第三章
中断与处理器调度

中断是与处理器管理密切相关的一个重要概念。确切地说，中断是实现多道程序设计的必要条件。如果没有中断，操作系统就无法获得系统的控制权，也就不能将处理器资源分配给不同的进程。

系统启动后，操作系统处于一种“待命”状态。等待什么命令呢？这就是中断。中断的产生原因有很多，除设备、时钟等外部因素外，程序错误、缺页、访管等都有可能导致中断发生。在有些书和系统中还使用“异常”（exception）、“自陷”（trap）等术语，在本书中统一使用“中断”来表示一切导致控制随机发生转移的事件。中断发生后，进入操作系统中相应的中断处理程序。该处理程序执行过程中，如果时机恰当，系统会考虑切换进程，即让现行进程下去（下降进程），另外一个进程上去（上升进程），该工作由处理器调度程序完成。可见，操作系统是中断驱动的。

本书强调中断与处理器调度之间的密切关系，即中断是处理器切换的前提条件。将二者结合起来讲述，有助于读者深刻理解操作系统的运作机理。

3.1 中断与中断系统


中断与中断系统

3.1.1 中断概念

定义 3-1 在程序运行过程中出现某种紧急事件，必须中止当前正在运行的程序，转去处理此事件，然后再恢复原来运行的程序，这个过程称为中断。

中断的实现需要硬件和软件的合作，硬件部分称为中断装置，软件部分称为中断处理程序。中断装置和中断处理程序统称为中断系统。

3.1.2 中断装置

中断装置是中断系统中的硬件部分，它的职能是发现并响应中断，具体包括以下几个步骤。

① 识别中断源。当有多个中断源同时存在时，由中断装置选择优先级别最高的中断源。

② 保存现场。将正在运行的进程的程序状态字及指令计数器中的内容压入系统栈。

③ 引出中断处理程序。将与中断事件相对应的中断向量由内存指定单元处取出并送入程序状态字及指令计数器中，如此便转入对应的中断处理程序。中断的响应过程如图 3-1 所示。

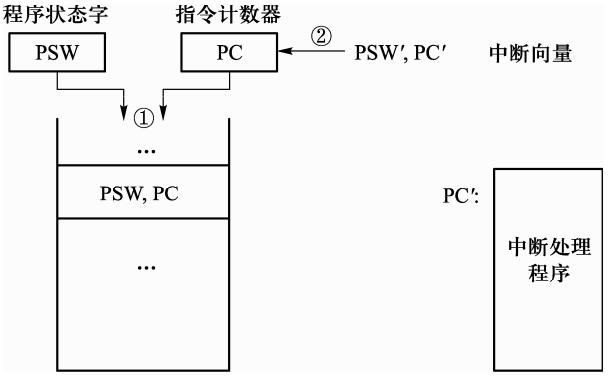

图 3-1 中断的响应过程

1. 中断源与中断字

引起中断的事件称为中断源。例如，外围设备执行完输入输出操作、程序执行非法指令、用户在终端上输入一条命令等，这些都是可能引起处理器中断的事件，因而都是中断源。

中断装置发现并响应中断后，转到相应的中断处理程序。该程序在处理中断时通常需要进一步了解与中断事件相关的一些信息，例如对于设备中断，需要知道设备传输情况。为此，硬件系统为每一个中断源设置一个寄存器，当中断发生时，硬件系统将与中断相关的详细信息存入该寄存器中，以便中断处理程序从中进一步分析触发中断的原因并采取相应的处理措施。用于保存与中断事件相关信息的寄存器称为中断寄存器。中断寄存器中的内容称为中断字。

2. 中断类型与中断向量

中断可以分为两大类，即强迫性中断和自愿性中断。下面分别加以介绍。

（1）强迫性中断

这类中断事件是正在运行的程序所不期望的。强迫性中断事件是否发生，何时发生，事先无法预知，因而运行程序可能在任意位置被打断。这类中断大致有以下几种。

① 时钟中断：如硬件实时时钟到时等。

② 输入输出中断：如设备出错、传输结束等。

③ 控制台中断：如系统操作员通过控制台发出命令等。

④ 硬件故障中断：如掉电、内存校验错误等。

⑤ 程序错误中断：如目态程序执行特权指令、地址越界、虚拟存储中的缺页故障或缺段故障、溢出、除数为 0 等。

（2）自愿性中断

这类中断事件是正在运行的程序有意识安排的。通常由于正在运行的程序执行访管指令而

引起自愿性中断事件，其目的是要求系统提供某种服务。这类中断的发生具有必然性，而且发生位置确定。

两种类型的中断事件与中断装置之间的关系如图 3-2 所示，其中图 3-2（a）为强迫性中断事件，图 3-2（b）为自愿性中断事件。

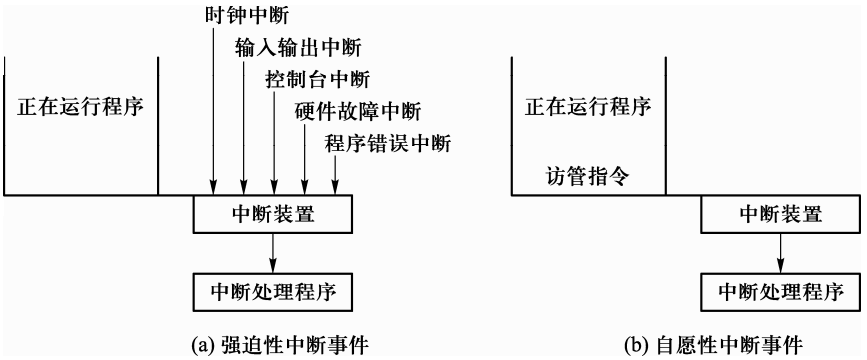

图 3-2 中断事件与中断装置之间的关系

因为对同类事件的处理方法是相同的，所以不是每个中断源都有一个中断处理程序，而是每类中断事件有一个中断处理程序。例如，系统中有 4 个类型相同的通道，每个通道都是一个中断源，但是由于它们的中断处理方法相同，因而可以共用同一个中断处理程序。

每个中断处理程序有一个入口地址（PC）及其运行环境（PSW），它们存放在内存中的固定单元处。当中断事件发生时，中断装置根据中断类别自动地将对应的 PSW 和 PC 送入程序状态字和指令计数器中，如此便转移到对应的中断处理程序。这种转移类似于向量转移，因而 PSW 和 PC 被称为中断向量，如图 3-3 所示。

操作系统的设计者根据各个中断处理程序的存储位置及运行环境来填写对应的中断向量。

3．中断嵌套与系统栈


中断嵌套与处理过程

如果系统在处理一个中断事件的过程中又响应了新的中断，则称发生中断嵌套。从理论上说，对于中断嵌套的层数是没有限制的。也就是说，允许任意多层的中断嵌套。不过，一般来说，在一个中断事件的处理过程中，只允许响应更紧迫的中断事件，也就是只允许优先级别更高的中断事件打断它，而硬件中断优先级别的个数通常是有限的，所以中断嵌套的实际层数大多不会超过中断优先级别的个数。图 3-4 给出中断嵌套的一般情形。

当中断发生时，需要保存被中断程序的现场信息，中断返回时需要恢复被中断程序的现场。对于嵌套中断来说，现场恢复的次序与现场保存的次序正好相反，故应当采用栈作为中断现场保存区域。因为操作系统需要访问这个栈，所以它应当被设在系统空间中，而且由于中断发生时，被中断程序的程序状态字和指令计数器的值是由硬件中断装置压入系统栈中的，因而系统栈区的位置实际上是由硬件确定的。对于图 3-4 所示的中断嵌套来说，系统栈区的内容如图 3-5

所示。

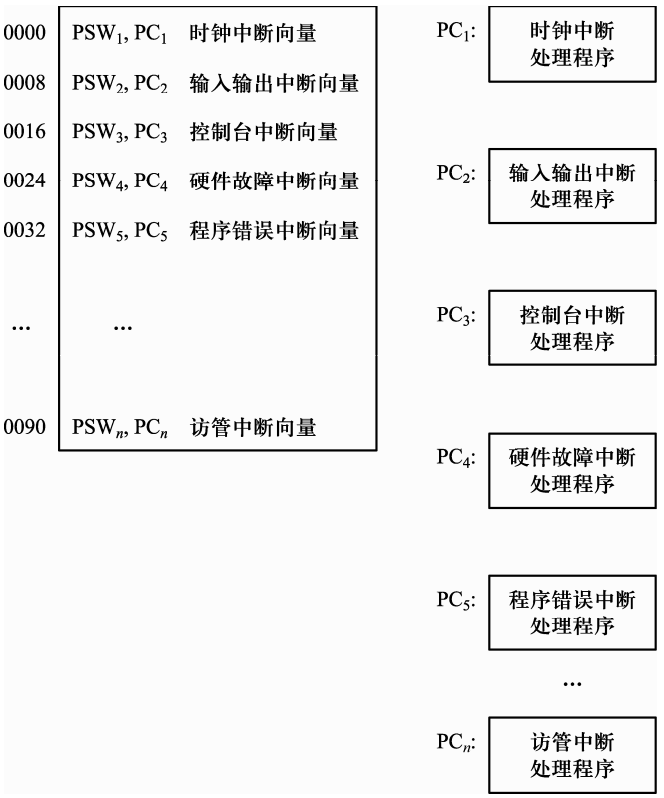

图 3-3 中断向量与中断处理程序

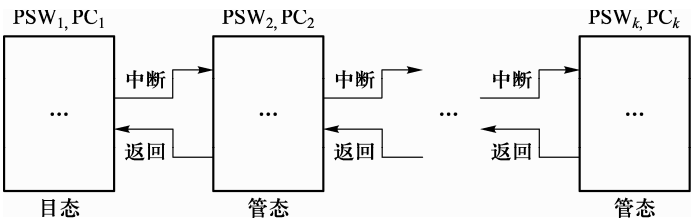

图 3-4 中断嵌套

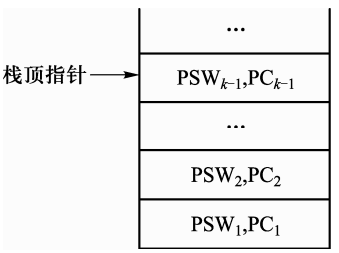

图 3-5 系统栈区

应当指出，系统栈区中所保存的内容除包括程序状态字和指令计数器的值之外，还有其他现场信息，如通用寄存器的内容等，这些信息是中断被响应后由中断处理程序根据需要而保存的。因而，中断装置实际上只保存最基本的现场信息。

顺便指出，系统栈还有另一个重要用途，即传送操作系统子程序之间相互调用的参数、返

回值以及返回地址，这一点与用户程序使用用户栈的情形相仿。由此可以看出，中断相当于一种特殊的子程序调用，只不过它的发生时刻具有不确定性。

4．中断优先级别与中断屏蔽

根据引起中断事件的重要性和紧迫程度，硬件将中断源分为若干个级别，称为中断优先级别。如果有多个中断同时发生，硬件系统将首先响应优先级别最高的中断请求。对于相同优先级别的中断，硬件系统将按照事先规定好的次序依次响应。

