用户态和内核态

Linux 内核的用途

Linux 内核有 4 项工作：

1. **内存管理：**追踪记录有多少内存存储了什么以及存储在哪里
2. **进程管理：**确定哪些进程可以使用中央处理器（CPU）、何时使用以及持续多长时间
3. **设备驱动程序：**充当硬件与进程之间的调解程序/解释程序
4. **系统调用和安全防护：**从流程接受服务请求

通过**系统调用**将Linux整个体系分为用户态和内核态（或者说内核空间和用户空间）。

内核，它是一种**特殊的软件程序**，特殊在哪儿呢？**控制计算机的硬件资源，例如协调CPU资源，分配内存资源，并且提供稳定的环境供应用程序运行。**

用户态切换到内核态

1）系统调用

这是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完成工作。例如 `fork()` 就是执行了一个创建新进程的系统调用。系统调用的机制核心是使用了操作系统为用户特别开放的一个中断来实现，如 Linux 的 `int 80h` 中断。

（2）异常

当 cpu 在执行运行在用户态下的程序时，发生了一些没有预知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关进程中，也就是切换到了内核态，如缺页异常。

（3）外围设备的中断

当外围设备完成用户请求的操作后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条即将要执行的指令而转到与中断信号对应的处理程序去执行，如果前面执行的指令是用户态下的程序，那么转换的过程自然就会是 由用户态到内核态的切换。如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后边的操作等。

这三种方式是系统在运行时由用户态切换到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。从触发方式上看，切换方式都不一样，但从最终实际完成由用户态到内核态的切换操作来看，步骤是一样的，都相当于执行了一个中断响应的过程。系统调用实际上最终是中断机制实现的，而异常和中断的处理机制基本一致。

进程与线程

进程和线程的区别

线程具有许多传统进程所具有的特征，故又称为轻型进程(Light—Weight Process)或进程元；而把传统的进程称为重型进程(Heavy—Weight Process)，它相当于只有一个线程的任务。在引入了线程的操作系统中，通常一个进程都有若干个线程，至少包含一个线程。

根本区别：进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位

资源开销：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

包含关系：如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

内存分配：同一进程的线程共享本进程的地址空间和资源，而进程之间的地址空间和资源是相互独立的

影响关系：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

执行过程：每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

进程

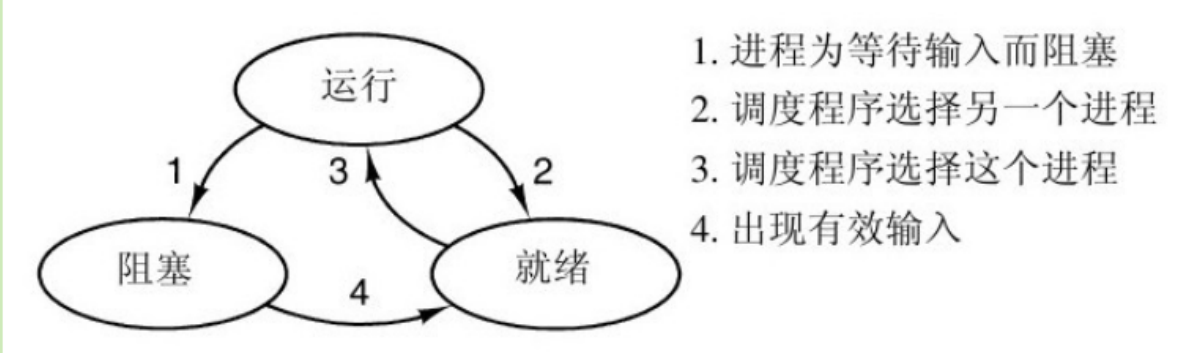
创建过程

在UNIX系统中，只有一个系统调用可以用来创建新进程：fork。这个系统调用会创建一个与调用进程相同的副本。**在调用了fork后，这两个进程（父进程和子进程）拥有相同的存储映像、同样的环境字符串和同样的打开文件。**通常，子进程接着执行execve或一个类似的系统调用，以修改其存储映像并运行一个新的程序。

在UNIX和Windows中，进程创建之后，父进程和子进程有各自不同的地址空间。如果其中某个进程在其地址空间中修改了一个字，这个修改对其他进程而言是不可见的。在UNIX中，子进程的初始地址空间是父进程的一个副本，但是这里涉及两个不同的地址空间，不可写的内存区是共享的（某些UNIX的实现使程序正文在两者间共享，因为它不能被修改）。但是，对于一个新创建的进程而言，确实有可能共享其创建者的其他资源，诸如打开的文件等。在Windows中，从一开始父进程的地址空间和子进程的地址空间就是不同的。

