一、 操作系统概述

一、考试大纲

（一）操作系统的概念、特征、功能和提供的服务
（二）操作系统的发展与分类
（三）操作系统的运行环境

二、知识点归纳

（一）操作系统的概念、特征、功能和提供的服务

1．操作系统的概念、目标和作用

一个完整的计算机系统由两大部分组成：计算机硬件和计算机软件。硬件是所有软件运行的物质基础；软件能充分发挥硬件潜能和扩充硬件功能，完成各种系统及应用任务，两者互相促进、相辅相成、缺一不可。计算机硬件是指计算机物理装置本身，由运算器、控制器、存储器、输入设备和输出设备五部分组成。计算机软件是指由计算机硬件执行以完成一定任务的程序及其数据。**计算机软件包括系统软件和应用软件。**系统软件包括操作系统、编译程序、连接装入程序、数据库管理系统等；应用软件是为各种应用目的而编制的程序。

在计算机上配置操作系统的**目的**有以下几点：

①方便用户使用。操作系统应该使计算机系统使用起来十分方便。

②有效性。OS 能够有效管理好系统中的各种硬件软件资源，并通过合理地组织计算机的工作流程，进一步改善资源的利用率及提高系统的吞吐量。

③可扩充性。OS 必须具有很好的可扩充性，应采用层次化结构，以便于增加新的功能层次和模块，并修改老的功能层次和模块。

④构筑开放环境。OS 应该构筑出一个开放环境，主要是指：遵循有关国际标准；支持体系结构的可伸缩性和可扩展性；支持应用程序在不同平台上的可移植性和可互操作性。

操作系统主要由以下的作用：

①OS 作为用户与计算机硬件系统之间的接口：为了使用户能灵活、方便地使用计算机和操作系统，操作系统提供了一组友好的用户接口，包括：1）程序接口；2）命令接口；3）图形接口。

②OS 作为计算机系统资源的管理者：资源包括两大类：硬件资源和软件资源。归纳起来资源分为四类：处理机、存储器、I/O 设备以及信息（数据和程序），OS 的主要功能是对这四类资源进行管理，即处理机管理、存储器管理、I/O 设备管理、文件管理。**（资源管理观点）**

③OS 用作扩充机器：在裸机上覆盖上 OS 后，便可获得一台功能显著增强、使用极为方便的多层扩充机器或多层虚机器。**（虚拟机观点）**

操作系统可定义为：操作系统是一组控制和管理计算机硬件和软件资源，合理地对各类作业进行调度，以及方便用户使用的程序的集合。

2．操作系统的特征

虽然不同的操作系统具有各自的特点，但它们都具有以下 4 个基本特征：

（1）并发性

并行性和并发性是既相似又有区别的两个概念，并发性是指两个或多个事件在同一时刻发生；并发性是指两个或多个事件在同一时间间隔内发生。在多道程序环境下，并发性是指宏观上在一段时间内有多道程序在同时运行，但在单处理机系统中，每一时刻仅能执行一道程序，故微观上这些程序是交替执行的。

（2）共享性

资源共享是指系统中的硬件和软件资源不再为某个程序所独占，而是供多个用户程序共同使用。

并发和共享是操作系统的两个最基本的特征，二者之间互为存在条件。一方面，资源的共享是以程序的并发执行为条件的，若系统不允许程序的并发执行，自然不存在资源共享问题；另一方面，若系统不能对资源共享实施有效的管理，也必将影响到程序的并发执行，甚至根本无法并发执行。

（3）虚拟性

在操作系统中，虚拟是指把一个物理上的实体变为若干个逻辑上的对应物，前者是实际存在的，后备是虚的，只是用户的一种感觉。

（4）异步性（不确定性）

在操作系统中，不确定性有两种含义：

①程序执行结果是不确定的，即对同一程序，使用相同的输入，在相同的环境下运行却可能获得完全不同的结果。亦即程序是不可再现的；

②多道程序环境下程序的执行是以异步方式进行的，换言之，每个程序何时执行，多个程序间的执行顺序以及完成每道程序所需要的时间都是不确定的，因而也是不可预知的。

3．操作系统的功能

操作系统的职能是负责系统中软硬件资源的管理，合理地组织计算机系统的工作流程，并为用户提供一个良好的工作环境和友好的使用界面。下面从 5 个方面来说明操作系统的基本功能。

（1）处理机管理。处理机管理的主要任务是对处理机的分配和运行实施有效的管理。在多道程序环境下，处理机的分配和运行是以进程为基本单位的，因此对处理机的管理可归结为对进程的管理。进程管理应实现下述主要功能：

①进程控制：负责进程的创建、撤消及状态转换。

②进程同步：对并发执行的进程进行协调。

③进程通信：负责完成进程间的信息交换。

④进程调度：按一定算法进行处理机分配。

（2）存储器管理。存储器管理的主要任务是对内行进行分配、保护和扩充。存储器管理应实现下述主要功能：

①内存分配：按一定的策略为每道程序分配内存。

②内存保护：保证各程序在自己的内存区域内运行而不相互干扰。

③地址映射：将地址空间的逻辑地址转换为内存空间与之对应的物理地址。

④内存扩充：为允许大型作业或多作业的运行，必须借助虚拟存储技术去获得增加内存的效果。

（3）设备管理：计算机外部设备的管理是操作系统中最庞杂、琐碎的部分。设备管理的主要任务是对计算机系统内的所有设备实施有效的管理。设备管理应具有下述功能：

①设备分配：根据一定的设备分配原则对设备进行分配。为了使设备与主机并行工作，还需采用缓冲技术和虚拟技术。

②设备传输控制：实现物理的输入输出操作，即启动设备、中断处现、结束处理等。

③设备独立性：即用户向系统申请的设备与实际操作的设备无关。

（4）文件管理。文件管理的主要任务是有效地支持文件的存储、检索和修改等操作，解决文件的共享、保密和保护问题。文件管理应实现下述功能：

①文件存储空间的管理：负责对文件存储空间进行管理，包括存储空间的分配与回收等功能。

②目录管理：目录是用来管理文件的数据结构，它能提供按名存取的功能。

③文件操作管理：实现文件的操作，负责完成数据的读写。

④文件保护：提供文件保护功能，防止文件遭到破坏。

（5）用户接口。为方便用户使用操作系统，操作系统提供了用户接口。操作系统通常提供如下几种类型的用户接口。

①命令接口：提供一组命令供用户直接或间接控制自己的作业。

②程序接口：提供一组系统调用供用户程序和其他系统程序调用。

③图形接口：图形用户接口采用了图形化的操作界面，用非常容易识别的各种图标将系统的各项功能、各种应用程序和文件直观、逼真地表示出来，用户可通过鼠标、菜单和对话框来完成各种应用程序和文件的操作。

4．操作系统提供的服务

操作系统为程序和用户提供了一系列的操作系统服务，这些服务可使程序员更容易地完成他的工作。

（1）操作系统的公共服务类型，主要有：程序执行、I/O 操作、文件系统操作、通信和差错检测等。

（2）系统调用中的作用，系统调用的类型是根据操作系统所提供服务的功能决定的，系统调用可分为进程管理、设备管理、文件管理、信息维护以及通信等。

（二）操作系统的发展与分类

操作系统的主要发展过程如下：

1．无操作系统时的计算机系统

（1）手工操作阶段

早期的计算机系统上没有配置操作系统，计算机的操作由程序员采用手工操作直接控制和使用计算机硬件。程序员使用机器语言编程，并将事先准备好的程序和数据穿孔在纸带或卡片上，从纸带或卡片输入机将程序和数据输入计算机。然后，启动计算机运行，程序员可以通过控制台上的按钮、开关和氖灯来操纵和控制程序，运行完毕，取走计算的结果，才轮到下一个用户上机。这种手工操作方式具有用户独占计算机资源、资源利用率低及 CPU 等待人工操作的缺点。

随着 CPU 速度的大幅度提高，手工操作的慢速与 CPU 运算的高速之间出现了矛盾，这就是所谓的人机矛盾。另一方面，CPU 与 I/O 设备之间速度不匹配的矛盾也日益突出。

（2）脱机输入/输出技术

为解决 CPU 与 I/O 设备之间速度不匹配的问题，将用户程序和数据在一台外围机（又称卫星机）的控制下，预先从低速输入设备输入到磁带上，当 CPU 需要这些程序和数据时，再直接从磁带机高速输入内存，从而大大加快程序的输入过程，减少 CPU 等待输入的时间，这就是脱机输入技术；类似地，当 CPU 需要输出时，无需直接把计算结果送至低速输出设备，而是高速地把结果送到磁带上，然后在外围机的控制下，把磁带上的计算结果由相应的输出设备输出，这就是脱机输出技术。

若输入/输出操作在主机控制下进行则称之为联机输入/输出。

2．单道批处理操作系统

批处理技术是指计算机系统对一批作业自动进行处理的一种技术。早期的计算机系统非常昂贵，为了能充分地利用它，应尽量让系统连续地运行，以减少空闲时间。为此通常把一批作业以脱机输入方式输入到磁带上，并在系统中配置监督程序（是一个常驻内存的程序，它管理作业的运行，负责装入和运行各种系统处理程序来完成作业的自动过渡），在它的控制下，先把磁带上的第一个作业传送到内存，并把运行的控制权交给该作业，当该作业处理完后又把控制权交还给监督程序，由监督程序再把第二个作业装入内存。计算机系统按这种方式对磁带上的作业自动地、一个接一个地进行处理，直至把磁带上的所有作业全部处理完

