Modern Operation Systems

现代操作系统这本书是本科同班的刘同学介绍的，当时我刚看完 CSAPP，发了个朋友圈，然后他就说《现代操作系统》这本书更加好。因为好奇，所以我就上了贼船^<^。

参考机：Intel Core i7 Haswell

一、 引论

1.1 操作系统

图 1-1 操作系统所处的位置

1) 多数计算机都有两种运行模式——内核态和用户态，在内核态的运行模式下，用户拥有对所有硬件的完全访问权，可以执行所有的机器指令，而在用户态下，所能执行的机器指令的子集。
2) 操作系统与其他软件不同的区别在于用户不能对操作系统进行修改，因为操作系统是由硬件保护的。
3) 什么是操作系统？
   - 作为拓展机器的操作系统
     文件其实是使用硬盘的一层抽象，直接深入理解和管理硬盘信息是非常困难的，但是我们有一个抽象——接口，或者叫硬盘驱动 disk driver。操作系统一个重要的任务是隐藏硬件，为用户提供给一个优雅，高效，清晰，一致的抽象。
   - 作为资源管理者的操作系统
     多路复用，包括时间复用和空间复用，合理有序地为各个程序分配资源。
4) 操作系统的历史
   - 第一台真正的数字计算机是英国数学家 Charles Babbage(1792-1871)穷其一生设计的，限于当时的生产能力和精度，而且它没有操作系统，最后也没有运行起来。有一个花絮时，当 Charles Babbage 意识到需要软件的时候，他雇佣了 Ada Lovelace 的年轻妇女，这是世界上第一个程序员^<

^。

- 第一代操作系统（1945-1955）——真空管和穿孔卡片
  所有的程序都是用机器语言编写的，需要将上千根电缆连接到插件板上形成电路。但是只能进行诸如正弦，余弦，对数和计算炮弹轨迹等简单的计算。
- 第二代操作系统（1955-1965）——晶体管和批处理系统
  主要用于科学与工程计算，典型的操作系统是 FMS，Fortran Monitor System，使用的还是穿孔卡片，也怪不得 fortran77 代码每一行前面会有严格的空格数量。
- 第三代操作系统（1965-1980）——集成电路和多道的程序设计
  分时系统 timesharing
- 第四代操作系统（1980-至今）——个人计算机
- 第五代（1990-至今）——移动计算机

5) 硬件简介
   大部分的硬件设备都需要配备相应的控制器才能与总线相连，总线在我看来就用来使得不同的设备进行通讯。
6) 后面还有很多东西，估计后面几章都会讲到，这里就不详细叙述了。

二、 进程与线程

2.1 进程 process 与线程 thread

1) 操作系统最核心的概念——进程
   传统操作系统中，每个进程有一个地址空间和一个控制线程，进程与进程之间的地址空间是不同的。而对于线程，同一个地址空间则允许运行多个线程。进程用于把资源集中在一起，共享物理内存，磁盘，打印机以及其他资源；而线程则是在 CPU 尚被调度执行的实体，共享同一个地址空间和其他的资源，在 CPU 密集型的多线程进程中，每个线程得到了真实 CPU 速度的三分之一。
2) 严格来讲，在某一个瞬间，CPU 只能运行一个进程，但在一秒内，他可能运行多个进程，让人产生并行的错觉——伪并行。而多核处理器实现的是真正的硬件并行。
3) 进程模型
   每个进程都拥有各自的程序
4) 进程的创建
   有四种主要事件会导致进程的创建
   - 系统的初始化
   - 正在运行的程序执行了创建进程的系统调用
   - 用户请求创建一个新的进程
   - 一个批处理作业的初始化

   从技术上看，新进程都是由一个已存在的进程执行了一个用于创建进程的系

统调用而创建的。比如在 UNIX 系统中用 fork，创建子进程，父进程和子进程拥有相同的内存映像，写时复制技术 copy-on-write 以保护私有内存区域。

5) 守护进程

停留在后台处理的诸如电子邮件，Web 页面，新闻，打印之类的程序进行一个

6) 进程的终止

有四种主要事件会导致进行的终止

- 正常退出（自愿的）

  在 UNIX 中是 exit，在 Windows 中是 ExitProcess
- 出错退出（自愿的，出现的错误是在计划中的）
- 严重错误（非自愿的）
- 被其他进程杀死（非自愿的）

  在 UNIX 中是 kill，在 Windows 中是 TerminateProcess。在某些系统中，父进程被杀死时，其创建的进程也会被杀死，但是 UNIX 和 Windows 都不是这样。

7) 进程的层次结构

在 UNIX 中，父进程和它所有的子进程以及后裔共同组成一个进程组，进程最多只有一个父进程。那最初的那个进程是如何产生的呢？以 UNIX 为例，在 UNIX 启动的时候，一个称为 init 的特殊进程出现在启动映像中，当他开始运行的时候，读入一个说明终端数量的文件。接着，为每个终端创建一个新进程。这些进程等待用户的登陆，如果用户登录成功，改登陆进程就会执行一个 shell 准备接收命令。所接收的命令会启动更多的进程，以此类推，所有的进程属于以 init 为根的一棵树。

在 Windows 中，相反，没有进程层次的概念，所有进程的地位都是相同的。唯一类似于进程结构的暗示是在创建进程的时候，父进程得到一个特殊的令牌，或者叫句柄，可以用来控制子进程。但是，父进程也有权可以把这个特殊的令牌传给其他进程使其获得子进程的控制权。

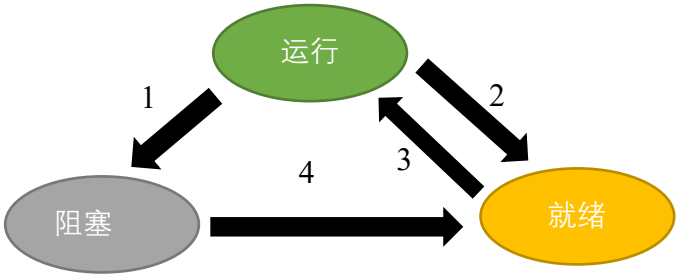
8) 进程的状态

- 运行态

  该进程实际占用 CPU；
- 就绪态

  可运行，但是由于其他进程正在运行而暂时停止；
- 阻塞态

  除非某种特殊外部事件发生，否则进程不能运行，如进程等待输入；



1：进程由于等待输入而被阻塞；
2：调度程序选择另外的进程；
3：调度程序选择该进程；
4：出现有效输入；

图 1-2 进程的三态

9) 进程模型

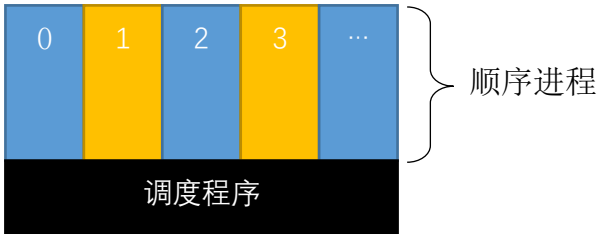


图 1-3 基于进程的操作系统最简单的模型

操作系统最底层的是调度程序，所有关于中断处理，启动进程和停止进程的具体细节都隐藏在调度程序中。不过，图 1-3 是很理想的情况，真实的操作系统比这复杂得多。

10) 多道程序设计模型

使用多道程序设计模型可以提高 CPU 的利用率，假设进程等待 I/O 输入的时间和在内存中运行的时间之比为 p，在内存中同时有 n 个进程（n 又叫做多道程序涉及的道数 degree of multiprogramming）那么，

$$程序利用率 = 1 - p^n$$

11) 线程

而对于线程，同一个地址空间则允许运行多个线程。

人们需要线程的原因：

- 在许多应用中同时允许同时发生多个活动，其中某些活动随着时间被阻塞后，由于同一个进程的多个线程是共享地址空间和数据的，所以另外一个活动可以继续进行，而不必考虑中断，定时器和上下文切换；
- 线程比进程更加轻量级，他们比进程更加容易创建，销毁和切换，创建一个线程比创建一个进程要快 10~100 倍；
- 涉及到性能方面的讨论。若多个线程都是 CPU 密集型的，那么并不能获得性能上的提高，如果存在大量的计算和大量的 I/O 处理，多个线程可以重叠进行，从而加快应用程序的执行速度。

12) 有限状态机 finite-state machine

每个计算都有一个被保存的状态，存在一个会发生且使得相关状态发生改变的事件集合。

在服务器中的应用：当网页请求到来的时候，有限状态机唯一的线程会考察这个请求，假如这个请求的网页在缓存中，这倒还好，假如需要读取磁盘的信息，则线程会保存目前的状态和数据，同时磁盘会进行读取，线程则会回应其他的请求，当其他的请求是新工作，则开始新的工作，如果其他的请求是新工作是之前磁盘的回答，则处理该回答，而则这种回答多数以信号或者中断的形式出现。每次服务器为某个请求工作的状态切换到另外一个状态的

时候，都比西显式的保存或者重新装入相应的计算状态。
其实有限状态机在我看来就是“贫穷家庭所做出的最大的努力”

表 1-1 构造服务器的三种方法

模型	特性
多线程	并行性，阻塞系统调用
单线程进程	无并行性，阻塞系统调用
有限状态机	并行性，非阻塞系统调用，中断

13) 经典的线程模型

在多线程系统中，同一个进程中的多个线程类似于多个进程，在运行过程中需要来回切换，由于同一个进程中的多个线程之间的切换速度非常快，所以接近“每个线程得到了真实 CPU 速度的几分之一”

每个线程可以处于若干种状态的任何一个：运行，阻塞，就绪或者终止。虽然同一个进程中的多个线程都共享同一个程序计数器，寄存器和地址空间，但是每个都有自己的堆栈，供各个被调用但是还没有从中返回的过程使用，记录各自不同的执行历史。在该堆栈中存放着相应过程中的局部变量以及调用完成之后的返回地址。

14) POSIX 线程

为了实现可移植的线程程序，IEEE 定义了线程的标准，定义的线程包叫做 pthread，pthread 包拥有 60 多个函数调用。

表 1-2 一些 pthread 函数的说明

模型	特性
pthread_create	创建一个新线程
pthread_exit	结束调用的线程
pthread_join	等待一个特定的线程退出
pthread_yield	释放 CPU 来运行一个新线程
pthread_attr_init	创建并初始化一个线程的属性结构
pthread_attr_destory	删除一个线程的属性结构

15) 在用户空间中实现线程

- 在用户空间中实现线程包

  一个最明显的优点是，用户级线程包可以在不支持线程的操作系统中实现。

- 在内核中实现线程包

表 1-3 一些多线程实现的方法

类型	优点	缺点
用户级线程（线程表在用户空间）	用户级线程包可以在不支持线程的操作系统中实现	如何实现阻塞系统的调用，一种可行的替代方案是如果需要阻塞的话就提前通知，在系统调用周围从事检查的这类代码称作“包装器”
	不需要陷入内核，不需要上下文切换，也不需要对内存高速缓存进行刷新，使得线程调度非常快捷	如果一个线程开始运行，该进程中的其他线程则不可能运行，除非第一个线程主动放弃 CPU 的使用，因为用户空间中没有时钟的中断，那自然也没有轮转调度
	允许每个进程有自己定制的调度算法	一旦发生内核陷阱，很难在已经阻塞的线程中进行进程的切换
内核级线程（线程表在内核空间）	所有能够阻塞线程的调用都是以系统调用的形式实现	开销比较大
		当一个多线程进程创建一个新的进程的时候，它到底需要复制所有的线程吗？这是一个问题
		当一个多线程进程收到信号的时候，到底哪个线程收到信号并做出处理呢？（注：信号是发给进程而不是线程的）
混合使用	兼并用户级线程和内核线程的好处，每个内核线程都对应着一个用户级线程集，可以对多个用户级线程进行多路复用	
调度程序激活机制		
弹出式线程	当有消息到达的时候才创建处理该消息的新的线程	如果某个线程运行的时间过长，有没有办法抢占它，就可能造成进来的消息丢失
使单线程代码多线程化		全局变量的使用会很麻烦

2.2 在进程间通讯 Inter Process Communication，IPC

1) 需要解决的问题
   - 一个进程如何把信息传递到另一个
   - 确保两个或者多个进程在关键活动中不会出现交叉
   - 确保正确的顺序
2) 竞争条件

   两个或多个进程读写某些共享的数据，而最后的结果取决于进程运行时的精确时序，称为“竞争条件”
3) 临界区域/临界区

   共享内存进行访问的程序片段称为临界区域 critical region/临界区 critical section，如果有办法可以避免两个进程同时处于临界区，则可以避免进程的竞争。这样的解决方案需要满足四个条件：
   - 任何两个进程不能同时处于临界区，即互斥 mutual exclusion；
   - 不应对 CPU 的速度和数量作任何的假设；
   - 临界区外的进程运行的进程不得阻塞其他进程；
   - 不得使进程无限期等待进入临界区；
4) 忙等待方案
   - 屏蔽中断

     包括屏蔽时钟中断，因为 CPU 在发生时钟中断的时候才会切换进程。如果时钟中断屏蔽了，意味着进程不能被切换，这就保证临界区在一个时间段最多只能被一个进程修改。但是把这么重要的权利交给用户紧张明显是不明智的；
   - 锁变量

