03 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

> *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。*
>
> *在多道批处理系统中，一个作业从提交到获得处理机执行，直至作业运行完毕，可能需要经历多级处理机调度。*

3.1.1 处理机调度的层次

- 高级调度(High Level Scheduling)
- 低级调度(Low Level Scheduling)
- 中级调度(Intermediate Scheduling)

3.1.2 处理机调度算法的目标

1.处理机调度算法的共同目标

- 资源利用率

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

- 公平性

  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。

- 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。

- 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.其他os各自目标

（1）批处理系统的目标

- 平均周转时间短

  对每个用户而言，都希望自己作业的周转时间最短。但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：

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

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与系统为它提供服务的时间Ts之比，即W = T/Ts。

平均带权周转时间则可表示为：

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

- 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。

- 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。

（2）分时系统的目标

- 响应时间快
- 均衡性

（3）实时系统的目标

- 截止时间的保证
- 可预测性

3.2 作业与作业调度

> *在多道批处理系统中，作业是用户提交给系统的一项相对独立的工作。*
>
> *操作员把用户提交的作业通过相应的输入设备输入到磁盘存储器，并保存在一个后备作业队列中。再由作业调度程序将其从外存调入内存。*

3.2.1 批处理系统中的作业

1.作业和作业步

- 作业(Job)
  不仅包含程序和数据，还配有一份作业说明书。
  批处理系统中，是以作业为基本单位从外存调入内存。
- 作业步(Job Step)
  每个作业中相对独立又相互关联的步骤

2.作业控制块(Job Control Block，JCB)

> *作业在系统中存在的标志，其中保存了系统对作业进行管理和调度所需的全部信息。*

通常在JCB中包含的内容有：作业标识、用户名称、用户账号、作业类型(CPU繁忙型、I/O 繁忙型、批量型、终端型)、作业状态、调度信息(优先级、作业运行时间)、资源需求(预计运行时间、要求内存大小等)、资源使用情况等。

3.作业运行的三个阶段和三种状态

> *作业从进入系统到运行结束，通常需要经历收容（后备状态）、运行（运行状态）和完成（完成状态）三个阶段。*

- 收容阶段：把作业输入到硬盘，建立JCB，放入作业后备队列；
- 运行阶段：被作业调度选中后，分配资源并建立进程，放入就绪队列；
- 完成阶段：回收分配的资源，将作业运行结果形成输出文件。

3.2.2 作业调度的主要任务

> *作业调度的主要任务是，根据JCB中的信息，检查系统中的资源能否满足作业对资源的需求，按照一定的调度算法，从外存的后备队列中选取某些作业调入内存，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等待调度。*

因此，也把作业调度称为 接纳调度(Admission Scheduling) 。在每次执行作业调度时，都需做出以下两个决定。

1. 接纳多少个作业：取决于多道程序度（Degree of Multiprogramming），即允许多少个作业同时在内存中运行；
2. 接纳哪些作业：取决于调度算法。

3.2.3 作业调度算法

1. 先来先服务(first-come first-served，FCFS)调度算法

> *FCFS是最简单的调度算法， FCFS算法既可用于作业调度，也可用于进程调度 。*

系统将按照作业到达的先后次序（系统中等待时间最长）来进行调度。

2.短作业优先(short job first，SJF)的调度算法

> *由于在实际情况中，短作业(进程)通常占有很大比例*

SJF算法是以作业的长短，即 运行时间 来计算优先级。

缺点：

- 必须知道作业运行时间
- 长作业周期变长
- 无法实现人机交互
- 无法解决紧迫性作业

3.优先级调度算法(priority-scheduling algorithm，PSA)

基于作业的紧迫程度，由 外部赋予 作业相应的优先级。

4.高响应比优先调度算法(Highest Response Ratio Next，HRRN)

