第3章 调度与死锁

CHAPTER

3.1 概 述

本章讨论 CPU 调度(schedule)策略和死锁问题。讨论 FIFO、SJF 和 RR 常用的调度策略。讨论预防死锁、避免死锁、检测死锁等问题。

在多道程序系统中,一个作业(job)从提交到执行,通常都要经历多次调度,而系统的运行性能在很大程度上取决于调度。CPU 调度使得多个进程有条不紊地共享一个 CPU。由于 CPU 的运行速度很快,调度的速度也很快,使每个用户进程在短时间内都有机会运行,就好像每个进程都有一个专用 CPU。或者可以说,CPU 调度为每个用户进程都提供了一台虚拟处理机。一个好的调度策略对于加快作业总的周转时间、提高单位时间内的作业吞吐量、实现系统总的设计目标,是十分重要的。本章讨论一般的调度策略。

调度问题与资源(resource)分配有关,比如在其他条件相当的情况下,应该优先调度占有资源多的进程,以便在这些进程运行完后,能够收回更多的资源。不合理的调度则有可能加剧进程对资源的争夺,导致资源利用率低,甚至出现死锁局面。

通常引入作业平均周转时间 T 和加权平均周转时间 W 作为衡量作业调度算法的测度。

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

其中,$T_i = F_i - A_i$,F_i=作业结束时间,A_i=作业到达时间,n 为作业数。

考虑到作业的结束时间与作业长度有关,故作业周转时间不能完全反映调度性能,再引入 $W_i = T_i / R_i$。

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

其中,R 为作业实际运行时间。

另一个评价算法的尺度是作业或进程的平均等待时间,即各个作业或进程进入可以调度的状态(作业成为收容状态,进程成为就绪态)到开始选

中的时间。

3.2 分级调度

可以打个比方说明调度之所以要分级,开运动会时,有几十人报名参加100m竞赛,不会一次决出冠军。组织者会设置报名、检录、竞赛几个阶段;竞赛阶段又分初赛、复赛、决赛,最终才能决定谁是冠军。在多道程序环境下,操作系统中面对众多进程,为了提高调度效率,也实行分级调度。

3.2.1 高级调度

高级调度又称作业调度或长程调度,用于决定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到就绪队列上,准备执行。

每次执行高级调度时,都需决定以下两点。

(1) 接纳多少个作业。这取决于多道程序调度,即允许有多少作业同时在内存中并发运行。

(2) 接纳哪些作业。这取决于所采用的调度算法。最简单的是先来先服务调度算法,它是将最早进入外存的作业调入内存。较常用的是SJF算法,即将外存上最短的作业调入内存。

3.2.2 中级调度

中级调度又称中程调度。它负责进程在内存和辅存对换区之间的对换。由于某种原因,一些进程处于阻塞(blocked)状态而暂时不能运行,为了缓和内存使用紧张的矛盾,中级调度将不能运行的进程暂时移到辅存对换区。在对换区的进程,若其等待的事件已发生,则它们要由阻塞状态变为就绪。为了使这些进程能继续运行,中级调度再次把它们调入内存。一个进程在其运行期间有可能被多次调进调出。

自UNIX采用进程对换(swapping)技术以来,现代许多操作系统都引入了这种机制。进程可以整个地在内存和辅存之间进出,增加内存中参与多道运行调度的进程数,或者说增加系统的多道程序设计能力,加快作业周转,提高系统资源利用率。

3.2.3 低级调度

低级调度又称进程调度或短程调度。它决定驻留内存就绪队列中的哪个进程获得处理机,然后由分派程序执行把处理机分配给该进程的“上下文切换”操作。进程调度是最基本的一种调度。

什么时候激活进程调度?从图3.1的进程的调度队列模型,可以发现在4种可能的情况下将激活进程调度。

(1) 在CPU上运行的那个进程正好运行完成。进程调度程序应该立即工作以选择下一个运行对象。

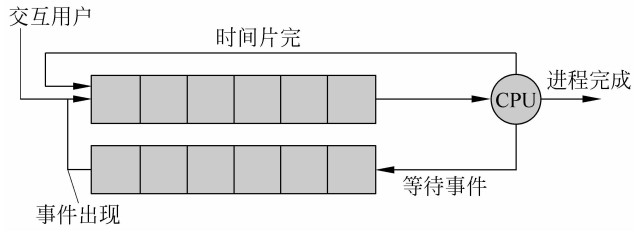

图 3.1　进程的调度队列模型

(2) 运行进程被阻塞,比如需要输入输出,需要等待某种消息或某种事件,运行进程将主动让出 CPU,此时应该施行调度。

(3) 运行进程因时间片到期而被剥夺运行权,进程将转换到就绪态,进程调度将被激活。

(4) 当有交互进程就绪到达时,或者有进程解除等待原因,比如输入输出完成,等待的事件已发生或信息已到达,由等待态转为就绪态时,实施抢占调度的系统,也会进行重新调度,以保证高优先级进程尽可能快得到运行机会。

每个支持进程的操作系统都有进程调度,但不一定有中级调度和高级调度,或者只有二者之一。一个配有三级调度的系统,其高级调度、中级调度与低级调度的关系如图 3.2 所示。

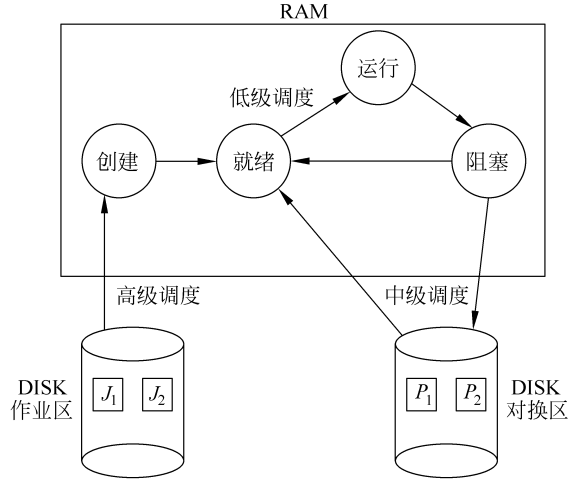

图 3.2　高级调度、中级调度与低级调度的关系

三级调度的对象不同,任务也不同。高级调度以作业为单位,调度频率相对较低。低级调度运行频率很高,才能保证各个进程在短时间内得到运行机会。但是它们的共同目的是让用户的程序尽快得到运行,尽可能地提高资源利用率。