     初始值为 0，当一个进程想要进入临界区时，先要检查一下所变量是否为 0，若为 0，则进入临界区并把锁变量设置为 1。若为 1，则等待直至锁变量变成 0。

     但是如果另外一个程序在第一个程序检查锁变量为 0 后并在修改前就进入了，同样会存在竞争的问题
   - 严格轮换法

     忙等待 busy waiting，连续测试一个变量直到某个值出现为止，由于这种方式浪费 CPU 的时间，所以通常应该避免，用于忙等待的锁，叫做自旋锁 spin lock，自旋锁的值为进入临界区进程的值。如果需要进入临界区的进程由于忙于做其他的事情迟迟不进入临界区并修改自旋锁，则下一个进程只好一直忙等待下去，造成不必要的开销。

     当一个进程比另一个进程慢许多的时候，轮流进入并不是一个好的办法。
   - Peterson 解法

     需要调用 enter_region 和 leave_region 两个函数，其实就是要检查进去和离开的两个值。用 turn 表示轮到那个进程准备进入临界区，用 interest[process]表示锁，另一个进程进入的时候检查前一个进入的进程的 interest 值是否已经解锁，如果解锁了，就进入，否则就不断循环检查——忙等待，

- TSL 指令 Test and Set Lock
  通过硬件锁住总线
  事实上，在处理器 1 上屏蔽中断对处理器 2 根本没有任何的影响。唯有锁住内存总线才是王道。同样需要调用 enter_region 和 leave_region 两个函数。
  使用 TSL 指令或者 XCHG 指令可以确保同一时刻只有一个 CPU 在对信号量进行操作；

5) 唤醒与睡眠
忙等待方案都会有很多的问题，其中包括优先级反转问题 priority inversion problem
生产者 producer-消费者 consumer 问题，其中生产者将信息放进缓存区，消费者读取缓存中的信息，而缓存区中的项数设置为 count。生产者代码检查 count 值，若 count 值等于 N，则睡眠，否则生产者在缓存区中放进一个数据项并将 count 值+1；消费者代码检查 count 值，若 count 值等于 0，则睡眠，否则消费者在缓存区中取走一个数据项并将 count 值-1；
为了避免 wakeup 信号丢失（那什么情况下 wakeup 信号会丢失呢？当进程清醒的时候接收到 wakeup 信号时，wakeup 信号会不起作用，即 wakeup 信号丢失）导致准备睡眠的进程后面没有及时被唤醒，需要增加一个唤醒等待位，即 wakeup 信号的“小仓库”。

6) 信号量 semaphore——累计唤醒次数
原子操作——一组操作要么不间断地执行，要么都不执行。
信号量是 E. W. Dijkstra 在 1965 年时提出来的方法，他建议设立两种操作：up 和 down。其中 up 操作是对信号量+1，而 down 操作则会检查信号量的值是否大于 0，若大于 0，则减 1，若为 0，则程序睡眠。
用信号量解决生产者-消费者问题，关键在于执行原子操作。通常将 up 和 down 作为系统调用实现，而且操作系统在执行一下操作的时候屏蔽全部中断：测试信号量，更新信号量以及在需要时使某个进程睡眠。这些操作只需要几行指令，所以屏蔽也不会带来什么副作用。如果使用多个 CPU，可以加入 TSL 指令或者 XCHG 指令。
解决方案：三个信号量：一个是 full，记录充满的缓冲槽数目，初值为 0；一个是 empty，记录空的缓冲槽数目，初值为缓冲区槽的数目；一个是 mutex，用来确保生产者和消费者不会同时访问缓冲区，初值为 1，保证同时只有一个进程可以进入临界区，称作“二元信号量 binary semaphore”。每个进程在进入临界区的时候执行 down 的操作，离开的时候执行 up 的操作，就可以实现互斥。

7) 互斥量 mutex
互斥量是一个可以处于两态之一的变量：解锁和加锁，0 表示解锁，其他所有的值表示加锁。
访问临界区：mutex_lock
离开临界区：mutex_unlock，如果多个线程被阻塞，则随机选取一个线程并允许它获得锁。
enter_region 和 mutex_lock 的区别是后者解锁失败的时候，他会调用

thread_yield 将 CPU 放弃给另一个线程，这样就没有忙等待。

8) 忙等待方案中大的锁变量问题在于：不同的进程需要共享一个锁变量，这里有两种方案，第一种就是共享数据结构，如信号量，可以存放在内核中，并且只有系统调用来访问。第二种就是现代操作系统 Linux 和 Windows 提供的方法，让进程共享一小部分的地址空间，如缓冲区和锁变量。
9) 死锁 dead lock
10) 管程需要 condition 变量的原因本质上就是程序执行顺序的不确定性.管程(monitor)只是保证了同一时刻只有一个进程在管程内活动,即管程内定义的操作在同一时刻只被一个进程调用(由编译器实现).但是这样并不能保证进程以设计的顺序执行,因此需要设置 condition 变量,让进入管程而无法继续执行的进程阻塞自己.

作者：钓雪

链接：https://www.zhihu.com/question/30641734/answer/105402533

来源：知乎

管程

一个管程（monitor）是一个由过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或软件包。

进程可在任何需要的时候调用管程中的过程，但他们不能在管程之外声明的过程中直接访问管程内部的数据结构。

管程是一种语言概念，C 语言不支持

作者：houjian914

链接：http://blog.csdn.net/houjian914/article/details/50762056

来源：博客

11) 消息传递

一个进程调用从一个给定的源接收的一条消息，如果没有消息可用，则接收者可能会被阻塞，直到一条消息到达，或者，带着一个错误码返回。

为了防止信息的丢失，发送者和接受者应该协议一致：一旦接收到消息，接收方马上回送一条特殊的确认 acknowledgement 消息，在一段时间内，发送方仍然没有接收到确认的消息，则重发消息。

那如果发送确认的消息丢失，发送方会重新发送消息，这时候接收者会接收到两条一样的消息，那接收者应该如何区分这两条重复的消息呢？有一个办法是在原始的消息里添加一个序列，防接收者接收到相同相同的消息的时候，检查验证序列，假如序列相同，则说明这是一条重复的消息，并把它丢掉。

12) 屏障

屏障用于一组进程的同步，当一个进程到达屏障的时候，他会被阻塞，直到进程组中其他所有的进程都到达的时候，才取消阻塞。

13) 避免锁——读-复制-更新

其实类似于写时复制

2.3 调度

1 简介

- 切换进程的代价是非常高的；
- 计算密集型进程(compute-bound)
  需要较长时间的 CPU 集中使用和较小频率 I/O 等待
- I/O 密集型进程(I/O-bound)
  需要较少时间的 CPU 集中使用和较大频率 I/O 等待
  由于对 CPU 的改进比对磁盘的改进快得多，所以未来对 I/O 密集型进程(I/O-bound)的调度显得更为重要。这里的基本思想是：如果需要运行 I/O 密集型进程，那么就应该让它尽快得到机会，以便发出磁盘请求并保持磁盘始终忙碌。

2 调度算法的分类

- 非抢占型调度算法
  在时钟中断的时候不会进行调度，在处理完时钟中断后，如果没有更高优先级的进程等待时，被中断的进程将继续运行。
- 抢占型调度算法
  挑选一个进程，并且让该进程运行某个固定时段的最大值，此时时钟发生中断，如果在该时段结束时，该进程仍在运行，则会被挂起，此时调度程序将会挑选另外的进程运行。

3 三种处理环境以及目标

- 批处理
  应用于商业环境，比如处理薪水册，存货清单，账目收入等，使用非抢占式调度算法或者对每个进程有长时间周期的抢占型调度算法都是可以接受的。
  目标是吞吐量——每小时最大作业数
  周转时间——从提交到终止间的最小时间
  CPU 利用率——保持 CPU 始终忙碌
- 交互式
  为了避免一个进程霸占 CPU 拒绝为其他进程服务，抢占是必须的，比如服务器。
  响应时间：快速相应请求
  均衡性：满足用户的期望
- 实时
  实时系统只运行那些推进现有应用的程序，抢占有时是不必要的，因为进程了解到他们可能长时间得不到运行，就赶快完成自己的工作，然后阻塞。
  满足截止时间：避免丢失数据
  可预测性：在多媒体系统中避免品质降低

4 批处理调系统中的调度

非抢占式的先来先服务（first-come first-served）：最简单，缺点是如果存在许多 I/O 密集型进程则会严重拖慢进度；

非抢占式的最短作业优先（shortest job first）：但是有必要指出的是，只有在所有作业都同时运行的情况下，最短作业优先才是最优化。系统也无法得知那个进程的运行时间最短。所以此算法无解；

抢占式的最短时间优先（shortest remaining time first）：当一个新的作业到达的时候，其整个时间同当前进程的剩余时间做对比，如果新的进程的运行时间比当前运行进程需要更少的时间，则当前的进程会被挂起，而运行新的进程。

5 交互式系统中的调度

- 轮转调度：一种最古老，最简单，最公平而且使用最广的算法，隐含的假设：所有的进程优先级相同。为每个进程分配一个时间片，允许进程在此时间片中运行。如果该进程在运行时被阻塞或者提前结束或者超过时间后，剥夺其 CPU 并分配给另外一个进程；这里面有个问题——时间片的长度分配，如果时间片分配的时间过短，由于进程切换或者叫上下文切换需要花费很多的时间，这样的开销比例就会增大，如果时间片的分配时间过长，此时虽然开销变小了，但是后面的进程相应时间会过长；
- 优先级调度：轮换调度+优先级，优先级需要动态调整或者静态调整，要不然优先级底的进程可能会出现饥饿现象。有一种方法是每经过一个时间片，优先级降一级。另外对于 I/O 密集型进程，可以动态地简单设定其优先级为 1/f，f 为该进程占用时间片的比例。
- 多级调度：轮换调度+优先级+多级时间片。最高优先级拥有最小的时间片数，次级的时间片是上一级时间片的两倍。
- 最短进程优先：需要使用经验加权求得估计时间，同时使用老化技术，最旧的时间权重越少。
- 保证调度
- 彩票调度：加权后看运气
- 公平分享调度

6 实时系统中的调度

硬实时 hard real time，一定要满足截止时间

软实时 soft real time，不希望偶尔错失截止时间，但是可以容忍。

7 策略和机制

调度机制 scheduling mechanism 是内核已经写好的，调度策略 scheduling policy 需要用户决定的，坚持调度机制和调度策略分离的原则

8 线程调度

- 用户级线程调度
- 内核级线程调度

9 哲学家就餐的经典 IPC 问题

三、 内存管理

存储管理器：操作系统中管理分层存储器体系的部分，主要作用是有效管理内存，即记录哪些内存正在使用，哪些内存是空闲的，在进程需要的时候分

配内存，在进程使用完后释放内存。

3.1 一种存储器抽象：地址空间

1) 要使多个应用程序处于内存中并且不会相互影响，需要解决两个问题：保护和重定位
2) 早期经典的动态重定位
   为每个 CPU 配置两个特殊的硬件寄存器——基址寄存器和界限寄存器。程序装内存连续的空闲位置且装载期间无须重定位。当一个进程运行的时候，程序的起始物理地址装载到基址寄存器中，程序的长度装载到界限寄存器中，每次程序运行访存取址的时候，CPU 硬件会把地址发送到内存总线前，自动把基址值加到进程发出的地址值上，然后在与界限寄存器进行比较，如果访问的地址超过了界限，就产生错误并中止访问。
   缺点：每次都需要进行加法和比较运算
3) 交换技术 swapping
   交换技术和虚拟内存都是解决内存超载的通用办法。
   交换技术：使用进程的时候把进程调入内存中，然后把他存回磁盘，空闲进程主要存储在磁盘上。
   交换在内存中产生多个空闲区(hole，也称为空洞)，把这些空洞合成一块的技术称为内存紧缩 memory compaction。
4) 虚拟空间
   32 的逻辑地址，分成两部分。前部分是代表虚拟的页号，后部分代表的是虚拟页偏移量，如果页面是 4KB 的话，那么这个后部分虚拟页偏移量占了 12 位，那么前面就是 32-12=20 位。这 20 位就是页表中所有的页表项的和。就是 2 的 20 次方，也就是 1M 个页表项，如果每个页表项占 4B 的话。那么这个页表就占了 4MB 的空间。一般都会有两级甚至更多的。用来减少页表占的空间。。。。。
   虚拟地址位=虚拟页号 VPN+虚拟偏移量 VON
   一个典型的页框号
   页框号位=高速缓存位+访问位+修改位+保护位+“在不在”位+页框号
   映射到物理地址为：
   物理地址位=物理页号 VPN（页框号）+物理（虚拟）偏移量 VON

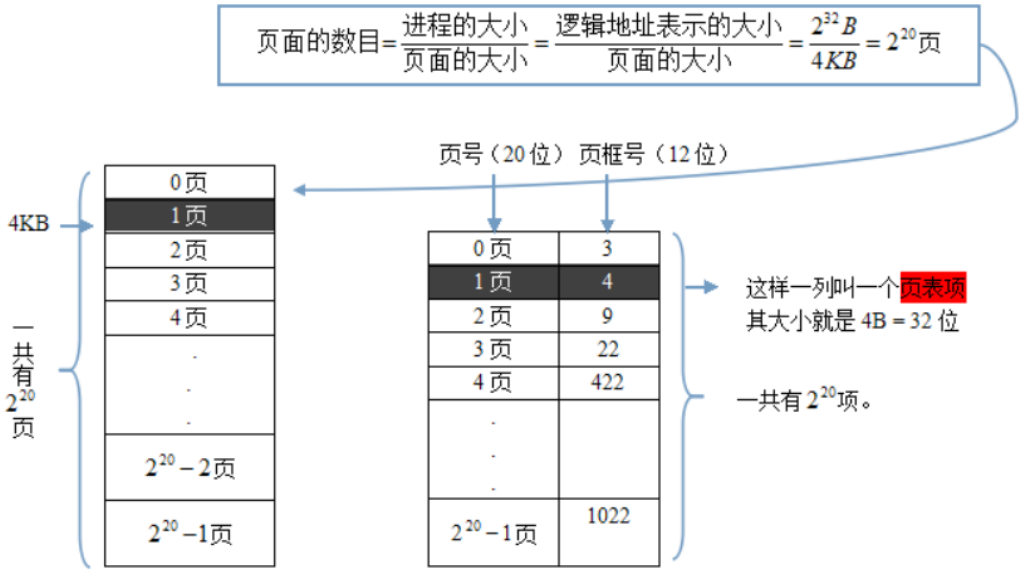
1.

