计算机操作系统

本部分主要是笔者在学习现代操作系统和一些相关面试题所做的笔记，如果出现错误，希望大家指出！

现代操作系统阅读笔记

第一章 引论

1. 操作系统定义

操作系统是运行在内核态的软件，它执行两个基本上独立的任务。

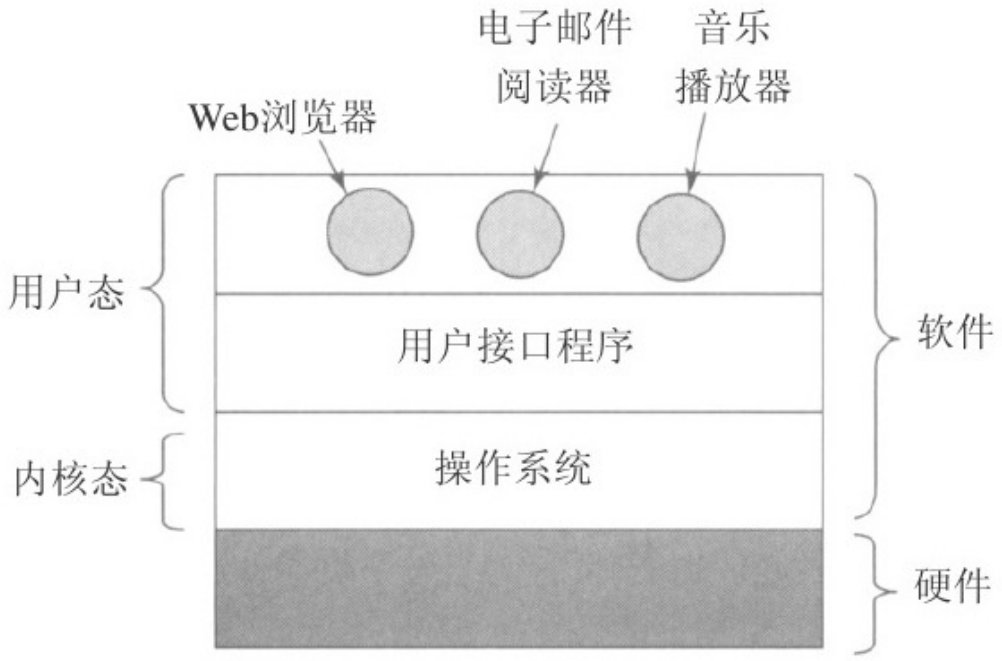
- 隐藏计算机底层硬件的实现，为用户及应用程序提供一个资源集的清晰抽象。
- 管理计算机硬件资源。

任何操作系统的核心是它可处理的系统调用集。这些系统带欧阳真实地说明了操作系统做的工作。

2. 计算机运行模式

多数计算机有两种运行模式：内核态和用户态。

软件中最基础的部分是操作系统，它运行在内核态。这内核态模式下，操作系统具有对所有硬件的完全访问权，可以执行机器能够运行的任何指令。软件的其余部分运行在用户态下，在用户态下，只能使用机器指令中的一个子集。



3. shell 与 GUI

用户与之交互的程序，基于文本的通常称为shell，而基于图标的则称为图形用户界面（GUI）。

它们并不是操作系统的一部分，它们是运行在用户态最低层次的用户接口程序

4. 对于抽象的理解

现代计算机系统中，大量使用了抽象这一概念。抽象是管理复杂性的一个关键。好的抽象可以把一个几乎不可能管理的任务划分为两个可管理的部分。其中第一部分是有关抽象的定义和实现，第二部分是随时用这些抽象解决问题。

以抽象的角度看操作系统，它的任务就是创建好的抽象，并实现和管理它所创建的抽象。

5. 多路复用资源方式

在时间上复用：当一种资源在时间上复用时，不同的程序或用户轮流使用它。

在空间上复用：每个客户得到资源的一部分。

6. I/O 设备的结构

I/O 设备一般包括两个部分：设备控制器和设备本身。控制器插在电路板上的一块芯片或一组芯片，这块电路板物理地控制芯片，它从操作系统接收命令。

控制器的任务是为操作系统提供一个简单的接口。每类设备控制器是不同的，所以需要不同的软件进行控制。专门与控制器对话，发出命令并接收响应的软件，称为设备驱动程序。为了使用设备驱动程序，必须要把设备驱动程序装入到操作系统中，这样它可在核心态中运行。

每个设备控制器都有少量的用于通信的寄存器，所有的寄存器的集合构成了I/O 空间。

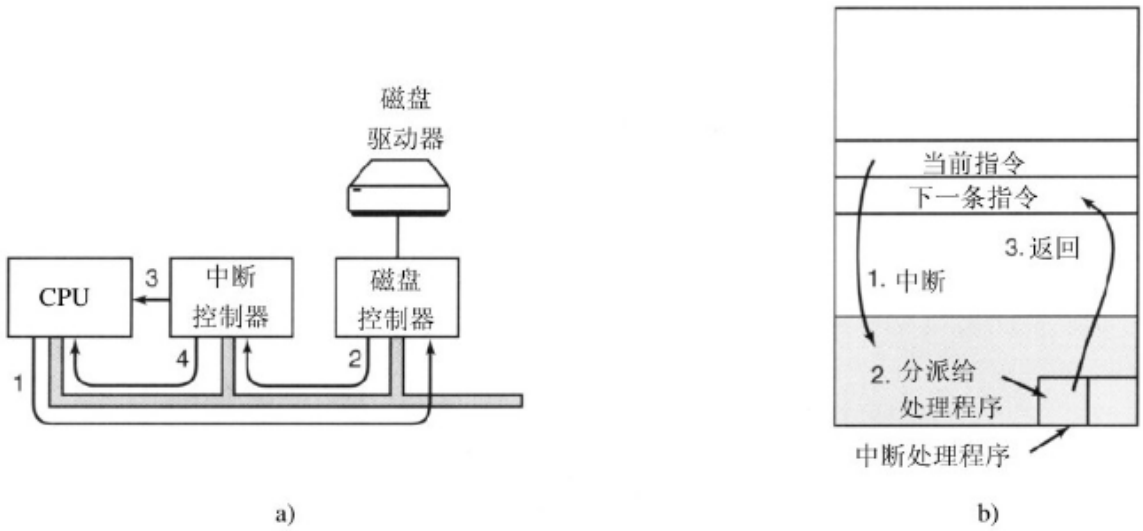
7. IDE 概念

IDE 表示集成驱动电子设备，是许多计算机的磁盘标准。

8. 实现输入输出的三种方式

第一种方式，用户程序发出一个系统调用，内核将其翻译成一个对应设备驱动程序的过程调用。然后设备驱动程序启动 I/O 并在一个连续不断的循环中检查该设备，看该设备是否完成了工作。当 I/O 结束后，设备驱动程序把数据送到指定的地方（若有此需要），并返回。然后操作系统将控制返回给调用者。这种方式称为忙等待（busy waiting），其缺点是要占据CPU，CPU 一直轮询设备直到对应的 I/O 操作完成。