3.3 常用调度算法

本节讨论作业调度与进程调度常用调度算法,指出每种算法的适用性。对于调度对象和调度级别不做特别声明的,可以认为对作业调度和进程调度都有一定的适应性。关于进程对换策略涉及内存分配,将在第4章讨论。

有下述两类调度算法。

(1) 非抢占方式。这种方式,一旦把处理机分配给进程后,便让该进程一直执行,直到该进程完成或发生某事件而被阻塞时,才能把处理机分配给其他进程。不允许任何进程抢占已经分配的处理机。

(2) 抢占方式。这种方式,允许调度程序根据某种原则,去停止某个正在执行的进程,将已分配的处理机重新分配给另一进程。抢占原则有以下几种。

① 时间片原则。

② 优先权原则。

③ 短作业优先原则。

3.3.1 FIFO 调度算法

FIFO(first in first out)算法即先进先出算法,是最简单的调度算法。其基本原则是按照作业到达系统或进程进入就绪队列的先后次序来选择。一个进程一旦占有了处理机,它就一直运行下去,直到该进程完成其工作或因等待某事件而不能继续运行时才释放处理机。FIFO算法实行不可抢占策略。

例 3.1 有表3.1所示的作业序列。

表 3.1 一个作业序列实例

作 业 号	到 达 时 间	运行时间/h	作 业 号	到 达 时 间	运行时间/h
1	8.00	2.00	3	9.00	0.10
2	8.50	0.50	4	9.50	0.20

注:为了便于计算,约定时间数据为十进制数,单位为小时(表3.2～表3.4同)。

按照FIFO算法,可以像表3.2那样计算。

表 3.2 FIFO 调度计算

调度顺序	作 业 号	到 达 时 间	开 始 时 间	结 束 时 间	周转时间/h	加权周转率
1	1	8.00	8.00	10.00	2.00	1
2	2	8.50	10.00	10.50	2.00	4
3	3	9.00	10.50	10.60	1.60	16
4	4	9.50	10.60	10.80	1.30	6.5

因为 FIFO 算法是按作业到达时间的先后来决定运行的先后，所以运行顺序为 1、2、3、4。具体计算方法如下。

(1) 周转时间(T_i)。

$$T_i = \text{结束时间} - \text{到达时间}$$

(2) 加权周转时间(W_i)。

$$W_i = \text{周转时间} / \text{运行时间}$$

(3) 平均周转时间(T)。

$$T = (T_1 + T_2 + T_3 + T_4)/4 = 1.73\text{h}$$

(4) 加权平均周转率(W)。

$$W = (W_1 + W_2 + W_3 + W_4)/4 = 6.88$$

评论：这种算法按先来后到原则调度，比较公平，但是不利于短作业。

3.3.2　SJF 调度算法

SJF(shortest job first)算法即短作业优先调度算法，是指对短作业或短进程优先调度的算法。SJF 算法照顾短作业，使短作业能比长作业优先执行。该调度算法是从作业的后备队列中挑选那些所需运行时间(估计时间)最短的作业进入主存运行。这种算法实行非抢占策略，一旦选中某个短作业后，就保证该作业尽可能快地完成运行并退出系统，运行中不允许被抢占。

继续用表 3.1 的作业序列例子，按照 SJF 算法，调度顺序应为 J_1(因为 8 点钟的时候，仅有这一个作业)、J_3、J_4、J_2。计算如表 3.3 所示。

表 3.3　JSF 调度计算

调度顺序	作　业　号	到达时间	开始时间	结束时间	周转时间/h	加权周转率
1	1	8.00	8.00	10.00	2.0	1
2	3	9.00	10.00	10.10	1.1	11
3	4	9.50	10.10	10.30	0.8	4.0
4	2	8.50	10.30	10.80	2.3	4.6

平均周转时间：

$$T = (T_1 + T_2 + T_3 + T_4)/4 = 1.55\text{h}$$

加权平均周转率：

$$W = (1 + 4.6 + 11 + 4)/4 = 5.15$$

以下可以证明，采用 SJF 算法，系统有最短的平均周转时间。

问题描述：给定一组作业 J_1、J_2、…、J_n，它们的运行时间分别为 T_1、T_2、…、T_n。假定这些作业同时到达，并且在一台处理机上以单道方式运行。试证明：若按 SJF 调度顺序运行这些作业，则平均周转时间 T 最小。

证明：不失一般性，假定调度顺序为 J_1、J_2、…、J_n，到达时间为 0，则作业 J_i 的平均周转时间为：

$$T'_i = T_1 + T_2 + T_3 + \cdots + T_i$$

所有作业的平均周转时间：

$$T = \frac{1}{n}\sum_{i=1}^{n} T'_i$$

显然，当 $T_1 \leqslant T_2 \leqslant T_3 \leqslant \cdots \leqslant T_n$ 时，每一个 T'_i 达到最小值($i=1$、2、…、n)，这是因为 T'_i 是 T_1、T_2、…、T_n 中前面 i 个数之和，现在是其中的最小 i 个数之和。因此 T'_i 最小，既然每个 T_i 最小，因此 T 最小。

3.3.3 HRN 调度算法

HRN (highest response ratio-next)算法即最高响应比优先调度算法。这是一种非抢占的作业调度策略。这种策略是 FIFO 算法与 SJF 算法的折中。按照此策略，每个作业在参与调度的时候都有一个响应比，其数值是动态变化的。它既是该作业要求服务时间的函数，也是该作业为得到服务所花的等待时间的函数。它能保证任何作业都不会被无限延迟。

作业的动态响应比计算公式如下：

$$\text{响应比 } R_p = (\text{等待时间} + \text{要求服务时间}) / \text{要求服务时间}$$
$$= 1 + \text{等待时间} / \text{要求服务时间}$$

由于等待时间加上要求服务时间，就是系统对该作业的响应时间，故该响应比又可表示为：

$$R_p = \text{响应时间} / \text{要求服务时间}$$

继续用表 3.1 的作业序列例子，按照表 3.4 计算。

表 3.4 HRN 调度计算

调度顺序	作业号	到达时间	开始时间	结束时间	周转时间/h	加权周转率
1	1	8.00	8.00	10.00	2.00	1
2	3	9.00	10.00	10.10	1.10	11
3	2	8.50	10.10	10.60	2.10	4.2
4	4	9.50	10.60	10.80	1.30	6.5