虚拟地址：32 位

页号 P	页内偏移量 W

首先32位的虚拟地址可表示的进程大小应该是2^32B = 4GB（暂时别去想页号P占多少位，W占多少位）

2.（根据页的定义和页面大小的定义）将进程进行分页：



https://www.cnblogs.com/0Nullptr/articles/6958964.html

图 3-1 页，页框，页表的区别

5) TLB 快速分页

TLB，Translation Lookaside buffer 翻译检测缓冲区，又叫做相联存储器 associate memory 或者快表。

工作原理：CPU 将一个虚拟地址放进 MMU 中进行转换时，硬件首先将虚拟地址和 TLB 中所有的表项同时（并行）匹配，如果虚拟页面在其中，而且不违反保护位，则页框号直接从 TLB 中读取而不必再访问页表，如果违反了保护位则触发异常。如果虚拟页面不在其中，则按正常的步骤在内存中查询，接着把 TLB 中的一项替换成新找到的页面

TLB 查询=有效位+虚拟页面号+修改位+保护位+页框号

让人惊奇的是，如果 TLB 大到（如 64 个表项）可以减少失效率，TLB 的软件管理就会变得足够有效，这样做的最大好处是可以获得一个非常简单的 MMU。

软失效 soft miss，不在 TLB 中在内存中，耗时几纳秒

硬失效 hard miss，不在 TLB 中也不在内存中，需要产生磁盘 I/O 耗时几毫秒，如果此时别的进程已经帮你带到内存中，你只需要调用即可，称为次要缺页错误，否则如果要自己亲自动手的话，那就叫做严重缺页错误。

6) 针对大内存的页表

多级页表

倒排页表（不知道这个说的是什么？）

7) 内存映射文件
内存映射文件是由一个文件到一块内存的映射。Win32 提供了允许应用程序把文件映射到一个进程的函数 (CreateFileMapping)。内存映射文件与虚拟内存有些类似，通过内存映射文件可以保留一个地址空间的区域，同时将物理存储器提交给此区域，内存文件映射的物理存储器来自一个已经存在于磁盘上的文件，而且在对该文件进行操作之前必须首先对文件进行映射。使用内存映射文件处理存储于磁盘上的文件时，将不必再对文件执行 I/O 操作，使得内存映射文件在处理大数据量的文件时能起到相当重要的作用。

3.2 页面置换算法

1) 最优页面配置算法
就是从该页面首次被访问开始，N 条指令不被访问后，N 值越大，越先被置换。
该算法唯一的问题是无法实现，因为无法预测下一次发生缺页中断的时间，就无法准确预测 N 值的大小。这跟最作业优先调度算法一样，系统也无法得知那个进程的运行时间最短。
2) NRU 最近未使用页面置换算法 Not Recently Used,
首先系统为每个页面位设置两个状态位，R 位和 W 位，而没经过一次时钟中断，R 位被置零，于是共有以下四种情况：
第 0 类：没有被访问，没有被修改；
第 1 类：没有被访问，已被修改；（实际上由第 3 类经过时间中断得来）
第 2 类：已被访问，没有被修改；
第 3 类：已被访问，已被修改；
NRU 随机从类编号最小的非空类中选择一个页面淘汰。
3) FIFO 先进先出算法
4) 第二次机会算法
其实跟 FIFO 先进先出算法很像，就是添加了一个 R 位，当 R 最近访问的时候置为 1，当轮到某个页面的时候，如果它的 R 值为 0，则立马被置换出去，如果 R 为 1，则将 R 置零然后排到队列的最后。
5) 时钟算法
跟第二次机会算法很像，但是第二次机会算法频繁使用链表把第一项放到最后，开销很大，这里使用，这里把所有的链表项做成首尾相连的时钟形状，不用频繁置换，减少开销。
6) LRU 最近最少使用算法 Least Resently Used
LRU 置换算法是选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面上次被访问以来所经历的时间 t，当需要淘汰一个页面时，选择现有页面中其 t 值最大的。
7) NFU 最不经常使用算法
8) 老化算法
9) 工作集算法
工作集：最近 k 次内存访问所访问过的页面的集合，函数 w(k, t)是在 t 时刻

工作集的大小。

若每执行几条指令程序就发生一次缺页中断，那么就称这个程序发生了颠簸

10) 工作集时钟算法

11) 算法比较

表 3-1 页面置换算法的比较

算法名称	注释
最优页面配置算法	不可实现，但是可以作为基准
NRU 最近未使用页面置换算法	LRU 很粗糙的近似
FIFO 先进先出算法	可能抛弃重要的页面
第二次机会算法	比 FIFO 有较大的改善
时钟算法	现实的
LRU 最近最少使用算法	很优秀，但很难实现
NFU 最不经常使用算法	LRU 相对粗略的近似
老化算法	非常近似 LRU 的有效算法
工作集算法	实现起来开销很大
工作集时钟算法	好的有效算法

3.3 分页系统中的设计问题

1) 局部分配策略与全局分配策略

   一般全局分配策略要好于局部分配策略

2) 负载控制

3) 页面大小

4) 共享页面

   写时复制

5) 共享库

   只使用相对偏移量的代码被称作位置无关代码 position-independence code

   内存映射文件，进程所有的改动都放在内存中完成直至进程推出，此时所有的最终结果才会被写入硬盘中

6) 分布式共享内存

   该方法允许网络上的多个进程共享一个页面集合。

3.4 有关实现的问题

1) 与分页有关的工作

2) 进程创建时

需要为进程的指令和数据的初始化分配空间，在内存中创建页表；
为页表在内存中分配空间；
为了在缺页中断的时候能让内存和磁盘交换进程，所以需要要为磁盘交换区分配空间；
同时需要在磁盘交换区中为程序正文和数据进行初始化，这是以免在发生缺页中断调入新进程的时候再次进行初始化花费时间；
当调度一个进程执行的时候，必须为新进程重置 MMU，刷新 TLB，已清除以前进程遗留的痕迹。新进程的页表必须称为当前的页表。
当缺页中断的时候，定位到合适的虚拟地址，在磁盘中找到相应的页面，同时在内存中合适的页框来存放新的页面，同时准备置换老的页面。
当程序退出的时候，如果是共享页面，则需要等到最后一个进程退出的时候，才释放页表，页面和页面在硬盘上所占有的空间（即页框）

3) 指令备份
这是一个关于进程由于发生缺页中断后怎么回到引起缺页中断指令的位置的问题。解决办法，利用一个系统隐藏的寄存器，在每条指令执行之前，把程序计数器的内容复制到该隐藏的寄存器，以消除引起缺页中断指令所造成的所有影响。其实这里我看不太懂，这个缺页中断能造成多大的影响？

4) 锁定内存中的页面
这主要是关于进程 I/O 阻塞过程中发生中断的问题，一般来讲，发生 I/O 阻塞的进程是很少概率被置换出内存的，所采用的策略是锁住内存中的页面保证它不被移出内存，锁住一个页面通常称为在内存中钉住 pinning 页面。另一种方法是在内核缓冲区中完成所有的 I/O 操作。

5) 后备储存
大多 UNIX 的做法——磁盘交换区
与每个进程对应的是其交换区的磁盘地址，即进程映像所保存的地方。
写回一个页面的时候也很简单，只需要把虚拟地址空间中页面的偏移量加到交换区的开始地址即可。
进程启动时必须初始化交换区，一种方法是将整个内存映像复制到交换区，以便随时可将内容装入，另一种方法是将整个进程装入内存，需要时换出。

- 这里先讨论第一种情况——在磁盘中事先分配交换空间
  但这里会有一个问题，就是当进程运行的时候，尽管他的正文大小是不变的，但是他的数据有时会增长，最好为正文，数据和堆栈分别保留交换区，并且允许这些交换区在磁盘上多于一个块。
  当进程发生缺页中断的时候，对应的页面就会发生替换。一般来讲，进程在磁盘中都有对应的镜像副本。假如进程的页面在内存中被更换的页面发生缺页中断前没有被修改过，那直接就使用磁盘中的副本，如果被修改过了，说明磁盘中的副本已经过期了，需要从内存中取出被替换的页面到磁盘中。
- 接着讨论第二种情况——在磁盘中没有事先分配交换空间
  页面在磁盘上没有固定的地址，当页面被换出的时候，要及时选择一个空磁盘并据此来更新磁盘映射（每个虚拟页面都有一个磁盘地址空间）

Windows 的做法——直接使用文件系统

当然了，不能保证总能够实现固定的交换分区，比如没有磁盘分区可用的时候，可以利用文件系统中一个或多个交大的，事前定位的文件。Windows 就是使用这种方法。程序正文通常是可读的，在内存紧张不得不把页面移除的时候，尽管抛弃它，需要的时候再从可执行文件中读入即可。共享库也是采用这种方法的。

6) 策略和机制分离

7) 控制系统复杂度的一种重要的方法就是把策略从机制中分离出来，Mach 等人首先应用了这种分离，其中存储管理系统被分为三个部分：

- 一个底层的 MMU 处理程序
- 一个作为内核一部分的缺页中断程序
- 一个运行在用户空间中的外部页面调度程序

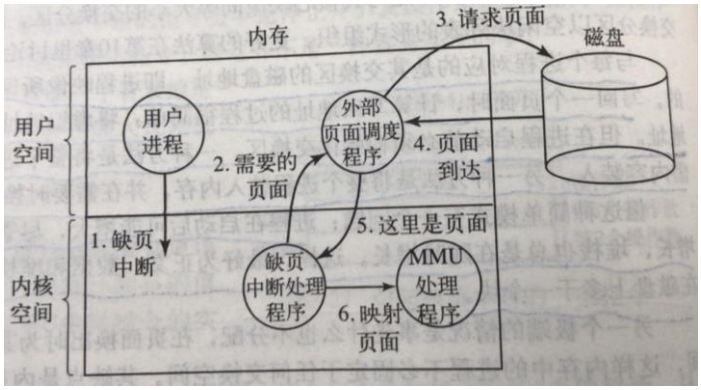


图 3-2 用一个外部页面调度程序来处理缺页中断

3.5 分段

说起为什么会出现分段的问题呢？其实程序在编译的时候需要管理很多的表和堆栈，运行的时候堆栈的大小是未知的，可大可小。如果虚拟地址都是一维的从大到小排列，一旦出现堆栈超出预期大小的时候，那虚拟地址该如何分配呢？解决问题的重点——分段。

1) 多个互相独立的称为段 segment 的地址空间，每个段由 0 到最大的线性地址序列构成，各个段的长度可以是 0 到某个允许的最大值之间的任何一个值。因为每个段都构成了一个独立的地址空间，所以它们可以独立地增大或者减小而不会影响到其他段。

需要强调的是，段是一个逻辑实体。一个段可能包括一个过程，一个数组，一个堆栈，一组数据变量，但他一般不会同时包含多种不同类型的内容。

2) 分段的好处

- 能简化对长度经常变动的数据结构的管理；
- 如果每个过程都位于一个独立的段中并且起始地址为 0，那么把单独编译好的过程链接起来的操作可以得到很大的简化。一个对段 n 中过程的调用将使用两个部分组成的地址（n, 0）来寻址到字 0。如果随后段 n 的过程被修改并被重新编译，及时新版本的程序比老版本的要大，也不需要对其他的过程进行修改，因为没有修改他们的其实地址。

- 分段有助于在在几个进程之间共享过程和数据，比如共享库；
- 不同种类的段可以收到不同种类的保护，不如有的只读，有的只写等；

3) 棋盘形碎片或者外部碎片 external fragmentation
系统运行一段时间后，由于内存中包含有很多的块，块中有很多的段，段与段之间很有可能会存在间隙或者叫空闲区。这种现象称为棋盘形碎片或者外部碎片 external fragmentation。这些空闲区不去使用的话就浪费了内存，需要通过内存紧缩来解决。
4) 分段与分页的结合——MULTICS
如果一个段太长了，以至于在一个页中放不下，则有可能把一个段分成好几页。MULTICS 是第一个实现了对页的分段支持的系统。
5) 不过，没有操作系统的开发者会仔细考虑分段，因为他们更青睐于其他的内存模型，这导致分段逐渐乏人问津，如今，即使 64 位版本的 x86 也不支持真正的分段。