第二种方式，设备驱动程序启动设备并且让该设备在操作完成时发出一个中断。设备驱动程序在这个时刻返回。操作系统接着在需要时阻塞调用者并安排其他工作进行。当设备驱动程序检测到该设备的操作完毕时，它发出一个中断通知操作完成。



第三种方式，为I/O使用一种特殊的直接存储器访问（Direct Memory Access，DMA）芯片，它可以控制在内存和某些控制器之间的位流，而无须持续的CPU干预。

9. CMOS 存储器

CMOS 存储器是易失性的。许多计算机利用 CMOS 存储器保持当前时间和日期。 CMOS 存储器和递增时间的时钟电路有一块小电池驱动，所以，即使计算机没有上电，时间也可以正确地更新。

10. USB 概念

USB 是通用串行总线，是用来将所有的慢速 I/O 设备，诸如键盘和鼠标，与计算机相连。USB 是一种集中式总线，其根设备每 1ms 轮询一次 I/O 设备，看是否有消息收发。所有的 USB 设备共享一个 USB 设备驱动器，于是就不需要为新的 USB 设备安装新的设备驱动器了。

11. 即插即用概念

在一般的计算机工作环境下，操作系统必须了解有些什么外部设备连接到计算机上，并对它们进行配置。这种需求导致了 Intel 和微软设计了一种名为即插即用的 I/O 系统。

在即插即用之前，每块 I/O 卡有一个固定的中断请求级别和用于其 I/O 寄存器的固定地址。即插即用所做的工作是，系统自动地收集有关 I/O 设备的信息，集中赋予 I/O 地址，然后通知每块卡所用的数值。

12. 计算机的启动

Pentium 的简要启动过程如下。在每个 Pentium 上有一块母板。在母板上有一个称为基本输入输出系统（Basic Input Output System，BIOS）的程序。在 BIOS 内有底层 I/O 软件，包括读键盘、写屏幕、进行磁盘 I/O 以及其他过程。现在这个程序存放在一块闪速 RAM 中，它是非可易失性的，但是在发现 BIOS 中有错时可以通过操作系统对它进行更新。

简要过程如下：

1. BIOS 开始运行。它首先检查计算机设备的状态信息是否正常。
2. 扫描并记录总线所连设备。
3. 依次搜索启动设备，导入操作系统。
4. 操作系统询问 BIOS ，获得配置信息，获取所有设备的驱动程序并调入内核。

5. 初始化有关表格，创建需要的任何背景进程，并在每个终端上启动登录程序或GUI。

13. 操作系统分类

大型机操作系统、服务器操作系统、多处理器操作系统、个人计算机操作系统、掌上计算机操作系统、嵌入式操作系统、传感器节点操作系统、实时操作系统、智能卡操作系统

14. 实时操作系统的基本概念

实时操作系统的特征是将时间作为关键参数。通常分为硬实时操作系统和软实时操作系统。

在硬实时操作系统中，某个规定的动作必须绝对地在规定的时刻（或规定的时间范围）发生。

在软实时操作系统中，偶尔违反最终时限是不希望的，但可以接受，并且不会引起任何实时性的损害。

15. UID

系统管理器授权每个进程使用一个给定的 UID 标识。每个被启动的进程都有一个启动该进程的用户 UID 。子进程与父进程拥有一样 UID。用户可以是某个组的成员，每个组也有一个 GID 标识。

16. 文件路径

在 UNIX 中，绝对路径名包含了从根目录到该文件的所有目录清单，它们之间用正斜线 `/` 隔开。最开始的正斜线标识这是从根目录开始的绝对路径。

在 MS-DOS 和 Windows 中，用反斜线 `\` 作为分隔符。

17. 文件系统安装

UNIX 一个重要概念是安装文件系统。几乎所有的个人计算机都有一个或多个光盘驱动器，可以插入 CD-ROM 和 DV D。它们几乎都有 USB 接口，可以插入 USB 存储棒（实际是固态磁盘驱动器）。为了提供一个出色的方式处理可移动介质，UNIX 允许把在 CD-ROM 或 DVD 上的文件系统接入到主文件树上。mount 系统调用允许把在 CD-ROM 上的文件系统连接到程序所希望的根文件系统上。

18. 特殊文件

提供特殊文件是为了使 I/O 设备看起来像文件一般。这样，就像使用系统调用读写文件一样，I/O 设备也可通过同样的系统调用进行读写。

有两类特殊文件：块特殊文件和字符特殊文件。

块特殊文件指那些由可随机存取的块组成的设备，如磁盘等。比如打开一个块特殊文件，然后读第4块，程序可以直接访问设备的第4块而不必考虑存放该文件的文件系统结构。

字符特殊文件用于打印机、调制解调器和其他接收或输出字符流的设备。按照惯例，特殊文件保存在 /dev 目录中。例如，/dev/lp是打印机。

19. 文件保护

UNIX 操作系统通过对每个文件赋予一个9位的二进制保护代码，对 UNIX 中的文件实现保护。该保护代码有三个3位字段，一个用于所有者，一个用于所有者同组（用户被系统管理员划分成组）中的其他成员，而另一个用于其他人。每个字段中有一位用于读访问，一位用于写访问，一位用于执行访问。这些位就是知名的 rwx 位。

20. 系统调用概念

如果一个进程正在用户态中运行一个用户程序，并且需要一个系统服务，比如从一个文件读数据，那么它就必须执行一个陷阱或系统调用指令，将控制转移到操作系统。操作系统接着通过参数检查，找出所需要的调用进程。然后，它执行系统调用，并把控制返回给在系统调用后面跟随着的指令。在某种意义上，进行系统调用就像进行一个特殊的过程调用，但是只有系统调用可以进入内核，而过程调用则不能。