在中断事件的处理过程中可能会发生新的中断，这就是中断嵌套。如前所述，中断嵌套是必要的。但是，如果对此不加以控制，低优先级别的中断源可能打扰高优先级别的中断事件的处理过程，甚至可能会使中断嵌套层数无限制地增长，直至系统栈溢出。为此，硬件系统提供了中断屏蔽指令，利用中断屏蔽指令可以暂时禁止中断源向处理器发送中断请求。当然，在需要的时候还可以利用硬件指令解除被屏蔽的中断源。

通常，在一个中断事件的处理过程中，程序屏蔽包括该级别在内的所有低优先级别的中断，但是允许更高优先级别的中断中途插入。这样，当发生中断嵌套时，嵌套中断事件的优先级别是按照响应的顺序依次递增的。

中断优先级别是由硬件规定的，因而不可改变。不过程序可以根据需要适当地调整中断事件的响应次序。例如，想要优先响应级别为 5 的中断事件，可以将优先级别高于 5 的中断源暂时屏蔽掉。

在操作系统的核心中通常有这样的程序段，它们的执行不允许任何中断事件打扰，此时应当屏蔽所有的中断源。屏蔽所有的中断源相当于关中断，处于关中断状态下执行的程序段应当尽量简短，否则可能会丢失信息，也会影响系统的并发性。

3.1.3 中断处理逻辑

中断装置响应中断请求后，通过交换中断向量引入中断处理程序。中断处理程序在处理中断的过程中可能需要使用通用寄存器，为此需要进一步保存现场信息，中断处理结束后再恢复现场。这些现场信息保存在系统栈中，在保存现场和恢复现场的过程中不响应新的中断，为此保存现场（恢复现场）之前需要关中断，保存现场（恢复现场）之后再开中断。在中断处理过程中，中断处理程序需要根据中断码（访管号）进一步分析中断源，然后进行相应的处理，最后根据情况决定是否需要切换进程。中断处理过程如图 3-6 所示。

在图 3-6 中，到达“中断处理完”条件判断的虚箭头，表示下降进程再次被调度选中运行时的处理过程，而不是当前新切换的上升进程。

由图 3-6 可以看到，等待可能会发生多次，例如一个进程执行读取文件系统调用 read(fd,15k,buf)，“15k”表示要读取的数据量，我们知道文件是保存在磁盘上的，磁盘以块为单位进行分配和读写，假定块的大小为 4 KB，15 KB 涉及 4 个盘块的 I/O 传输，由于 I/O 传输的速度比较慢，每次进程启动磁盘后都会进入等待状态，等待 I/O 传输结束，I/O 结束时产生中断信号将其唤醒，进程获得 CPU 后将块内所需内容复制到进程空间由 buf 确定的位置，这样在不考虑缓存效果的前提下，进程可能会等待 4 次。

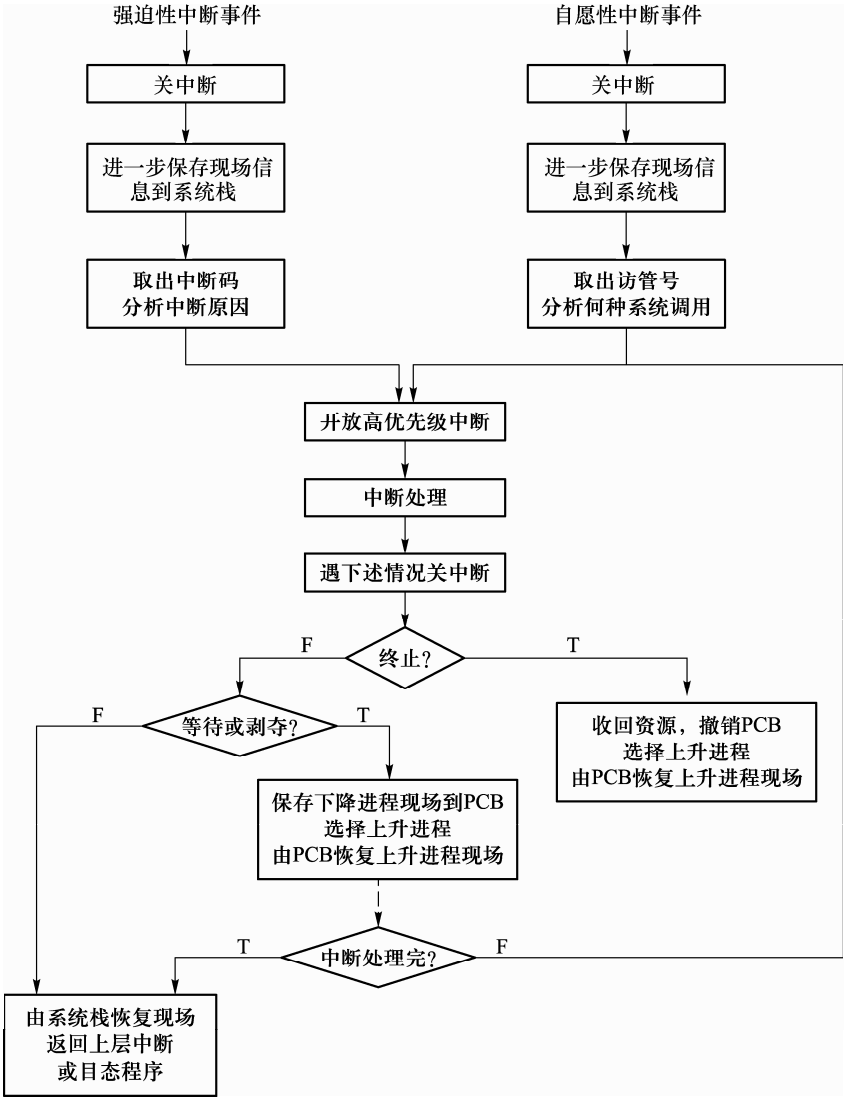

图 3-6 中断处理过程

剥夺也可能发生多次，假设一个进程进入管态需要运行 100 ms，处理器分派时间片的长度为 20 ms，则可能发生 10 次剥夺。当然，剥夺与等待也可能交替发生，即进程由目态进入管态后，在执行操作系统代码的过程中既有等待也有剥夺。

需要注意等待或剥夺发生的时刻是系统处于核心态，中断可能有嵌套，也可能没有嵌套。

3.1.4 现场级别与保存位置

需要注意区分两种“现场”，即“中断现场”与“进程切换现场”，中断现场保存在系统栈

中，进程切换现场保存在进程 PCB 中。注意等待和剥夺都是由操作系统完成的，因而等待和剥夺均发生在核心态，因而保存在 PCB 中的现场都是核心级别的现场，而并非目态程序的现场。

3.1.5 嵌套中断与系统栈

目态程序运行时，其对应的系统栈是空的；发生中断时，栈底是目态现场（包括 PSW、PC、中断处理程序使用的寄存器），然后是中断处理程序的嵌套函数调用的返回点、参数、局部变量、返回值；如果中断有嵌套，接下来是核心级别现场，然后是嵌套函数调用的返回点、参数、局部变量、返回值，嵌套可能是多重的。

3.1.6 进程状态转换的分解图


进程状态转换分解图

考虑系统状态与中断，可以给出更加完整的进程状态转换关系，如图 3-7 所示。目态运行的用户程序 *P* 发生中断时进入该进程的核心态，嵌套中断发生时仍处于该进程的核心态，嵌套中断返回时仍处于该进程的核心态，非嵌套中断返回时回到目态 *P′*。若 *P*=*P′*则说明中断未导致进程切换，若 *P*≠*P′*则说明发生了进程切换。

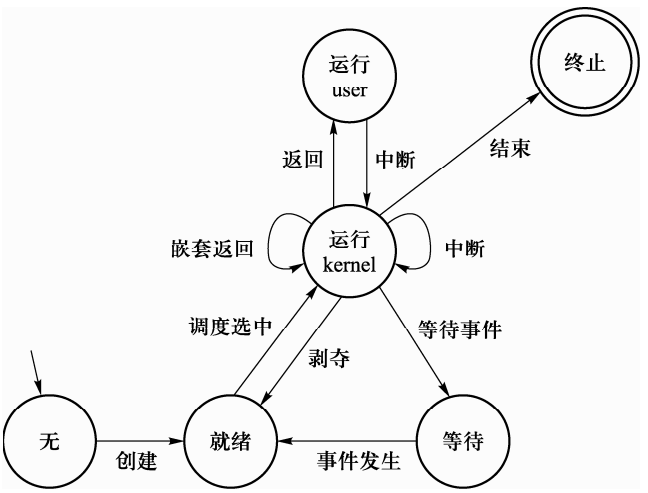

图 3-7 考虑系统状态的进程状态转换关系

表 3-1 给出了进程状态转换与操作系统动作之间的对应关系，可以看作是进程状态转换的分解图，详尽阐明了操作系统并发的实现机理。

表 3-1 进程状态转换的分解动作

事件	状态转换	操作系统动作
创建进程	无→就绪	建立 PCB，分配内存等必要的资源，初始化 PCB（地址映射寄存器，PSW 为目态，PC=0，管态 SP=栈底，目态 SP=栈底，通用寄存器 regs=0，浮点寄存器 fregs=0）

续表

事件	状态转换	操作系统动作
首次被调度选中	就绪→核心运行	PCB 切换，由 PCB 恢复地址映射寄存器、通用寄存器、浮点寄存器、目态 SP、管态 SP
首次返回到目态	核心运行→目态运行	恢复 PSW 和 PC
目态发生中断	目态运行→核心运行	目态现场压入系统栈
等待某事件发生	核心运行→等待	保存核心级别现场到 PCB（地址映射寄存器，regs，fregs，SP，PSW，PC），PCB 入相应的等待队列
所等待事件发生	等待→就绪	PCB 出等待队列，入就绪队列
再次被调度选中	就绪→核心运行	由 PCB 恢复核心级别现场（地址映射寄存器，regs，fregs，SP，PSW，PC）
非嵌套中断返回	核心运行→目态运行	由系统栈恢复现场，先按保存相反次序恢复中断处理程序用到的寄存器，最后恢复 PSW 和 PC
目态运行发生中断	目态运行→核心运行	目态级别现场压入系统栈
发生嵌套中断	核心运行→核心运行	核心级别现场压入系统栈
嵌套中断返回	核心运行→核心运行	由系统栈恢复现场，先按保存相反次序恢复中断处理程序用到的寄存器，最后恢复 PSW 和 PC
剥夺处理器	核心运行→就绪	保存核心级别现场到 PCB（地址映射寄存器，regs，fregs，SP，PSW，PC），PCB 入就绪队列
进程结束	核心运行→终止	收回占用内存等资源，撤销 PCB

对于非嵌套中断返回，即由“核心运行”转变为“目态运行”，需将 PSW 和 PC 由系统栈中弹出，送入程序状态字和指令计数器中，这两件事必须由一条指令完成，其原因留给读者思考。本小节的内容很重要，它揭示了操作系统运行的内部机理，建议读者结合图 3-6，图 3-7 和表 3-1，深刻理解本小节内容。


中断处理

由以上介绍可知，计算系统在运行时，时而运行目态程序，时而运行操作系统程序，运行用户程序时工作在用户态，运行操作系统程序时工作在系统态，这种工作模式称为“双态操作”（dual-mode operation）。

