第3章 处理机调度与死锁

本章学习目标

本章主要介绍两部分内容，即操作系统中处理机的调度与死锁问题。操作系统中的调度分为三个层次，即作业调度、对换和进程调度。本章主要介绍作业调度、进程调度的概念、过程以及作业调度与进程调度的算法，并利用这些算法解决一些实际问题；介绍死锁的概念、产生死锁的原因，以及解决策略。

通过本章的学习，读者应掌握以下内容：

- 掌握处理机的三级调度；
- 掌握作业调度及进程调度的概念及过程；
- 掌握调度算法的目标；
- 掌握典型的作业调度、进程调度算法，并利用其解决实际问题；
- 掌握死锁的概念、产生的原因及死锁的必要条件；
- 掌握死锁的预防方法；
- 掌握利用银行家算法避免死锁的方法；
- 掌握死锁的检测与解除的方法。


视频讲解

3.1 作业管理

3.1.1 作业的概念及分类

1. 作业的概念

作业是用户在一次解题或一个事务处理过程中要求计算机系统所做工作的集合。也可以说，把一次应用业务处理过程，从输入开始到输出结束，用户要求计算机所做的相关该次业务处理的全部工作，称为一个作业。例如，打印一个文件、检索一个数据库、发送一个邮件等，都可视为一个作业。

从系统的角度讲，作业是一个比较广泛的概念，它由程序、数据和作业说明书组成。系统通过作业说明书控制程序和数据，使它们运行和操作，并且在批处理系统中，作业是加载内存的基本单位，系统将作业调入内存并执行。

2. 作业的分类

依据计算机系统作业处理方式的不同，可把作业分成两大类：脱机作业和联机作业。

也可以称为批处理型作业和交互型作业。脱机作业是指用户不能直接与计算机系统交互，中间必须通过操作员干预的作业。这种作业通常在批处理系统中使用，所以称为批处理作业。联机作业是指用户和计算机系统直接交互，用户通过终端或控制台键盘上的操作命令或图形窗口界面等方式，控制其作业的运行，这种作业也称为交互式作业或终端型作业。通常，操作系统对这两类作业的管理方式是不同的。

脱机作业多出现在批处理系统中，而联机作业多出现在分时系统中。在分时和批处理兼容的系统中，将终端作业作为前台作业，而把批处理作业作为后台作业。一般情况下，前台作业的优先权较高，作业响应及时。在前台无作业时，可调度后台的批处理作业，达到提高系统效率的目的。

3.1.2　作业的状态

一个作业从提交给计算机系统到执行结束，退出系统，一般要经历提交、后备、执行和完成四个状态。

(1) 提交状态，一个作业在处于从输入设备进入外部存储设备的过程称为提交状态。处于提交状态的作业，因其信息尚未全部进入系统，所以不能被调度程序选中。

(2) 后备状态，也称为收容状态。输入管理系统不断地将作业输入到外存中对应部分(或称输入井)。若一个作业的全部信息已全部被输入到输入井，则在它还未被调度执行之前，该作业处于后备状态。

(3) 执行状态。作业调度程序从后备作业中选取若干个作业到内存投入运行，以及为被选中作业建立进程并分配必要的资源，这时，这些被选中的作业处于执行状态。从宏观上看，这些作业正处在执行过程中；从微观上看，在某一时刻，由于处理机总数少于并发执行的进程数，不是所有被选中作业都占有处理机，其中的大部分处于等待资源或就绪状态。哪个作业的哪个进程被分配给处理机，这是进程调度要完成的任务。

(4) 完成状态。当作业运行完毕，但它所占用的资源尚未全部被系统回收时，该作业处于完成状态。在这种状态下，系统需做如打印结果、回收资源等类似的善后处理工作。

3.1.3　作业管理的功能

作业管理的功能包括作业调度和作业控制。所谓作业调度，就是按照某种作业调度算法从后备作业队列中选择一个作业加载内存并运行。作业控制就是按照作业控制语言的解释程序读取用户作业说明书，具体控制作业的执行，并按照规定的步骤对作业进行处理。

操作系统为用户提供各种操作命令，来组织作业的工作流程和控制作业的运行，这是操作系统提供的作业控制级的用户接口。批处理作业与交互式作业的作业控制方式是不同的。

1. 作业控制块

通常，系统为每个作业建立一个作业控制块(Job Control Block，JCB)，用以记录这些有关信息。正如系统通过进程控制块(Process Control Block，PCB)而感知进程的存在一样，

系统通过JCB而感知作业的存在。系统在作业进入后备状态时,为它建立了JCB,从而使该作业可被作业调度程序选中。当该作业执行完毕进入完成状态后,系统又撤销其JCB,释放有关资源并撤销该作业。每个作业的状态、在各个阶段所需要和已分配的资源都记录在它的JCB中,根据JCB中的有关信息,作业调度程序对作业进行调度和管理。当一个作业执行结束进入完成状态时,系统负责回收分配给它的资源,撤销它的作业控制块。

对于不同的批处理系统,其JCB的内容也有所不同。JCB的主要内容如表3-1所示。

表3-1 作业控制块

主要功能	说 明
作业名	
作业类型	计算型 管理型 图形设计型
资源要求	内存量 外存量 外设类型及数量 软件支持工具库函数
当前状态	提交状态 后备态 运行态 完成
资源使用情况	进入系统的时间 开始执行时间 已运行时间 内存地址 外设台数
作业的优先级	

从表3-1可以看出,JCB的主要功能如下。

作业名:由用户提供并由系统将其转换为系统可识别的作业标识符。

作业类型:该作业属于计算型(要求CPU时间多)、管理型(要求输入输出量大)或图形设计型(要求高速图形显示)等。

资源要求:要求的内存量和外存量、外设类型及数量,以及要求的软件支持工具库函数等。资源要求均由用户提供。

当前状态:该作业当前所处的状态。显然,只有当作业处于后备状态时,该作业才可以被调度。

资源使用情况:包括作业进入系统的时间、开始执行时间、已执行时间、内存地址、外设数量等。作业进入系统的时间是指作业的全部信息进入输入井、作业的状态为后备状态的时间。开始执行时间指该作业被调度程序选中,其状态由后备状态变为执行状态的时间。内存地址指分配给该作业的内存区起始地址。外设数量指分配给该作业的外设

实际数量。

作业的优先级：优先级用来决定该作业的调度次序。优先级既可以由用户给定，也可以由系统动态计算产生。

2. JCB 的组织方式

每个作业有一个 JCB，在系统中可以通过作业表、作业队列对各个作业的 JCB 进行组织。

1）作业表

所有的 JCB 构成一个表，称为作业表，如表 3-2 所示。作业表存放在外存固定区域，通常其长度是固定的，这就限制了系统所能同时容纳的作业数量。系统输入程序、作业调度程序和系统输出程序都需要访问作业表。

表 3-2　作业表

JCB_1	JCB_2	…	JCB_i	…	JCB_n

2）作业队列

对于系统中的作业，系统将它们的 JCB 构成一个或多个队列，便于系统控制和访问。JCB 队列比 JCB 表具有更大的灵活性，因此被更多的操作系统采用。

3.1.4　作业与进程的关系

作业是用户向计算机提交任务的任务实体。一个作业是指在一次应用业务处理过程中，从输入开始到输出结束，用户要求计算机所做的有关该次业务处理的全部工作，如一次计算、一个控制过程等。进程是计算机为了完成用户任务实体而设置的执行实体，是系统分配资源的基本单位。显然，计算机要完成一个任务，必须有一个以上的执行实体，即一个作业总是由一个或多个进程组成的。作业分解为进程的过程：首先，系统为一个作业创建一个根进程；其次，在执行作业控制语句时，根据任务要求，系统或根进程为其创建相应的子进程；最后，进行进程调度，为各子进程分配资源和调度各子进程执行，以完成作业要求的工作。

3.2　分级调度

视频讲解

操作系统一个非常重要的功能就是管理计算机资源，提高系统的效率。对处理机的管理是操作系统的基本功能之一。在早期的计算机系统中，对 CPU 的管理是十分简单的，因为那时它和其他系统资源一样，被一个作业所独占，不存在处理机分配和调度问题。随着多道程序设计技术和各种类型的操作系统的出现，各种不同的 CPU 管理方式开始使用，为用户提供了不同性能的操作系统。例如，在多道批处理系统中，为了提高处理机的效率和增加作业吞吐量，当调度一批作业组织多道运行时，要尽可能使作业组织合理，但由于在批处理系统中，一旦把作业提交给计算机系统，直到作业运行完成，用户都不能插手干预自己的作业，而且作业的响应时间一般都较长，因此，在用户看来，这是一台没有交互、速度较慢的处

理机。但是,在批处理系统中,其资源的利用率和系统的吞吐量高。在分时系统中,由于用户使用交互式会话的工作方式,系统必须有较快的响应时间,使得每个用户都感到如同只有他自己一人在使用这台机器,因此,系统在调度作业执行时,要首先考虑每个用户作业得到处理机的均等性。这样,系统资源的利用率就不如批处理系统。由此可以看出,根据操作系统的要求不同,处理机管理的策略也是不同的。

一个批处理型作业从进入系统并驻留在外存的后备队列上开始,直至作业运行完毕,可能要经历以下三级调度,即作业调度、对换和进程调度。

1. 作业调度

作业调度又称高级调度或长调度,用于选择把外存上处于后备队列中的哪些作业调入内存,并为它们创建进程、分配必要的资源。然后,再将新创建的进程排在就绪队列上,准备执行。

在批处理系统中,作业进入系统后,是先驻留在外存上的,因此需要有作业调度的过程,以便将它们分批地装入内存。在分时系统中,为了做到及时响应,用户通过键盘输入的命令或数据等,都是被直接送入内存的,因而无须再配置作业调度机制。类似地,在实时系统中,通常也不需要作业调度。

2. 对换

对换又称交换调度或中级调度,其主要任务是按照给定的原则和策略,将处于外存交换区中的就绪状态或等待状态的进程调入内存,或把处于内存就绪状态或内存等待状态的进程交换到外存交换区。交换调度主要涉及内存管理与扩充,将在内存管理部分讨论这个问题。

3. 进程调度

进程调度又称为低级调度或微观调度,其主要任务是按照某种策略和算法,将处理机分配给一个处于就绪状态的进程。在确定了占用处理机的进程后,系统必须进行进程上下文切换以建立与占用处理机进程相适应的执行环境。进程调度可分为如下两种方式。

(1) 非抢占方式。非抢占方式不允许进程抢占已经分配出去的处理机。采用非抢占调度方式时,可能引起进程调度的原因有正在执行的进程执行完成,或因发生某事件而不能继续执行;执行中的进程因提出 I/O 请求而暂停执行;在进程通信或同步过程中执行了某种原语操作,如 P 操作(wait()操作)、Block 原语、Wakeup 原语等。

非抢占调度方式的优点是实现简单、系统开销小,适用于大多数批处理系统环境。但它很难满足紧急任务的要求,因而可能造成难以预料的后果。显然,在要求比较严格的实时系统中,不宜采用这种调度方式。

(2) 抢占方式。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,将处理机收回,重新分配给另一个进程。抢占的原则有优先权原则、短作业(或短进程)优先原则和时间片原则等。

作业调度与进程调度的关系如图 3-1 所示。

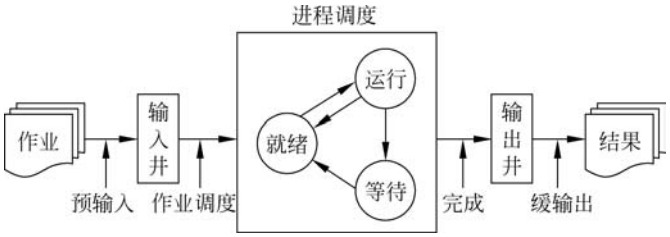

图 3-1　作业调度与进程调度的关系

3.3　作业调度

3.3.1　作业调度的功能

作业调度的主要任务是：根据作业控制块中的信息，审查系统能否满足用户作业的资源需求，以及按照某一作业调度算法，从外存的后备状态作业队列中选择某些作业将其装入内存并执行。为了完成这一任务，作业调度程序应完成以下功能。

(1) 确定数据结构。作业调度程序根据各个作业的 JCB 提供的信息对作业进行调度和管理。

(2) 确定调度算法。按照一定的调度算法，从后备作业队列中挑选出一个或几个作业投入运行，即将这些作业由后备状态转变为执行状态。这一工作由调度程序完成。作业调度程序的调度时机和调度原则通常与系统的设计目标有关，并由许多因素确定。

