第 3 章　处理器调度与死锁

在当今的数字化时代，处理器调度与死锁问题显得愈发重要。处理器调度是操作系统中的核心功能之一，它负责合理地分配处理器资源，确保各个任务能够高效、有序地执行。而死锁问题，则是在并发系统中必须面对的挑战，它可能导致系统资源的浪费和性能的下降。本章将深入探讨处理器调度与死锁的相关概念和技术，帮助读者更好地理解并应对这些关键问题。

3.1　处理器调度概述

3.1.1　处理器调度的层次

处理器调度主要是对处理器运行时间进行分配，即按照一定算法或策略将处理器运行时间分配给各个用户进程，同时要尽量提高处理器的使用效率。调度算法的优劣、调度程序的实现方法直接影响到系统并发运行的整体效率。

一个进程在处理器上运行之前，必须占有一定的系统资源（如 CPU、内存和 I/O 设备等）。为了合理地安排进程占用的这些资源，为进程在处理器上运行做准备，操作系统也需要对其他资源进行调度，选择进程占用系统的其他资源。例如，系统满足某进程的磁盘 I/O 请求进行磁盘输入/输出，这称为磁盘调度。

在现代操作系统中，按照调度所实现的功能来分，通常把处理器分配给进程或线程的调度称为低级调度。除此之外，还有中级调度和高级调度，它们一起构成三级调度体系，如图 3-1 所示。其中，低级调度是该体系中不可缺少的最基本调度。将调度层次分为高、中和低三级调度，主要根据调度的信息量和频度划分，没有技术的高低之分。

1. 高级调度（作业调度）

高级调度的调度对象是作业。它的主要功能是根据特定的算法，从外存的后备作业队列中选择若干作业调入内存，并为这些作业创建相应的进程，分配所需的资源，最后将它们放入就绪队列等待执行。高级调度主要用于多道批处理系统，而在分时和实时系统中可能并不设置这一级别的调度。

2. 中级调度（内存调度）

中级调度的调度对象是进程。它负责管理进程在内存和外存之间的交换过程。当中

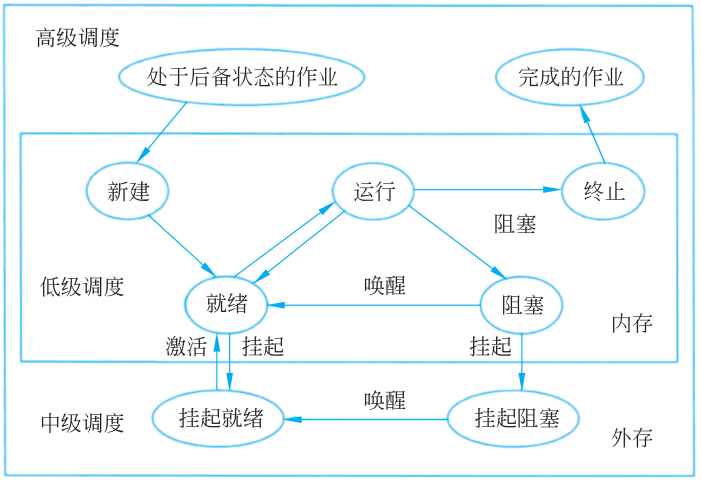

图 3-1　处理器的三级调度体系

某些进程暂时无法运行时，中级调度会将这些进程调至外存等待（即挂起）。而当这些进程具备了运行条件，且内存有足够的空闲空间时，中级调度会再次将这些进程从外存调入内存，并放入就绪队列中等待执行。这一级别的调度实质上执行了存储器管理中的对换功能，有助于解决内存空间紧张的问题。例如，Linux 操作系统专门要设置一个交换分区，主要用于中级调度。

3. 低级调度（进程调度）

低级调度的调度对象是进程或内核级线程。这是最基本的一种调度级别，其主要功能是决定哪个进程应该获得处理器的使用权。低级调度执行分配 CPU 资源的实际过程，它会根据一定的算法将 CPU 资源分配给就绪队列中的一个进程。这一级别的调度在多道批处理、分时和实时系统中都是必需的，是确保系统高效、稳定运行的关键环节。

具有中级调度的系统中，进程除了具有三个基本状态外，还具有挂起就绪和挂起阻塞两个状态。中级调度如图 3-2 所示，图中带箭头的直线表示中级调度程序所做的工作。

图 3-2　中级调度

中级调度实际上是存储管理中的对换功能，它控制进程对内存的使用。在虚拟存储管理系统中，进程只有被中级调度选中，才有资格占用内存。中级调度可以控制进程对内存的使用，从某种意义上讲，中级调度可通过设定内存中能够接纳的进程数来平衡系统负载，在一定时间内起到平滑和调整系统负载的作用。

3.1.2 作业和作业调度

在计算机领域中，作业指的是要在计算机上执行的任务或程序。这些作业可能由用户提交，也可能是由系统自动生成的任务。每个作业通常都会包含一些相关的数据和操作，用以完成特定的计算或处理任务。

作业调度是计算机系统中的一种重要机制，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，并按照一定的算法，从外存的后备队列中选取某些作业调入内存，为它们创建进程、分配必要的资源，然后再将新创建的进程插入就绪队列，准备执行。

作业调度算法的选择和设计是作业调度的关键。常用的作业调度算法包括先来先服务(First Come First Serve，FCFS)、短作业优先(Shortest Job First，SJF)、优先级调度算法、高响应比优先调度(Highest Response Ratio Next，HRRN)算法等。这些算法各有优缺点，适用于不同的应用场景和需求。

