配置文件。依据配置文件的配置（参数）去加载内核，依据这个，就选择你启动的模式，比如说我是在正常启动，还是说我是在安全模式启动，还是说我是在一个调试状态下启动我的系统。那这些区别都会读出来之后，它导致我在加载内核的时候的一些内核会不一样，或者说我加载的时候的参数会不一样。

Linux系统的启动过程

1. 启动第一步 加载BIOS
2. 启动第二步 读取主引导记录MBR

   系统找到BIOS所指定的硬盘的MBR后，就会将其复制到0×7c00地址所在的物理内存中。其实被复制到物理内存的内容就是Boot Loader，而具体到你的电脑，那就是lilo或者grub了。

3. 启动第三步Boot Loader

   Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核做好一切准备。

4. 启动第四步 加载内核
5. 启动第五步 用户层init依据inittab文件来设定运行等级

   内核被加载后，第一个运行的程序便是/sbin/init，该文件会读取/etc/inittab文件，并依据此文件来进行初始化工作。

   其实/etc/inittab文件最主要的作用就是设定Linux的运行等级，其设定形式是“:
   id:5:initdefault:”，这就表明Linux需要运行在等级5上。

6. 启动第六步init进程执行rc.sysinit

   Linux系统执行的第一个用户层文件就是/etc/rc.d/rc.sysinit脚本程序，它做的工作非常多，包括设定PATH、设定网络配置（/etc/sysconfig/network）、启动swap分区、设定/proc等等。

7. 启动第七步 启动内核模块

   具体是依据/etc/modules.conf文件或/etc/modules.d目录下的文件来装载内核模块。

8. 启动第八步 执行不同运行级别的脚本程序

   根据运行级别的不同，系统会运行rc0.d到rc6.d中的相应的脚本程序，来完成相应的初始化工作和启动相应的服务。

9. 启动第九步 执行/etc/rc.d/rc.local

This script will be executed *after* all the other init scripts.

You can put your own initialization stuff in here if you don't

want to do the full Sys V style init stuff.

10. 启动第十步

    执行/bin/login程序，进入登录状态

操作系统的中断是什么

定义：中断就是在计算机执行期间，系统内发生任何非寻常的或者非预期的**急需处理**事件，使得CPU暂时中断当前正在执行的程序，保存现场后自动去执行相应的处理程序，待处理完后再返回到中断处向下执行。

两类：

1. 一种是由CPU外部引起的，称为**外中断。**指来自CPU执行指令以外的事件的发生，如**I/O**中断、**时钟**中断
2. 一种是来自CPU内部事件或程序执行中引起的中断，例如程序**非法操作**，地址越界、浮点溢出）称为**内中断**，或者（异常，陷入）。

- 中断和异常（陷入）的区别

  主要**区别**是**信号的来源**，看是来自CPU外部，还是CPU内部。

- 中断处理程序的过程

  1. 测定是否有未响应的中断信号
  2. 保护被中断进程的CPU环境。
  3. 转入相应的设备处理程序。切换到内核状态
  4. 中断处理。
  5. 恢复CPU的现场并退出中断。

用户态和内核态

内核态与用户态是操作系统的两种运行级别，在内核态下CPU可执行**任何**指令，在用户态下CPU只能执行**非特权指令。**当CPU处于内核态，可以随意进入用户态；而当CPU处于用户态时，用户从用户态切换到内核态只有在**系统调用**和**中断**两种情况下发生，一般程序一开始都是运行于用户态，当程序需要使用系统资源时，就必须通过调用软中断进入内核态。

- 用户态切换到内核态的三种方式

  1. 系统调用：用户态进程主动要求切换到内核态的一种方式。用户态进程通过系统调用，申请使用操作系统提供的服务程序，以完成工作。
  2. 异常：
  3. 中断

程序和进程的区别

1. 程序是一个静态概念，进程是一个动态概念
2. 进程需要分配具体的地址空间，一般由PCB，代码段，数据段组成，所以进程里面不仅仅只有代码
3. 进程是资源分配的最小基本单位
4. 不同的进程可以包含同一程序