毕，由于系统对作业的处理是成批进行的、且在内存中始终只保持一道作业，故称为单道批处理系统。其主要特征是：①自动性；②顺序性；③单道性。

3．多道批处理技术

多道程序设计的基本概念：多道程序设计技术是将多个作业存放在内存中并允许它们交替执行，这些作业共享处理机时间和外围设备以及其他资源。当一道程序因某种原因（如 I/O 请求）而暂停执行时，CPU 立即转去执行另一道程序。在操作系统中引入多道程序设计技术后，会使系统具有多道、宏观上并行、微观上串行的特点。

在单道批处理系统中，内存中仅有一道作业，使得系统中仍有较多的空闲资源，致使系统的性能较差，20 世纪 60 年代引入多道程序设计技术后，形成了多道批处理技术，进一步提高了资源利用率和系统的吞吐量。

在多道批处理系统中，用户所提交的作业都先存放在外存并排成一个队列，该队列称为“后备队列”；然后，由作业调度程序按一定的算法从后备队列中选择若干个作业调入内存，使它们共享 CPU 和系统中的各种资源，以达到提高资源利用率和系统的吞吐量的目的。其主要特征是：①多道性；②无序性；③调度性。

4．分时操作系统

（1）分时系统的产生

如果说，推动多道批处理系统形成和发展的主要动力是提高资源利用率和系统吞吐率，那么，推动分时系统形成和发展的主要动力，则是用户的需要。体现在人-机交互、共享主机、便于用户上机等方面。

（2）分时系统的特征

分时系统与多道批处理系统相比，具有完全不同的特征：

①多路性。指一台计算机与若干台终端相连接，系统按分时原则为每个用户服务。宏观上，是多个用户同时工作，共享系统资源；微观上，则是每个用户作业轮流运行一个时间片。多路性亦即同时性，它提高了资源利用率，从而促进了计算机更广泛地应用。

②独立性。每个用户各占一个终端，彼此独立操作、互不干扰。

③及时性。用户的请求能在很短时间内获得响应。

④交互性。用户可通过终端与系统进行广泛的人机对话。其广泛性表现在：用户可以请求系统提供各方面的服务，如文件编辑、数据处理和资源共享等。

5．实时操作系统

（1）实时系统的引入

虽然多道批处理系统和分时系统已获得较为令人满意的资源利用率和响应时间，从而使计算机的应用范围日益扩大，但它们仍然不能满足以下两个领域的需要：

①实时控制：实时控制系统通常是指以计算机为中心的生产过程控制系统，又称为计算机控制系统。例如钢铁冶炼和钢板轧制的自动控制，化工、炼油生产过程的自动控制等。

②实时信息处理：在实时信息处理系统中，计算机能及时接收从远程终端发来的服务请求，根据用户提出的问题对信息进行检索和处理，并在很短时间内对用户做出正确回答，如机票订购系统，情报检索系统等。

（2）实时任务的类型

①按任务执行时是否呈现周期性来划分：分为周期性实时任务和非周期性实时任务。

②根据对截止时间的要求来划分：分为硬实时任务和软实时任务。

（3）实时系统与分时系统的比较

①多路性；　②独立性；　③及时性；　④交互性；　⑤可靠性

实时操作系统的主要特点是响应及时和可靠性高。系统必须保证对实时信息的分析和处理的速度要快，而且系统本身要安全可靠，因为在生产过程的实时控制、航空订票等实时事

务系统，信息处理的延误或丢失往往会带来不堪设想的后果。

随着计算机硬件及其应用的不断发展，操作系统的类型也逐渐多样化，如何对这些操作系统进行分类取决于分类的方法，即所依据的标准。下面列出了三种分类方法。

（1）按用户数目分为单用户操作系统和多用户操作系统。其中，单用户操作系统又分为单任务操作系统和多任务操作系统。

（2）按硬件结构分为单 CPU 操作系统、多 CPU 操作系统、网络操作系统、分布式操作系统和多媒体操作系统。

（3）按使用环境分为批处理操作系统、分时操作系统和实时操作系统。这是最常用的一种分类方法。

批处理操作系统、分时操作系统和实时操作系统是三种基本的操作系统类型。如果一个操作系统兼有批处理、分时处理和实时处理系统三者或其中两者的功能，那就形成了通用操作系统。

（三）操作系统的运行环境

操作系统的运行环境主要包括计算机系统的硬件环境和由其它系统软件形成的软件环境，以及操作系统和使用它的人之间的关系。

硬件环境主要包括中央处理器（CPU）、存储系统、中断机制、I/O 技术和时钟等方面。下面主要说明 CPU 状态和中断机制。

特权指令：只能由操作系统使用的指令。如：修改程序状态字、开关中断、置中断向量、启动设备执行 I/O 操作、设置硬件实时钟、停机等

非特权指令：特权指令之外的指令，这些指令的执行不影响其它用户以及系统状态.如算术运算指令、逻辑运算指令、取数存数指令、访管指令等

1．CPU 状态—管态和目态

计算机系统中，操作系统程序作为用户程序的管理者和控制者，享有用户程序所不能享有的某些特权，为避免错误地使用特权指令，将 CPU 的运行状态分为管态和目态。由程序状态(PSW)寄存器内的标志触发器来进行标识。

管态又称为系统态或核心态，操作系统程序在管态下运行，能执行包括特权指令在内的所有指令。

目态又称为用户态或常态，外层用户程序在目态下运行，不可执行特权指令。若出现特权指令、CPU 能识别出程序非法使用指令，形成一个程序性中断事件，中止程序的执行。

目态--管态

其转换的唯一途径是通过中断

管态--目态

可用设置 PSW(修改程序状态字)可实现

2．中断机制

(1)中断的定义：所谓中断是指系统发生某一事件后，CPU 暂停正在执行的程序转去执行处理该事件的程序过程，处理中断事件的程序称为中断处理程序，产生中断信号的那个部件称为中断源。**硬件的中断机构与处理这些中断的程序统称为中断系统。**

(2)中断的类型

不同的计算机系统．其产生中断的原因及其处理方式均不同，通常将系统内的所有中断分为若干类。

①根据中断信号的含义和功能分为以下五类；

机器故障中断：因机器发生错误(电源故障，内存读数错误等)而产生的中断，用以反映硬件故障，以便进入诊断程序。

I/O 中断：由输入/输出设备引起的中断，用以反映通道或外部设备工作状态。

外中断：由各种外部事件引起的中断，用以反映外部的要求。如时钟的定时中断，控制台发控制信息等。

程序性中断：因程序中错误使用指令或数据引起的中断，用以反映程序执行过程中发生的例外情况。如定点溢出，除数为 0，地址越界等。

访管中断：由于程序执行了“访管”指令（系统调用）而产生的中断，用于反映用户程序所请求操作系统为其完成某项工作。

②根据中断信号的来源分为两类；

中断，也称外中断，指来自 CPU 以外事件的中断，是与当前运行程序无关的暂停事件。对它的处理不必完全依赖当前程序的运行现场，具有较低的中断优先级，可被临时屏蔽。

异常，也称内中断或陷入，指源自 CPU 内部事件的中断，是与当前运行程序相关的暂停事件，对其处理要依赖于当前程序的运行现场，均具有较高的优先级，一旦出现应立即处理。

③根据是否是当前程序期望的分为两类：

强迫性中断： 正在运行的程序所不期望的，由于某种硬件故障或外部请求引起的

- 输入/输出(I/O)中断：主要来自外部设备通道
- 程序性中断：运行程序中本身的中断

  (如溢出,缺页中断,缺段中断,地址越界)
- 时钟中断
- 控制台中断
- 硬件故障

自愿性中断（访管中断）： 用户在程序中有意识安排的中断，是由于用户在编制程序时因为要求操作系统提供服务，有意使用“访管”指令（系统调用），使中断发生

- 执行 I/O，创建进程，分配内存
- 信号量操作，发送/接收消息

3.中断优先级与中断向量

中断优先级指中断装置响应中断的次序，是由硬件设计时固定的规定级别高的中断优先响应。一般情况下，优先级的高低顺序为：机器故障中断，访管中断，程序性中断，外部中断，输入输出中断。

中断屏蔽即禁止中断出现或响应中断，可以改变中断响应的顺序。

二、进程管理

一、考试大纲

（一）进程与线程：

1．进程概念

2．进程的状态与转换

3．进程控制

4．进程组织

5．进程通信

共享存储系统；消息传递系统；管道通信

6．线程概念与多线程模型

（二）处理机调度

1．调度的基本概念

2．调度时机、切换与过程

3．调度的基本准则

4．调度方式

5．典型调度算法

先来先服务调度算法；短作业（短任务、短进程、短线程）优先调度算法；时间片轮转调度算法；优先级调度算法；高响应比优先调度算法；多级反馈队列调度算法。

(三)进程同步

1．进程同步的基本概念

2．实现临界区互斥的基本方法

软件实现方法；硬件实现方法

3．信号量

4．管程

5．经典同步问题

生产者—消费者问题；读者—写者问题；哲学家进餐问题。

（四）死锁

1．死锁的概念

2．死锁处理策略

3．死锁预防

4．死锁避免

系统安全状态；银行家算法

5．死锁的检测和解除

二、知识点归纳

（一）进程与线程

1．进程概念

（1）前趋图

前驱图是一个有向无循环图，图中的每个结点可以表示一条语句、一个程序段或进程，结点间的有向边表示两个之间存在偏序或前趋关系“→”：

$\rightarrow=\{(P_i,P_j)|$ (P_i 必须在 P_j 开始执行之前完成}

若$(P_i,P_j) \in \rightarrow$，记为 $P_i \rightarrow P_j$，则称 P_i 是 P_j 的直接前趋，P_j 是 P_i 的直接后继。若存在一个