21. POSIX

UNIX 有很多不兼容的版本，从而导致了混乱。为了能使编写的程序能够在任何版本的 UNIX 系统运行，IEEE提出了一个 UNIX 标准，称为 POSIX，目前大多数 UNIX 版本都支持他。 POSIX 标准定义了凡是 UNIX 必须支持的小型系统调用接口。

22. Windows Win32 API

Windows 和 UNIX 的主要差别在于编程方式。一个 UNIX 程序包括做各种处理的代码以及从事完成特定服务的系统调用。相反，一个 Windows 程序通常是一个事件驱动程序。其中主程序等待某些事件发生，然后调用一个过程处理该事件。

在 UNIX 中，系统调用（如read）和系统调用所使用的库过程（如read）之间几乎是一一对应的关系。换句话说，对于每个系统调用，差不多就涉及一个被调用的库过程。

在 Windows 中，情况就大不相同了。首先，库调用和实际的系统调用是几乎不对应的。微软定义了一套过程，称为应用编程接口（Application Program Interface，Win32 API），程序员用这套过程获得操作系统的服务。

Win32 并不是非常统一的或有一致的接口。其主要原因是由于 Win32 需要与早期的在 Windows 3.x 中使用的16位接口向后兼容。

Windows 中没有类似 UNIX 中的进程层次，所以不存在父进程和子进程的概念。在进程创建之后，创建者和被创建者是平等的。

23. 操作系统结构

单体结构、层次式结构、微内核、客户机-服务器模式、虚拟机、外核、

24. 微内核的概念

在微内核设计背后的思想是，为了实现高可靠性，将操作系统划分成小的、良好定义的模块，只有其中一个模块——微内核——运行在内核态上，其余的模块，由于功能相对弱些，则作为普通用户进程运行。特别地，由于把每个设备驱动和文件系统分别作为普通用户进程，这些模块中的错误虽然会使这些模块崩溃，但是不会使得整个系统死机。

25. 机制与策略分离原则

策略指的是做什么，机制指的是怎么做。例如一个比较简单的调度算法是，对每个进程赋予一个优先级，并让内核执行在具有最高优先级进程中可以运行的某个进程。这里，机制（在内核中）就是寻找最高优先级的进程并运行之。而策略（赋予进程以优先级）可以由用户态中的进程完成。在这个方式中，机制和策略是分离的，从而使系统内核变得更小。

26. make 程序

在 UNIX 系统中，有个名为 make 的程序（其大量的变体如 gmake、pmake 等），它读入 Makefile，该 Makefile 说明哪个文件与哪个文件相关。make 的作用是，在构建操作系统二进制码时，检查此刻需要哪个目标文件，而且对于每个文件，检查自从上次目标文件创建之后，是否有任何它依赖（代码和头文件）的文件已经被修改了。如果有，目标文件需要重新编译。在大型项目中，创建 Makefile 是一件容易出错的工作，所以出现了一些工具使该工作能够自动完成。

第二章 进程与线程

一、进程

1. 进程模型

在进程模型中，计算机上所有可运行的软件，通常也包括操作系统，被组织成若干顺序进程，简称进程。一个进程就是一个正在执行程序的实例，包括程序计数器、寄存器和变量的当前值。

由于CPU在各进程之间来回快速切换，所以每个进程执行其运算的速度是不确定的。而且当同一进程再次运行时，其运算速度通常也不可再现。所以，在对进程编程时决不能对时序做任何确定的假设。

2. 进程的创建

有4种主要事件导致进程的创建：

- **系统初始化**

  > 启动操作系统时，通常会创建若干个进程。其中有些是前台进程，也就是同用户（人类）交互并且替他们完成工作的那些进程。其他的是后台进程，这些进程与特定的用户没有关系，相反，却具有某些专门的功能。停留在后台处理诸如电子邮件、Web 页面、新闻、打印之类活动的进程称为守护进程

- **执行了正在运行的进程所调用的进程创建系统调用**

  > 一个正在运行的进程经常发出系统调用，以便创建一个或多个新进程协助其工作。在所要从事的工作可以容易地划分成若干相关的但没有相互作用的进程时，创建新的进程就特别有效果。

- **用户请求创建一个新进程**

  > 在交互式系统中，键入一个命令或者点（双）击一个图标就可以启动一个程序。这两个动作中的任何一个都会开始一个新的进程，并在其中运行所选择的程序。

- **一个批处理作业的初始化**

> 最后一种创建进程的情形仅在大型机的批处理系统中应用。用户在这种系统中（可能是远程地）提交批处理作业。在操作系统认为有资源可运行另一个作业时，它创建一个新的进程，并运行其输入队列中的下一个作业。

在 UNIX 系统中，只有一个系统调用可以用来创建新进程：fork。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。

在 Windows 中，一个Win32函数调用 CreateProcess 既处理进程的创建，也负责把正确的程序装入新的进程。

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。

3. 进程的终止

进程在创建之后，它开始运行，完成其工作。但永恒是不存在的，进程也一样。迟早这个新的进程会终止，通常由下列条件引起：

- **正常退出（自愿的）**

  > 多数进程是由于完成了它们的工作而终止。在 UNIX 中该调用是exit，而在 Windows 中，相关的调用是 ExitProcess 。

- **出错退出（自愿的）**

  > 进程终止的第二个原因是进程发现了严重错误。

- **严重错误（非自愿）**

  > 进程终止的第三个原因是由进程引起的错误，通常是由于程序中的错误所致。

- **被其他进程杀死（非自愿）**

  > 第四种终止进程的原因是，某个进程执行一个系统调用通知操作系统杀死某个其他进程。在 UNIX 中，这个系统调用是 kill 。在 Win32中对应的函数是 TerminateProcess 。

4. 进程的层次结构

某些系统中，当进程创建了另一个进程后，父进程和子进程就以某种形式继续保持关联。子进程自身可以创建更多的进程，组成一个进程的层次结构。

在 UNIX 中，进程和它的所有子女以及后裔共同组成一个进程组。

在 Windows 中没有进程层次的概念，所有的进程都是地位相同的。惟一类似于进程层次的暗示是在创建进程的时侯，父进程得到一个特别的令牌（称为句柄），该句柄可以用来控制子进程。但是，它有权把这个令牌传送给某个其他进程，这样就不存在进程层次了。