> *HRRN同时考虑了作业的等待时间和作业运行时间。*

引入一个动态优先级，令它随等待时间延长而增加，这将使长作业的优先级在等待期间不断地增加，等到足够的时间后，必然有机会获得处理机。该优先级的变化规律可描述为：

$$优先权=\frac{等待时间+要求服务时间}{要求服务时间}$$

由于等待时间与服务时间之和就是系统对该作业的响应时间，故该优先级又相当于响应比RP。据此，优先又可表示为：

$$R_P=\frac{等待时间+要求服务时间}{要求服务时间}=\frac{响应时间}{要求服务时间}$$

3.3 进程调度

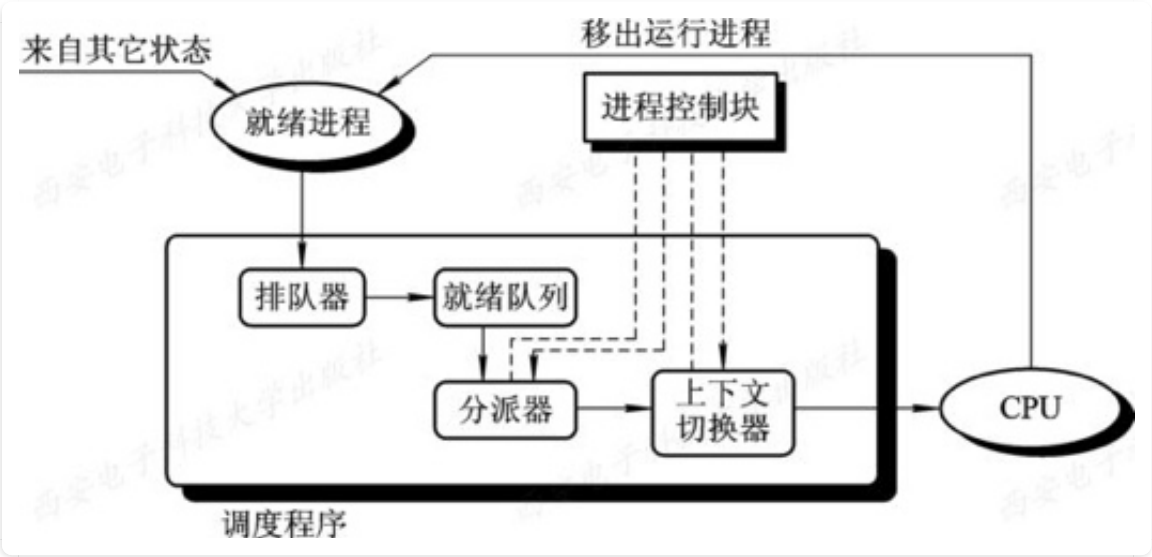
3.3.1 进程调度的任务、机制和方式

1.进程调度的任务

- 保存处理机的现场信息
- 按某种算法选取进程
- 把处理器分配给进程

2.进程调度机制

1. 排队器：将就绪进程按照一定策略排成一队或多队
2. 分派器
3. 上下文切换器



3.进程调度方式

（1）非抢占方式(Nonpreemptive Mode)

一旦把处理机分配给某进程后，就一直让它运行下去，决不会因为时钟中断或任何其它原因去抢占当前正在运行进程的处理机。

（2）抢占方式(Preemptive Mode)

> *这种调度方式允许调度程序根据某种原则，去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。在现代OS中广泛采用抢占方式，这是因为：对于批处理机系统，可以防止一个长进程长时间地占用处理机，以确保处理机能为所有进程提供更*

> *为公平的服务。在分时系统中，只有采用抢占方式才有可能实现人—机交互。在实时系统中，抢占方式能满足实时任务的需求。但抢占方式比较复杂，所需付出的系统开销也较大。*

