多处理机系统的逻辑分析与设计

蔡希尧 编著

西北电讯工程学院出版社

多处理机系统的逻辑分析与设计

蔡希尧　编著

西北电讯工程学院出版社

1987

内 容 简 介

本书对合作的多处理机系统在逻辑分析和设计方面做了较全面的论述。书的内容包括并发程序设计，作业分配，系统结构，造型和分析，设计方法等5个方面，对这个领域中近年来所发表的主要研究成果做了系统的概括。书的内容新颖，叙述得体，适合于计算机组织与系统结构、计算机应用、计算机软件等专业的研究生、高年级大学生、大学教师以及在这个领域中工作的工程技术人员使用。

多处理机系统的逻辑分析与设计

蔡希尧 编著

责任编辑 徐德源

西北电讯工程学院出版社出版发行

西北电讯工程学院印刷厂印刷

新华书店经销

开本 787×1092 1/16 印张 12 2/16 字数 294 千字

1987年6月第1版 1987年6月第1次印刷 印数 1—3 000

ISBN 7-5606-0018-2/TP·0008

统一书号：15322·88

定价：2.10元

前　言*

由于微电子学和微型机的迅速发展和它们的价格不断下降，使用多个微机构成的多机系统，已在工业、商业、科学研究、管理和军事等领域得到越来越多的应用。

就使用的目的来说，多处理机系统可分为两大类，一类是以共享资源为目的的多机系统，这类系统大多是松耦合的，计算机网络属于这一类。另一类则是以合作处理为目的，以紧耦合为主，但趋势是朝着松耦合方向发展。关于第一类多机系统已有较多的论著，对于计算机网络，已有国际通用的标准，对于局域网，生产厂商可提供典型的配置，这方面的内容不是本书讨论的重点。第二类多机系统使用多个处理机合作完成一个任务，在这类系统中，同样存在资源共享问题，但还需要解决其他的问题，例如程序执行过程中处理机的相互通信等等。在分析和设计上，第二类多机系统比第一类更加复杂和困难，本书将讨论这一类多机系统。

设计一个合作的多机系统，需要解决的主要问题有系统结构，作业分配，程序设计以及适当的系统设计方法等。此外，为了对系统的性能和行为进行研究和分析，需要解决模型的建造，寻找合适的分析工具。在本书中，将对这些问题逐一进行讨论。

全书分为七章。第一章是概论。第二章和第三章是并发程序设计，讨论多机系统中并发程序的特点和机制，以及用于并发处理的程序设计语言。在第二章中讨论以共享变量为基础的并发程序设计问题，在第三章中讨论以消息传递为基础的并发程序设计问题，而且以几种有代表性的记法和语言为线索开展讨论。并发程序设计语言是比常用的顺序程序设计语言高一层次的语言，是描述多机系统并发行为的最好工具，也是设计多机系统所必需的手段。这两章所讨论的内容，是20世纪70年代以来在程序设计领域中研究工作的主要成就之一。

第四章讨论作业分配，这是影响多机系统能否有效工作的一个关键问题。多机系统的吞吐率随着处理机数目的增加会出现饱和现象，以致于下降，这主要是由处理机之间的通信开销引起的。作业分配不当，将加剧这种现象，将使系统中的各个处理机在负载很不平衡的情况下工作。在这一章中将讨论有关作业分配的基本理论和算法设计等问题。

第五章讨论多机系统的结构。两类多机系统在结构上有许多相似之处，所以本章中的不少内容，可适用于两类多机系统。此外，在这一章中还讨论多机系统中的操作系统问题。从逻辑功能上来讲，操作系统是计算机系统结构所应解决的主要问题。