四、 文件系统

4.1 长期储存信息的三个问题

所有的计算机应用程序都需要存储和检索信息会遇到三个问题：
- 第一个问题是：但储存容量受限于虚拟空间大小的限制。大的程序如航空订票系统，银行系统等，这些存储空间又显得太小。
- 第二个问题是：进程终结时，它保存的信息也会随之丢失，但有的程序需要做到即使进程崩溃，信息也不能丢失；
- 第三个问题是：经常需要多个进程同时访问同一信息或者其中部分信息。

解决方法必须满足三个要求：
- 能够储存大量信息；
- 使用信息的进程终止时，信息仍旧存在；
- 必须能使多个进程并发访问有关信息；

文件是进程创建的信息逻辑单元，从总体上讲，操作系统中处理文件的部分称为文件系统 file system。

4.2 文件

1) 在 Windows 95 和 Windows 98 使用的是 MS-DOS 的操作系统，使用的是 FAT-16 和 FAT-32。后来的操作系统换成了更先进的 NTFS。后来巨硬公司开发了 FAT-32 的拓展版本——ExFAT，现在 ExFAT 是巨硬公司唯一能满足）OS X 读写操作的文件系统。
2) 三种文件结构

- 字节序列 所有 UNIX 版本和 Windows 也是采用这种文件模型。
- 记录序列 读操作一个记录，写操作重写或者追加一个记录，是以前穿孔卡片年代流行的文件系统，现在基本没有了。
- 树 记录数，每条记录的固定位置都有一个键，按照键进行快速查找。这主要用在一些处理商业数据的大型计算机系统中。

3) 目录

记录文件的位置

- 一级目录系统 只有一个目录，目录下面全是文件
- 层次目录系统 现在个人计算机常用的文件目录系统
- 路径名

4) 文件系统的实现

- 连续分配 随着删减次数的增多，容易产生空闲区
- 链表分配 为每个文件构造磁盘块链表，虽然顺序读取文件很快，但是每次都需要从头读起，显然这种操作太慢了；
- 采用内存中的表进行链表分配 内存中的这个表称为文件分配表 File Allocation Table, FAT 速度快，但是缺点也很明显占用内存多；
- i 节点 用 i 节点的数据结构列出文件属性和文件块的磁盘地址；

后面感觉用处不大，就暂时不搞了

五、 输入/输出

下面详细介绍几种 I/O 设备：磁盘，时钟，键盘和显示器，最后还有电池管理。

5.1 I/O 设备

1) I/O 设备分为块设备 block device 和字符设备 character device

- 块设备把信息存储在固定大小的块中，每个块有自己的地址，通常块的大小在 512 字节到 65536 字节之间，所有的传输都是以一个或者多个连续的块为单位
- 块设备的基本特征是每个块都独立于其他块而读写。硬盘，蓝光光盘和 USB 盘都是常见的块设备。
- 磁盘是公认的可寻址设备
- 字符设备是以字符为传输单位发送或者接收一个字符流，而不考虑任何块结构。字符设备是不可寻址的，也没有任何寻道操作。如：打印机，网络接口，鼠标等。
- 但是这种分法并不完美，因为时钟他具有中断功能，但是却不具备发送数据的功能，显示屏同理。但是这种分法已经算具有足够的一般性了。

2) 设备控制器

- 设备控制器可分为机械部分和电子部分，电子部分称作设备控制器（device controller）或者适配器（adapter）

- 控制器的任务是把串行的位流转换为字节块，并进行必要的错误校正工作，字节块通常首先在控制器内部的一个缓冲区中按位进行组装，然后在对校验和进行校验，并证明字节块没有错误后，再将他复制到主存。
- LCD 显示屏的控制器也是一个串行设备，从内存中读取包含字符字节，然后改变背光的极化方式显示相应的像素。

5.2 内存映射 I/O

1) 内存映射 I/O

每个控制器有几个寄存器用来与 CPU 进行通讯。

每个控制寄存器被分配一个 I/O 端口 I/O port 号，这是一个 8 位或者 16 位的整数，所有的 I/O 端口空间（I/O port space）并且收到保护是的普通的用户程序不能对其进行访问（只有操作系统可以访问）

那具体 CPU 是如何与设备的控制寄存器和数据缓冲区进行通讯呢?这里有两个方案：

- 第一种方案： 内存地址空间和 I/O 地址空间是不同的。使用命令 IN 和 OUT 对控制器进行访问。
- 第二种方案：每个控制寄存器被分配唯一的内存地址，并且不会有内存被分配到这一地址。这样的系统称为内存映射 I/O memory-mapping I/O。这很方便，但是也有缺点：系统一旦对设备控制寄存器进行高速缓存，将会引发灾难性的后果。

5.3 直接存储器存取 Direct Memory Access，DMA

1) 直接存储器存取 Direct Memory Access，DMA

无论系统有没有内存映射 I/O，都需要寻址设备控制器以便跟它们交换数据。无论 DMA 处于什么地方，他都能独立于 CPU 而访问系统总线。它包含若干个可以被 CPU 读写的寄存器，其中包括一个内存地址寄存器，一个字节计数寄存器和一个或者多个控制寄存器。控制寄存器要指定使用的 I/O 端口，传送方向（从 I/O 设备读/写到 I/O 设备），传送单位（每次一个字节或者每次一个字），以及在一次突发传送中要传送的字节数。

2) DMA 工作原理
   - 没有使用 DMA 时

     为了解释 DMA 的工作原理，让我们首先看一下没有使用 DMA 时磁盘如何读的。首先，控制器从磁盘中串行的，一位一位地读一个块（一个或者多个扇区），直至将整块信息放入控制器的内部缓冲区中。接着，他计算校验和，以保证没有读错误的发生。然后控制器产生一个中断，此时操作系统就重复地从缓冲区一次一字节或者一个字地读取该块的信息，并将其存入内存中。
   - 使用 DMA 时

     使用 CPU 对 DMA 进行编程，让他知道需要向哪个或者哪几个控制器读

写数据，而数据在磁盘缓冲区直接通过总线存到内存中。不过好处是可以进行多路传输。

5.4 重温中断

1) 精确中断

需要满足一以下四个要求：

- PC 程序计数器保存在一个已知的地方；
- PC 所指向的指令之前的所有指令已经完全执行；
- PC 所指向的指令之后的所有指令都没有执行；
- PC 所指向的指令的执行状态是已知的；

代价：CPU 及其复杂的中断逻辑，这里的代价不在时上，而是在芯片面积和设计的复杂性上。如果不是为了往后兼容，芯片面积越大的话可以用于更大的片上的高速缓存，从而可以使 CPU 速度更快。

2) 不精确中断

不满足以上四个要求的中断；

代价：操作系统更为复杂而且运行得更加缓慢。

5.5 I/O 软件原理与层次

1) 一个关键的概念是：设备独立性

可以访问任意 I/O 设备而无需事先指定设备。

包括：

- 统一命名 uniform naming
- 错误处理 error handling
- 同步（synchronous，即阻塞）和异步（asynchronous，即中断驱动）
- 缓冲（buffering）

2) 层次结构

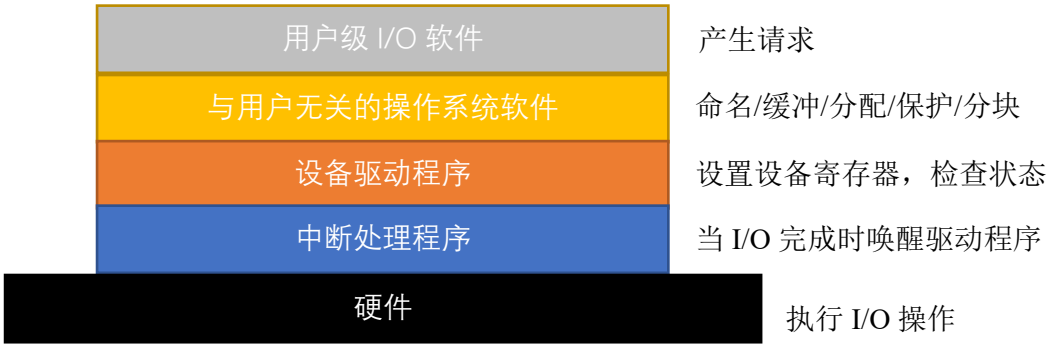


图 5-1 I/O 软件系统的层次

5.6 盘

1) 磁盘

低级格式化的结果是磁盘容量减少，减少的量取决于前导码，扇区间间隙和 ECC 大小以及保留的备用扇区的数目。

2) 电梯算法 elevator algorithm
电梯按照一个方向移动，直到在那个方向上没有请求为止，然后改变方向

3) 错误处理
磁盘的线性密度大约为 5000b/mm，这需要非常精密的氧化物涂层，按照这样的标准去制造而没有瑕疵这是很不可能的。

六、 死锁

6.1 资源

1) 大多数的死锁都跟资源有关系，那什么是资源呢？资源就是随着时间的推移，必须能获得，使用以及释放的任何东西。
2) 可抢占资源和不可抢占资源
- 可抢占资源 preemptable resource
  可以从拥有他的进程中抢占而不会产生任何的副作用，如存储器。
- 不可抢占资源
  在不引起相关计算失败的情况下，无法把它从占有它的进程中抢占出来。如蓝光光盘刻录机，因为被另外的进程抢占后，蓝光光盘刻录机会被划坏。

总的来说，死锁是跟不可抢占资源有关系的，因为可抢占资源都是可以通过进程之间的重新分配资源得到化解。

3) 请求资源可以抽象为三个过程\
   - 请求资源
   - 使用资源
   - 释放资源
4) 一种允许用户管理资源的可能方法是为每个资源配置一个信号量，信号量为 down 的时候请求资源，使用资源，信号量为 up 的时候释放资源，但是这种方法在只有一种资源的时候可以很好的工作。问题就出在于出现两种或以上资源的时候，试看如下代码：

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void){
    down(&resource_1);
    down(&resource_2);
    use_both_resource();
    up(&resource_2);
    up(&resource_1);
}

void process_B(void){
    down(&resource_1);
    down(&resource_2);
    use_both_resource();
    up(&resource_2);
    up(&resource_1);
}
```

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void){
    down(&resource_1);
    down(&resource_2);
    use_both_resource();
    up(&resource_2);
    up(&resource_1);
}

void process_B(void){
    down(&resource_2);
    down(&resource_1);
    use_both_resource();
    up(&resource_1);
    up(&resource_2);
}
```

假如现在同时有两个进程：进程 A 和进程 B

- 情况 1：进程 A 或者进程 B 同时拥有两个资源，另一个进程只好被阻塞，等前面的进程释放资源；
- 情况 2：两个进程同时拥有一个不相同的资源，接着他们又同时向对方请求另外一个资源的时候，就会被相互阻塞，此时就会形成死锁。

6.2 死锁 deadlock 简介

1) 定义

如果一个进程的集合中的每一个进程都在等待只能由该进程集合中的其他进程才能引发的事件，那么，该进程的集合就是死锁的。

如果“其他进程才能引发的事件”指的是其他进程所释放的资源，这种死锁称为资源死锁 resource deadlock

2) 资源死锁的四个必要条件

该条件是由 Coffman 等人在 1971 年提出的：

- 互斥条件：资源要么被使用，要么处于可被使用的状态（互斥独占）；
- 占有和等待条件：已经占有某个资源的进程可以请求新的资源（得一想二）；
- 不可抢占条件：资源一旦被使用，其他资源就不能强制占用，除非资源被正在使用大的进程显式地释放（保护私有财产）。
- 环路等待条件：死锁发生的时候，系统必须有两个或两个以上的进程组成的一条环路，该环路中的每一个进程都在等待着下一个进程所占有的条件。（环路死锁）

3) 资源分配图

Holt（1972）指出如何用有向图建立上述四个条件的模型，圆形节点表示进程，方形节点表示资源。

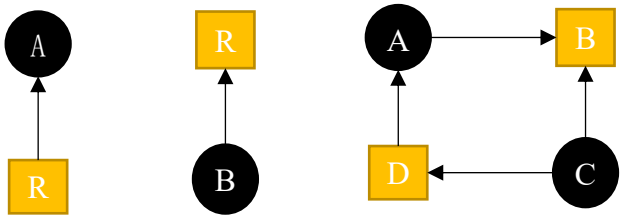


图 6-1 表示进程 A 占有资源 R

图 6-2 表示进程 B 请求资源 R

图 6-3 表示死锁的状态实例

4) 处理死锁的四种策略

- 忽略该问题。也许如果你忽略它，它也会忽略你；（这是什么意思？逃避问题吗？）
- 检测死锁并恢复，让死锁发生，检测它们是否发生，一旦发生死锁，采取行动解决问题；
- 仔细对资源进行分配，动态地避免死锁；
- 通过破坏引起死锁的四个必要条件之一，防止死锁的产生；

6.3 鸵鸟算法

把头埋在沙子里，假装根本没有问题的发生。

6.4 死锁检测和死锁恢复

1) 每种类型一个资源的死锁检测

利用链表检测每一个资源出现的个数，如果出现两个或以上，说明出现死锁，如果出现最后一个箭头后面没有找到相应的项，说明此路径没有死锁。

但是这策略实施起来比较麻烦，而且开销比较大，并不是一个好的算法，只能说存在这样的解决算法；

2) 每种类型多个资源的死锁检测

$$\text{总资源矩阵 } \mathrm{E} = [\qquad\qquad]$$

$$\text{剩余资源矩阵 } \mathrm{A} = [\qquad\qquad]$$

$$\text{当前分配资源矩阵 } \mathrm{C} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

当前请求资源矩阵 R = $\begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1m} \\ R_{21} & R_{22} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$