5. UNIX 启动时的初始化

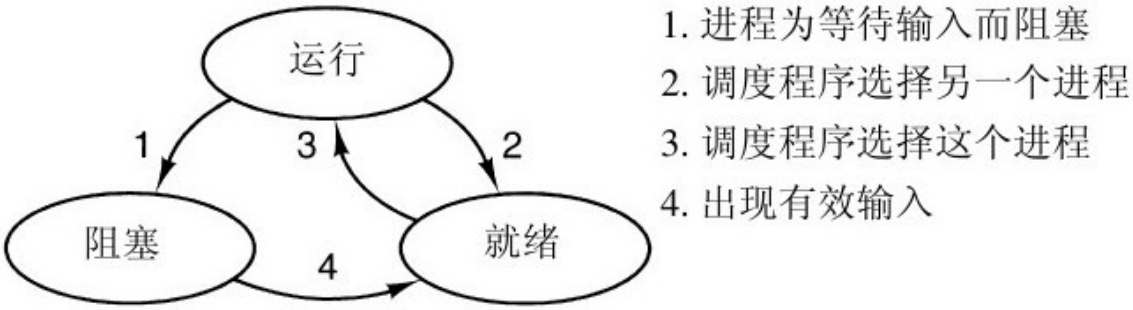
一个称为 init 的特殊进程出现在启动映像中。当它开始运行时，读入一个说明终端数量的文件。接着，为每个终端创建一个新进程。这些进程等待用户登录。如果有一个用户登录成功，该登录进程就执行一个 shell 准备接收命令。所接收的这些命令会启动更多的进程，以此类推。这样，在整个系统中，所有的进程都属于以 init 为根的一棵树。

6. 进程的状态

进程存在三种状态：

- 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）。

状态间的转化关系为



7. 进程的实现

为了实现进程模型，操作系统维护着一张表格（一个结构数组），即进程表。每个进程占用一个进程表项。（也可称为进程控制块。）该表项包含了进程状态的重要信息，包括程序计数器、堆栈指针、内存分配状况、所打开文件的状态、账号和调度信息，以及其他在进程由运行态转换到就绪态或阻塞态时必须保存的信息，从而保证该进程随后能再次启动，就像从未被中断过一样。

8. 多道程序设计模型

采用多道程序设计可以提高 CPU 的利用率。从概率的角度来看 CPU 的利用率。假设一个进程等待 I/O 操作的时间与其停留在内存中时间的比为　。当内存中同时有 n 个进程时，则所有 n 个进程都在等待 I/O（此时 CPU 空转）的概率是 p^n 。CPU 的利用率由下面的公式给出：

> CPU 利用率 = $1-p^n$

二、线程

1. 线程的使用原因

人们需要多线程的主要原因是，在许多应用中同时发生着多种活动。其中某些活动随着时间的推移会被阻塞。通过将这些应用程序分解成可以准并行运行的多个顺序线程，程序设计模型会变得更简单。

第二个关于需要多线程的理由是，由于线程比进程更轻量级，所以它们比进程更容易（即更快）创建，也更容易撤销。在许多系统中，创建一个线程较创建一个进程要快10～100倍。

需要多线程的第三个原因涉及性能方面的讨论。若多个线程都是 CPU 密集型的，那么并不能获得性能上的增强，但是如果存在着大量的计算和大量的 I/O 处理，拥有多个线程允许这些活动彼此重叠进行，从而会加快应用程序执行的速度。

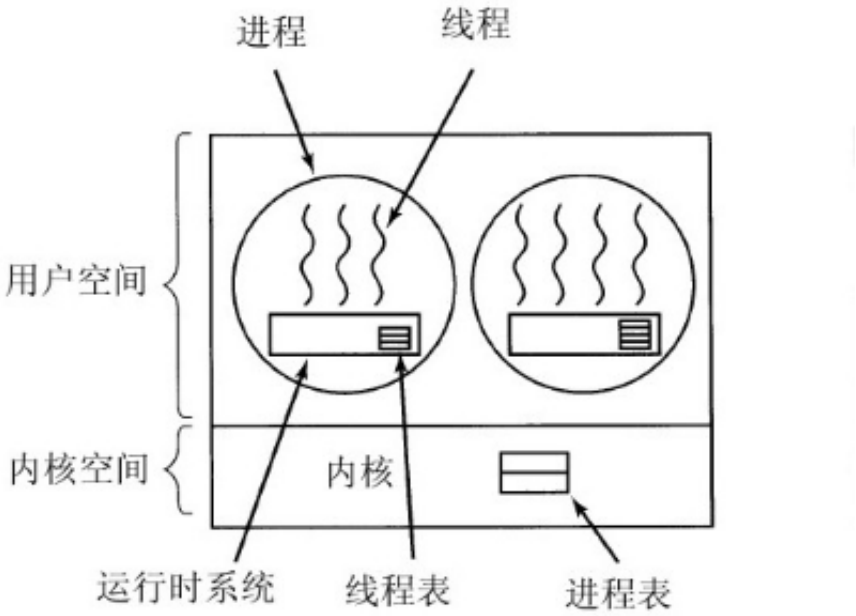
2. 线程模型

进程拥有一个执行的线程，通常简写为线程。在线程中有一个程序计数器，用来记录接着要执行哪一条指令。线程拥有寄存器，用来保存线程当前的工作变量。线程还拥有一个堆栈，用来记录执行历史，其中每一帧保存了一个已调用的但是还没有从中返回的过程。尽管线程必须在某个进程中执行，但是线程和它的进程是不同的概念，并且可以分别处理。进程用于把资源集中到一起，而线程则是在CPU上被调度执行的实体。

线程给进程模型增加了一项内容，即在同一个进程环境中，允许彼此之间有较大独立性的多个线程执行。在同一个进程中并行运行多个线程，是对在同一台计算机上并行运行多个进程的模拟。

3. 在用户空间中实现线程

把整个线程包放在用户空间中，内核对线程包一无所知。从内核角度考虑，就是按正常的方式管理，即单线程进程。线程在一个运行时系统的顶部运行，这个运行时系统是一个管理线程的过程的集合。我们已经见过其中的四个过程：pthread_create ，pthread_exit ，pthread_join 和 pthread_yield 。不过，一般还会有更多的过程。



在用户空间管理线程时，每个进程需要有其专用的线程表，用来跟踪该进程中的线程。这些表和内核中的进程表类似。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

