2.7　线程(Threads)的基本概念

2.7.1　线程的引入

如果说，在OS中引入进程的目的是为了使多个程序能并发执行，以提高资源利用率和系统吞吐量，那么，在操作系统中再引入线程，则是为了**减少程序在并发执行时所付出的时空开销，使OS具有更好的并发性**。

1. 进程的两个基本属性

① 进程是一个可拥有资源的独立单位。一个进程要能独立运行，它必须拥有一定的资源，包括用于存放程序正文、数据的磁盘和内存地址空间，以及它在运行时所需要的I/O设备、已打开的文件、信号量等；

② 进程同时又是一个可独立调度和分派的基本单位。每个进程在系统中有唯一的PCB，系统可根据其PCB感知进程的存在，也可以根据其PCB中的信息，对进程进行调度，还可将断点信息保存在其PCB中。反之，再利用进程PCB中的信息来恢复进程运行的现场。

1. 进程的两个基本属性

① 进程是一个可拥有资源的独立单位。一个进程要能独立运行，它必须拥有一定的资源，包括用于存放程序正文、数据的磁盘和内存地址空间，以及它在运行时所需要的I/O设备、已打开的文件、信号量等；

② 进程同时又是一个可独立调度和分派的基本单位。每

正是由于进程有这两个基本属性，才使进程成为一个能独立运行的基本单位，从而也就构成了进程并发执行的基础。

可将断点信息保存在其PCB中。反之，再利用进程PCB中的信息来恢复进程运行的现场。

2. 程序并发执行所需付出的时空开销

为使程序能并发执行，系统必须进行以下的一系列操作：

1）创建进程　系统在创建进程时必须为它分配除CPU以外的所有资源，如内存空间、I/O设备，以及建立相应的PCB。

2）撤消进程　系统撤消进程时，必须先回收其所占的资源，然后再撤消PCB。

3）进程切换　进程切换时，由于要保留当前进程的CPU环境和设置新进程的CPU环境，因而需花费不少处理机时间。

由于进程是一个资源拥有者，因而在创建、撤消、切换中，系统必须为之付出较大的时空开销，故系统中的进程**数目不宜过多**，进程**切换的频率也不宜过高**，这限制了并发程度的进一步提高。

3. 线程——作为调度和分派的基本单位

进程既是拥有资源的独立单位，又是可独立调度和分派的基本单位。若能将进程的上述两个属性分开，由操作系统分开处理，即能使多个程序更好地并发运行，同时又尽量减少系统的开销。

★对于作为调度和分派的基本单位，不同时作为拥有资源的基本单位，以做到“轻装上阵”；

★而对于拥有资源的基本单位，又不对之进行频繁的切换。

在这种思想指导下，形成了线程概念。

线程只作为调度和分派的基本单位，而不作为资源分配的基本单位。

- 一个进程通常包括多个线程。

2.7.2　线程与进程的比较

1. 调度的基本单位
2. 并发性
3. 拥有资源
4. 独立性
5. 系统开销
6. 支持多处理机系统

2.7.2　线程与进程的比较

1. 调度的基本单位

传统OS中，进程是作为独立调度和分派的基本单位，因而进程是能独立运行的基本单位；
引入线程的OS中，把线程作为调度和分派的基本单位，因而线程是能独立运行的基本单位。

5. 系统开销

同一进程中，线程的切换不会引起进程的切换，但从一个进程中的线程切换到另一个进程中的线程时，必然会引起进程的切换。

2.7.2　线程与进程的比较

1. 调度的基本单位

2. 并发性

引入线程的OS中，进程之间可以并发执行，一个进程中的多个线程之间也可以并发执行，不同进程中的线程同样也能并发执行。

6. 支持多处理机系统

2.7.2　线程与进程的比较

1. 调度的基本单位
2. 并发性
3. 拥有资源

进程拥有资源，并作为系统中拥有资源的一个基本单位。

线程本身不拥有系统资源，仅有一点必不可少的、能保证独立运行的资源。

2.7.2　线程与进程的比较

1. 调度的基本单位
2. 并发性
3. 拥有资源
4. 独立性

同一进程中的不同线程之间的独立性比不同进程之间的独立性低得多。
原因：为了防止进程间彼此干扰和破坏，每个进程都拥有一个独立的地址空间和其他资源，除了共享全局变量外，不允许其他进程访问。
但是同一进程中的不同线程共享进程的内存地址空间和资源。