第六章研究多机系统的造型与分析，我们采用队和Petri网作为建造模型的工具，对多机系统的性能和行为进行分析，并得出一些有指导意义的结论。

最后一章以计算机系统结构和软件工程所阐述的理论和方法为依据，从逻辑上对多机系统的设计做了阐述，并对多机系统的可靠性问题做了探讨。

以上内容，实际上是5个专题，在学习时可以在次序上加以变更。

本书是在作者近几年来讲授多处理机系统所用的讲稿，以及作者所在的软件研究室近年来从事多机系统科研任务所积累的经验的基础之上编写而成的。在讲授这门课程时，没有合适的参考书，主要是搜集期刊上的论文写成讲稿，经过几年的积累，加上集体的实践经验总

* 本书的部分研究工作，得到中国科学院基金的支持。

结，有了一定的基础，而教学和科研工作需要有一本关于合作的多处理机系统的书，于是对积累的资料和经验加以筛选、综合和整理，加上我们自己的一些观点和工作，写成此书。多处理机系统虽然在应用上已经比较广泛，但对这类系统的性能和行为，工作机制、分析方法和程序设计等方面，都还存在着大量有待深入研究的问题；这类系统的性能评价和测量、系统优化和正确性验证等方面，更是有待开拓的领域。所以，本书所包括的内容，只反映了目前发展阶段的部分情况，希望以后有机会能加以充实和提高。

书中有两节加了*号，即3.8，4.7，是两个专门的问题，分别由蔡表东和黄欣执笔。这两个问题是这两位年青的作者在研究多处理机系统时所写的报告和论文中的一部分内容。为了和书的行文一致，对原文做了一点修改，列入相应的各章。

在编写本书时，得到了作者所在的软件研究室的同事们的多方关心和鼓励，得到了林庆元同志的支持。西北电讯工程学院出版社给予了许多方便，谨向他们表示感谢。

作　者

1986年8月

目　录

第一章　概论 ……1
参考文献 ……2
第二章　并发程序设计（一） ……3
2.1　基本概念……3
2.2　并发进程执行的表示……6
2.3　互斥和关键区……7
2.4　信号量 ……11
2.5　条件关键区 ……14
2.6　管程 ……16
2.7　支持管程的程序设计语言 ……22
2.8　路径表达式 ……29
参考文献……31
第三章　并发程序设计（二） ……33
3.1 卫式命令……33
3.2 以消息传递实现进程间的通信……34
3.3 远方过程调用……37
3.4 通信的顺序进程……39
3.5 分布进程……44
3.6 通信端口……48
3.7 Ada 中的并发机制……52
3.8 并发程序设计语言中的不确定性* ……57
参考文献……69
第四章　作业分配……72
4.1　概述 ……72
4.2　图论法 ……73
4.3　极小极大图匹配算法 ……79
4.4　整数规划法 ……85
4.5　启发式作业分配 ……89
4.6　动态分配 ……93
4.7　一种新的启发式作业分配方法—HGMA* ……96
参考文献 ……103
第五章　系统结构 ……104
5.1　概述……104
5.2　互连网络……107
5,3　总线互连……115

5.4 紧耦合系统……119
5.5 处理机间的数据通信……121
5.6 链路层的通信协议……126
5.7 操作系统……129
参考文献……136
第六章 造型与分析……138
6.1 多机系统的排队模型……138
6.2 Petri 网模型及其分析……145
参考文献……160
第七章 设计方法……161
7.1 概述……161
7.2 虚拟机的概念……162
7.3 要求分析与功能分解……164
7.4 硬件的选择……167
7.5 软件设计中的问题……172
7.6 可靠性……182
参考文献……184
译名对照表……186

第一章 概 论

如何提高计算机资源的利用率，提高计算机系统的吞吐率，是计算机领域中长期以来的一个努力方向。

早期的一个解决办法是使机器一直忙着，采用批处理的方法来有效地使用机器，操作系统就是围绕这一使用方式来设计的。

随着高级程序设计语言的出现，用户逐渐能够自己编写程序。这样一来，仅有批处理方式就不能适应应用要求了，于是出现了分时制，多个用户可在同一机器上同时工作，他们的程序是彼此无关的，计算机系统周期地给不同的程序分配使用的时间。