序列 $P_i \rightarrow P_j \rightarrow \cdots \rightarrow P_k$，则称 P_i 是 P_k 的前趋。在前趋图中，没有前趋的结点称为初始结点，没有后继的结点称为终止结点。

（2）程序的顺序执行

一个程序通常由若干个程序段所组成，它们必须按照某种先后次序来执行，仅当前一个操作执行完后，才能执行后继操作，这类计算过程就是程序的顺序执行过程。

程序顺序执行时有如下特征。

1）顺序性：处理机的操作严格按照程序所规定的顺序执行，即每一个操作必须在下一个操作开始之前结束。

2）封闭性：程序一旦开始运行，其执行结果不受外界因素影响，因为程序在运行时独占系统的各种资源，故这些资源的状态(除初始状态外)只有本程序才能改变。

3）可再现性：只要程序执行时的初始条件和执行环境相同，当程序重复执行时，都将获得相同的结果。

（3）程序的并发执行

程序的并发执行是指若干个程序(或程序段)同时在系统中运行，这些程序(或程序段)的执行在时间上是重叠的，即一个程序(或程序段)的执行尚未结束，另一个程序(或程序段)的执行已经开始。

程序并发执行时有如下特征。

1）间断性：“走走停停”，一个程序可能走到中途停下来，失去原有的时序关系；

2）失去封闭性：共享资源，受其他程序的控制逻辑的影响。如：一个程序写到存储器中的数据可能被另一个程序修改，失去原有的不变特征。

3）不可再现性：由于失去封闭性，外界环境在程序的两次执行期间发生变化，失去原有的可重复特征。

（4）进程的定义与特征

1）进程的定义

进程是具有独立功能的可并发执行的程序在一个数据集合上的运行过程，是系统进行资源分配和调度的独立单位。或者说，“进程”是进程实体的运行过程。

2）进程的特征

①动态性。进程是程序的一次执行过程，因此，动态性是进程最基本的特性。动态性还表现为：“它由创建而产生，由调度而执行，由得不到资源而暂停执行，以及由撤销而消亡。

②并发性。这是指多个进程实体同存于内存中，能在一段时间内同时运行。并发性是进程的重要特征。

③独立性。这是指进程实体是一个能独立运行的基本单位，同时也是系统中独立获得资源和独立调度的基本单位。

④异步性。这是指进程按各自独立的、不可预知的速度向前推进。

⑤结构特征。从结构上看，进程实体是由程序段、数据段及进程控制块三部分组成，有人把这三部分统称为“进程映像”。

2．进程的状态与转换

（1）进程有三种基本状态：

1）就绪状态。当进程已分配到除 CPU 以外的所有必要资源，只要获得 CPU，便可立即执行。

2）执行状态。进程已获得 CPU，正在执行，单处理机系统中，只有一个进程处于执行状态。

3）阻塞状态。进程不具备运行的条件，如申请的资源（除 CPU 外）未满足，等待某个

事件等。

（2）进程状态的转换

进程在运行期间不断地从一个状态转换到另一个状态，进程的各种调度状态依据一定的条件而发生变化，它可以多次处于就绪状态和执行状态，也可多次处于阻塞状态，但可能排在不同的阻塞队列。

进程的三种基本状态及其转换如图 2.1 所示。

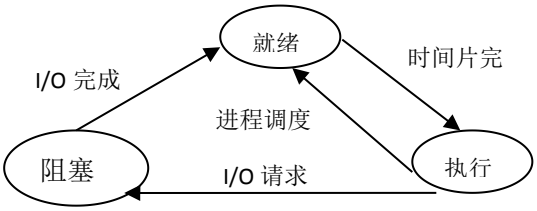

图 2.1　进程的三种基本状态及转换

（3）进程控制块

为了管理和控制进程的运行，操作系统为每个进程定义了一个数据结构——进程控制块(PCB)，用于记录进程的属性信息。**系统根据 PCB 感知进程的存在，PCB 是进程存在的惟一标志。**当创建一个进程时，系统为该进程建立一个 PCB；当进程执行时，系统通过 PCB 了解进程的现行状态信息；当进程结束时，系统收回其 PCB，该进程随之消亡。

一般来说，不同操作系统中的 PCB 所包含的内存多少会有些差异，但通常包括下面所列出的内容：

l）进程标识符：每个进程都有惟一的进程标识符，以区别于系统内部的其他进程。在进程创建时，由系统为进程分配惟一的进程标识号。

2）进程当前状态：说明进程的当前状态，以作为进程调度程序分配处理机的依据。

3）进程队列指针：用于记录 PCB 链表中下一个 PCB 的地址。为了查找方便，系统中的 PCB 可能组织成多个链表，如就绪队列、阻塞队列等。

4）程序开始地址：进程执行的开始地址。

5）进程优先级：反映进程要求 CPU 的紧迫程度。优先级高的进程可优先获得处理机。

6）CPU 现场保护区：当进程因某种原因释放处理机时，CPU 现场信息被保存在 PCB 的该区域中，以便在进程重新获得处理机后能继续执行。

7）通信信息：记录进程在执行过程中与别的进程所发生的信息交换情况。

8）家族联系：有的系统允许进程创建子进程，从而形成一个进程家族树。在 PCB 中必须指明本进程与家族的关系，如它的子进程与父进程的标识。

9）占有资源清单：列出进程所需资源及当前已分配资源清单。

3. 进程控制

进程控制的职责是对系统中的所有进程实施有效的管理，其功能包括进程的创建、进程的撤消、进程的阻塞与唤醒、进程的挂起与激活等。进程控制一般是由 OS 的内核来实现，OS 提供一组原语来实现。

所谓原语是一种特殊的系统功能调用，原语是由若干条机器指令构成的，它可以完成一个特定的功能，其特点是原语执行时不可被中断，所以原语操作具有原子性。

（1）进程创建

进程创建是由创建原语实现的，当需要时，进程就可以建立一个新进程。被创建的进程称为子进程，建立进程的进程称为父进程，而子进程又可以根据需要创建自己的子进程，从而构成了进程图。

引起创建进程的典型事件有分时系统中的用户登录、批处理系统中的作业调度、系统提供服务、应用进程本身的应用请求等。

一旦操作系统发现了要求创建新进程的事件后，便调用进程创建原语 creat()，按下述步骤创建一新进程。

①申请空白 PCB；

②为新进程分配资源；

③初始化进程控制块；

④将新进程插入就绪队列。

（2）进程的终止

当进程完成任务或者遇到异常情况和外界干预需要结束时，应通过调用进程终止原语来终止进程。终止进程的实质是回收 PCB。具体回收过程是：

①根据被终止进程的标识符从 PCB 集合中检索出该进程的 PCB，从中读出该进程的状态。

②若被终止进程正处于执行状态，应立即终止该进程的执行并设置调度标志为真，用于指示该进程被终止后应重新进行调度，选择一新进程，把处理机分配给它。

③若该进程还有子孙进程，还应将其所有子孙进程予以终止，以防它们成为不可控的。

④将该进程所拥有的全部资源，或者归还其父进程或者归还给系统。

⑤将被终止进程的 PCB 从所在队列中移出，等待其他程序来搜集信息。

（3）进程阻塞与唤醒

当一个进程期待的某一事件尚未出现时，该进样调用阻塞原语 block()将自己阻塞起来。阻塞原语的主要操作过程如下：在阻塞一个进程时，由于该进程正处于执行状态，故应中断处理机，保存该进程的 CPU 现场，停止运行该进程，然后将该进程插入到等待该事件的等待队列中，再从就绪队列中选择一个新的进程投入运行。

对处于阻塞状态的进程，当该进程期待的事件出现时，由发现者进程调用唤醒原语 wakeup()将阻塞的进程唤醒，使其进入就绪状态。唤醒原语的主要操作如下：将被唤醒进程从相应的等待队列中移出，将状态改为就绪并插入相应的就绪队列。

（4）进程的挂起与激活

当出现引起进程挂起的事件时，系统利用挂起原语 suspend()将指定进程或处于阻塞的进程挂起。挂起原语的执行：检查被挂起进程的状态，若处于活动就绪，则改为静止就绪，若处于活动阻塞，则改为静止阻塞，将该进程 PCB 复制到内存指定区域，若挂起的进程正在执行，则重新进行进程调度。

当发生激活进程的事件时，系统利用激活原语 active()将指定进程激活。激活原语先将进程从外存调入内存，检查该进程的状态，若处于静止就绪，则改为活动就绪，若处于静止阻塞，则改为活动阻塞。若采用抢占调度策略，则新进程进入就绪队列时，检查是否要重新进行进程调度。

4．进程组织

在一个系统中，通常存在着许多进程，它们有的处于就绪状态，有的处于阻塞状态，而且阻塞的原因各不相同，为了调度和管理进程方便起见，需要将各进程的进程控制块用适当的方法组织起来。目前常用的组织方式有链接方式和索引方式两种。

（1）链接方式：链接方式是将具有同一状态的 PCB，用其中的链接字链接成一个队列，多个状态对应多个不同的链表，如就绪链表、阻塞链表和空白链表等。对其中的就绪队列常按进程优先级的高低排列，把优先级高的进程的 PCB 排在队列前面。此外，可根据阻塞原因的不同而把处于阻塞状态的进程的 PCB，排成等待 I/O 操作完成的队列和等待分配内存的队列等。如图 2.2 所示。