3.1.7 中断处理例程

1. 输入输出中断处理

输入输出中断一般由通道发出，分为以下两种情况。

（1）输入输出操作正常结束

如果需要继续传输数据，准备好数据后即可启动通道。如果发出输入输出操作请求的进程正在等待数据传输完成，则应将其唤醒。

（2）输入输出传输错误

通常需要进行输入输出复执。如果输入输出错误是由于外部干扰等因素而引起的，通过一两次复执便可成功；如果输入输出错误是由于设备故障而引起的，则任意多次复执也无效，因而可规定一个复执次数的上界，如 3 次复执不成功便认为是设备存在故障，通知系统操作员。

2．时钟中断处理

（1）时钟类型

时钟可分为硬件时钟和软件时钟两类。

① 硬件时钟。如前所述，硬件时钟又分为两种，即绝对时钟与间隔时钟。绝对时钟不发生中断，间隔时钟定时地（如每隔 20 ms）产生中断信号。

② 软件时钟。软件时钟是利用硬件定时机构和程序来实现的，读者可以设想其实现方法。软件时钟也是不发生中断的。

（2）时钟中断处理

时钟中断被触发时，中断处理程序要做许多与系统管理和维护有关的工作，主要包括以下内容。

① 进程管理。在采用时间片轮转处理器调度算法的系统中，记录进程已经占有处理器时间并判断时间片是否用完。在采用可抢占 CPU 动态优先数处理器调度算法的系统中，重新计算各个进程的优先数并判断是否有高优先级别的进程出现。

② 作业管理。记录作业在输入井中等待的时间，以及当前的优先级别，以便作业调度程序据此决定下一个将要进入系统执行的作业。

③ 资源管理。动态统计运行进程占有和使用处理器等资源的时间等。

④ 事件处理。在实时系统中，定时向被控制对象发送控制信号等。

⑤ 系统维护。定时运行死锁检测程序、定时运行系统记账程序等。

⑥ 实现软件时钟。利用硬件间隔时钟和一个存储单元可以实现软件时钟。例如，假设硬件间隔时钟每隔 10 ms 产生一次中断，某一程序每隔 1 000 ms 执行一次，则可以这样确定该程序的执行时刻：初始时，将某内存单元 timer 的值置为 100，以后每发生一次时钟中断就将 timer 的值减 1。当 timer 的值为 0 时，执行该程序，同时将 timer 的值再置为 100，为下一次执行该程序定时。

3．控制台中断处理

系统操作员可以利用控制台向系统发出中断请求。当按下控制台上的某一个按键后，就会产生一个中断信号，该中断信号相当于一条操作命令，操作系统将转到相应的中断处理程序，其中断请求内容以及处理方法与具体系统有关。

4．硬件故障中断处理

由于这类故障是由硬件本身引起的，排除这类故障必须有人工干预。中断处理程序所能做的工作是保存现场信息，并向系统操作员报告故障以便维修和校正，在故障消除时估计故障所造成的破坏，进而对系统进行恢复。

（1）电源故障处理

当发生电源故障时，如掉电，硬件设备能够保证继续工作一段时间，操作系统可以利用这段时间做以下 3 项工作。

① 将寄存器内容和内存信息写至外存储器。

② 停止外围设备工作。

③ 停止处理器工作。

当故障排除后，可进行系统恢复，主要工作如下。

① 启动处理器执行恢复程序。

② 启动外围设备工作。

③ 将断电时保存到外存储器中的信息取回到对应的寄存器和内存中。

当然，上述处理掉电中断的方法是不完美的，其间必然会丢失某些信息。理想的解决方案是预防，即保证不发生电源故障。目前由于不间断电源（uninterruptible power supply，UPS）的出现，这件事已经成为可能。不间断电源同时具有蓄电和稳压两种作用，它与计算机系统的连接方式如图 3-8 所示。当市电发生故障时，不间断电源可以继续供电足够长的时间，如 20 分钟、1 小时等。利用这段时间，操作员可以在做完必要的处理后停止机器运行。

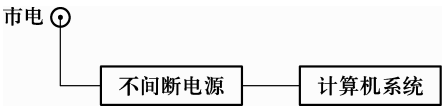

图 3-8 不间断电源

（2）主存故障处理

这是由于内存校验线路发现奇偶校验错误或海明校验错误而引起的中断。中断处理程序可以对错误单元进行检测并在确认错误后将其所在区域永久性地划分为不可用区域。

5．程序错误中断处理

一般来说，如果一个中断事件只影响正在运行的进程自身，不影响其他进程和操作系统，则既可由操作系统处理，也可由用户自行处理；如果一个中断事件可能影响其他进程或操作系统，则只能由操作系统处理，因而根据程序错误中断的性质可以有两种处理策略。

（1）只能由系统处理的中断

只能由系统处理的中断有内存访问地址越界、执行非法特权指令、缺页故障、缺段故障等。这类中断完全由操作系统统一处理。如果中断是由于程序错误而引起的，通常向系统操作员汇报出错的进程号、出错位置和错误性质等，并要求系统操作员干预；如果中断是由于缺页故障或缺段故障而引起的，则要将所需页或段动态地调入内存。

（2）可以由用户处理的中断

可以由用户处理的中断有浮点数溢出、阶码下溢、除数为 0 等。不同的用户对这类错误可能有不同的处理方法，因而可以由各个用户自行处理。如果用户程序没有提供特定的处理程序，将由操作系统按照标准处理方法加以处理。

下面重点讲述用户如何编制中断处理程序，以及中断发生后系统是如何转到用户自编中断处理程序的。

在某些高级语言中，通常提供一种调试语句，其形式如下：

on <中断条件> goto <中断续元入口>

将用户自编的中断处理程序称为中断续元，将中断续元的入口地址称为中断续元入口。例如：

on divide_zero goto LA;

表示当发生除数为 0 的中断事件时，转向标号为 LA 的语句处。LA 是中断续元入口，即：

LA：除数为 0 中断续元

对于发生在不同位置的同种中断事件，用户可以给出不同的处理方法。例如，若在执行上述调试语句后又执行如下的调试语句。

on divide_zero goto LB;

则当以后发生除数为 0 的中断后将转到 LB 去处理。

运行前，编译系统为每个用户程序生成一个中断续元入口表，如图 3-9 所示。该表的长度为系统允许用户处理的中断的个数。每个中断事件与哪个表目相对应是事先规定好的，但是表的内容初始时只有中断续元运行环境这一项，且均标明为目态，中断续元入口均为 0。当执行到调试语句时，与中断事件相对应的中断续元入口才被填写到对应的表目中。

当发生可由用户自行处理的中断事件时，操作系统根据中断事件查找中断续元入口表。如果其值为 0，表明用户未规定中断处理方法，由操作系统按照标准方法进行处理；如果其值不为 0，则由用户规定中断处理方法，应当根据查到的中断续元入口转到用户自编的中断处理程序。

中断事件1:	中断续元运行环境1	中断续元入口1
中断事件2:	中断续元运行环境2	中断续元入口2
…	…	…
中断事件*m*:	中断续元运行环境*m*	中断续元入口*m*

图 3-9 中断续元入口表

这里有两个重要的细节问题：一是如何转到中断续元，二是中断续元执行完毕如何返回被中断的用户程序。现在分别叙述如下。

（1）转到中断续元

将中断续元运行环境和中断续元入口一并送入硬件寄存器 PSW 和 PC，至此操作系统已经完成中断处理，中断续元的执行如同执行一个普通的用户子程序。

（2）返回被中断的用户程序

在中断发生时，被中断程序的现场信息已经被压入系统栈中，而中断续元运行于目态，它执行完毕将由用户栈区中恢复现场。为此，操作系统在转到中断续元之前还应当做一件事，即将系统栈中的现场信息弹出并压入用户栈中。

图 3-10 给出用户自行处理中断的过程，对步骤 ①～⑩解释如下。

① 目态程序在运行时发生溢出中断。

② 硬件寄存器 PSW 和 PC 的值压入系统栈。

③ 与中断源对应的中断向量被送入寄存器 PSW 和 PC。

④ 执行对应的操作系统中断处理程序。

⑤ 访问用户程序中的中断续元表，看用户是否要求自行处理中断，假定是。

⑥ 系统栈中的现场信息弹出并压入用户栈。

⑦ 中断续元运行环境及入口被送入寄存器 PSW 和 PC。

⑧ 控制权转到中断续元。

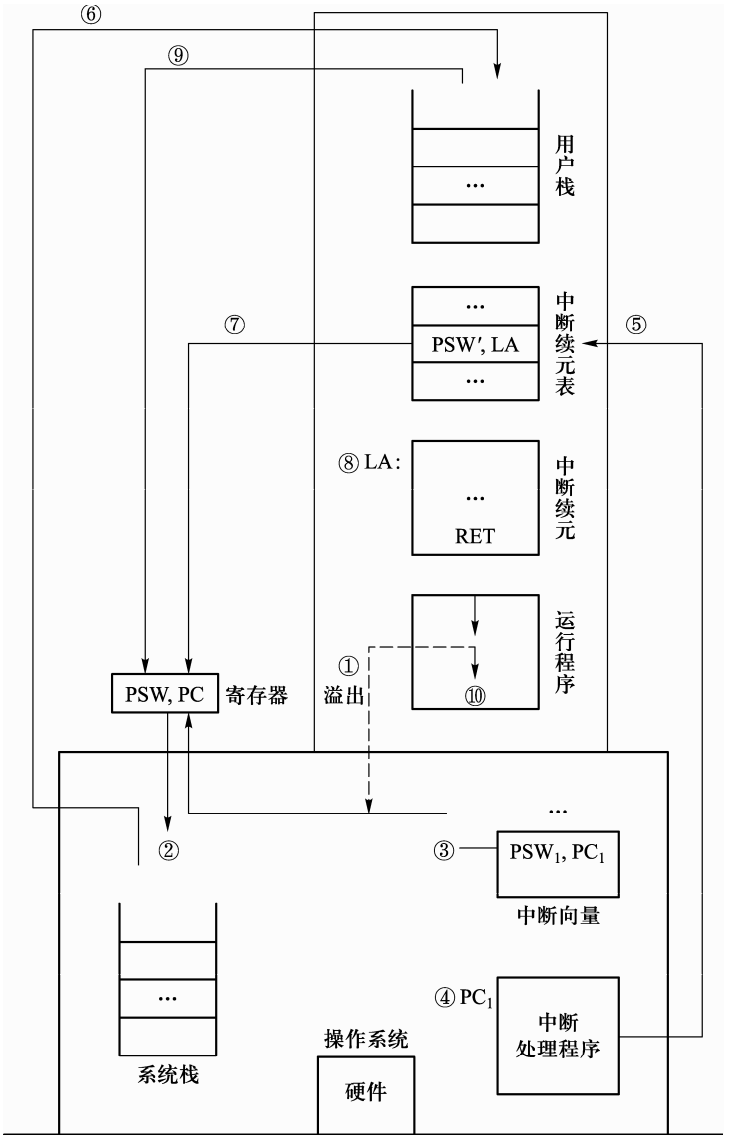

图 3-10 用户自行处理中断的过程

⑨ 中断续元执行完毕，遇到 RET 指令由用户栈弹出现场信息并送入寄存器 PSW 和 PC。

⑩ 返回中断处继续执行。

步骤⑦完成后操作系统已经完成中断处理，中断续元的执行犹如目态子程序的执行，执行后由目态栈恢复现场，与子程序返回的情况相同。

6. 自愿性中断处理

如前所述，这类中断是由于用户程序在目态执行访管指令而引起的，其目的是要求系统为

其提供某种服务。访管指令由指令码和访管中断号组成，即

SVC *n* （3-1）

其中，SVC（supervisor call）为指令码，表明是访管指令; *n* 为访管中断号，其值是一个整数，表示访问要求的类型。中断发生时，硬件中断装置将访管中断号 *n* 送入旧的程序状态字内的中断码字段，访管中断总控程序从系统栈中将其取出，并据此转入对应的服务程序。