抢占原则：

- 优先权原则
- 短进程优先原则
- 时间片原则

3.3.2 轮转调度算法（round robin，RR）

> *在分时系统，最简单也较为常用的是基于时间片的轮转调度算法，该算法采用公平的处理剂分配方式，即让就绪队列上每个进程每次仅运行一个时间片。*

1.轮转法的基本原理

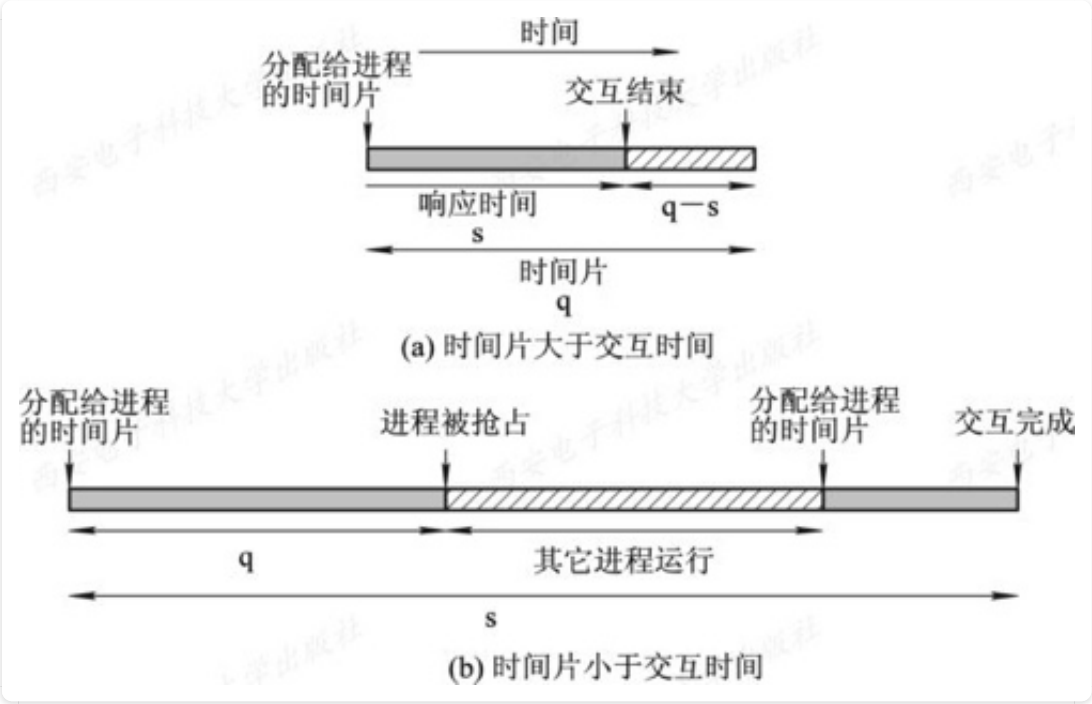
在轮转(RR)法中，系统将所有的就绪进程按FCFS策略排成一个就绪队列。系统可设置每隔一定时间(如30ms)便产生一次中断，去激活进程调度程序进行调度，把CPU分配给队首进程，并令其执行一个时间片。当它运行完毕后，又把处理机分配给就绪队列中新的队首进程，也让它执行一个时间片。这样，就可以保证就绪队列中的所有进程在确定的时间段内，都能获得一个时间片的处理机时间。

2.进程切换时机

1. 若一个时间片尚未用完，正在运行的进程便已经完成，就立即激活调度程序，将它从就绪队列中删除，再调度就绪队列中队首的进程运行，并启动一个新的时间片。
2. 在一个时间片用完时，计时器中断处理程序被激活。如果进程尚未运行完毕，调度程序将把它送往就绪队列的末尾。

3.时间片大小的确定

如图，其中图(a)是时间片略大于典型交互的时间，而图(b)是时间片小于典型交互的时间。



时间片分别为q = 1和q = 4时对平均周转时间的影响：

		A	B	C	D	E	
		0	1	2	3	4	
		4	3	4	2	4	
RR q=1		15	12	16	9	17	
		15	11	14	6	13	11.8
		3.75	3.67	3.5	3	3.33	3.46
RR q=4		4	7	11	13	17	
		4	6	9	10	13	8.4
		1	2	2.25	5	3.33	2.5

3.3.3 优先级调度算法

1.优先级调度算法的类型

优先级进程调度算法，是把处理机分配给就绪队列中优先级最高的进程。这时，又可进一步把该算法分成两种：

- 非抢占式优先级调度算法
- 抢占式优先级调度算法

2.优先级的类型

静态优先级

静态优先级是在创建进程时确定的，在进程的整个运行期间保持不变。优先级是利用某一范围内的一个整数来表示的，例如0～255中的某一整数，又把该整数称为优先数。确定进程优先级大小的依据有如下三个：