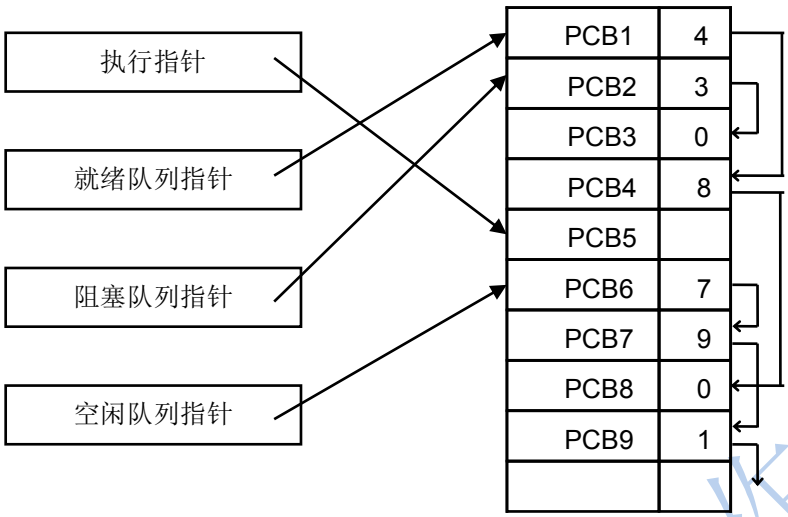

图2.2　链接方式

（2）索引方式：索引方式是系统根据所有进程的状态建立几张索引表，将同一状态的进程归入一个索引表，并把各索引表在内存的首地址记录在内存的一些专用单元中，再由索引指向相应的进程控制块，多个状态对应多个不同的索引表，如就绪索引表和阻塞索引表等。如图 2.3 所示。

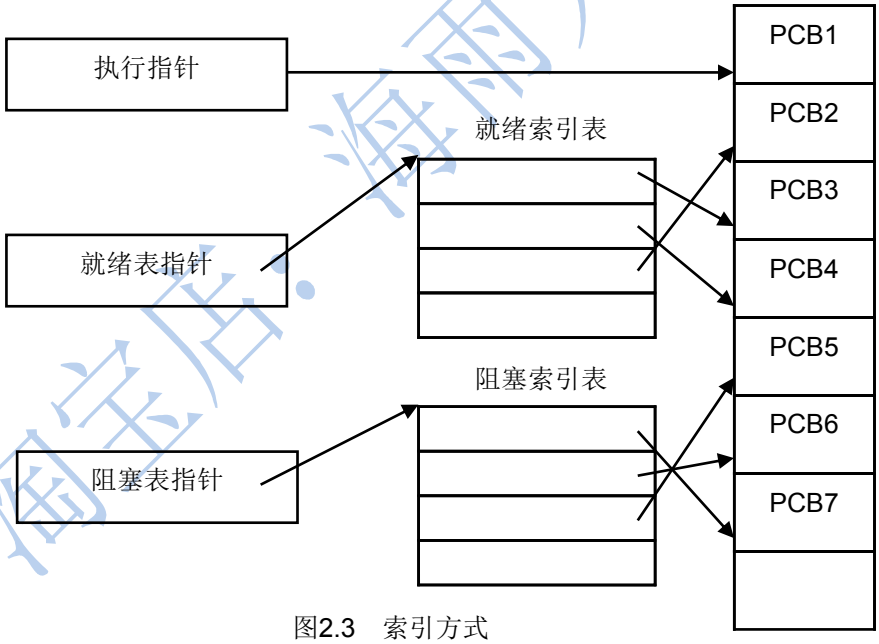

图2.3　索引方式

5．进程通信

进程通信是指进程之间的信息交换。进程互斥与同步就是一种进程间的通信方式。**由于进程互斥与同步交换的信息量较少且效率较低，因此称这两种进程通信方式为低级进程通信方式，相应地也将 wait、signal 原语（P、V 原语）称为两条低级进程通信原语。**

下面介绍的进程通信为高级进程通信方式。所谓高级进程通信方式是指进程之间以较高

的效率传送大量数据。

（1）进程通信的类型

目前，高级进程通信方式可分为 3 大类：共享存储器系统、消息传递系统以及管道通信系统。

1）共享存储器系统

为了传输大量数据，在存储器中划出一块共享存储区，诸进程可通过对共享存储区进行读或写来实现通信。进程在通信前，应向系统申请建立一个共享存储区，并指定该共享存储区的关键字；若该共享存储区已经建立，则将该共享存储区的描述符返回给申请者。然后，申请者把获得的共享存储区附接到本进程的地址空间上；这样，进程便可以像读、写普通存储器一样读、写共享存储区。

2）消息传递系统

在消息传递系统中，进程间的数据交换以消息为单位，程序员直接利用系统提供的一组通信命令(原语)来实现通信。操作系统隐藏了通信的实现细节，大大简化了通信程序编制的复杂性，因而获得了广泛的应用。消息传递系统因其实现方式不同可分为以下几种。

①直接通信方式：发送进程直接把消息发送给接收进程，并将它挂在接收进程的消息缓冲队列上，接收进程从消息缓冲队列中取得消息。

②间接通信方式：发送进程把消息发送到某个中间实体中，接收进程从中取得消息。这种中间实体一般称为信箱，故这种通信方式也称为信箱通信方式。这种通信方式被广泛应用于计算机网络中，现在称为电子邮件系统。

3）管道通信

管道是用于连接读进程和写进程以实现它们之间通信的共享文件，向管道提供输入的发送进程（即写进程）以字符流形式将大量的数据送入管道，而接收管道输出的接收进程(即读进程)可以从管道中接收数据。

（2）消息缓冲通信

消息缓冲通信是一种直接通传方式，即发送进程直接发送一个消息给接收进程。所谓消息是指一组信息，通常由消息头和消息正文组成。在通信时，发送进程采用发送原语向接收进程发送一个消息，而接收进程则采用接收原语接收来自发送进程的一个消息。发送原语的主要工作是申请分配一个消息缓冲区，然后将消息正文传送到该缓冲区中，并向缓冲区中填写消息头，再将该消息缓冲区挂到接收进程的消息链上。接收原语的主要工作是把消息链上的消息逐个读入到接收进程的接收区中并进行处理。

（3）信箱通信

信箱通信是一种间接通信方式。信箱是一种数据结构，其中存放信件。当一个进程(发送进程)要与另一个进程(接收进程)通信时，可由发送进程创建一个链接两进程的信箱，通信时发送进程只须把它的信件投入信箱，接收进程就可以在任何时候取走信件而不会丢失。

信箱逻辑上分成信箱头和信箱体两部分。信箱头中存放有关信箱的描述。信箱体由若干格子组成，每格存放一信件，格子的数目和大小在创建信箱时确定。信件的传递可以是单向的，也可以是双向的。

在单向信箱通信方式中，只要信箱中有空格，发送进程便可向信箱中投递信件，若所有格子都已装满，则发送进程或者等待，或者继续执行，待有空格子时再发送。类似地，只要格子中装有信件，接收进程便能取出一信件。若信箱为空，接收进程或者等待，或者继续执行。

在双向通信方式中，信箱中既有发送进程发出的信件，也有接收进程的回答信件。由于发送进程和接收进程均以各自独立的速度向前推进，当发送进程发送信件的速度超过接收进程的接收速度时，会产生上溢(信箱满)。反之，会产生下溢，即接收进程向空信箱索取信件。

这就需要在两个进程之间进行同步控制，当信箱满时发送进程应等待，直至信箱有空格子时再发送；对接收进程，当信箱空时，它也应等待，直至信箱中有信件时再接收。

信箱通信方式中也使用原语操作，如创建信箱原语、撤消信箱原语、发送与接收原语等。另外，在许多时候，存在着多个发送进程和多个接收进程共享信箱的情况。

6．线程概念与多线程模型

(1)线程的基本概念

自从 20 世纪 60 年代提出进程的概念后，在操作系统中一直都是以进程作为独立运行的基本单位，在操作系统中引入进程的目的是为了使多个程序并发执行，以改善资源利用率及提高系统吞吐量；到 80 年代中期，人们提出了比进程更小的独立运行的基本单位——线程，**在操作系统中引入线程，是为了减少程序并发执行时所付出的时空开销，进一步提高系统内程序并发执行的程度，提高系统吞吐量，使操作系统具有更好的并发性。**

线程的定义有以下几种提法，这些提法可以相互补充。

（1）线程是进程内的一个执行单元。

（2）线程是进程内的一个可调度实体。

（3）线程是程序(或进程)中相对独立的一个控制流序列。

（4）线程是执行的上下文，其含义是执行的现场数据和其他调度所需的信息。

（5）线程是进程内一个相对独立的、可调度的执行单元。

线程是进程的一个实体，是被系统独立调度和分派的基本单位，线程自己基本上不拥有资源，只拥有一点在运行时必不可少的资源(如程序计数器、一组寄存器和栈)，但它可以与同属一个进程的其他线程共享进程拥有的全部资源。一个线程可以创建和撤销另一个线程；同一个进程中的多个线程之间可以并发执行。

(2)线程的实现机制

对于通常的进程，不论是系统进程还是用户进程，在进行切换时都要依赖于内核中进程的调度。因此，我们说，不论什么进程都是与内核有关的，是在内核支持下进行切换的，对于线程来说，则可分为两类：一类是内核支持线程，它们是依赖于内核的。即无论是在用户进程中的线程，还是系统进程中的线程，它们的创建、撤销和切换都由内核实现。在内核中保留了一张线程控制块，内核根据该控制块而感知该线程的存在并对线程进行控制。另一类是用户级线程。它仅存在于用户级中，对于这种线程的创建、撤销和切换，都不利用系统调用来实现，由应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制。因而这种线程与内核无关。相应地，内核也并不知道有用户级线程的存在。

这两种线程各有优缺点，因此，如果将两种方法结合起来，则可得到两者的全部优点，将两种方法结合起来的系统称之为多线程的操作系统。内核支持多线程的建立、调度与管理。同时，系统中又提供使用线程库的便利，允许用户应用程序建立、调度和管理用户级的线程。