在 8.00 这一时刻只有作业 1 到达所以先运行。因为此调度算法是非抢占式的，所以一直到作业 1 运行完，在时刻 10.00 才决定下一个作业，而此时作业 2、3、4 都已到达，则分别对它们进行响应比计算。

$$R_{P_2} = 4, \quad R_{P_3} = 11, \quad R_{P_4} = 3.5$$

因此，此时调度作业 3，作业 3 完成后再按此方法求当时各作业的 RP，决定调度哪个作业，选择作业 2。最后运行作业 4。

平均周转时间：

$$T = (T_1 + T_2 + T_3 + T_4)/4 = 1.625\text{h}$$

加权平均周转率：

$$W = (W_1 + W_2 + W_3 + W_4)/4 = 5.68$$

3.3.4　RR 调度算法

RR（round robin，时间片轮转）调度算法是一种剥夺式的调度算法，主要用于进程调度。系统将所有就绪进程按先来先服务的原则排成一个队列，每次调度时把 CPU 分配给队首进程，并让它执行一个时间片。当执行的时间片用完时，由一个计时器发出时钟中断，调度程序便据此信号来停止该进程的执行，然后该处理机分配给就绪队列中新的队首进程，同时也保证它执行一个时间片。这就可保证就绪队列中的所有进程在一定的时间内均能获得一个时间片的处理机执行时间。但是在进程执行期间，虽然时间片未到期，由于自身的原因，例如，因为要启动输入输出，或者要等待某种信号，或者由于程序自身出现异常而无法继续执行时，这种算法也立即启动抢占 CPU 并切换给其他进程。简单轮转调度模型如图 3.3 所示。

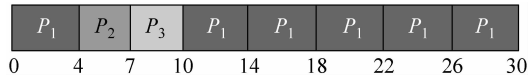

图 3.3　简单轮转调度模型

例 3.2　进程为 P_1、P_2、P_3，对应的 CPU 周期为 24、3、3（时间单位）。若取时间片＝4，则轮转法执行情况如图 3.3 所示。

P_2 最先完成，其周转率为 7，其次完成 P_3 进程的周转率为 10，则 P_1 的周转率为 30。平均周转率 W 的计算公式如下：

$$W = (W_1 + W_2 + W_3)/3 = (7 + 10 + 30)/3 = 16$$

如果用平均等待时间来衡量，则

$$\text{平均等待时间} = (0 + 4 + 7)/3 = 3.67(\text{时间单位})$$

3.3.5　优先级调度算法

将给每个进程（或作业）规定一个优先级，比如给实时进程以高优先级，如果优先级在进程运行中可以依据某种策略改变，则称为动态优先级。调度时选择优先级最高的进程（或作业）。

例 3.3　有 5 个进程 P_1、P_2、P_3、P_4、P_5，到达时间皆为 0，其预计运行时间和优先级如表 3.5 所示。

表 3.5　5 个进程的优先级

进　　程	CPU 时间/ms	优　先　级
P_1	5	2
P_2	5	0
P_3	5	3
P_4	4	1
P_5	3	2

按照优先级调度,可以得到如图3.4所示的调度结果。

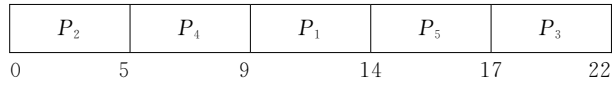

图3.4 优先级调度

5个进程的平均等待时间 = (9 + 0 + 17 + 5 + 14)/5ms = 9ms

优先级算法一般实行可抢占策略。它能保证紧迫作业及时被调度,又能使每个进程在短时间内有机会运行。在CPU速度很快,RAM容量很大,多道程序能力很强的现代计算机系统中,比较适合采用这种调度方法,UNIX和Windows均采用了这种调度方法。

3.3.6 多级反馈队列调度

1. 队列组织与调度策略

一个好的调度机制应该是尽可能快地决定一个作业的性质并据其特性调度该作业。调度策略应能优待短作业,以保持系统高的调度周转率,又能优待受I/O制约的作业,以便更好地利用输入输出设备。

多级反馈队列提供实现上述目的的机制,它组织如图3.5所示的多级反馈队列网络结构。一个新进程首先进入队列网络的第1级队列末尾,根据先进先出原则在队列中移动直到它获得处理机。如果该作业完成或由于等待I/O,或等待其他事件的完成而放弃处理机,该作业将离开队列网络。若进程在它自愿放弃处理机前耗尽时间片,则该进程被放在下一级(第2级)队列的末尾。若第一个队列为空,则在它达到下级队列的队首时该进程应获得服务。进程使用完每一级队列提供的时间片之后,被移到下一较低级队列的末尾。通常在一个底层队列中,进程按轮转调度法调度,直到进程运行结束。

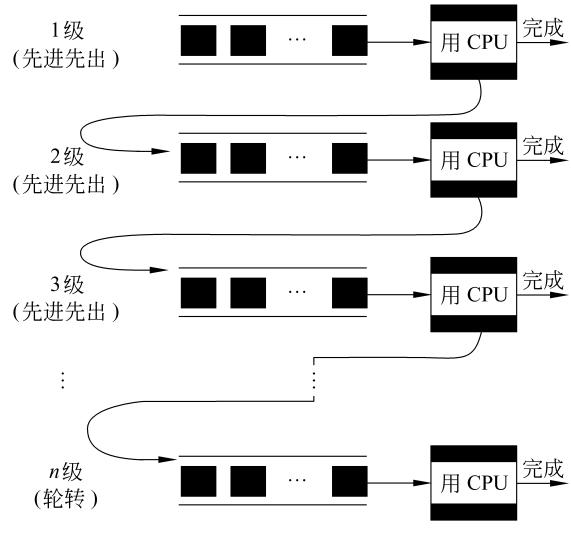

图3.5 多级反馈队列调度

调度优先级逐级地降低，调度时，总是首先挑选优先级最高的处于第 1 级的进程。在某个给定队列中的进程，只有当所有较高级队列均为空时才可运行。一个运行着的进程可被新到达的更高级队列的进程所剥夺。