2.7.2　线程与进程的比较

1. 调度的基本单位
2. 并发性
3. 拥有资源
4. 独立性
5. 系统开销

创建或撤销进程时，系统都要为之分配和回收PCB和其他资源，如内存和I/O设备等，OS为此付出的开销明显大于线程创建或撤销时所付出的开销。
进程切换时，涉及到进程的上下文切换，远高于线程的切换代价。

2.7.2　线程与进程的比较

1. 调度的基本单位
2. 并发性
3. 拥有资源
4. 独立性
5. 系统开销
6. 支持多处理机系统

传统的进程，即单线程进程，不论有多少处理机，该进程只能运行在一个处理机上。
多线程进程可以将一个进程的多个线程分配到多个处理机上并行执行。

2.7.3　线程的属性

线程有四种属性：

(1) 轻型实体：线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证其独立运行的资源；

(2)独立调度和分派的基本单位：在多线程OS中，线程是能独立运行的基本单位，因而也是独立调度和分派的基本单位。由于线程很“轻”，故线程的切换非常迅速且开销小；

(3)可并发执行；

(4)共享进程资源：同一进程中的各个线程都可以共享该进程所拥有的资源，所有线程都具有相同的地址空间。这意味着线程可以访问该地址空间中的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。

2.7.4　线程的状态和线程控制块

1. 线程运行的三个状态

与传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。相应地，线程在运行时也具有下述三种基本状态：

(1) 执行状态，表示线程已获得处理机而正在运行；

(2) 就绪状态，指线程已具备了各种执行条件，只须再获得CPU便可立即执行；

(3) 阻塞状态，指线程在执行中因某事件受阻而处于暂停状态，例如，当一个线程执行从键盘读入数据的系统调用时，该线程就被阻塞。

2. 线程控制块TCB

每个线程配置一个线程控制块TCB，记录所有用于控制和管理线程的信息。
TCB中包括线程标识符和一组状态参数。

状态参数包括：

- ◆寄存器：程序计数器PC和堆栈指针内容
- ◆堆栈：通常保存局部变量和返回地址
- ◆线程运行状态：执行状态、就绪状态、阻塞状态
- ◆优先级
- ◆线程专用存储器：用于保存线程自己的局部变量拷贝
- ◆信号屏蔽：对某些信号加以屏蔽

3. 多线程OS中的进程属性

通常在多线程OS中的进程都包含了多个线程，并为它们提供资源。

OS支持在一个进程中的多个线程能并发执行，但此时的进程就**不再作为一个执行的实体**。

多线程OS中的进程有以下属性：

(1) 进程是一个可拥有资源的基本单位。

(2)进程已不是可执行的实体。

(3)多个线程可并发执行。

3. 多线程OS中的进程属性

通常在多线程OS中的进程都包含了多个线程，并为它们提供资源。

OS支持在一个进程中的多个线程能并发执行，但此时的

进程仍具有与执行相关的状态。
所谓进程处于“执行”状态，是指该进程中的某线程正在执行。

(2)进程已不是可执行的实体。

(3)多个线程可并发执行。

2.8　线 程 的 实 现

2.8.1　线程的实现方式

线程已在许多系统中实现，但各系统的实现方式并不完全相同。

常见的实现方式包括用户级线程、内核支持线程和混合类型的线程。

1. 内核支持线程KST(Kernel Supported Threads)

在OS中的所有进程，无论是系统进程还是用户进程，都是在操作系统内核的支持下运行的，是与内核紧密相关的。而内核支持线程KST同样也是在内核的支持下运行的，它们的创建、阻塞、撤消和切换等，也都是在内核空间实现的。