(3) 分配资源。为被选中的作业的进程分配运行时所需要的系统资源，如内存和外设等。作业调度程序在调度一个作业进入内存时，必须为该作业建立相应的进程，并且为这些进程提供所需的资源。对处理机的分配工作由进程调度程序来完成。

(4) 善后处理。在一个作业执行结束时，作业调度程序输出一些必要的信息，例如执行时间、作业执行情况等，然后收回该作业所占用的全部资源，撤销与该作业有关的全部进程和该作业的作业控制块。

作业从后备状态到执行状态的转换过程如图 3-2(a)所示，从执行状态到完成状态的转换过程如图 3-2(b)所示。

Linux 系统的作业一旦输入，就直接进入内存，建立相应的进程，进入下一级的调度。因此，Linux 系统没有作业调度的概念。

3.3.2　调度算法的目标

从操作系统的设计角度，如何选择作业调度及进程调度的方式和算法，都要依据操作系统的类型及设计目标。调度算法的主要目标是提高系统的功能和效率。不同的操作系统会有差别，根据不同操作系统的目标，会有不同的调度算法，如一种算法可能有利于某一类作业或进程的运行，而不利于其他类作业或进程。如，在批处理系统、分时系统和实时操作系统中，通常采用不同的调度算法。

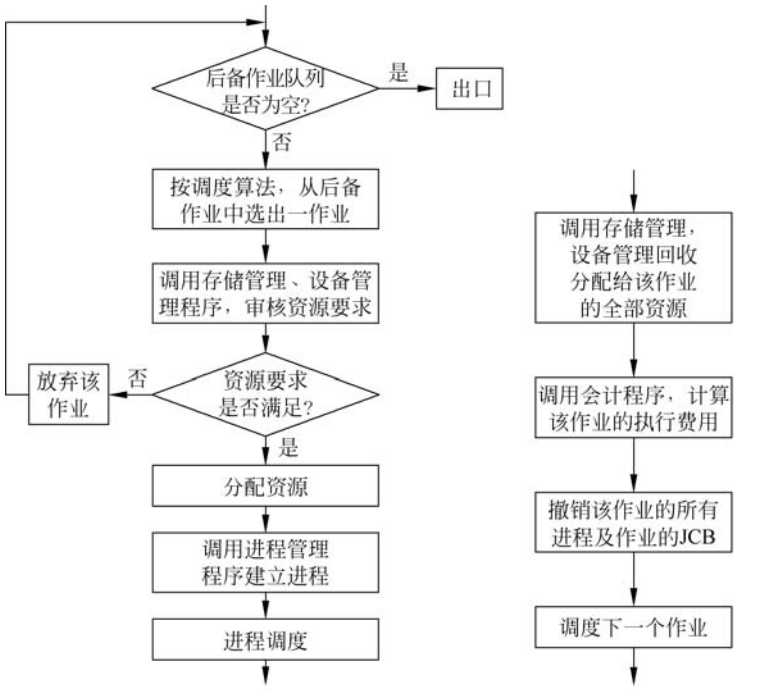

图 3-2　作业调度与进程调度

1. 面向系统的目标

设计调度算法时应考虑整个系统的效率，具体包括如下因素：

(1) 处理机及各类资源的利用率。提高资源的利用率就是提高系统的效率，要尽可能使系统中的处理机和各类资源保持有效的忙碌状态。

(2) 平衡性。系统中可能存在多种类型的作业及进程，有的属于计算类型，有的属于输入输出类型，使系统中的 CPU 和各种外设都能经常处于忙的状态，调度算法应尽可能保持系统资源的平衡使用。

(3) 优先权准则。在批处理、分时和实时系统中选择调度算法时都可遵循优权准则，以便使某些紧急作业能够得到及时处理。

(4) 公平性。系统应使各进程都能获得合理的 CPU 时间，不会发生进程饥饿现象。公平性是相对的，对相同类型的进程获得相同的服务，对于不同类型的进程，应提供不同的服务。

2. 批处理系统的目标

(1) 周转时间短。通常把周转时间的长短作为衡量批处理系统的性能、选择作业调度方式与算法的重要准则之一。

周转时间：从作业被提交给系统开始到作业终止为止的这段时间间隔，称为作业周转时间，用 T 表示。它包括作业在外存后备队列上等待调度的时间、进程在就绪队列上等待

进程调度的时间、进程占用CPU执行的时间和进程等待I/O操作完成的时间。对于一个用户来说,作业周转时间越短越好。

作业 J_i 的周转时间 T_i 定义为

$$T_i = T_{ei} - T_{si}$$

其中,T_{ei} 为作业 J_i 的完成时间;T_{si} 为作业 i 的提交时间。

也可将作业的周转时间 T_i 定义为

$$T_i = T_{i等待} + T_{i运行}$$

其中,$T_{i等待}$ 为作业 J_i 等待的时间;$T_{i运行}$ 为作业 J_i 的实际运行时间。

作为计算机系统的管理者,总是希望平均周转时间最短,这不仅能有效地提高系统资源的利用率,而且还可使大多数用户感到满意。若一个作业流含有 n 个作业,每个作业的周转时间为 T_i,则它的平均周转时间 T 为

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

一个作业的周转时间 T 与系统为它提供服务的时间(即作业要求运行时间)T_s 之比称为带权周转时间,即带权周转时间表示为

$$W = T/T_s$$

因为周转时间 T=等待时间+运行时间,因此,W 也可表示为

$$W = 1 + \frac{等待时间}{运行时间}$$

从公式可以看出,W 值的变化范围是 $W \geqslant 1$,即 $W=1$ 为 W 取值的最小值。可以看出,带权周转时间越接近1,该作业相对等待时间越短,系统性能越高。

平均带权周转时间可表示为

$$\overline{W} = \frac{1}{n}\left[\sum_{i=1}^{n} \frac{T_i}{T_{si}}\right]$$

(2) 系统的吞吐量高。吞吐量是指在单位时间内系统所完成的作业数,它与批处理作业的平均长度有密切关系。

(3) 处理机利用率高,各类资源的平衡利用。尽量使I/O繁忙的作业(即输入输出型作业)与CPU繁忙的作业(即计算型作业)搭配运行,使CPU及各种外设并行执行。

3. 分时系统的目标

(1) 响应时间快。常用响应时间的长短来评价分时系统的性能。响应时间是指从用户提交一个作业请求开始,直至系统首次产生响应为止的时间。它包括从键盘输入的请求信息传送到处理机的时间、处理机执行响应处理的时间和将所形成的响应信息在终端显示器上显示出来的时间。

(2) 均衡性。用户对响应时间的要求并非完全相同,通常用户对较复杂的任务允许响应时间较长,而对于较简单的任务响应时间则较短。均衡性是指系统响应时间的快慢与用户所请求服务的复杂性相适应。

4. 实时系统的目标

(1) 截止时间的保证。这是选择实时调度算法的重要准则。截止时间是指某任务必

须完成的最晚时间。对于严格的时实系统,它的调度方式和算法必须满足截止时间的要求。

(2) 可预测性。系统根据用户需求的可预测性,选择适当的调度算法实现预测的功能,进而提高系统的实时性。

3.4 进程调度

一般情况下,在单 CPU 系统中,处在就绪状态的用户进程数可能不止一个。同时,系统进程也可能需要使用处理机,这就要求进程调度程序按一定策略,动态地把处理机分配给处于就绪队列中的某一个进程,使之执行。本节介绍进程调度的功能、进程调度发生的时机以及进程调度引起的进程上下文切换等。

3.4.1 进程调度的功能

进程调度的功能可总结为如下几个方面。

1. 记录系统中所有进程的执行情况

在进行进程调度以前,进程管理模块已将系统中各进程的执行情况和状态特征记录在各进程的 PCB 中。而且,进程管理模块根据各进程的状态特征和资源需求,将各进程的 PCB 表排成相应的队列并进行动态队列转接。进程调度模块通过各进程 PCB 的变化,掌握系统中所有进程的执行情况和状态特征,并在适当的时候从就绪队列中选出一个进程占据处理机。

2. 从就绪状态队列中选择一个进程

进程调度的主要功能是在就绪状态的进程 PCB 队列中,按照一定的策略选择一个进程,分配给处理机并使之执行。不同的系统设计目的对应有不同的选择策略。例如,静态优先数算法系统开销较少,因此适合于分时系统的轮转法和多级反馈队列轮转法。策略的选择决定了调度算法的性能。

3. 进行进程上下文的切换

一个进程的上下文包括进程的状态、有关变量和数据结构的值、机器寄存器的值和 PCB 以及有关程序、数据等。一个进程的执行必然是在进程的上下文中执行。当正在执行的进程由于某种原因让出处理机时,系统要做进程上下文切换,以使调度程序新选中的进程得以执行。当进程上下文切换时,系统应首先检查是否允许做上下文切换(有些情况下,上下文切换是不允许的,如系统正在执行某个不允许中断的原语时);然后,系统要保留有关被切换进程的相关信息,以便切换回该进程时顺利恢复该进程的执行。在系统保留了 CPU 现场之后,调度程序从就绪状态队列中选择一个进程,并装配该进程的上下文,将 CPU 的控制权交给它。

3.4.2 进程调度的时机

在并发执行的环境下，有如下引起进程调度的事件。

1. 完成任务

正在执行的进程运行完成，主动释放对 CPU 的控制。

2. 等待资源

由于等待某些资源或事件发生，正在运行的进程不得不放弃 CPU，进入阻塞状态。

3. 运行时间已到

在分时系统中，当前进程使用完规定的时间片，时钟中断，使该进程让出 CPU。

4. 进入睡眠状态

执行中的进程自己调用阻塞原语将自己阻塞起来，进入睡眠状态。

5. 发现标志

执行完系统调用，即核心处理完陷入事件后，从系统程序返回用户进程时，可认为系统进程执行完毕，从而可调度新的用户进程执行。

以上都是在 CPU 执行不可剥夺方式下引起进程调度的原因。当 CPU 执行方式是可剥夺时，引起进程调度的事件除上述五种外，还应再加一条：优先级变化，如以下标题 6 所述。

6. 优先级变化

就绪队列中某进程的优先级高于当前执行进程的优先级时，将引起进程调度。

只要上述几种原因之一发生，操作系统就将进行进程调度。

3.4.3 进程上下文的切换

进程的上下文由正文段、数据段、硬件寄存器的内容以及有关数据结构组成。硬件寄存器主要包括存放 CPU 将要执行的下条指令虚地址的程序计数器、指出机器与进程相关联的硬件状态的处理机状态寄存器 PS、存放过程调用时所传递参数的通用寄存器 R 和堆栈指针寄存器 S 等。数据结构包括 PCB 等在内的所有与执行该进程有关的管理和控制用表格、数组、链等。当进程调度发生时，系统要做进程上下文的切换。进程上下文的切换主要包括以下四个方面。

1. 决定是否要做以及是否允许做上下文切换

检查分析进程调度原因，以及当前执行进程的资格和 CPU 执行方式的检查等。在操作系统中，进程上下文切换程序在如上所述时机进行上下文的切换。

2. 保存当前执行进程的上下文

这里所说的当前执行的进程是指要停止执行的进程。如果上下文切换程序不是被那个当前执行进程所调用,且不属于该进程,则所保存的上下文应该是先前执行进程的上下文,以便该进程下次执行时恢复它的上下文。

3. 选择一个处于就绪态的进程

按照某种进程调度算法选择一个处于就绪态的进程。

4. 使被选中的进程执行

恢复被选中进程的上下文,给它分配处理机,使之执行。

3.4.4 Linux 系统中进程调度发生的时机

对于 Linux 系统,没有设置专门的调度进程,需要进程调度时,调用一个特定的调度函数来完成该功能。通常,Linux 系统中进程调度发生的时机有以下几种。

(1) 用户创建一个新的进程时,系统把它加到就绪队列中,返回该进程的进程标识号。这时,调度函数开始执行,这样可以保证系统具有很好的响应特性。

(2) 正在执行的进程申请资源或等待某个事件的发生,而得不到满足时,该进程进入等待状态;当正在执行的进程完成了任务或得到特定的信号而退出,将转入僵死状态。这两种进程状态转换完成后,调用调度函数,选择新的进程分配给 CPU。

(3) 分时系统中,进程执行完一定的时间片后,释放 CPU 资源。

(4) Linux 系统提供了两级保护,用户进程可以在用户态和核心态下运行。具有不同的特权级别,可以访问不同的地址空间。用户进程可以在用户态和核心态这两种模式之间进行切换,用户态通过中断或系统调用就可以转入核心态。其中,中断是应进程外部发来信息的要求而转入核心态,函数调用则是应进程内部要求转入核心态;而从核心态切换到用户态,则需要一定的硬件支持。当进程从核心态返回到用户态时,将调用调度函数,发生调度。