- 进程类型
- 进程对资源的需求
- 用户要求 #### 动态优先级 动态优先级是指在创建进程之初，先赋予其一个优先级，然后其值随进程的推进或等待时间的增加而改变，以便获得更好的调度性能。 ## 3.3.4 多队列调度算法 >如前所述的各种调度算法，尤其在应用于进程调度时，由于系统中仅设置一个进程的就绪队列，即低级调度算法是固定的、单一的，无法满足系统中不同用户对进程调度策略的不同要求，在多处理机系统中，这种单一调度策略实现机制的缺点更显突出，由此，多级队列调度算法能够在一定程度上弥补这一缺点。

3.3.5 多级反馈队列(multileved feedback queue)调度算法

1.调度机制

多级反馈队列调度算法的调度机制可描述如下：

(1) 设置多个就绪队列

(2) 每个队列都采用FCFS算法

当新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统。否则，即它在一个时间片结束时尚未完成，调度程序将其转入第二队列的末尾等待调度；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，依此类推。当进程最后被降到第n队列后，在第n队列中便采取按RR方式运行。

(3) 按队列优先级调度

调度程序首先调度最高优先级队列中的诸进程运行，仅当第一队列空闲时才调度第二队列中的进程运行；换言之，仅当第1～(i–1)所有队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时又有新进程

进入任一优先级较高的队列，此时须立即把正在运行的进程放回到第i队列的末尾，而把处理机分配给新到的高优先级进程。

2.调度算法的性能

在多级反馈队列调度算法中，如果规定第一个队列的时间片略大于多数人机交互所需之处理时间时，便能较好地满足各种类型用户的需要。

- 终端型用户
- 短批处理作业用户
- 长批处理作业用户

3.3.6 基于公平原则的调度算法

1.保证调度算法

> *保证调度算法是另外一种类型的调度算法，它向用户所做出的保证并不是优先运行，而是明确的性能保证，该算法可以做到调度的公平性。一种比较容易实现的性能保证是处理机分配的公平性。如果在系统中有n个相同类型的进程同时运行，为公平起见，须保证每个进程都获得相同的处理机时间1/n。*

在实施公平调度算法时系统中必须具有这样一些功能：

- 跟踪计算每个进程自创建以来已经执行的处理时间。
- 计算每个进程应获得的处理机时间，即自创建以来的时间除以n。
- 计算进程获得处理机时间的比率，即进程实际执行的处理时间和应获得的处理机时间之比。
- 比较各进程获得处理机时间的比率。如进程A的比率最低，为0.5，而进程B的比率为0.8，进程C的比率为1.2等。
- 调度程序应选择比率最小的进程将处理机分配给它，并让该进程一直运行，直到超过最接近它的进程比率为止。

2.公平分享调度算法

分配给每个进程相同的处理机时间，显然，这对诸进程而言，是体现了一定程度的公平，但如果各个用户所拥有的进程数不同，就会发生对用户的不公平问题。

3.4 实时调度

> *在实时系统中，可能存在着两类不同性质的实时任务，即**HRT任务**和**SRT任务**，它们都联系着一个**截止时间**。为保证系统能正常工作，实时调度必须能满足实时任务对截止时间的要求。*

3.4.1 实现实时调度的基本条件

1. 提供必要的信息

就绪时间

是指某任务成为就绪状态的起始时间，在周期任务的情况下，它是事先预知的一串时间序列。

开始截止时间和完成截止时间

对于典型的实时应用，只须知道开始截止时间，或者完成截止时间。

处理时间

一个任务从开始执行，直至完成时所需的时间。

资源要求

任务执行时所需的一组资源。

优先级

如果某任务的开始截止时间错过，势必引起故障，则应为该任务赋予“绝对”优先级；如果其开始截止时间的错过，对任务的继续运行无重大影响，则可为其赋予“相对”优先级，供调度程序参考。

2.系统处理能力强

在实时系统中，若处理机的处理能力不够强，则有可能因处理机忙不过，而致使某些实时任务不能得到及时处理，从而导致发生难以预料的后果。假定系统中有m个周期性的硬实时任务HRT，它们的处理时间可表示为Ci，周期时间表