先来先服务：按照作业提交或进程变为就绪状态的先后次序进行调度，简单且易于实现，但可能不利于短作业和 I/O 繁忙的作业。

短作业优先：选择执行时间较短的作业进行调度，旨在减少大多数作业的等待时间，但可能导致长作业长时间得不到执行。

优先级调度算法：根据作业的优先级进行调度，优先级高的作业优先执行。这种算法可以确保重要任务得到优先处理，但可能需要外部设定合理的优先级。

高响应比优先调度算法：同时考虑作业的等待时间和执行时间，旨在平衡长短作业的调度需求。

作业调度还需要考虑资源的合理分配和管理，包括处理器、内存、磁盘等资源的分配，以确保作业能够正常执行并充分利用系统资源。

对于有依赖关系的作业，作业调度需要按照一定的顺序和时序要求进行调度。同时，对于有时间限制的作业，如实时任务或紧急任务，需要优先调度以确保在规定的时间内完成。

3.1.3 进程调度

进程调度有两种基本的方式，分别为非抢占式(non-preemptive)和抢占式(preemptive)。

(1) 非抢占式。进程一旦获得处理器变为运行状态，其他进程就不能中断它的运行，即使当前阻塞进程中出现了优先级更高的请求，运行状态进程也不允许该进程抢占处理器，直到该进程运行完成或发生某个事件使其主动放弃处理器，才能调度其他进程获得处理器。

采用非抢占式调度时，引起进程调度的常见原因如下。

① 正在运行的进程运行完毕或因发生某事件而不能再继续运行。

② 运行中的进程提出 I/O 请求而暂停运行，如等待慢速的 I/O 设备传输数据等。

③ 在进程通信或同步过程中运行了某种原语操作，如 P 原语、阻塞原语等，使正在运

行的进程因资源无法满足而“主动”放弃处理器的使用权。

这种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操作系统，但它难以满足实时任务的要求。因此，在要求比较严格的实时操作系统中，一般不宜采用此类调度方式。

(2) 抢占式。抢占式调度是指在进程并发运行中，如果就绪进程中某个进程优先级比当前运行进程的优先级还高，无论当前正在运行的进程是否结束，都允许高优先级进程抢占当前的处理器并立即运行。

抢占式调度可确保高优先级进程立即获得处理器。抢占式调度在实际系统中具有重要意义。为了帮助读者理解抢占式调度的必要性，不妨举一个源于现实生活中的例子。一个父亲有两个孩子，儿子 5 岁，女儿 4 岁。一天，父亲正在为今天过生日的女儿做蛋糕，两个孩子在院里玩耍。在制作过程中，儿子突然跑进来说，他的手在玩耍时被划破了，正在大量流血，让父亲赶紧给包扎一下伤口。在这个例子中，为女儿做蛋糕的过程可看作一个进程，给儿子包扎伤口的过程可看作另一个进程。父亲正在运行做蛋糕进程，该进程需要的时间较长。如果系统不允许抢占式调度，那么父亲就不能及时中止做蛋糕进程给儿子包扎伤口，这显然不符合日常逻辑。父亲应马上中止当前的做蛋糕进程，并要保存好现场，记录好蛋糕做到了什么环节。然后立即切换，运行包扎伤口进程，包扎完后恢复做蛋糕现场，继续做蛋糕。

在某些计算机系统中，为了实现某种目的，有些进程需要优先运行，只有采用抢占式调度才能满足它们的需求。抢占式调度对提高系统吞吐率、加速系统响应时间都有好处。

① 在支持抢占式调度的系统中，一般的抢占原则如下。

- 优先权原则：就绪的高优先权进程有权抢占低优先权进程的 CPU。
- 短作业优先原则：就绪的短作业有权抢占长作业的 CPU。
- 时间片原则：一个时间片耗尽后，系统重新进行进程调度。

② 引起抢占式调度的事件可归纳为以下三类。

- 运行进程的时间片耗尽。为了让所有就绪状态进程都有机会在处理器上运行，可以把 CPU 的运行时间按一定长度分成时间片。每个进程获得 CPU 后只运行一个时间片，时间片耗尽后，系统把 CPU 切换给其他进程。当内核发生时钟中断时，系统便检查和计算进程的剩余时间片。如果运行状态进程的时间片耗尽，内核程序马上剥夺其处理器，并分配给下一个轮转进程。
- 当实时信号或实时事件发生时，内核必须立即将处理器分配给相应的实时进程，以满足实时处理任务的需求。
- 当某个就绪状态进程的优先级高于当前进程优先级时，内核进程剥夺低优先级进程占有的处理器，并分配给高优先级进程。

任何操作系统都必须支持非抢占式调度，而对抢占式调度的支持可视具体需要而定。一般情况下，通用操作系统都支持基于时间片的抢占式调度，实时操作系统则必须支持严格的、满足规定的抢占式调度。

3.2 调度算法

3.2.1 调度算法的目标

进程调度目标是指进程调度需要达到的最终结果或目的。一般而言，有以下几种调度目标。

（1）公平性。保证每个进程得到合理的处理器时间和运行速度。例如，不能由于采用某种调度算法而使得某些进程长时间得不到处理器的运行而出现“饥饿”现象。要在保证某些进程优先权的基础上，最大限度地实现进程运行的公平性。

（2）高效率。保证处理器得到充分利用，不让处理器由于空闲等待而浪费大量时间，力争使处理器的绝大部分时间都在“忙碌”地运行有效指令。

（3）低响应时间。保证交互式命令的及时响应和运行。对于交互式命令或交互性较强的进程，不能让用户等待过长时间，必须保证在规定时间内给出运行结果。

（4）高吞吐量。要实现系统高吞吐量，缩短每个进程的等待时间。

（5）特殊应用要求。保证优先运行实时进程或特殊应用进程，以满足用户的特殊应用要求。

不同类型的操作系统有不同的调度目标。下面介绍一下常见操作系统中作业/进程的调度目标。

（1）多道批处理操作系统。多道批处理操作系统强调高效利用系统资源和高作业吞吐量。进程提交给处理器后就不再与外部进行交互，系统按照调度策略安排它们运行，直到各个进程完成为止。

