操作系统基本概念

2019年7月3日　　23:31

基本特征：

- 并发：宏观上一段时间内可以运行多个程序，，两个程序交织运行
  - 通过引入进程与线程，使得程序能够并发运行
- 并行：同一时刻能运行多个指令，指的是真正的同时运行多个程序
  - 需要硬件支持，如多核处理器、分布式计算系统
- 共享：系统中的资源可以被多个并发进程共同使用
  - 互斥共享：互斥共享的资源称为临界资源
    - 临界资源：例如打印机等（在同一时刻只允许一个进程来访问，需要同步机制来实现）
  - 同时共享
- 虚拟：一个物理实体转换为多个逻辑实体，主要有两种虚拟技术
  - 时间分复用技术：多个进程在同一个CPU上并发执行，每个进程轮流占用CPU，每次只执行一小个时间片段并快速切换
  - 空间分复用技术：虚拟内存（将物理内存抽象为地址空间，每个进程都有各自的地址空间）
- 异步：进程不是一次性执行完，而是走走停停，已未知的速度向前推进

基本功能：

- 进程管理：进程控制，进程同步，进程通信，死锁处理等
- 内存管理：内存分配，地址映射，内存保护与共享，虚拟内存
- 文件管理：文件存储空间的管理，目录管理，文件读写管理和保护
- 设备管理：完成用户的I/O请求，使用各种设备，

系统调用：上述基本功能就属于系统调用，属于内核态

Linux系统调用函数：

Task	Commands
进程控制	fork(); exit(); wait();
进程通信	pipe(); shmget(); mmap();
文件操作	open(); read(); write();
设备操作	ioctl(); read(); write();
信息维护	getpid(); alarm(); sleep();

Task	Commands
进程控制	fork(); exit(); wait();
进程通信	pipe(); shmget(); mmap();
文件操作	open(); read(); write();
设备操作	ioctl(); read(); write();
信息维护	getpid(); alarm(); sleep();
安全	chmod(); umask(); chown();

屏幕剪辑的捕获时间: 2019/7/3 23:45

- 大内核：将操作系统功能作为一个紧密结合的整体放到内核，由于各个模块共享信息所以有很高的性能
- 微内核：将操作系统一部分系统功能移出内核，从而降低内核的复杂性。溢出的部分根据分层的原则分成若干服务，相互独立。单因为频繁的在用户态和核心态切换，所以有性能损失

中断分类

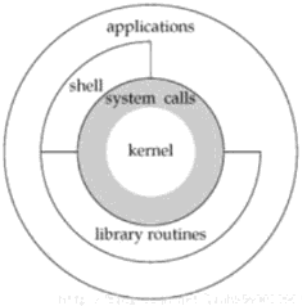
- 外中断：由CPU执行指令以外的时间引起，（例：I/O完成中断，表示设备输入输出已经完成，CPU可以发送下一个IO请求）
- 异常：由CPU执行指令的内部事件引起，如算术溢出，地址越界等
- 陷入：在用户程序中使用系统调用

用户态和内核态

2019年7月4日 星期四 13:38

用户态和内核态指的是：CPU执行程序的两种级别

- 高执行级别，代码可以执行特权命令，访问任意的物理地址（内核态）
- 低执行级别，代码的掌控范围会受到限制，只能在对应级别允许的范围内活动



Intel X86架构CPU又四个级别（0-3），0级最靠近核心，特权最高，3级特权最低，此时：

- 用户态位于3级，不能访问0级的地址空间
- 但当一个进程因为系统调用进入内核代码中执行程序时，则处于内核态
- 当用户运行一个程序，初始处于用户态，如果要执行文件操作，网络数据发送等操作则必须通过write，send等系统调用，这些系统调用会调用内核的代码，进程会切换到0级，执行3G-4G的内核地址空间去执行内核代码来完成相应操作，执行完毕后又会切换回3级，达到安全保护的作用