矩阵 C 和 R 的每一行代表一个进程的资源分配和请求。将当前请求资源矩阵 R 的每一行和 A 进行比较，假如 A 中某一个元素小于 R 中相对应的元素，说明该进程无法等到运行，只能等其他可运行的进程运行释放资源后，在进行比较看能不能运行。如果矩阵 R 中再没有可以通过 A 释放资源的进程，这时候死锁就发生了。

也就是说，假如存在某个调度次序使得各个进程能依次得到资源请求的满足，那就不会发生死锁了。

6.5 从死锁中恢复

1) 利用人工干预抢占恢复

强制挂起某个进程，将其资源分配给发生死锁的进程，让其顺利渡过死锁后，再归还相应的资源。有点某性社国家主动干预经济的意味，但是主动干预有主动干预的好处，起码资源能得到较为合理的利用。置于选择挂起哪个进程，则很大程度上取决于哪一个进程拥有比较容易回收的资源。

2) 利用回滚恢复

其实就是让发生死锁的某个进程恢复到过去的某一个不存在竞争目标资源的状态，好让另一个进程可以顺利得以运行，最后如果想要恢复原来的状态，则必须一直等到该目标资源可用为止。

3) 通过杀死进程恢复

- 第一种情况：杀死环中的进程以释放资源，如果幸运额是可以一次性解决问题，否则则需要多杀几个，这方法有点恐怖；
- 第二种情况：可以杀死环外的进程以释放资源；

以上两种情况最好选择哪些可以重新运行而且不会带来副作用的进程。所谓的副作用，举个例子，在数据库中某个进程的作用是对某个值进行累加，假如你杀死了他，再次运行的时候他就把累加值变成加 2 了，这就出问题了。

6.6 死锁避免

1) 利用资源轨迹图

就是眼看还差一部就发生死锁了，赶快把某个进程挂起；

2) 安全状态与不安全状态

利用每种类型多个资源的死锁检测的四个矩阵+调度算法

安全状态：系统利用某个调度算法能保证所有进程都能顺利完成；

不安全状态：系统利用某个调度算法不能保证所有进程都能顺利完成；

3) 单个资源的银行家算法 banker’s algorithm

4) 多个资源的银行家算法
5) 但是银行家算法有一个要求就是需要事先知道最大进程数，但是在系统中进程数是在不断变化的，所以只有极少的系统使用银行家算法来避免死锁。不过有些系统可以使用银行家算法的启发式算法来避免死锁，其实就是做出进程数的预估。

6.7 死锁预防

1) 破坏互斥条件
   让一个资源不被独占。
   假脱机打印机技术 spooling printer：可以允许若干个进程同时产生输出。该模型中唯一真正请求使用物理打印机的进程是打印机守护进程，由于守护进程绝不会请求别的资源，所以不会因打印机而产生死锁。
   解释：所谓“脱机”，百度百科给出的解释是离线下访问(磁盘缓存)中的网页。
   假脱机技术即 SPOOLing（Simultaneous Peripheral Operating On-Line）SPOOLing 技术是用于将一台独占设备改造成共享设备的一种行之有效的技术。当系统中出现了多道程序后，可以利用其中的一道程序，来模拟脱机技术输入时的外围控制机的功能，把低速输入输出设备上的数据传送到高速磁盘上；再用另一道程序来模拟脱机输出时外围控制机的功能，把数据从磁盘传送到低速输出设备上。这样便可以在主机的直接控制下，实现脱机输入、输出功能。
   小思考：尽量不要分配那些绝对必须的资源，尽量做到尽可能少的进程可以真正请求资源。
2) 破坏占有和等待条件
   有两种方案。
   - 第一种方案：在进程运行之前给他配齐所有需要的资源，但是有一个非常直接和现实的问题是，你无法知道进程在运行过程中到底需要多少进程，假如你一开始就知道需要多少进程，就可以采用银行家算法。
   - 第二种方案是：在进程运行并请求相关资源过程中，先释放当先被占用的资源。
3) 破坏不可抢占条件
   采用资源虚拟化的方法，那到底什么是资源虚拟化方法啊，书上没有详细讲，网上也没有找到确切的答案。
4) 破坏环路等待条件
   有两种方案。
   - 第一种方案：保证每一个进程在任何时刻都只能占用一个资源，如果要请求另一个资源，必须首先释放另外一个资源。
   - 第二种方案是：将所有的资源编号，但是所有的请求必须按照资源编号的顺序（升序）提出，这个方法的缺点是过于死板不灵活，而且开销很大。以至于编号方法无法使用。

6.8 其他问题

1) 两阶段死锁
2) 通讯死锁
3) 进程 A 向进程 B 发送消息，如果信息丢失，则进程 A 会被阻塞，由于 B 没有收到 A 的回复，则 B 也会被阻塞。
4) 幸运的是，有一种解决办法：超时策略，超过一定的时间重新发送资源。
5) 活锁
6) 当进程意识到他不能获取所需要的下一个资源的时候，他就会礼貌性地释放已经获得的锁，然后等待 1ms，然后再尝试一次。但是万一对面的进程也同时采取同样的方法，那两个也都会被锁住。

七、 虚拟化和云

虚拟化的主要思想是虚拟机监控程序 Virtual Machine Monitor，VMM。在同一物理硬件上创建出多台虚拟机的假象。VMM 又称为虚拟机管理程序 hypervisor。

- 隔离性：在一个机器上运用虚拟化技术可以创建不同的操作系统，一个系统崩了不会影响到另外一个；
- 物理机数量的减少节省了硬件和电力开销以及机架空间的占用；
- 设置检查点和虚拟机迁移方便；
- 软件开发方便和兼容性

7.1 历史

1) 虚拟化技术并不是什么新鲜的技术，早在 1974 年该理论就已经明确了。

7.2 虚拟化的必要条件

1) 虚拟机管理程序需要在三个维度上有良好的表现：
   - 安全性
     虚拟机管理程序应该完全掌控虚拟资源；
     有些安全指令可以直接执行，一些不安全的指令需要由解释器来模拟；
   - 保真性
     程序在虚拟机上执行的行为应与在裸机上执行是一样的；
   - 高效性
     虚拟机上运行的大部分代码应不受虚拟机管理程序的干涉。
2) 敏感指令和特权指令
   - 每个包含内核态和用户态的 CPU 都有一个特殊的指令集合，其中的指令在内核态和用户态中执行的行为不同，这些指令包括进行 I/O 操作和修改 MMU 的设置的指令。这些指令 Popek 和 Goldberg 称作“敏感指令

sensitive instruction”。

- 另外还有一种指令集合，在用户使用的时候会导致陷入，Popek 和 Goldberg 称作“特权指令 privilege instruction”。Popek 和 Goldberg 他们的论文首次提出虚拟化的一个必要条件是敏感指令必须是特权指令的子集。简单来讲就是如果用户想要做不应该在用户态做的事情时，内核必须陷入。

3) VT 技术和 SVM 技术

- 关于虚拟技术，2005 年起，Intel CPU 开发的是 Virtualization Technology，AMD CPU 开发的是 Secure Virtual Machine。
- VT 技术的思想核心是创建可以运行虚拟机的容器，客户操作系统在容器中启动并持续运行，直至触发异常并陷入虚拟机管理程序，虚拟机管理程序可以管理硬件。有了这些拓展后，在 X86 平台实现经典的陷入并模拟 trap-and-emulate 虚拟机称为可能。
- 没有这种技术之前，比如 1999 年的 VMvare 都是通过二进制翻译技术将一些不安全代码替换成安全代码来执行的。

7.3 第一类和第二类虚拟机管理程序

1) 第一类和第二类虚拟机管理程序

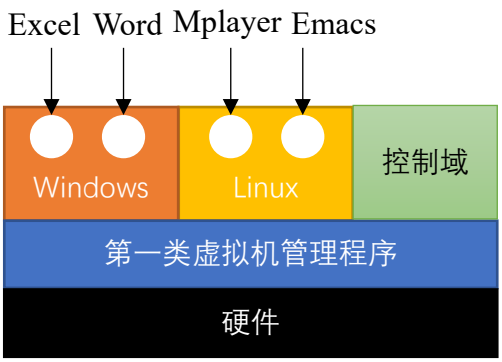


图 7-1 第一类虚拟机管理程序

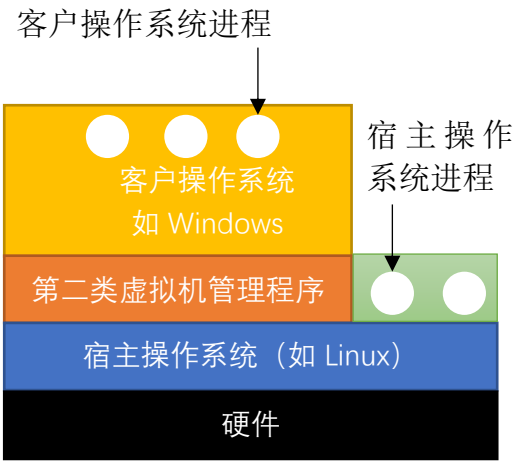


图 7-2 第二类虚拟机管理程序

2) 客户操作系统和宿主操作系统

运行在两类虚拟机管理程序尚的操作系统都称作客户操作系统 guest operating system

VMware Workstation 是首个 x86 平台的第二类虚拟机管理程序。

3) 高效虚拟化技术

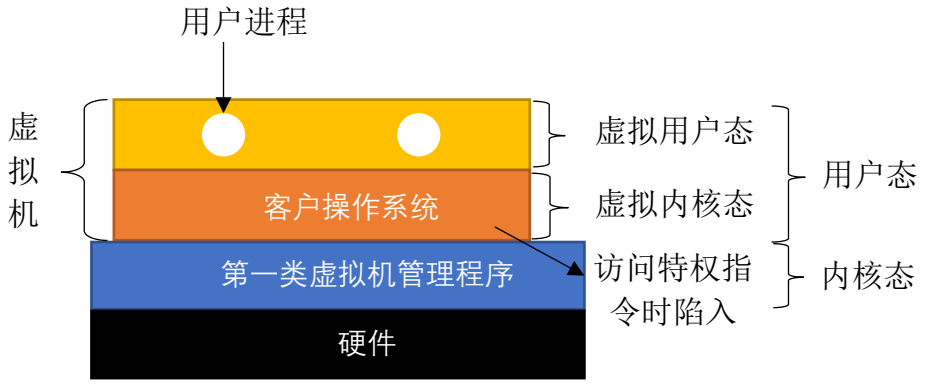


图 7-3 当虚拟机中的操作系统执行了一个内核指令时，如果支持虚拟化技术，那么它会陷入虚拟机管理程序

虚拟机管理程序可以检查这条指令是由虚拟机中的客户操作系统执行的还是用户程序执行的，如果是前者，虚拟机管理程序将安排这条指令功能的正确执行，否则，虚拟机管理程序将模拟真实硬件面对用户态执行敏感指令时的行为。

4) 在不支持虚拟化的平台上实现虚拟化

多年来，x86 支持四个特权级，用户程序运行在第三级，第 0 级是最高特权级，允许执行任何指令，正常运行中，操作系统位于第 0 级。客户操作系统安排到一个中间特权级

5) 虚拟化的开销比较大

6) 全虚拟化和半虚拟化

全虚拟化是指客户操作系统没有经过修改，并且会由于敏感指令而陷入。而半虚拟化是指客户操作系统经过修改，但是敏感指令会被替换成虚拟化调用。

7) 内存虚拟化

对于每台虚拟机，虚拟机管理程序都需要创建一个影子页表 shadow page table，将虚拟机使用的虚拟页映射到他分配给虚拟机的实际物理页上。

两种缺页异常

- 第一种：客户机导致的缺页异常，虽然由虚拟机管理程序捕获，但需要交给客户机处理。
- 第二种：后者是虚拟机管理程序导致的缺页异常，处理方式是更新影子页表。

7.4 I/O 虚拟化

1) 设备穿透 device pass through

允许将物理地址直接分配给特定的虚拟机，通常，设备地址空间与客户物理地址空间完全相同比较有利，而这依赖于 I/O MMU。I/O MMU 可以将设备地址与虚拟机地址映射为相同的空间，并且这一映射对设备和虚拟机来说都是透明的。

2) 设备隔离 device isolation

保证设备可以直接访问其分配到的虚拟机的内存空间而不影响其他虚拟机

的完整性。

3) 单根 I/O 虚拟化
理想的情况下，虚拟化技术能提供单个设备到多个虚拟机中等效设备的穿透能力而且没有额外开销。如果硬件本身能进行虚拟化，则虚拟化单一设备以使每个虚拟机都认为自己拥有对设备的独占式访问会容易得多。在 PCIe 标准里这种虚拟化称作单根 I/O 虚拟化。
4) 磁盘以扇区大小的块来读写数据。对扇区的访问时间 access time 主要由三部分组成：寻道时间 seek time，旋转时间 rotational latency 和传送时间 transfer time。
5) 授权问题

7.5 云

美国国家标准与技术研究院 National Institute of Standards and technology
云大的五个必要特征：

- 按需自主服务
  无需人为操作用户就能自动为用户提供资源；
- 普适的网络访问
  所有的资源都可以用过网络用标准化的机制访问，以支持各种异构设备；
- 资源池
- 快速可伸缩
  可以根据用户需求弹性甚至是自动获取和释放资源；
- 服务可计量
  云提供商按服务类型计量用户使用的资源。