与优先级逐级地降低相反，给予进程的时间片则逐级地变大。因此，一个进程在队列网络中时间愈长，通常将处于较低级别队列，在它获得处理机时允许的时间片也越长。

2. 优缺点分析

(1) 短进程和受 I/O 制约的进程将受到调度优待。由于受 I/O 制约的进程在进入网络时有很高的优先级，因此可能很快地获得处理机。在选择第一级队列的时间片时，可使这个时间片大小适当，使大多数受 I/O 制约的作业在耗尽它的时间片之前，就可能提出 I/O 请求，使得 I/O 设备忙碌起来。在该进程请求 I/O 后，它就离开网络。可见，这样的进程确实受到了优待。

(2) 受 CPU 制约的进程将会有最大运行效率。对于需要大量处理机时间的受 CPU 制约的作业，当它进入网络的最高队列时，由于队列的优先级高，可以很快获得第一次处理机服务，耗尽它的时间片，然后该进程被移到下一个较低队列。现在，该进程一旦获得处理机，它获得的时间片会比在最高队列时获得的时间片大。用户再次用完它的整个时间片，然后放到下一个更低队列的末尾。该进程如此不断地逐级移到较低队列去，每下降一级，等待的时间就越长，而进程在每次获得处理机时都用完它的全部时间片(除非被另一刚到达的进程剥夺)。最后，受 CPU 制约的进程移到最低级队列，在这个队列中按轮转调度法调度，直到结束。由于较高队列中的进程优先级较高，所以长时间进程被调度的机会较少。如果定义进程的运行效率 v 的计算公式如下：

$$v = \text{总运行时间} / \text{调度次数}$$

显然，受 CPU 制约的进程将会有最大运行效率。

(3) 系统可以自动判断进程的类别，并对各个进程做出恰当的处理，将它们包容在一起。多级反馈队列根据进程行为的不同，将进程分类。短进程将很快得到调度和周转，而需要长时间运行的进程，将逐渐地沉入底层。也有较好的机制处理进程进入队列和离开队列。在分时系统中，每次当一个进程离开队列网络时，用该进程所在的最低级队列的标识来标记该进程，在该进程重新进入队列网络时，直接送到该进程原先离开的那个队列。此时，调度程序采用直接推断法，即一个进程最近的行为是该进程将来行为的指示。所以，一个返回到队列网络的受 CPU 制约的进程并不放到较高级的队列，从而避免了与高优先级短进程或受 I/O 制约的进程争夺 CPU 的冲突。

(4) 能够自动适应进程性质的变化。进程性质可能发生变化，例如，从受 CPU 制约变为受 I/O 制约。为了解决这个问题，可以标记进程上次在网络中停留的时间，当进程重新进入网络时，即可根据上述标记将进程放入正确的队列。一个进程可能正处于从受 CPU 制约到受 I/O 制约的变化过程中，当系统决定进程的性质正在发生变化时，这个进程起初将获得一些停滞不变的对待。而调度机制对这一改变会做出快速响应。使系统对进程行为的改变具有良好反应的另一种方法是，允许进程在它每次时间片还未耗尽前自愿放弃该处理机时，将该进程在反馈队列网络中上移一个级别。

多级反馈队列机制常用的一种变型是当一个进程在被移到下一较低队列前，要循环

地通过每个队列几次,通常,通过每个队列的循环次数将随着进程移到低级队列而增加。

多级反馈队列是自适应机制的一个极好例子,它可以对进程行为的改变自动作出响应,增强了系统对变化作出反应的灵敏度。这是这种调度策略最大的特点。

一般说来,软件越完善,开销就越大,自适应机制的开销一般比非自适应机制大,但开销的增加与得到的好处相比仍然是合算的。

综上所述,多级反馈队列调度被公认为是比较好的调度算法,被一些系统广泛采用。

3.4 死锁问题

死锁是操作系统中一个与资源分配和调度有关的问题。本节介绍死锁的基本概念,讨论死锁的预防、避免、检测、解除的一般方法。

死锁是指多个进程因竞争资源而造成的一种僵局,若无外力作用,这些进程都将永远不能再向前推进。如图3.6所示,如果甲、乙进程的共同进展路径进入危险区时,一定会进入禁区而发生死锁。

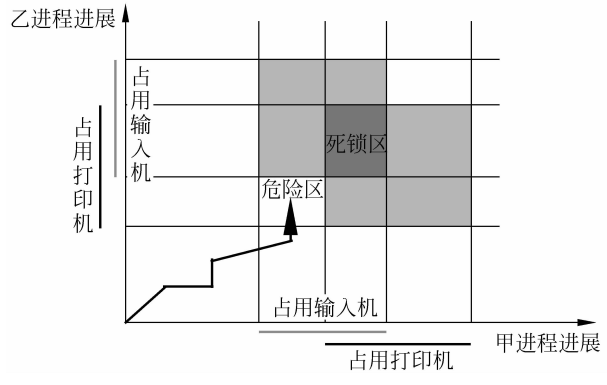

图3.6 死锁概念

3.4.1 产生死锁的必要条件

具备下列4个条件之一时,就可能会产生死锁。

(1) 互斥条件。某些资源有排他地使用性质,不能保证资源被进程任意共享。

(2) 请求并保持条件。进程已经拥有部分资源,又还要继续申请资源。

(3) 不剥夺条件。进程已经拥有的资源,不能被系统强行收回以做它用。

(4) 环路等待条件。两个进程P_1、P_2互相等待被对方已经占用的资源R_1、R_2,如图3.7所示。环路等待条件也可能存在于多个进程之间。

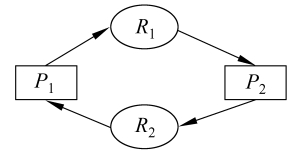

图3.7 两个进程环路等待

3.4.2　预防死锁

破坏产生死锁的 4 个必要条件中的一个或多个，使系统不可能进入死锁状态，这就是死锁的预防。因为互斥使用是某些资源的特定性能，比如打印机不能同时被多个进程共用。故预防死锁通常是摒弃“请求和保持条件”、摒弃“不剥夺条件”以及摒弃“环路等待条件”。

1. 摒弃“请求和保持条件”