进程状态

- 运行态（该时刻进程实际占用CPU）
- 就绪态（可运行，但因为其他进程正在运行而暂时停止）
- 阻塞态（除非某种外部事件发生，否则进程不能运行）



进程实现

第一列中的字段与进程管理有关。其他两列分别与存储管理和文件管理有关。

进程管理 寄存器 程序计数器 程序状态字 堆栈指针 进程状态 优先级 调度参数 进程ID 父进程 进程组 信号 进程开始时间 使用的CPU时间 子进程的CPU时间 下次报警时间	存储管理 正文段指针 数据段指针 堆栈段指针	文件管理 根目录 工作目录 文件描述符 用户ID 组ID

处理中断的流程：

1. 硬件压入堆栈程序计数器等。
2. 硬件从中断向量装入新的程序计数器。
3. 汇编语言过程保存寄存器值。
4. 汇编语言过程设置新的堆栈。
5. C中断服务例程运行（典型地读和缓冲输入）。
6. 调度程序决定下一个将运行的进程。
7. C过程返回至汇编代码。
8. 汇编语言过程开始运行新的当前进程。

父子进程：

在unix/linux中，正常情况下，子进程是通过父进程创建的，子进程在创建新的进程。子进程的结束和父进程的运行是一个异步过程,即父进程永远无法预测子进程 到底什么时候结束。 当一个 进程完成它的工作终止之后，它的父进程需要调用wait()或者waitpid()系统调用取得子进程的终止状态。

僵尸进程和孤儿进程

僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用`wait`或`waitpid`获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。

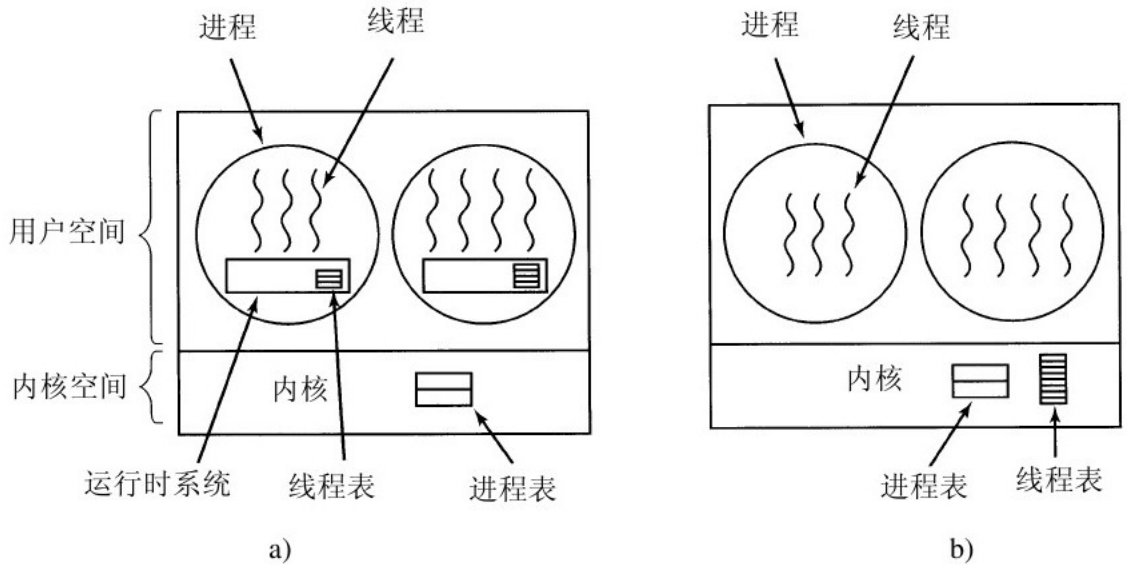
孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

线程崩溃影响进程吗?

线程没有独立的地址空间，如果崩溃，会发信号，如果没有错误处理的handler，OS一般直接杀死进程。就算是有handler了处理，一般也会导致程序崩溃，因为很有可能其他线程或者进程的数据被破坏了。

线程

线程实现



有两种主要的方法实现线程包：**在用户空间中和在内核中**。这两种方法互有利弊，不过混合实现方式也是可以的。

用户级线程

在用户空间管理线程时，**每个进程需要有其专用的线程表（thread table）**，用来跟踪该进程中的线程。这些表和内核中的进程表类似，不过它仅仅记录各个线程的属性，如每个线程的程序计数器、堆栈指针、寄存器和状态等。该线程表由运行时系统管理。当一个线程转换到就绪状态或阻塞状态时，在该线程表中存放重新启动该线程所需的信息，与内核在进程表中存放进程的信息完全一样。