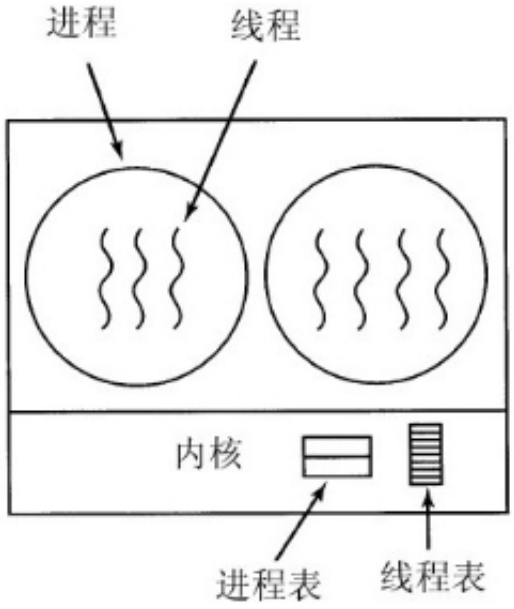
1. 用户级线程包可以在不支持线程的操作系统上实现。
2. 线程的切换可以在几条指令内完成。进行类似于这样的线程切换至少比陷入内核要快一个数量级（或许更多）。
3. 保存线程状态的过程和调度程序都只是本地过程，所以启动它们比进行内核调用效率更高。另一方面，不需要陷阱，不需要上下文切换，也不需要对内存高速缓存进行刷新，这就使得线程调度非常快捷。
4. 它允许每个进程有自己定制的调度算法。

缺点

1. 第一个问题是如何实现阻塞系统调用。假设在还没有任何击键之前，一个线程读取键盘。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。
2. 页面故障问题。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。
3. 如果一个线程开始运行，那么在该进程中的其他线程就不能运行，除非第一个线程自动放弃 CPU 。
4. 通常在经常发生线程阻塞的应用中才希望使用多个线程。对于那些基本上是CPU密集型而且极少有阻塞的应用程序而言，没有很大的意义。

4. 在内核中实现线程

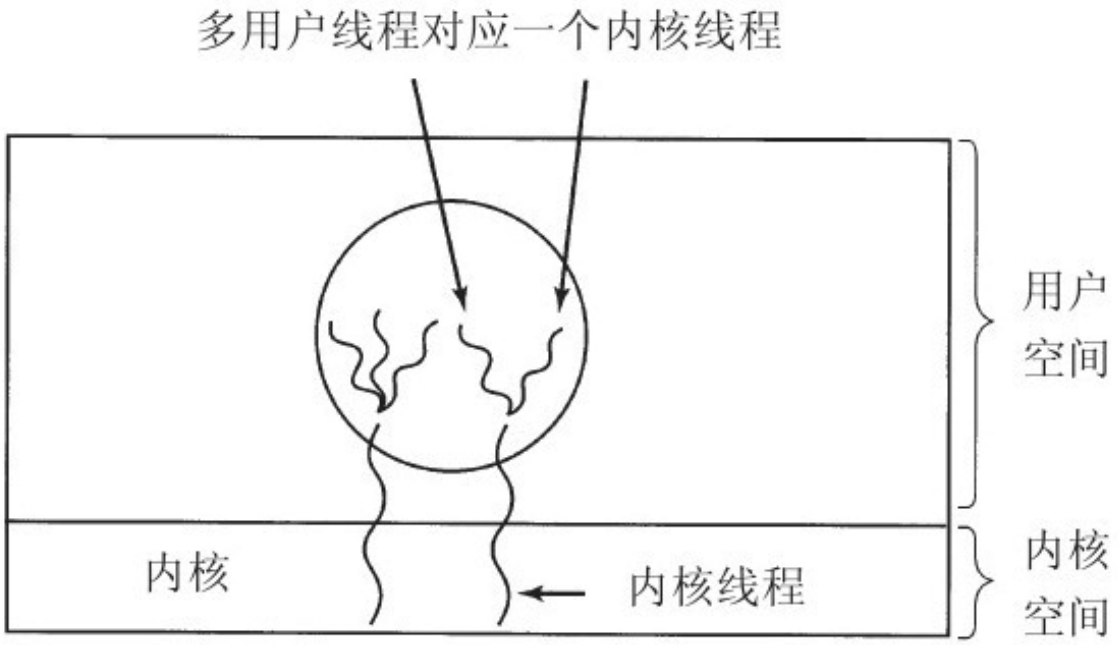
在内核中实现线程时，内核中有用来记录系统中所有线程的线程表。当某个线程希望创建一个新线程或撤销一个已有线程时，它进行一个系统调用，这个系统调用通过对线程表的更新完成线程创建或撤销工作。内核的线程表保存了每个线程的寄存器、状态和其他信息。



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。

5. 混合实现

人们已经研究了各种试图将用户级线程的优点和内核级线程的优点结合起来的方法。一种方法是使用内核级线程，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。

6. 调度程序激活机制

调度程序激活工作的目标是模拟内核线程的功能，但是为线程包提供通常在用户空间中才能实现的更好的性能和更大的灵活性。

使该机制工作的基本思路是，当内核了解到一个线程被阻塞之后，内核通知该进程的运行时系统，。内核通过在一个已知的起始地址启动运行时系统，从而发出了通知，这个机制称为上行调用。一旦如此激活，运行时系统就重新调度其线程。

调度程序激活机制的一个目标是作为上行调用的信赖基础，这是一种违反分层次系统内在结构的概念。

7. 弹出式线程

一个消息的到达导致系统创建一个处理该消息的线程，这种线程称为弹出式线程。

弹出式线程的关键好处是，由于这种线程相当新，没有历史这样，就有可能快速创建这类线程。对该新线程指定所要处理的消息。使用弹出式线程的结果是，消息到达与处理开始之间的时间非常短。

三、 进程间通信

进程间通信需要关注的三个问题：

1. 一个进程如何把信息传递给另一个。
2. 如何确保两个或更多的进程在关键活动中不会出现交叉。
3. 正确的顺序。

1. 竞争条件

两个或多个进程读写某些共享数据，而最后的结果取决于进程运行的精确时序，称为竞争条件。

2. 临界区

在某些时候进程可能需要访问共享内存或共享文件，或执行另外一些会导致竞争的操作。我们把对共享内存进行访问的程序片段称作临界区域或临界区。如果我们能够适当地安排，使得两个进程不可能同时处于临界区中，就能够避免竞争条件。

对于保证使用共享数据的并发进程能够正确和高效地进行协作，一个好的解决方案，需要满足以下4个条件：

- 任何两个进程不能同时处于其临界区。
- 不应对CPU的速度和数量做任何假设。
- 临界区外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。