示为Pi，则在单处理机情况下，必须满足下面的限制条件系统才是可调度的：

提高系统处理能力的途径有二：一是采用单处理机系统，但须增强其处理能力，以显著地减少对每一个任务的处理时间；二是采用多处理机系统。假定系统中的处理机数为N，则应将上述的限制条件改为：

3.采用抢占式调度机制

在含有HRT任务的实时系统中，广泛采用抢占机制。这样便可满足HRT任务对截止时间的要求。但这种调度机制比较复杂。

4.具有快速切换机制

为保证硬实时任务能及时运行，在系统中还应具有快速切换机制，使之能进行任务的快速切换。该机制应具有如下两方面的能力：

对中断的快速响应能力

对紧迫的外部事件请求中断能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短，以免耽误时机(其它紧迫任务)。

快速的任务分派能力

为了提高分派程序进行任务切换时的速度，应使系统中的每个运行功能单位适当的小，以减少任务切换的时间开销。

3.4.2 实时调度算法的分类

可以按不同方式对实时调度算法加以分类：
根据实时任务性质,可将实时调度的算法分为硬实时调度算法和软实时调度算法；

按调度方式,则可分为非抢占调度算法和抢占调度算法。

1.非抢占式调度算法

非抢占式轮转调度算法

非抢占式优先调度算法

2.抢占式调度算法

可根据抢占发生时间的不同而进一步分成以下两种调度算法：

- 基于时钟中断的抢占式优先级调度算法。
- 立即抢占(Immediate Preemption)的优先级调度算法。

3.4.3 最早截止时间优先EDF(Earliest Deadline First)算法

1.非抢占式调度方式用于非周期实时任务

2.抢占式调度方式用于周期实时任务

图3–7示出了将该算法用于抢占调度方式之例。在该例中有两个周期任务，任务A和任务B的周期时间分别为20 ms和50 ms，每个周期的处理时间分别为10 ms和25 ms。图3–7 最早截止时间优先算法用于抢占调度方式之例

3.4.4 最低松弛度优先LLF(Least Laxity First)算法

该算法在确定任务的优先级时，根据的是任务的紧急(或松弛)程度。任务紧急程度愈高，赋予该任务的优先级就愈高，以使之优先执行。

该算法主要用于可抢占调度方式中。假如在一个实时系统中有两个周期性实时任务A和B，任务A要求每20 ms执行一次，执行时间为10 ms，任务B要求每50 ms执行一次，执行时间为25 ms。由此可知，任务A和B每次必须完成的时间分别为：A1、A2、A3、…和B1、B2、B3、…，见图3–8。

利用ELLF算法进行调度的情况，具有两个周期性实时任务的调度情况：

3.4.5 优先级倒置(priority inversion problem)

1.优先级倒置的形成

当前OS广泛采用优先级调度算法和抢占方式，然而在系统中存在着影响进程运行的资源而可能产生“优先级倒置”的现象，即高优先级进程(或线程)被低优先级进程(或线程)延迟或阻塞。我们通过一个例子来说明该问题。

假如P3最先执行，在执行了P(mutex)操作后，进入到临界区CS–3。在时刻a，P2就绪，因为它比P3的优先级高，P2抢占了P3的处理机而运行，如图3–10所示。 优先级倒置示意图

2.优先级倒置的解决方法

一种简单的解决方法是规定：假如进程P3在进入临界区后P3所占用的处理机就不允许被抢占。
图3–11示出了采用动态优先级继承方法后，P1、P2、P3三个进程的运行情况。

3.5 死锁概述

3.5.1 资源问题

> *在系统中有许多不同类型的资源，其中可以引起死锁的主要是，需要采用互斥访问方法的、不可以被抢占的资源，即在前面介绍的临界资源。系统中这类资源有很多，如打印机、数据文件、队列、信号量等。*