什么是操作系统

操作系统是一个系统软件，负责管理协调硬件，软件等计算机资源的工作，为上层的应用程序，用户提供简单的服务接口。进程管理、内存管理、文件管理、IO设备管理。

主要功能：

1. 进程管理：进程控制，进程同步，进程通信和进程调度
2. 内存管理：内存分配，内存保护，地址映射，内存扩充
3. IO设备管理：管理所有外围设备，包括完成用户IO请求，为用户进程分配IO设备，提高IO设备利用率
4. 文件管理：管理用户文件和系统文件，方便使用的同时保证安全性。包括磁盘存储空间管理，目录管理，文件读写管理以及文件共享及保护
5. 提供用户接口：系统调用

操作系统的特性

1. 并发：同一时段可以多个程序执行
2. 共享：系统中的资源可以被多个并发执行的进线程共同使用
3. 虚拟：把一个物理实体虚拟为多个
4. 不可预知：进程以走走停停的方式执行，且以一种不可预知的速度推进

进程管理

什么是进程间通信？通信的目的?

定义：进程用户空间是相互独立的，一般而言是不能相互访问的。但很多情况下进程间需要互相通信，来完成系统的某项功能。进程通过与内核及其它进程之间的互相通信来协调它们的行为。

目的：

1. 数据传输：一个进程需要将它的数据发送给另一个进程
2. 共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）
3. 资源共享：多个进程之间共享同样的资源。为了作到这一点，需要内核提供锁和同步机制。
4. 进程控制：有些进程希望完全控制另一个进程的执行（如**Debug进程**），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

方式 7种：

1. 管道：半双工的通信方式，数据单向流动，用于父子进程间的通信。
2. 命名管道FIFO：允许在不同进程间通信
3. **消息队列：**消息队列是由消息的链表，
4. 共享内存：映射一段内存可以被其他进程访问，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 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。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。
5. 信号量：整型信号量（相当于互斥锁）和记录型信号量（相当于互斥锁加条件变量，用于生产者消费者模型）
6. 信号：用于通知接收进程某个事件已经发生
7. socket：可用于不同主机之间及其间的进程通信。

进程与线程的区别?

- 进程是资源分配的基本单位，进程需要管理好它的资源，PCB，进程需要维护其上下文包括环境变量，进程所掌控的资源，有中央处理器，有内存，打开的文件，映射的网络端口等等。
- 线程是调度的基本单位，线程由进程创建，由同一个进程创建的多个线程是共享进程的资源。线程关注的是利用CPU去执行代码。因此，线程的切换的成本要小很多，只涉及CPU以及相关寄存器。

用户级线程（协程）和内核级线程：区别在于对于操作系统是否可见

- 协程

  为什么是更轻量级的? 因为协程创建的代价很小 线程比协程的创建需要的资源大了成百上千倍. 那这是怎么做到的呢? 协程只需要很小很小的空间. 其他的东西都是多个协程共享的.
  这个时候可能会有人问了 共享的不就会出现并发问题了吗?不是的协程对cpu并不可见. 也就是cpu是看不见协程的同一时刻一个cpu只会运行一个协程任务那么这样是不是就保证了安全性?

进程调度算法

1. FCFS 先进先出非抢占式
2. SJF（短作业优先） 默认为非抢占式
3. HRRN（高响应比） 非抢占式
4. 时间片轮转
5. 优先级调度
6. 多级反馈队列 -- 抢占式，可能会造成饥饿

一个进程可用创建多少线程？和什么有关?

一个进程可以创建的线程数由可用虚拟空间和进程的栈的大小共同决定。

在x86 32位系统的情况下，进程的用户可用虚拟地址空间是2G，系统占用2G，默认分配栈大小为1M时，创建线程的最大数量就为2G/1M=2048个。

内存管理

内存保护（地址保护机制）

1. 在CPU中设置一对上，下限寄存器，存放进程的上下限地址，进程的指令要访问某个地址时，cpu检查是否越界
2. 采用重定位寄存器和界地址寄存器进行越界检查，重定位寄存器存放的是进程的物理起始地址，界定位寄存器存放的是最大逻辑地址

内存空间扩展

1. 交换技术：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中已准备运行条件的进程换入内存(中级调度：内存调度)
2. 虚拟存储技术：当进程运行时，先将其中一部分装入内存，另一部分暂留在磁盘，当要执行的指令或访问的数据不在内存时，由操作系统自动完成将他们从磁盘调入内存的工作