（2）分时操作系统。分时操作系统更关心多个用户的公平性和及时响应性，它不允许某个进程长时间占用处理器。分时操作系统多采用时间片轮转调度算法或在其基础上改进的其他调度算法，但处理器在各个进程之间频繁切换会增加系统的时空开销，延长各个进程在系统中的存在时间。因此，分时操作系统最关注的是交互性和各个进程的均衡性，对进程的运行效率和系统开销要求并不苛刻。

（3）实时操作系统。实时操作系统必须保证实时进程的请求得到及时响应，往往不考虑处理器的使用效率。实时操作系统采取的调度算法和其他类型操作系统采取的调度算法有很大不同，其调度算法的最大特点是可抢占性。

（4）通用操作系统。通用操作系统对进程调度没有特殊限制和要求，选择进程调度算法时，主要追求处理器使用的公平性以及各类资源使用的均衡性。

设计操作系统时，设计者选择哪些调度目标在很大程度上取决于操作系统自身的特点。

3.2.2 先来先服务调度算法

先来先服务调度算法的基本原则是按照进程或作业请求的先后顺序进行调度，即先

到达的请求会先被调度执行。这种策略简单直观，易于实现。

尽管 FCFS 算法简单易行，但在某些情况下可能不是最优选择。特别是当系统中存在大量短作业和少量长作业时，长作业可能会长时间占据处理器，导致短作业长时间等待。因此，在选择使用 FCFS 算法时需要考虑作业或进程的到达模式和服务时间分布。

FCFS 算法适用于那些对实时性要求不高且作业或进程到达时间相对均匀的系统。在一些简单的批处理系统或早期计算机系统中得到广泛应用。

3.2.3　短作业(进程)优先算法

短作业(进程)优先算法要求用户或系统对每个作业或进程的执行时间有一个预估。在作业调度时，系统会依据这些预估时间，选择执行时间最短的作业进行调度。如果多个作业具有相同的执行时间，则可以按照它们到达的顺序进行调度。

SJF 算法的实现可以分为非抢占式和抢占式两种。非抢占式 SJF 算法中，一旦 CPU 开始执行一个作业，就会一直执行到该作业完成，除非发生 I/O 请求等中断。而抢占式 SJF 算法则允许当一个新作业到达，且其执行时间比当前正在执行的作业短时，操作系统可以中断当前作业，将 CPU 分配给新作业。

SJF 算法存在下述问题。

(1) 长作业饥饿问题：在持续有短作业到达的情况下，长作业可能会长时间得不到调度，从而导致“饥饿”现象。这会影响系统的公平性和用户体验。

(2) 预估执行时间的准确性：SJF 算法的有效性高度依赖于对作业执行时间的准确预估。然而，在实际应用中，准确预估作业的执行时间往往是困难的，这可能会影响算法的性能。

(3) 抢占式 SJF 的实现难度：虽然抢占式 SJF 算法在理论上具有更优的性能，但它的实现需要操作系统具备相应的抢占能力，这增加了系统的复杂性和开销。

(4) 不适用于所有场景：SJF 算法主要适用于批处理系统和部分交互式系统。对于需要严格保证作业执行顺序或具有特殊实时性要求的系统，SJF 算法可能不是最佳选择。

与先来先服务(FCFS)调度算法相比，SJF 算法在减少平均等待时间和提高系统吞吐量方面具有明显优势。然而，FCFS 算法在实现上更为简单，且不存在长作业饥饿的问题。与优先级调度算法相比，SJF 算法不需要显式地为每个作业分配优先级，而是根据执行时间自动进行优先级排序。

3.2.4　优先级调度算法

优先级调度算法的原理是根据任务的优先级来决定其执行的顺序。每个任务或进程都被赋予一个优先级，当系统需要选择下一个要执行的任务时，它会选择优先级最高的任务。这种策略可以确保重要的任务能够优先得到处理资源。

优先级调度算法实现方式可以分为以下两种。

(1) 静态优先级调度：在这种方式下，任务的优先级在创建时就被固定下来，不会随时间而改变。这种方法的优点是简单且开销小，但缺点是它不能动态地适应系统状态的变化。

(2) 动态优先级调度：与静态优先级不同，动态优先级调度会根据任务的状态、等待时间、执行时间等因素动态地调整任务的优先级。这种方法更加灵活，可以根据系统的实际情况来优化任务的执行顺序。

优先级调度算法广泛应用于各种需要确保重要任务优先执行的系统中，如实时系统、多媒体系统等。为了克服低优先级任务的“饥饿”问题，可以采用一些改进策略，如优先级继承、优先级老化等。

3.2.5 时间片轮转算法

1. 算法原理

时间片轮转(Round Robin Scheduling，RR)算法是将 CPU 的处理时间划分为若干时间片，每个进程被分配一个时间段，即它的时间片，用于执行。系统按照先来先服务(FCFS)的原则，将所有就绪的进程排成一个队列。每次调度时，CPU 分配给队首的进程，并允许它执行一个时间片的长度。当时间片用完时，计时器会发出时钟中断请求，调度程序据此信号停止当前进程的执行，并将其移至就绪队列的末尾。然后，调度程序选择就绪队列中的新队首进程，并分配给它一个时间片进行执行。这个过程不断重复，以确保所有就绪队列中的进程都能获得 CPU 的执行时间。时间片轮转算法广泛应用于多道批处理系统和分时系统中，如 UNIX/Linux 操作系统就采用了这种调度算法。

2. 时间片长度的选择

时间片的大小对系统性能有着重要影响。如果时间片设得太短，会导致进程切换频繁，从而增加了系统开销，降低了 CPU 效率。这是因为进程切换需要保存和装入寄存器值、更新各种表格和队列等操作，这些都需要消耗一定的 CPU 时间。相反，如果时间片设得太长，虽然减少了进程切换的次数，但可能导致对短的交互请求的响应变差。例如，在一个分时系统中，如果有多个用户几乎同时发出请求，而时间片设置过长，那么某个用户可能需要等待较长时间才能获得 CPU 的执行权。