优点：1）在用户空间管理线程时，每个进程需要有其专用的线程表（thread table），用来跟踪该进程中的线程。这些表和内核中的进程表类似，不过它仅仅记录各个线程的属性，如每个线程的程序计数器、堆栈指针、寄存器和状态等。该线程表由运行时系统管理。当一个线程转换到就绪状态或阻塞状态时，在该线程表中存放重新启动该线程所需的信息，与内核在进程表中存放进程的信息完全一样。2）它允许每个进程有自己定制的调度算法。例如，在某些应用程序中，那些有垃圾收集线程的应用程序就不用担心线程会在不合适的时刻停止，这是一个长处。3）用户级线程还具有较好的可扩展性，这是因为在内核空间中内核线程需要一些固定表格空间和堆栈空间，如果内核线程的数量非常大，就会出现问题。

缺点：1）如何实现阻塞系统调用？ 让该线程实际进行该系统调用是不可接受的，因为这会停止所有的线程。**使用线程的一个主要目标是，首先要允许每个线程使用阻塞调用，但是还要避免被阻塞的线程影响其他的线程。**2）如果某个程序调用或者跳转到了一条不在内存的指令上，就会发生**页面故障**，而操作系统将到磁盘上取回这个丢失的指令（和该指令的“邻居们”），这就称为页面故障。在对所需的指令进行定位和读入时，相关的进程就被阻塞。如果有一个线程引起页面故障，内核由于甚至不知道有线程存在，通常会把整个进程阻塞直到磁盘I/O完成为止，尽管其他的线程是可以运行的。

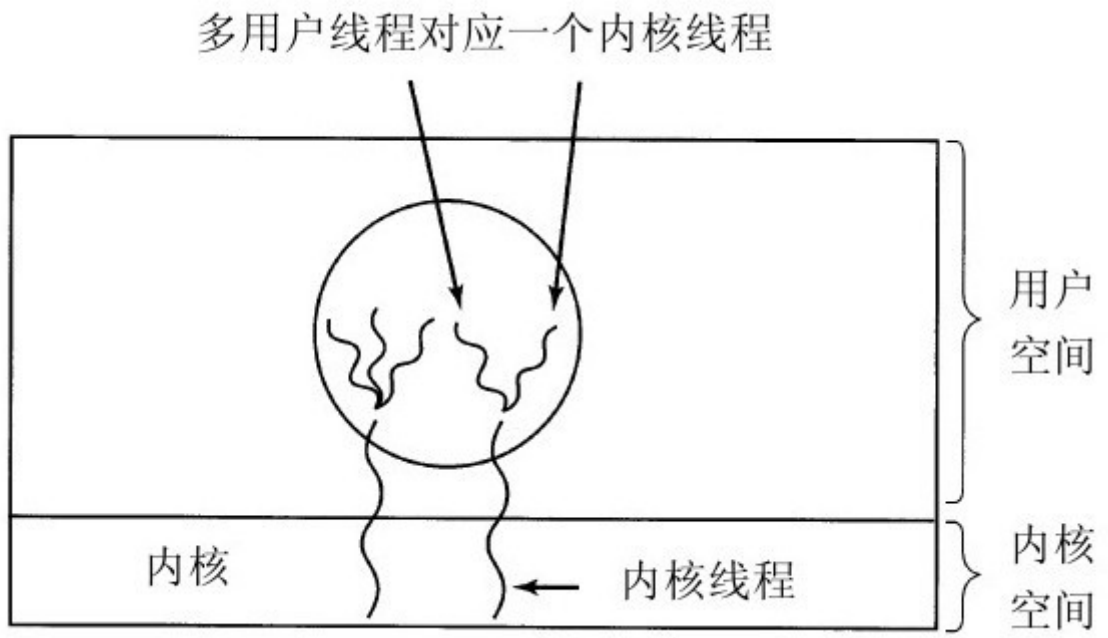
内核级线程

每个进程中也没有线程表。相反，在内核中有用来记录系统中所有线程的线程表。当某个线程希望创建一个新线程或撤销一个已有线程时，**它进行一个系统调用**，这个系统调用通过对线程表的更新完成线程创建或撤销工作。

内核的线程表保存了每个线程的寄存器、状态和其他信息。这些信息和在用户空间中（在运行时系统中）的线程是一样的，但是现在保存在内核中。这些信息是传统内核所维护的每个单线程进程信息（即进程状态）的子集。另外，内核还维护了传统的进程表，以便跟踪进程的状态。