3. 忙等待的互斥

（1） 屏蔽中断

在单处理器系统中，最简单的方法是使每个进程在刚刚进入临界区后立即屏蔽所有中断，并在就要离开之前再打开中断。屏蔽中断后，时钟中断也被屏蔽。CPU只有发生时钟中断或其他中断时才会进行进程切换，这样，在屏蔽中断之后CPU将不会被切换到其他进程。于是，一旦某个进程屏蔽中断之后，它就可以检查和修改共享内存，而不必担心其他进程介入。

缺点：

1. 若一个进程屏蔽中断后不再打开中断，整个系统可能会因此终止。
2. 如果系统是多处理器（有两个或可能更多的处理器），则屏蔽中断仅仅对执行disable 指令的那个CPU 有效。其他 CPU 仍将继续运行，并可以访问共享内存。

但是对内核来说，当它在更新变量或列表的几条指令期间将中断屏蔽是很方便的。

所以结论是：屏蔽中断对于操作系统本身而言是一项很有用的技术，但对于用户进程则不是一种合适的通用互斥机制。

（2） 锁变量

设想有一个共享（锁）变量，其初始值为0。当一个进程想进入其临界区时，它首先测试这把锁。如果该锁的值为0，则该进程将其设置为1并进入临界区。若这把锁的值已经为1，则该进程将等待直到其值变为0。于是，0就表示临界区内没有进程，1表示已经有某个进程进入临界区。

缺点：锁变量的读写不是原子操作，可能被其他进程中断

假设一个进程读出锁变量的值并发现它为0，而恰好在它将其值设置为1之前，另一个进程被调度运行，将该锁变量设置为1。当第一个进程再次能运行时，它同样也将该锁设置为1，则此时同时有两个进程进入临界区中。

（3）严格轮换法

定义一个整型变量 turn ，初始值为0，用于记录轮到哪个进程进入临界区，并检查或更新共享内存。开始时，进程0检查 turn ，发现其值为0，于是进入临界区。进程1也发现其值为0，所以在一个等待循环中不停地测试 turn ，看其值何时变为1。连续测试一个变量直到某个值出现为止，称为忙等待。

```
while (TRUE) {
    while (turn != 0)        /* 循环 */
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

a)

```
while (TRUE) {
    while (turn != 1)        /* 循环 */
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

b)

图 2-23 临界区问题的一种解法（在两种情况下请注意分号终止了while语句）：a)进程0；b)进程1

只有在有理由认为等待时间是非常短的情形下，才使用忙等待。用于忙等待的锁，称为自旋锁（spin lock）。

缺点：

1. 采用忙等待的方式，会浪费 CPU 时间。
2. 该方案要求两个进程严格地轮流进入它们的临界区，会造成一个临界区外运行的进程阻塞其他进程的情况。

（4）Peterson 解法

在使用共享变量（即进入其临界区）之前，各个进程使用其进程号0或1作为参数来调用 enter_region 。该调用在需要时将使进程等待，直到能安全地进入临界区。在完成对共享变量的操作之后，进程将调用 leave_region ，表示操作已完成，若其他的进程希望进入临界区，则现在就可以进入。

```
#define FALSE  0
#define TRUE   1
#define N      2                                    /* 进程数量 */

int turn;                                           /* 现在轮到谁？ */
int interested[N];                                  /* 所有值初始化为0（FALSE） */

void enter_region(int process);                     /* 进程是0或1 */
{
    int other;                                      /* 其他进程号 */

    other = 1 - process;                            /* 另一方进程*/
    interested[process] = TRUE;                     /* 表明所感兴趣的 */
    turn = process;                                 /* 设置标志 */
    while (turn == process && interested[other] == TRUE); /*空语句 */
}

void leave_region(int process)                      /* 进程：谁离开？ */
{
    interested[process] = FALSE;                    /* 表示离开临界区 */
}
```

（5）TSL 指令

TSL 指令是硬件支持的一种方案，称为测试并加锁，它将一个内存字 lock 读到寄存器 RX 中，然后在该内存地址上存一个非零值。

读字和写字操作保证是不可分割的，即该指令结束之前其他处理器均不允许访问该内存字。执行 TSL 指令的 CPU 将锁住内存总线，以禁止其他 CPU 在本指令结束之前访问内存。因此不会出现前面第二种方法锁变量的问题。

```
enter_region:
    TSL REGISTER,LOCK          | 复制锁到寄存器并将锁设为1
    CMP REGISTER,#0            | 锁是零吗？
    JNE enter_region           | 若不是零，说明锁已被设置，所以循环
    RET                        | 返回调用者，进入了临界区

leave_region:
    MOVE LOCK,#0               | 在锁中存入0
    RET                        | 返回调用者
```

为了使用 TSL 指令，要使用一个共享变量 lock 来协调对共享内存的访问。当 lock 为0时，任何进程都可以使用 TSL 指令将其设置为1，并读写共享内存。当操作结束时，进程用一条普通的 move 指令将 lock 的值重新设置为0。

一个可替代 TSL 的指令是 XCHG ，它原子性地交换了两个位置的内容，它本质上与 TSL 的解决办法一样。所有的Intel x86 CPU 在低层同步中使用 XCHG 指令。

```
enter_region:
        MOVE REGISTER,#1            | 在寄存器中放一个1
        XCHG REGISTER,LOCK          | 交换寄存器与锁变量的内容
        CMP REGISTER,#0             |锁是零吗？
        JNE enter_region            | 若不是零，说明锁已被设置，因此循环
        RET                         | 返回调用者，进入临界区

leave_region:
        MOVE LOCK,#0                | 在锁中存入0
        RET                         | 返回调用者
```

缺点：

1. 采用忙等待的方式，会浪费 CPU 时间。

4. 睡眠与唤醒

Peterson 解法和 TSL 或 XCHG 解法都是正确的，但它们都有忙等待的缺点。这种方法不仅浪费了CPU时间，而且还可能引起预想不到的结果。

我们可以使用睡眠与唤醒的机制，使它们在无法进入临界区时将阻塞，而不是忙等待。

最简单的是 sleep 和 wakeup 。 sleep 是一个将引起调用进程阻塞的系统调用，即被挂起，直到另外一个进程将其唤醒。wakeup 调用有一个参数，即要被唤醒的进程。

缺点：

参考生产者-消费者问题，发给一个（尚）未睡眠进程的 wakeup 信号会出现丢失，从而出现生产者和消费者同时睡眠的情况。

一种快速的弥补方法是修改规则，加上一个唤醒等待位。当一个 wakeup 信号发送给一个清醒的进程信号时，将该位置1。随后，当该进程要睡眠时，如果唤醒等待位为1，则将该位清除，而该进程仍然保持清醒。但原则上讲，这并没有从根本上解决问题。