因此，在选择时间片长度时，需要综合考虑系统的响应时间要求、就绪队列中进程的数目以及系统的处理能力等因素。理想的情况是，时间片的长度应该足够长，以减少进程切换的开销，但同时又不能太长，以免影响系统的响应时间。

3. 算法的优点与缺点

(1) 优点。

公平性：时间片轮转算法确保每个进程都有机会获得 CPU 的执行权，避免了某些进程长时间独占 CPU 的情况。

响应时间短：对于短作业来说，由于每个进程都能在时间片结束后立即获得 CPU 的执行权(如果它仍在就绪队列中)，因此响应时间相对较短。

（2）缺点。

时间片大小的选择难题：如前所述，时间片的大小对系统性能有显著影响，选择不合适的时间片长度可能导致系统性能下降。

不适用于实时系统：时间片轮转算法无法保证进程的响应时间，因此对于需要严格实时性的系统来说可能不适用。

3.2.6　多级队列调度算法

1. 算法原理与基本概念

多级队列调度算法将进程划分为多个队列，通常为 RQ1、RQ2 等，每个队列具有不同的优先级。例如，RQ1 可能采用时间片长度为 7 的时间片轮转算法，而 RQ2 则可能采用短进程优先算法。这些队列按照优先级从高到低排列，确保高优先级的进程能够优先获得 CPU 资源。

在此算法中，每个进程首先被放入最高优先级的队列中执行。如果进程在分配的时间片内未能完成执行，它将被移至下一个较低优先级的队列中。这个过程会持续进行，直到进程在最低优先级的队列中完成执行。

2. 算法特点与优势

灵活性：多级队列调度算法能够根据进程的执行情况动态调整其优先级。这种灵活性使得算法能够适应不同性质的进程，如 CPU 密集型或 I/O 密集型进程，从而提高系统的整体性能。

公平性：该算法通过为每个进程分配时间片并确保所有进程都有机会执行，从而实现了公平的 CPU 时间分配。这避免了某些进程长时间占用 CPU 资源，导致其他进程得不到执行的情况。

适应性：多级队列的设置使得系统可以根据进程的性质进行调整。例如，对于需要快速响应的进程，可以将其放入高优先级队列以确保其及时执行；而对于执行时间较长或资源消耗较大的进程，可以将其放入低优先级队列以避免对系统性能产生过大影响。

3.2.7　多级反馈队列调度算法

1. 算法原理

多级反馈队列调度算法通过设置多个不同优先级的就绪队列来实现进程的调度。每个队列都有自己的时间片长度，通常优先级越高的队列，其时间片越短。新创建的进程或刚从阻塞状态唤醒的进程会被加入最高优先级的队列中。若进程在分配的时间片内未完成执行，则会被移至下一个较低优先级的队列。这种降级的过程会持续进行，直到进程在某一队列中完成执行。

2. 算法特点

多级队列与优先级：算法中设置了多个队列，每个队列具有不同的优先级。这种设

计可以灵活应对不同类型的进程，确保重要的进程能够优先得到执行。

时间片轮转与降级策略：在每个队列内部，进程按照时间片轮转的方式进行调度。若进程在当前队列的时间片内未完成，则会被降级到下一个队列。这种策略既保证了进程的公平性，又避免了某个进程长时间占用 CPU 资源。

动态调整：算法可以根据系统的实际情况动态调整进程的优先级和队列的分配。例如，当系统负载较轻时，可以将更多的进程放在高优先级的队列中，以提高系统的响应速度；而当系统负载较重时，则可以适当增加低优先级队列的进程数量，以确保系统的稳定性。

3. 算法实施

队列数量与时间片设置：在实施多级反馈队列调度算法时，需要首先确定队列的数量以及每个队列的时间片长度。队列的数量和时间片的设置需要根据系统的实际需求和负载情况来进行调整。

进程迁移与调度：当进程在一个队列中的时间片用完且未完成执行时，它会被迁移到下一个较低优先级的队列中。同时，调度器会按照优先级从高到低的顺序依次检查各个队列，选择最高优先级队列中的进程进行执行。

抢占式与非抢占式：多级反馈队列调度算法可以是抢占式的，即当高优先级队列中有新进程加入时，可以抢占当前正在执行的低优先级进程的 CPU 资源；也可以是非抢占式的，即当前正在执行的进程会一直执行到完成或主动放弃 CPU 资源。

3.2.8 高响应比优先调度算法

高响应比优先调度算法适用于需要综合考虑作业等待时间和执行时间的场景，如批处理系统或需要处理大量作业的环境。在这些场景中，HRRN 能够提供相对公平且高效的作业调度方案。

1. 算法原理

高响应比优先调度算法的基本原理是，根据作业的响应比来分配 CPU 资源。响应比是作业等待时间与要求服务时间的比值再加 1，即响应比＝（等待时间＋要求服务时间）/要求服务时间。这个比值反映了作业对 CPU 的渴望程度，比值越高，表示该作业越应该优先得到执行。

算法的具体步骤如下。

（1）当一个作业到达时，系统计算其响应比。

（2）根据响应比的大小，对所有就绪的作业进行排序。

（3）选择响应比最高的作业进行调度执行。

（4）如果多个作业同时到达，则可以根据到达时间或其他因素来确定优先级。

2. 算法特点

（1）综合考虑等待时间与执行时间：与 FCFS 和 SJF 相比，HRRN 不仅考虑了作业的执行时间，还考虑了作业的等待时间。这使得长作业和短作业都有机会得到及时执行，

避免了长作业长时间等待的问题。

（2）动态优先级：HRRN 通过计算响应比来动态确定作业的优先级。这使得作业的优先级不是固定的，而是随着等待时间的增加而提高。这种动态优先级机制能够更好地适应系统状态的变化。