为了对内核线程进行控制和管理，在内核空间也为每一个内核线程设置了一个线程控制块，内核根据该控制块而感知某线程的存在，并对其加以控制。

当前大多数OS都支持内核支持线程。

内核支持线程主要有四个主要优点：

(1) 在多处理器系统中，内核能够同时调度同一进程中的多个线程并行执行；

(2) 如果进程中的一个线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程；

(3) 内核支持线程具有很小的数据结构和堆栈，线程的切换比较快，切换开销小；

(4) 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率。

内核支持线程的缺点：

线程切换的开销较大。

同一个进程中，从一个线程切换到另一个线程时，需要从用户态转到核心态进行。因为用户进程的线程在用户态运行，而线程调度和管理是在内核实现的，频繁的在用户态和核心态转换导致系统开销较大。

2. 用户级线程ULT(User Level Threads)

用户级线程是在用户空间中实现的。对线程的创建、 撤消、同步与通信等功能，都无需内核的支持，即用户级线程是与内核无关的。

这些线程的任务控制块都是设置在用户空间，而线程所执行的操作也无需内核的帮助，因而**内核完全不知道用户级线程的存在**。

在一个系统中的用户级线程的数目可以达到数百个至数千个。

使用用户级线程方式有许多优点：

(1) 线程切换不需要转换到内核空间，可以节省模式切换的开销。

(2) 调度算法可以是进程专用的。在不干扰OS调度的情况下，不同的进程可以根据自身需要选择不同的调度算法，对自己的线程进行管理和调度，与OS的低级调度算法无关。

(3) 用户级线程的实现与OS平台无关，因为对于线程管理的代码是属于用户程序的一部分。

用户级线程方式的主要缺点则在于：

(1) 系统调用的阻塞问题。

在基于进程机制的OS中，大多数系统调用将使进程阻塞，因此，当线程执行一个系统调用时，不仅该线程被阻塞，而且，进程内的所有线程会被阻塞。而在内核支持线程方式中，则进程中的其它线程仍然可以运行。

(2) 多线程应用不能利用多处理机进行多重处理。

内核每次分配给一个进程的仅有一个CPU，因此，进程中仅有一个线程能执行，在该线程放弃CPU之前，其它线程只能等待。

- 对于设置了用户级线程的系统，调度仍是以进程为单位进行。在采用轮转调度算法时，各个进程轮流执行一个时间片，对于诸进程来说看似公平，但假如在进程A中包含了一个用户级线程，进程B中包含100个用户级线程，则进程A中线程运行的时间将是进程B中各线程运行时间的100倍，因此实质上并不公平。
- 若设置了系统级线程，则调度是以线程为单位进行的。在采用轮转调度算法时，各个线程轮流执行一个时间片。同样假设进程A中仅有一个内核支持线程，进程B中有100个内核支持线程，则进程B可获得的CPU时间是进程A的100倍，而且进程B可以使100个系统调用并发工作。

3. 组合方式

有些OS把用户级线程和内核支持线程两种方式进行组合，提供了组合方式ULT/KST 线程。在组合方式线程系统中，内核支持多个内核支持线程的建立、调度和管理，同时，也允许用户应用程序建立、调度和管理用户级线程。