（二）处理机调度

1．调度的基本概念

在多道程序环境下，进程数目往往多于处理机数目，这就要求系统能按某种算法，动态地把处理机分配给就绪队列中的一个进程，分配处理机的任务是由处理机调度程序完成的。一个作业从提交开始直至完成，往往经历下述三级调度：

(1)高级调度

高级调度又称宏观调度、作业调度或长程调度，其主要任务是按一定的原则从外存上处于后备状态的作业中选择一个或多个作业，给它们分配内存、输入输出设备等必要的资源，并建立相应的进程，再将新创建的进程排在就绪队列上，准备执行。高级调度的运行频率较低，通常为几分钟一次。

（2）低级调度

低级调度又称进程调度、短程调度或微观调度，其主要任务是按照某种策略和方法从就绪队列中选取一个进程，将处理机分配给它。进程调度的运行频率很高，一般几十毫秒要运行一次。

（3）中级调度

中级调度又称中程调度或交换调度，引入中级调度的目的是为了提高内存利用率和系统吞吐量。其主要任务是按照给定的原则和策略，将处于外存对换区中的重又具备运行条件的进程调入内存，或将处于内存的暂时不能运行的进程交换到外存对换区。

2．调度时机、切换与过程

（1）引起进程调度的因素主要有：

①正在执行的进程执行完毕，或因发生某事件而不能再继续执行；

②执行中的进程因提出 I/O 请求而暂停执行；

③在进程通信或同步过程中执行了某种原语操作，如 wait 原语、block 原语、wakeup 原语等；

④当进程执行完系统调用功能从核心态返回用户态时，发现调度标志已设置，即系统中某进程的优先级已高于当前执行进程的优先级，系统会调用优先级高的进程执行。

⑤各进程按时间片运行，当一个时间片用完后，便停止该进程的执行而重新进行调度。

（2）进程切换过程

从进程 A 切换到进程 B 的过程为：当系统正在用户态执行进程 A 的代码，若此时发生进程切换，首先通过时钟中断或系统调用进入 OS 核心，然后保存进程 A 的上下文，再恢复进程 B 的上下文（CPU 寄存器和一些表格的当前指针），最后转到用户态执行进程 B 的代码。

3．调度的基本准则

选择调度方式和算法的准则，有的是面向用户的，有的是面向系统的。

(1)面向用户的准则

①周转时间短。通常把周转时间的长短作为评价批处理系统的性能。周转时间是指从作业被提交给系统开始，直到作业完成为止的这段时间间隔。

设 T_i 是第 i 作业的周转时间，则平均周转时间描述为：$T=\frac{1}{n}[\sum_{i=1}^{n}T_i]$。作业的周转时间 T 与系统为它提供服务的时间 T_s 之比，即 $W=T/T_s$，称为带权周转时间，平均带权周转时间表示为：$W=\frac{1}{n}[\sum_{i=1}^{n}\frac{T_i}{T_{si}}]$。

②响应时间快。常把响应时间的长短作为评价分时系统的性能。所谓响应时间，是从用户通过键盘提交一个请求开始，直至系统首次产生响应为止的时间。

③截止时间的保证。这是评价实时系统性能的重要指标。所谓截止时间，是指某任务必须开始执行的最迟时间，或必须完成的最迟时间。

④优先权准则。在批处理、分时和实时系统中，可遵循优先权准则，以便让某些紧急的作业能得到及时处理，在较严格的场合，还需选择抢占式调度方式。

4．调度方式

所谓进程调度方式是指当某一个进程正在处理机上执行时，若有某个更为重要或紧迫的进程需要进行处理，即有优先级更高的进程进入就绪队列，此时如何分配处理机。通常有两种进程调度方式。

（1）抢占方式。抢占方式又称剥夺方式、可剥夺方式。这种调度方式是指当一个进程

正在处理机上执行时，若有某个更为重要或紧迫的进程需要使用处理机，则立即暂停正在执行的进程，将处理机分配给更重要或紧迫的进程。

（2）非抢占方式。非抢占方式又称非剥夺方式、不剥夺方式、不可抢占力式。这种调度方式是指当一个进程正在处理机上执行时，即使有某个更为重要或紧迫的进程进入就绪队列，仍然让正在执行的进程继续执行，直到该进程完成或发生某种事件而进入阻塞状态时，才把处理机分配给更为重要或紧迫的进程。

5．典型调度算法

调度算法是指根据系统的资源分配策略所规定的资源分配算法。对于不同的系统和系统目标，通常采用不同的调度算法。典型的调度算法有以下几种：

（1）先来先服务调度算法

先来先服务（FCFS）调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度，在作业调度中，每次调度都从后备作业中，选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源，创建进程；用于进程调度时，是按照进程进入就绪队列的先后次序来分配处理机。先来先服务算法采用非剥夺的调度方式，即一旦一个进程占有处理机，它就一直运行下去，直到该进程完成其工作或因等待某一事件而不能继续执行时才释放处理机。

（2）短作业（进程）优先调度算法

可以分别用于作业调度和进程调度，短作业优先（SJF）调度算法，是从后备对列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先（SPF）调度算法，是从就绪队列中选出一估计运行时间最短的进程，将处理机分配给它。

（3）时间片轮转调度算法

在时间片轮转调度算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择队列中的第一个进程执行，并规定执行一定时间，该时间称为时间片。当该进程用完这一时间片时，系统将它送至就绪队列队尾，再把处理机分配给下一个就绪进程。这样，处于就绪队列中的进程，就可以依次轮流地获得一个时间片的处理时间，然后回到队列尾部，如此不断循环，直至完成为止。

（4）优先级调度算法

优先级调度算法是一种常用的进程调度算法，其基本思想是把处理机分配给优先级最高的进程，该算法的核心问题是如何确定进程的优先级。

进程的优先级用于表示进程的重要性及运行的优先性。进程优先级通常分为两种：静态优先级和动态优先级。

静态优先级是在创建进程时确定的，确定之后在整个进程运行期间不再改变。

动态优先级是指在创建进程时，根据进程的特点及相关情况确定一个优先级，在进程运行过程中再根据情况的变化调整优先级。

基于优先级的调度算法还可按调度方式不同分为非剥夺优先级调度算法和可剥夺优先级调度算法。

非剥夺优先级调度算法的实现思想是系统一旦将处理机分配给就绪队列中优先级最高的进程后，该进程便一直运行下去，直到由于其自身的原因(任务完成或等待事件)主动让出处理机时，才将处理机分配给另一个更高优先级的进程。

可剥夺优先级调度算法的实现思想是将处理机分配给优先级最高的进程，使之运行。在进程运行过程中，一旦出现了另一个优先级更高的进程时，进程调度程序就停止原运行进程，而将处理机分配给新出现的高优先级进程。

（5）高响应比优先调度算法

此算法是先来先服务算法和短作业优先算法的折衷，为每个作业引入一个动态优先权

（响应比），使作业的优先级随着等待时间的增加而提高，响应比的计算公式为：

响应比=（等待时间+要求服务时间）/要求服务时间

（4）多级反馈队列调度算法

多级反馈队列调度算法可以满足各种类型进程的需要，是目前公认的一种较好的调度算法，实现思想如下：

第一，设置多个就绪队列，并为每个队列赋予不同的优先级。第 1 个队列优先级最高，第 2 个队列次之，其余队列的优先级逐个降低。每个队列中进程执行的时间片大小也各不相同，进程所在队列的优先级越高，其相应的时间片就越短。通常，第 *i*+1 个队列的时间片是第 *i* 个队列时间片的两倍。

第二，当一个新进程进入系统时，首先将它放入第一个队列的末尾，按先来先服务的原则排队等待调度。当轮到该进程执行时，如能在此时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二个队列末尾，再同样地按照先来先服务原则等待调度执行；如果它在第二个队列个运行一个时间片后仍未完成，再以同样方法转入第 3 个队列。如此下去，最后一个队列中使用时间片轮转调度算法。

第三，仅当第一个队列空闲时，调度程序才调度第二个队列中的进程运行；仅当第 1 至(*i*-1)个队列均为空时，才会调度第 *i* 个队列中的进程运行。当处理机正在为第 *i* 个队列中的某进程服务时，若又有新进程进入优先级较高的队列中，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在执行的进程放回第 *i* 个队列末尾，重新将处理机分配给新进程。

(三)进程同步

1．进程同步的基本概念

进程同步的主要任务：使并发执行的诸进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性。

（1）进程间的两种制约关系

①间接相互制约关系。源于资源共享，同处于一个系统中的进程，必然共享某种系统资源，如共享 CPU、共享 I/O 设备等，如两个进程 A 和 B，若系统已将惟一的打印机分配给 B，此时如果 A 进程提出打印请求时，则进程 A 只能阻塞；一旦 B 将打印机释放，才能使 A 进程由阻塞改为就绪状态。此时进程同步的主要任务是保证诸进程能互斥地访问临界资源，为此，系统中的资源应不允许用户进程直接使用，而应有系统统一分配。