从根本上说,这些情况可以归结为两类:一是进程本身自动放弃处理机,发生调度,包括进程转换到等待和僵死状态,这类调度是用户进程可以预测的;二是由核心态转入用户态时发生调度,这类调度发生最为频繁。系统调用完成和内核处理完中断之后,系统由核心态转入用户态,时间片用完使系统发送的时钟中断,本质上也是一种中断,而就绪队列加入新进程的工作只能由内核操作完成,无疑是发生于内核态。

3.5 调度算法

本节讨论各种典型的作业调度算法和进程调度算法。在操作系统中,调度的实质是一种资源分配,因而调度算法是根据系统的资源分配策略,规定资源分配的算法。现有的各种调度算法中,有的适用于作业调度,有的则适用于进程调度,也有一些既适合于作业调度,又

适合于进程调度。对于不同的系统和调度目标，应采用不同的调度算法。例如，在批处理系统中，考虑作业的平均周转时间、平均带权周转时间等因素，采用的调度算法有先来先服务调度算法、短作业优先的调度算法、优先级调度算法、高响应比优先调度算法等；在分时系统中，为了保证系统具有合理的响应时间，通常采用的调度算法有轮转等；在实时操作系统中，通常采用的调度算法有优先级调度算法、最低松弛度优先等。下面介绍这些典型的调度算法。

3.5.1　先来先服务调度算法


视频讲解

先来先服务(First Come First Served，FCFS)调度算法是一种简单的调度算法，它既适用于作业调度，也适用于进程调度。

先来先服务算法是按照作业或进程到达的先后次序来进行调度。当在作业调度中采用该算法时，每次调度都是从后备队列中选择一个最先进入该队列的作业，将它调入内存，为其创建进程、分配相应的资源，将该作业的进程放入就绪队列。在进程调度中采用该算法时，每次调度是从就绪队列中选择一个最先进入该队列的进程，并给它分配处理机。

FCFS算法比较利于长作业或进程，而不利于短作业或进程。

例 3-1　现有四个作业，假设它们按顺序依次到达，但到达的前后时间忽略不计。要求运行时间分别为2s、60s、2s、2s，如表3-3(a)所示。假设作业提交时刻为0。若系统按照FCFS算法进行作业调度，要求计算各作业的开始运行时间、运行结束时间、周转时间和带权周转时间。

通过计算，运算结果如表3-3(b)所示。

表 3-3　FCFS 算法示例

(a) 各作业运行时间

作业名	到达次序	运行时间/s
A	1	2
B	2	60
C	3	2
D	4	2

(b) 计算结果

作业名	到达次序	运行时间/s	开始执行时间	完成时间	周转时间/s	带权周转时间
A	1	2	0	2	2	1
B	2	60	2	62	62	1.03
C	3	2	62	64	64	32
D	4	2	64	66	66	33
平均值					48.5	16.76

从表3-3(b)可以看出，作业C的带权周转时间长达32，作业D的带权周转时间也较大。如果作业D的要求运行时间长一些，则它的带权周转时间就会相应地降低。

3.5.2 短作业(进程)优先调度算法

短作业优先调度(Shortest Job First,SJF)算法或短进程调度(Shortest Process First,SPF)算法是指对短作业或短进程优先调度的算法。这里,作业或进程的长短是以作业或进程要求运行时间的长短来衡量的。在把短作业优先调度算法作为作业调度算法时,系统将从外存后备作业队列中选择估计运行时间最短的作业,优先将它调入内存执行。这种算法适合作业调度和进程调度。

例 3-2 仍然用例 3-1 中的四个作业,采用 SJF 算法对它们的开始执行时间、运行结束时间、周转时间和带权周转时间进行讨论。对于相同长度的作业,通常按照 FCFS 算法执行。通过计算,可得如表 3-4 所示的结果。

表 3-4 SJF 算法示例

作业名	到达次序	运行时间/s	开始执行时间	完成时间	周转时间/s	带权周转时间
A	1	2	0	2	2	1
B	2	60	6	66	66	1.1
C	3	2	2	4	4	2
D	4	2	4	6	6	3
平均值					19.5	1.8

表 3-3 列出了各进程的开始执行时间、完成时间、周转时间和带权周转时间。比较表 3-3(b)和表 3-4 可以看出,SJF 算法给短作业带来明显的改善,同时降低了作业的平均周转时间,提高了整个系统的性能。

SJ(P)F 算法也存在一些不容忽视的缺点。

(1) 该算法对长作业不利。若系统不断有短作业进入,将造成长作业无限期延迟而得不到调度。

(2) 该算法未考虑作业的紧迫度,因而不能保证紧迫作业的及时处理。

(3) 由于作业或进程的长短只是由用户估计的,而用户又可能有意无意地缩短作业的估计运行时间,因此不一定保证做到真正意义上的短作业优先调度,因此该调度算法经常作为其他调度算法的比较算法。

3.5.3 高响应比优先调度算法

该算法通常用于作业调度。

高响应比优先调度(Highest Response First,HRF),也称为 HRN(Highest Response Next)算法。FCFS 算法只考虑作业等待时间长短而忽略作业的长度,而 SJF 算法的主要不足是长作业的运行得不到保证,两者都具有片面性。为了克服这两种算法的缺点,可以采用一种折中的方法,既让短作业优先,又考虑系统内等待时间过长的作业,这就是 HRF 算法。该调度策略在考虑每个作业等待时间的长短的同时估计所需的执行时间长短,从中选出响应比最高的作业投入运行。响应比 R 可定义为

$$R=\frac{\text{已等待时间}+\text{要求服务时间}}{\text{要求服务时间}}$$

即

$$R=1+\frac{\text{已等待时间}}{\text{要求服务时间}}$$

所谓响应比 R 高者优先调度，是指每次挑选一个作业投入执行时，先计算此时后备作业队列中每个作业的响应比 R 值，选择一个 R 值最高者投入执行。注意公式中的“已等待时间”为一个动态数据，它随着作业等待时间的增长而增加，因此作业等待时间越长，响应比越大。因此，一个长期未被调度的作业，只要等待足够长的时间，总有机会成为响应比高者，从而得以执行；同样长度的作业，其 R 值可以从其等待调度的时间长短来区分；同样等待时间的作业，要求执行时间少的作业 R 值高。因此，该算法既照顾了短作业，又考虑了作业的等待时间。当然，利用该算法时，每次进行调度之前，都要先计算响应比，因此增加了系统的开销。

例 3-3　系统的作业调度采用高响应比优先调度算法，有四个作业先后进入后备状态队列，到达系统时间和所需运行时间如表 3-5 所示，说明各作业的运行顺序，计算各作业的周转时间、带权周转时间，及平均周转时间、平均带权周转时间。

表 3-5　HRF 算法示例

作业名	到达系统时间/ms	要求服务时间/ms
J1	0	20
J2	5	15
J3	10	5
J4	15	10

这四个作业的调度采用 HRF 算法，过程如下：

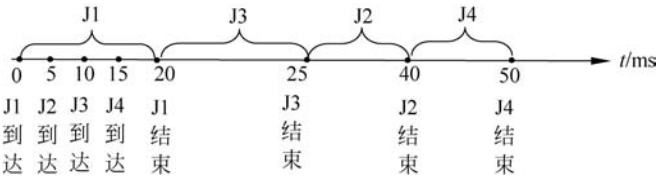

图 3-3　执行过程图

$t=0$ 时，只有作业 J1 到达，J1 执行，直到 $t=20$，作业 J1 执行结束。

$t=20$ 时，作业 J1 运行结束。作业 J2、J3、J4 都已到达后备队列，需要计算各作业的响应比。

$R_2=1+$J2 已等待时间/J2 需运行时间$=1+15/15=2$。

$R_3=1+$J3 已等待时间/J3 需运行时间$=1+10/5=3$。

$R_4=1+$J4 已等待时间/J4 需运行时间$=1+5/10=1.5$。

根据高响应比优先作业调度算法，作业 J3 的响应比 R_3 最大，所以选择 J3 运行。

$t=25$ 时，作业 J3 运行结束。作业 J2、J4 的响应比如下：

$R_2=1+$J2 已等待时间/J2 需运行时间$=1+(25-5)/15=2.3$。

$R_4=1+$J4 已等待时间/J4 需运行时间$=1+(25-15)/10=2$。

根据高响应比优先作业调度算法，作业 J2 的响应比 R_3 最大，所以选择 J2 运行。

$t=40$ 时，作业 J2 运行结束。作业 J4 开始运行，直到 $t=50$ 作业 J4 运行结束。

因此，这四个作业的运行顺序为：J1→J3→J2→J4。

周转时间和带权周转时间分别为：

$$T_1=(20-0)\text{ms}=20\text{ms} \quad W_1=20/20=1$$
$$T_2=(40-5)\text{ms}=35\text{ms} \quad W_2=35/15=2.33$$
$$T_3=(25-10)\text{ms}=15\text{ms} \quad W_3=15/5=3$$
$$T_4=(50-15)\text{ms}=35\text{ms} \quad W_4=35/10=3.5$$

平均周转时间 $\overline{T}=(T_1+T_2+T_3+T_4)/4=(20+35+15+35)\text{ms}/4=26.25\text{ms}$

平均带权周转时间 $\overline{W}=(W_1+W_2+W_3+W_4)/4=(1+2.33+3+3.5)=2.46$

3.5.4 优先级调度算法

优先级调度(Hightest Priority First，HPF)算法也叫优先权调度算法，可用于作业调度和进程调度。这种算法可用于批处理系统，也可用于实时系统。

系统或用户按某种原则，为作业或进程指定一个优先级。在进程或作业进行调度时，调度程序根据优先级进行调度。

1. 优先级的类型

在优先级调度算法中，优先级用来表示作业或进程所享有的调度优先权。该算法的关键是确定进程或作业的优先级。优先级有两类：静态优先级和动态优先级。静态方法根据作业或进程的静态特性，在作业或进程开始执行前就确定它们的优先级，一旦开始执行后就不能改变；动态算法则不同，它把作业或进程的静态特性和动态特性结合起来确定作业或进程的优先级，随着作业或进程的执行，其优先级不断发生变化。

(1) 静态优先级。静态优先级是在创建进程时确定的，且在进程的整个运行期间保持不变。通常，优先级是利用某一范围内的一个整数来表示的。例如，0～7 或 0～255 的某一整数，把该整数称为优先数。优先数具体用法有所不同，有的系统优先数大的优先级别高，而有的系统正好相反，优先数小的优先级别高，如 UNIX 系统。

确定进程优先级的依据有如下三个方面：

① 进程类型。通常，系统进程的优先级高于用户进程的优先级。对于系统进程，也可以根据其所要完成的功能划分为不同的类型。例如，调度进程、I/O 进程、中断处理进程、存储管理进程等。这些进程还可进一步划分成不同类型和赋予不同的优先级。例如，在操作系统中，对键盘中断的处理优先级和对电源掉电中断的处理优先级是不同的。

② 进程对资源的需求。例如，根据估计所需处理机的时间、内存需求、I/O 设备的类型及数量等来确定作业的优先级。

③ 用户要求。用户根据作业的紧急程度确定一个适当的优先级。当用户将自己的作业赋一个较高的优先级时，系统对该用户收取较高的费用。

通常，将作业的静态优先级作为它所属进程的优先级。

(2) 动态优先级。动态优先级是指在创建进程时所赋予的优先级，是随进程的推进或

等待时间的增加而改变的，以便获得更好的调度性能。例如，可以规定，在就绪队列中的进程，随其等待时间的增长，其优先权也越来越高；处在运行状态的进程，它的优先级越来越低。若所有进程都具有相同的优先权初值，则最先进入就绪队列的进程，将因其动态优先权变得最高而优先获得处理机，这也就是 FCFS 算法。若所有的就绪进程具有各不相同的优先权初值，那么对于优先权初值低的进程，在等待了足够的时间后，其优先权便可能升为最高，从而获得处理机。高响应比优先算法实际上就属于动态优先级调度算法。

2. 优先级调度算法的类型

根据调度程序是否可以在一个作业或进程运行过程中抢占，优先级调度算法又分为非抢占式优先级调度算法和抢占式优先级调度算法。

(1) 非抢占式优先级调度算法。若进程调度采用非抢占式优先级调度算法，系统在就绪队列中选择一个优先级最高的进程，分配给处理机后，该进程便一直执行下去，直至完成；只有因发生其他事件，使该进程进入阻塞状态，该进程才放弃处理机，系统在就绪态进程中，再找另一个优先级最高的进程，将处理机分配给它。这种调度算法主要用于批处理系统中，也可用于某些对实时性要求不严的实时系统中。