1.可重用性资源和消耗性资源

可重用性资源

> *可重用性资源是一种可供用户重复使用多次的资源*

它具有如下性质：

- 每一个可重用性资源中的单元只能分配给一个进程使用，不允许多个进程共享。
- 进程在使用可重用性资源时，须按照这样的顺序：
    - 请求资源。如果请求资源失败，请求进程将会被阻塞或循环等待。
    - 使用资源。进程对资源进行操作，如用打印机进行打印；
    - 释放资源。当进程使用完后自己释放资源。
- 系统中每一类可重用性资源中的单元数目是相对固定的，进程在运行期间既不能创建也不能删除它。

可消耗性资源

可消耗性资源又称为临时性资源，它是在进程运行期间，由进程动态地创建和消耗的，它具有如下性质：

- 每一类可消耗性资源的单元数目在进程运行期间是可以不断变化的，有时它可以有许多，有时可能为0；
- 进程在运行过程中，可以不断地创造可消耗性资源的单元，将它们放入该资源类的缓冲区中，以增加该资源类的单元数目。
- 进程在运行过程中，可以请求若干个可消耗性资源单元，用于进程自己的消耗，不再将它们返回给该资源类中。 ### 2. 可抢占性资源和不可抢占性资源 #### 可抢占性资源 可把系统中的资源分成两类，一类是可抢占性资源，是指某进程在获得这类资源后，该资源可以再被其它进程或系统抢占。 #### 不可抢占性资源 另一类资源是不可抢占性资源，即一旦系统把某资源分配给该进程后，就不能将它强行收回，只能在进程用完后自行释放。

3.5.2 计算机系统中的死锁

1.竞争不可抢占性资源引起死锁

通常系统中所拥有的不可抢占性资源其数量不足以满足多个进程运行的需要，使得进程在运行过程中，会因争夺资源而陷入僵局。

我们可将上面的问题利用资源分配图进行描述，用方块代表可重用的资源(文件)，用圆圈代表进程，见图3–12所示共享文件时的死锁情况 。

2.竞争可消耗资源引起死锁

现在进一步介绍竞争可消耗资源所引起的死锁。图3-13示出了在三个进程之间，在利用消息通信机制进行通信时所形成的死锁情况。

3.进程推进顺序不当引起死锁

除了系统中多个进程对资源的竞争会引发死锁外，进程在运行过程中，对资源进行申请和释放的顺序是否合法，也是在系统中是否会产生死锁的一个重要因素。

进程推进顺序合法

在进程P1和P2并发执行时，如果按图3-14中的曲线①所示的顺序推进：P1：Request(R1)→P1：Request(R2)→P1：Releast(R1)→P1：Release(R2)→P2：Request(R2)→P2：Request(R1)→P2：Release(R2)→P2：Release(R1)，两个进程可顺利完成。类似地，若按图中曲线②和③所示的顺序推进，两进程也可以顺利完成。我们称这种不会引起进程死锁的推进顺序是合法的。

进程推进顺序非法

若并发进程P1和P2按图3-14中曲线④所示的顺序推进，它们将进入不安全区D内。此时P1保持了资源R1，P2保持了资源R2，系统处于不安全状态。此刻，如果两个进程继续向前推进，就可能发生死锁。例如，当P1运行到P1：Request(R2)时，将因R2已被P2占用而阻塞；当P2运行到P2：Request(R1)时，也将因R1已被P1占用而阻塞，于是发生了进程死锁，这样的进程推进顺序就是非法的。

3.5.3 死锁的定义、必要条件

1.死锁的定义

> *在一组进程发生死锁的情况下，这组死锁进程中的每一个进程，都*

> *在等待另一个死锁进程所占有的资源。*
>
> ***2.产生死锁的必要条件***
>
> *虽然进程在运行过程中可能会发生死锁，但产生进程死锁是必须具备一定条件的。综上所述不难看出，产生死锁必须同时具备下面四个必要条件，只要其中任一个条件不成立，死锁就不会发生：*

- 互斥条件
- 请求和保持条件
- 不可抢占条件
- 循环等待条件 ## 3.6 处理死锁的方法 ## 3.6.1 预防死锁 >预防死锁的方法是通过破坏产生死锁的四个必要条件中的一个或几个，以避免发生死锁。由于互斥条件是非共享设备所必须的，不仅不能改变，还应加以保证，因此主要是破坏产生死锁的后三个条件。