（3）公平性：由于响应比的计算方式，HRRN 能够在一定程度上保证作业的公平性。即使一个作业的执行时间很长，只要它的等待时间足够长，它的响应比也会提高，从而有机会得到执行。

3.2.9　基于公平原则的调度算法

1. 公平原则的重要性

在计算机系统中，资源如 CPU 时间、内存、网络带宽等都是有限的。当多个用户或进程同时请求这些资源时，如何公平、高效地分配它们成为一个关键问题。基于公平原则的调度算法正是为了解决这一问题而诞生的。它不仅有助于防止某些用户或进程长时间占用资源而导致其他用户无法得到服务，还能提高系统的整体性能和用户满意度。

2. 公平调度算法的实现

（1）公平队列调度。公平队列调度是基于公平原则的调度算法中最常用的一种。在这种调度策略下，用户的请求被分配到不同的队列中，然后系统按照一定的规则（如轮询、加权轮询等）从各个队列中取出请求进行处理。这种方式确保了每个用户都能按照其请求的先后顺序得到服务，有效避免了某些用户长时间等待的问题。

（2）多级反馈队列调度。除了公平队列调度外，多级反馈队列调度也是实现公平原则的一种有效方式。在这种调度策略中，系统设置了多个不同优先级的队列，每个队列具有不同的时间片长度。新进程首先进入最高优先级的队列，若在该队列的时间片内未完成执行，则会被降级到下一个较低优先级的队列。这种方式既考虑了进程的优先级，又通过时间片轮转的方式保证了公平性。

3. 公平调度算法的应用场景

基于公平原则的调度算法在多个领域都有广泛的应用。例如，在云计算环境中，云服务提供商需要公平地为每个用户提供计算资源。通过采用公平调度算法，可以确保每个用户都能根据其需求获得相应的资源，从而避免了资源过度集中或浪费的情况。

在操作系统中，基于公平原则的调度算法也被广泛应用于进程调度。通过公平地分配 CPU 时间片，系统可以确保每个进程都能得到执行的机会，从而提高了系统的整体性能和响应速度。

3.3　实时调度

3.3.1　实现实时调度的基本条件

实现有效的实时调度需要满足以下几个基本条件。

(1) 接收并处理关键信息。系统必须能够接收关于任务的关键信息,这包括任务的就绪时间、截止时间、处理时间、资源要求和优先级。

(2) 足够的处理能力。实时系统需要具备足够的处理能力,以确保实时任务能按时完成。这可以通过增强单处理器能力或采用多处理器系统来实现。

(3) 可调度的评估。系统的可调度性取决于所有实时任务的处理时间与周期时间的比例,这在多处理器系统中还需考虑处理器的数量。

3.3.2 实时调度算法分类

实时调度算法可以根据不同的标准进行分类。

(1) 根据调度表和可调度性分析是脱机还是联机实现,可分为静态调度和动态调度。

(2) 按系统分类,可分为单处理器调度、集中式多处理器调度和分布式处理器调度。

(3) 按任务是否可抢占,可分为抢占式调度和不可抢占式调度。

3.3.3 最早截止时间优先算法

最早截止时间优先(Earliest Deadline First,EDF)算法的核心思想是根据任务的截止时间来动态分配优先级,以确保所有任务都能在其截止时间之前完成。

1. EDF 算法的基本原理

EDF 算法的基本原理非常简单直观:在每个新的就绪状态,调度器会从那些已就绪但还没有完全处理完毕的任务中选择最早截止时间的任务,并将执行该任务所需的资源分配给它。这种策略确保了最紧急的任务能够优先得到处理,从而满足实时性要求。

当有新任务到来时,调度器会立即计算并更新任务的顺序。这意味着,正在运行的任务可能会被新到来的更紧急的任务打断。这种抢占式的调度方式,使得系统能够灵活应对各种突发情况,确保任务能够按时完成。

2. EDF 算法的特点

(1) 动态优先级分配:EDF 算法根据任务的截止时间来动态分配优先级。这意味着,任务的优先级并不是固定的,而是随着其截止时间的临近而提高。这种动态优先级分配机制使得 EDF 算法能够很好地适应实时系统的需求。

(2) 抢占式调度:在 EDF 算法中,如果一个新任务的截止时间早于当前正在运行的任务,那么新任务会立即抢占 CPU 资源,以确保其能够在截止时间之前完成。这种抢占式调度方式提高了系统的灵活性和响应速度。

(3) 最优调度策略:在单 CPU、任务之间相互独立且没有关联的场景下,EDF 算法被证明是一种最优的调度策略。它能够最大限度地提高 CPU 的利用率,并确保所有任务都能在其截止时间之前完成。

3.3.4 最低松弛度优先算法

最低松弛度优先(Least Laxity First,LLF)算法是根据任务的剩余处理时间和截止

时间来动态确定任务的优先级，以确保系统能够优先处理最紧急的任务，从而最大限度地降低任务的响应时间和延迟。

1. LLF 算法的基本原理

在 LLF 算法中，每个任务都有一个称为“松弛度”(Slack Time)的参数，它表示当前时间到任务截止时间的剩余时间。松弛度的计算公式为：任务的松弛度＝任务的截止时间－当前时间－任务的剩余处理时间。这个公式可以帮助系统了解每个任务的紧迫程度。