(2) 抢占式优先级调度算法。若进程调度采用抢占式优先级调度算法，系统在就绪队列中选择一个优先级最高的进程，分配给处理机，使之执行。在其执行期间，如果又出现了另一个优先级更高的进程，进程调度程序就立即停止当前进程的执行，重新将处理机分配给新到的优先级更高的进程。因此采用这种调度算法时，每当系统中出现一个新的就绪进程 i 时，就将其优先级 P_i 与正在执行的进程 j 的优先级 P_j 进行比较，如果 $P_i \leqslant P_j$，则原进程 P_j 继续执行；如果 $P_i > P_j$，则立即停止 P_j 的执行，做进程切换，使 i 进程投入执行。显然，这种抢占式优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

例 3-4　系统中有四个进程，情况如表 3-6 所示，进程调度采用抢占式优先级调度算法，优先数小的优先级别高。

(1) 写出进程执行的顺序。

(2) 计算每个进程所在作业的周转时间，以及平均周转时间。

表 3-6　四个进程的情况

进程名	到达时间	需要运行时间/min	优先数
A	10:00	40	5
B	10:20	30	3
C	10:30	50	4
D	10:50	20	6

解：(1) 各作业进入时间和结束时间如表 3-7 所示。

表 3-7　作业开始时间和结束时间

作业名	到达时间	开始运行时间	结束时间
A	10:00	10:00	10:20
B	10:20	10:20	10:50

续表

作业名	到达时间	开始运行时间	结束时间
C	10:30	10:50	11:40
A	10:00	11:40	12:00
D	10:50	12:00	12:20

作业执行的序列为 A→B→C→A→D,如图 3-4 所示。

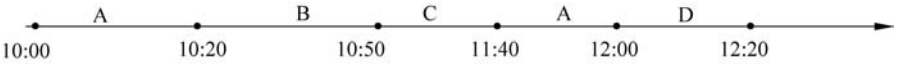

图 3-4 例 3-4 作业执行的序列

(2) 根据周转时间=完成时间—提交时间,可得各作业的周转时间为:

$$T_A = 12:00 - 10:00 = 120(\text{min})$$

$$T_B = 10:50 - 10:20 = 30(\text{min})$$

$$T_C = 11:40 - 10:30 = 70(\text{min})$$

$$T_D = 12:20 - 10:50 = 90(\text{min})$$

$$\text{平均周转时间 } \overline{T} = (T_A + T_B + T_C + T_D)/4 = 77.5(\text{min})$$

在这个题目中,注意进程调度算法是优先数小的优先级别高;再者,进程调度采用的是抢占式优先级调度算法,如果是非抢占式调度算法,结果也是不同的。这里用的是静态优先级,如果是动态优先级,则优先级需要用一个包含变量的表达式来表示。

例 3-5 有一个具有两道作业的批处理系统,作业调度采用短作业优先的调度算法,进程调度采用以优先数为基础的抢占式调度算法,假设优先数小的优先级别高。表 3-8 所示为作业序列。

(1) 列出所有作业进入内存的时间及作业结束时间。

(2) 计算平均周转时间。

表 3-8 例 3-5 的作用序列

作业名	到达时间	估计运行时间/min	优先数
A	10:00	40	5
B	10:20	30	3
C	10:30	50	4
D	10:50	20	6

解:两道作业的批处理系统是指内存中能同时容纳两道作业,即系统最多能有两道作业并发执行。作业只有被作业调度程序选中,进入内存后,才能够进行进程调度。

10:00 时,作业 A 到达。因系统的后备作业队列中没有其他作业,进程就绪队列中也没有进程,故作业调度程序将作业 A 调入内存并将它排到就绪队列上,进程调度程序选中它并运行。

10:20 时,作业 B 到达。因系统的后备作业队列中没有其他作业,所以该作业被作业调度程序选中,加载到内存。作业 B 的优先级高于作业 A 的优先级,进程调度程序暂停作业

A 的进程运行，将作业 A 放入就绪队列，调度作业 B 运行。此时，系统中已有两道作业。作业 A 已运行 20min，还需运行 20min。

10:30 时，作业 C 到达。因系统已有两道作业，所以作业 C 只能在后备作业队列中等待作业调度。此时，作业 B 已运行了 10min，还需运行 20min 才能完成。

10:50 时，作业 B 运行 30min 后结束运行，作业 D 到达。作业后备队列中有 C、D 两道作业，按照短作业优先的作业调度算法，作业 D 被作业调度程序选中，作业 C 仍在后备队列中等待。在内存中，作业 A 的优先级高于作业 D，因此作业 A 先运行。需 20min 才能完成。

11:10 时，作业 A 运行结束。系统中只有作业 D 在内存中运行，作业调度程序将作业 C 装入内存运行，因作业 C 的优先级高于作业 D，进程调度程序选中作业 C 运行，作业 D 等待。

12:00 时，作业 C 运行 50min 后结束运行，进程调度程序选中作业 D 运行。

12:20 时，作业 D 运行 20min 后结束运行。

总结以上分析，可知：

(1) 各作业进入时间和结束时间如表 3-9 所示。

表 3-9　作业时间

作业名	进 入 时 间	结束时间
A	10:00	11:10
B	10:20	10:50
C	11:10	12:00
D	10:50	12:20

作业执行的序列为 A→B→B→A→C→D，如图 3-5 所示。

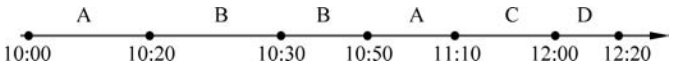

图 3-5　作业执行的序列

(2) 根据周转时间＝完成时间－提交时间，可得各作业的周转时间为

$$T_A = 11:10 - 10:00 = 70(\text{min})$$

$$T_B = 10:50 - 10:20 = 30(\text{min})$$

$$T_C = 12:00 - 10:30 = 90(\text{min})$$

$$T_D = 12:20 - 10:50 = 90(\text{min})$$

$$\text{平均周转时间} = (T_A + T_B + T_C + T_D)/4 = 70(\text{min})$$

3.5.5　时间片轮转调度算法

视频讲解

时间片轮转调度(Round-Robin，RR)算法主要用于分时系统中的进程调度。

轮转调度的实现原理为系统把所有就绪进程按先入先出的原则排成一个队列，新来的进程加到就绪队列末尾，每当执行进程调度时，就绪队列的队首进程总是先被调度程序选中，在 CPU 上运行一个时间片的时间。时间片是一个小的时间单位，通常为 10～100ms 数量级。当进程用完分给它的时间片后，系统的计时器发出时钟中断，调度程序进行调度，停止该进程的运行，并把它放入就绪队列的末尾；随后，进行进程切换，把 CPU 分给就绪队列的队首进程，同样让它运行一个时间片，如此往复。时间片轮转调度算法的原理如图 3-6 所示。

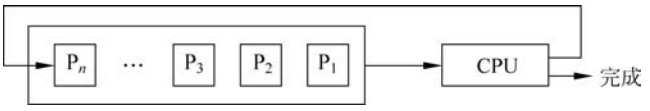

图 3-6 时间片轮转调度算法原理

在时间片轮转调度算法中,时间片的大小对系统的性能会造成较大影响。时间片轮转调度算法是一种剥夺性算法,因为进程切换会占用 CPU 的时间,这个开销与时间片大小有关。因此,若时间片很小,进程调度频繁发生,会增加系统的开销;反之,如果时间片选择太长,随着就绪队列中进程数量的增加,轮转一次所需的总时间增长,就会造成对每个进程的响应速度变慢,特别地,如果时间片大到让一个交互式进程足以完成这一次任务,那么时间片轮转调度算法便退化成 FCFS 算法。通常情况下,时间片的大小是由系统来确定的,为了满足用户对响应时间的要求,解决方法是,要么限制就绪队列中进程的个数,要么采用变化的时间片长度,依据当前负载情况及时调整时间片的大小。所以,现代操作系统在确定时间片长度时,要从进程的数量、进程切换所需要的开销、CPU 利用率、响应时间长短等多个因素进行考虑。

时间片的长短通常由以下几个因素确定。

(1) 系统的响应时间。在进程数目一定时,时间片的长短正比于系统对响应时间的要求。一个较为可取的时间片大小的确定方法是,时间片略大于一次典型的交互时间,使得大多数交互式进程能在一个时间片内完成,从而达到交互式进程响应时间短的目的。

(2) 就绪队列中进程的数目。当系统要求的响应时间一定时,就绪队列中的进程数越多,时间片应越小。

(3) 进程的转换时间。若进程的转换时间为 t,时间片为 q,为保证系统开销不大于某个标准,应使比值 t/q 不大于某一数值,如 1/10。

(4) CPU 运行指令速度。CPU 运行速度快,时间片可以短些;反之,则应取得长些。

在时间片轮转调度算法中,进程切换的时机可以分为两种情况:①若一个时间片还没用完,当前进程已经运行完成,此时立即激活调度程序,进行进程切换,将当前进程撤销,选择一个就绪进程分配一个时间片;②基于时钟中断的进程调度,只有当一个时间片用完时,才进行进程切换。现代操作系统在时间片轮转算法中,常常采用第一种方法。

例 3-6 考虑下述四个进程 A、B、C、D 的执行情况。设它们依次进入就绪队列,但前后时间忽略不计。四个进程分别需要运行 12、5、3、6 个时间单位,如表 3-10(a)所示。计算当时间片分别为 $q=1$、$q=4$ 时各进程的开始运行时间、完成时间、周转时间及带权周转时间。假设进程切换时机是一个进程运行完成时立即切换。

表 3-10 例 3-6 的表格及算法比较

(a) 各进程运行时间

进程名	到 达 时 间	运行时间(时间单位)
A	0	12
B	0	5
C	0	3
D	0	6

续表

（b）当 $q=1$、$q=4$ 时时间片轮转调度算法的比较							
进程名 到达时间		到达 时间	运行 时间	开始 时间	完成 时间	周转 时间	带权周 转时间
时间片 $q=1$	A	0	12	0	26	26	2.17
	B	0	5	1	17	17	3.4
	C	0	3	2	11	11	3.67
	D	0	6	3	20	20	3.33
	平均周转时间 $\overline{T}=18.5$　　平均带权周转时间 $\overline{W}=3.14$						
时间片 $q=4$	A	0	12	0	26	26	2.17
	B	0	5	4	20	20	4
	C	0	3	8	11	11	3.67
	D	0	6	11	22	22	3.67
	平均周转时间 $\overline{T}=19.75$　　平均带权周转时间 $\overline{W}=3.38$						

解：（1）当 $q=1$ 时，通过分析和计算可以得到表 3-10(b)所示的结果。

（2）当 $q=4$ 时，各进程执行情况如图 3-7 所示。

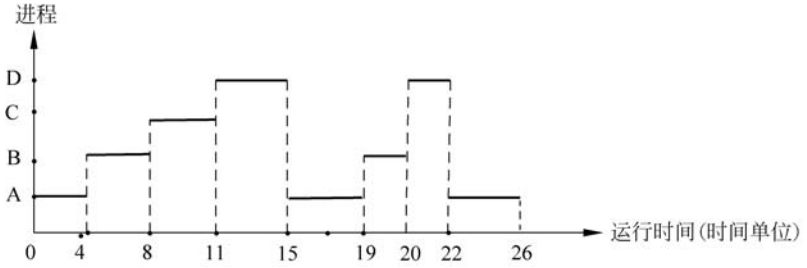

图 3-7　$q=4$ 时进程执行情况

从上面的比较可以看出，时间片的大小对时间片轮转调度算法的性能有较大影响。如果时间片足够长，使每个进程都能在这段时间内运行完毕，那么时间片轮转调度算法就退化为先来先服务算法，显然，对用户的响应时间必然大大加长。但是，如果时间片太短，那么CPU 在进程间的切换工作就非常频繁，从而导致系统开销增加，因为每个时间片末尾都产生时钟中断，进行进程切换，这个工作需要系统开销。

3.5.6　多级队列调度算法

该算法适合进程调度。

多级队列调度(Multiple-Level Queue，MLQ)算法是先来先服务调度算法、时间片轮转调度算法和优先级算法的综合。该调度算法的主要思想是将系统中的就绪进程，按其优先级高低不同，分为两级或多级队列。进程调度时，首先从高优先级就绪队列中挑选进程，只有高一级队列为空时，才从低优先级就绪队列中选取。每个队列中可按先来先服务(FCFS)算法或时间片轮转调度算法进行调度。在按时间片轮转调度算法调度时，每个就绪队列的时间片 q 值大小可以不同。一般说来，优先级高的就绪队列，其 q 值可设较小值。