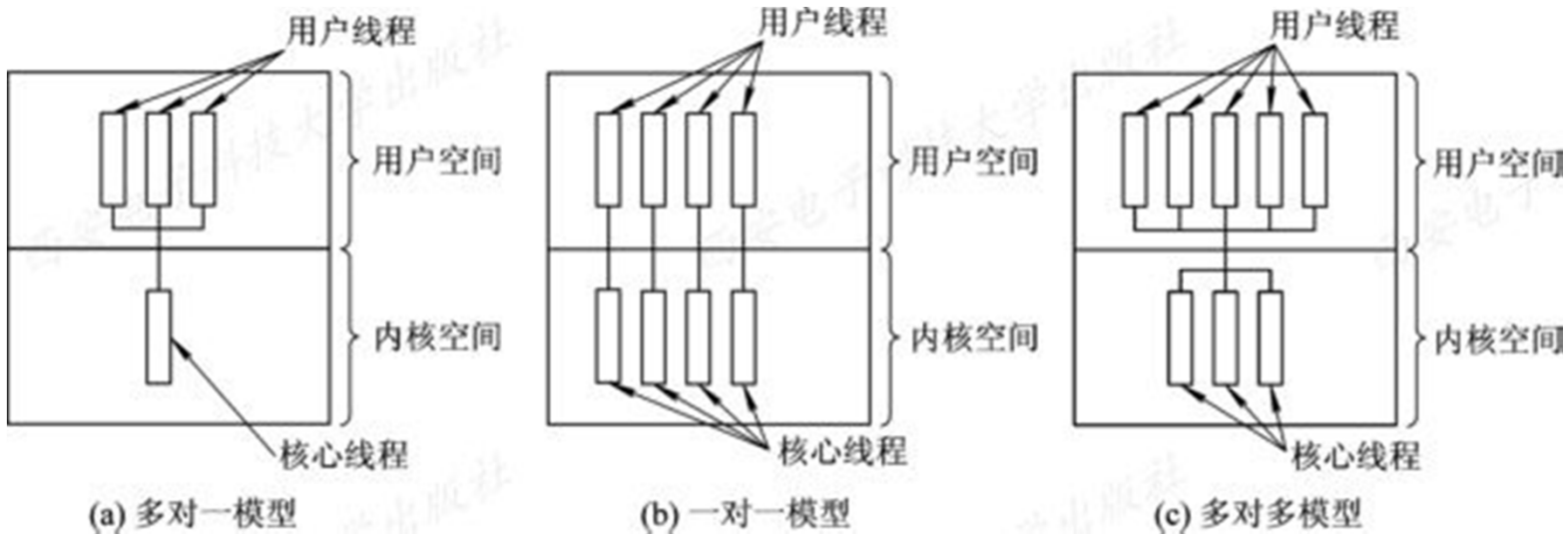
用户线程
用户空间
内核空间
核心线程
(a) 多对一模型
用户线程
用户空间
内核空间
核心线程
(b) 一对一模型
用户线程
用户空间
内核空间
核心线程
(c) 多对多模型

图2-18 多线程模型

2.8.2　线程的实现

1. 内核支持线程的实现

在仅设置了内核支持线程的OS中，一种可能的线程控制方法是，系统在创建一个新进程时，便为它分配一个任务数据区PTDA(Per Task Data Area)，其中包括若干个线程控制块TCB空间，如图2-19所示。

图2-19　任务数据区空间

2. 用户级线程的实现

用户级线程在用户空间实现，所有的用户级线程都具有相同的结构，都运行在一个中间系统上。实现中间系统有两种方法：运行时系统和内核控制线程。

1) 运行时系统(Runtime System)

所谓“运行时系统”，实质上是用于管理和控制线程的函数（过程）的集合，其中包括用于创建和撤消线程的函数、线程同步和通信的函数，以及实现线程调度的函数等。正因为有这些函数，才能使用户级线程与内核无关。

运行时系统中的所有函数都驻留在用户空间，并作为用户级线程与内核之间的接口。

2) 内核控制线程

这种线程又称为轻型进程LWP(Light Weight Process)。每一个进程都可拥有多个LWP，同用户级线程一样，每个LWP都有自己的数据结构(如TCB)，其中包括线程标识符、优先级、状态，另外还有栈和局部存储区等。LWP也可以共享进程所拥有的资源。LWP可通过系统调用来获得内核提供的服务，这样，当一个用户级线程运行时，只须将它连接到一个LWP上，此时它便具有了内核支持线程的所有属性。这种线程实现方式就是组合方式。