②直接相互制约关系。源于进程合作，此时进程同步的主要任务是保证相互合作的诸进程在执行次序上的协调，不会出现与时间有关的差错。

（2）临界资源和临界区

在计算机中有许多资源一次只能允许一个进程使用，我们把一次仅允许一个进程使用的资源称为临界资源。如打印机和一些共享变量。

进程中访问临界资源的那段代码称为临界区。

（3）同步机制应遵循的准则

①空闲让进。当没有进程处于临界区时，可以允许一个请求进入临界区的进程立即进入自己的临界区。

②忙则等待。当已有进程进入自己的临界区时，意味着相应的临界资源正被访问，因而所有其他试图进入临界区的进程必须等待，以保证诸进程互斥地访问临界资源。

③有限等待。对要求访问临界资源的进程，应保证该进程能在有效时间内进入自己的临界区，以免陷入“死等”状态。

④让权等待。当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”。

2．实现临界区互斥的基本方法

临界区互斥的实现既可以用软件的方法，也可以用硬件的方法。下面对这两种方法进行介绍。

（1）软件方法

用软件方法解决互斥和同步问题较复杂，为了说明将介绍的正确算法中某些变量设置的必要性，首先给出两个不正确算法。

1）不正确算法 1

两个进程(P0，P1)共享一个公用变量 turn，turn=i 时进程 Pi 可进入临界区。进程 Pi 的结构如下：（turn 初值为 0 或 1 无关紧要。i 和 j 分别为 0 或 1，i＝1—j。

```
repeat
    while turn＜＞i do skip;
    临界区；
    turn：=j;
    其他代码；
until   false;
```

该算法满足互斥性要求，但它要求执行临界区的进程必须严格地交替进行，若 turn=0 且 P1 希望进入临界区，尽管 P0 已用完并退出临界区，该算法也不能让 P1 进入其临界区运行。

2）不正确算法 2

不正确算法 1 的问题在于，它只记住了哪个进程允许进入它的临界区，但没有记录每个进程的状态。为了避免这种情况，可以用下面的数组来代替变量 turn:

```
var flag: array[0..1] of boolean;
```

该数组的每个元素的初值均为 false。若且 flag[i]为 turn，则表示进程 Pi 正在其临界区执行。进程 Pi 的一般结构为：

```
repeat
while flag[j] do skip；
flag[i]:＝true；
临界区；
flag[i]:＝false；
剩余区；
untll   false;
```

这里，首先查看是否有另一进程在其临界区中，若有则等待；否则置本进程的 flag 为 true，并进入临界区。当离开临界区时，又置其 flag 为 false，以允许另一进程进入其临界区。然而该算法并未保证一次只有一个进程在其临界区内执行。例如考虑以下执行序列：

T0：P0 进入 while 语句并发现 flag[1]＝false；

T1：P1 进入 while 语句并发现 flag[0]＝false；

T2：P1 置 flag[1]为 true 并进入临界区；

T3：P0 置 flag[0]为 true 并进入临界区进入。

于是，P0 与 P1 都进入了临界区，从而违反了互斥的要求。

3）正确算法（只针对双进程)

该算法将上述两种不正确算法中的有利思想结合在一起。进程间共享两个共用变量：

```
var flag：array[0..1] of boolean；turn：0..1；
```

最初 flag[0]＝flag[1]＝false，turn 的初值是无关紧要的。进程 Pi 的一般结构为：

```
repeat
```

flag[i]:=true;turn:=j;;
while(flag[j] and turn=j) do skip;
临界区；
flag[i]:= false;
剩余区；
until false；

(2)硬件方法解决互斥问题：

1）中断屏蔽方法

当一个进程正在使用处理器执行它的临界区代码时，要防止其他进程再进入其临界区访问的最简单方法是禁止一切中断发生，或称之为屏蔽中断、关中断。因为 CPU 只在发生中断时引起进程切换，屏蔽中断就能保证当前运行进程将临界区代码顺利地执行完，从而保证了互斥的正确实现，然后再开中断。下面是用中断屏蔽方法实现互斥的典型模式：

…
关中断；
临界区；
开中断；
…

并关中断方法来实现过程间互斥，既简单又有效。但它存在一些不足，如该法限制了处理器交替执行程序的能力，其执行的效率将会明显降低；对于核心来说，当它在执行更新变量或列表的几条指令期间将中断关掉是很方便的，但将关中断的权力交给用户进程则很不明智，若一个进程关中断之后不再开中断，则系统可能会因此终止。

2）硬件指令方法

许多计算机中都提供了专门的硬件指令，完成对一个字或字节中的内容进行检查和修改，或交换两个字或字节的内容的功能。用一条指令来完成对共享变量的检查和修改两个功能，这样中断不可能发生，所以不会影响共享变量数据的完整性。

实现这种功能的硬件指令有两种：

①TS(Test-and-set)指令。该指令的功能是读出指定标志后把该标志设置为真。

```
boolean    TS(boolean    *lock)
{
boolean    old；
old=*lock；
*lock=true；
return     old;
}
```

为了实现多个进程对临界资源的互斥访问，可以为每个临界资源设置一个共享的布尔变量 lock，表示资源的两种状态：true 表示正被占用，false 表示空闲，初值为 false。在进程访问临界资源之前，利用 TS 检查和修改标志 lock；若有进程在临界区，则重复检查，直到其他进程退出。利用 TS 指令实现进程互斥的算法可描述为：

While　TS(＆lock)；
进程的临界区代码 CS；
Lock=false；
进程的其他代码；

②swap 指令。该指令的功能是交换两个字或字节的内容，描述如下：

```
Swap(boolean  *a，boolean  *b)
{
Boolean  temp；
temp＝*a；
a=*b；
*b=temp；
}
```

利用 swap 指令实现进程互斥时，应为每个临界资源设置一个共享布尔变量 lock，初值为 false；在每个进程中再设置一个局部布尔变量 key，用于与 lock 交换传息。在进入临界区之前利用 swap 指令交换 lock 与 key 的内容，然后检查 key 的状态；若有进程在临界区时，重复交换和检查过程，直到其他进程退出。利用 swap 指令实现进程互斥的算法描述为：

```
key＝true；
while  (key!＝false)  swap(＆lock，&key)；
进程的临界区代码 CS；
lock＝false；
进程的其他代码；
```

3．信号量

荷兰学者 Dijkstral965 年提出的信号量机制是一种卓有成效的进程同步工具。在长期且广泛的应用中，信号量机制又得到了很大的发展，它从整型信号量经记录型信号量，进而发展为“信号量集”机制。

（1）整型信号量机制

整型信号是一个整型量，除初始化外，仅能通过两个标准的原子操作 wait(s)和 signal(s)来访问。这两个操作很长时间以来一直被分别称为 P、V 操作。Wait 和 signal 操作可描述为：

```
Wait(s)：while (s≤0)
          no-op;
          s--;
    signal(s)：  s++;
```

整型信号量的主要问题是，只要 s≤0，wait 操作就会不断地测试，因而没有做到“让权等待”。

（2）记录型信号量机制

在记录型信号量中，除了需要一个用于代表资源数目的整型变量 value 外，还应增加一个进程链表 L，用于链接上述的所有等待进程。

```
typedef  struct  {
    int value；
    1ist  of  proces s  *L；
}semaphore；
```

相应的 wait 和 signal 操作可描述为：

```
void  wait(static  semaphore  s)
{     s.value--；
    if  (s.value＜0)
    block(s.L)；
}
void  signal(static  semaphore  s)
{     s.value++；
```

```
        if   (s.value＜=0)
        wakeup(s.L);
    }
```

在记录型信号量机制中，s.value 的初值表示系统中某类资源的数目，因而又称为资源信号量，每次的 wait 操作意味着进程请求一个单位的资源，因此描述为 s.value--；当 s.value＜0 时，表示资源已分配完毕，因而进程调用 block 原语，进行自我阻塞，放弃处理机并插入到信号量链表 s.L 上。可见，该机制遵循了让权等待准则。**此时 s.value 的绝对值表示在该信号量链表中已阻塞进程的数目。**每次 signal 操作，表示执行进程释放一个单位资源．故 s.value++操作表示资源数目加 1。若加 l 后仍是 s.value＜=0，则表示在该信号链表中仍有等待该资源的进程被阻塞，故还应调用 wakeup 唤醒进程。如果 s.value 的初值等于 l，则此时的信号量转化为互斥信号量。

(3)信号量的应用

1）利用信号量实现互斥

为使多个进程能互斥地访问某个临界资源，只需为该资源设置一个互斥信号量 mutex，并将其初值设置为 1，然后将访问该资源的临界区置于 wait(mutex)和 signal(mutex)之间。下面的算法描述了如何利用信号量实现进程 P1 和 P2 的互斥，信号量 mutex 的初值为 1。

```
P1 进程                          P2 进程
    …                               …
wait(mutex);                    wait(mutex);
critical   section;             critical   section;
signal(mutex);                  signal(mutex);
    …                               …
```

2）利用信号量实现前趋关系

若干进程为了完成一个共同任务而并发执行。然而，这些并发进程之间根据逻辑上的需要，有的操作可以没有时间上的先后次序，但有的操作有一定的先后次序。我们可以用前面介绍的前趋图来描述进程在执行次序上的先后关系。

例如：S1、S2、S3、S4 为一组合作进程，其前驱图如图 2.4 所示，试用 wait、signal 操作完成这 4 个进程的同步。

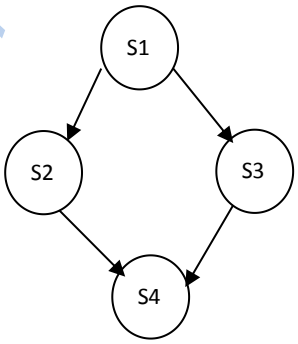

图 2.4　四个合作进程的前趋图

图 2.4 说明任务启动后 S1 先执行，当 S1 结束后，S2、S3 可以开始执行。S2、S3 完成后，S4 才能开始执行。为了确保这一执行顺序，设四个同步信号量 f1、f2、f3、f4，其初值均为 0，这四个进程的同步描述如下：

```
semaphore    f1=0        /*表示进程 S2 是否可以开始执行*/
semaphore    f2=0        /*表示进程 S3 是否可以开始执行*/
```

```
semaphore    f3=0        /*表示进程 S4 是否可以开始执行*/
semaphore    f4=0        /*表示进程 S4 是否可以开始执行*/
main()
{
cobegin
        S1();
        S2();
        S3();
        S4();
coend
}
S1()
{
        ⋮           /* “⋮”表示进程中的程序代码，下同*/
        Signal(f1);
        Signal(f2);
}
S2()
{
        Wait(f1);
          ⋮
        Signal(f3);
}
S3()
{
        Wait(f2);
          ⋮
        Signal(f4);
}
S4()
{
        Wait(f3);
         Wait(f4);
          ⋮
}
```