进程的优先级可以事先给定。例如，在通用操作系统中既有终端的分时作业，又有批处

理作业。可以将终端上的分时用户进程定为高优先级,而把非终端用户进程定为低优先级。有的系统把使用I/O设备频繁的进程定为高优先级或中优先级,有的系统设更多的级,如系统作业、交互型作业、编辑型作业、批处理型作业、学生型作业等,这样,系统中就设置了多个级别的就绪队列,并赋予它们不同的优先级。也可以对每个队列采用不同的调度算法,如对系统作业可采用优先级优先调度算法,对交互性作业采用时间片轮转调度算法,对批处理型作业队列采用FCFS算法。仅当无系统作业时才运行交互作业;仅当无系统型作业和交互型作业时才运行编辑型作业;学生型作业队列优先级最低,只有当系统中无其他类型作业时才运行学生型作业。

这种算法的性能及实用性能很好,且容易实现,被目前流行的操作系统所采用,如UNIX、Linux和Windows NT等。

3.5.7 多级反馈队列调度算法

该算法适合进程调度。

1. 多级反馈队列调度算法的实现思想

该算法是在多级队列法的基础上加进“反馈”措施,具体描述如下。

(1) 系统中设置多个就绪队列,各个队列具有不同的优先级:第一个队列的优先级最高,第二个队列次之,以下各个队列的优先级逐个降低。

(2) 各就绪队列中进程的运行时间片不同,高优先级队列的时间片小,低优先级队列的时间片大,如从高到低依次加倍。

(3) 一个新进程进入系统后,首先被放入第一队列的末尾,各队列按FCFS方式排队,每个进程运行一个时间片;如某个进程在一次运行时没有完成工作,则把它转到下一级队列的末尾。

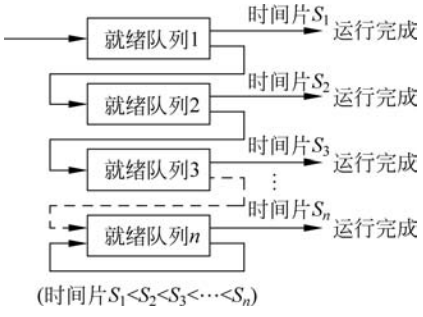

图3-8 多级反馈队列调度算法示意

(4) 系统先运行第一队列中的进程,第一队列为空后,才运行第二队列中的进程,以此类推。最后一个队列(最低级)中的进程仍然采用时间片轮转的方式进行调度。

(5) 当比运行进程更高级别的队列中有一个新进程时,它将抢占运行进程的处理机,而被抢占的进程回到原队列的末尾。

多级反馈队列调度算法示意图如图3-8所示。

2. 多级反馈队列调度算法的几点说明

(1) 照顾输入输出型进程是系统的宗旨。其目的在于充分利用外部设备,以及对终端交互及时地予以响应。输入输出型进程通常进入最高优先级队列,能很快得到处理机。另外,第一级队列的时间片的大小也应使之大于大多数I/O型进程产生一个输入输出所需的运行时间。这样既能使输入输出型进程得到及时的处理,也可避免过多进程间转接操作,减少了系统开销。

(2) 照顾了较短的作业。如果作业或进程仅在第一队列中执行一个时间片即可完成，作业的周转时间就非常短。对于稍长的作业，通常也只需在第二队列和第三队列各执行一个时间片即可完成。

(3) 计算进程总是用尽时间片，而由最高级逐次进入低级队列，虽然运行优先级降低了，等待时间也较长，但终究得到较大的时间片来运行。同时，对于长作业，响应时间较短。

(4) 在有些分时系统中，一个进程由于输入输出完成而要求重新进入就绪队列时，并不是将它放入最高级别的就绪队列，而是进入因输入输出而要求离开原来的一级就绪队列，这就需要对进程所在的就绪队列进行记录。这样做的好处是有些计算型进程，偶尔产生一次输入输出要求，输入输出完成后仍然需要很长的处理机运行时间，为减小进程的调度次数和系统开销，不让它从最高级队列逐次下降，而是直接放入原来所在的队列。

以上介绍了几种典型的调度算法。在单道批处理系统中，作业调度的主要任务是解决作业之间的自动接续问题，减少操作员的干预，提高系统资源的利用率。所采用的调度算法比较简单，通常采用先来先服务调度(FCFS)算法、短作业(进程)优先调度算法和高响应比优先调度算法。在多道批处理系统中，大多数多道程序系统的作业调度采用以下策略：先来先服务、考虑优先级、分时和优先级相结合、综合考虑资源要求等。分时系统通常采用时间片轮转调度算法、多级队列调度算法及多级反馈队列调度算法等调度策略。实时系统的调度通常采用如抢占式优先级、多级队列轮转及多级反馈队列轮转等策略。

3.6　Linux 系统的调度算法

Linux 系统作业调度非常简单，或者说没有作业调度，作业一旦提交，就直接进入内存，建立相应的进程，进入下一级的调度。交换调度主要涉及系统存储管理的内容，将在内存管理中进行研究。Linux 系统中的内核级线程和进程在表示、管理调度方面没有差别，系统也没有专门的线程调度，采用进程调度统一处理进程和内核级线程。因此，在此主要讨论 Linux 系统中的进程调度方法。

3.6.1　Linux 系统的进程调度策略

Linux 系统的调度程序就是内核中的 schedule()函数，它的主要任务是在就绪队列 run_queue 中选出一个进程并投入运行。schedule()函数需要确定以下参数：

(1) 进程调度算法 Policy。

(2) 进程过程中剩余的时间片 Counter。

(3) 进程静态优先级 Priority，在 Linux 2.4 版本中取消了这个变量，但是它所代表的默认值 20 作为常数还在继续使用。

(4) 实时进程的优先级 Rt_priority。

(5) 用户可控制的 Nice 因子。

这些参数的变量被存放在进程控制块中相应的调度成员中。

Linux 系统提供了三种进程调度算法，这三种算法可由用户通过宏定义来选择。可使用的调度策略如表 3-11 所示。

表 3-11 调度策略

调度策略标志	所代表的调度策略
# define SCHED_OTHER	普通的分时进程
# define SCHED_FIFO	先进先出的实时进程
define SCHED_RR	基于优先级的轮转算法(动态优先级调度)
# define SCHED_YIELD	不是调度策略,表示进程让出 CPU

内核把进程分为实时进程和非实时进程(普通进程)两种。实时进程获得 CPU 比普通进程优先。用户可以通过系统调用 sched_setschedule()函数改变自己的调度策略,通过系统调用 sys_setpriority()和 sys_nice()改变其静态优先级。一旦进程变为实时进程,它的子孙进程也是实时进程。

3.6.2 Linux 系统的优先级调度策略

进程的优先级是一些短整数,代表每个进程的相对获得 CPU 的权值。因此,进程的优先级越高,得到 CPU 的机会也就越大。Linux 内核又进一步把进程优先级分为静态优先级、动态优先级以及实时优先级。

1. 静态优先级

Linux 进程的静态优先级为 task_struct 结构中的一个分量 Priority。

进程静态优先级 Priority 是在进程建立时继承父进程的,用户可以通过系统调用 nice()或 setpriority()来设置,除此之外,Priority 在一个进程整个运行期间其值保持不变。在 Linux 1.0 版中,静态优先级的取值范围是 1～35,普通用户使用 nice()系统调用设置静态优先级时只能在这个范围内设置,其后的版本中新增的 setpriority()系统调用允许超级用户将一个进程的动态优先级定义为任意大的值。当然,如果将一个进程的动态优先级设得过大,系统将降级为一个单任务的操作系统。

2. 普通进程的调度策略

对于普通进程即非实时进程,Linux 的调度策略采用抢占式动态优先级调度算法。这是一种经常被采用的进程调度算法,这种算法不设进程就绪队列。进程调度子程序 schedule()总是选择动态优先级最高的进程来分配给 CPU,如图 3-9 所示。

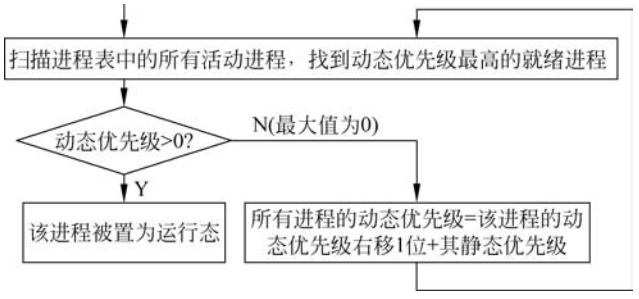

图 3-9 Linux 的进程调度过程

动态优先级是通过 task_struct 结构中的 Counter 分量定义的。Counter 表示在抢占式中断调度前该进程还能运行的时间，这个时间的单位是 tick，即时钟中断的发生周期。在 Linux 中，一个 tick 可以是 10ms，即每隔 10ms 发生一次时钟中断。因此，如果一个进程在某个时刻的动态优先级是 55，那么该进程这次使用 CPU 至少还能再连续运行 550ms，在此期间不能发生抢占式中断调度，即在此期间只能进行快中断处理，除非该进程因需等待 I/O 等原因而主动退出运行态，否则不能被抢占式中断调度强迫退出运行态。如果某个时刻，系统中共有四个就绪进程，它们的优先级分别是 2、20、30、10，那么调度程序显然会选中动态优先级为 30 的那个进程来使用 CPU。从时间片的角度来看，Linux 的时间片是动态时间片，同一时刻上不同进程的时间片是不同的，同一进程在不同时刻的时间片也是不同的，Linux 的进程调度算法总是选中当前时刻时间片最大的那个进程来使用 CPU。

进程动态优先级 Counter 的值的产生和变化与进程静态优先级有关。

在进程建立时，进程动态优先级的值是从父进程(当时的值)继承的。之后，在该进程存在期间，进程动态优先级是变化的。每当时钟中断发生时，当前运行进程的 Counter 值减 1，这种递减过程会在下列两种情况下停止。

(1) 当 Counter 递减到 0 时，进程就会在时钟中断处理过程中被迫从运行态进入就绪态。系统中处于就绪态的进程，其进程动态优先级不一定全为 0；对于那些从运行态进入就绪态的进程，其进程动态优先级一定为 0；对于那些从等待态进入就绪态的进程，其进程动态优先级通常不为 0。从运行态进入就绪态的进程，会在就绪态中一直保持动态优先级为 0，这样，不为 0 的那些进程会很快进入运行态，又很快进入等待状态或以 0 值进入就绪态，直到系统中的所有就绪态进程的动态优先级都为 0，再在进程调度程序中被重新置为各自进程的静态优先级的值，然后按动态优先级的值依次进入运行态运行。

(2) 对于处于运行态的进程，可能在 Counter 还没递减到 0 时，进程就因需等待某个事件的发生而进入了等待态。处于等待态的进程，其动态优先级通常逐渐增加，也可能不变，但不可能减小。

从图 3-9 可以看出 CPU 的调度过程。请注意该过程的“N”这一分支，这一步是当所有就绪进程的动态优先级都为 0 时，重新计算所有进程的动态优先级，所有就绪进程的动态优先级都被设为该进程的静态优先级。这一步有三点需要注意：

(1) 并不是当所有进程的动态优先级都递减到 0 时才一起这样做；

(2) 所有进程都参加了重新计算，包括就绪态进程和等待态进程；

(3) 并不是直接将静态优先级值赋给动态优先级，而是利用一个专用的计算公式给出 Linux 进程的动态优先级的计算方法。Counter 的计算公式为

```
p->counter = (p->counter >> 1) + p->priority      //p 为每个进程的 task 结构指针
```

对每个进程，将该进程的动态优先级的值右移一位(即除以 2 后取整)后，再加上该进程的静态优先级的值，就得到该进程的动态优先级的新值。对就绪进程，实质上仍是直接赋值(因为此时所有就绪进程的动态优先级已都为 0)。但对于处于等待态的进程来说，这个公式表示等待态进程的动态优先级大于其静态优先级，但不会比静态优先级的两倍更大。

Counter 的这种变化带来了如下效果：

(1) 等待态进程会有较高的优先级,而且处于等待态时间越长的进程,其进程动态优先级就越高(直至等于其静态优先级两倍,此后便不再变化,直至进入运行态),这就意味着,I/O 较多的进程(通常从运行态退出时是进入等待态)会有较高的优先级。