进程与线程

2019年7月6日 0:41

进程与线程基本

2019年6月29日 23:54

- 定义
  - 进程（火车）：
    - 是对运行程序的一个封装
    - 是系统进行资源调度的最小单位
    - 实现了操作系统的并发
  - 线程（车厢）：
    - 是进程的子任务
    - 是CPU调度的最小单位
    - 实现了进程内部的并发
    - 是操作系统可识别的最小执行单位
    - 每个线程都肚子占用一个虚拟处理器
    - 每个线程完成不同任务，单都共享当前所属进程的内存空间
- 两者都是CPU工作时间段的一个描述，只不过颗粒大小不同

- 区别

			系统开销	通信			
进程	拥有独立的内存单元	资源分配的最小单位	创建和撤销时，因为涉及资源的操作，所以开销大		变成调试简单可靠性高，但创建销毁开销大	进程间不会相互影响	适应于多核，多机分布
线程	多个线程共享所属进程的内存空间	CPU调度的最小单位	只涉及寄存器的操作，开销极小	同一进程的线程因为具有相同的内存空间，同步与通信较容易	开销小，切换速度快，但编程调试复杂	一个线程挂掉会使得当前进程挂掉	使用与多核

- 优缺点
  - 线程执行开销小，但不利于资源的管理和保护
  - 而进程正相反

多线程

2019年7月4日 星期四 13:28

定义：值一个进程中又多个线程

为什么要用多线程：

- 更好的利用CPU资源
  - 如果只有一个线程，则第二个线程必须等到第一个线程结束之后才能进行
  - 如果使用多线程则可以在主线程执行的同事执行其他任务
- 进程间不能共享数据，但线程可以，所以多线程可以更好地安排数据的调用
- 创建线程代价比创建进程小得多

线程安全
考虑的是线程之间的共享变量

如何创建线程： 使用pthread_create(thread, attr, start_routine, arg)函数

- 参数

参数	描述
thread	指向线程标识符指针。
attr	一个不透明的属性对象，可以被用来设置线程属性。您可以指定线程属性对象，也可以使用默认值 NULL。
start_routine	线程运行函数起始地址，一旦线程被创建就会执行。
arg	运行函数的参数。它必须通过把引用作为指针强制转换为 void 类型进行传递。如果没有传递参数，则使用 NULL。

屏幕剪辑的捕获时间: 2019/7/5 23:59

- 返回值：如果线程创建成功，函数返回0，若返回值不为0则失败
- 例：int ret = pthread_create(&tids[i], NULL, say_hello, NULL);

线程同步

2019年7月6日 0:04

为什么要线程同步：

- 目的：正确处理多线程并发时的问题
    - 例如线程的等待，多个线程访问统一数据时的互斥，防死锁等

线程同步的方式

- 临界区
    - 多个 线程访问一个独占性资源时候，可以使用临界区对象
      拥有临界区的线程可以访问被保护起来的资源或代码段，如果此时其他线程想访问，则被挂起，知道拥有临界区的线程用完为止
    - 原理：将多个线程串行起来访问公共资源或代码
    - 特点：速度快，适合控制数据的访问权限
- 互斥量（多用于互斥对象机制）：
    - 和临界区非常相似，只是其允许在进程间使用
      而临界区只限制与同一进程的各个线程之间使用
    - 只有拥有互斥对象的线程才有访问公共资源的权限
    - 特点：比临界区更节省资源，更效率
- 信号量
    - 通过使用一个计数器来限制可以同时使用某资源的线程数量
    - 特点：允许多个线程同时访问统一资源，但有数目限制
- 事件（信号）机制
    - 允许一个线程在处理完一个任务后，主动唤醒另一个线程执行任务
    - 特点：通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程优先级比较的操作

多进程

2019年7月6日 0:41

- 创建进程的两种方式
  - 操作系统创建
  - 父进程创建
- 创建子进程
  - fork函数
    - pid_t fork(void);
    - 出错返回1，父进程中返回pid>0,子进程中pid==0
- 进程间共享内存的原理

多进程和多线程各自的利弊

2019年7月10日 星期三 00:11

- 再说多线程和多进程各自的适用性
  - 多线程
    - 多线程之间共享一个进程的地址空间
    - 线程间通信简单，同步复杂
    - 线程创建，销毁和切换简单，速度快，占用内存少
    - 适用于多核分布式系统
    - 但是线程间会相互影响，一个线程意外终止会导致统一进程的其他线程页终止，可靠性弱
  - 多进程
    - 多进程间拥有各自独立运行的地址空间，进程间不会相互影响，可靠性强
    - 但是进程创建，销毁和切换复杂，速度慢，占用内存多，进程间同心复杂，但是同步简单
    - 适用于多核，多机分布