在实际使用时，用户程序与操作系统之间还需要相互传递参数和返回值。如此，用户使用访管指令的一般形式如下：

准备参数

SVC *n* （3-2）

取返回值

根据具体访管要求的约定，参数和返回值既可以通过寄存器传递，也可以通过内存空间的栈传递。对于后者，操作系统必须能够访问进程空间。

通常将（3-2）称为系统调用命令，它除包含访管指令外，还有准备参数和取返回值。为了使用起来方便，在高级语言中一般将其写为与过程调用相类似的形式，即

返回值＝系统调用名称(参数 1,参数 2,…,参数 *m*); （3-3）

当然，编译程序会将（3-3）翻译成形如（3-2）的形式。其中，系统调用名称对应（3-1），不同的系统调用名称对应不同的整数 *n*。在有的书中，也将（3-3）称为代表（3-2）的宏指令或广义指令。

不同系统所提供的服务项目在数量和形式上各有差别，通常分为以下几类。

① 与文件有关的系统调用，如建立文件、撤销文件、打开文件、关闭文件、读写文件、文件指针定位等。

② 与进程有关的系统调用，如创建进程、撤销进程、创建线程、监督进程运行状况等。

③ 与通信有关的系统调用，如发送消息、接收消息等。

④ 与同步有关的系统调用，如 P 操作、V 操作等。

访管指令的执行将触发自愿性中断，由中断向量引导进入系统调用总控程序。该程序由旧的程序状态字中取出中断码 *n*，然后根据 *n* 的值转到对应的服务程序。为了方便转移的实现，系统中通常设有一个系统调用驱动表，如图 3-11 所示。根据访管中断号查找此表便可找到对应系统调用命令处理程序的入口。

系统调用号 (访管中断号)	服务程序入口
1	$Addr_1$
2	$Addr_2$
…	…
n	$Addr_n$

图 3-11 系统调用驱动表

3.2 处理器调度


处理器调度

处理器调度（CPU scheduling）指 CPU 资源在可运行实体之间的分配。在不支持线程的系统中，CPU 是分配给进程的，目前大多数操作系统都支持线程，此时线程是 CPU 资源分配的基本单位。这里需要强调的是，操作系统可见的线程是核心级别线程，如果线程是在用户级实现的，操作系统实际调度的实体仍然是进程。

处理器资源管理需要解决 3 个问题：按照什么原则分配处理器，即需要确定处理器调度算法；什么时候分配处理器，即需要确定处理器调度时机；如何分配处理器，即需要给出处理器调度过程。

3.2.1 处理器调度算法

因为在一个计算机系统中处于可运行状态的进程个数通常比处理器的个数要多，所以当处理器空闲时，需要从就绪态的进程中选择一个使其投入运行。具体选择哪一个要按照某一种算法确定。从资源的角度来看，该算法确定了处理器的分配策略，故称其为处理器调度算法；从资源使用者的角度来看，该算法确定了进程的运行次序，故也称其为进程调度算法。处理器调度算法的选择与系统的性能、效率等都有着直接的关系，需要根据系统的设计目标认真加以选择。

调度算法的选择应当与系统的设计目标保持一致，同时考虑公平性与用户满意度。具体考虑以下指标。

① CPU 利用率：使 CPU 尽量处于忙碌状态。

② 吞吐量：单位时间内所处理计算任务的数量。

③ 周转时间：从计算任务就绪到处理完毕所用的时间。

④ 响应时间：从任务就绪到开始处理所用的时间。

⑤ 系统开销：系统调度进程过程中所付出的时空代价。

不同类型系统的设计目标差别较大，例如批处理系统希望处理尽量多的计算任务，即希望吞吐量大；分时系统要求响应及时；实时系统要求满足任务的开始截止期和（或）完成截止期等。

进程的运行需要处理器资源，执行输入输出操作需要设备资源，这两类资源往往是交替使用的，周而复始，直至结束。对处理器的一次连续使用称为 CPU 阵发期（CPU burst cycle），对设备的一次连续使用称为 I/O 阵发期（I/O burst cycle）。进程实际执行时为 CPU 阵发期与 I/O 阵发期的交替，形式如下：

CPU 阵发期→I/O 阵发期→CPU 阵发期→……→I/O 阵发期→CPU 阵发期

在 CPU 阵发期内，进程使用 CPU 进行计算；在 I/O 阵发期内，进程启动并等待输入输出操作完成。处于 CPU 阵发期的进程所需要的处理时间称为阵发时间（burst time）。不同进程的 CPU 阵发时间不同，同一进程在不同 CPU 阵发期的 CPU 阵发时间也不相同。

处理器调度考虑当前处于 CPU 阵发期内的进程的集合，根据各个进程的 CPU 阵发时间，按照某种策略（即处理器调度算法）安排其执行次序。由于阵发时间只能在阵发期结束时才能确切得知，因而调度时只能根据以前的阵发时间来推断本次阵发时间。令 τ_n 是估计的第 n 个 CPU 阵发期的长度，则有如下递推公式：

$$\tau_{n+1}=\alpha t_n+(1-\alpha)\tau_n$$

其中，t_n 的值是进程最近一次 CPU 阵发期的长度，属最近信息；τ_n 是估算的第 n 个 CPU 阵发期的长度，它包含进程的历史信息。参数 α（$0\leqslant\alpha\leqslant1$）控制 t_n 和 τ_n 在公式中所起的作用：当 $\alpha=0$ 时，$\tau_{n+1}=\tau_n$；当 $\alpha=1$ 时，$\tau_{n+1}=t_n$。通常 α 取值为 0.5。

如果作业 J_i 进入系统的时间为 t_s，完成时刻为 t_f，则作业的周转时间（turnaround time）为

$$T=t_f-t_s$$

实际上，作业的周转时间是作业等待时间与处理时间之和。作业的平均周转时间（average turnaround time）定义为所有作业周转时间与作业道数的比值，即

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

如果作业 J_i 的周转时间为 T，所需运行时间为 R，则定义

$$W=\frac{T}{R}$$

为该作业的带权周转时间。由于 T 是等待时间与处理时间之和，故带权周转时间总是不小于 1 的。平均带权周转时间定义为所有作业的带权周转时间与作业道数的比值，即

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

通常，用平均周转时间来衡量对同一作业流执行不同作业调度算法时所呈现的调度性能；用平均带权周转时间衡量对不同作业流执行同一调度算法时所呈现的调度性能。显然，这两个值都是越小越好。

处理器调度通常采用先到先服务、最短作业优先、最短剩余时间优先、最高响应比优先、最高优先数优先、循环轮转、分类排队、反馈排队等算法。下面分别加以介绍。


处理器调度算法 1

1．先到先服务算法

先到先服务（first come first service，FCFS）算法按照进程申请 CPU 的次序，即进入就绪态的次序来调度。先到先服务算法具有公平的优点，不会出现饿死的情况。其缺点是短进程（线程）的等待时间长，从而平均等待时间较长。

例 3-1 考虑如下 CPU 阵发进程，时间单位为 ms（毫秒）。

进程	到达时间	CPU 阵发时间
P_1	0	27
P_2	1	3
P_3	2	5

CPU 调度状况可用甘特图（Gantt chart）来表示，如图 3-12 所示。

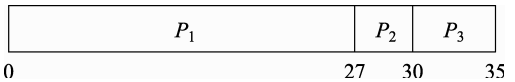

图 3-12 先到先服务算法

先到先服务算法调度性能指标如表 3-2 所示。

表 3-2 先到先服务算法调度性能指标

进程	到达时间/ms	运行时间/ms	开始时间/ms	完成时间/ms	周转时间/ms	带权周转时间/ms
P_1	0	27	0	27	27	1
P_2	1	3	27	30	29	9.67
P_3	2	5	30	35	33	6.6
平均周转时间 $\overline{T} \approx 29.67$ ms，平均带权周转时间 $\overline{W} \approx 5.76$ ms						

2．最短作业优先算法

最短作业优先（shortest job first，SJF）算法按照 CPU 阵发时间递增的次序调度，易于证明其平均周转（等待）时间最短。

例 3-2 对于如下进程集合、到达时间和 CPU 阵发时间：

进程	到达时间	CPU 阵发时间
P_1	0	12
P_2	0	5
P_3	0	7
P_4	0	3

采用最短作业优先算法，甘特图如图 3-13 所示，平均等待时间为（0+3+8+15）/4=6.5（ms）。

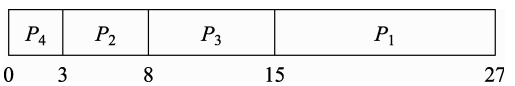

图 3-13 最短作业优先算法

最短作业优先算法调度性能指标如表 3-3 所示。

表 3-3 最短作业优先算法调度性能指标

进程	到达时间/ms	运行时间/ms	开始时间/ms	完成时间/ms	周转时间/ms	带权周转时间/ms
P_1	0	12	15	27	27	2.25
P_2	0	5	3	8	8	1.6
P_3	0	7	8	15	15	2.14
P_4	0	3	0	3	3	1
平均周转时间 $\overline{T} = 13.25$ ms，平均带权周转时间 $\overline{W} \approx 1.75$ ms						

最短作业优先算法最大限度地降低了平均等待时间，但是也带来不公平性：一个较长的就绪任务可能由于短任务的不断到达而长期得不到运行机会，发生饥饿，甚至被饿死。

3．最短剩余时间优先算法

最短剩余时间优先（shortest remaining time next，SRTN）算法是剥夺式调度算法，当 CPU 空闲时，选择剩余时间最短的进程或线程。当一个新进程或线程到达时，比较新进程所需时间与当前运行进程的估计剩余时间。如果新进程所需的运行时间短，则切换运行进程。该算法实质上是可剥夺形式的最短作业优先算法。