为了进一步提高计算机系统的吞吐率，除了积极改进元器件以外，还产生了以多个处理机构成一个计算机系统的设想。在这样的系统中，资源可以共享，处理机可以合作来完成任务。

20 世纪 70 年代初期微处理机的出现，给计算机技术带来巨大的冲击。从系统结构的角度来说，如何有效地利用这一廉价的处理机资源，构成一个功能强大的计算机系统，被提到日程上来了。这种计算机系统，含有数量相当多的处理机，以一定的通信网络连接在一起，能够代替大型机的工作。显然，这是很引人注目的一个方向，也使得对多处理机系统的研究工作，进入一个新的时期。

从摆脱冯·诺曼式结构的束缚，创造一个新的计算机系统结构来看，多机系统是目前能够做得到的一种有效的结构形式。正在开展的第五代计算机的研究工作中，多指令流多数据流(MIMD)是一种主要的结构形式，这是高度并发的多机系统。

多处理机系统研究的主要内容包括：

(1) 系统结构。体现系统结构的两个主要问题是有效的互连技术和控制方式。

(2) 作业分配。一个大程序按照信息隐蔽的原则分解为多个模块以后，如何分配给多个处理机，使处理机既做到负载均衡，又能够以最佳的方式(例如最小的代价，最短的时间等)来完成任务。

(3) 程序设计。多机系统蕴涵着资源共享，是一个并发系统，要采用并发程序设计的原则，妥善地解决进程间的互斥、同步和通信等问题。

(4) 程序设计语言。对于并发程序设计，应当采用什么样的语言呢？本世纪 70 年代以来，对于这个问题进行了大量的研究工作，提出了一些新的概念，设计了一些支持并发程序设计的记法和语言。进一步的研究仍在继续。

(5) 分析与设计方法。对多机系统的严格理解，要有形式的分析方法和有效的设计方法，这是很明显的事情，但却是难度较大的开拓性的工作。

本书将就以上这些问题，逐一地进行讨论。就现在已经达到的科学技术水平来说，对于多处理机系统，上面所列举的主要问题，应当说是知之甚少，大量的问题还有待深入的开发和研究。但是，多机系统的应用问题却已经摆在议事日程之上，要求实际上予以解决。因此，我们需要学习多处理机系统的工作原理和有关的新的概念，学习分析和设计多处理机系统的可行的技术和方法，这正是编写本书的动力。

鉴于一方面是知之甚少，一方面是迫切的应用要求这种矛盾，本书在控制驱动的概念之

下，充分利用顺序程序设计已经成熟的结果，来讨论有关的问题。至于在第五代计算机的研究工作中所已经提出的数据驱动和需求驱动等非冯·诺曼控制方式，以及以并发为基础而把顺序执行作为例外的语言等有关的论述，未列入本书。

参考文献

[1] B.A.Bowen and R.J.A.Buhr, The Logical Design of Multiple-Microprocessor Systems, Prentice-Hall, Inc., 1980

[2] R.E.Kahn, A New Generation in Computing, IEEE Spectrum, Vol. 20, No.11, 1983, 36-41

[3] A.L.Davis, Computer Architecture, IEEE Spectrum, Vol.20, No.11, 1983, 94-99

[4] J.R.Mc Graw, Data Flow Computing—Software Development, IEEE Trans., Vol. C-29, No.12, 1980, 1095-1103

第二章　并发程序设计(一)

2.1　基本概念[1]~[6]

计算机系统中的并发性概念,在本世纪50年代中期已经出现。当时,由于大容量磁芯存贮器和一些外围设备的发明，计算机的结构发生急剧的变化，使程序的指令数可以达到10^4～10^6。为了提高计算机的工作效率，曾考虑使快速的中央处理器在多个低速的外围设备中轮流转接，而让外围设备同时工作，这是早期的并行操作的例子。