2. LLF 算法的实现步骤

(1) 计算松弛度：对于系统中的每个任务，根据其剩余处理时间和截止时间，利用上述公式计算出松弛度。

(2) 排序就绪队列：系统维护一个按松弛度排序的就绪队列，松弛度最低的任务排在队列的最前面。

(3) 任务调度：调度器从就绪队列中选择松弛度最低的任务进行执行。当该任务的松弛度减为 0 时，它必须立即抢占 CPU，以保证按截止时间的要求完成任务。

(4) 更新队列：当有新的任务进入就绪状态或者当前运行的任务状态发生变化时，系统会重新计算松弛度并更新就绪队列。

3. LLF 算法的特点

(1) 动态优先级：LLF 算法根据任务的实际情况动态分配优先级，这使得系统能够灵活应对各种任务需求。

(2) 抢占式调度：当某个任务的松弛度变为最低时，它可以抢占正在运行的任务，从而确保最紧急的任务得到优先处理。

(3) 保证响应时间：由于优先处理松弛度最低的任务，LLF 算法能够最大限度地降低任务的响应时间和延迟。

3.3.5　优先级倒置

1. 优先级倒置的产生原因

优先级倒置通常发生在以下场景：一个低优先级的任务先占用了某个临界资源(如信号量、互斥锁等)，而此时一个高优先级的任务也需要访问该资源。由于资源已被低优先级任务占用，高优先级任务不得不等待。如果此时有一个中优先级的任务(其优先级高于低优先级任务但低于高优先级任务)就绪，它可能会抢占 CPU，导致高优先级任务长时间得不到执行，这就是所谓的优先级倒置。

具体来说，产生优先级倒置的条件包括系统采用可抢占式的优先级调度策略，不同优先级的任务需要访问共享资源，低优先级任务先占用了共享资源。

2. 优先级倒置的后果

优先级倒置会导致高优先级任务的响应时间延长，甚至可能错过其规定的实时期限。在实时系统中，这种延迟可能会导致严重的后果，如系统性能下降、任务失败或整体系统

的不稳定。例如，在 CAN 总线系统中，优先级倒置可能导致高优先级消息的送达时间延迟，从而影响系统的实时性和可靠性。

3. 解决优先级倒置的方法

为了解决优先级倒置问题，可以采取以下几种方法。

（1）优先级继承方法。当高优先级任务被低优先级任务阻塞时，低优先级任务会临时继承高优先级任务的优先级，以便尽快执行完毕并释放资源。这种方法可以有效地解决优先级倒置问题，但可能会引入额外的复杂性和开销。

（2）优先级置顶协议（Priority Ceiling Protocol）。该协议规定，获得共享资源的任务的运行时优先级比其他任何尝试访问该资源的任务的优先级都要高。这可以确保一旦某个任务占用了资源，它就能尽快完成并释放资源，从而避免阻塞其他高优先级的任务。

（3）避免不同优先级线程共享资源。通过适当的代码组织和设计，可以尽量避免不同优先级的线程共享相同的资源。例如，可以为不同优先级的任务分配独立的资源，或者采用同步机制来确保资源的有序访问。

（4）使用不可抢占区。在某些关键区域，可以禁止任务的抢占，从而确保高优先级任务不会被低优先级任务打断。然而，这种方法需要谨慎使用，因为它可能会降低系统的并发性和响应性。

3.4 死锁概述

3.4.1 资源问题

在计算机系统中，资源是有限的，而多个进程可能同时需要这些资源来完成各自的任务。当多个进程争夺同一资源时，就可能出现资源分配的问题。如果系统不能合理地分配和管理这些资源，就可能导致死锁的发生。

（1）资源的互斥使用。某些资源，如打印机、文件等，需要被互斥地使用。即在一个时刻只能有一个进程或线程访问该资源。如果多个进程或线程同时请求这类资源，并且都不愿意释放已占用的资源，就会形成死锁。这是因为每个进程或线程都在等待其他进程释放资源，而自身又持有其他进程所需的资源，从而形成了一种环形等待的状态。

（2）资源的有限性。系统中的资源是有限的，包括 CPU、内存、磁盘空间、网络带宽等。当多个进程或线程同时请求这些资源时，如果资源的数量不足以满足所有请求，就会发生资源争用的情况。这种资源的有限性是导致死锁的一个重要原因。

（3）资源的请求与保持。当一个进程或线程在请求一个新的资源时，如果它已经持有了其他资源，并且不愿意在请求新资源之前释放已持有的资源，那么就有可能发生死锁。这是因为其他需要这些资源的进程或线程会被阻塞，等待资源的释放。如果多个进程或线程都存在这种情况，就会形成一个相互等待的僵局。

（4）资源的循环等待。当多个进程或线程之间形成一种环形等待链时，也会发生死锁。即每个进程或线程都在等待下一个进程释放资源，而下一个进程又在等待其他进程

释放资源，如此循环下去。在这种情况下，每个进程或线程都无法继续执行，因为它们都在等待其他进程释放所需的资源。

3.4.2　死锁的定义、产生的原因、必要条件

死锁是指多个进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法向前推进。此时系统处于死锁状态，这些永远在互相等待的进程称为死锁进程。

死锁产生的原因通常源于多个进程对资源的争夺。具体来说，有以下几点。

（1）系统资源不足：当系统资源不足以满足所有进程的需求时，进程之间就会争夺这些资源，从而导致死锁。

（2）进程运行推进的顺序不当：如果进程的运行顺序不合理，也可能导致死锁。例如，两个进程分别占据了某些资源，并都在等待对方释放资源，这时就形成了死锁。

（3）资源分配不当：如果系统在分配资源时没有考虑到进程之间的依赖关系，或者分配策略不合理，也可能导致死锁。

产生死锁有 4 个必要条件，它们是互斥条件、请求与保持条件、不可抢占条件和循环等待条件。只有这 4 个条件同时成立时，才会发生死锁。

（1）互斥条件：进程对分配到的资源进行排他性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其他进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

（2）请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3）不可抢占条件：进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

（4）循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

3.4.3　资源分配图

资源分配图是一种用于描述系统中资源和进程之间关系的图形化表示。在资源分配图中，节点可以表示进程或资源，边则表示进程对资源的请求或分配关系。通过分析资源分配图，可以更清晰地了解系统中的资源分配情况，以及可能存在的死锁问题。