采用这种方法预防死锁时，系统要求所有进程要一次性地申请在进程整个推进过程中所需要的全部资源。于是，要么不给进程分配资源，让进程等待，这样不会发生死锁；要么满足进程的全部资源要求，该进程在运行期间，将摒弃请求条件，不会再提出资源要求，因而也不会发生死锁。

这是一种静态的分配方法，优点是简单且安全。但也有明显的缺点，如下。

(1) 进程难以一次性地提出全部资源要求。

(2) 只要有一种资源不能满足该进程的分配要求，其他资源也全部不分配给该进程而让进程等待，可能延迟进程的推进。

(3) 某些资源可能进程仅仅在最后阶段才使用，或者只使用一个短暂时间，也必须一开始就分配给它独占，造成资源严重浪费。

2. 摒弃“不剥夺条件”

采用这种方法时，不要求进程一次性地提出全部资源要求，进程可以在只满足当前资源要求的情况下运行，在需要新的资源时才提出请求。但是一个已经保持了某些资源的进程，当它再提出新的要求而不能立即满足时，必须释放它占有的所有资源，待以后需要时再重新提出申请。

这是一种动态的分配方法，可以减少资源被长时间独占且闲置，因而提高了资源利用率。但是进程放弃已经占用但尚未用完的资源可能要付出很大的代价。动态分配比较复杂，也增加了系统的开销。

3. 摒弃“环路等待条件”

系统将所有资源都编上唯一的序号，申请资源必须严格按资源递增的顺序提出，这样在所形成的资源分配图中，不可能再出现环路，因而摒弃“环路等待条件”。

这种方法也是动态分配方法，提高了资源利用率和系统吞吐量。但是，为系统中各种类型资源分配序号，难以照顾所有用户的编程习惯；按规定次序申请资源的方法，可能会限制用户自由的编程思路；为系统中各种类型资源所分配的序号，必须相对稳定，这就限制了添加新类型设备的方便性。

例 3.4　一个无死锁的充分条件是，若系统中有 n 个进程，m 个资源，资源是逐个申请的，每个进程至少申请一个资源，假定所有进程申请资源的总数 $\sum_{i=1}^{n} R_i < m+n$，则系统不可能有死锁。

设每个进程 P_i 申请的资源数为 $R_i(i=1,2,\cdots,n)$，则：

$$R_i \leqslant m$$

从考察死锁的角度看,如果有的进程不申请资源,或者只申请其所需的少部分资源,这不是最坏的情况。如果有的进程申请其全部资源,这也不是最坏的情况,因为要么满足它,用完可收回;要么不分配,令其等待。最坏的分配情况是每个进程均获得了 R_i-1 个资源,即每个进程最大限度地从系统获得了部分资源,但并非全部,尚不能运行完从而释放其所占的资源。根据题意,已分配资源总数 R 的计算公式如下:

$$R=\sum_{i=1}^{n}(R_i-1)=\sum_{i=1}^{n}R_i-n<m+n-n=m$$

说明在最坏的情况下,系统中都还至少会有一个资源供分配,所以不会有死锁。

3.4.3 死锁避免及银行家算法

从系统当前状态 S 出发,对于当前的资源申请,假定实施分配,系统是否能保持安全状态?银行家算法从假定实施分配开始,逐个检查各进程,在做了这种分配后哪个进程能完成其工作,然后就能够释放其全部资源,再进而检查哪个进程又能完成其工作,也释放其全部资源。重复以上步骤,如果能够找出进程的一种分配序列,使所有进程都能相继完成工作,则状态是安全的。对于进程的资源要求,假定实施的分配可以保证系统处于安全状态,便可以接受并进行真正的资源分配。如果不能保证系统处于安全状态,便拒绝分配,令申请资源的进程等待。

安全状态是指系统能按某种顺序,如 P_1、P_2、…、P_n,来为每个进程分配其所需资源,直至最大需求,使每个进程都可顺序完成。P_1、P_2、…、P_n 便称为安全分配系列。若系统不存在这样一个安全序列,则称系统处于不安全状态。只要系统处于安全状态,系统便可避免进入死锁状态。因此,避免死锁的实质在于:如何使系统不进入不安全状态。

例 3.5 安全状态的例子。

假定系统中有3个进程及 R 资源13个。目前的进程与资源占有状态如表3.6所示。

表 3.6 进程—资源安全占有状态表

进 程	资源需求总量	已 持 有 数	目前申请量	未分配资源数
P_1	10	5	5	3
P_2	5	3	2	
P_3	6	2	4	

分析发现,存在安全序列 P_2、P_1、P_3 或 P_2、P_3、P_1,使所有进程能够运行完。故目前系统处于安全状态,不会发生死锁。

例 3.6 不安全状态的例子。

在上述安全状态基础上,进程 P_3 申请两个资源,如果没有做安全状态检查满足了它的要求,将出现表3.7所示的不安全状态,因为系统已经无法保证任何进程的最大资源要求。

不安全状态不一定就会死锁,因为某些进程随后可能还会释放已经占有的资源,但是安全状态则肯定不会死锁。

表 3.7　进程—资源不安全占有状态表

进　　程	资源需求总量	已 持 有 数	目前申请量	未分配资源数
P_1	10	5	5	1
P_2	5	3	2	
P_3	6	4	2	

银行家算法是一种使系统永远处于安全状态的算法。它每遇到一次资源申请，都要先试分配，经过一系列计算，看是否能找出一个安全分配系列，使系统能够处于安全状态。找得到安全分配系列才进行分配，否则拒绝分配，让申请者等待。例如，在表 3.6 的基础上，P_3 提出两个资源要求时将予以拒绝，避免系统进入不安全状态。

再来讨论银行家算法的实现。假设系统中有 n 个进程和 m 种资源，银行家算法需要以下几种数据结构。

Available 阵列：一个长度为 m 的数组存放着目前尚未分配的各种资源的数目，如 Available[i]=3 表示目前 R_i 资源仍有 3 项尚未分配给任何进程。

Max 阵列：一个 $n\times m$ 的矩阵记录着每个进程对每种资源所需要的数目，如 Max[i,j]=5 表示进程 P_i 需要 5 个 R_j 资源以完成工作。

Allocation 阵列：一个 $n\times m$ 的矩阵记录着每个进程所持有的各种资源的数量，如 Allocation [i,j]=2 表示目前进程 P_i 持有 2 项 R_j 资源。