5. 信号量

信号量是一个整型变量用来累计唤醒次数，供以后使用。一个信号量的取值可以为0（表示没有保存下来的唤醒操作）或者为正值（表示有一个或多个唤醒操作）。

对信号量一共有两种操作：down 和 up （分别为一般化后的 sleep 和 wakeup ）。

对一信号量执行 down 操作，则是检查其值是否大于0。若该值大于0，则将其值减1（即用掉一个保存的唤醒信号）并继续；若该值为0，则进程将睡眠，而且此时down操作并未结束。

对一信号量执行 up 操作，会对信号量的值增1。如果一个或多个进程在该信号量上睡眠，信号量的值仍旧是0，但在其上睡眠的进程会被唤醒一个。

检查数值、修改变量值以及可能发生的睡眠和唤醒操作均作为一个单一的、不可分割的原子操作完成。所谓原子操作，是指一组相关联的操作要么都不间断地执行，要么都不执行。

6. 互斥量

如果不需要信号量的计数能力，有时可以使用信号量的一个简化版本，称为互斥量（mutex）。互斥量仅仅适用于管理共享资源或一小段代码。由于互斥量在实现时既容易又有效，这使得互斥量在实现用户空间线程包时非常有用。

互斥量是一个可以处于两态之一的变量：解锁和加锁。当一个线程（或进程）需要访问临界区时，它调用 mutex_lock 。如果该互斥量当前是解锁的（即临界区可用），此调用成功，调用线程可以自由进入该临界区。另一方面，如果该互斥量已经加锁，调用线程被阻塞，直到在临界区中的线程完成并调用 mutex_unlock 。如果多个线程被阻塞在该互斥量上，将随机选择一个线程并允许它获得锁。

```
mutex_lock:
    TSL REGISTER,MUTEX      | 将互斥信号量复制到寄存器，并且将互斥信号量置为1
    CMP REGISTER,#0         | 互斥信号量是0吗？
    JZE ok                  | 如果互斥信号量为0，它被解锁，所以返回
    CALL thread_yield       | 互斥信号量忙；调度另一个线程
    JMP mutex_lock          | 稍后再试
ok: RET                     | 返回调用者；进入临界区

mutex_unlock:
    MOVE MUTEX,#0           | 将mutex置为0
    RET                     | 返回调用者
```

enter_region 和 mutex_lock 的代码很相似，但有一个关键的区别。

当 enter_region 进入临界区失败时，它始终重复测试锁（忙等待）。实际上，由于时钟超时的作用，会调度其他进程运行。这样迟早拥有锁的进程会进入运行并释放锁。

在（用户）线程中，情形有所不同，因为没有时钟停止运行时间过长的线程。结果是通过忙等待的方式来试图获得锁的线程将永远循环下去，决不会得到锁，因为这个运行的线程不会让其他线程运行从而释放锁。因此当 mutex_lock 取锁失败时，它调用 thread_yield 将 CPU 放弃给另一个线程。这样，就没有忙等待。在该线程下次运行时，它再一次对锁进行测试。

7. 条件变量

条件变量允许线程由于一些未达到的条件而阻塞。

与条件变量相关的最重要的两个操作是 pthread_cond_wait 和 pthread_cond_signal 。前者阻塞调用线程直到另一其他线程向它发信号（使用后一个调用）。

条件变量（不像信号量）不会存在内存中。如果将一个信号量传递给一个没有线程在等待的条件变量，那么这个信号就会丢失。

8. 管程

管程是一种高级同步原语，管程有一个很重要的特性，即任一时刻管程中只能有一个活跃进程，这一特性使管程能有效地完成互斥。

当一个进程调用管程过程时，该过程中的前几条指令将检查在管程中是否有其他的活跃进程。如果当一个进程调用管程过程时，该过程中的前几条指令将检查在管程中是否有其他的活跃进程。如果

管程提供了一种实现互斥的简便途径，通过临界区互斥的自动化，管程比信号量更容易保证并行编程的正确性。

9. 消息传递

这种进程间通信的方法使用两条原语 send 和 receive ，它们像信号量而不像管程，是系统调用而不是语言成分。

前一个调用向一个给定的目标发送一条消息，后一个调用从一个给定的源（或者是任意源，如果接收者不介意的话）接收一条消息。如果没有消息可用，则接收者可能被阻塞，直到一条消息到达，或者，带着一个错误码立即返回。

10. 屏障

在有些应用中划分了若干阶段，并且规定，除非所有的进程都就绪准备着手下一个阶段，否则任何进程都不能进入下一个阶段。可以通过在每个阶段的结尾安置屏障来实现这种行为。当一个进程到达屏障时，它就被屏障阻拦，直到所有进程都到达该屏障为止。

四、调度

当计算机系统是多道程序设计系统时，通常就会有多个进程或线程同时竞争CPU。只要有两个或更多的进程处于就绪状态，这种情形就会发生。如果只有一个CPU可用，那么就必须选择下一个要运行的进程。在操作系统中，完成选择工作的这一部分称为调度程序，该程序使用的算法称为调度算法。