内存空间的分配与回收

1. **连续分配方式**(为用户进程分配的是一个连续的内存空间)单一连续，固定分区，动态分区
2. **离散分配方式**
   - **基于分页存储管理**
     1. 概念：

将内存划分为一个个大小相等的分区，每个分就是一个页框(物理块)，将用户进程的地址空间分为与页框一样大小的区域称作页或者页面，操作系统以页框为单位为各个进程分配内存空间，各个页面不必连续存放。

2. 地址转换（进程地址空间->物理地址）

   页号+页内偏移量
   页号 = 逻辑地址 / 页面长度
   页内偏移量 = 逻辑地址 % 页面长度

   因此只要知道对应页号的起始地址+页内偏移量

3. 页表

   为了知道页面在内存中存放的位置，操作系统为每个进程建立一张页表，由页号与物理块号组成，页表记录的是进程页面和实际存放的内存块之间的对应关系

4. 基本地址变换机构

   根据逻辑地址算出页号，页内偏移量，然后通过**页表寄存器**判断出页号是否越界，没有越界就去查询页表(内存中)找到对应内存的物理块号，再加上偏移量，页表寄存器里面有页表在内存中的起始地址和页表长度

5. 具有快表的地址变换机构（TLB）

   快表：联想寄存器，访问速度比内存快恨多，用来存放当前访问的若干页表项，以加速地址变换。

6. 多级页表 建立页目录项

   单级页表的缺点：
   1.页表必须连续存放，当页表很大时，需要占用多个连续的页框
   2.没有必要让整个页表常驻内存，因为进程在一段时间内可能只需要访问某个特定的页面

- **基于分段存储管理**

  1. 进程的地址空间，按照**自身的逻辑关系**划分为若干个段，每一个段都有一个段名，每段从0开始编址，内存分配以段为单位进行分配，每个段在内存中占据连续空间，但各段之间可以不相邻
  2. 段表
  3. 通常，程序员把子程序、操作数和常数等不同类型的数据划分到不同的段中，并且每个程序可以有多个相同类型的段

- 对比

  1. 分页的主要目的是为了实现离散分配，**提高内存利用率**，所以分页是针对系统的行为，对用户不可见 **操作系统角度**
  2. 分段则是为了更好地**满足用户需求**，一个段通常包含一组属于一个逻辑模块的信息，分段对用户可见，用户编程时需要显示的给出段名 **用户角度**
  3. 页的大小固定由系统决定，段的长度不固定，决定于用户编写的程序
  4. 分段管理更容易实现信息的共享与保护

- **基于段页式存储管理**

  1. 分段+分页

     段号的位数决定了每个进程最多分几个段，页号位数决定每个段最多有几个页，页内偏移量决定页面大小

  2. 根据逻辑地址得到段号，页号，页内偏移量，查段表寄存器是否越界，查段表，查页表，再访问内存

  3. 所以使用段页式分配会多一次的访问内存

虚拟内存

理论基础：局部性原理，先装入一部分可能会用到的程序进内存，先执行，暂时用不到的留在外存当程序执行时，访问信息不再内存，操作系统负责从外存调入内存，若内存空间不够则需要选择换出一部分信息

特性：多次性、对换性、虚拟性

- 如何实现？

  请求分页存储管理、请求分段存储管理、请求段页式存储管理

- 与传统的非连续内存存储管理的区别？

  在程序执行过程中，当访问的信息不再内存，需要调入内存，若内存不够，需要将内存中暂时不用的信息换出到外存

- **请求分页式存储管理**

  1. 页表机制

     增加了四个标志位：状态位、访问字段、修改位、外存地址

  2. 缺页中断机构

     页面置换算法

- **发生缺页中断之后的过程？**

  1. 进程陷入内核态
  2. 检查要访问的虚拟地址是否合法
  3. 查找/分配一个物理页
  4. 填充物理页内容（读取磁盘）
  5. 建立映射关系（虚拟地址到物理地址）
  6. 恢复中断现场