优点： 1）所有能够**阻塞线程的调用都以系统调用的形式**实现，这与运行时系统过程相比，代价是相当可观的。当一个线程阻塞时，内核根据其选择，可以运行同一个进程中的另一个线程（若有一个就绪线程）或者运行另一个进程中的线程。而在用户级线程中，运行时系统始终运行自己进程中的线程，直到内核剥夺它的CPU（或者没有可运行的线程存在了）为止。

混合实现



使用内核级线程，然后将用户级线程与某些或者全部内核线程多路复用起来

采用这种方法，内核只识别内核级线程，并对其进行调度。其中一些内核级线程会被多个用户级线程多路复用。如同在没有多线程能力操作系统中某个进程中的用户级线程一样，可以创建、撤销和调度这些用户级线程。在这种模型中，每个内核级线程有一个可以轮流使用的用户级线程集合。

进程同步

进程通信

进程同步与进程通信很容易混淆，它们的区别在于：

- 进程同步：控制多个进程按一定顺序执行；
- 进程通信：进程间传输信息。

进程通信是一种手段，而进程同步是一种目的。也可以说，为了能够达到进程同步的目的，需要让进程进行通信，传输一些进程同步所需要的信息。

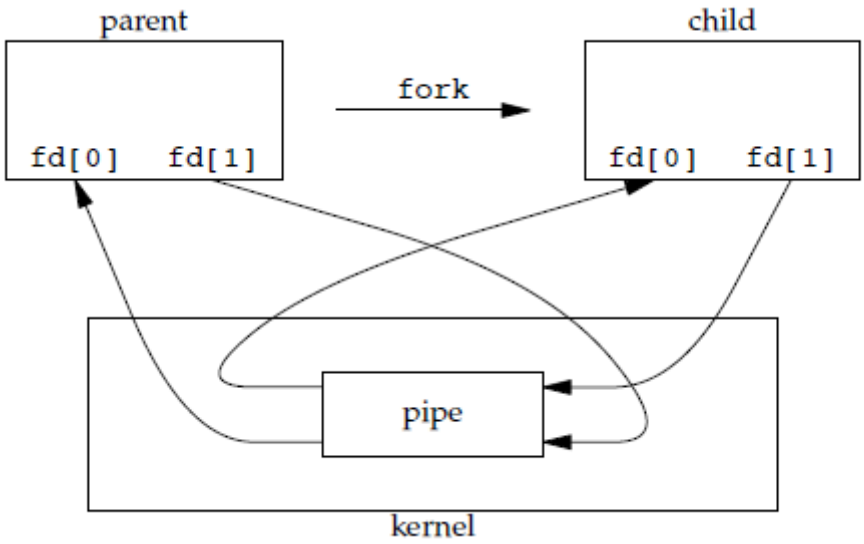
1. 管道

管道是通过调用 pipe 函数创建的，fd[0] 用于读，fd[1] 用于写。

```
#include <unistd.h>
int pipe(int fd[2]);
```