1. 何时调度

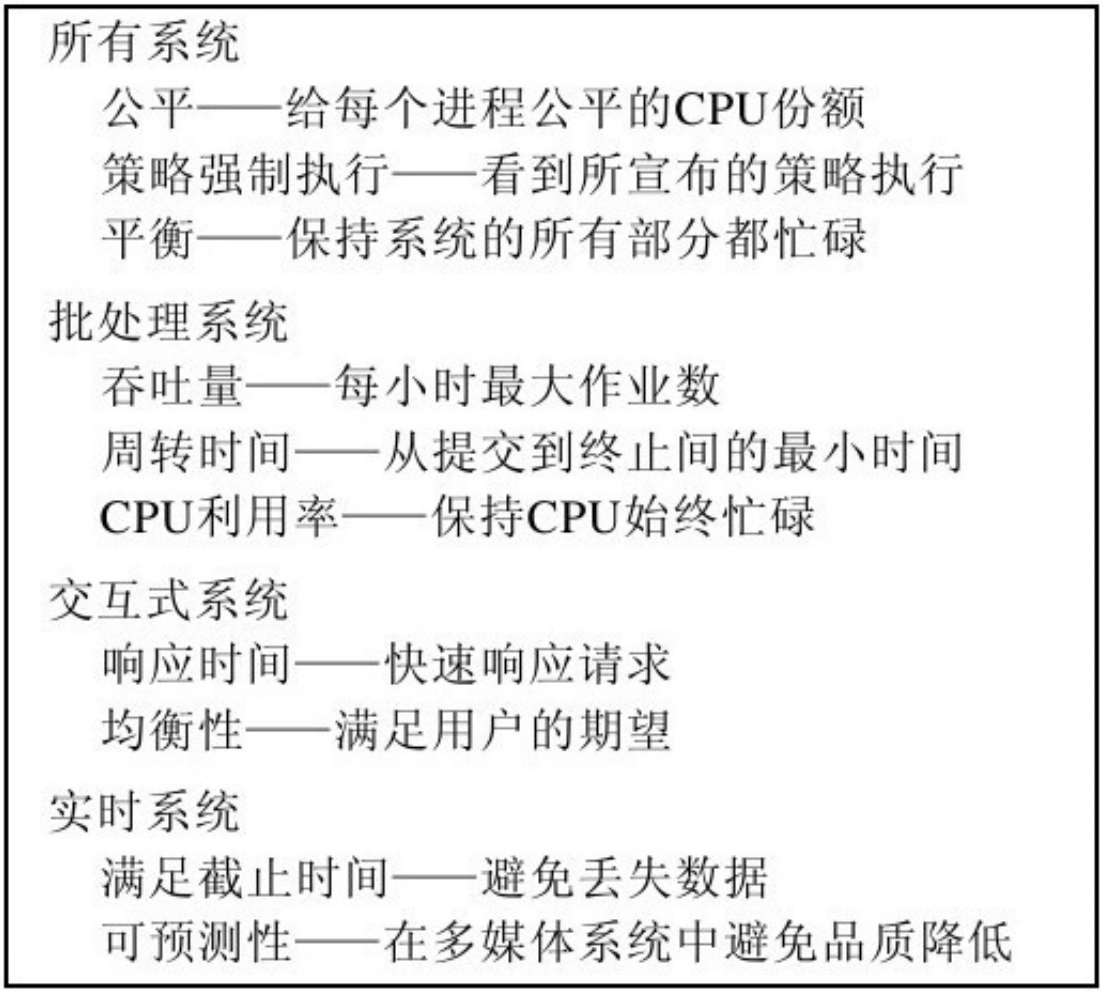
1. 在创建一个新进程之后，需要决定是运行父进程还是运行子进程。
2. 在一个进程退出时必须做出调度决策。
3. 当一个进程阻塞在 I/O 和信号量上或由于其他原因阻塞时，必须选择另一个进程运行。
4. 第四，在一个 I/O 中断发生时，必须做出调度决策。

2. 调度算法分类

1. 批处理。
2. 交互式。
3. 实时。

3. 调度算法的目标

为了设计调度算法，有必要考虑什么是一个好的调度算法。某些目标取决于环境（批处理、交互式或实时），但是还有一些目标是适用于所有情形的。



4. 批处理系统中的调度

（1）先来先服务

在所有调度算法中，最简单的是非抢占式的先来先服务算法。使用该算法，进程按照它们请求 CPU 的顺序使用 CPU 。

优点：

这个算法的主要优点是易于理解并且便于在程序中运用。

缺点：

平均等待时间过长。

（2）最短作业优先

当输入队列中有若干个同等重要的作业被启动时，调度程序应使用最短作业优先算法。

只有在所有的作业都可同时运行的情形下，最短作业优先算法才是最优化的。

（3）最短剩余时间优先

最短作业优先的抢占式版本是最短剩余时间优先算法。使用这个算法，调度程序总是选择剩余运行时间最短的那个进程运行。

5. 交互式系统中的调度

（1）轮转调度

一种最古老、最简单、最公平且使用最广的算法是轮转调度。每个进程被分配一个时间段，称为时间片，即允许该进程在该时间段中运行。如果在时间片结束时该进程还在运行，则将剥夺 CPU 并分配给另一个进程。如果该进程在时间片结束前阻塞或结束，则 CPU 立即进行切换。

需要注意的是，时间片设得太短会导致过多的进程切换，降低了CPU效率；而设得太长又可能引起对短的交互请求的响应时间变长。将时间片设为20ms～50 ms通常是一个比较合理的折中。

（2）优先级调度

每个进程被赋予一个优先级，允许优先级最高的可运行进程先运行。为了防止高优先级进程无休止地运行下去，调度程序可以在每个时钟滴答（即每个时钟中断）降低当前进程的优先级。如果这个动作导致该进程的优先级低于次高优先级的进程，则进行进程切换。

（3）多级队列

将一组进程按优先级分成若干类，并且在各类之间采用优先级调度，而在各类进程的内部采用其他调度方式。

（4）最短进程优先

对于批处理系统而言，由于最短作业优先常常伴随着最短响应时间，所以如果能够把它用于交互进程，那将是非常好的。

（5）保证调度

向用户作出明确的性能保证，然后去实现它。

一种很实际并很容易实现的保证是：若用户工作时有 n 个用户登录，则用户将获得 CPU 处理能力的 1/n 。类似地，在一个有 n 个进程运行的单用户系统中，若所有的进程都等价，则每个进程将获得 1/n 的 CPU 时间。看上去足够公平了。

（6）彩票调度

向进程提供各种系统资源（如 CPU 时间）的彩票。一旦需要做出一项调度决策时，就随机抽出一张彩票，拥有该彩票的进程获得该资源。在应用到 CPU 调度时，系统可以掌握每秒钟50次的一种彩票，作为奖励每个获奖者可以得到20 ms 的 CPU 时间。

（7）公平分享调度

到现在为止，我们假设被调度的都是各个进程自身，并不关注其所有者是谁。

为了避免这种情形，某些系统在调度处理之前考虑谁拥有进程这一因素。在这种模式中，每个用户分配到 CPU 时间的一部分，而调度程序以一种强制的方式选择进程。这样，如果两个用户都得到获得50% CPU 时间的保证，那么无论一个用户有多少进程存在，每个用户都会得到应有的 CPU 份额。

6. 策略和机制

我们讨论的调度算法中没有一个算法从用户进程接收有关的调度决策信息，这就导致了调度程序很少能够做出最优的选择。

解决问题的方法是将调度机制与调度策略分离，也就是将调度算法以某种形式参数化，而参数可以由用户进程填写。

在这里，调度机制位于内核，而调度策略则由用户进程决定。