进程间通信的方式

2019年6月30日 0:15

- 管道
  - 无名（匿名）（普通）管道：
    - 数据只能在一个方向上流动，具有固定的读端和写端
    - 只能用于具有亲缘关系的进程之间的通信（父子或兄弟）（无名、有名管道的最主要区别）
    - 是一种特殊的文件（只能存在于内存中）
  - 有名（命名）管道
    - 是FIFO文件，存在于文件系统中，可以通过文件路径名来指出
    - 可以在无关的进程之间交换数据
- 消息队列
  - 相当于是一个存放消息的容器
  - 通过异步处理提高系统性能，降低系统耦合性
  - 需要消息时从消息队列取消息
- 信号量
  - 与多线程同步的信号量类似
  - 用于实现进程间的互斥与同步，而不是用于存储进程间通信数据

线程进程操作主要接口

2019年7月6日　　0:56

- fork()和pthread_create()

  负责创建。调用fork()后返回两次，一次标识主进程一次标识子进程；调用pthread_create()后得到一个可以独立执行的线程。

- wait()和pthread_join()

  负责回收。调用wait()后父进程阻塞；调用pthread_join()后主线程阻塞。

- exit()和pthread_exit()

  负责退出。调用exit()后调用进程退出，控制权交给系统；调用pthread_exit()后线程退出，控制权交给主线程。

屏幕剪辑的捕获时间: 2019/7/6 0:57

*死锁（待补充详细死锁例子）

2019年7月7日 星期日 23:36

- 定义：两个或两个以上的进程因争夺资源而造成的相互等待的情况
- 四个必要条件：
  - 互斥条件
    - 进程所分配到的资源部允许其他进程同时访问，只能等待
  - 请求和保持条件
    - 进程获得一定的资源后，又对其他资源发出请求，但是该资源可能被其他进程占有，此时请求阻塞，但该进程不会释放自己已经占有的资源
  - 环路等待条件
    - 发生死锁后，必然存在一个进程-资源之间的环形链
  - 不可剥夺条件
    - 进程已经获得的资源，在未完成使用之前，不可被剥夺，只能在使用后自己释放
- 如何解决
  - 破坏上述四个条件之一
    - 资源一次性分配，破坏请求与保持条件
    - 可剥夺资源
      - 当进程新的资源未得到满足时，释放已占有的资源
    - 资源有序分配法（破坏环路等待条件）
      - 系统给每类资源赋予一个序号，每个进程按编号递增的请求资源，释放则相反

网络I/O模型（四种）

2019年7月6日　　0:38

- 阻塞式
  - 表示一旦调用I/O函数必须等到整个I/O完成才能返回
  - 效率太低
- 非阻塞I/O
  - 调用I/O函数之后会立即返回操作权限（与阻塞式的最大不同）
  - 但实际上用户进程需要不停的访问kernel是否准备好，所以整体效率依旧非常低
- I/O多路复用（事件驱动模型）
  - 通过记录I/O流的状态来同时管理多个I/O，提高服务器的吞吐能力
  - select函数
    - 缺点：两次拷贝耗时，轮询所有fd耗时，连接数太小（1024）
    - 优点：跨平台
  - poll函数
    - 缺点：大量拷贝，重复报告耗时
    - 优点：连接数没有限制
  - epoll函数
    - 没有连接数限制
    - 只有活跃可用的fd才会调用callback函数
    - 内存拷贝是利用文件映射的方式，减少了复制的开销
    - 但只有存在大量的空闲连接和不活跃连接的时候，epoll的效率才会比select和poll高
  - 三个函数的形象比喻

> 1. **阻塞IO**, 给女神发一条短信, 说我来找你了, 然后就默默的一直等着女神下楼, 这个期间除了等待你不会做其他事情, 属于备胎做法.