它具有以下限制：

- 只支持半双工通信（单向交替传输）；
- 只能在**父子进程或者兄弟进程**中使用。

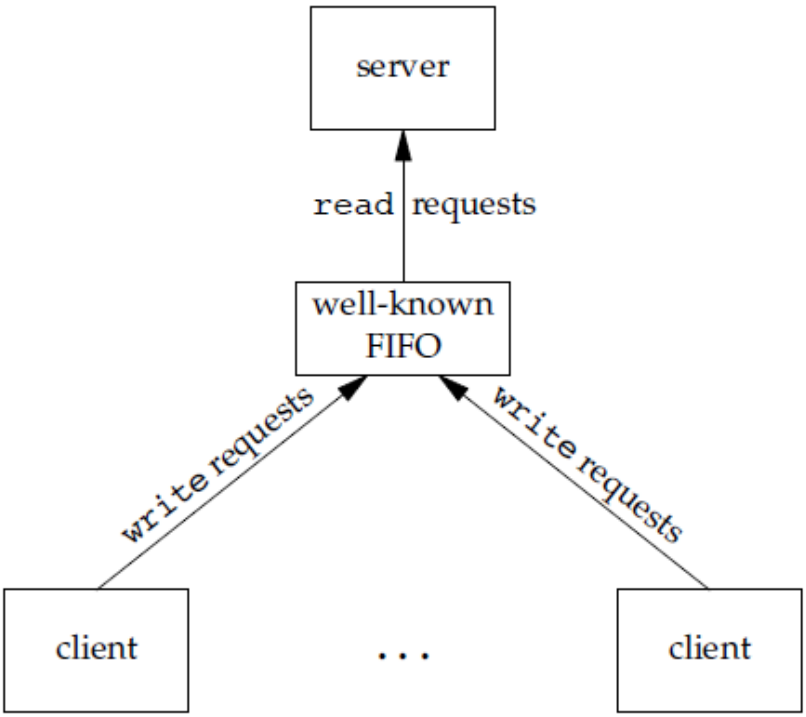


2. FIFO

也称为命名管道，去除了管道只能在父子进程中使用的限制。

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

FIFO 常用于客户-服务器应用程序中，FIFO 用作汇聚点，在客户进程和服务器进程之间传递数据。



3. 消息队列

相比于 FIFO，消息队列具有以下优点：

- 消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；
- 避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；
- 读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。

4. 信号量

它是一个计数器，用于为多个进程提供对共享数据对象的访问。

5. 共享存储

允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。

由于多个进程共享一段内存，因此需要依靠某种同步机制（如信号量）来达到进程间的同步及互斥。

多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用内存的匿名段。

6. 套接字

与其它通信机制不同的是，它可用于不同机器间的进程通信。

死锁

资源死锁的条件

死锁的四个条件：

- **互斥条件。**每个资源要么已经分配给了一个进程，要么就是可用的。
- **占有和等待条件。**已经得到了某个资源的进程可以再请求新的资源。
- **不可抢占条件。**已经分配给一个进程的资源不能强制性地被抢占，它只能被占有它的进程显式地释放。
- **环路等待条件。**死锁发生时，系统中一定有由两个或两个以上的进程组成的一条环路，该环路中的每个进程都在等待着下一个进程所占有的资源。