4．管程

虽然信号量机制是一种既方便又有效的进程同步机制，但每个要访问临界资源的进程都必须自备同步操作 wait(s)和 signal(s)。这就使大量的同步操作分散在各个进程中。这不仅给系统的管理带来麻烦，而且还会因同步操作的使用不当而导致系统死锁。这样，在解决上述问题的过程中，便产生了一种新的同步工具——管程。

(1)管程的基本概念

一个管程定义了一个数据结构和能为并发进程所执行(在该数据结构上)的一组操作，这组操作能同步进程和改变管程中的数据。由定义可知，**管程由三部分组成：①局部于管程的共享变量说明；②对该数据结构进行操作的一组过程；③对局部于管程的数据设置初值**

的语句。此外，过需为该管程赋予一个名字。

管程具有以下特点：

1）局部于管程的数据结构，仅能被局部于管程的过程所访问。

2）一个进程只有通过调用管程内的过程才能进入管程访问共享数据。

3）任一时刻，最多只能有一个进程在管程中执行。即进程互斥地通过调用内部过程进入管程，当某进程在管程内执行时，其他想进入管程的进程必须等待。

(2)条件变量

在利用管程实现进程同步时，必须设置两个同步操作原语 wait 和 signal。当某进程通过管程请求临界资源而未能满足时，管程便调用 wait 原语使该进程等待，并将它排在等待队列上。仅当另一进程访问完并释放之后，管程又调用 signal 原语唤醒等待队列中的队首进程。

通常，等待的原因可有多个，为了区别它们，引入了条件变量 condition。管程中对每个条件变量都需予以说明，其形式为：condition x，y;。该变量应置于 wait 和盛 signal 之前，即可表示为 x. wait 和 x. signal。

x. signal 操作的作用是重新启动一个被阻塞的进程，如果没有进程被阻塞，则 x.signal 操作不产生任何后果，这与信号量机制中的 signal 操作不同。

(3)利用管程解决生产者。消费者问题

利用管程方式来解决生产者—消费者问题时，首先为它们建立一个管程，命名为 Producer-Consumer，描述如下：

```
monitor    producer—consume    {
    int    in, out, count;
    item    buffer[n];
    condition    notfull, notempty;
    entry    put(item)
   {
   if (count>=n)
    notfull.wait;
    buffer(in)=nextp;
    in=(in+1)    mod    n;
    count++;
    if notempty.queue
    notempty.signal;
   }
    entry get(item)
    {
    if (count<=0)
    notempty.wait;
   nextc=buffer(out);
   out:=(out+1)    mod    n;
   count--;
   If notfull.queue
    notfull.signal;
   }
    {in=out=0;
    count=0;
```

```
 }
}
```

利用管程解决生产者—消费者问题时，生产者—消费者可描述如下：

```
producer()
{    while(1)
   {   producer an item in nextp;
       producer—consume.put(item);
   }
}
Consumer()
{
      while(1)
    {   producer—consume.get(item);
        Cinsumer the item in nextc;
      }
}
Main()
{
   cobegin{
     producer();
     consumer();
   }
}
```

5．经典进程的同步问题

(1)生产者—消费者问题

生产者—消费者问题常发生在一组协同操作进程中，该问题是许多并行进程间存在的内在关系的一种抽象。通常可描述如下：有一个或多个生产者产生某种类型的产品(数据、记录、字符等)并放置在缓冲区中，生产者消费者共享一个有界缓冲池；有一个或多个消费者从缓冲中取产品，每次取一项，系统保证对缓冲区的重复操作，也就是说，在任何时候只有一个代理(生产者或消费者)可以访问缓冲区。

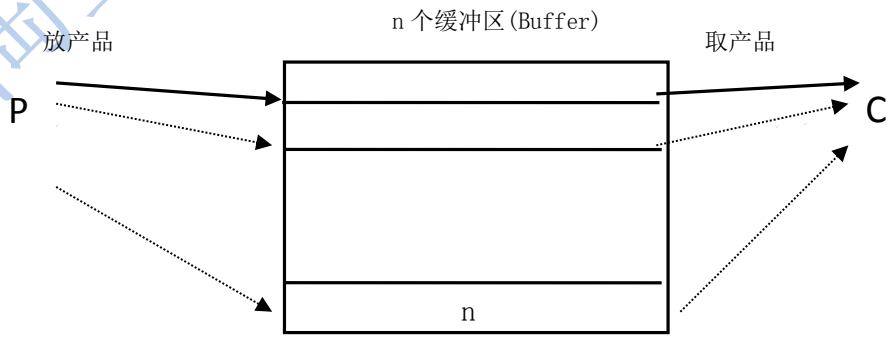

图 2.5 生产者—消费者问题

生产者和消费者的同步关系表现为：一旦缓冲池满时，生产者必须等待消费者提供空缓冲区单元，一旦缓冲池中所有单元全为空时，消费者必须等待生产者向缓冲区单元中放入产品。

对于所有的代理来说，缓冲区又是临界资源：任何一个进程在对某个缓冲区单元进行“存”或“取”操作时和其他进程互斥进行。

为解决生产者一消费者这一类问题，应该设置两个同步信号量，一个说明空缓冲单元的数目，用 empty 表示，其初值为有界缓冲区的大小 n；另一个说明满缓冲单元的数目(即产品数目)，用 full 表示，其初值为 0。因为有界缓冲区是一个临界资源，必须互斥使用，所以，需要设置一个互斥信号量 mutex，其初值为 l。生产者一消费者问题的同步描述如下：

```
semaphore   full=0;        /*满缓冲单元的数目*/
semaphore   empty=n;       /*空缓冲单元的数目*/
semaphore   mutex＝1;      /*对有界缓冲区进行操作的互斥信号量*/
item   buffer[n]
int    in=out=0
main()
{
Cobegin
Producer();
Consumer();
Coend
}
Producer(0
{
While(true)
{
Produce an item in nextp;
…
Wait(empty);
Wait(mutex);
Buffer[in]=nextp;
In=(in+1) mod n;
Signal(mutex);
Signal(full);
}
}
Consumer()
{
While(true)
  {
wait(full);
wait(mutex);
nextc:=buffer(out);
out:=(out+1) mod n;
```

```
signal(mutex);
signal(empty);
...
consume the item in nexc;
  }
}
```

程序中有一点需要读者注意，无论在生产者进程还是在消费者进程中，wait 操作的次序都不能颠倒，否则将可能造成死锁。

（2）读者—写者问题

一个数据文件或记录(统称数据对象)，可被多个进程共享。其中，有些进程要求读，而另一些进程对数据对象进行写或修改。我们把只要求读的进程称为“reader 进程”，其他进程称为“writer 进程”。允许多个 reader 进程同时读一个共享对象，但决不允许一个 writer 进程和其他 reader 进程或 writer 进程同时访问共享对象。所谓读者—写者问题是只保证一个 writer 进程必须与其他进程互斥地访问共享对象的同步问题。

为实现肥 reader 进程与 writer 进程读或写时的互斥，设置了一个互斥信号量 wmutex。再设置一个整型变量 readcount 以表示正在读的进程数目。由于只要有一个 read 进程在读，便不允许 writer 进程去写，因此，仅当 readcount==0 时，表示尚无 reader 进程在读时，reader 进程才需要执行 wait(wmutex)操作。若 wait(wmutex)操作成功，reader 进程便可去读，相应地 readcount++。同理，仅当 reader 进程在执行了 readcount--操作后其值为 0 时，才需执行 signal(wmutex)操作，以便让 writer 进程写。又因为 readcount 是一个可被多个 reader 进程访问的临界资源，因此，应为它设置一互斥信号量 rmutex。读者—写者问题可描述如下：

```
semaphore    rmutex =1;
semaphore    wmutex =1;
int          readcount=0;
  reader( )
{
    While(1)
    {
        wait(rmutex);
        if    (readcount= =0)
        wait(wmutex);
        readcount++;
        signal(rmutex);
           ...                                    ...
        perform read operation;
          ...                                    ...
        wait(rmutex);
        readcount--;
      if (readcount= =0)
       signal(wmutex);
        signal(rmutex);
      }
}
```

```
    writer()
{
  while(1)
  {
      wait(wmutex);
      perform write operation;
      signal(wmutex);
   }
}
main()
{
  cobegin
  {
    reader();
    writer();
  }
 coend
}
```

（3）哲学家进餐问题

哲学家进餐问题是典型的同步问题，该问题描述有五个哲学家围坐在一圆桌旁，圆桌上有五个碗和五只筷子，他们的生活方式是交替地进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有拿到两只筷子时才能进餐，进餐完毕，放下筷子继续思考。

经分析，桌子上的筷子是临界资源，为实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组，描述如下：

Semaphore chopstick[4]；

信号量初值为 1，第 i 位哲学家的活动描述为：

```
While(1)
  {   wait(chopstick[i]);
      wait(chopstick[(i+1) mod 5]);
      …
      eat;
      …
      signal(chopstick[i]);
      signal(chopstick[(i+1) mod 5]);
      …
      think;
      }
```