> 2. **非阻塞IO**, 给女神发短信, 如果不回, 接着再发, 一直发到女神下楼, 这个期间你除了发短信等待不会做其他事情, 属于专一做法.

> 3. **IO多路复用**, 是找一个宿管大妈来帮你监视下楼的女生, 这个期间你可以些其他的事情. 例如可以顺便看看其他妹子,玩玩王者荣耀, 上个厕所等等. IO复用又包括 select, poll, epoll 模式. 那么它们的区别是什么?
> 3.1 **select大妈** 每一个女生下楼, select大妈都不知道这个是不是你的女神, 她需要一个一个询问, 并且select大妈能力还有限, 最多一次帮你监视1024个妹子
> 3.2 **poll大妈**不限制盯着女生的数量, 只要是经过宿舍楼门口的女生, 都会帮你去问是不是你女神
> 3.3 **epoll大妈**不限制盯着女生的数量, 并且也不需要一个一个去问. 那么如何做呢? epoll大妈会为每个进宿舍楼的女生脸上贴上一个大字条,上面写上女生自己的名字, 只要女生下楼了, epoll大妈就知道这个是不是你女神了, 然后大妈再通知你.

    - 三个函数的区别
        - 操作方式及效率
            - select和poll都是遍历，O(n)
            - epoll是回调，O(1)
        - 最大连接数
Select	1024/2048
            - poll，epoll无上限
        - fd拷贝
            - select和poll每次都需要把fd集合从用户态拷贝到内核态
            - epoll通过mmap映射共享同一块存储，避免了fd从内核赋值到用户空间
- 异步I/O模型
    - 当用户进程发起I/O请求后立即返回，知道内核发送一个信号，告知进程I/O已完成
      在整个过程中，都没有进程被阻塞
    - 告诉女神我来了，然后你就去打LOL了，一直到女神下楼发现找不到你了，女神打电话通知你，你才来到宿舍门口找女神

内存管理

2019年7月7日 星期日 23:46

*（待补充）页表寻址

2019年7月10日 星期三 00:10

*（待补充）缺页中断

2019年7月10日 星期三 00:10

虚拟内存

2019年7月8日 星期一 12:48

- 目的
  - 为了让物理内存扩充为更大的逻辑内存，从而让程序获得更多的可用内存
- 原理
  - 操作系统将内存抽象成地址空间
  - 每个程序拥有自己独立的地址空间，这个地址空间被分割成多个块，每一块称为一个页
  - 这些页被映射到物理内存，但不需要映射到连续的物理内存，页不需要所有页都必须在物理内存中
  - 也就是允许程序不需要全部调入内存就可以运行，使得有限的内存运城大程序成为可能
- 虚拟内存的重要意义
  - 定义了一个连续的虚拟地址空间，使得程序的编写难度降低
  - 把内存扩展到硬盘空间只是使用虚拟内存的必然结果
    虚拟内存空间会存在硬盘中，并且会被内存缓存
- 虚拟内存的好处
  - 缓存：把主存看做是一个存储在硬盘上的虚拟地址空间的高速缓存，并且只在主存中缓存活动区域
  - 内存管理：为每个进程提供了一个一致的地址空间，从而降低了程序员对内存管理的复杂性
  - 内存保护：保护了每个进程的地址空间不会被其他进程破坏
- 分段和分页的区别
  - 虚拟内存使用的是分页，也就是将地址空间划分成固定大小的页，每一页再与内存进行映射
  - 分段的做法是把每个表分成段，一个段构成一个独立的地址空间
    - 每个段的长度可以不同，并且可以动态增长
  - 段页式
    - 将程序的地址空间划分陈多个拥有独立地址空间的段，每个段上的地址划分成大小相同的页
    - 这样既拥有分段系统的共享和保护，又拥有分页系统的虚拟内存功能
  - 分段和分页的比较
    - 对程序员的透明性
      - 分页透明
      - 分段需要程序员划分每个段
    - 地址空间的维度
      - 分页是一维
      - 分段是二维

- 大小是否可变
  - 页的大小不可变
  - 段的大小可以动态改变
- 出现原因
  - 分页主要是为了实现虚拟内存，从而获得更大的地址空间
  - 分段是为了使程序和数据可以被划分为逻辑上独立的地址空间，并且有助于共享和保护