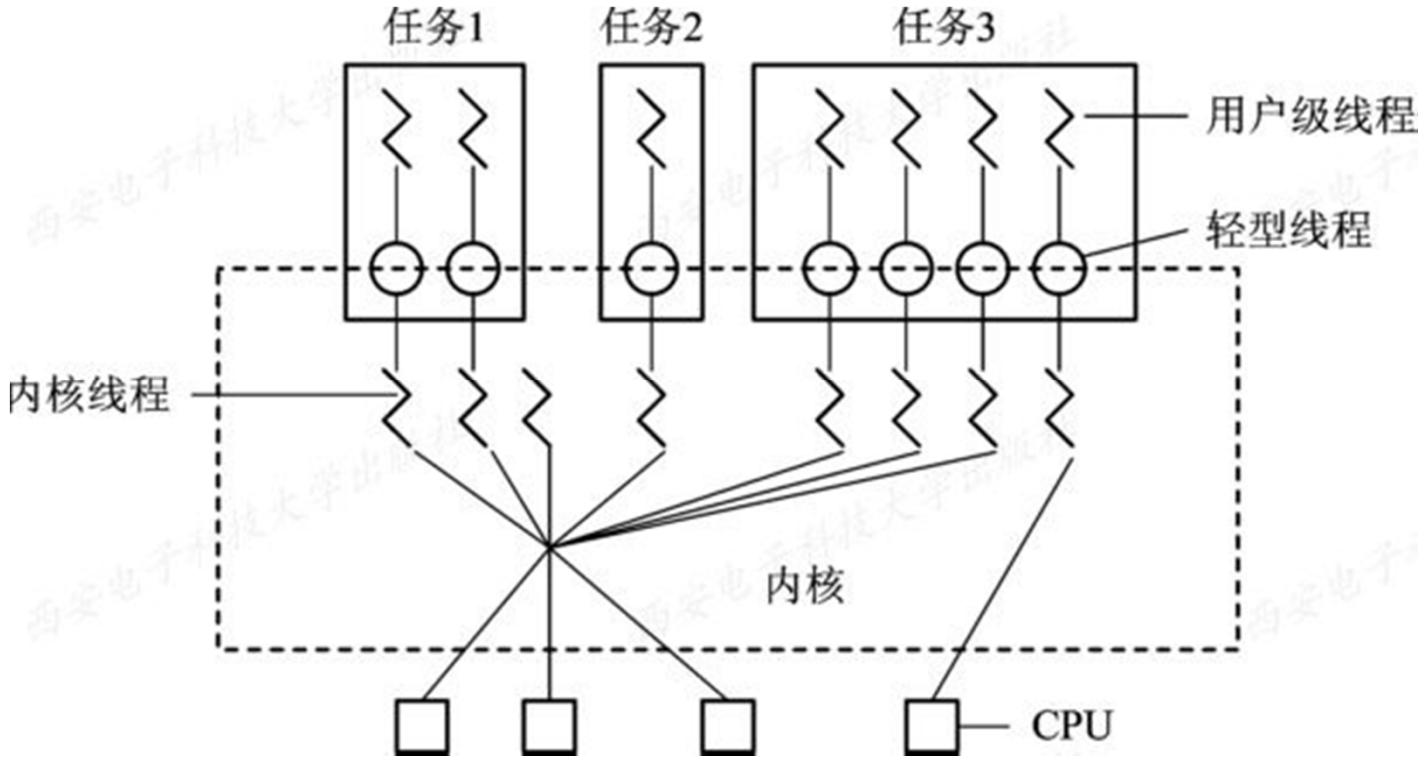

图2-20　利用轻型进程作为中间系统

2.8.3　线程的创建和终止

1. 线程的创建

应用程序在启动时，通常仅有一个线程在执行，人们把该线程称为“初始化线程”，它的主要功能是用于创建新线程。在创建新线程时，需要利用一个线程创建函数(或系统调用)，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小，以及用于调度的优先级等。在线程的创建函数执行完后，将返回一个线程标识符供以后使用。

2. 线程的终止

当一个线程完成了自己的任务(工作)后，或是线程在运行中出现异常情况而须被强行终止时，由终止线程通过调用相应的函数(或系统调用)对它执行终止操作。但有些线程(主要是系统线程)，它们一旦被建立起来之后，便一直运行下去而不被终止。在大多数的OS中，线程被中止后并不立即释放它所占有的资源，只有当进程中的其它线程执行了分离函数后，被终止的线程才与资源分离，此时的资源才能被其它线程利用。

第三章　处理机调度与死锁

3.1 处理机调度的层次和调度算法的目标
3.2 进程作业与调度
3.3 进程调度
3.4 实时调度
3.5 死锁概述
3.6 预防死锁
3.7 避免死锁
3.8 死锁的检测与解除