- **页面置换算法？**

  1. **最佳置换算法：**选择以后最晚用到的
  2. **FIFO：**先进先出。选择在内存驻留时间最长的页面进行置换：可用链表记录，可能产生belady异常，增加物理块，缺页次数反而增大。
  3. **LRU**：选择最长时间没有被引用的页面进行置换
  4. **时钟置换算法：**在页表项中增加访问位，记录访问的情况，然后组成一个链表，页面装入内存时，访问位置0，访问后置1，进行页面置换时，换出访问位为0的，如果此时访问位为1，则置为0，并将指针调到下一位，直到找到可置换的
  5. **LFU：**发生缺页中断时，置换访问次数最少的页面

- **抖动问题**

  刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数(分配给进程的物理块不够)

  为进程分配的物理块太少，会使进程发生抖动现象

死锁

- 死锁的必要条件

  互斥条件，请求且保持，不可剥夺，循环等待

- **死锁避免**的方法

  用某种方法防止系统进入不安全状态，从而避免死锁(银行家算法)

安全序列：指系统按照某一种方式分配资源，则每个进程都能顺利完成，只要能找到只要一个安全序列，那么系统就是安全的，如果一个都找不到，那系统就进入了不安全状态，如果系统处于安全转态，就一定不会发生死锁，进入不安全转态可能发生死锁(可能有进程提前归还了一些资源)

银行家算法的核心思想:在资源分配之前预先判断这次分配是否会导致系统进入不安全状态，如果会则暂时不答应这次请求，让该进程先阻塞等待

什么是活锁？与死锁的区别？

活锁指的是 任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。 活锁和死锁的区别在于，**处于活锁的实体是在不断的改变状态**，所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

活锁应该是一系列进程在轮询地等待某个不可能为真的条件为真。活锁的时候进程是不会blocked，这会导致耗尽CPU资源。

为解决活锁可以引入一些随机性，例如如果检测到冲突，那么就暂停随机的一定时间进行重试。这回大大减少碰撞的可能性

临界区

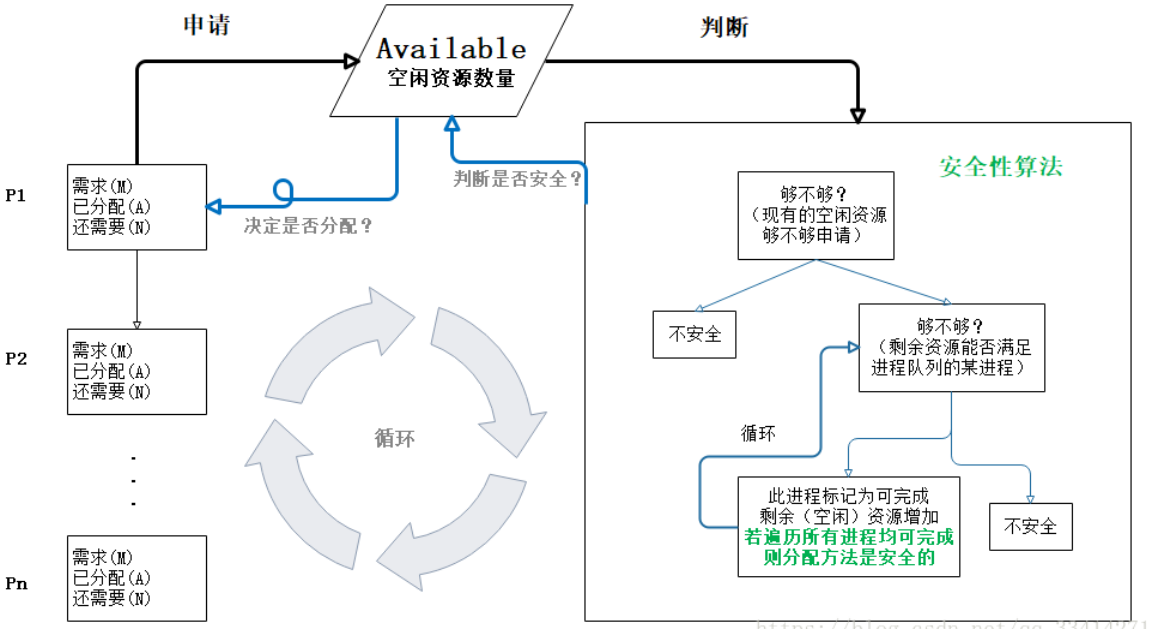
每个进程中，访问临界资源的那段程序块被称为临界区。每次只准许一个进程进入临界区，进入后不允许其他进程进入