页面置换算法

2019年7月7日 星期日 23:55

- 分页系统地址映射
  - 内存管理单元（MMU）管理着地址空间和物理内存的转换
  - 页表：存储着页（程序地址空间）和页框（物理内存空间）的映射表
- 页面置换算法
  - 问题
    - 程序运行过程中，如果要访问的页不在内存中，就会发生缺页中断，从而将该页调入内存中
    - 如果此时内存已经没有空闲空间，系统就必须从内存中调出一个页面到磁盘兑换区中来腾出空间
  - 页面置换算法定义
    - 可以将内存看成磁盘的缓存
    - 在缓存系统中，缓存的大小有限，当有新的缓存到达时，需要淘汰一部分已经存在的缓存，这样才有空间存放新的缓存数据
  - 页面置换算法目的
    - 使页面置换频率最低（缺页率最低）
- 六种页面置换算法
  - 最佳（OPT）
    - 选择的被换出的页面将是最长时间内不再被访问，通常可以保证获得最低的缺页率
    - 只是一种理论上的算法，因为无法知道一个页面多长时间不再被访问
  - 最近最久未使用（LRU）
    - 将过去最近最久未使用的页面换出
    - 需要在内存中维护一个包含所有页面的链表
    - 当一个页面被访问时，将这个页面移到链表表头，这样就能保证链表表尾的页面是最近最久未使用的
    - 但是因为每次访问都需要更新链表，所以这种方式代价很高
  - 最近未使用（NRU）
    - 每个页面设置两个状态位：R和M
    - 当页面被访问时设置页面R为1，被修改时设置M为1
    - R位会定时被清零
    - NRU会有限换出已经被修改的脏页面（R为0，M为1），而不是被频繁使用的干净页面（R为1，M为0）
  - 先进先出（FIFO）
    - 选择换出的页面是最先进入的页面
    - FIFO会将那些经常被访问的页面也被换出，从而使缺页率升高
  - 第二次机会算法

        - 解决FIFO的缺点
        - 当页面被访问（读写）时设置该页面R位为1，需要替换的时候，检查最老页面的R位
        -

如果R为0	那么这个页面既老，而且有没有被使用	可以立刻被置换掉
如果R为1	就将R位清0，并把该页面放到链表的尾端，修改它的装入时间使得它就像刚装入的一样	继续从链表的头部开始搜索

    - 时钟（Clock）
        - 第二次机会算法需要的链表中移动页面，降低了效率
        - 时钟算法使用环形链表将页面链接起来，使用一个指针指向最老的页面

（补充）虚拟地址空间

2019年7月6日　0:41

- 为什么虚拟空间
  - 防止不同的进程同时在物理内存中运行，而对物理内存的争夺和践踏
- 原理
  - 让不同进程在运行过程中时，它所看到的是，自己独自占有了当前系统的4G内存
- 虚拟内存的重要意义
  - 定义了一个连续的虚拟地址空间，使得程序的编写难度降低
  - 把内存扩展到硬盘空间只是使用虚拟内存的必然结果
    虚拟内存空间会存在硬盘中，并且会被内存缓存
- 虚拟内存的好处
  - 缓存：把主存看做是一个存储在硬盘上的虚拟地址空间的高速缓存，并且只在主存中缓存活动区域
  - 内存管理：为每个进程提供了一个一致的地址空间，从而降低了程序员对内存管理的复杂性
  - 内存保护：保护了每个进程的地址空间不会被其他进程破坏
- 虚拟内存的实现（没看懂呢）

> 虚拟内存是对内存的一个抽象。支持虚拟内存的CPU需要通过虚拟寻址的方式来引用内存中的数据。CPU加载一个虚拟地址，然后发送给MMU进行地址翻译。地址翻译需要硬件与操作系统之间紧密合作，MMU借助页表来获得物理地址。
>
> - 首先，MMU先将虚拟地址发送给TLB以获得PTE（根据VPN寻址）。
> - 如果恰好TLB中缓存了该PTE，那么就返回给MMU，否则MMU需要从高速缓存/内存中获得PTE，然后更新缓存到TLB。
> - MMU获得了PTE，就可以从PTE中获得对应的PPN，然后结合VPO构造出物理地址。
> - 如果在PTE中发现该虚拟页没有缓存在内存，那么会触发一个缺页异常。缺页异常处理程序会把虚拟页缓存进物理内存，并更新PTE。异常处理程序返回后，CPU会重新加载这个虚拟地址，并进行翻译。