Need 阵列：一个 $n\times m$ 的矩阵记录着目前每个进程需要各种资源的数量，如 Need [i,j]=2 表示目前进程 P_i 需要 2 项 R_j 资源以完成工作。

银行家算法需要一个安全算法来测试系统是否处于安全状态，以及一个资源要求算法来决定是否允许资源要求。现将两个算法介绍如下。

(1) 安全算法(safety algorithm)。

① 声明两个长度为 m 与 n 的数组 Work 与 Finish，并将 Work 初始化为 Available，Finish 数组中所有元素初始为 FALSE。

② 寻找 i 使得 Finish[i]=FALSE 而且 Need[j]<=Work[i]，如果找不到这样的 i，执行步骤(4)。

③ Work[i]=Work[i]+Allocation[i]；Finish[i]=TRUE；执行步骤(2)。

④ 如果 Finish 数组中所有元素都为 TRUE，则系统目前处于安全状态中；否则处于不安全状态。

(2) 资源要求算法(resource request algorithm)。

① 声明 $n\times m$ 的 Request 数组存放进程所要求各项资源的数量，Request[i,j]=3 表示进程 P_i 要求 3 项 R_j 资源。

② 如果 Request[i]<=Need[i]，执行步骤(3)；否则因为进程要求过多的资源而发生错误。

③ 如果 Request[i]<=Available[i]，则执行步骤(4)；否则因为目前系统中尚未分配的资源不足，进程 P_i 必须等待。

④ 进行以下的运算：

$$\text{Available}[i]=\text{Available}[i]-\text{Request}[i]$$
$$\text{Allocation}[i]=\text{Allocation}[i]+\text{Request}[i]$$
$$\text{Need}[i]=\text{Need}[i]-\text{Request}[i]$$

使用安全算法检验运算后的结果，如果处于安全状态则允许分配该资源给 P_i；否则 P_i 必须等待，并且回存步骤④执行前的结果，使系统保持原来的进程—资源状态。

下面举例做更进一步的说明。

例 3.7 假设系统中有5个进程 P_1、…、P_5，3种资源 A、B、C，数量分别为13、10、9，进程—资源分配如表3.8所示。

表 3.8 进程—资源表

进　程	Max 需要资源数目			Allocation 持有资源数目			Available 系统未分配资源数目		
	A	***B***	***C***	***A***	***B***	***C***	***A***	***B***	***C***
P_1	8	0	2	5	0	0	3	1	2
P_2	5	2	1	3	1	0			
P_3	1	2	2	0	1	2			
P_4	7	6	4	2	5	2			
P_5	3	3	5	0	2	3			

目前系统正处于安全状态下，因为可以找出一组安全序列 P_5、P_3、P_2、P_1、P_4。如果此时进程 P_5 要求资源(3,0,1)，通过银行家算法运算后，发现在分配这些资源给 P_5 之后仍然可以使系统处于安全状态，存在一组安全序列 P_5、P_3、P_2、P_1、P_4，所以系统可以允许分配这些资源给 P_5。不过，如果此时进程 P_4 要求资源(2,1,2)，通过银行家算法运算之后，发现这样分配会使系统进入不安全状态，因此系统不能允许分配这些资源给 P_4。

银行家算法存在的问题是计算量很大，效率低。

3.4.4 死锁的检测

死锁检测算法主要是检查系统中是否存在循环等待条件。最常用的检测死锁的方法就是对进程资源图的化简。

进程资源图的化简是指一个进程的所有资源要求均能被满足的情况下，假若这个进程得到所需的所有资源，从而该进程的工作就能不断地取得进展，直到最后完成其全部运行任务，并释放出全部资源。那么，该资源分配图可被这个进程所化简。假如一个资源分配图可以被所有进程所化简，那么称该图是可化简的，因而系统未出现死锁。假如该图不能被其上所有进程所化简，则称该图是不可化简的，系统出现了死锁。

例 3.8 假定有进程 P_1、P_2、P_3，有资源 R_1、R_2 各3个。当前进程对资源的申请和占有关系如图3.8所示，P_1、P_2 各占有 R_1、R_2 一个，P_3 占有 R_2 又申请 R_1。图3.8称为进程—资源图。

图 3.8 可以按照 P_2、P_3、P_1 的顺序依次约简，如图 3.9～图 3.11 所示。

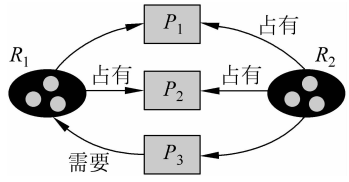

图 3.8　进程—资源图

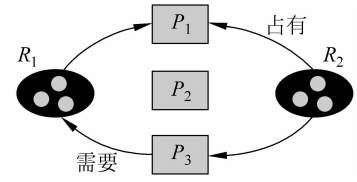

图 3.9　约简进程—资源图(约简 P_2)

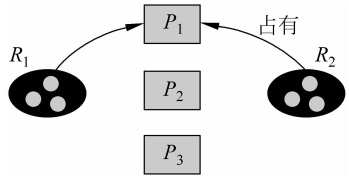

图 3.10　约简进程—资源图(再约简 P_3)

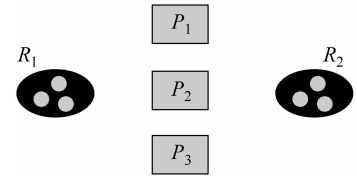

图 3.11　约简进程—资源图(再约简 P_1)

这是个很简单的例子，对于这个例子也可以按照 P_3、P_2、P_1 的顺序，或者按照 P_1、P_2、P_3 的顺序约简，结果是一样的。可以证明，对于任何情况，是否出现死锁的结论与约简顺序无关。

3.5　重点演示和交互练习：优先级调度算法

这里将第一章的多道程序设计技术和本章介绍的调度算法相结合，进行模拟调度的交互练习，以便切实掌握调度算法的运用，并体验调度算法对于加快作业周转的效率。假定系统中有一个 CPU，一台输入输出设备，支持最多 A、B、C 三道程序并发执行(可能当前只有其中一道或者两道)。系统采用优先但不可剥夺的调度策略，总是选择优先级最高的进程运行。不失一般性，假定 A、B、C 三者优先级 A 最高，B 其次，C 最低。系统允许任意给定作业结构和它们的组合，算法将立即计算出每个作业的周转时间和总的周转时间，并画出调度时序图。