60年代初期分时概念出现以后，多个用户可以同时使用同一机器，有了多道程序系统，相应的操作系统必须具有并发的控制功能。

与多个用户同时使用一个机器相对应，是单个用户为了达到某一目的，在同一机器上同时运行多个作业。这时候也有多道程序在执行，但它们来自单个用户，称之为多道作业。

以后，又在多个处理机上，在共享存贮器的条件下进行并发操作，这种情况被称为多道处理。

目前，即使是微处理机，它们的操作系统一般都能支持多道程序和多道作业，有的也能支持多机的多道处理。发展的趋势是走向更加高度的并发操作，超大规模集成电路的发展支持了多机并发系统的实现。

但是，并发程序的执行有一个不同于顺序程序执行的特点，这就是时间相依的特性。我们常用的程序，语句总是一行接一行地执行，前一行语句执行完毕以后，后一行语句才能开始执行。我们称这类程序为顺序程序，它的运行结果与执行的速率无关，也就是不依赖于时间的长短。所以把顺序程序放在不同的机器上执行,只要机器能够接受程序所使用的语言，当工作状态相同且输入相同时，输出是相同的。另一方面，把一个顺序程序的执行中断，只要能够保留中断现场，那么，不管中断时间的长短，当它恢复运行后，仍然能够得到所希望的结果。顺序程序的这一重要特性，是大家所熟悉的。

并发程序却不具备这一特性，现举例说明如下：

例1　设x是两个程序P1和P2的共享变量，其初始值为0。P1和P2分别为

```
P1: x:=x+1;
P2: x:=x+2;
```

它们在同一机器上并发执行的结果是什么呢?

我们知道，赋值语句是由3个不可分割的基本操作组成的,它们是:(1) 给寄存器加载;(2) 进行运算;(3) 将结果送给寄存器。P1和P2都包含这3个基本操作，但这些基本操作可以任意交织，而交织的情况有赖于P1和P2的执行速率。显然，交织的情况不同,计算的结果也就不同。

例2　设共享变量x的初始值为100，两个程序分别为

```
P1: x:=x+10;
P2: if x>100 then print x;
    else print x-50;
```

当 P1 和 P2 并发执行时，其结果决定于两者的交织情况。如 P1 先执行完毕，再执行 P2，其结果为 110；如 P2 先执行完毕，再执行 P1，打印结果是 50；如果先执行 P2 中的第一句，再执行 P1，最后执行 P2 中的第二句，打印结果是 60。

这种时间相依的行为所引起的执行结果的不同，是很难发现的，因为从一次执行到另一次执行，语句的交织次序并不是固定的，而是依赖于各自的速率，即使输入的数据保持一样，输出也可能是不同的。

要克服这一困难，需要解决以下问题。

首先要设法摆脱并发程序的时间相依特性。解决的办法是把并发程序加以分解，使其成为较小的模块，这些模块是顺序程序，作为并发系统的最高组成单元。当一个并发系统能够用一组顺序程序来描述、设计和开发时，就有希望使并发系统成为可控制的。

由于并发蕴涵着资源共享，作为组成单元的顺序程序，往往需要加以一定的扩充，例如附加某些状态信息等等。这种普遍化的顺序程序称为进程，它是高于程序码的表示系统功能的一个抽象，是并发程序中的主动因素。

其次，要设法找到一些机制，来解决共享资源时的互斥、同步以及进程间的通信等问题，这些机制必须是安全可靠的，并且要便于使用。

第三要设计语言，它们具有上述的能够支持并发处理的机制，能够表示普遍化的概念，有准确的定义，有简明的表示方法，并且有效和安全。

以上这 3 个方面，经过十多年的努力，已经找到了一些有效的解决办法。

在并发程序中，一部分时间内，多个进程可以独立地并行地工作，而另外一些时候，它们要访问公共的资源。进程中使用公共资源的那一段程序叫做关键区。当一个进程被允许进入关键区时，其他进程不能同时进入，只有在使用关键区的那个进程退出以后，等待进入关键区的其他进程中的一个才能进入，这就是互斥。关键区那段程序是顺序性的，原来以并行方式运行的多个进程，当它们竞相进入关键区时，互斥机制强迫它们建立一定的时序关系，使它们的执行按顺序进行。不同进程的关键区以任意交织的次序严格地顺序执行。