(2) 计算较多的进程(通常从运行态退出时是进入就绪态)会有较低的优先级,因为它不仅不会像等待进程那样增加优先级,而且会在就绪态等待较长时间。

大多数操作系统都有调度队列,如 Solaris 和 Windows 2000/NT,调度算法都采用多重队列动态优先级算法。Linux 中进程动态优先级的上述变化过程在一定程度上达到了多重队列调度算法的效果。

3.6.3 实时进程的调度策略

对于实时进程,Linux 系统使用两种调度策略:先来先服务调度算法和时间片轮转调度算法。因为实时进程对时间响应要求较高,为了保证实时进程能够优先于普通进程运行,内核为实时进程增加了第三种优先级,称为实时优先级。实时优先级被保存在进程控制块 PCB 的 Rt_priority 成员中,它是一个 0～99 的整数。实时进程使用静态优先级 Priority 常数和动态优先级 Counter 完成的功能与普通进程相同,但是 Counter 不作为实时进程调度的依据,这与普通进程不同。


视频讲解

3.7 死锁问题

计算机系统中有很多资源,如打印机、磁带机、一个文件的索引节点等,在多道程序设计环境中,若干进程往往要共享这类资源,而且一个进程所需的资源不止一个,这样,若干进程就会相互竞争有限的资源,因得不到满足而陷入阻塞状态。

系统发生死锁现象不仅浪费大量的系统资源,甚至导致整个系统崩溃,带来灾难性后果。所以,对死锁问题在理论上和技术上都必须给予高度重视,应采取一些有效的措施预防和避免死锁的发生,也可以采用相关的措施检测死锁。死锁一旦发生,要解除死锁,只有这样才能保证系统的安全与顺利执行。

3.7.1 死锁的概念

死锁是进程死锁的简称,是由 Dijkstra 于 1965 年研究银行家算法时首先提出来的,也是计算机操作系统乃至并发程序设计中非常重要但又最难处理的问题之一。掌握操作系统中对于死锁的处理方法,对于指导我们的现实生活会有积极的意义。

下面来看一个死锁的例子,如图 3-10 所示,假设某系统中,进程 P_1 和进程 P_2 并发执行。进程 P_1 已占有临界资源 R_1,进程 P_2 已占有临界资源 R_2。在以后运行的过程中,进程 P_1 再申请 R_2 将得不到满足,因此被加入到阻塞态队列;进程 P_2 又申请 R_1,因 R_1 被 P_1 占有,且 P_1 不能运行,因此不能释放 R_1,所以进程 P_2 也被放入阻塞态队列。这样,系统中的两个进程都因等待对方的临界资源而不能继续运行,这种现象称为死锁。该例中的临界资源 R_1 和 R_2

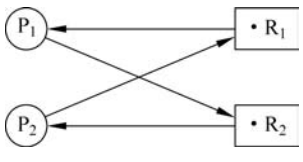

图 3-10 进程 P_1、P_2 陷入死锁状态

可能是打印机、磁带机和缓冲区等。

死锁是指多个进程循环等待其他进程占有的资源，因而无限期僵持下去的局面，也可以说死锁是进程之间无限期互相等待永不发生的事件。很显然，如果没有外界的作用，死锁中的各个进程将永远处于阻塞状态。

实际上，在前面的一些例子中，我们已经成功地解决过死锁问题，如生产者-消费者问题中，对于生产者进程，先做 wait(empty)，后进行 wait(mutex)；在消费者进程中，先做 wait(full)，再做 wait(mutex)，这种先后顺序也是为了避免死锁的发生。

研究死锁是为了让死锁不发生，这也就是死锁的预防和避免；在系统中应该能够检测死锁，当死锁发生时，要解除死锁。

3.7.2　解决死锁问题的基本方法

为保证系统的正常运行，应事先采取措施，预防或避免死锁的发生。在系统已出现死锁后，要及时检测到死锁并解除死锁。目前用于处理死锁问题的方法如下。

1. 死锁的预防

采取某种策略，限制并发进程对资源的请求，从而保证死锁的必要条件在系统执行的任何时间都得不到满足。

2. 死锁的避免

在分配资源时，根据资源的使用情况提前做出预测，给定一个合适、安全的进程推进顺序，从而避免死锁的发生。这种方法实现起来有一定难度，但在一些较完善的系统中常用这种方法。

3. 死锁的检测

允许系统发生死锁。系统设有专门的机构，当死锁发生时，该机构能够检测到死锁的发生，并能确定参与死锁的进程及相关资源。

4. 死锁的解除

这是与死锁检测相配套的措施，用于将进程从死锁状态中解脱出来。

由于操作系统的并发与共享以及随机性等特点，通过预防和避免死锁的手段达到排除死锁的目的十分困难，需要相当大的系统开销，对资源的利用也不够充分。死锁的检测与解除则相反，不必花费多少执行时间就能发现死锁并从死锁中恢复出来。因此，实际操作系统很多都采用了后两种方法。

3.7.3　产生死锁的原因及必要条件

1. 产生死锁的原因

从上例可以看出，系统产生死锁的根本原因可归结为以下两点。

(1) 各进程竞争有限的资源。系统提供的资源数量有限，远不能满足并发进程对资源

的需求。可将系统中的资源分为可剥夺性资源和不可剥夺性资源。可剥夺性资源是指某进程在获得这类资源后,该资源可被其他进程抢占,如处理机、内存。优先权高的进程可以剥夺优先权低的进程的处理机;内存区可由内存管理程序把一个进程从一个存储区移到另一个存储区,即剥夺了该进程原来占有的存储区。不可剥夺性资源是指进程一旦占有该资源就不可抢占,不管其他进程的优先权是否高于当前进程,只能在该进程用完后自行释放,如打印机、磁带机等。

大多数情况下,引起死锁的资源竞争是指对于不可剥夺性资源的竞争。另外一种因资源竞争而死锁的资源是一些临时性资源,也称为消耗性资源,这种资源是被另一进程使用很短的时间而以后便无用的资源,如消息等。

(2) 进程推进顺序不当。图 3-10 中,两进程并发执行,占有资源并申请另一资源,这时因各进程互相等待另一进程释放临界资源而陷入死锁状态。如果此时先让进程 P_1 运行,待进程 P_1 获得它所需的两个资源 R_1 和 R_2 后,马上让 P_2 开始运行,这样就可以避免死锁的发生。以后将要讨论的银行家算法就是要找到一个合适、安全的进程推进顺序,以保证系统的正常运行。

2. 死锁产生的必要条件

通过以上对于死锁的分析,可以看出,系统如果发生死锁,一定同时具备下列四个条件,即死锁的必要条件。

(1) 互斥条件。对于一个排他性资源,某一时刻最多只能由一个进程占有,不能同时分配给两个以上的进程。只有占有该资源的进程放弃该资源后,其他进程才能占有该资源,如打印机是临界资源,各进程必须互斥地使用。

(2) 占有且申请条件。进程至少已经占有一个资源,但又申请新的资源;由于该资源已被其他进程占有,此时该进程阻塞,但是它在等待新资源时,仍不释放已占有的资源。

(3) 不可抢占条件,也称为不剥夺条件。进程所获得的资源在未使用完毕之前,资源申请者不能强行从资源占有者进程夺取该资源,只能等待占有该资源的进程释放该资源后,其他进程才可以使用。

(4) 环路条件。存在一个进程等待序列$\{P_1,P_2,\cdots,P_n\}$,其中 P_1 等待进程 P_2所占有的资源,P_2 等待进程 P_3所占有的资源,…,P_{n-1}等待进程 P_n 所占有的资源,P_n 等待进程 P_1 所占有的资源,形成一个循环等待的环路。

上面四个条件是在死锁发生时同时出现的,可以利用它的逆否命题,即四个条件中只要有一个不满足,则死锁不会发生。这正是预防死锁所需要考虑的方法。


视频讲解

3.8 死锁的预防

在系统中采用各种方法都是为了防止死锁的发生,它们在实现上也分别采用了不同的原理:死锁的预防是根据死锁的四个必要条件,即只要有一个条件不满足,则死锁不发生。

死锁的必要条件中,第一个条件是“互斥条件”,对于可分配的资源要互斥使用,这是由资源的固有特性决定的,不可改变。因此,只有通过摒弃后面三个条件之一,使它们中的一条不成立来达到预防死锁的目的。

3.8.1 摒弃占有且申请条件

可以采用资源的静态预分配策略或释放已占资源策略。

1. 资源的静态预分配策略

在进程运行以前，一次性地向系统申请它所需的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不执行。只有当系统能够满足当前进程的全部资源的需求时，才一次性地将所申请的资源全部分配给该进程。这种方法存在一些缺点：

(1) 一般情况下，一个进程在执行期前不可能知道它所需的全部资源，因为进程执行时是动态的。

(2) 资源的利用率低。无论所分配资源何时用到，一个进程只有在占有所需全部资源后才能执行。

(3) 降低了进程的并发性，因此系统的效率不高。由于资源有限，能分配到全部资源的进程数量必然减少。

2. 释放已占资源策略

仅当进程没有占用资源时才允许它去申请资源。如果进程已经占用了某些资源而再申请资源，应先归还所占用的资源后再申请新资源。例如，进程对存放在磁带机上的文件进行处理，然后将处理结果打印输出。进程可以先申请磁带机，得到磁带机后进程就可以开始执行，启动磁带机，读出文件后进行处理，把处理后的文件保存在自己的工作区。然后释放磁带机，再申请打印机。得到打印机后，就可把处理好的文件打印输出。

这种方法仍会使一些进程处于等待资源状态。进程所申请的资源可能已被其他进程占用，只能等待占用者进程释放资源后，系统才可能分配给申请进程。

3.8.2 摒弃不可抢占条件

我们采取的策略是隐式抢占。约定如果一个进程已经占有了某些资源又要申请另外的资源，而被申请的资源不满足时，该进程必须等待，同时释放已占有的资源，以后再进行申请。它所释放的资源可以重新被分配给其他进程，这就相当于该进程占有的资源被隐式地抢占了，这种预防死锁的方法实现起来较为困难。

3.8.3 摒弃环路条件

实行资源的有序分配策略，即把资源事先分类编号，按序分配，使进程在申请、占有时不会形成环路。例如，令输入机的序号为 1、打印机的序号为 2、磁带机的序号为 3、磁盘的序号为 4 等。所有进程对资源的请求必须严格按照资源序号递增的次序提出，这样在资源分配图中，就能保证不再存在环路。

这些预防死锁的策略与前面两种策略相比，其资源利用率和系统吞吐量都有较为明显的改善。但也存在缺点，如各类资源所分配的序号必须相对稳定，这就限制了新类型设备的增加；虽考虑到大多数作业在实际使用资源时的顺序，但也经常会出现作业使用各类资源的

顺序与系统规定的顺序不同,造成对资源的浪费;为方便用户,系统对用户在编程时所施加的限制条件应尽量多,然而这种按规定次序申请的方法必然会限制用户自然、简单地编程。


视频讲解

3.9 死锁的避免

以上讲到的死锁的预防是排除死锁的静态策略,它使产生死锁的四个必要条件不能同时具备,从而对进程申请资源的活动加以限制,以保证死锁不会发生。死锁的避免是一种排除死锁的动态策略,给系统中并发执行的进程找到一个安全的推进顺序,而不限制进程有关申请资源的命令,对进程所发出的每一个申请资源的活动都加以动态的检查,并根据检查结果决定是否进行资源分配,即在资源分配过程中预测是否会出现死锁,如果不会死锁,则分配资源;若有发生死锁的可能,则加以避免。这种方法的关键是确保资源分配的安全性。

3.9.1 系统的安全状态

由于避免死锁的策略允许进程动态地申请资源,系统需提供某种方法在进行资源分配之前先计算资源分配的安全性,若此次分配不会导致系统进入死锁状态,则将资源分配给进程,否则进程等待。

安全状态是指系统中的所有进程能够按照某种次序分配资源,并且依次运行完毕,进程序列$\{P_1,P_2,\cdots,P_n\}$就是安全序列。如果存在这样一个安全序列,则系统是安全的,称此时系统处于安全状态。如果系统不存在这样一个序列,则称系统是不安全的。

例 3-7 有三个客户 C_1、C_2、C_3 向银行家贷款。该银行家的资金总额为 10 个资金单位,其中客户 C_1 要借 9 个资金单位,客户 C_2 要借 3 个资金单位,客户 C_3 要借 8 个资金单位,总计 20 个资金单位。若 T_0 时刻,客户占用和还需申请资源的状态如表 3-12 所示,银行家应如何分配资金?