(1) 优先级调度算法交互练习。

运行我们提供的“优先级调度算法”可执行程序，可以进行优先级调度交互练习。将出现如图 3.12 所示的交互界面。

单击“进程行为设置”主菜单，进入如图 3.13 所示的对话框。可以任意给定 1～3 个作业结构和它们的组合。

给定如图 3.14 所示的进程 A、B、C 的结构。

开始时，三个进程都争夺 CPU，按照优先级，应该先调度进程 A，然后是进程 B，最后是进程 C。在经过 10ms 后，进程 A 使用设备 I/O，CPU 能够与外部设备并行工作，调度程序将调度进程 B 在 CPU 上执行，这期间恰好与进程 A 的输入输出操作在物理上并行。

输入进程行为数据后，单击“确定”按钮，系统将模拟优先级调度策略，画出如图 3.15 所示的调度时序图，给出各个进程的周转时间以及全部进程完成总的周转时间。这张图

图 3.12 优先级调度实践的启动界面

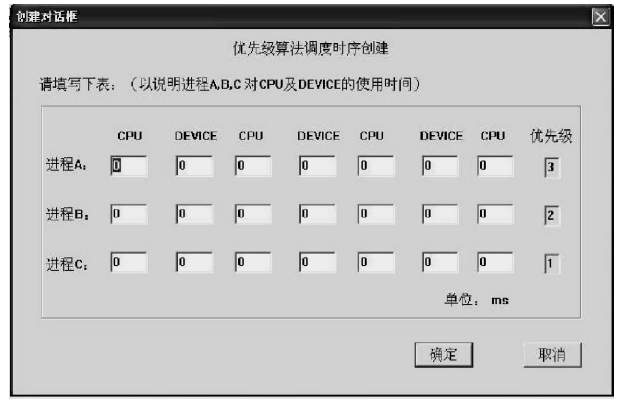

图 3.13 多进程调度设置初始界面

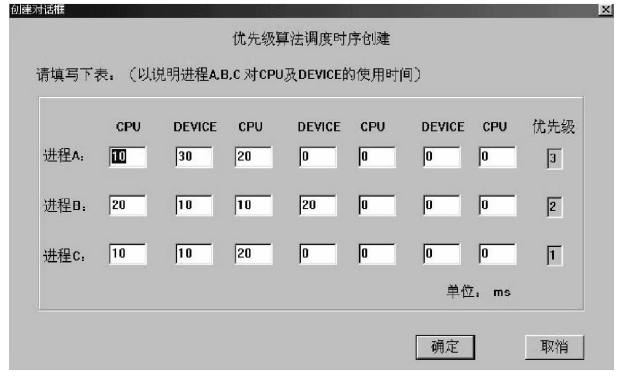

图 3.14 设置欲调度运行的进程实例

很好地体现了并发性和调度策略。

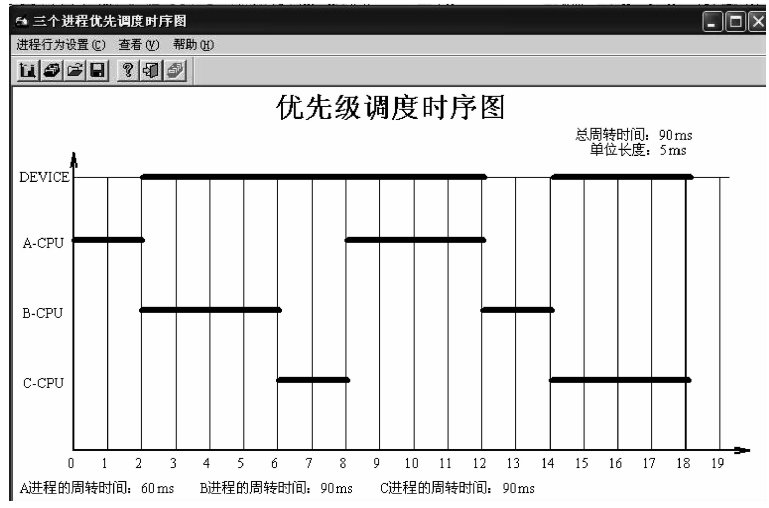

图 3.15　优先调度时序图

图 3.15 中显示，进程 A 的周转时间为 60ms，进程 B、C 的周转时间均为 90ms，即进程 B、C 是同时完成的。总的周转时间也是 90ms。

(2) 本书配套的网络课件中，有多道程序并发执行交互练习的例子。读者可以进入操作系统教学网站，单击主窗口中的“操作系统引论”|“操作系统的形成”|“练习”。

采用的例子是，假定系统中有一个 CPU、一台输入输出设备，在 $t=0$ 时刻，内存中同时到达 3 个程序 X、Y、Z，它们的行为如表 3.9 所示。

表 3.9　3 个程序 *X*、*Y*、*Z* 的行为

程　　序	I/O	CPU	I/O	CPU	I/O
X	20	20	20	30	20
Y		40	30	40	10
Z		30	20	30	

假定依程序 X、Y、Z 的优先顺序进行并发调度(不设就绪队列)，画出它们的调度时序，计算周转时间。在 Web 交互界面上有一个接收“你的答案”的区域，并提供“检测答案”的按钮。若答对了会得到夸奖。错了也不要紧，可以单击“清除错误答案”按钮重新输入新的答案。

小　　结

本章主要讨论 CPU 调度策略和死锁问题。CPU 调度使得多个进程有条不紊地共享一个 CPU，使每个用户进程在短时间内都有机会运行，就好像每个进程都有一个专用的虚拟处理机。一个负载较大的系统可能通过两级、三级调度才能让作业在处理机上执行。

一个好的调度策略对于加快作业总的周转时间、提高单位时间内的作业吞吐量、实现系统总的设计目标,是十分重要的。通常引入作业平均周转时间 T 和加权平均周转率 W 作为衡量调度算法的测度。