资源共享是进程间的一种合作形式。并发进程之间还可以有其他的合作形式。进程之间的合作需要同步，有时还需要通信。

同步是对多个事件在时序上的一种限制。在并发程序设计中要采用某种机制对一个进程的执行予以延时以满足时序的限制。关键区的互斥具有这一性质，所以它也是一种同步机制。

通信则允许一个进程的执行影响另一个进程的执行。因此，通信往往需要以同步为前提。并发进程间的通信有三种基本的方式：

(1) 通过共享变量达到通信。在这种方式中，进程是主动的成分，它们往往要调用一个过程对某种对象实现操作，达到通信的目的。

(2) 两个进程之间直接传递消息以进行通信。在这种方式中，发送进程和接收进程中无论哪一方先准备好了，都要等待另一方到达通信点后，才能进行通信。所以这类通信的方式有时也叫做握手式的通信。

(3) 远方过程调用。它的基础是消息传递，但进程之间的交互作用是通过调用其他处理机上的过程来实现的。

也有人把进程间的通信分成同步通信和非同步通信两大类：

(1) 同步通信。发送进程发一个消息给接收进程，并等待接收进程的应答，发送进程收

到应答，说明接收进程已经收到消息。有时，发送进程在发送消息后，要等待接收进程送回操作结果。显然，前者是同步接收，后者是远方过程调用。

(2) 非同步通信。不需要应答，也不等待对方处理的结果。由于是不同步的，因此这种通信需要缓冲寄存，用以存放已经发出但尚未收到的消息。缓冲寄存的设置如图 2.1-1 所示。

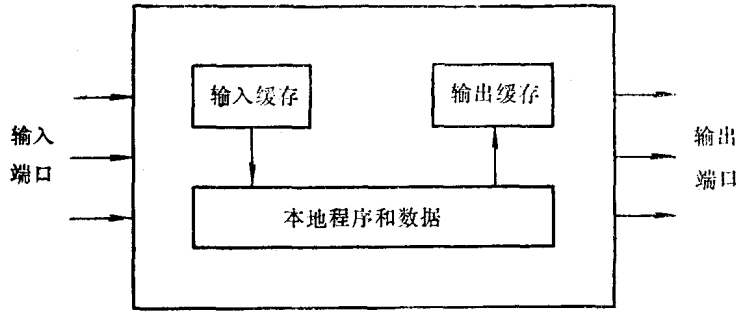

图 2.1-1 非同步通信

实际上，在这种通信中，只有发送进程已经把消息放在缓存中，接收进程才能把消息取走。这也有一个先后的时序关系，只是发送的消息要延迟一段时间后才到达接收进程，而这一延迟可以是相当长的。这样，接收到的消息，已不能代表发送进程的当前状态，所以叫做非同步通信。

除了这两类通信以外，还可以用广播方式，多对发送和接收方式等来达到进程间通信的目的。

并发程序的实现，目前常用的支持有：

(1) 利用已有的操作系统来实现并发处理。目前许多机器，包括近几年新出厂的微型机，它们的操作系统大多都具有多道程序，多道作业和多道处理的功能，一般是通过系统调用的方式来达到这些功能。

(2) 使用支持并发程序设计的语言，来完成并发处理。

并发程序是以顺序程序为基础而构成的。因此，用于并发程序设计的语言，往往以已有的顺序程序设计语言为基础，引入一些新的概念，加以必要的扩展而构成。

一类并发程序设计语言，进程间的相互作用以共享变量为基础，语言以进程为主动的成分，共享变量为被动成分。这类语言称为面向过程的语言，如 Concurrent Pascal, Modula, Mesa, Edison等。