例 3-3 对于如下进程集合、到达时间和 CPU 阵发时间：

进程	到达时间	CPU 阵发时间
P_1	0	12
P_2	1	9
P_3	3	6
P_4	5	3

采用最短剩余时间优先算法，甘特图如图 3-14 所示。

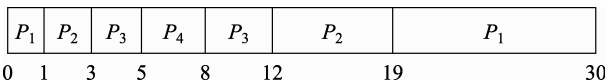

图 3-14 最短剩余时间优先算法

最短剩余时间优先算法调度性能指标如表 3-4 所示。

表 3-4 最短剩余时间优先算法调度性能指标

进程	到达时间/ms	运行时间/ms	开始时间/ms	完成时间/ms	周转时间/ms	带权周转时间/ms
P_1	0	12	0	30	30	2.5
P_2	1	9	1	19	18	2
P_3	3	6	3	12	9	1.5
P_4	5	3	5	8	3	1
平均周转时间 $\overline{T}$ = 15 ms，平均带权周转时间 $\overline{W}$ = 1.75 ms						

4．最高响应比优先算法

最高响应比优先（highest response ratio next，HRN）算法是先到先服务算法和最短作业优先算法的折中，响应比的计算公式如下：

$$RR = \frac{BT + WT}{BT} = 1 + \frac{WT}{BT}$$

其中，RR 为响应比，BT 为 CPU 阵发时间，WT 为等待时间。显然，对于同时到达的任务，处理时间较短的任务将被优先调度，处理时间较长的任务将随其等待时间的增加而动态提升其响应比，因而不会出现饥饿现象。

5. 最高优先数优先算法


处理器调度算法 2

采用最高优先数优先（highest priority number first）算法，在每一个进程的进程控制块中有一个由数字表示的优先数。当需要进行处理器分配时，系统在可运行的进程中选择优先数最高者使其投入运行。进程优先数的大小应当与进程所对应事件的紧迫程度相对应。如果一个进程所对应的事件比较紧迫，则其优先数应当比较高；如果一个进程所对应的事件不太紧迫，则其优先数可以比较低。可见，进程的优先数反映了进程运行的优先级别，故又将其称为优先级法。

选用这种调度算法，尚有两个问题需要解决，即如何确定进程的优先数及何时进行处理器调度。

关于进程的优先数，有以下两种确定方法。

① 静态优先数（static priority）。每个进程在进入系统时被赋予一个优先数，该优先数在进程的整个生存周期内是固定不变的。这种优先数确定方法的优点是比较简单，开销较小；其缺点是公平性差，可能会造成低优先数进程长期等待。

② 动态优先数（dynamic priority）。每个进程在创建时被赋予一个优先数，该优先数在进程的生存周期内是可以动态变化的。比如，当进程获得某种资源时，应当提高它的优先级，以便尽快获得处理器投入运行，这样可以避免资源的浪费。又如，当进程处于就绪态时，它的优先级应当随着其等待处理器时间的增长而提高，以使各个进程获得处理器的机会基本均等。这种优先数确定方法的优点是资源利用率高，公平性好；其缺点是开销较大，实现较为复杂。

关于处理器的调度时刻，有以下两种选择方法。

（1）非剥夺式

所谓非剥夺式（non-preemptive，也称非抢先），就是一个进程不能将处理器资源强行地从正在运行的进程那里抢占过来。亦即一旦一个进程被进程调度程序所选中，它将一直运行下去，直至出现如下两种情形：该进程因某种事件而等待，或者该进程运行完毕。这种方法的优点是系统开销较小，其缺点是不能保证当前正在运行的进程永远是系统内当前可运行进程中优先数最高的进程。

（2）剥夺式

所谓剥夺式（preemptive，也称抢先），就是一个进程能够将处理器资源强行地从正在运行的进程那里抢占过来。此时，当发生如下 3 种情形时可能发生进程切换：正在运行的进程因某种事件而等待；正在运行的进程运行完毕；出现新的就绪进程，该进程的优先级高于正在运行的进程的优先级。注意，此处的“出现”有两种情况：其一是某一进程被唤醒（由等待态转换为就绪态），其二是动态创建了新的进程。这种方法的优点是能够保证正在运行的进程永远是系统内当前可运行进程中优先数最高的进程；其缺点是处理器在进程之间的切换比较频繁，系统开销较大。

例 3-4 对于如下进程集合、到达时间、优先级和 CPU 阵发时间：

进程	到达时间	优先级	CPU 阵发时间
P_1	0	0	8
P_2	2	1	5
P_3	4	3	7
P_4	0	2	3
P_5	5	7	2

采用最高优先数优先算法，甘特图如图 3-15 所示，调度性能指标如表 3-5 所示。

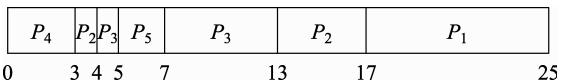

图 3-15 最高优先数优先算法

表 3-5 最高优先数优先算法调度性能指标

进程	到达时间/ms	运行时间/ms	优先级	开始时间/ms	完成时间/ms	周转时间/ms	带权周转时间/ms
P_1	0	8	0	17	25	25	3.13
P_2	2	5	1	3	17	15	3
P_3	4	7	3	4	13	9	1.29
P_4	0	3	2	0	3	3	1
P_5	5	2	7	5	7	2	1
平均周转时间 $\overline{T}$ = 10.8 ms　　平均带权周转时间 $\overline{W}$ ≈ 1.88 ms							

6．循环轮转算法

采用循环轮转（round-robin，RR）算法，系统为每一个进程规定一个时间片（time slice），所有进程按照其时间片的长短轮流地运行。也就是说，将所有就绪进程排成一个队列，即就绪队列。每当处理器空闲时便选择队列头部的进程投入运行，同时分给它一个时间片。当此时间片用完时，如果此进程既未结束其 CPU 阵发期也未因某种原因而等待，则剥夺此进程所占有的处理器，将其排在就绪队列的尾部，并选择就绪队列中的队头进程运行。循环轮转算法在实现时又分两种情形，即基本轮转和改进轮转。

（1）基本轮转

对于基本轮转来说，分给所有进程的时间片的长度是相同的，而且是不变的。若不考虑数据传输等待，系统中的所有进程以基本上均等的速度向前推进。

（2）改进轮转

对于改进轮转来说，分给不同进程的时间片的长度是不同的，而且（或者）是可变的。此时，系统可以根据不同进程的不同特性为其动态地分配不同长度的时间片，以便达到更灵活的调度效果。

对于循环轮转算法来说，时间片的长度需要认真加以考虑。如果时间片过长，则会影响系

统的响应速度；如果时间片过短，则会频繁地发生进程切换，增加系统的开销。通常，时间片的长度为几十毫秒至几百毫秒。循环轮转算法特别适用于分时系统，具有公平、响应及时等特点。

例 3-5 对于如下进程集合、到达时间和 CPU 阵发时间：

进程	到达时间	CPU 阵发时间
P_1	0	12
P_2	0	4
P_3	0	7
P_4	0	3

假定进程 P_1、P_2、P_3、P_4 依次进入就绪队列，但是彼此相差时间很短，可以近似认为“同时”到达，采用基本轮转算法，假定时间片长度为 2 ms，甘特图如图 3-16 所示。调度性能指标如表 3-6 所示。

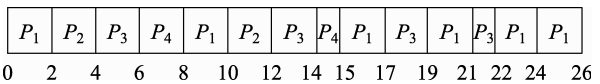

图 3-16 基本轮转算法

表 3-6 循环轮转算法调度性能指标

进程	到达时间/ms	运行时间/ms	开始时间/ms	完成时间/ms	周转时间/ms	带权周转时间/ms
P_1	0	12	0	26	26	2.17
P_2	0	4	2	12	12	3
P_3	0	7	4	22	22	3.14
P_4	0	3	6	15	15	5
平均周转时间 $\overline{T}$ = 18.75 ms，平均带权周转时间 $\overline{W}$ ≈ 3.33 ms						

7．分类排队算法

分类排队（multilevel queue，MLQ）算法是以多个就绪进程队列为特征的。这些队列将系统中所有可运行的进程按照某种原则加以分类，以实现所期望的调度目标。例如，在通用操作系统中，可以将所有就绪进程排成如下 3 个队列。

Q_1：实时就绪进程队列

Q_2：分时就绪进程队列

Q_3：批处理就绪进程队列

当处理器空闲时，首先选择 Q_1 中的进程。如果该队列为空，则选择 Q_2 中的进程。如果上述两个队列均为空，则选择 Q_3 中的进程。在每个队列内部又可以分别采用最高优先数优先算法和（或）循环轮转算法，如 Q_1 和 Q_3 采用最高优先数优先算法，Q_2 采用循环轮转算法。

8. 反馈排队算法

在分类排队算法中，尽管系统中设有多个进程就绪队列，但是一个进程仅属于一个就绪队列，即进程不能在不同的就绪队列之间移动。反馈（feedback）排队算法也是以多个进程就绪队列为特征的，在每个队列中通常采用循环轮转算法，所不同的是，进程可以在不同的就绪队列之间移动。这些就绪队列的优先级依次递减，而其时间片的长度则依次递增，如图 3-17 所示。当一个进程被创建或所等待的事件发生时，进入第一级就绪队列。当某队列的一个进程获得处理器并且使用完该队列所对应的时间片后，如果它尚未结束，则进入下一级就绪队列。当最后一级队列的一个进程获得处理器并且使用完该队列所对应的时间片后，如果它尚未结束，则进入同一级就绪队列。当处理器空闲时，先选择第一级就绪队列中的进程，当该级队列为空时，选择第二级就绪队列的进程，以此类推。

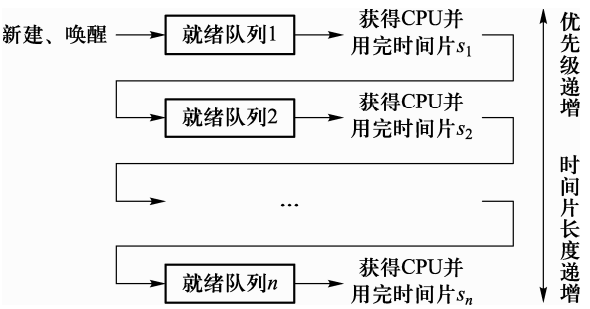

图 3-17 反馈排队算法

这种进程调度算法有以下特点。

① 短进程优先处理，因为运行时间短的进程在经过前面几个队列之后便执行完毕，而这些队列中的进程将被优先调度。

② 设备资源利用率高，因为与设备打交道的进程可能会由于下面两个原因而进入等待态：所申请的资源被占用，或者启动了数据传输。当进程得到所等待的资源或数据传输完成时，它将进入第一级就绪队列，尽快获得处理器并使用资源。

③ 系统开销较小，因为运行时间长的进程最后进入低优先级的队列，这些队列的时间片较长，因而进程的调度频率较低。

应当指出，如果高优先级队列一直不为空，则低优先级队列中的进程可能长时间得不到运行的机会，如此便可能会发生“饥饿”现象。为解决这一问题，可以根据某种原则将低优先级队列中的进程移到高优先级队列中去。反馈排队调度是适应性（adaptive）调度的一个很好的例子。