3.1　处理机调度的层次和调度算法的目标

在多道程序系统中，**调度的实质是一种资源分配**，处理机调度是对处理机资源进行分配。

- **处理机调度算法**是指根据处理机分配策略所规定的处理机分配算法。
- 对于大型系统运行时的性能，如系统吞吐量、资源利用率、作业周转时间或响应的及时性等，在很大程度上都取决于处理机调度性能的好坏。

3.1.1　处理机调度的层次

1. 高级调度(High Level Scheduling)

2. 低级调度(Low Level Scheduling)

3. 中级调度(Intermediate Scheduling)

1. 高级调度——又称作业调度或长调度

定义　用于决定把外存上后备队列中哪些作业调入内存，并为它们创建进程、分配必要的资源，然后将新创建的进程插入到就绪队列中，准备运行。

每次作业调度，都需做以下两个决定：

- 接纳多少个作业——取决于多道程序度
- 接纳哪些作业——取决于调度算法

▲内存中同时运行的作业数目太多，会影响系统的服务质量。如，周转时间长。
▲内存中同时运行的作业数目太少，会导致系统资源利用率和系统吞吐量低。

高级调度的对象是“作业”

2．低级调度

又称进程调度或短调度

定义

用来决定就绪队列中的哪个进程应获得处理机，然后再由分派程序执行把处理机分配给该进程的具体操作。

- 低级调度的对象是“进程”或“内核级线程”
- 低级调度是最基本的一种调度，在多道批处理、分时和实时三种类型的OS中，都必须配置这一级调度

2．低级调度

又称进程调度或短调度

定义　用来决定就绪队列中的哪个进程应获得处理机，然后再由分派程序执行把处理机分配给该进程的具体操作。

进程调度可采用下述两种调度方式：

（1）非抢占方式

（2）抢占方式

一旦把处理机分配给某进程后，便让它一直执行，直到该进程完成或发生某事件而被阻塞时，才把处理机分配给其它进程，决不允许某进程抢占已经分配出去的处理机。
优点：实现简单，系统开销小。
缺点：难于满足紧急任务的要求。

2．低级调度

又称进程调度或短调度

定义　用来决定就绪队列中的哪个进程应获得处理机，然后再由分派程序执行把处理机分配给该进程的具体操作。

进程调度可采用下述两种调度方式：

（1）非抢占方式

（2）抢占方式

允许调度程序根据某种原则，暂停某个正在执行的进程，将已分配给该进程的处理机重新分配给另一进程。

抢占原则有：

- 优先权原则
- 短作业优先原则
- 时间片原则

3. 中级调度——又称内存调度

挂起和激活，存储器管理中的对换功能。

主要目的：为了提高内存的利用率和系统的吞吐量。

三种调度相比较：

进程调度——运行频率最高

作业调度——频率最低

- 10-100ms进行一次调度
- 为了避免调度本身占用太多CPU时间，不宜使进程调度算法太复杂
- 短程调度

3. 中级调度——又称内存调度

挂起和激活，存储器管理中的对换功能。

主要目的：为了提高内存的利用率和系统的吞吐量。

三种调度相比较：

进程调度——运行频率最高

作业调度——频率最低

中级调度——界于两者之间

- 一般发生在一批作业运行完毕并退出系统，又需要重新调入一批作业进入内存时
- 几分钟一次
- 长程调度

3. 中级调度——又称内存调度

挂起和激活，存储器管理中的对换功能。

主要目的：为了提高内存的利用率和系统的吞吐量。

三种调度相比较：

进程调度——运行频率最高

作业调度——频率最低

中级调度——界于两者之间

- 基于上两者之间
- 中程调度

3.1.2　处理机调度算法的目标

操作系统如何选择调度方式和算法，很大程度上取决于操作系统的类型及其设计目标。

1. 处理机调度算法的共同目标

(1) 资源利用率。

为提高系统的资源利用率，应使系统中的处理机和其它所有资源都尽可能地保持忙碌状态。

处理机的利用率可用以下方法计算：

$$\text{CPU 的利用率} = \frac{\text{CPU 有效工作时间}}{\text{CPU 有效工作时间} + \text{CPU 空闲等待时间}}$$