另一类并发程序设计语言，进程间的相互作用依靠发送和接收来完成，没有共享的被动对象，称为面向消息的语言，如 CSP, GYPSY 和 PLITS 等。

以消息传递为基础，结合前两类语言的特点，构成第三类语言，进程间的相互作用采用远方过程调用，称之为面向操作的语言，如 Ada, DP 和 SR 等。

对于这些并发程序设计语言，以后将择要加以介绍。

在结束本节的时候，根据前面的论述，我们给并发程序一个定义：一个并发程序含有两个或更多个进程，它们可以在时间上互相重叠，程序中的动作单元(不可分割的单元)可以任意交织地执行。

这个定义，表明了进程的并发性，同时也指明了并发程序既可以在单个处理机上执行，也可以在多个处理机上执行。我们将按照这一定义展开有关并发程序设计的讨论。

2.2 并发进程执行的表示[5],[7],[8]

对于并发进程的执行，已经提出多种表示的方法，现在介绍其中的3种。

1. 用 fork 和 join 表示并发执行

设 P1 和 P2 是两个进程，P1 是调用者，P2 是被调用者。当 P_1 执行到 fork P2 时，P2 被启动，于是 P1 和 P2 同时执行。当 P1 执行到 join P2时，如 P2 还没有结束，则 P1 要等待到 P2 的结束，两者取得了同步，然后 P1 继续执行。如果 P1 到达 join P2，而 P2 这时候已经结束，则 P1 可以往下执行。这种早期的表示并发执行的方法，如果使用得当，功能是强的。但是，这是一种非结构化的程序构造，fork 命令和 go to 语句很相似，使用不当时将会引起许多麻烦。在 UNIX 操作系统和 PL/I，Mesa 等语言中，并发执行采用 fork 和 join 的概念。

2. 用 **cobegin** 和 **coend** 表示并发执行

这是结构化的并发执行的表示，是 Dijkstra 于 1968 年最早采用的，它的表示形式如下：

```
cobegin
  statement 1;
  statement 2;
    ⋮
  statement n;
coend;
```

在 **cobegin** 和 **coend** 之间，n 个语句同时执行。图 2.2-1 形象地表明了这 n 个语句的执行情况。

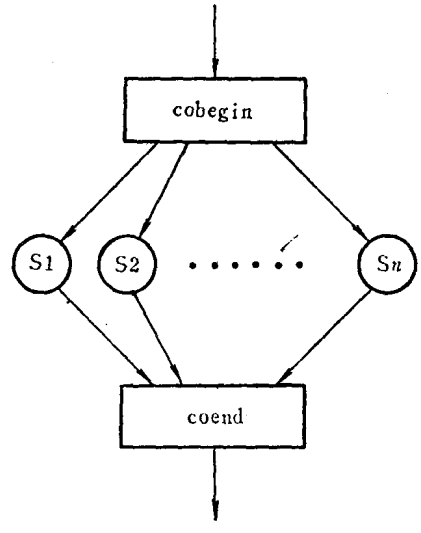

图 2.2-1 用 **cobegin** 和 **coend** 表示的并发执行

在并发程序中，有些进程彼此独立，不共享数据，称这些进程是不相关联的，刚才所举的 n 个语句，就属于这一类。它们分别独立地同时执行，最后一个结束时，并发执行也就结束了。另外一些进程，它们共享某些数据，或者互相之间要通信，它们是相关联的进程，也是并发执行的，因此也可以用现在讨论的表示法来表示它们的执行，在程序中有特殊的记法指出进程间的互斥、同步和通信，可是还没有通用的图示法来表示。

用 **cobegin** 和 **coend**，能够比较方便地表示嵌套的并发结构，例如：

```
cobegin
   S1;
   begin
      S2;
      cobegin S3; S4 coend
      S5;
   end
   S6;
coend
```

在这一程序中，S1，S6 和 **begin** 与 **end** 之间的语句，是并发执行的，而在 **begin** 和 **end** 之中，S3 和 S4 是嵌套的并发执行。图 2.2-2 形象地表示了它们的并发执行的关系。