上述描述中，哲学家饥饿时，总是先拿他左边的筷子，成功后，再去拿他右边的筷子，成功后进餐，进餐毕，放下他左右两边的筷子，虽然上述方法可保证不会有相邻哲学家同时进餐，但有可能引起死锁，假若五位哲学家同时饥饿而各自拿起左边的筷子时，使五个信号量 chopstick 均为 0；当他们再试图去拿右边的筷子时，都将因五筷子可拿而无限等待。可用下述方法解决：

1）至多允许四个哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐。

2）奇数号哲学家先拿他左边的筷子，再去拿右边的筷子；偶数号哲学家先取他右边的筷子，再取他左边的筷子。

3）仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐，否则，一只筷子也不分给他。

下面给出第 2 种方法的描述：

Semaphore chopstick[4]；

信号量初值为 1，第 i 位哲学家的活动描述为：

```
While(1)
  { if (odd(i))
      {wait(chopstick[i]);
       wait(chopstick[(i+1) mod 5]); }
     else   {
       wait(chopstick[(i+1) mod 5]);
        wait(chopstick[i]);
       }
     …
     eat;
     …
     signal(chopstick[i]);
     signal(chopstick[(i+1) mod 5]);
     …
     think;
     }
```

（四）死锁

1．死锁的概念

（1）死锁的定义：死锁是指多个进程因竞争系统资源或相互通信而处于永久阻塞状态，若无外力作用，这些进程都将无法向前推进。

（2）死锁产生的原因和必要条件

死锁产止的原因是竞争资源和进程的推进顺序不当。如果系统中供多个进程共享的资源如打印机、公用队列等，其数目不足以满足诸进程的需要时，从而可能导致死锁的产生。虽然资源竞争可能导致死锁，但是资源竞争并不等于死锁，只有在进程运行过程中请求和释放资源的顺序不当时，才会导致死锁。

死锁产生的必要条件有以下 4 条：

（1）互斥条件：进程对所分配的资源进行排它性控制使用，即在一段时间内某资源只由一个进程占用。

（2）请求和保持条件：指进程已经保持了至少一个资源，但又提出了新的资源请求，在等待分配新资源的同时，进程继续占有已分配到的资源。

（3）不剥夺条件：进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能在使用完时由进程自己来释放。

（4）环路等待条件：存在一种进程——资源的环形链，链中的每一个进程已获得的资源同时被链中的下一个进程所请求。

2．死锁处理策略

处理死锁的方法主要有以下几种：

（1）预防死锁。通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件，来预防发生死锁。

（2）避免死锁。在资源的动态分配过程中，用某种方法防止系统进入不安全状态，从而避免死锁。

（3）检测死锁

这种方法并不须事先采取任何限制性措施，允许系统在运行过程中发生死锁，但可通过系统所设置的检测机构，及时地检测出死锁的发生。

（4）解除死锁。当检测到系统中已发生死锁时，须将进程从死锁状态解脱出来。

3．死锁预防

预防死锁的方法是使四个必要条件中的第 2、3、4 条件之一不能成立，必要条件 1 是由设备的固有条件所决定的，不能改变，还应加以保证。

（1）摒弃“请求和保持”条件

采用这种方法，可使用预先资源静态分配法。资源静态分配法要求进程在运行之前一次申请它所需的全部资源，若系统有足够资源分配给该进程，便把其需要的所有资源分配给它，这样，进程在整个运行期间，便不会再提出资源要求；若有一种资源不能满足，其它空闲资源也不分配给该进程，而让该进程等待。这样可以保证系统不发生死锁。这种方法既简单又安全，但降低了资源利用率。

（2）摒弃“不剥夺”条件

为了破坏不剥夺条件，可以制定这样的策略：一个已获得了某些资源的进程，若新的资源请求不能立即得到满足，则它必须释放所有已获得的资源，以后需要资源时再重新申请。这意味着，一个进程已获得的资源在运行过程中可被剥夺，从而破坏了个剥夺条件。该策略实现起来比较复杂且要付出很大代价，释放已获得资源可能造成前一段工作的失效，重复申请和释放资源会增加系统开销，降低系统吞吐量。

（3）摒弃“环路等待”条件

为破坏环路等待条件，可以采用有序资源分配法。有序资源分配法是将系统中的所有资源按类型赋予一个编号，要求每一个进程均严格按照编号递增的次序请求资源，同类资源一次申请完。

4．死锁避免

在预防死锁的几种方法中，总的说来都施加了较强的限制条件，从而使实现较简单，但却严重地损害了系统性能。而在避免死锁的方法中，所施加的限制条件较弱，因而有可能获得令人满意的系统性能。在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终都处于安全状态，便可避免死锁的发生。

（1）安全状态与不安全状态

所谓安全状态，是指系统能按某种进程顺序如（P_1，P_2，…，P_n)(称<P_1，P_2，…，P_n>序列为安全序列)来为每个进程分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利完成。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。

虽然并非所有不安全状态都是死锁状态，但当系统进入不安全状态后，便可能进而进入死锁状态；反之，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。

（2）利用银行家算法避免死锁

最有代表性的避免死锁算法，是 Dijkstra 的银行家算法。这是由于该算法能用于银行系统现金贷款的发放而得名。为实现银行家算法，系统中必须设置若干数据结构。

①可利用资源向量 Available。它是一个具有 m 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值为系统中所配置的该类全部可用资源数目，其数值随该类

资源的分配与回收而动态改变。如果 Available[j]=*K*，表示系统中现有 R_j 类资源 *K* 个。

②最大需求矩阵 Max。它是一个 *n*×*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*，表示进程 *i* 需要 R_j 类资源的最大数目为 *K*。

③分配矩阵 Allocation。它是一个 *n*×*m* 的矩阵，它定义了系统中每一类资源当前已分配给每个进程的资源数。如果 Allocation[*i*, *j*]= *K*，表示进程 *i* 当前已分得 R_j 类资源的数目为 *K*。

④需求矩阵 Need。它是一个 *n*×*m* 的矩阵，用以表示每一个进程尚需的各类资源数目，如果 Need[*i*, *j*]= *K*，表示进程 i 还需要 R_j 类资源 *K* 个，方能完成其任务。

上述三个矩阵存在下述关系：

Need[*i*, *j*]＝Max[*i*, *j*]- Allocation[*i*, *j*]

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 *K* 个 R_j 类型的资源，当 P_i 发出资源请求后，系统按银行家算法进行检查：

①如果 $Request_i \leq Need_i$，则转向步骤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 $Request_i \leq Available$，则转向步骤③；否则，表示尚无足够资源，P_i 须等待。

③系统试探着把要求的资源分配给进程 P_i，并修改下面数据结构中的数值：

$Available=Available-Request_i$ ；

$Allocation_i= Allocation_i + Request_i$；

$Need_i= Need_i - Request_i$；

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 P_i 以完成本次分配；否则，将试探分配作废，恢复原来的资源分配状态，让进程 P_i 等待。

系统所执行的安全性算法可描述如下：

①设置两个向量：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，执行安全性算法开始时，Work= Available；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，Finish[i]=true。

②从进程集合中找到一个能满足下述条件的进程：

Finish[i]=false 并且 $Need_i \leq Work$

如找到，则执行步骤③，否则执行步骤④。

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

$Work = Work + Allocation_i$；

Finish[i]=true；

Go to step 2；

④如果所有过程的 Finish[i]=true，则表示系统处于安全状态，否则系统处于不安全状态。

5．死锁的检测与解除

当系统为进程分配资源时，若未采取任何限制性措施来保证不进入死锁状态，则系统必须提供检测和解除死锁的手段。

（1）资源分配图

系统死锁可利用资源分配图来描述。该图由表示进程的圆圈和表示一类资源的方框组成，其中方框中的一个小圆圈代表一个该类资源，请求边是由进程 P_i 指向方框中的 R_j，表示进程 P_i 请求一个单位的 R_j 资源，而分配边则由方框 R_j 指向进程 P_i，表示把一个单位的 R_j 资源分配给进程 P_i。

(2)死锁定理

我们可以利用资源分配图加以简化的方法，来检测系统处于某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 P_i，在顺利的情况下运行完毕，释放其占有的全部资源，这相当于消去的所有请求边和分配边，使之成为孤立结点。

②由于释放了资源，这样能使其它被阻塞的进程获得资源继续运行。

③在经过一系列简化后，若能消去图中的所有的边，使所有进程结点都孤立，则称该图是可完全简化的，反之是不可完全简化的。

可以证明：S 状态为死锁状态的充分条件是当且仅当 S 状态的资源分配图是不可完全简化的。该条件称为死锁定理。

（3）死锁的检测

检测死锁的一个常见算法描述如下(算法中 Available、Allocation、Need、Work 的含义与银行家算法相同，L 为一个初值为空的进程表)：

①将不需要、不占有资源的进程(即 $Need_i$= $Allocation_i$ =0 的进程)记入表 L 中，并令 Work 等于 Available。

②从进程集合中找一个未记入 L 中的，且 $Need_i \leqslant Work$ 的进程，若不存在这样的进程，则转步骤③；否则，将该进程记入表 L 中，增加工作向量 Work = Work + $Allocation_i$，并重复执行步骤②。

③若不能把所有进程都记入 L 表中，则表示这些进程将发生死锁。

（4）死锁的解除

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的两种方法是：

①剥夺资源。从其他进程剥夺足够数量的资源给死锁进程，以解除死锁状态。

②撤消进程。最简单的撤消进程方法是使全部死锁进程都夭折掉，稍微温和一点的方法是按照某种顺序逐个地撤消进程，直至有足够的资源可用，使死锁状态消除为止。