例如，在一个简单的系统中，如果有两个进程 P1 和 P2，以及两个资源 R1 和 R2。当 P1 占用了 R1 并请求 R2，而 P2 占用了 R2 并请求 R1 时，就形成了一个循环等待的情况。这种情况可以在资源分配图中清晰地表示出来，从而帮助我们识别和解决死锁问题。

3.4.4　死锁的预防

死锁预防主要是通过破坏产生死锁的 4 个必要条件中的一个或多个来避免死锁的发生。

（1）破坏互斥条件。这在实际系统中往往是不可能的，因为某些资源本身就是非共享的。

(2) 破坏请求与保持条件。可以采用预先静态分配法,即进程在运行前一次性申请完它所需要的全部资源。这种方法简单、易于实现且安全,但资源利用率低、灵活性差。另一种方法是通过动态分配来避免这种情况,即进程在申请资源时,如果它不能立即获得所需要的全部资源,就必须释放已经占有的资源,从而破坏“请求与保持”条件。

(3) 破坏不可抢占条件。当一个进程获得某项资源的使用权后,若新的请求无法被满足,它必须释放已获得的所有资源,待以后需要时再重新申请。这就意味着,一个进程已占有的资源会被暂时释放,或者说是被抢占,从而破坏了不可抢占条件。

(4) 破坏循环等待条件。采用顺序资源分配法。首先给系统中的所有资源类型进行编号,每一个进程必须按编号递增的顺序请求资源,同类资源(即编号相同的资源)一次申请完。这种方法的问题在于资源的编号必须相对稳定,这就限制了新设备类型的增加、资源的增加、资源的正常使用以及限制用户简单、自主地编程,给系统维护带来麻烦,甚至还需要防止用户误操作而破坏系统管理规则。

3.4.5 死锁的避免

死锁避免的基本思想是动态地检测资源分配状态,以确保系统不会进入不安全状态,从而预防死锁的发生。与死锁预防不同,死锁避免不需要严格限制资源的分配方式,而是在资源分配过程中动态地检查是否会导致死锁,并据此决定是否进行资源分配。

在死锁避免策略中,系统通常采用一种称为“银行家算法”的资源分配算法。该算法模拟了银行家在贷款时确保自身流动性的策略,以确保在分配资源后,系统仍然能够满足其他进程的最大资源需求。

银行家算法的基本步骤如下。

(1) 检查请求。当一个进程请求资源时,系统首先检查该请求是否超过了该进程事先声明的最大需求。

(2) 试探性分配。如果请求没有超过最大需求,系统则进行试探性的资源分配,即假设满足该进程的请求。

(3) 安全性检查。在试探性分配后,系统通过安全性算法来检查当前资源分配状态是否安全。安全性算法会尝试找到一个安全序列,即一个能够使所有进程都顺利完成资源分配和释放的进程序列。

(4) 资源分配决策。如果系统处于安全状态,即存在安全序列,则系统正式分配资源给请求的进程;否则,系统将拒绝该请求,并可能让进程等待或采取其他措施以避免死锁。

3.4.6 死锁的检测和解除

1. 死锁的检测

死锁的检测是允许死锁发生,但操作系统会不断监视系统进展情况,判断死锁是否真的发生的过程。具体方法如下。

（1）资源分配图检测。通过资源分配图来检测死锁。资源分配图由进程节点、资源节点及边组成。利用资源分配图，可以找出那些既不阻塞又不是孤点的进程，并尝试消去其请求边和分配边，使之成为孤立的节点。如果最后所有的进程节点都被孤立，则系统没有发生死锁；否则，系统中存在死锁。

（2）定时检测。设定一个检测的周期，如每天、每周或特定时间间隔进行一次死锁检测。这种方法虽然可能会带来一定的系统开销，但可以及时发现并处理死锁情况。

（3）资源利用率下降时检测。当系统资源利用率低于某个预设值时，启动死锁检测算法。这种方法可以在系统资源紧张时及时发现并解决死锁问题，提高系统效率。

2. 死锁的解除

当检测到系统中已发生死锁时，需要采取措施将进程从死锁状态中解脱出来。常见的死锁解除方法包括以下几种。

（1）撤销陷于死锁的全部进程。这是一种简单直接的方法，但可能会导致较大的损失，因为所有相关进程都需要重新开始。

（2）逐个撤销陷于死锁的进程，直到死锁不存在。这种方法相对灵活，可以根据实际情况选择撤销哪些进程，以减少损失。

（3）从陷于死锁的进程中逐个强迫放弃所占用的资源，直至死锁消失。这种方法需要精确地确定哪些资源是导致死锁的关键，并强制释放这些资源。

（4）从另外一些进程那里强行剥夺足够数量的资源分配给死锁进程，以解除死锁状态。这种方法需要谨慎操作，以避免对系统造成更大的影响。

3.5　思政案例

处理器管理是操作系统的重要组成部分，主要涉及进程管理、处理器调度及死锁等问题。在这一领域，同样可以深入挖掘思政元素，将专业知识传授与价值引导相结合。以下是对处理器管理中思政元素的挖掘及思政教学案例的探讨。

3.5.1　思政元素

竞争意识：处理器管理涉及多个进程对有限处理器资源的竞争。这可以引申出现代社会中无处不在的竞争现象，从而培养学生的竞争意识。通过竞争，个体能够不断成长，企业能够创新发展，国家能够提升整体实力。

合作精神：在处理器管理中，多个进程之间需要协调合作，以实现资源的合理分配和系统的稳定运行。这体现了合作精神的重要性。在信息化时代，团队合作已成为完成各项任务、项目开发和公司管理不可或缺的因素。通过培养学生的团队合作精神，可以提升其集体意识和协作能力。