1.破坏“请求和保持”条件

为了能破坏“请求和保持”条件，系统必须保证做到：当一个进程在请求资源时，它不能持有不可抢占资源。该保证可通过如下两个不同的协议实现：

1. 第一种协议

该协议规定，所有进程在开始运行之前，必须一次性地申请其在整个运行过程中所需的全部资源。

2. 第二种协议

该协议是对第一种协议的改进，它允许一个进程只获得运行初期所需的资源后，便开始运行。

2.破坏“不可抢占”条件

为了能破坏“不可抢占”条件，协议中规定，当一个已经保持了某些不可被抢占资源的进程，提出新的资源请求而不能得到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请。这意味着进程已占有的资源会被暂时地释放，或者说是被抢占了，从而破坏了“不可抢占”条件。

3.破坏“循环等待”条件

一个能保证“循环等待”条件不成立的方法是，对系统所有资源类型进行线性排序，并赋予不同的序号。

3.6.2 避免死锁

> *避免死锁同样是属于事先预防的策略，但并不是事先采取某种限制措施，破坏产生死锁的必要条件，而是在资源动态分配过程中，防止系统进入不安全状态，以避免发生死锁。这种方法所施加的限制条件较弱，可能获得较好的系统性能，目前常用此方法来避免发生死锁。*

1.系统安全状态

在死锁避免方法中，把系统的状态分为安全状态和不安全状态。当系统处于安全状态时，可避免发生死锁。反之，当系统处于不安全状态时，则可能进入到死锁状态。

1. 安全状态

在该方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。

. 安全状态之例

假定系统中有三个进程P1、P2和P3，共有12台磁带机。进程P1总共要求10台磁带机，P2和P3分别要求4台和9台。假设在T0时刻，进程P1、P2和P3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

3. 由安全状态向不安全状态的转换

如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。

3.7.2 利用银行家算法避免死锁

最有代表性的避免死锁的算法是Dijkstra的银行家算法。起这样的名字是由于该算法原本是为银行系统设计的，以确保银行在发放现金贷款时，不会发生不能满足所有客户需要的情况。在OS中也可用它来实现避免死锁。

1. 银行家算法中的数据结构

为了实现银行家算法，在系统中必须设置这样四个数据结构，分别用来描述系统中可利用的资源、所有进程对资源的最大需求、系统中的资源分配，以及所有进程还需要多少资源的情况。

(1) 可利用资源向量Available。

(2) 最大需求矩阵Max。

(3) 分配矩阵Allocation。

(4) 需求矩阵Need。

2. 银行家算法

设Requesti是进程Pi的请求向量，如果Request i[j]=K，表示进程Pi需要K

个Rj类型的资源。当Pi发出资源请求后，系统按下述步骤进行检查：

(1) 如果Request i[j]≤Need[i, j]，便转向步骤(2)； 否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

(2) 如果Request i[j]≤Available[j]，便转向步骤(3)； 否则，表示尚无足够资源，Pi须等待。

(3) 系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：

Available[j] = Available[j] – Request i[j];

Allocation[i, j] = Allocation[i, j] + Request i[j];

Need[i, j] = Need[i, j] – Request i[j];

(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。若安全，才正式将资源分配给进程Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程Pi等待。

3. 安全性算法

系统所执行的安全性算法可描述如下：

(1) 设置两个向量：① 工作向量Work，它表示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，Work := Available；② Finish：它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做Finish[i] := false；当有足够资源分配给进程时，再令Finish[i] := true。

(2) 从进程集合中找到一个能满足下述条件的进程：

① Finish[i]=false;

② Need[i, j]≤Work[j];

若找到，执行步骤(3)，否则，执行步骤(4)。

(3) 当进程Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

```
Work[j] = Work[j]+Allocation[i, j];
Finish[i] =true;
go to step 2;
```