3.2.2 处理器调度时机

为了下面说明方便，引入一个概念：进程切换。如果在时刻 T_1 进程 P_1 在运行，在时刻 T_2

进程 P_2 在运行，且 $P_1 \neq P_2$，则在时刻 T_1 和时刻 T_2 之间发生了进程切换，并将 P_1 称为下降进程，P_2 称为上升进程。

处理器调度程序是操作系统低层中的一个模块，在系统运行的过程中，除非显式地调用到该模块，否则系统不会由运行一个进程转去运行另一个进程。也就是说，不会发生进程切换。那么，何时有可能调用到处理器调度程序呢？前提条件是必须进入操作系统，即处于系统态，因为处于用户态运行的用户程序不可能直接调用操作系统中的任何模块。前面已经说明，中断是系统由目态转换为管态的必要条件。据此，假如在时刻 T_1 与时刻 T_2 之间发生了进程切换，则在时刻 T_1 与时刻 T_2 之间一定发生过中断。也就是说，中断是进程切换的前提，也可以说操作系统是中断驱动（interrupt driven）的。

应当指出，中断是可以嵌套的，而且嵌套层数在原则上没有限制。如果发生中断嵌套，系统将一直处于管态。中断返回是逐层进行的，仅当返回到最外层时才退回到目态，如图 3-18 所示。

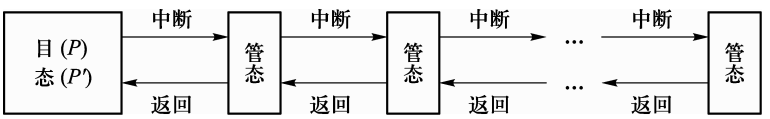

图 3-18 中断与系统状态

那么，中断是否会导致进程切换呢？回答是否定的。也就是说，中断不是进程切换的充分条件。例如，一个进程执行一个系统调用命令将一条消息发给另一个进程，该命令的执行将通过中断进入操作系统，操作系统处理完消息的发送工作后可能返回原调用进程，也可能选择一个新的进程。在图 3-18 中，如果 $P=P'$，则中断并未导致进程切换。如果 $P \neq P'$，则中断导致了进程切换。

进一步探讨，在何种情况下中断可能导致进程切换，在何种情况下中断不会导致进程切换呢？对于这个问题不能给出统一的回答，这与系统中所选择的进程调度策略等因素有关。一般来说，如果在中断处理完后原进程不再具备继续运行的条件，则一定会发生进程切换；反之，如果在中断处理完后原进程仍然具备继续运行的条件，则既可能发生进程切换，也可能不发生进程切换，这与处理器调度策略有关。

例如，一个进程欲读取文件内容，通过读文件系统调用命令进入管态，操作系统在做完必要的准备工作后启动设备进行数据传输。该传输需要一定的时间，因为原进程需要等待数据传输的完成，此时处理器可以转去运行其他进程，发生进程切换，这是原运行进程不再具备运行条件的情形。又如，在采用时间片轮转处理器调度算法的系统中，一个进程在运行过程中发生时钟中断进入管态，操作系统完成与时间相关的系统维护工作后，如果发现此时原进程的时间片已经用完，则将发生进程切换，否则不会发生进程切换，这是原进程仍然具备运行条件的情形。

一定能引起进程切换的中断原因有进程运行终止、进程等待资源、进程等待数据传输的完

成等；可能引起进程切换的中断原因有时钟中断、接收到设备输入输出中断信号等。

3.2.3 处理器调度过程

定义 3-2 与进程相关的运行环境称为进程上下文，包括地址映射寄存器、通用寄存器的状态以及 SP、PSW、PC 等。由一个进程的上下文转换到另一个进程的上下文的过程称为上下文切换，也称为进程切换。

中断事件处理过程中或处理完后，如果需要进行进程切换，则转到处理器分派程序（dispatcher）选择下一个运行进程。处理器调度需要经历 3 个主要步骤，这 3 个步骤一般是在关中断状态下完成的。

1．保存下降进程现场

当前核心态的现场信息（包括地址映射寄存器、通用寄存器以及 SP、PSW、PC）被保存到下降进程的进程控制块中。注意，这里保存的是核心级别的现场，以后再次调度到该进程时将继续核心态尚未执行完的代码。

这里有两种特殊情形：系统初次启动时无下降进程，这一步不执行；下降进程为终止进程，直接转善后处理。

2．选择将要运行的进程

按照处理器调度算法在就绪队列中选择一个进程，准备将其投入运行，被选中的进程称为上升进程。为防止出现就绪队列为空的情况，在系统中通常安排一个特殊的进程，该进程永远也进行不完，且其调度级别最低，称为“闲逛”进程。当系统中无其他进程可运行时，就运行这个“闲逛”进程。“闲逛”进程是容易构造的，如一个死循环程序、一个计算 π 值的程序等所对应的进程都可以作为“闲逛”进程。

3．恢复上升进程现场

由于进程下降时已经将其现场信息保存在对应的进程控制块中，进程上升时应从其进程控制块中恢复现场。进程现场恢复的步骤是：先恢复地址映射寄存器、通用寄存器及 SP 等内容，最后恢复 PSW 和 PC。注意：这里恢复的也是核心级别的现场。

进程调度的 3 个步骤需要执行一定数量的 CPU 指令才能完成。这些指令通常是用汇编语言编写的，而且经过认真优化，以尽量节省处理器时间。由于处理器调度是经常性的工作，节省一条指令都会对提高系统效率产生积极的影响。

3.3 调度级别与多级调度


调度级别与多级调度

前面介绍的处理器调度也称为低级调度（low-level scheduling）或短程调度（short-term scheduling，或称短期调度），它将处理器资源分配给进程或线程使其真正向前推进。在现代操作系统中，除低级调度外，还有中级调度和高级调度，它们与低级调度一同构成多级调度。当然，从理论上说调度可以有更多的级别，但是实际实现的多级调度以两级和三级为主。

3.3.1 交换与中级调度

交换（swapping）是进程在内存和外存储器之间的调度，交换的目标一般有两个：一是缓解内存空间等资源紧张的矛盾，二是减小并发度以降低系统开销。虽然提高并发度可以提高系统资源利用率，从而提高系统效率，但并发的“度”并不是越高越好。并发度过高会导致激烈的资源竞争而使进程经常等待其他进程所占用的资源，从而降低进程推进速度，甚至可能导致死锁。并发度过高还会导致 CPU 资源在进程或线程之间的频繁切换，从而增加系统开销。由于系统中的进程可以根据需要派生子进程，系统对多道程序的并发程度必须有控制能力。

中级调度（middle-level scheduling）也称为中程调度（medium-term scheduling，或称中期调度），是系统控制并发程度的一个调度级别。当系统并发度过高时，将内存中的某些进程暂时交换到外存储器，待以后系统并发度较低时再调回内存。当然，进程在内存和外存储器之间的调度也需要依据某种调度原则，即调度算法。一般依据系统的设计目标确定。具有中级调度的进程状态转换关系如图 3-19 所示。

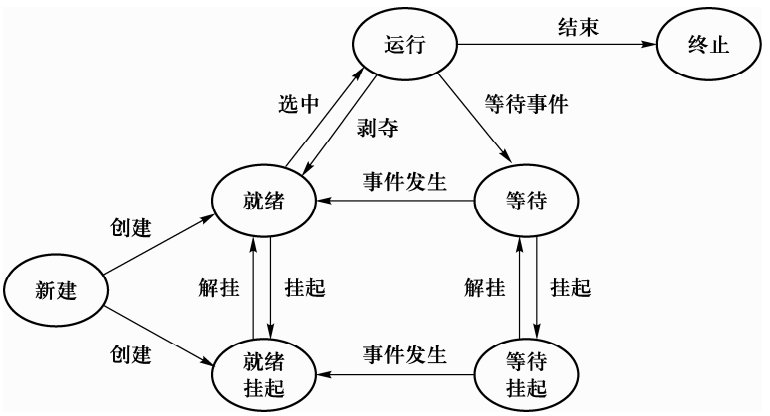

图 3-19 具有中级调度的进程状态转换关系

3.3.2 作业与高级调度


作业

作业调度（job scheduling）又称高级调度（high-level scheduling）或长程调度（long-term scheduling，或称长期调度），其职能是将一个作业由输入井调入内存，并为其建立相应的进程，使其具有运行的资格。

一般来说，一个作业的处理可以分为若干相对独立的执行步骤，称为作业步。每个作业步都可能对应一个进程或线程。例如，一个用 C 和汇编两种语言书写的程序，作为批作业处理大致包括以下步骤：运行 C 语言编译程序对 C 代码部分进行编译，运行汇编程序对汇编代码部分进行汇编，运行连接装配程序对前两步产生的浮动程序进行连接装配，执行上一步产生的目标代码。以上 4 个步骤需要运行不同的程序，因而需要 4 个进程完成。由于前两个步骤可以并行进行，因而同时建立两个进程对其进行处理。

作业一般经历“提交”“后备”“执行”“完成”和“退出”这 5 个状态：由输入机向输入井传输的作业处于“提交”状态，进入输入井尚未调入内存的作业处于“后备”状态，被调度选中进入内存处理的作业处于“执行”状态，处理结果传送到输出井的作业处于“完成”状态，由输出井向打印设备传送的作业处于“退出”状态。作业由“提交”到“后备”的状态转换由假脱机输入完成，由“后备”到“执行”、由“执行”到“完成”的状态转换由作业调度完成，由“完成”到“退出”的状态转换由假脱机输出（SPOOL 输出）完成。作业状态转换关系如图 3-20 所示。