(2) 公平性。公平性是指应使诸进程都获得合理的CPU 时间，不会发生进程饥饿现象。公平性是相对的，对相同类型的进程应获得相同的服务；但对于不同类型的进程，由于其紧急程度或重要性的不同，则应提供不同的服务。

(3) 平衡性。由于在系统中可能具有多种类型的进程，有的属于计算型作业，有的属于I/O型。为使系统中的CPU和各种外部设备都能经常处于忙碌状态，调度算法应尽可能保持系统资源使用的平衡性。

(4) 策略强制执行。对所制订的策略其中包括安全策略，只要需要，就必须予以准确地执行，即使会造成某些工作的延迟也要执行。

2. 批处理系统的目标

(1) 平均周转时间短

- 周转时间：从作业被提交给系统开始，到作业完成为止的这段时间间隔。
- 包括四部分时间：
  ① 作业在外存后备队列上等待作业调度的时间
  ② 进程在就绪队列上等待进程调度的时间
  ③ 进程在CPU上执行的时间
  ④ 进程等待I/O操作完成的时间

对**每个用户而言**，都希望自己作业的周转时间最短；

作为**计算机系统的管理者**，则总是希望能使**平均周转**时间最短，这不仅会有效地提高系统资源的利用率，而且还可使大多数用户都感到满意。

应使作业周转时间和作业的平均周转时间尽可能短。否则，会使许多用户的等待时间过长，这将会引起用户特别是短作业用户的不满。

可把平均周转时间描述为：

$$T = \frac{1}{n}\left[\sum_{i=1}^{n} T_i\right]$$

为了进一步反映调度的性能，更清晰地描述各进程在其周转时间中，等待和执行时间的具体分配状况，往往使用**带权周转时间**，即作业的周转时间T_i与系统为它提供服务的时间T_s之比，即

$$W = \frac{T_i}{T_s}$$

平均带权周转时间则可表示为：

$$W = \frac{1}{n}\sum_{i=1}^{n}\frac{T_i}{T_s}$$

(2) 系统吞吐量高。由于吞吐量是指在单位时间内系统所完成的作业数，因而它与批处理作业的平均长度有关。

事实上，如果单纯是为了获得高的系统吞吐量，就应尽量多地选择短作业运行。

(3) 处理机利用率高。对于大、中型计算机，CPU价格十分昂贵，致使处理机的利用率成为衡量系统性能的十分重要的指标；而调度方式和算法又对处理机的利用率起着十分重要的作用。

如果单纯是为使处理机利用率高，应尽量多地选择计算量大的作业运行。

由上所述可以看出，这些要求之间是存在着一定矛盾的。

3. 分时系统的目标

(1) 响应时间快

- 响应时间：从用户通过键盘提交一个请求开始，直到屏幕上显示出处理结果为止的一段时间间隔。
- 包括三部分时间：
  ① 请求信息从键盘输入开始，直至将其传送到处理机的时间
  ② 处理机对请求信息进行处理的时间
  ③ 将所形成的的响应信息回送到终端显示器的时间

(2) 均衡性

- 均衡性：系统响应时间的快慢应与用户所请求服务的复杂性相适应
- 用户通常对较复杂任务的响应时间允许较长，对较简单任务的响应时间要短

4. 实时系统的目标

(1) 截止时间的保证

- 截止时间：某任务必须开始执行的最迟时间，或必须完成的最迟时间。
- 实时系统中，调度算法的一个主要目标是保证实时任务对截止时间的要求。
  ① HRT任务（硬实时任务）：调度方式和调度算法必须确保对截止时间的要求，否则可能造成难以预料的后果；
  ② SRT任务（软实时任务）：调度方式和调度算法应基本上能保证对截止时间的要求。

(2) 可预测性

- 多媒体系统中的双缓冲，可实现第i帧的播放和第i+1帧的读取并行处理，进而提高其实时性。
- 前台播放，后台缓冲