屏幕剪辑的捕获时间: 2019/7/6 23:23

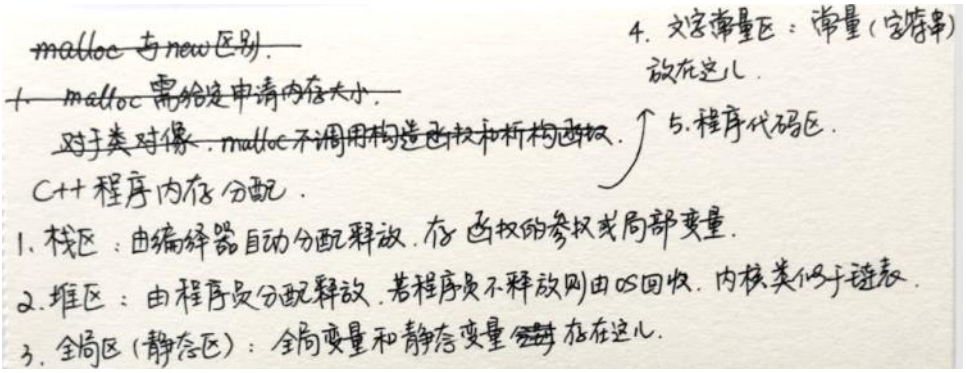
分段和分页

2019年7月16日 星期二 21:57

- 分段
  - 什么是分段
    - 就是把虚拟地址空间中的虚拟内存组织成一些长度可变的称为段的内存块单元
  - 什么是段
    - 段的定义有三个参数
      - 段基地址，段限长，段属性
    - 着三个参数被存储在一个称为段描述符的结构项中
  - 段的作用
    - 段可以用来存放程序的代码，数据，堆栈
    - 存放系统数据结构
  - 段的存储地址
    - 系统中所有使用的段都包含在处理器线性地址空间中
- 分页
  - 什么是分页
    - 分页在分段之后进行
    - 进一步将线性地址转化为物理地址
  - 分页机制
    - 当使用分页时，每个段被划分为若干个页面，页面会被存储在物理内存或者硬盘中
    - 操作系统通过维护一个页目录和一些页表来存储这些页面
    - 当程序想要访问线性地址空间中的一个地址位置时，处理器就会使用页目录和页表把线性地址转换成一个物理地址，然后在这个内存地址上执行操作
- 分页和分段的区别
  - 分页使用固定大小的内存块，分段则是大小可变的内存块（最大区别）
    - 是因为分页是为了操作系统更好的管理内存
    - 分段是为了用户来处理复杂的逻辑分区
  - 地址空间
    - 页的地址是一维的
      - 单一的线性空间
      - 程序员只需要一个地址就可以
    - 段的地址是二维的
      - 程序员既需要给出段名，还需要段内地址