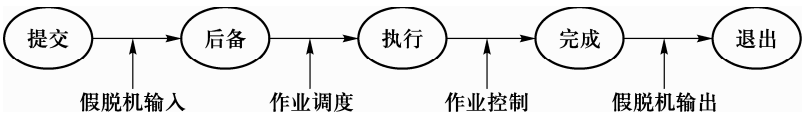

图 3-20 作业状态转换关系

图 3-20 中的假脱机输入和假脱机输出将在虚拟设备的相关内容中介绍；作业调度和作业控制是两段系统程序，通常以系统进程的模式运行，位于操作系统中的高层。

1．批处理作业调度程序

该程序按照作业调度算法在后备作业集合中选择作业，并为其建立作业控制进程来处理该作业，其工作流程如图 3-21 所示。

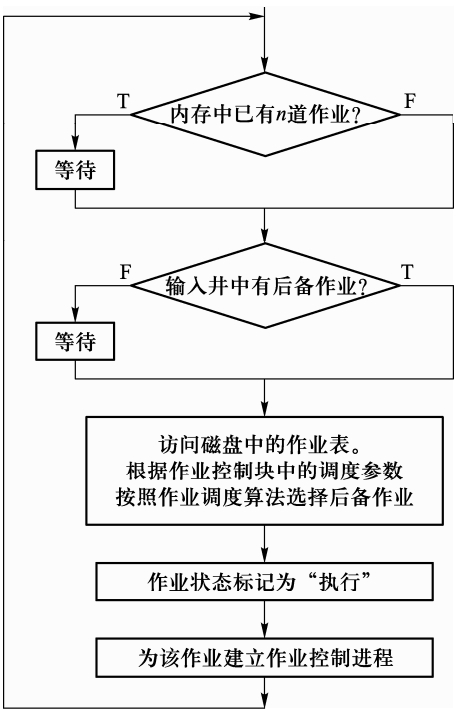

图 3-21 作业调度程序 1

2. 批处理作业控制程序

该程序读入作业并按作业说明书处理作业，对每个作业步可能建立相应的子进程，直到作业处理完毕，其工作流程如图 3-22 所示。

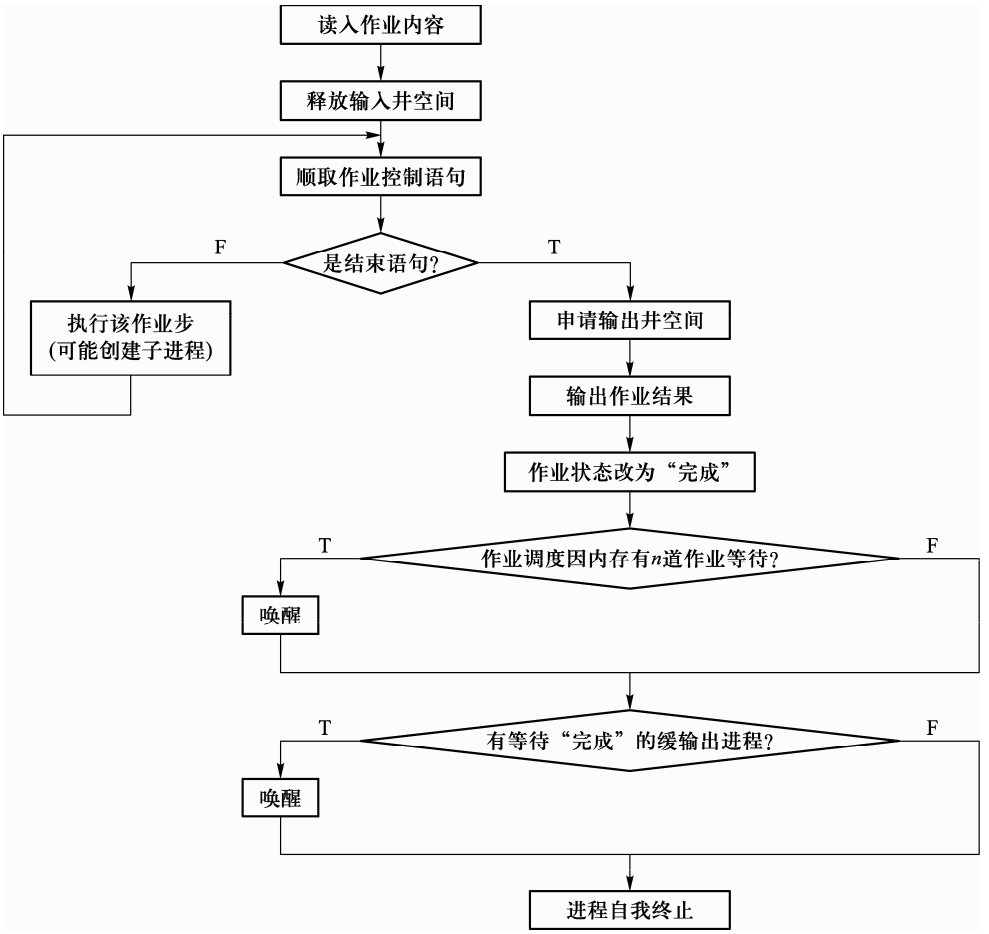

图 3-22 作业控制程序

作业调度使作业以进程的形式进入内存并获得运行资格，但真正获得 CPU 运行还需要经过进程调度或线程调度。作业调度、中级调度、进程调度构成调度的 3 个主要级别。


实时调度与多处理器调度

3.4 实时调度

在实时系统中，时间起着至关重要的作用，即对每一个实时任务，都有一个时间约束要求，如在何时必须开始处理、在何时必须处理完毕等。在一个实

时应用系统中可能有多个实时任务，每个实时任务都有其时间约束，实时调度（real-time scheduling）的目标就是合理地安排这些任务的执行次序，使之满足各个实时任务的时间约束条件。

定义 3-3 满足实时任务各自时间约束条件的调度称为实时调度。

实时任务按其发生规律可以分为以下两类。

① 随机性实时任务：由随机事件触发，其发生时刻不确定。

② 周期性实时任务：每隔固定时间发生一次。

按对时间约束的强弱程度又可分为“硬实时”（hard real-time）和“软实时”（soft real-time）。前者必须满足任务截止期的要求，错过截止期可能导致严重的后果；后者则期望满足截止期要求，但是错过截止期仍然可以容忍。目前支持硬实时的系统并不多。

与实时调度相关的概念有：实时任务产生并可以开始处理的时间，称为就绪时间（ready time）；实时任务最迟开始处理的时间，称为开始截止期（starting deadline）；实时任务处理所需要的处理器时间，称为处理时间（processing time）；实时任务最迟完成时间，称为完成截止期（completion deadline）；发生周期性实时任务的间隔时间，称为发生周期（occurring period）；实时任务的相对紧迫程度通常用优先级表示。表 3-7 给出两个周期性实时任务。

表 3-7 周期性实时任务

进程	就绪时间/ms	处理时间/ms	完成截止期/ms	发生周期/ms
A	0	10	20	20
B	0	25	50	50

对于周期性实时任务来说，令 C_i 为任务 P_i 的处理时间，T_i 为任务 P_i 的发生周期，则任务 $P_1, P_2, \cdots, P_n$ 可调度的必要条件为

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

当其值为 1 时，处理器已经达到最大利用率（100%），这是理想调度算法可以达到的最好结果。对于某些算法，上述公式并非可调度的充分条件。

对于非周期性实时任务来说，如果其到达服从参数为 λ 的泊松分布（λ 为单位时间内任务到达系统的次数），则任务的平均到达时间间隔为 $\frac{1}{\lambda}$。以 $\frac{1}{\lambda}$ 作为非周期性实时任务的平均周期，通过测试得到 $\frac{1}{\lambda}$ 的值。如果非周期性实时任务的 $\frac{1}{\lambda}$ 值较小，不大于某一个阈值，也就是说发生的频率较高，可将其视为周期性实时任务处理。

实时调度有剥夺式与非剥夺式之分。剥夺式调度实现灵活，在满足调度约束条件的前提下可以充分发挥处理器利用率，但是实现略显复杂；非剥夺式调度实现简单，但是为满足约束条件，处理器往往不能满负载运行。

3.4.1 最早截止期优先调度

最早截止期优先（earliest deadline first，EDF）调度优先选择完成截止期最早的实时任务。对于新到达的实时任务，如果其完成截止期先于正在运行任务的完成截止期，则重新分派处理器，即剥夺。可以证明，对于最早截止期优先调度算法来说，公式为

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

这是实时任务可调度的充分条件。例如，对于某进程集合示例，按最早截止期、可抢先原则调度得到的结果如图 3-23 所示。

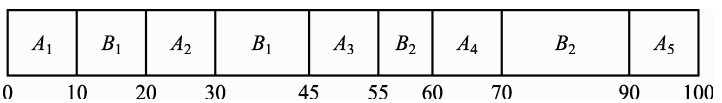

图 3-23 最早截止期优先调度结果

3.4.2 单调速率调度

单调速率调度（rate-monotonic scheduling，RMS）于 1973 年由 Liu 和 Layland 提出，面向周期性实时任务，属于非剥夺式调度的范畴。速率单调调度将任务的周期作为调度参数，其发生频度越高，则调度级别越高。Lin 和 Layland 已经证明，速率单调调度算法可调度的条件如下：

$$\sum_{i=1}^{n}\frac{C_i}{T_i}\leqslant n(2^{\frac{1}{n}}-1)$$

对应不同的 n 值，表 3-8 给出速率单调调度算法可调度的上限值。可以看出，随着任务数的增加，调度限制趋向 ln 2≈0.693。

表 3-8 速率单调调度算法可调度的上限值

n	$n(2^{\frac{1}{n}}-1)$
1	1.0
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
…	…
∞	ln 2≈0.693

例如，考虑表 3-9 所示的 3 个周期性实时任务。

表 3-9 周期性实时任务

进程	发生周期 T_i/ms	处理时间 C_i/ms
A	100	20
B	150	40
C	350	100

由于

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+\frac{C_3}{T_3}\approx 0.2+0.267+0.286=0.753\leqslant 3(2^{\frac{1}{3}}-1)\approx 0.780$$

因而可知速率单调调度能够满足所有任务的调度要求。具体调度结果如图 3-24 所示。

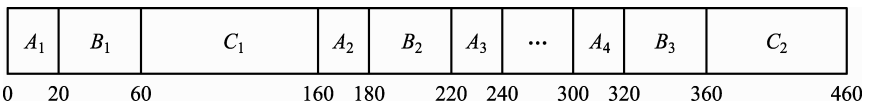

图 3-24 速率单调调度结果

从 CPU 利用效率来说，速率单调调度算法不及最早截止期优先调度算法效率高，但是具有以下优势：速率单调调度算法是非剥夺式的，实现简单，开销小；二者效率相差得并不十分明显，对于具体任务集合来说，处理器利用效率可达 90%左右。

3.4.3 最小裕度优先调度

最小裕度优先(minimum laxity first)算法类似于最早截止期优先调度算法，但它的优先级基于一个进程的裕度，令 *D* 为任务截止期，*T* 为当前时间，*C* 为剩余处理时间，则裕度 *L* 的计算公式如下：

$$L=D-(T+\mathrm{C})$$

例如，假定 *T*=5，*C*=3，*D*=9，则 *L*=1。如果一个任务的裕度为 0，则它必须立即分派，否则将错过截止期。与最早截止期优先调度算法相比，最小裕度优先算法中的优先级更加精确。可以证明，对于最小裕度优先调度算法，可调度的充分条件与最早截止期优先调度算法相同。

前面的讨论假定系统中只有一个处理器，就绪进程数量与处理器数量之间为多对一的关系，这是比较简单的情况。目前随着处理器价格的下降，多处理器已经成为服务器的普遍配置，一些个人计算机主板也增加了 CPU 插槽，许多商业操作系统如 Linux、Windows 等都提供对多处理器的支持。本书将在第 12.3.2 节讨论多处理器（多核）调度问题。

3.5 系统举例

Linux 与 Windows 调度

3.5.1 Linux 进程调度

Linux 在调度级别上考虑 3 种特征的进程：实时先进先出，实时轮转，分

时。调度基于 Goodness 度量指标，涉及以下一些参数。

priority：1~40（默认值为 20），可以通过 nice 系统调用进行调整。nice(value)中 value 的取值范围为（–20,20），以与经典 UNIX 保持兼容，但是在内部对 value 值进行反向处理，取 priority=20–value。

counter：进程尚可运行的剩余时间。对于运行进程来说，每个时钟间隔（10 ms，称为一个瞬间）将 counter 值减 1，当所有就绪进程的 counter 配额下降至 0 时，重新计算所有进程（包括等待进程）的 quantum 值。Goodness 值的计算方法如下：

If (Real-time) Goodness=1000 + priority;

If (Timesharing && counter=0) Goodness=0;

If (Timesharing && counter>0) Goodness=counter + priority；

调度发生在以下时刻：运行进程的 counter 值减至 0，运行进程执行系统调用 exit，运行进程因等待输入输出操作、信号量而被阻塞，原来具有高 Goodness 的进程被解除阻塞。容易看出调度效果：实时优先于分时，交互和 I/O 进程优先于 CPU 进程。