表 3-12 例 3-7 客户占用和还需申请资源的状态

客户	已分配资源	还需申请资源
C_1	2	7
C_2	2	1
C_3	4	4

解:银行家手里剩余的资源单位数是 10−(2+2+4)=2。此时银行家只有将资金分配给 C_2 才能最终收回贷款。然后,再将资金依次分配给 C_3 和 C_1,这样银行家能最终收回所有贷款。这里就存在一个安全序列$\{C_2,C_3,C_1\}$,此时系统是安全的。

3.9.2 由安全状态向不安全状态的转化

如果不按照安全序列的顺序分配资源,系统可能由安全状态进入不安全状态。例如,若在 T_0 时刻以后,C_3 又申请到 2 个资金单位,则系统进入不安全状态。因为,此时无法再找到一个安全序列,结果造成系统产生死锁。由此可见,当 C_3 申请资源时,尽管当时系统中还有可用资源,但是不能分配给它,必须让它等待。按银行家的术语说,某客户若无偿还能力

时，就不要贷款给他。

3.9.3　银行家算法

最具有代表性的避免死锁的算法是 Dijkstra 的银行家算法，由于该算法可能用于银行现金贷款而得名。一个银行家把他的固定资金贷给若干顾客，只要不出现一个顾客借走所有资金后还不够，银行家的资金应是安全的。银行家需要一个算法保证借出去的资金在有限时间内可以收回。

假定顾客分成若干次进行贷款，并在第一次贷款时说明他的最大借款额。具体算法如下：

(1) 顾客的贷款操作依次顺序进行，直到全部操作完成。

(2) 银行家对当前顾客的贷款操作进行判断，以确定其安全性，看能否支持顾客贷款，即该客户能否运行完成。

(3) 安全时，贷款；否则，暂不贷款。

3.10　利用银行家算法避免死锁


视频讲解

3.10.1　银行家算法中的数据结构

1. 可利用资源向量 Available

可利用资源向量也称为空闲向量，是一个含有 m 个元素的数组。每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 Available$[j]=K$，则表示系统中现有 R_j 类资源 K 个。

2. 最大需求矩阵 Max

最大需求矩阵是一个 $n\times m$ 的矩阵，它定义了系统中 n 个进程中的每一个进程对 m 类资源的最大需求。如果 Max$[i,j]=K$，则表示第 i 个进程需要 R_j 类资源的最大数目为 K。

3. 分配矩阵 Allocation

分配矩阵也叫占有矩阵，是一个 $n\times m$ 的矩阵，它定义了系统中每一进程已占有的每一类资源数。如果 Allocation$[i,j]=K$，则表示第 i 个进程当前已分得 R_j 类资源的数目为 K。

4. 需求矩阵 Need

需求矩阵也叫申请矩阵，是一个 $n\times m$ 的矩阵，用以表示每一个进程尚需的各类资源数。如果 Need$[i,j]=K$，则表示第 i 个进程还需要 R_j 类资源 K 个才能完成任务。

显然，前三个矩阵之间存在如下关系：

$$\text{Need}[i,j]=\text{Max}[i,j]-\text{Allocation}[i,j]$$

3.10.2 银行家算法的实现

1. 进程申请资源的情况

设 $Request_i$ 是进程 P_i 的请求向量，如果 $Request_i[j]=K$，表示进程 P_i 需要 R_j 类资源的个数为 K。$Request_i$ 与 Need[i]的关系可能为以下三种情况。

(1) $Request_i>Need[i]$。这种情况表示该进程的资源需求已超过系统所宣布的最大值，因此认为出错。

(2) $Request_i=Need[i]$。这种情况表示该进程现在对它所需的全部资源一次申请完成。

(3) Requesti<Need[i]。这种情况表示该进程现在对它所需资源再进行部分的申请，剩余的资源以后可再次申请。

2. 银行家算法的描述

当进程 P_i 发出资源请求后，系统按下述步骤进行检查。

(1) 如果 $Request_i \leqslant Need[i]$，便转向步骤(2)；否则显示出错，因为它所需的资源数已超过它事先要求的最大值。

(2) 如果 $Request_i \leqslant Available$，便转向步骤(3)；否则，表示尚无足够资源，$P_i$ 须等待。

(3) 假设系统将资源分配给 P_i，则需修改如下数据结构的值：

```
Available := Available - Requesti;
Allocation[i] := Allocation[i] + Requesti;
Need[i] := Need[i] - Requesti;
```

(4) 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。如果是安全的，则将资源真正地分配给进程 P_i，否则，将本进程的试探分配作废，恢复原来的资源分配状态，进程 P_i 等待。

3. 安全性算法

工作向量 Work 表示在算法执行过程中，系统可提供给进程继续运行所需的各类资源数目，它含有 m 个元素，在执行安全算法开始时，初始值 Work := Available。

完成向量 Finish 表示系统能否运行完成。它有 n 个分量，分别表示各进程是否可执行完成。若 Finish[i] := true，表示第 i 个进程能够获得足够的资源，运行完成；若 Finish[i] := false，表示该进程不能获得所需全部资源，不能运行完成。

设初值 Finish[i] := false，$i=0,1,2,\cdots,n-1$。当有足够资源分配给第 i 个进程时，再令 Finish[i] := true。

安全算法的步骤如下。

(1) 设置两个工作向量。设置工作向量 Work、完成向量 Finish，并赋初值。

(2) 进行安全性检查。从进程集合中查找一个能满足下述条件的进程 i：

```
Finish[i]: = false
Needi≤Work;
```

若找到这样的进程，执行步骤(3)；若找不到这样的进程，则转步骤(4)。

(3)

```
Work := Work + Allocation[i];
Finish[i] := true;
```

返回步骤(2)。

因为进程 P_i 若能执行完成，则能够释放它所占的资源。

(4) 若所有进程的 Finish[i]：=true 都满足，则表示系统处于安全状态，正式将资源分配给进程 P_i；否则，系统处于不安全状态，系统不能进行这次试探性分配，恢复原来的资源分配状态，让进程 P_i 等待。

如果已判定系统处于安全状态，则通过运算过程同时可以找到一个安全序列。

银行家算法具有较好的理论意义。但由于在实际系统中，难以预见或获得各个进程申请的最大资源向量，且实际运行进程的个数是动态变化的，因此银行家算法在实际系统中难以实施，或实施过程代价过大。

3.10.3　银行家算法的应用

例 3-8　某系统有 A、B、C 类型的三种资源，在 T_0 时刻进程 P_1、P_2、P_3、P_4 对资源的占用和需求情况如表 3-13 所示，此刻系统可用资源向量为(2,1,2)。

表 3-13　各进程对资源的占用和需求情况

资源请求进程	最大需求量 Max			已分配资源 Allocation		
	A	B	C	A	B	C
P_1	3	2	2	1	0	0
P_2	6	1	3	4	1	1
P_3	3	1	4	2	1	1
P_4	4	2	2	0	0	2

(1) 将系统中各种资源总数和进程对资源的需求数目用向量或矩阵表示出来。

(2) 判定此刻系统的安全性。如果是安全的，写出安全序列，如果是不安全的，写出参与死锁的进程。

(3) 如果此时 P_1 和 P_2 均再发出资源请求向量 Request 为(1,0,1)，为了保持系统安全性，应如何分配资源给这两个进程？说明所采用策略的原因。

(4) 如果(3)中的请求都立刻满足后，系统此刻是否处于死锁状态？最终能否进入死锁状态？若能，说明参与死锁的进程；若不能，说明原因。

解：(1) 由题意可知，Available=(2,1,2)

$$\begin{aligned}\text{系统资源总数向量} &= \text{Available} + \text{Allocation}\\ &= (2,1,2) + (7,2,4)\\ &= (9,3,6)\end{aligned}$$

进程对资源的需求矩阵

$$\text{Need}=\text{Max}-\text{Allocation}=\begin{pmatrix}3&2&2\\6&1&3\\3&1&4\\4&2&2\end{pmatrix}-\begin{pmatrix}1&0&0\\4&1&1\\2&1&1\\0&0&2\end{pmatrix}=\begin{pmatrix}2&2&2\\2&0&2\\1&0&3\\4&2&0\end{pmatrix}$$

(2) 采用银行家算法进行计算步骤如下：

```
Work = Available = (2,1,2)
Finish = (false,false,false,false)
```

① 因为 Need[2]<Work，故系统可以满足 P_2 对资源的请求，将资源分配给 P_2 后，P_2 可执行完成，然后释放它所占有的资源。因此

```
Finish[2] = true;
Work = Work + Allocation[2] = (2,1,2) + (4,1,1) = (6,2,3)
```

② 此时，Need[1]<Work，故 P_1 可执行完成。

```
Finish[1] = true;
Work = Work + Allocation[1] = ( 6,2,3) + (1,0,0) = (7,2,3)
```

③ 此时，Need[3]<Work，故 P_3 可执行完成。

```
Finish[3] = true;
Work = Work + Allocation[3] = ( 7,2,3) + (2,1,1) = (9,3,4)
```

④ 此时，Need[4]<Work，故 P_4 可执行完成。

```
Finish[4] = true;
Work = Work + Allocation[4] = ( 9,3,4) + (0,0,2) = (9,3,6)
```

系统至少可以找到一个安全的执行序列，如(P_2，P_1，P_3，P_4)，使各进程正常运行终结。

(3) 系统不能将资源分配给进程 P_1，因为虽然可利用资源还可以满足进程 P_1 现在的需求，但是一旦分配给进程 P_1 后，就找不到一个安全执行的序列保证各进程能够正常运行终结，所以进程 P_1 应该进入阻塞状态。

(4) 系统满足进程 P_1 和 P_2 的请求后，没有立即进入死锁状态，因为这时所有进程没有提出新的资源申请，全部进程均没有因资源未得到满足而进入阻塞状态。

例 3-9 某系统有同类资源 m 个，n 个并发进程可共享该类临界资源。求每个进程最多可申请多少个该类临界资源，保证系统一定不发生死锁。

分析：要使系统不发生死锁，则每个进程能获得的资源数最大值应是让每个进程得到最大需求数减 1，另外还有一个空闲资源。这样，就有一个进程可以申请到所需资源，执行完成后，再释放资源，以此类推，最终整个系统中所有的进程都能执行完成。

解：设每个进程最多申请该类资源的最大量为 x。

每个进程最多申请 x 个资源，则 n 个进程最多同时申请的该类临界资源数为 $n\times x$。

为保证系统不发生死锁，应满足下列不等式：

$$n(x-1)+1\leqslant m \qquad (*)$$

这是因为进程最多申请 x 个资源，最坏的情况是每个进程都已得到了($x-1$)个资源，现均申请最后一个资源。只要系统至少还有一个资源就可使其中一个或几个进程得到所需

的全部资源，在它们执行结束后归还的资源可供其他进程使用，因而不可能发生死锁。

解不等式(*)，可得

$$x \leqslant 1+[(m-1)/n]$$

即 x 的最大值为 $1+[(m-1)/n]$。因而，当每个进程申请资源的最大数值为 $1+[(m-1)/n]$ 时，系统肯定不会发生死锁。

3.11　死锁的检测与解除

死锁的预防和避免都是对资源的分配加以限制，操作系统解决死锁问题的另一条途径是死锁的检测方法。死锁检测方法与死锁预防和避免策略不同，这种方法对资源的分配不加限制，只要有剩余的资源，就可把资源分配给申请的进程，允许系统有死锁发生，这样做的结果可能会造成死锁。关键是当死锁发生时系统能够尽快检测到，以便及时解除死锁，使系统恢复正常运行。因此，采用这种方法必须解决三个问题：一是何时检测死锁的发生；二是如何判断系统是否出现了死锁；三是当发现死锁发生时如何解除死锁。

3.11.1　死锁检测的时机

由于死锁检测算法允许系统发生死锁，因此最好的情况是一旦死锁产生，就能立即检测到，也就是说死锁发现得越早越好。

死锁一般是在资源分配时发生的，所以将死锁的检测时机定在有资源请求时比较合理，但是采用这种方法检测的次数过于频繁，若没有死锁，将占用 CPU 非常大的时间开销。

另一种方法是周期性地检测，即每隔一定时间检测一次，或者根据 CPU 的使用效率去检测，先为 CPU 的使用率设定一个最低的阈值，每当发现 CPU 使用率降到该阈值以下时就启动检测程序，以减少由于死锁造成 CPU 的无谓操作。