死锁发生时，以上四个条件一定是同时满足的。如果其中任何一个条件不成立，死锁就不会发生。

鸵鸟算法

最简单的解决方法是鸵鸟算法：把头埋到沙子里，假装根本没有问题发生。

如果死锁平均每5年发生一次，而每个月系统都会因硬件故障、编译器错误或者操作系统故障而崩溃一次，那么大多数的工程师不会以性能损失和可用性的代价去防止死锁。

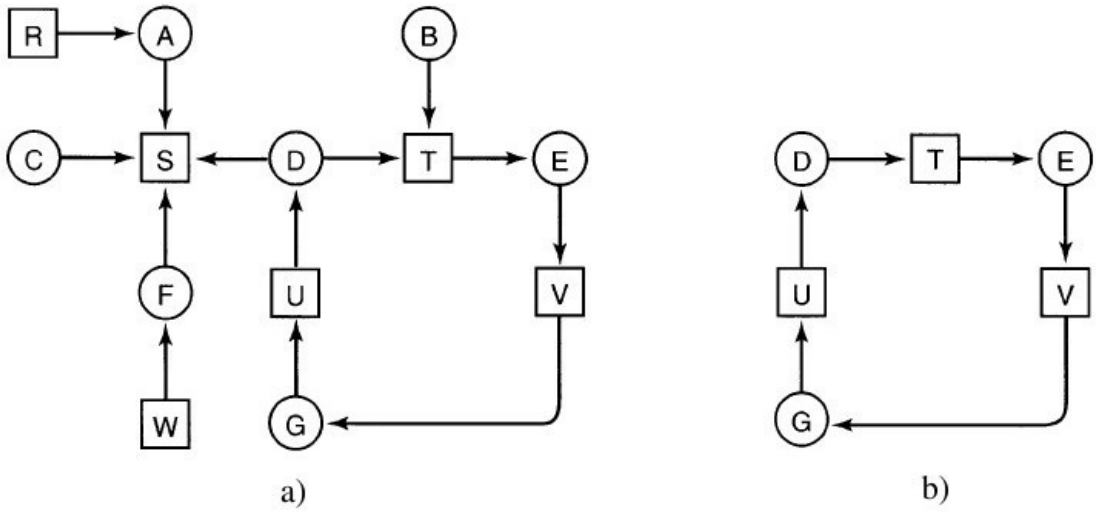
死锁检测和死锁恢复

使用这种方法，**系统不会阻止死锁的发生，而是允许死锁发生，当检测到死锁发生后，采取措施进行恢复。**

死锁检测

每种类型一个资源

构造一张资源分配图，如图。圆框为进程，方框为资源。



可以观察到图a中，包含了图b的环。在这个环中，我们可以看出进程D、E、G已经死锁。

判断图中是否有环可以是使用**深度优先搜索**和**拓扑排序**。

每种类型多个资源

构造资源矩阵，如图。

现有资源

$(E_1, E_2, E_3, \cdots, E_m)$

当前分配矩阵

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

第n行是进程n当前已分配到的资源

可用资源

$(A_1, A_2, A_3, \cdots, A_m)$

请求矩阵

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

第2行是进程2需要的资源

这四种数据结构之间有一个重要的恒等式。具体地说，某种资源要么已分配要么可用。这个结论意味着：

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

换言之，如果我们将所有已分配的资源j的数量加起来再和所有可供使用的资源数相加，结果就是该类资源的资源总数。

死锁检测算法就是基于向量的比较。我们定义向量A和向量B之间的关系为A≤B以表明A的每一个分量要么等于要么小于和B向量相对应的分量。从数学上来说，A≤B当且仅当且Ai ≤Bi (0≤i≤m)。

每个进程起初都是没有标记过的。算法开始会对进程做标记，进程被标记后就表明它们能够被执行，不会进入死锁。当算法结束时，任何没有标记的进程都是死锁进程。该算法假定了一个最坏情形：所有的进程在退出以前都会不停地获取资源。

死锁检测算法如下：

1. 寻找一个没有标记的进程Pi ，对于它而言R矩阵的第i行向量小于或等于A。
2. 如果找到了这样一个进程，那么将C矩阵的第i行向量加到A中，标记该进程，并转到第1步
3. 如果没有这样的进程，那么算法终止。
4. 算法结束时，所有没有标记过的进程（如果存在的话）都是死锁进程。

死锁恢复

1. 利用抢占恢复
2. 利用回滚恢复
3. 杀死进程

死锁避免

系统必须能够判断分配资源是否安全，并且只能在保证安全的条件下分配资源，避免死锁。

银行家算法对每一个请求进行检查，检查如果满足这一请求是否会达到安全状态。若是，那么就满足该请求；若否，那么就推迟对这一请求的满足。为了看状态是否安全，银行家看他是否有足够的资源满足某一个客户。如果可以，那么这笔投资认为是能够收回的，并且接着检查最接近最大限额的一个客户，以此类推。如果所有投资最终都被收回，那么该状态是安全的，最初的请求可以批准。

单个资源的银行家算法

多个资源的银行家算法

死锁预防

死锁预防，即破坏发生死锁的四个条件，使死锁不可能发生。

破坏互斥条件

几乎无法破坏，资源本来就是独享的。

破坏占有和等待条件

- 方法一： 一次分配所有资源，否则不分配
- 方法二：当请求资源时，先暂时释放其当前占用的所有资源，然后再尝试一次获得所需的全部资源。

破坏不可抢占条件

设定优先级，优先级高德进程可以抢占被其他低优先级进程已经持有的资源。

破坏环路等待条件

- 方法一：保证每一个进程在任何时刻只能占用一个资源，如果要请求另一个资源，它必须先释放第一个资源。这种方法局限性比较大。
- 方法二：给所有资源编号，进程在提出资源请求时，必须按照资源编号（升序）提出。这样，资源分配图中不会出现环。

存储管理

虚拟内存

虚拟内存的目的是**为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。**

为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。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，也不需要所有页都必须在物理内存中。当程序引用到不在物理内存中的页时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令。

从上面的描述中可以看出，虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB 的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。

分页

在没有虚拟内存的计算机上，系统直接将虚拟地址送到内存总线上，读写操作使用具有同样地址的物理内存字。在使用虚拟内存的情况下，虚拟地址不是被直接送到内存总线上，而是被送到内存管理单元（Memory Management Unit，MMU），MMU把虚拟地址映射为物理内存地址。

内存管理单元（MMU）管理着地址空间和物理内存的转换，其中的页表（Page table）存储着页（程序地址空间）和页框（物理内存空间）的映射表。

一个虚拟地址分成两个部分，一部分存储页面号，一部分存储偏移量。

下图的页表存放着 16 个页，这 16 个页需要用 4 个比特位来进行索引定位。例如对于虚拟地址（0010 000000000100），前 4 位是存储页面号 2，读取表项内容为（110 1），页表项最后一位表示是否存在于内存中，1 表示存在。后 12 位存储偏移量。这个页对应的页框的地址为 （110 000000000100）。