遵守：空闲让进 忙则等待 有限等待 让权等待

银行家算法

银行家算法是一种最有代表性的**避免死锁**的算法。在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。

安全状态：假设资源P1申请资源，银行家算法先试探的分配给它（当然先要看看当前资源池中的资源数量够不够），若申请的资源数量小于等于Available，然后接着判断分配给P1后剩余的资源，能不能使进程队列的某个进程执行完毕，若没有进程可执行完毕，则系统处于不安全状态（即此时**没有一个进程能够完成并释放资源**，随时间推移，系统终将处于死锁状态）。



死锁避免：**即找到一种资源分配方式，可以使得系统处于安全状态。**

设备管理

磁盘调度算法

1. 先来先服务算法
2. 最短寻找时间优先--每次都优先响应距离磁头最近的磁道访问请求(可能导致饥饿)
3. 扫描算法(scan)：只有当磁头移动到最边缘磁道时才改进移动方向，缺点是对各个磁道的响应频率不均衡，才访问的磁道可能一会又要访问
4. 循环扫描算法(c-scan)：只有磁头朝某个方向移动时才会响应请求，移动到边缘后立即让磁头移动到起点位置，途中不响应任何请求
5. look算法--scan算法的改进，只要在磁头移动方向上不再有请求，就要改变磁头方向
6. c-look算法:c-scan算法的改进

减少磁盘延迟的方法

1. 交替编号：让编号相邻的扇区在物理上不相邻，因为我们读取完一个扇区后需要一段时间处理才可以进行读入下一个扇区
2. 错位编号

实际问题

Linux/进程的虚拟地址空间布局

- 布局图

   Linux进程的虚拟地址空间布局

进程分配内存的两种方式--brk()和mmap()系统调用

- 如何查看进程发生缺页中断的次数

  `ps -o majflt, minflt -C program`

  majflt代表major fault，中文名叫大错误，minflt代表minor fault，中文名叫小错误。这两个数值表示一个进程自启动以来所发生的缺页中断的次数。

- Linux进程分配内存的两种方式，brk和mmap（不考虑内存共享）

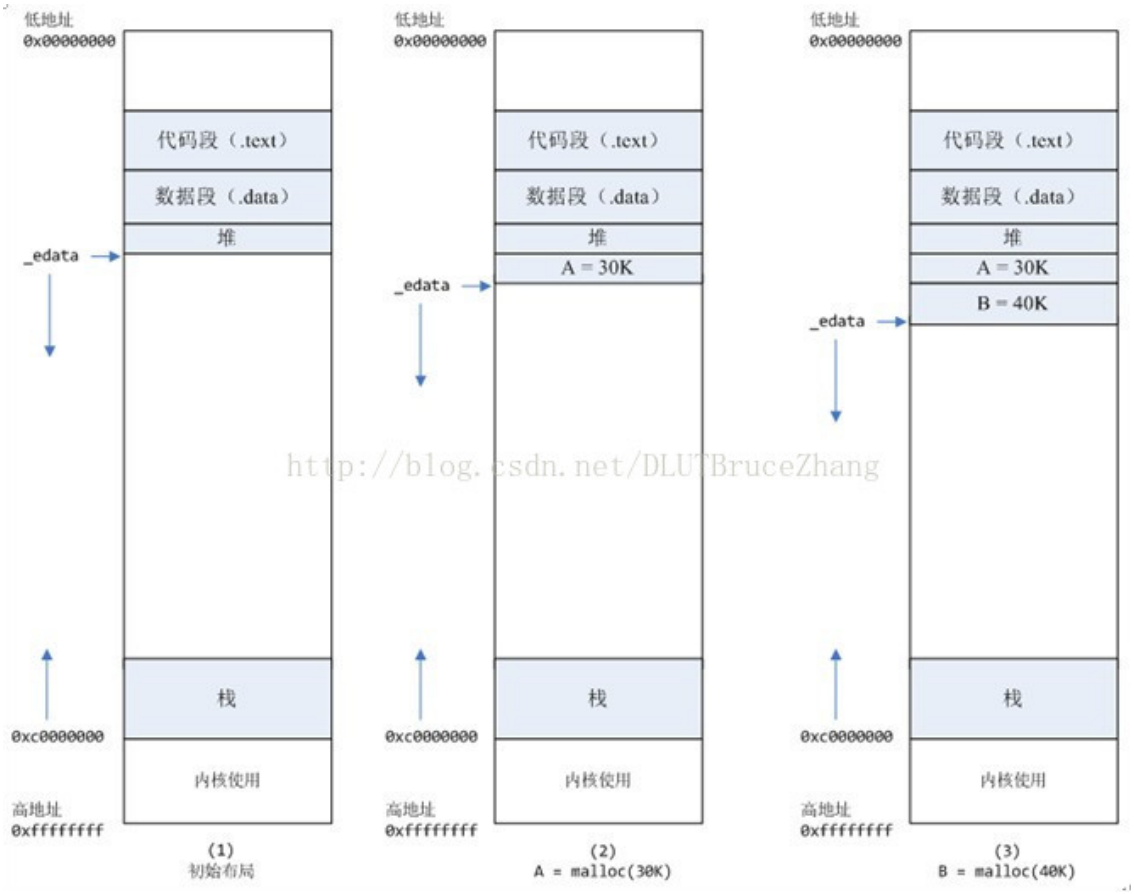
  1. brk是将数据段(.data)的最高地址指针_edata往高地址推；
  2. **mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。**