Linux 2.0 是支持对称多处理器的第一个 Linux 核心，进程或线程可以同时运行在多个处理器上。为满足核心非剥夺同步的要求，对称多处理器通过唯一的核心自旋锁（spin-lock）来保证在任何时刻最多只有一个处理器执行核心代码。Linux 2.1 支持真正意义上的对称多处理器：将一个自旋锁分解为若干个相互独立的自旋锁，分别用于保护核心代码不相交的子集。

3.5.2 Windows 10 线程调度

1．基本调度

Windows 10 核心以线程作为调度的基本单位，调度算法为可剥夺动态优先级算法，结合时间片轮转算法的思想，这里的时间片称为时间配额（quantum）。优先级的范围为 0（低）～31（高），其中 16～31 称为实时优先级，1～15 称为可变优先级，0 为系统闲置优先级。当没有实际需要运行的线程时，一个核心级别护线程负责对页面清零。

实时优先级一般用于系统线程，用户线程欲将优先级提升为实时优先级需要有权限。请注意这里尽管使用了“实时”这一术语，系统并不支持截止期描述，因而不是真正意义上的实时调度。

用户线程一般具有一个基本优先级。线程的基本优先级可以等于其所属进程的基本优先级，也可以与其进程的基本优先级相差 2 个级别。

可变优先级的线程在活动期内，其实际优先级可以在一定的范围内浮动，体现了优先级的动态性。动态优先级不会低于基本优先级，也不会高于 15。例如，基本优先级为 4 的进程，其线程的基本优先级在 2～6 的范围内，动态优先级可在 2～15 的范围内变动，如图 3-25 所示。

如果一个线程用完时间配额而让出处理器，其实际优先级将下降。而在等待被唤醒后，其实际优先级将上升，具体上升幅度视条件而定。例如输入输出操作完成时可上升 1～8 个级别（磁盘 1、串口 2、键盘 4、声卡 8）。此外，前台进程中的线程完成一个等待操作、因窗口活动而唤

醒 GUI 线程、就绪超过一定时限未获得处理器时也会提升其优先级，但优先级不会超过 15。可见，偏重 CPU 的线程的优先级较低，而交互性较强和偏重输入输出操作的线程的优先级较高。

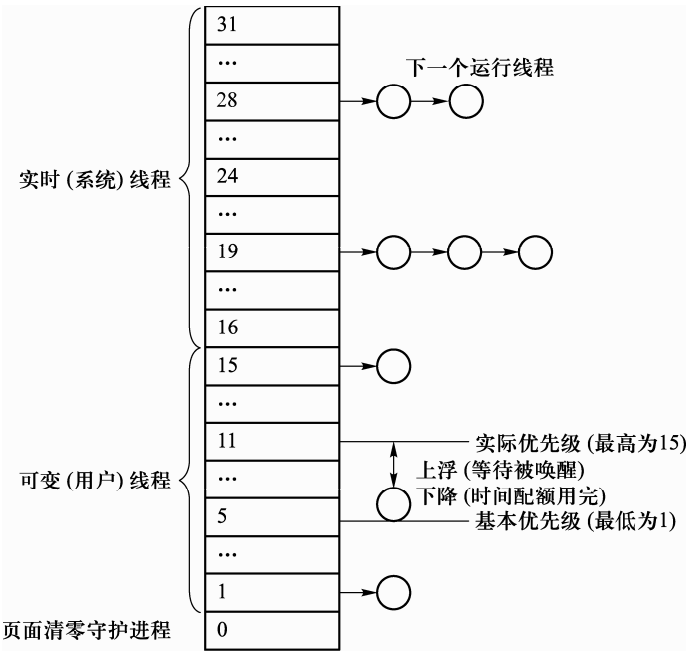

图 3-25 Windows 10 的调度队列

在默认情况下，在 Windows 10 专业版中线程开始时的时间配额为 6，在服务器版中为 36。在服务器版中取较大的时间配额是为了保证应用程序有足够的时间来完成用户请求并回到等待态。每次时钟中断，系统将运行线程的时间配额减去一个固定值（3）。如果时间配额用完，将选择另一个线程运行。可见在专业版中一个线程的默认运行时间为 2 次时钟中断，在服务器版中默认运行时间为 12 次时钟中断。

当被唤醒线程优先级高于运行线程优先级，或某就绪线程的优先级动态变化时，就称发生了抢先。这时，被抢先线程回到相应就绪队列，对实时线程而言会重新分配完整时间配额，而其他线程则保持剩余配额。配额长度为 6～36。时钟中断每 15 ms 发生一次，配额会减 3，那么 2～12 次时钟中断（30～180 ms）后配额就会用完，配额用完后进入就绪队列，优先级下降。

可以通过修改注册项 Win32PrioritySeparation 来显式控制时间配额的长度，该注册项包括 3 个字段，其中第 4、5 位表示时间配额长度；第 2、3 位表示前后台变化；第 0、1 位表示前台线程时间配额提升，其取值含义如图 3-26 所示。

2. SMP 上的线程调度

系统定义亲和掩码来表示线程与 CPU 的亲和关系，每个进程有一个处理器亲和掩码，线程的亲和掩码是通过继承其进程的亲和掩码而来的。默认所有进程（线程）的亲和掩码为系统中

所有处理器的集合。可以通过 SetProcessAffinityMask 或 SetThreadAffinityMask 函数来修改亲和掩码。每个线程有两个理想处理器（ideal processor）：首选处理器和第二处理器（在内核线程控制块中），理想处理器在线程创建时随机确定，目的是分散各个线程与处理器对应关系，以保证负载均衡，也可通过 SetThreadIdealProcessor 来修改理想处理器。在 SMP 上处理器与线程之间具有双向选择关系。

图 3-26 Windows 10 的时间配额调度

（1）就绪线程对处理器的选择。若有空闲处理器，则选择次序为：首选处理器，第二处理器，当前执行处理器（正执行调度代码），由高到低顺序找空闲的处理器；若无空闲处理器，则采用抢先策略，选择次序为：首选处理器，第二处理器，可运行的编号最大的处理器，不能抢先进入相应的就绪队列。

（2）处理器对就绪线程的选择。空闲处理器会调度上次在此 CPU 上运行（二级缓冲利用）的线程，线程的理想处理器是该 CPU。处理器也可基于对线程的约束来选择线程，例如，选择处于就绪状态时间超过 2 个时间配额、线程优先级别大于等于 24 的线程。

习 题 三

1. 用户栈有哪些用途？系统栈有哪些用途？
2. 堆（heap）在进程调度中的用途是什么？
3. 试说明下述概念之间的联系与差别。

（1）系统调用命令 （2）访管指令 （3）广义指令

4. 为什么说中断是进程切换的必要条件，但不是充分条件?
5. 试分析中断与进程状态转换之间的关系。
6. 中断发生时，旧的 PSW 和 PC 为何要压入系统栈?
7. 何谓中断向量? 用户能否修改中断向量的值?
8. 中断向量的存储位置是否可由程序改变？为什么？中断向量的值是如何确定的?
9. 有人说，“中断发生后，硬件中断装置保证处理器进入管态。”这种说法准确吗?试说明理由。
10. 为什么在中断处理过程中通常允许高优先级别的中断事件中途插入，而不响应低优先

级别的中断事件?

11. 为什么说“关中断”会影响系统的并发性?

12. 假如关中断后操作系统进入死循环，将会产生什么后果?

13. 为什么不允许目态程序执行关中断指令及中断屏蔽指令?

14. 如果没有中断，是否能够实现多道程序设计? 为什么?

15. 下列中断源中哪些通常是可以屏蔽的，哪些通常是不可屏蔽的?

（1）输入输出中断 （2）访管中断 （3）时钟中断 （4）掉电中断

16. 下列中断事件中哪些可由用户自行处理，哪些只能由操作系统中断处理程序统一处理? 为什么?

（1）溢出 （2）地址越界 （3）除数为 0 （4）非法指令 （5）掉电

17. 如果中断由用户程序自行处理，为何需要将被中断程序的断点由系统栈弹出并压入用户栈?

18. 对于下述中断与进程状态转换之间的关系，各举两个例子说明之。

（1）定会引起进程状态转换的中断事件

（2）可能引起进程状态转换的中断事件

19. 若在 T_1 时刻进程 P_1 运行，在 T_2 时刻进程 P_2 运行，且 $P_1 \neq P_2$，则在 $T_1 \sim T_2$ 期间内一定发生过中断。这种说法对吗? 为什么?

20. 进程上下文包括哪些内容? 请尽量考虑完整。

21. 中断时，现场保存在什么地方? 进程切换时，现场保存在什么地方? 为什么这样处理?

22. 进程切换时，上升进程的 PSW 和 PC 为何必须由一条指令同时恢复?

23. 某系统采用可抢占处理器的静态优先数调度算法，请问何时会发生抢占处理器的现象?

24. 某系统采用可抢占处理器的动态优先数调度算法，请问何时会发生抢占处理器的现象?

25. 在实时系统中，采用不可抢占处理器的优先数调度算法是否合适?为什么?

26. 在分时系统中，进程调度是否只能采用时间片轮转调度算法?为什么?

27. 有人说，在采用等长时间片轮转处理器调度算法的分时操作系统中，各个终端用户所占处理器的时间总量是相同的。这种说法对吗? 为什么?

28. 对于下述处理器调度算法，分别画出进程状态转换图。

（1）时间片轮转算法

（2）可抢占处理器的优先数调度算法

（3）不可抢占处理器的优先数调度算法

29. 举出两个例子说明操作系统访问进程空间的必要性。

30. 根据进程和线程的组成，说明进程调度和线程调度各需要完成哪些工作。

31. 系统资源利用率与系统效率是否一定成正比?如果不是，试举例说明之。

32. 设有按 P_1、P_2、P_3、P_4 次序到达的 4 个进程，CPU 阵发时间如表 3-10 所示，采用先

到先服务算法和最短作业优先算法，画出甘特图，并计算各自的平均等待时间。

表 3-10 CPU 阵发时间

进程	CPU 阵发时间/ms
P_1	20
P_2	8
P_3	5
P_4	18

33. 设有周期性实时任务集合如表 3-11 所示，用最早截止期优先算法和单调速率调度算法是否可以调度？画出相应的甘特图。

表 3-11 周期性实时任务集合

任务	发生周期 T_i/ms	处理时间 C_i/ms
A	30	10
B	40	15
C	50	5

34. 简述处理器调度的过程。

35. 试分析 Linux 进程调度算法的调度效果。

36. 设有 4 个作业 J_1、J_2、J_3、J_4，其提交时刻与运行时间如表 3-12 所示。试采用先到先服务、最短作业优先、最高响应比优先算法，分别求出各个作业的周转时间、带权周转时间，以及所有作业的平均周转时间、平均带权周转时间。

表 3-12 作业提交时刻与运行时间

作业名	提交时刻	运行时间/h
J_1	10:00	2
J_2	10:20	1
J_3	10:50	0.5
J_4	11:10	0.8

37. 什么是低级调度？低级调度的职能是什么？

38. 什么是中级调度？中级调度的职能是什么？

39. 什么是高级调度？高级调度的职能是什么？