3.5.2 课程思政案例

1. 进程管理与合作精神

(1) 教学内容：进程管理基础、资源共享与冲突、进程同步与互斥。

(2) 思政育人目标：培养团队合作精神，增强沟通与协调能力，树立大局观念。

(3) 案例设计。

【引入阶段】

情境设定：首先，通过生动的动画或故事引入多任务操作系统的概念，展示多个进程如何在同一时间内争夺有限的处理器资源。强调这种场景类似于现实生活中的团队合作，每个成员都需要为了共同的目标而努力，同时又要协调好彼此之间的资源分配。

提出问题：引导学生思考“在这样一个复杂的系统中，如何确保每个进程都能得到合理的处理器时间，同时又不影响系统的整体稳定性和效率?”进而引出进程管理的概念，并暗示这与团队合作中的资源分配和协调有着异曲同工之妙。

【讲解阶段】

理论讲授：详细讲解进程管理的基本概念、作用以及常见的调度算法。通过图表和实例，展示不同调度算法对系统性能的影响，让学生理解进程管理的重要性。

思政融入：在讲解过程中，穿插合作精神的重要性。强调在团队中，每个成员都应有明确的角色定位，积极贡献自己的力量，同时也要学会倾听和尊重他人的意见，共同为团队目标的实现而努力。将进程之间的协同工作与团队合作相类比，加深学生对合作精神的理解。

【讨论与分析阶段】

小组讨论：将学生分成若干小组，每组分配一个具体的场景(如项目管理、科研合作等)，要求他们讨论并制定一个合理的资源分配和协调方案。鼓励学生从进程管理的角度思考，如何将合作精神融入团队工作中。

案例分析：选取一些成功的团队合作案例进行分享和分析，让学生看到合作精神在实际工作中的重要作用。同时，也可以分析一些失败的案例，引导学生思考其中的原因和教训。

【实践环节】

模拟实验：设计一个模拟实验，让学生扮演不同的进程角色，在限定的时间内完成一系列任务。通过调整调度算法和合作策略，观察并记录不同情况下的系统性能(如任务完成时间、资源利用率等)。

角色扮演：在团队项目中，鼓励学生扮演不同的角色(如项目经理、技术专家、协调员等)，通过实际操作体验团队合作中的资源分配、沟通协调等过程。

【总结与反思】

知识总结：回顾进程管理的基本概念、调度算法以及合作精神在团队合作中的重要性。强调在复杂系统中，有效的管理和协调是实现高效运行的关键。

思政升华：引导学生反思在团队合作中自己的表现和不足，思考如何进一步提升自己的合作精神和协调能力。同时，鼓励学生将所学知识和思政素养应用到实际生活中，为未来的学习和工作打下坚实的基础。

2. 处理器调度与竞争意识

（1）教学内容：处理器调度的基本概念和目的，常见的调度算法（如 FCFS、SJF、RR、优先级调度等）及其特点，调度算法对系统性能的影响。

（2）思政育人目标：培养学生的竞争意识，理解竞争对于个人成长和社会发展的重要性；引导学生学会在竞争中保持积极心态，寻找提升自我的途径。

（3）案例设计。

【引入阶段】

情境再现：通过动画或视频展示处理器调度的过程，让学生直观感受不同进程如何竞争处理器时间。强调这种竞争现象在现实生活中也普遍存在。

提出问题：引导学生思考“在处理器调度中，为什么需要竞争？如何才能在竞争中保持优势？”进而引出竞争意识的概念，并探讨其在个人成长和职业发展中的作用。

【讲解阶段】

理论讲授：详细介绍处理器调度的基本概念、调度算法以及它们对系统性能的影响。通过实例分析，让学生理解不同调度算法下进程之间的竞争关系。

思政融入：在讲解过程中，穿插竞争意识的培养。强调在竞争中保持积极的心态、不断提升自己的能力和技能的重要性。同时，也要引导学生认识到竞争与合作并不矛盾，良好的竞争环境可以促进团队和个人的共同成长。

【讨论与分析阶段】

案例分析：选取一些具有代表性的竞争案例（如职场竞争、学术竞赛等），引导学生分析其中的竞争策略和成功要素。鼓励学生思考如何在竞争中保持优势并避免陷入恶性竞争。

小组讨论：组织学生讨论如何在个人成长和职业发展中培养竞争意识。鼓励学生分享自己的经验和见解，并相互学习借鉴。

【实践环节】

模拟竞赛：设计一场模拟竞赛（如编程比赛、知识竞赛等），让学生在实践中体验竞争的过程和乐趣。通过竞赛的形式激发学生的竞争意识和求胜欲望。

职业规划：引导学生制定个人职业规划，明确自己的职业目标和发展路径。鼓励学生思考如何在未来的职业生涯中保持竞争优势并不断提升自己。

【总结与反思】

知识总结：回顾处理器调度的基本概念、调度算法以及竞争意识在个人成长和职业发展中的作用。强调在竞争中保持积极心态和不断提升自己的重要性。

思政升华：引导学生反思在竞争中的表现和不足，思考如何更好地平衡竞争与合作的关系。鼓励学生将所学知识和思政素养应用到实际生活中，为未来的学习和工作做好充分准备。

小结

本章深入探讨了处理器调度与死锁的相关问题。介绍了处理器调度的不同层次和多种调度算法，这些算法在提高系统效率和响应速度方面发挥着关键作用。同时，也分析了死锁问题的严重性和复杂性，阐述了预防、避免、检测和解除死锁的各种策略。

通过本章的学习，读者不仅可以掌握处理器调度和死锁的基本概念和技术，还学会了如何在实际应用中应对这些问题。

思考练习

1. 请简述处理器调度的三个层次及其主要特点。
2. 描述至少三种常见的调度算法，并比较它们的优缺点。
3. 什么是死锁？它产生的原因是什么？请列举死锁的必要条件。
4. 如何预防死锁的发生？请至少给出两种策略。
5. 当系统发生死锁时，应如何检测和解除？请简述你的方案。