  这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

  在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk，mmap，munmap这些系统调用实现的。

- 内存分配原理

  情况一：malloc小于128k的内存，使用brk分配内存，将_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)

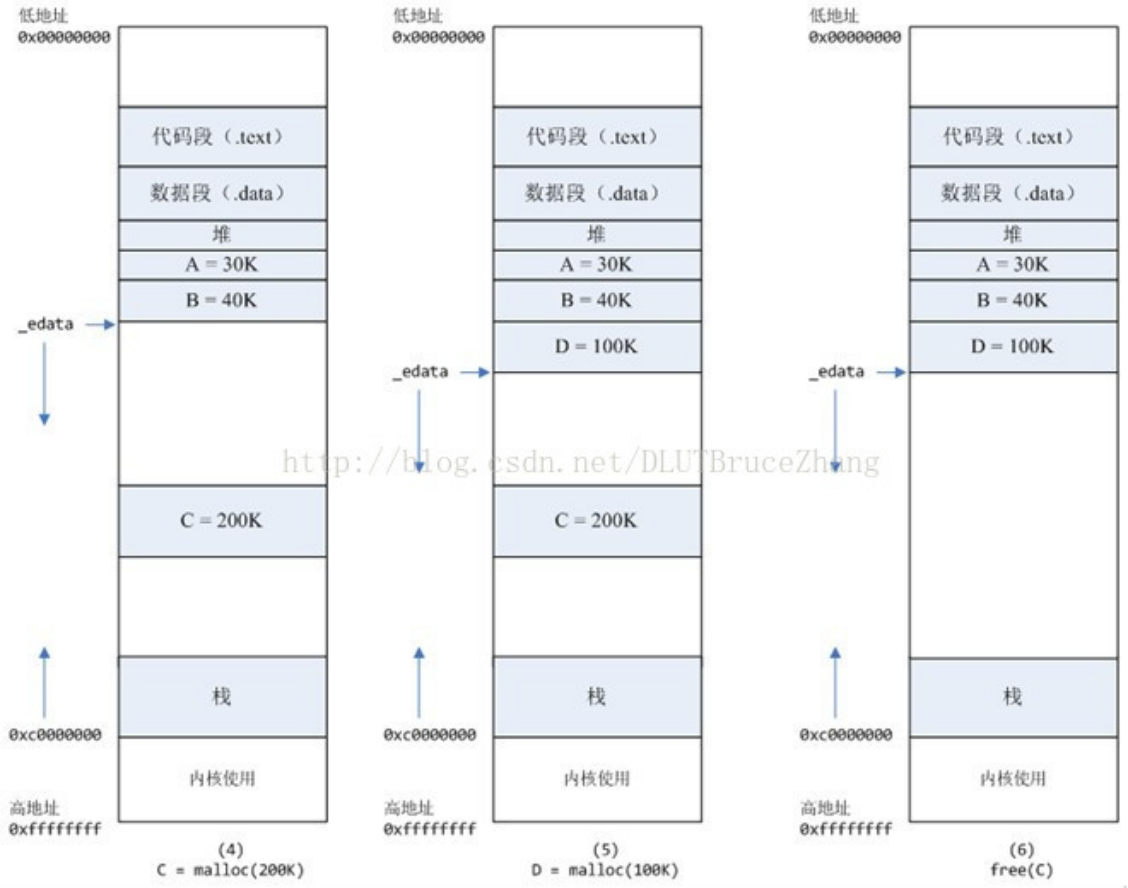


1. 进程启动的时候，其（虚拟）内存空间的初始布局如图1所示。其中，mmap内存映射文件是在堆和栈的中间（例如libc-2.2.93.so，其它数据文件等），为了简单起见， 省略了内存映射文件。_edata指针（glibc里面定义）指向数据段的最高地址。
2. 进程调用A=malloc(30K)以后，内存空间如图2：

   malloc函数会调用brk系统调用，将_edata指针往高地址推30K，就完成虚拟内存分配。你可能会问： 只要把_edata+30K就完成内存分配了？

   事实是这样的，_edata+30K只是完成虚拟地址的分配， A这块内存现在还是没有物理页与之对应的， 等到进程第一次读写A这块内存的时候，发生缺页中断，这个时候，内核才分配A这块内存对应的物理页。 **也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。**

情况二：malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)，如下图：



1. 进程调用C=malloc(200K)以后，内存空间如图4：默认情况下，malloc函数分配内存，如果请求内存大于128K（可由M_MMAP_THRESHOLD选项调节），那就不是去推_edata指针了，而是利用mmap系统调用，从堆和栈的中间分配一块虚拟内存。 这样子做主要是因为:brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候紧缩看下面），而mmap分配的内存可以单独释放。当然，还有其它的好处，也有坏处，再具体下去，有兴趣的同学可以去看glibc里面malloc的代码了。