1) 云即服务
2) 虚拟机迁移
热迁移 live migrations：虚拟机在迁移的时候仍然可以运行，使用内存预复制迁移 pre-copy memory migration，能在虚拟机提供服务的时候同时复制内存页。
3) VMware Workstation 的案例
这个有时间再详细看吧

八、 多处理机系统

共享储存器多处理机，简称多处理机 multiprocessor：
其两个或者更多个 CPU 全部共享访问一个公用的 RAM。

8.1 多处理机

1) 所有的多处理机都具有每个 CPU 可访问全部存储器的性质

2) 统一存储器访问 Uniform Memory Access，UMA
读出每个字的速度一样快的；
3) 非一致存储器访问 Nonuniform Memory Access，NUMA
读出每个字的速度不一样快的；
4) 基于总线的 UMA 多处理机体系结构
最简单的多处理机是基于单总线的，若有 32 个或者 64 个 CPU 时，就不能忍了，这种系统完全受总线带宽的限制，多数 CPU 在大部分时间里是空闲的。
解决的方案：为每个 CPU 添加一个高速缓存 cache，这个缓存可以位于 CPU 芯片的内部，CPU 附近，在处理器板上或所有这三种方式的组合。
5) 使用交叉开关的 UMA 多处理机
6) 多级交换网络/omega 网络
omega 网络是一种阻塞网络，并不是每组请求都可被同时处理，冲突可以在一条连线或者一个开关中发生。
7) NUMA 多处理机
像 UMA 一样，这种及其为所有的 CPU 提供了一个统一的地址空间，但与 UMA 不同的是，访问本地存储器模块快于访问远程存储器模块。但是性能不如 UMA 机器的性能。
主要区别是：
   - 具有对所有 CPU 都可见的单个地址空间；
   - 通过 LOAD 和 STORE 指令访问远程存储器；
   - 访问远程存储器慢于访问本地存储器；
8) 多核芯片
虽然 CPU 可能共享高速缓存或者不共享，但是他们都共享内存，考虑到每个内存的字都有位移的值。当其中某个 CPU 修改了该字，所有的其他高速缓存中的该字都会被自动地并且原子性的删除来确保一致性，这个过程称为窥探 snooping。
9) 众核芯片
处理器的核数可能有几十，几百甚至成千上万个，超大量核带来的一个问题是：用来保持缓存一致性的机制会变得非常的复杂和昂贵。更糟糕的是，保持缓存目录的一致性还将消耗大量的内存，这就是著名的一致性壁垒。所以工程师们正考虑要放弃缓存一致性。
图形处理单元 GPU 是当今最为常见的众核。
10) 异构多核
在一块芯片尚封装了不同类型的处理器的系统被称为异构多核处理器。
11) 在多处理机操作系统
每个 CPU 有

8.2 多处理机操作系统类型

1) 每个 CPU 都有自己的操作系统
最简单的办法是静态地把存储器划分成和 CPU 一样多的各个部分，为每个

CPU 提供其私有存储器以及操作系统的各自私有副本。实际上 n 个 CPU 以 n 个独立计算机形式运行。这样做的一个明显的优点是，允许所有的 CPU 共享操作系统的代码，而且只需要提供数据的私有副本。
这机制比有 n 个分离的计算机要好，因为它允许所有的机器共享一套磁盘和其他 I/O 设备，它还允许灵活共享存储器。
缺点：没有共享进程，没有共享物理页面，缓存不一致

2) 主从多处理机
再此模型中，从机 CPU2，CPU3…运行客户程序，操作系统的一个副本及其数据表都在 CPU 1 上，而不在其他所有的 CPU 上。为了在该 CPU 1 上进行处理，所有的系统调用都重定向到 CPU1 上，如果有剩余的 CPU 时间，还可以在 CPU1 上运行用户进程。
它解决了每个 CPU 都有自己的操作系统的四个问题，但是有一个问题是 CPU1 要处理其余所有 CPU 的系统调用，很容易会过载。所以这个模型对于小型多处理机还是可行的，但是对于大型的处理机就有点力不从心了。

3) 对称多处理机 Symmetric MultiProcessor，SMP
存储器中有操作系统的一个副本，但任何 CPU 都可以运行它。在有系统调用时，进行系统调用的 CPU 陷入内核并处理系统调用。任何 CPU 都可以运行操作系统，但任一时刻只有一个 CPU 可以运行操作系统，这一方法称为大内核锁 Big Kernel Lock，BLK
大多数现代多处理机都采用这种管理方式。为这类及其编写操作系用的困难，不在于代码增多，而在于如何划分为可以由不同的 CPU 并行执行的临界区而不会相互干扰。

8.3 多处理机同步

1) 为了防止多处理器的竞争，需要使用 TSL test and set lock。锁住总线，阻止其他的 CPU 访问它。但是这种方法使用了自旋锁 spin lock。因为请求的 CPU 知识在原地尽可能快的对锁进行循环测试，这样做不仅完全浪费了提出请求的各个 CPU 的时间，而且还给总线或者存储器增加了大量的负载，严重降低了所有其他 CPU 从事正常工作的速度。
解决办法：自旋，切换，自选和切换的组合

8.4 多处理机的调度

1) 在单处理机上，解决的问题是一维的，“接下来要运行那个线程”。如果是内核级线程的系统，发展趋势是内核选择线程作为调度单位。
2) 在多处理机上，调度是二维的，“接下来必须决定在哪一个线程运行以及在哪一个 CPU 上运行”
3) 分时
最简单的算法是，为就绪线程维护一个系统级的数据结构，他可能只是一个链表，但更多的情况可能是对应不同优先级一个链表集合。但这种分时算法

的缺点在于自旋锁的问题。

为了克服这个问题，有的采用了智能调度 smart scheduling。就是给持有自旋锁的线程设置一个进程范围内的标志位，以表示目前它拥有一把自旋锁。当他释放该自旋锁的时候，就清除这个标志。

其他补充：亲和调度 affinity schedule：尽量使一个线程在他前一次运行过的同一个 CPU 上运行。

两级调度算法 two-level scheduling：它把负载大致平均分配在可运行的 CPU 上；它尽可能发挥高速缓存的亲和力的优势；通过为每个 CPU 提供一个私有的就绪线程链表，使得对就绪线程链表的竞争减到了最小，因为试图使用另一个 CPU 的就绪线程链表的机会相对较小。

4) 空间共享

在任何一个时刻，全部 CPU 被静态的划分成若干个分区，每个分区都运行一个进程中的线程。随着时间的流逝，新的进程建立，旧的进程中止，CPU 分区大小和数量都会发生变化。

实际情况中，采用最短作业优先算法要胜过先来先服务算法是非常困难的，因为很难获得准确的作业时间。

明显的优点：消除了多道程序设计，从而消除了上下文切换的开销。

明显的缺点：当 CPU 被阻塞或者根本无事可做的时间被浪费了，只能等到再次就绪。

于是，人们寻找既可以调度时间，又可以调度空间的算法，特别是对于要创建多个线程而这些线程通常需要彼此通讯的线程。

5) 群调度 Gang Scheduling

群调度由三个部分组成：

- 把一组相关线程作为一个单位，即一个群 gang，一起调度；
- 一个群中的所有成员在不同的分时 CPU 上同时运行；
- 群众所有成员共同开始和结束其时间片；

群调度的思想是，让一个进程的所有线程在不同的 CPU 上同时运行，这样，如果其中一个线程向另一个线程发送请求，接收方几乎立即得到消息，并且几乎能够立即应答。

8.5 多计算机

多计算机是紧耦合 CPU，不共享存储器，每台计算机有自己的存储器

1) 互连技术

直径，任意两个节点之间的最长路径

- 星型拓扑结构

  现代交换型以太网就采用这种类型的拓扑结构
- 网格

  是一种在许多商业系统应用的二维设计，它相当规整，容易拓展为大规模系统，直径=节点总数的平方根；
- 双凸面

  网格的变形，这种拓扑结构不仅交网格具有更强的容错能力，而且直径

也比较小，因为对焦之间的通讯只要两跳。

- 立方体
  是一种规则的三维拓扑结构。
- 超立方体
  许多并行计算机采用这种超立方体拓扑结构，因为其直径随着维数的增加而线性增加。换句话说，直径是节点数的自然对数。然而，其代价就是扇出数量以及由此而来的连接数量成本的大量增加。

2) 交换机制
   - 存储转发包交换 store-and-forward packet switching
     每个消息被分解成为最大长度限制的块，称为包 packet。当整个包到达一个输入缓冲区时，它被复制验证其路径通向下一个交换机的队列中。直到整个包到达目标节点的交换机。
     尽管存储转发包比较灵活且有效，但是他存在通过互连网络时增加时延的问题。
   - 电路交换
     它由第一个交换机建立，通过所有交换机而到达目标交换机的一条路径，一旦该路径建立起来了，比特流就从源到目的地通过整个路径不断地尽快的输送，在所有涉及的交换机中，没有中间缓冲，电路交换建立需要一个建立的过程，需要一点时间，但是一旦建立完成，速度就非常快。在发送完毕后，该路径必须被拆除。
3) 网络接口
   在多计算机中，所有的节点里都需要一块插板卡，由于比特率必须以恒定速度连续进行，如果包在主 RAM 上，速度不能保证恒定，所以需要在插板卡内置专门的 RAM。
4) 下面关于网络的部分后面再看《计算机网络 自顶而下方法》

九、 安全

漏洞：日益庞大的操作系统和应用导致系统不乏错误，当这些错误涉及安全类别的时候，我们称之为漏洞 vulnerability。
安全中的核心方法——密码学，以及不同的安全认证方式。

9.1 环境安全

1) 安全
   安全分为三个部分 CIA Confidentiality，Integrity，Avaliability
   - 机密性
   - 完整性
   - 可用性
2) 黑客
   将那些试图闯入计算机系统但是不属于破解者或者黑帽的那些人为黑客。

3) 被动攻击和主动攻击
被动攻击指的是试图窃取信息，主动攻击会使计算机程序行为异常。
4) 加密
将消息或者文件进行转码，除非获得密钥，否则很难回复原消息。
5) 程序加固
在程序中加入保护机制从而使得攻击者很难破坏程序；
6) 可信计算基
可信系统的核心是最小的可信计算基 Trusted Computing Base, TCB。
典型的 TCB 包括了大多数的硬件，操作系统核心中的一部分，大多数或所有掌握超级用户权限的用户程序等。必须包含在 TCB 中的操作系统功能有：进程创建，进程切换，内存面管理以及部分和 I/O 管理。在安全设计中，为了减少空间以及纠正错误，TCB 通常完全独立于操作系统的其他部分。
7) 访问监视器
TCB 一个重要组成部分是访问监视器，他要求所有有关安全的系统请求必须经过访问监视器的检查才能运行。
8) 保护域
域 domain 是（对象，权限）对的集合，每一对组合指定一个对象和一些可在其上运行的操作子集。任何时间，每个进程都会早某个保护域中运行。
9) 最低权限原则 Principle of Least Authority，POLA
一般而言，一个域拥有最少的对象和满足其完成共工作所需的最低权限时，安全性将达到最好。
10) 访问控制列表
列表里包含了所有可访问对象的域以及这些域如何访问这些对象的方法。主要用来管理（用户对）文件的访问。
11) 权能字
每个进程都有一个权能字列表，与每个进程关联的可访问的对象列表，以及每个对象尚可执行的操作。每个权能字赋予所有者针对特定对象的权限。
权能字可放在用户空间中，并用加密方法管理，这种方法特别适合于分布式操作系统。
权能字中通常包含可用于所有对象的普通权限，如：
- 复制全能字
- 复制对象
- 移除权能字
- 销毁对象

9.2 安全系统的形式化模型

1) Bell-LaPadula 模型
最广泛使用的多级安全模型是 Bell-LaPadula 模型，最初是为军方所设计的，后来推广到其他领域。
简单安全规则：在密级 k 上面裕兴的进程只能读取同一密级或者更低密级的对象；

*规则：在密级 k 上面运行的进程只能写同一密级或者更高密级的对象；

2) Biba 模型

跟 Bell-LaPadula 模型完全相反的模型

- 简单完整性规则

  往下写没有往上写

- 完整性*规则

  往上读，不能向下读

3) 隐蔽信道

在该系统内服务器进程不能把客户机进程合法获得的信息泄漏给协作进程，Laopson 把这个问题叫做界限问题（confinement problem）

http://blog.csdn.net/otianye/article/details/51258541

经常抓包分析的小伙伴们可能注意到了一个现象，有时候你获取的 HTTP 包有点别扭——HTTP 包的状态行或者消息报头有冗余（字段等）、HTTP 消息报头中的一些关键词大小写不规整（比如“Date”写为“daTe”等）……这个时候你要注意了，可能这些“特殊处理”在悄悄的传递信息！这，就是传说中的“隐蔽信道”!