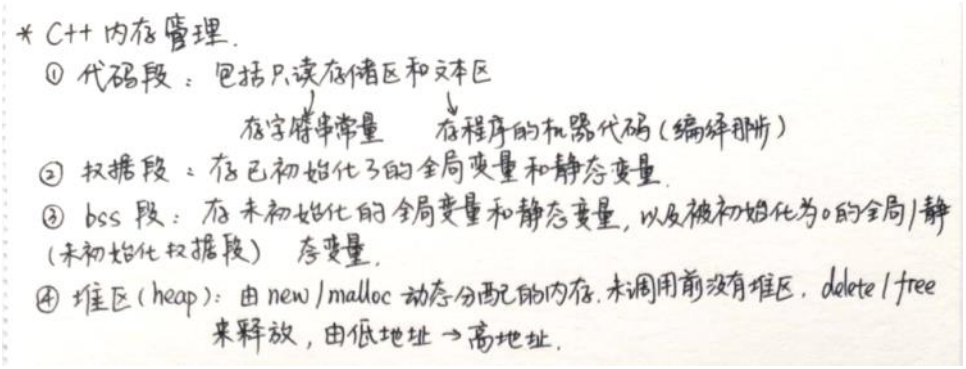
C++程序内存分配

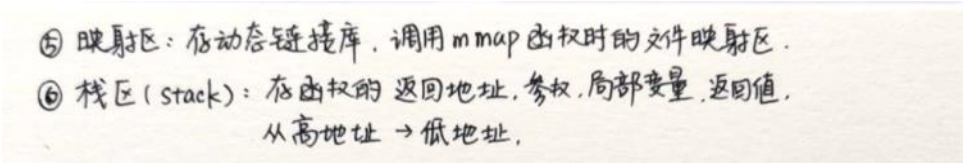
2019年7月8日 星期一 14:00



C++程序内存管理

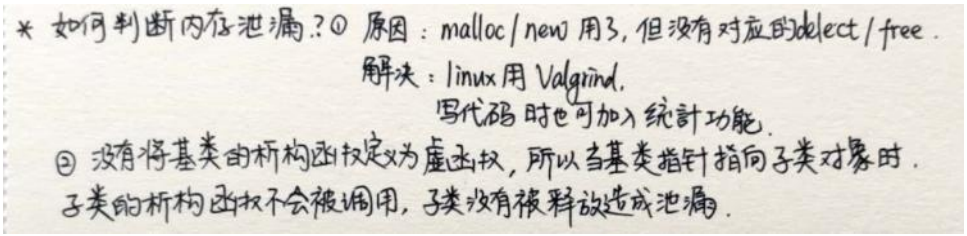
2019年7月8日 星期一　　14:02





C++如何判断内存泄漏

2019年7月8日 星期一 14:03



链接

2019年7月7日 星期日　　23:57

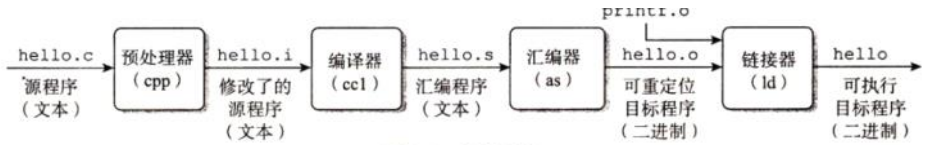
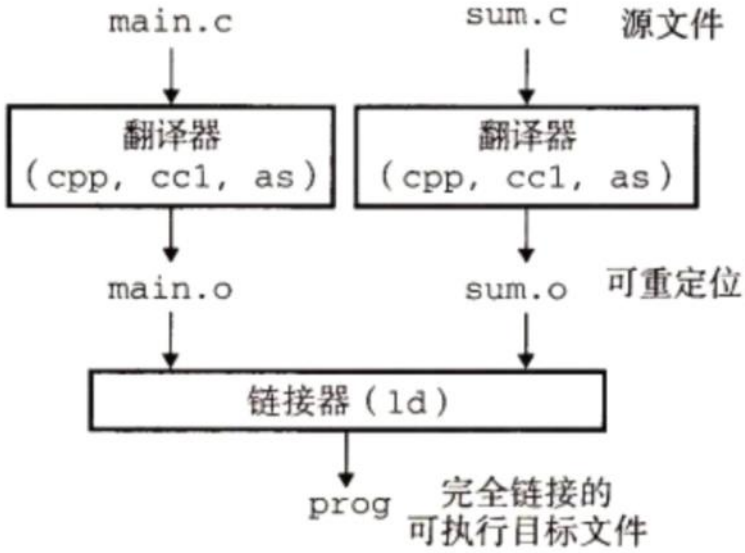
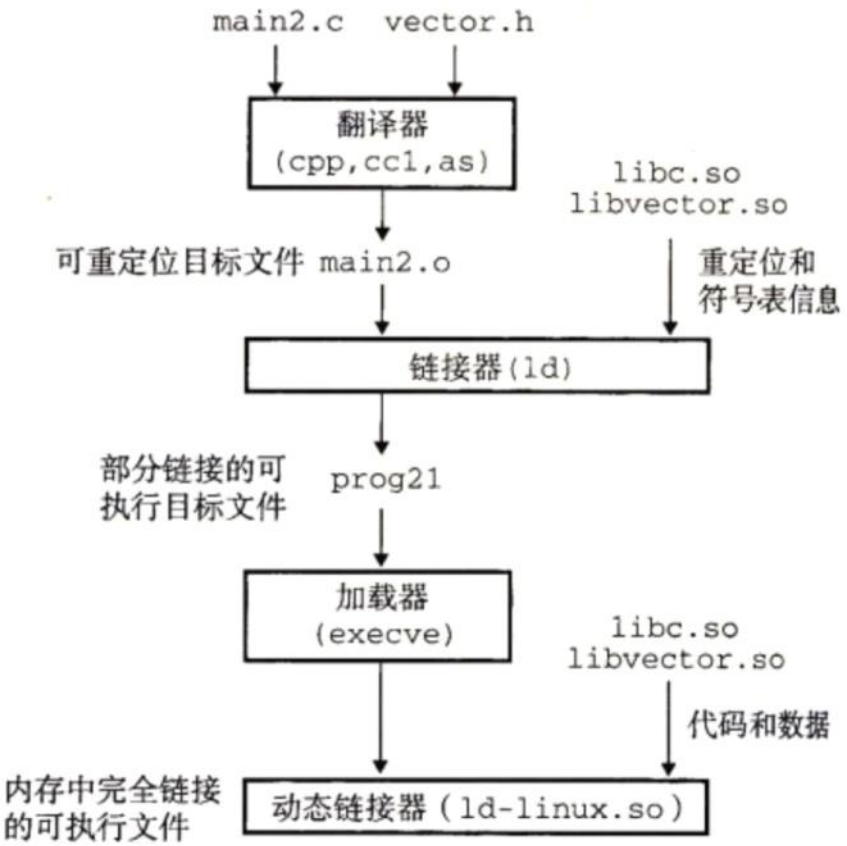
- 编译系统
  - .c程序的编译过程
    - 