2. free(C)之后，C对应的虚拟内存和物理内存一起释放
3. free(B), B对应的虚拟内存和物理内存都没有释放，因为只有一个_edata指针，如果往回推，那么D这块内存怎么办呢?

   当然，B这块内存，是可以重用的，如果这个时候再来一个40K的请求，那么malloc很可能就把B这块内存返回回去了。
4. free(D)以后，B和D连接起来，变成了140k的空闲内存，于是内存紧缩。

内存紧缩（trim）：当最高地址空间的空闲内存超过128K（可由M_TRIM_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。

Linux内存管理

- 内核空间
  - 页

    内核的内存管理基本单位
  - 区

    内核把页划分为不同的区
  - 字节分配与释放

    kmalloc和vmalloc

分配函数	区域	连续性	大小	释放函数	优势
kmalloc	内核空间	物理地址连续	最大值128K-16	kfree	性能更佳
vmalloc	内核空间	虚拟地址连续	更大	vfree	更易分配大内存
malloc	用户空间	虚拟地址连续	更大	free	

    - slab层

      slab分配器的作用：

        - 对于频繁地分配和释放的数据结构，会缓存它；
        - 频繁分配和回收比如导致内存碎片，为了避免，空闲链表的缓存会连续的存放，已释放的数据结构又会放回空闲链表，不会导致碎片；
        - 让部分缓存专属单个处理器，分配和释放操作可以不加SMP锁；

- 用户空间

  用户空间中进程的内存，往往称为**进程地址空间**。Linux采用**虚拟内存技术**

    - 地址空间

      每个进程都有一个32位或64位的地址空间，取决于体系结构。 一个进程的地址空间与另一个进程的地址空间即使有相同的内存地址，也彼此互不相干，对于这种共享地址空间的进程称之为线程。一个进程可寻址4GB的虚拟内存（32位地址空间中），但不是所有虚拟地址都有权访问。对于进程可访问的地址空间称为**内存区域**。每个内存区域都具有对相关进程的可读、可写、可执行属性等相关权限设置。