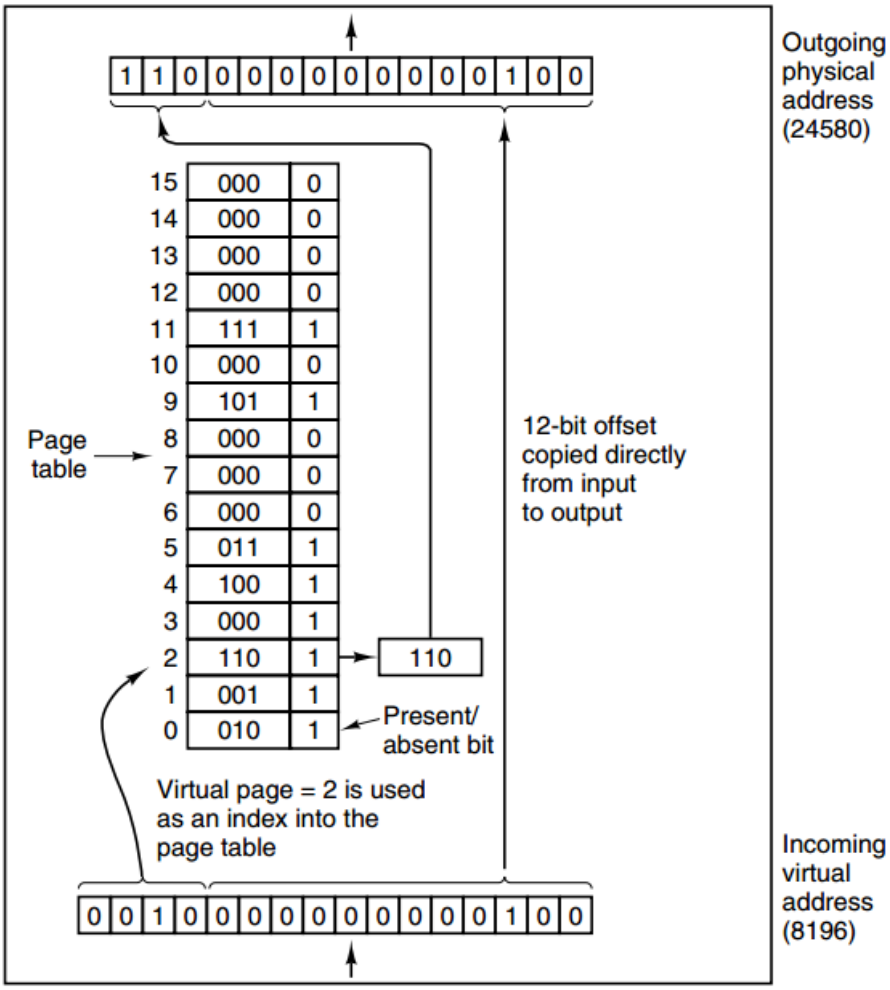


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

页面置换算法

最近未使用(NRU)