      图 1-3　编译系统
      - 预处理阶段
        - 处理#开头的预处理命令（头文件）
      - 编译阶段
        - 翻译成汇编文件
      - 汇编阶段
        - 将汇编文件翻译成可重定位目标文件
      - 链接阶段
        - 将可重定位目标文件和printf.o等预先单独编译好的目标文件进行合并
        - 得到最终的可执行目标文件
- 静态链接
  - 静态链接器
    - 输入：一组可重定位目标文件
    - 输出：一个完全链接的可执行目标文件
    - 干了什么?
      - 符号解析
        - 每个符号对应一个函数，一个全局变量或者一个静态变量
        - 目的：将每个符号引用与一个符号定义关联起来
      - 重定位
        - 通过把每个符号定义与一个内存位置关联起来，然后修改所有对这些符号的引用，使得他们指向这个内存位置



- 目标文件
  - 可执行目标文件
    - 可以直接在内存中执行
  - 可重定位目标文件
    - 可与其他可重定位目标文件在链接阶段合并，创建一个可执行目标文件
  - 共享目标文件
    - 一种特殊的可重定位目标文件，可以在运行时被动态加载进内存并链接
- 动态链接
  - 静态链接的问题
    - 静态库更新时，整个程序都要重新进行链接
    - 对于printf这种标准函数库，如果每个程序都要有代码，会极大浪费资源
  - 共享库
    - 为了解决静态库的问题
    - Linux下用.so后缀
      Windows下称为DLL
    - 特点
      - 给定的文件系统中一个库只有一个文件，所有引用该库的可执行目标文件都共享这个文件，它不会被复制到引用它的可执行文件中
      - 在内存中，一个共享库的.text节（已编译程序的机器代码）的一个副本可以被不同的不同的正在运行的进程共享


main2.c
vector.h
翻译器
(cpp,cc1,as)
libc.so
libvector.so
可重定位目标文件 main2.o
重定位和
符号表信息
链接器(ld)
部分链接的可
执行目标文件
prog21
加载器
(execve)
libc.so
libvector.so
代码和数据
内存中完全链接
的可执行文件
动态链接器（ld-linux.so）