本章结合实例具体讨论了 FIFO、SJF、HRN 和 RR 和多级反馈队列 5 种常用的调度策略,给出了 T、W 的计算方法。FIFO 比较简单,体现了先来后到的公平原则,有利于长作业。SJF 照顾短作业,有最好的调度性能,使单位时间内平均作业吞吐量最大。HRN 则是上述两种策略的折中,既照顾短作业,也兼顾长作业。RR 算法使得每个作业在短时间内都有运行机会,可以兼顾长短作业,特别适合于进程调度。动态优先级算法,能够保证实时进程得到及时处理,多级反馈队列调度,是一种有自适应能力的调度算法,既能优待短的、以 I/O 为主的优先级高的进程,也能照顾以计算为主的进程。它能够自动地逐步判断进程的性质,做出适应各自特性的调度,表现出较多优点。动态优先级调度算法和多级反馈队列调度算法是当前较流行的调度算法。

本章讨论了产生死锁的 4 个必要条件:互斥条件、请求并保持条件、不剥夺条件和环路等待条件。分别给出了基于破坏请求并保持条件、不剥夺条件和环路等待条件的预防死锁的方法。

避免死锁的银行家算法从假定实施资源分配开始,力图能够找出进程的一种分配序列,使所有进程都能相继完成工作,则进行真正的资源分配。否则便拒绝分配,令申请资源的进程等待。这种方法的缺点是计算量大。

本章还用列出并约简进程—资源图的方法来检测系统是否发生了死锁。

发生了死锁怎么办?有选择地杀死(kill)一些进程,不然就重新引导系统。

习　题

3.1　处理机调度通常可分为哪三级?为什么要分级?

3.2　低级调度的功能是什么?为什么说它把一台物理的 CPU 变成了多台逻辑的 CPU?

3.3　假定系统中陆续有如表 3.10 所示的作业序列到达(表中数字为十进制)。

表 3.10　作业序列

作　业　号	到 达 时 间	运 行 时 间
1	10.0	0.3
2	10.2	0.5
3	10.4	0.1
4	10.5	0.4
5	10.8	0.1

作业调度程序自 10 时起开始调度,试分别用 FIFO、SJF、HRN 调度算法,计算周转时间 T 和加权平均周转率 W。

3.4　有如表 3.11 所示的进程序列，这些进程几乎同时依序到达。

表 3.11　进程序列

进　　程	CPU 周期	优　先　级
P_1	9	2
P_2	3	3
P_3	6	1

试用 FIFO、SJF 以及优先级调度算法计算它们的平均周转时间 T 和加权周转率 W。

3.5　对于 3.4 题的 3 个进程，试用 FIFO、SJF 以及优先级调度算法计算它们的平均等待时间。

3.6　对于 3.4 题的 3 个进程，试用轮转调度算法，取时间片为 1 个单位时间，计算它们的平均等待时间。

3.7　假定系统中有一个 CPU，一台输入输出设备，支持最多 A、B、C 这 3 个进程并发执行。如表 3.12 所示，系统采用优先但不可剥夺的调度策略，总是选择优先级最高的进程运行。假定 A、B、C 三者优先级 A 最高，B 其次，C 最低。

表 3.12　进程 *A*、*B*、*C* 的行为　　单位：ms

进　　程	CPU	I/O	CPU
A	30	60	20
B	20	30	40
C	30	10	20

试计算出每个进程的周转时间和总的周转时间，并画出调度时序图。

提示：将答案与光盘上“优先级调度”的计算结果对照。

3.8　何谓死锁？产生死锁的必要条件是什么？

3.9　图 3.16 表示一条带闸门的运河，其上有两座公路吊桥。运河与公路的交通都是单方向的。驳船前进到离 A 吊桥 100m 时就鸣笛示意，若桥上无车辆，吊桥就吊起，待船尾通过再放下。对 B 桥也同样处理。设船长 100m。

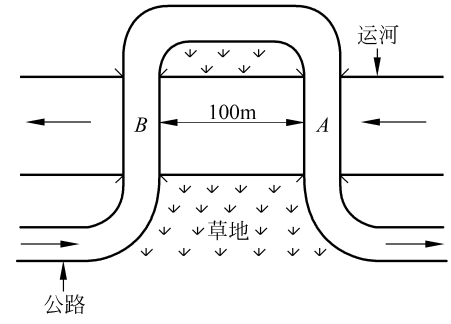

图 3.16　交通路线

(1) 车辆前进中是否可能发生死锁？在什么情况下发生？

(2) 按预防死锁的某种策略，用 P、V 操作写出汽车—驳船的同步算法。

3.10　利用银行家算法判断表 3.13 和表 3.14 中的状态是否为安全状态。如果是安全的，给出一种安全的分配序列。如果是不安全的，则说明为什么可能出现死锁。

表 3.13 状态一(可分配台数为 1)

进 程	占有台数	最大需求台数
进程 1	2	6
进程 2	4	7
进程 3	5	6
进程 4	0	2

表 3.14 状态二(可分配台数为 1)

进 程	占有台数	最大需求台数
进程 1	3	6
进程 2	5	7
进程 3	3	6
进程 4	0	2

3.11 如表 3.15 所示,假定系统中所有资源都是相同的,只可以一个一个地获得和释放这些资源,每个进程都不要求使用比系统资源总数更多的资源。试说明在下列系统中是否可能发生死锁。

现在假设没有一个进程需要两个以上资源,说明在表 3.16 中每个系统中是否可能发生死锁。

表 3.15 系统 1～5 的进程和资源

系 统	进程数目	资源总数
系统 1	1	1
系统 2	1	2
系统 3	2	1
系统 4	2	2
系统 5	2	3

表 3.16 系统 6～10 的进程和资源

系 统	进程数目	资源总数
系统 6	1	2
系统 7	2	2
系统 8	2	3
系统 9	3	3
系统 10	3	4

3.12 假定系统中有 2 个进程 P_1、P_2,有 2 个资源 R_1,3 个资源 R_2。进程 P_1 占有 R_1、R_2 各 1 个,又再申请 R_1、R_2 各 1 个。进程 P_2 占有 R_1、R_2 各一个,又再申请 1 个 R_2。试画出进程—资源图,并约简该图,以判断系统是否发生了死锁。

3.13 考虑系统中有 4 个相同类型的资源,当前有 3 个进程,每个进程最多需要 2 个资源。这种情况是否会发生死锁?为什么?

3.14 试用一个与本书介绍的不同的方法证明:若系统中有 n 个进程,m 个资源,资源是逐个申请的,每个进程至少申请一个资源,假定所有进程申请资源的总数 $R<m+n$,则系统不可能有死锁。