每个页面都有两个状态位：R 与 M，当页面被访问时设置页面的 R=1，当页面被修改时设置 M=1。其中 R 位会定时被清零。可以将页面分成以下四类：

- R=0，M=0
- R=0，M=1
- R=1，M=0
- R=1，M=1

当发生缺页中断时，NRU 算法随机地从类编号最小的非空类中挑选一个页面将它换出。

NRU 优先换出已经被修改的脏页面（R=0，M=1），而不是被频繁使用的干净页面（R=1，M=0）。

最近最久未使用（LRU）

LRU 将最近最久未使用的页面换出。

先进先出

选择换出的页面是最先进入的页面。

该算法会将那些经常被访问的页面换出，导致缺页率升高。

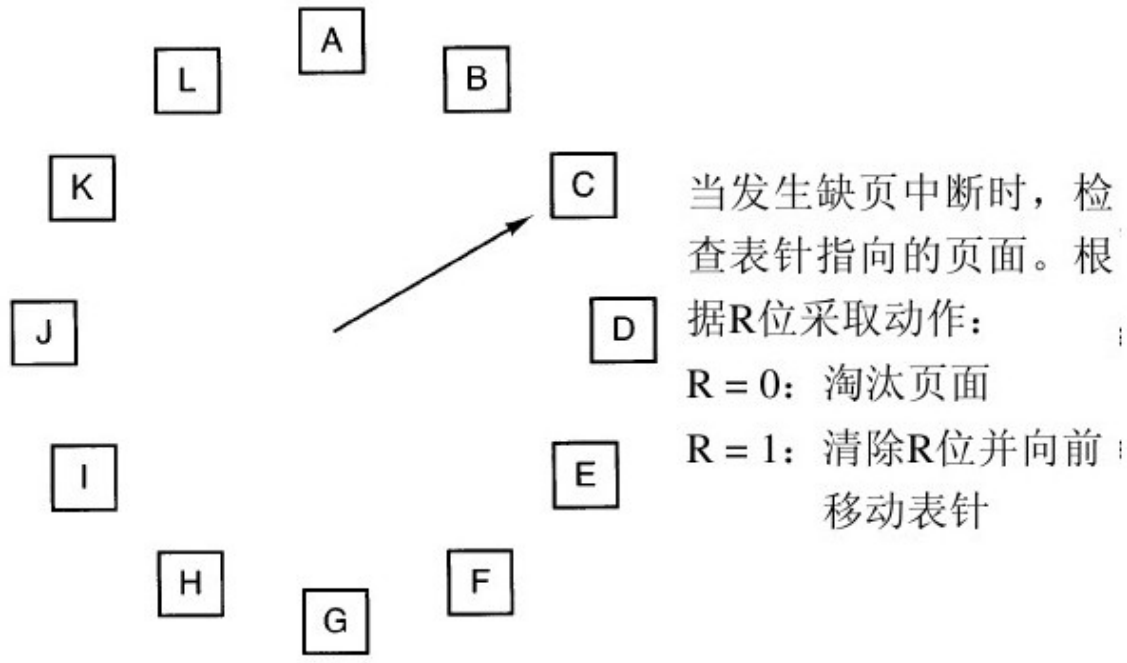
第二次机会算法

FIFO 算法可能会把经常使用的页面置换出去，为了避免这一问题，对该算法做一个简单的修改：

当页面被访问 (读或写) 时设置该页面的 R 位为 1。需要替换的时候，检查最老页面的 R 位。如果 R 位是 0，那么这个页面既老又没有被使用，可以立刻置换掉；如果是 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它就像刚装入的一样，然后继续从链表的头部开始搜索。

时钟算法

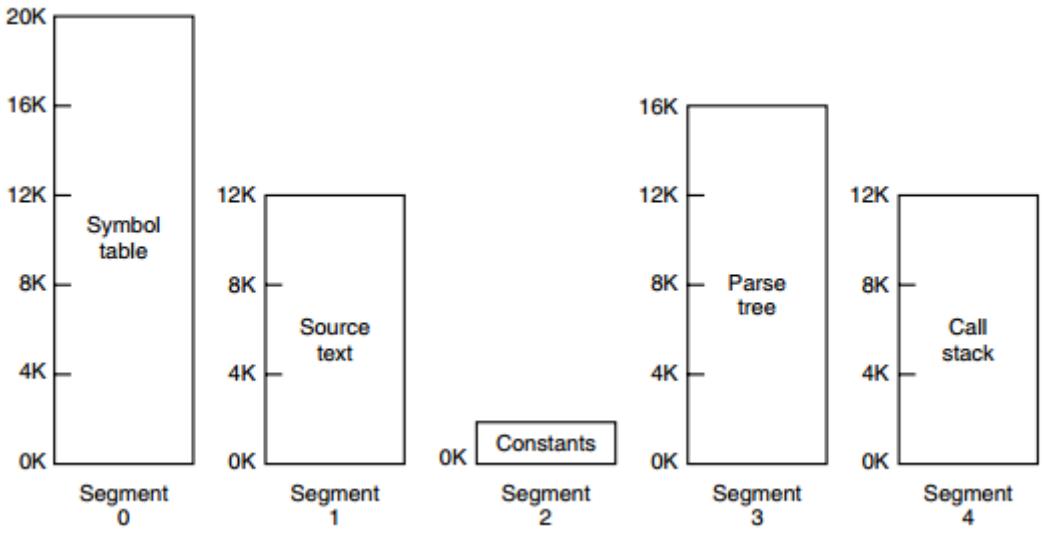
尽管第二次机会算法是一个比较合理的算法，但它经常要在链表中移动页面，既降低了效率又不是很有必要。一个更好的办法是把所有的页面都保存在一个类似钟面的环形链表中，一个表针指向最老的页面。



当发生缺页中断时，算法首先检查表针指向的页面，如果它的R位是0就淘汰该页面，并把新的页面插入这个位置，然后把表针前移一个位置；如果R位是1就清除R位并把表针前移一个位置，重复这个过程直到找到了一个R位为0的页面为止。

分段

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

程序的地址空间划分成多个拥有独立地址空间的段，每个段上的地址空间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

分页与分段比较

- 对程序员的透明性：分页透明，但是分段需要程序员显式划分每个段。
- 地址空间的维度：分页是一维地址空间，分段是二维的。
- 大小是否可以改变：页的大小不可变，段的大小可以动态改变。
- 出现的原因：分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要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。