3.11.2　死锁的检测

1. 利用资源分配图

系统对资源的分配情况可以用有向图加以描述，该图由结对组成 $G=(V,E)$。其中，V 是顶点的集合，E 是有向边的集合，顶点集合可分为 $P=\{P_1,P_2,\cdots,P_n\}$，由系统中全部进程组成；$R=\{r_1,r_2,r_3\}$，由系统中的全部资源组成。

由边组成的集合 E 中，每一个元素都是一个有序结对 (p_i,r_j) 或 (r_j,p_i)。其中，p_i 是 P 中的一个进程 $(p_i\in P)$，r_j 是 R 中的资源类型 $(r_j\in R)$。如果 $(p_i,r_j)\in E$，则存在一条从进程 p_i 指向资源 r_j 的有向边，进程 p_i 申请一个 r_j 资源单位。当前 p_i 在等待资源。如果 $(p_i,r_j)\in E$，则有向边从资源 r_j 指向进程 p_i，表示有一个 r_j 资源分配给进程 p_i。边 (p_i,r_j) 称为申请边，而边 (r_j,p_i) 称为赋给边。在资源分配图中，用圆圈表示每个进程，用方框表示各种资源的类型，方框中圆点的数量表示该类资源的个数。当然，申请边只能指向方框，而赋给边必须指向方框中的一个圆点。

图 3-11 所示的资源分配图表示 P_1 申请临界资源 R，同时 R 已被进程 P_2 占有。临界资源 R 类中只有一个资源。

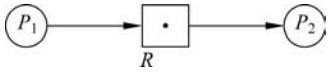

图 3-11　资源分配图

2. 资源分配图的简化

利用资源分配图进行死锁检测的目的是判定当前状态是否发生死锁。如果有进程没有被阻塞、进程没有请求边,那么一个资源分配图就能够进行化简。

通过消除所有指向进程的分配边,可以化简资源分配图。如果一个资源分配图不能通过任何一个进程化简,那么它就是不可被简化的;如果有一个化简序列导致图中没有任何种类的边,那么它就是可完全简化的。

3. 死锁定理

通过资源分配图可以很直观地得到系统中的进程使用资源的情况。显然,如果图中不出现封闭的环路,则系统中不会存在死锁;如果系统出现由各有向边组成的环路,则是否产生死锁,还需进一步分析:如果环路可以通过简化的方式取消,则系统一定不产生死锁;如果环路通过化简的方式仍不能取消,即不能再进行简化,则系统一定会产生死锁,这就是著名的死锁定理。

某系统状态S为死锁状态的充分条件是,当S状态的资源分配图是不可完全简化的,即如果资源分配图中不存在环路,则系统不存在死锁;如果资源分配图中存在环路,并且不可再简化,则系统产生死锁。

例如,在图3-10中,存在进程与资源的环路 $P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1$,且不可再被简化,因此系统死锁。

例 3-10 如图3-12和图3-13所示的资源分配图,试分析两系统是否发生死锁。

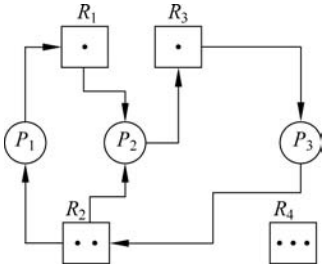

图3-12 例3-10资源分配图1

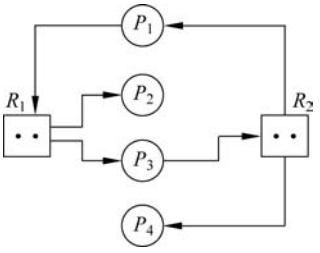

图3-13 例3-10资源分配图2

解:图3-12中存在两个环路 $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$ 和 $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$。通过分析,不能再对该资源分配图进行简化,所以,该系统处于死锁状态,进程 P_1、P_2 和 P_3 都参与了死锁。在图3-13中,虽然也存在一个环路,但该资源分配图是可以简化的,因此该系统并不死锁。

4. 死锁检测中的算法

死锁检测中的数据结构和算法类似于银行家算法。算法中的Available、Allocation、Request、Work的意义与银行家算法的相同,L是初值为空的进程表。

(1) 置初值:

```
Work = Available;
```

将不需要也不占用资源的进程记入表 L 中。

(2) 从进程集合中找到一个满足下列条件的进程 i：

$$Request_i \leqslant Work,$$

且该进程不在表 L 中。

若能找到这样的进程，则将该进程放入表列 L 中，然后增加工作向量：

```
Work = Work + Allocation[i];
```

返回步骤(2)。

若不能找到这样的进程，则执行步骤(3)。

(3) 若不能把所有进程记入表列 L 中，则表明系统状态 S 将发生死锁。

当发现系统处于不安全状态时，就要执行死锁的恢复策略。

3.11.3　死锁的解除

一旦在死锁检测时发现了死锁，就要解除死锁，使系统从死锁状态中恢复过来。一般采用两种方式来解除死锁：一种是终止一个或几个进程的执行以破坏循环等待；另一种是抢占参与死锁的进程的资源。

1. 终止进程

终止参与死锁的进程执行，系统可收回被终止进程所占用的资源，并进行再分配，以达到解除死锁的目的。

(1) 终止涉及死锁的所有进程。这种方法能彻底破坏死锁的循环等待状态，但将付出很大的代价，因为有些进程可能已经运行了很长时间，由于被终止而使产生的部分结果也被删除了，在重新执行时还要再次进行计算及运行。

(2) 一次终止一个进程。这种方法是每次终止一个涉及死锁的进程执行，收回它所占的资源作为可分配的资源，然后再由死锁检测程序判断是否仍然存在死锁。若死锁依然存在，则再终止一个处于死锁中的进程，如此循环直到死锁解除。

死锁解除后，应在适当的时机让被终止的进程重新执行。当重启运行进程时应让进程从头开始执行，也有的系统在进程执行的过程中设置校验点，重新启动时让进程回退到发生死锁之前的那个校验点开始执行。设置校验点对执行时间长的进程来说是有必要的，但系统的开销较大。

2. 抢占资源

从参与死锁的一个或几个进程中抢夺资源，把释放的资源再分配给另一些参与死锁的进程，直到死锁解除。采用此方法应注意以下三点。

(1) 抢占哪些进程的资源。希望能以最小的代价结束死锁，因此必须关注参与死锁的进程所占有的资源数，以及它们已经执行的时间等因素。

(2) 被抢夺进程的恢复。如果一个进程被抢夺了资源就无法继续执行，因而应该让它返回到某个安全状态并记录有关的信息，以便重新启动该进程执行。

(3) 进程的“饿死”。如果经常从同一个进程中抢占资源,则该进程总是处于资源不足的状态而不能完成,该进程就会被“饿死”。因此,一般情况下总是从执行时间短的进程中抢夺资源,以免该现象发生。

此外,还有进程回退策略,即让参与死锁的进程回退到没有发生死锁前的某一点处,并由此点处继续执行,以求再次执行时不再发生死锁。虽然这是个较理想的办法,但操作起来系统开销极大,要有堆栈这样的机构记录进程的每一步变化,以便以后的回退,有时无法做到。

本章小结

本章主要介绍操作系统中的三级调度与死锁。

处理机的三级调度:高级调度是作业调度,中级调度是内外存对换,低级调度是进程调度。用户交给计算机的一个任务称为作业,作业有四个基本状态:提交状态、后备状态、运行状态和完成状态。从后备态队列中选择一个作业执行,称为作业调度。进程调度是指按照一定的算法从就绪态队列中选择一个进程运行。在本章中分析了作业调度及进程调度的过程。

不同的操作系统类型追求的目标是有差异的。总的来讲调度算法追求的目标是系统的高效率、CPU 以及资源的利用率。不同的操作系统的目标因素有:系统的效率、吞吐量及各类资源的平衡利用、作业的周转时间、作业的响应时间、作业的截止时间、优先权准则等。

作业调度算法和进程调度算法有先来先服务调度算法、短作业(短进程)优先调度算法、高响应比优先调度算法、优先级调度算法和时间片轮转调度算法。其中,优先级调度算法包括非抢占式优先级调度算法和抢占式优先级调度算法。

死锁是指多个进程循环等待其他进程占有的资源,因而无限期地僵持下去的局面。死锁产生的原因有各进程间竞争有限的资源、进程推进顺序不当。产生死锁的必要条件有互斥条件、占有且申请条件、不可抢占条件、环路等待条件。解决死锁的基本方法有死锁的预防、死锁的避免、死锁的检测、死锁的解除。预防死锁是利用死锁的四个必要条件,只要保证一个条件不满足,死锁就不会发生;死锁的避免通常使用银行家算法,在并发执行的进程中寻找一个安全序列,如果能找到,假设系统按照安全序列的顺序分配资源,系统不会死锁。如果在进程的推进过程中,对资源的分配不加以限制,就有可能造成死锁,这时需采用死锁的检测方法,对系统是否陷入死锁加以检测。死锁的检测可以利用死锁定理,即死锁的充要条件是当且仅当资源分配图是不可完全被简化的;也可利用死锁检测算法来实现。如果发生死锁,则采用死锁的解除策略来解除死锁。

习题 3

3-1 处理机调度的主要目的是什么?

3-2 高级调度与低级调度的功能是什么?

3-3 处理机调度一般可分为哪三级?其中哪一级调度必不可少?为什么?

3-4　作业在其存在的过程中分为哪四种状态？

3-5　作业提交后是否立刻加载内存？为什么？

3-6　在批处理系统、分时系统和实时系统中，各采用哪几种进程或作业调度算法？

3-7　什么是实时调度？与非实时调度相比，有何区别？

3-8　在批处理系统、分时系统和实时系统中，各采用哪几种进程（作业）调度算法？

3-9　在操作系统中，引起进程调度的主要因素有哪些？

3-10　假设有四道作业，它们的提交时刻及需要执行的时间如表 3-14 所示。

表 3-14　各作业提交时刻及需要执行的时间

作业号	提交时刻	执行时间
1	10:00	2h
2	10:20	1h
3	10:40	50min
4	10:50	30min

试计算在单道程序环境下，采用先来先服务调度算法和短作业优先调度算法时的平均周转时间和平均带权周转时间，并指出它们的调度顺序。

3-11　在单 CPU 条件下有下列要执行的作业序列，如表 3-15 所示。作业到来的时间是按作业编号顺序进行的（后面作业依次比前一个作业迟到 1s）。

（1）用一个执行时间图描述在采用非抢占式优先级算法时执行这些作业的情况。

（2）对于上述算法，各个作业的周转时间是多少？平均周转时间是多少？

表 3-15　第 3-11 题表

作　业	运行时间/s	优 先 级
1	10	2
2	4	3
3	3	5

3-12　现有四个作业，假设它们按顺序依次到达，假设作业提交时刻为 0，但到达的前后时间忽略不计。要求运行时间分别为 2s、60s、2s、2s，如表 3-16 所示。系统按照高响应比优先调度算法进行作业调度，要求计算各作业的执行顺序、开始运行时间、运行结束时间、周转时间和带权周转时间。

表 3-16　第 3-12 题表

作业名	到达次序	要求运行时间/s
A	1	2
B	2	10
C	3	3
D	4	2

3-13　什么叫死锁？死锁产生的原因和必要条件有哪些？

3-14　在解决死锁问题的几个方法中，哪种方法最易实现？哪种方法资源利用率最高？

3-15　设时间片为一个时间单位。现有四个进程，每个进程比上一个进程迟到一个时

间单位,各进程要求运行时间如表 3-17 所示。系统对进程调度采用时间片轮转调度算法。试计算各进程的完成时间和周转时间。

表 3-17 习题 3-15 表

进程	进入时间	要求运行时间/s
A	0	12
B	1	5
C	2	3
D	3	1

3-16 什么是安全状态?为什么安全状态可以向不安全状态转化?

3-17 假定某系统有五个进程 P_1、P_2、P_3、P_4、P_5 共享三类资源 A、B、C。资源类 A 共有 17 个资源,资源类 B 共有 5 个资源,资源类 C 共有 20 个资源。某一时刻,各进程对资源的需求和占用情况如表 3-18 所示。

表 3-18 习题 3-17 表

请求进程	最大资源需求量			已分配资源数量		
	A	**B**	**C**	**A**	**B**	**C**
P_1	5	5	9	2	1	2
P_2	5	3	6	4	0	2
P_3	4	0	11	4	0	5
P_4	4	2	5	2	0	4
P_5	4	2	4	3	1	4

(1) 判定系统的安全性。若安全,给出安全序列;若不安全,说明原因。

(2) 在该时刻若进程 P_2 请求资源(0,3,4),是否能分配资源?为什么?

3-18 试说明 Linux 系统的进程调度算法所采用的策略。