      内存区域可包含的对象：

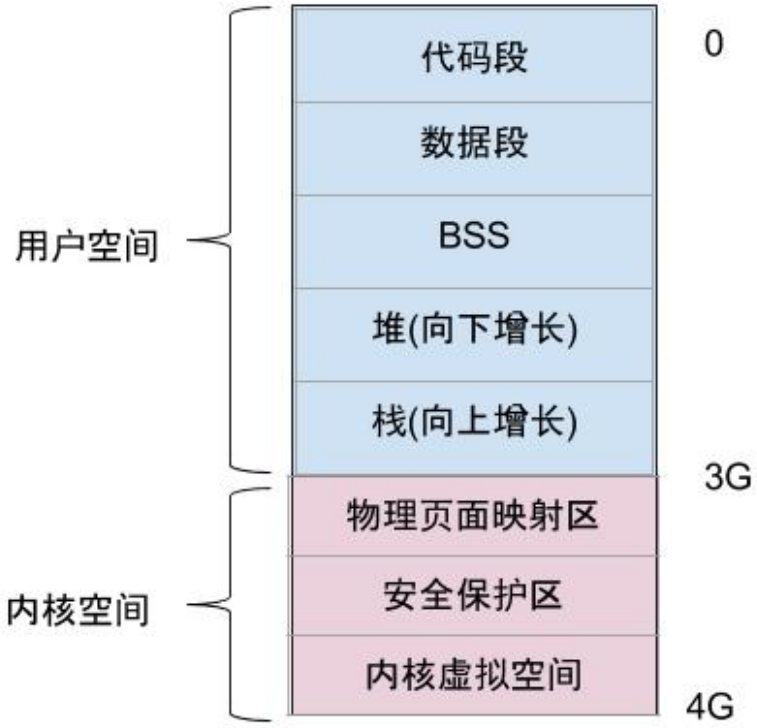
        - 代码段(text section)： 可执行文件代码
        - 数据段(data section)： 可执行文件的已初始化全局变量（静态分配的变量和全局变量）。
        - bss段（Block Started by Symbol）：程序中未初始化的全局变量，零页映射（页面的信息全部为0值）。静态变量
        - 进程用户空间栈的零页映射（进程的内核栈独立存在并由内核维护）
        - 每一个诸如C库或动态连接程序等共享库的代码段、数据段和bss也会被载入进程的地址空间
        - 任何内存映射文件
        - 任何共享内存段
        - 任何匿名的内存映射（比如由**malloc()分配**的内存）堆区

      这些内存区域不能相互覆盖，每一个进程都有不同的内存片段。

- 进程与内存

    - 对于普通进程对应的内存空间包含5种不同的数据区：

        - 代码段
        - 数据段
        - BSS段 特点是:可读写的，在程序执行之前BSS段会自动清0。
        - 堆：动态分配的内存段，大小不固定，可动态扩张(malloc等函数分配内存)，或动态缩减(free等函数释放)；
        - 栈：存放临时创建的局部变量；



4G进程地址空间

- 内存分配

  进程分配内存，陷入内核态分别由brk和mmap完成，但这两种分配还没有分配真正的物理内存，真正分配在后面会讲。

  - brk： **数据段**的最高地址指针_edata往高地址推
    - 当malloc需要分配的内存<M_MMAP_THRESHOLD（默认128k）时，采用brk;
    - brk分配的内存需高地址内存全部释放之后才会释放。(由于是通过推动指针方式)
    - 当最高地址空间的空闲内存大于M_TRIM_THRESHOLD时(默认128k)，执行内存紧缩操作；
  - do_mmap：在堆栈中间的文件映射区域找空闲的虚拟内存
    - 当malloc需要分配的内存>M_MMAP_THRESHOLD（默认128k）时，采用do_map();
    - mmap分配的内存可以单独释放
- 物理内存
  - 物理内存只有进程**真正去访问虚拟地址**，发生缺页中断时，才分配实际的物理页面，建立物理内存和虚拟内存的映射关系。
  - **应用程序操作的是虚拟内存**；而**处理器直接操作的却是物理内存。**当应用程序访问虚拟地址，必须将虚拟地址转化为物理地址，处理器才能解析地址访问请求。
  - 物理内存是通过分页机制实现的
  - 物理页在系统中由也结构struct page描述，所有的page都存储在数组mem_map[]中，可通过该数组找到系统中的每一页。