简单来说，通信信道在设计的时候不是用来传递信息的（如 HTTP 结构），但是一些“高手”通过一些手段（上面列了几种）使用它们达到信息传递的目的，这是一种很隐蔽的事情很容易被忽略，这种信道其实就是“隐蔽信道”，它是信息隐藏技术的扩展。

隐蔽信道的存在，对安全操作系统是一个重大的威胁。因此对高安全等级的安全操作系统，各种标准都要求进行隐蔽信道分析。1985 年，美国国防部发布了橘皮书 TCSEC，这是第一个“计算机安全产品评估标准”（很多重要标准都以此为重要参考，比如国标 GB 17859-1999、GBT 28452-2012 等），明确规定在对 B2 级（对应国家信息安全等级中的第四级）以上的高等级安全操作系统进行评估时，必须分析隐蔽信道。

隐蔽信道的分类方法有很多种，通常的划分方式是“存储隐蔽信道”和“时间隐蔽信道”两类，而“时间隐蔽信道”又分为“普通时间隐蔽信道”和“网络时间隐蔽信道”等。信息化时代下，计算机网络协议是网络环境的重要支撑，也逐步成为“隐蔽信道”的沃土。

例子：隐写术

9.3 密码学原理

1) 私钥加密技术

单字母替换，总共有 26！$\approx 4*10^{26}$ 个方案，不过这个还是比较容易破解的，破解的关键在自然语言的统计特性，特别是在一小段密文里面。

对于严格的安全系统来说，最少使用 256 位密钥，因为它的破译空间是 256！$\approx 1.2*10^{77}$

缺点：发送者和接收者必须拥有同一个密钥，他们甚至需要物理上的接触，才能传递密钥

2) 公钥加密技术
   特点：加密密钥和解密密钥是不同的，通过加密密钥是推不出解密密钥的。这种特性下，加密密钥可被公开而只有解密密钥处于秘密的状态。
   公钥机制的主要问题是运算速度比对称密钥机制慢数千倍。
3) 数字签名
4) 可信平台模块
   TPM，一种加密处理器芯片。

9.4 软件漏洞

1) 对于缓冲区溢出漏洞的三大解决方案：
   栈金丝雀保护
   在程序调用的地方，编译器在栈中插入一段代码来保存一个随机的金丝雀值，就在返回地址之下。从调用返回时，编译器插入代码检测这个金丝雀值，如果这个值变了，就是出错了，这种情况下最好停下来检查一下。
   数据执行保护
   具体来说就是许多现代操作系统试图确保数据段是可写的，但不可执行；而文本段是可执行的，但不可写。
   地址空间布局随机化；
2) 后门陷阱
3) 登陆欺骗
4) 特洛伊木马
5) 病毒的工作原理
   - 共事者病毒；
   - 可执行程序病毒；
   - 内存驻留病毒
   - 引导扇区病毒
   - 设备驱动病毒
   - 宏病毒
   - 源代码病毒

十、 实例研究——UNIX

10.1UNIX 和 Linux 历史

1) UNIX 一开始只是年轻研究员 Ken Thompson 的一个业余的研究项目。
2) 20 世纪四五十年代，但是电脑也算是个人计算机，只是采用个人按时间租赁的方式。

3) POSIX 项目，前三个字母是 Portable Operating System，IX 与 UNIX 的构词相似。POSIX 委员会制定了 1003.1 的标准，它规定了每一个符合标准的 UNIX 系统必须提供库函数。而大多数库函数会引发系统调用。虽然 1003.1 标准只解决了系统调用的问题，但是一些相关的文档对线程，应用程序，网络及 UNIX 的其他特性进行了标准化。

4) MINIX

MINIX 是一种类 UNIX 的系统，微内核背后的思想是在内核中只提供最少的功能，从而使它可靠和高效。因此内存管理和文件系统被用作用户进程实现，内核只负责进程间的信息传递。微内核的主要缺点是用户态和内核态的额外切换会带来较大的性能损失。

后来 MINIX 继续发展，并且拥有一个比较活跃的用户群体。后面该系统就被移植到 ARM 处理器中，可运用于嵌入式系统。

10.2Linux 简介

1) 一个芬兰的学生 Linux Torvald 借鉴了 MINIX，写出了 Linux，其实也就是重写了 MINIX，他是一个整体式的设计，把整个操作系统都包含在内核中去了。

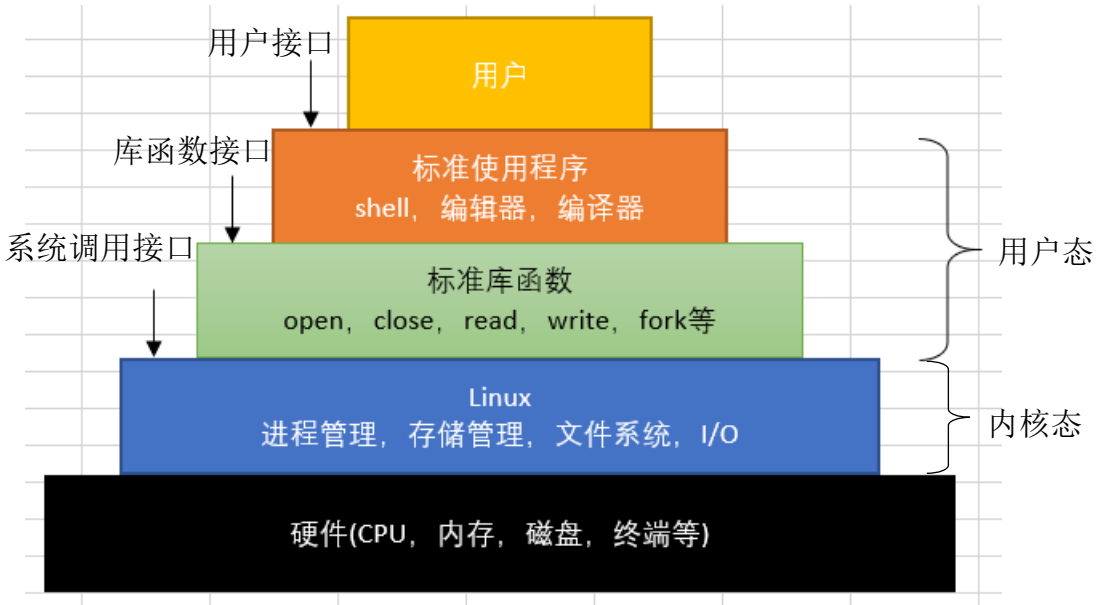


图 10-1 Linux 系统中的层次结构

2) Linux 具有三种不同的接口

- 真正的系统调用接口
- 库函数接口
- 由标准应用程序构成的接口。

3) shell

shell 也只是一个用户程序。它仅仅需要从键盘中读取数据，向显示屏输出数据和运行其他程序的能力。

当 shell 被启动的时候，它首先初始化自己，然后在屏幕上输出一个提示符 prompt，通常是一个美元的符号，并等待用户输入命令行

等用户输入第一个字后，搜索这个字代表的程序名字，然后调用 fork 命令创建一个子进程来运行它，子程序通过调用系统调用 exec 来执行用户命令，以 exec 函数的第一个参数命名的文件替换掉子进程原来的全部核心映像，此时 shell 将会挂起自己直至该程序运行完毕，之后尝试读入下一条命令。

```
while (TRUE) {
    type_prompt( );                          /*永远重复*/
    read_command(command, params);           /*在屏幕上显示提示符*/
                                             /*从键盘读取输入行*/
    pid = fork( );                           /*创建子进程*/
    if (pid < 0) {
        printf("Unable to fork 0");          /*错误状态*/
        continue;                            /*重复循环*/
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);            /*父进程等待子进程*/
    } else {
        execve(command, params, 0);          /*子进程执行操作*/
    }
}
```

图 10-2 一个高度简化的 shell

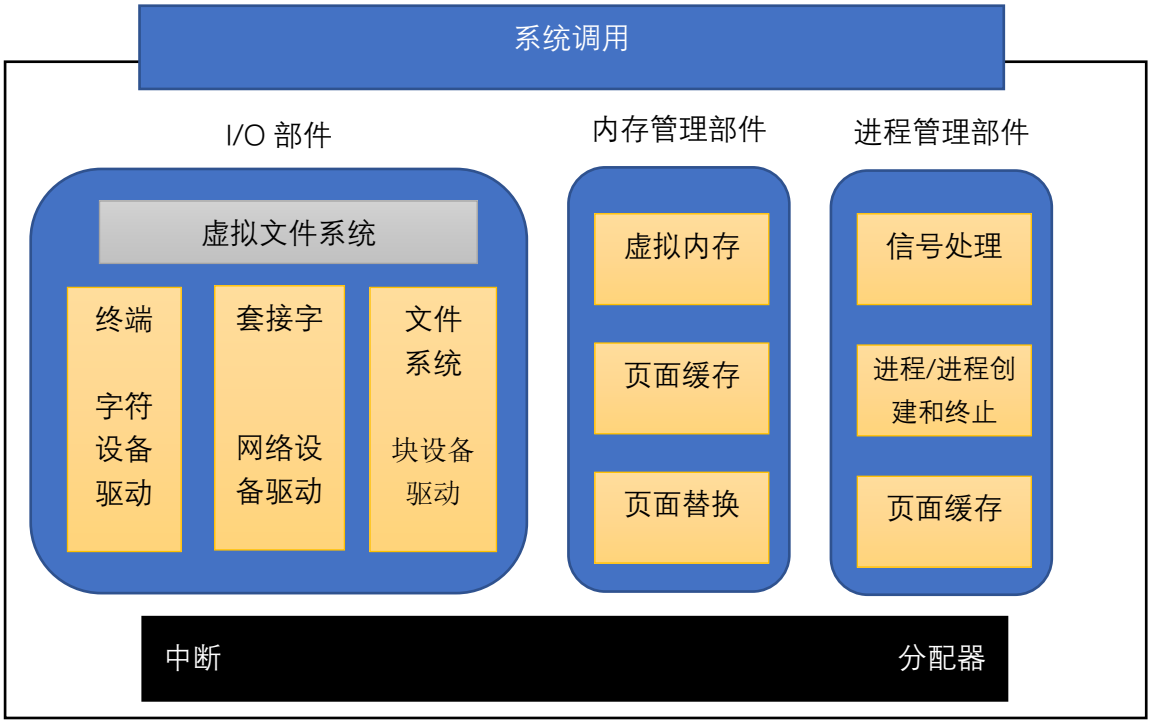


图 10-3 Linux 内核结构

10.3Linux 中的进程

1) 守护进程 daemon，后台程序；
计划任务 cron daemon 是一个典型的守护程序，他每分钟检查一下是否有工

作需要他去做。

2) 父进程和子进程

父进程和子进程都拥有自己的私有内存映像，如果在调用 fork 之后父进程修改了属于它的一些变量，这些变化对于子进程来说是不可见的，反之亦然。但是，父进程和子进程可以共享已经打开的文件。

事实上，父进程和子进程的内存映像，变量，寄存器以及其他所有的东西都是相同的，那问题就来了，如何区分这两个进程？秘密在于 fork 系统调用给子进程返回一个零值，给父进程返回一个非零值，这个非零值是子进程的进程标识符 Process Identifier，PID

```
pid = fork( );                  /*如果创建成功，则父进程pid>0*/
if (pid < 0) {
    handle_error( );            /*创建失败（比如内存或某些表溢出）*/
} else if (pid > 0) {
                                /*这里是父进程的代码*/
} else {
                                /*这里是子进程的代码*/
}
```

图 10-4 Linux 中的进程创建

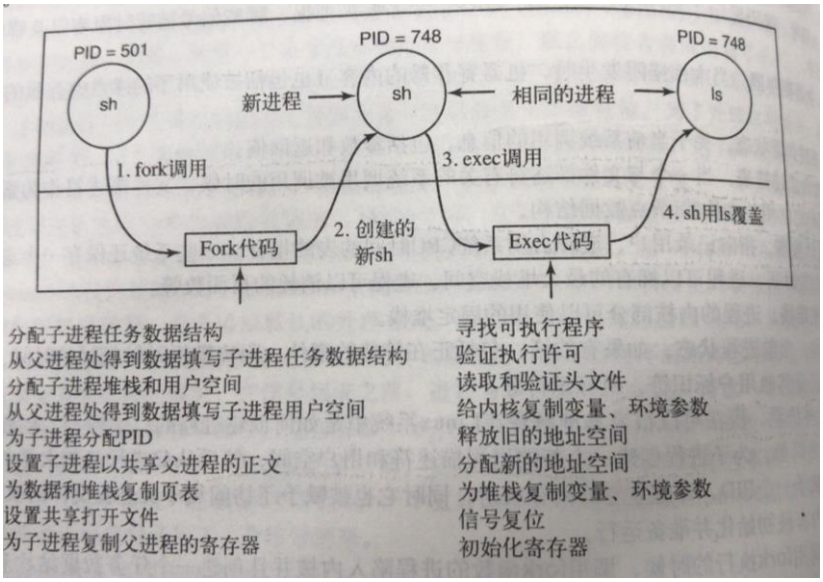


图 10-5 shell 执行命令 ls 的步骤

3) 任意进程都有属于自己的 PID，如果一个进程想知道自己的 PID，可以调用系统调用 getpid

4) 管道

Linux 系统中的进程可以通过一种消息传递的方式进行通讯，两个进程之间，可以建立通道，一个进程向通道中写入字节流，另一个读取字节流。

管道是同步的，因为如果一个进程试图从一个空的管道中读取数据，这个几次呢哼就会被挂起指导通道有数据为止。

5) 软中断

Linux 系统中另一种进程间通讯的方式，但是只能局限于进程组内的通讯。如果一个进程希望获得发送给他的信号，则必须指定一个信号处理函数，当信号过来的时候，控制立即切换到信号处理函数