(4) 如果所有进程的Finish[i]=true都满足，则表示系统处于安全状态；否则，系统处于不安全状态。

3.6.3 检测死锁与解除死锁

如果在系统中，既不采取死锁预防措施，也未配有死锁避免算法，系统很可能会发生死锁。在这种情况下，系统应当提供两个算法：

① 死锁检测算法。该方法用于检测系统状态，以确定系统中是否发生了死锁。

② 死锁解除算法。当认定系统中已发生了死锁，利用该算法可将系统从死锁状态中解脱出来。

3.8.1 死锁的检测

为了能对系统中是否已发生了死锁进行检测，在系统中必须：① 保存有关资源的请求和分配信息；② 提供一种算法，它利用这些信息来检测系统是否已进入死锁状态。

1. 资源分配图(Resource Allocation Graph)

系统死锁，可利用资源分配图来描述

该图是由一组结点N和一组边E所组成的一个对偶G = (N, E)，它具有下述形式的定义和限制：

(1) 把N分为两个互斥的子集，即一组进程结点P={P1, P2, …, Pn}和一组资源结点R={R1, R2, …, Rn}，N = P∪R。在图3–19所示的例子中，P = {P1, P2}，R = {R1, R2}，N = {R1, R2}∪{P1, P2}。

(2) 凡属于E中的一个边e∈E，都连接着P中的一个结点和R中的一个结点，e = {Pi, Rj}是资源请求边，由进程Pi指向资源Rj，它表示进程Pi请求一个单位的Rj资源。E = {Rj, Pi}是资源分配边，由资源Rj指向进程Pi，它表示把一个单位的资源Rj分配给进程Pi。图3–19中示出了两个请求边和两个分配边，即E = {(P1, R2), (R2, P2), (P2, R1), (R1, P1)}。

2．死锁定理

我们可以利用把资源分配图加以简化的方法(图3–19)，来检测当系统处于S状态时，是否为死锁状态。简化方法如下：

(1) 在资源分配图中，找出一个既不阻塞又非独立的进程结点Pi。在顺利的情况下，Pi可获得所需资源而继续运行，直至运行完毕，再释放其所占有的全部资源，这相当于消去Pi的请求边和分配边，使之成为孤立的结点。在图3–20(a)中，将P1的两个分配边和一个请求边消去，便形成图(b)所示的情况。

(2) P1释放资源后，便可使P2获得资源而继续运行，直至P2完成后又释放出它所占有的全部资源，形成图(c)所示的情况，即将P2的两条请求边和一条分配边消去。

(3) 在进行一系列的简化后，若能消去图中所有的边，使所有的进程结点都成为孤立结点，则称该图是可完全简化的；若不能通过任何过程使该图完全简化，则称该图是不可完全简化的。

3．死锁检测中的数据结构

死锁检测中的数据结构类似于银行家算法中的数据结构：

(1) 可利用资源向量Available，它表示了m类资源中每一类资源的可用数目。

(2) 把不占用资源的进程(向量Allocation=0)记入L表中，即Li∪L。

(3) 从进程集合中找到一个Requesti≤Work的进程，做如下处理：① 将其资源分配图简化，释放出资源，增加工作向量Work =Work + Allocation i。② 将它记入L表中。

(4) 若不能把所有进程都记入L表中，便表明系统状态S的资源分配图是不可完全简化的。因此，该系统状态将发生死锁。

3.8.2 死锁的解除

1. 终止进程的方法

1) 终止所有死锁进程

这是一种最简单的方法，即是终止所有的死锁进程，死锁自然也就解除了，但所付出的代价可能会很大。因为其中有些进程可能已经运行了很长时间，已接近结束，一旦被终止真可谓“功亏一篑”，以后还得从头再来。还可能会有其它方面的代价，在此不再一一列举。

2) 逐个终止进程

稍微温和的方法是，按照某种顺序，逐个地终止进程，直至有足够的资源，以打破循环等待，把系统从死锁状态解脱出来为止。但该方法所付出的代价也可能很大。因为每终止一个进程，都需要用死锁检测算法确定系统死锁是否已经被解除，若未解除还需再终止另一个进程。另外，在采取逐个终止进程策略时，还涉及到应采用什么策略选择一个要终止的进程。选择策略最主要的依据是，为死锁解除所付出的“代价最小”。但怎么样才算是“代价最小”，很难有一个精确的度量。

2. 付出代价最小的死锁解除算法

一种付出代价最小的死锁解除算法如图3–21所示。