6) 几个系统调用函数的说明

- pid=fork(); //创建一个与父进程一样的子进程；
- pid=waitpid(pid, &statloc, opts); //等待子进程终止；

  第一个参数可以使调用者等待某一个特定的子进程，如果为-1，说明任何一个子进程结束系统调用 waitpid 即可返回；第二个参数用来储存子进程退出状态（正常退出，异常退出和退出值）的变量地址。这个参数可以让父进程知道子进程现在所处的状态。第三个参数决定了如果没有子进程结束运行大的话，调用者是阻塞还是返回。
- pid=exec(name, argv, envp); //替换进程的核心映像，将参数和环境复制到内核，释放旧的地址空间和页表；

  第一个参数为待执行文件的文件名，指向参数数组的指针和指向环境数组的指针。如指令：

  cp file1 file2

  cp 的主程序包含一个函数声明

  main(argc, argv, envp)

  argc 表示命令行中包括程序名在内项的数目，这里是 3，第二个参数 argv 是一个指向数组的指针，argv[0]=cp; argv[1]= file1; argv[2]= file2; 第三个参数 envp 是一个指向环境的指针，比如主目录名等信息。

  当利用 exec 函数执行一个新的进程的时候，系统立刻会收到一个缺页中断的陷阱，使得第一个焊有代码的页面从可执行文件调入内存。通过这种方式，不需要预先加载任何东西，所以程序可以快速的开始运行。
- exit(status); //终止进程运行并返回状态值

  进程结束时会调用这个函数，它有一个参数，即退出状态，这个参数最后会传递给父进程调用 waitpid 函数的第二个参数——状态参数。状态参数的低字节部分包含着子进程的退出状态，0 表示正常结束，其他的值代表各种不同的错误。如果一个进程退出但是它的父进程没有在等待它，这个进程就进入僵死状态 zombie state。最后当父进程等待它时，这个进程才会结束。
- s=sigaction(sig, &act, &oldaction); //定义信号处理的动作
- 函数的额第一个参数是希望捕捉的动作，第二个参数是一个指向结构的指针，这个结构中包括一个指向信号处理函数的指针以及一些其他的位和标志，第三个参数是指向结构的指针，这个结构接收系统返回当前正在进行的信号处理的相关信息，有可能以后这些信息需要恢复。
- s=kill(pid, sig) //发送信号到进程。
- s=pause(); //挂起调用程序直至下一个信号出现。

7) Linux 中的内核是多线程的，并且它所拥有的与任何用户进程无关的内核级线程，这些内核级线程执行内核代码;

8) 虽然父进程和子进程都有自己的地址空间，但是为了节省内存，同时节省复

制内存的昂贵代价，现代 Linux 系统都是使用“欺骗”的手段来替代复制，它赋予子进程属于它的页表，但是这些页表都是指向父进程的页面，同时这些页面标记为可读。当进程试图写入数据的时候，不管是子进程还是父进程，都采用“写时复制”的技术。

9) Linux 中的线程

pid=clone(function, stack_ptr, sharing_flag, arg)　　//用来创建新线程

假如是在原来的进程中创建新的进程，则共享原来的地址空间；如果是在新的进程中创建新的进程，则新的线程立马获得完整的地址空间副本。stack_ptr 是私有堆栈的指针初始化。

一个进程中所有的线程都会拥有与该进程中第一个线程相同的 PID

10.4Linux 中的调度和启动顺序

首先认识到，Linux 中的线程是内核线程，所以 Linux 系统的调度是基于线程的，而不是进程的。

1) 为了调度，Linux 将线程分为三类

- 实时先入先出

  实时先入先出线程具有最高优先级，他不会被其他线程抢占，除非那是一个刚刚准备好的，拥有更高优先级的实时先入先出线程。

- 实时轮转

  实时轮转线程基本跟实时先入先出线程差不多，只是其都有一个时间量，当超过这个时间量之后该线程就可以被抢占。

- 分时；

简单说：https://zhidao.baidu.com/question/8840604.html

实时要求反应快，尽可能下达任务就能完成，所以实时系统多是单任务的，而且要去掉我们平时用的操作系统里冗余的部分，来达到尽可能快的要求。实时系统一般用在航天、导弹等这些需要快速计算的领域。

分时则是一种实用方式，指的是多个任务公用一个 cpu，只不过轮流来，你用一会，他用一会，由于转换的比较快，每个人都认为只有自己用似的。

分时系统就很多了，常见的 unix，linux，windows 都是分时的。

2)

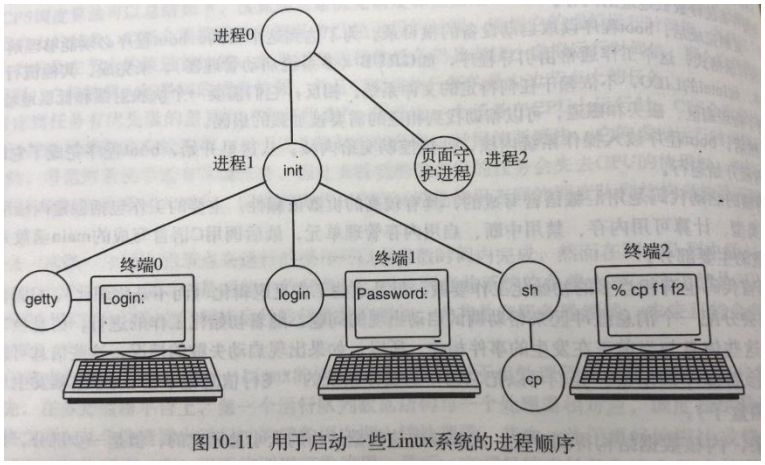


图10-11　用于启动一些Linux系统的进程顺序

图 10-6 用于启动一些 Linux 系统的进程顺序

10.5Linux 中的内存管理

1) 每个 Linux 进程都有一个地址空间，逻辑上由三段组成：代码（正文，包含了程序可执行代码的机器指令），数据（初始化的变量和未初始化的变量 BSS）和堆栈段。
对于未初始化的变量 BSS，为了避免分配一个全 0 的物理页框，在初始化的时候，Linux 就分配了一个静态零页面，即一个全 0 的写保护页面，当加载程序的时候 ，为初始化的数据区域被设置为指向该零页面。当一个金正真正要写这个区域的时候，再运用“写时复制”技术。
2) 共享代码段
数据段和栈段从来不共享，除非是同一个父进程下的子进程。

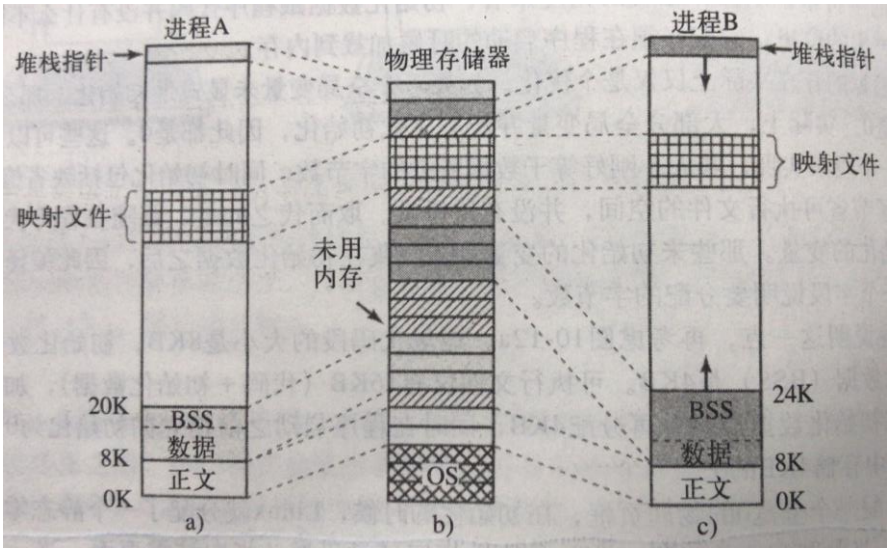


图 10-7 两个进程共享正文和映射文件

3) 内存映射文件
使用这个特性使我们可以把一个文件映射到进程空间的一部分而该文件就可以像位于内存中的字节数组一样被读写。但在不同的虚拟内存地址上。
这样做的好处是其中一个进程对文件的写可以被其他进程马上看到，实际上，通过映射一个临时文件（所有的进程退出后就被丢弃），这种机制可以为多进程共享内存提高带宽。
6) POSIX 没有给内存管理配置任何系统调用的标准，这有点奇怪，因为 malloc 这是这种系统调用的做法，在一些圈子里，这种方法被认为是推卸责任。
另外，将一组连续的虚拟页映射到任意一个文件的任意位置的表示法称作内存映射 memory mapping。Linux 提供一个称为 mmap 的系统调用，允许应用程序自己做内存映射。而 unmap 的系统调用，用来移除一个被映射的文件。
7) 简化共享
操作系统为每个线程安排一个单独私有的连续的虚拟页面，另外，对于共享的代码，操作系统允许不同的虚拟页映射到相同的物理页面作为这部分代码的副本，如内核和 C 标准库的副本。

8) Linux 的内存分为三个部分：内核+内存映射+被划分的页框
9) Linux 为了维护一个映射，该映射包含了所有系统物理内存的使用情况的信息，如区域，空闲页框等。为了做到这一点：
   - 页描述符

   首先需要维护一个页描述符数组，称为 mem_map，页描述符是 page 类型，而且系统中每一个物理页框都有一个页描述符。每个页描述符都有一个指针，页面非空闲的时候指向所属的地址空间，另有一堆指针可以使得他跟其他的描述符形成双向来拿表，来记录所有的空闲页框和一些其他的域。
   - 区域描述符

   由于物理内存被分成区域，Linux 为每个区域为维护一个区域描述符，包含了每个区域的内存的利用情况，页面的置换算法。
10) 内存分配机制

   Linux 支持多种内存分配机制，主要机制是页面分配器，它使用了著名的伙伴算法。它是一个支持以 2 为底的幂分割。

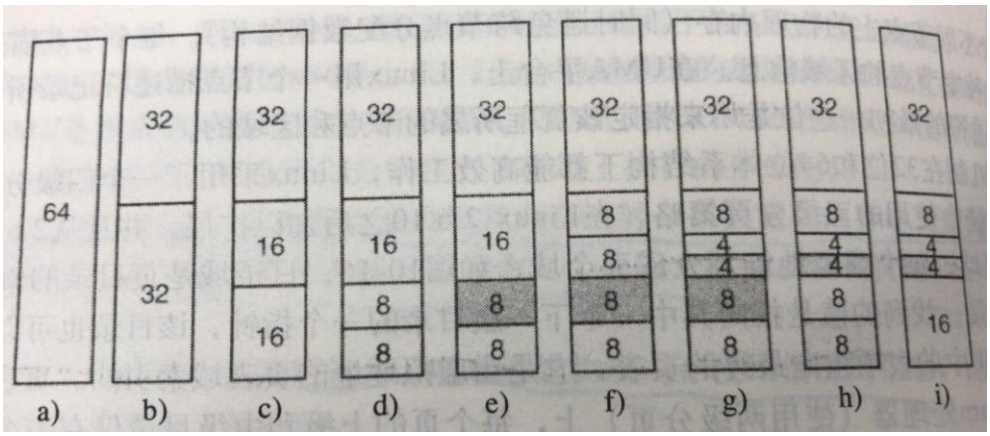


图 10-8 伙伴算法的操作

   但是，这种算法会导致大量的碎片。是想一下如果想要 65 页面的块，必须要请求一个 128 页的块。

   为了缓解这个问题，Linux 还有一个内存分配器，slab 分配器，它使用伙伴算法获得内存块，但是之后从其中切出 slab 更小的单元并且分别进行管理。
11) Linux 中的分页

   内存管理单元时一个页，并且几乎所有的内存管理部件以页为操作粒度。

   Linux 分页的思想：为了运行，一个进程并不需要完全在内存中，实际上所需要的是用户结构和页表。如果这些被换进内存，那么进程就被认为是“在内存中”，可以被调度使用。代码，数据和栈段的页面是动态载入的，仅仅是他们被引用的时候，如果用户结构和页表不存在内存中，直到交换器把他们载入内存才能运行。

   代码段和映射文件换页到他们各自在磁盘上的文件。而其他的都被换页到分页分区或者一个固定长度的分页文件，叫做交换区。
12) 页面置换算法——PFRA 页框回收算法

   四种页面：
   - 不可回收的 unreclaimable

     锁定页面，或者内核态栈

- 可交换的 swappable
  必须在回收之前写回交换区或者分页磁盘分区；
- 可同步的 syncable
  如果页面是 dirty 的必须写回到磁盘
- 可丢弃的 discardable
  可以被立即回收

10.6文件系统

1) POSIX 提供了一种更灵活的细粒度的机制，允许一个进程使用一个不可分割的操作对小到一个字节，大到整个文件加锁。加锁机制要求加锁者标志加锁的文件，开始位置以及要加锁的字节数。如果加锁成功，系统会在表格中说明要求加锁的字节如数据库的一条记录已被锁住。

10.7Android

1)

十一、 实例研究——Windows 8

11.1Windows8.1 的历史

1) 巨硬公司开发 Windows 系统大抵可以分为四个阶段：MS-DOS，基于 MS-DOS 的 Windows，基于 NT 的 Windows 和现代 Windows。
2) NT，new technology，重点是方便在不同的处理器之间切换以及安全性和可靠性，并兼容 MS-DOS 的 Windows。

3) 感觉硬件部分在学下去用处已经不大了，花了 20%时间感觉几乎达到了 80%的效果，后面主要关注网络基础的部分。