3．用 **coroutine** 表示并发执行

coroutine 用对称的形式实现控制的转移，每一个 **coroutine** 可以看做是实现一个进程。在程序的执行过程中，用 **call** 来启动 **coroutine** 以 **return** 把控制交还给调用者，而 **coroutine** 之间的控制转移，则用 **resume** 来表示。**resume** 语句的作用如同执行过程调用一样，把控制转移给指名的程序，并保留足够的状态信息以控制 **resume** 之后的执行。控制返回给原来的 **coroutine** 时，仍然执行 **resume**，图 2.2-3 表示了 **coroutine** 的执行情况。

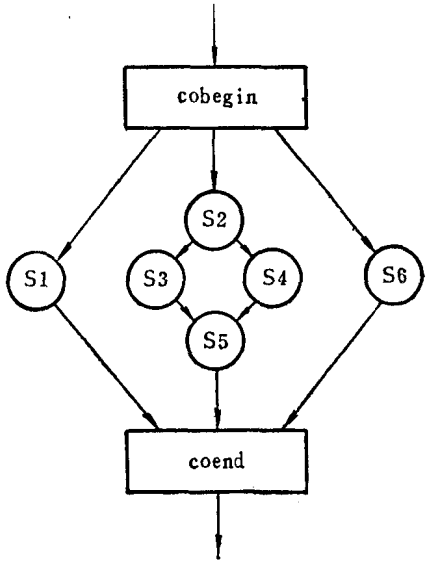

图 2.2-2　用 **cobegin** 和 **coend** 表示嵌套的并发执行

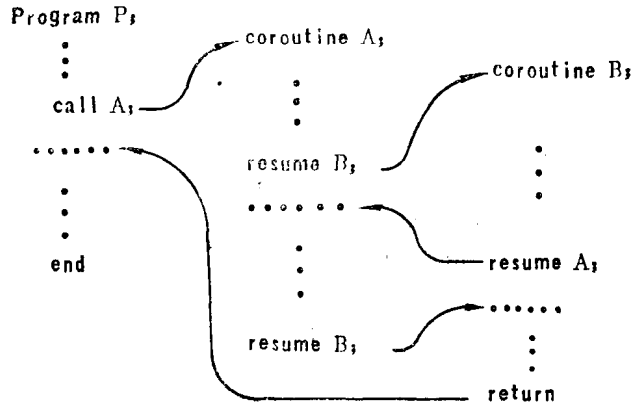

图 2.2-3　用 **coroutine** 表示并发执行

coroutine 的语义只允许在同一时间执行一个子程序，所以它不适宜于真正的并发处理，而可以用来组织单个机器上的并发程序。多道程序的执行可以用 **coroutine** 来实现。

2.3　互斥和关键区[2],[3],[5],[8]~[10]

在本章第一节中，已经指出互斥是并发处理时的一个重要的机制。当两个或更多个进程要同时使用共享资源时，只允许一个进程使用资源，其他的进程一律不准使用，这叫做互斥。每个进程使用共享资源的那一段程序，叫做关键区。在实现互斥时，每个进程的关键区是不可分割的操作。

Dijkstra 最早对互斥机制进行了严格的分析，并且开创了用普通的程序设计语言解决并发控制的问题。他的这一创造，写入 1965 年发表的一篇短文中[9]。

假设有 n 个进程要共享资源，当互斥的功能实现时，只有一个进程进入关键区，其他 $n-1$ 个进程都在关键区以外等待。解这个问题的程序，除了要做到这一点以外，还需要满足以下的要求：

(1) 对 n 个进程来说，解是对称的，也就是不引入静态的优先等级。

(2) 不计较进程的相对速率，也不假设进程的速率是不变的。

(3) 当一个进程停止在关键区之外时，不会阻止其他进程的运行。

(4) 解的结构迟早能让 n 个进程中的一个进程进入关键区，不致于无限期地作不出决定。

按照以上这些要求，可以设计出一个安全可靠的能够保证互斥的程序。“忙等待”是早期设计的互斥机制，目前在实际工作中仍然使用。在这一机制中，当一个进程已经在关键区时，其他企图进入关键区的进程都要等待，它们在等待的同时，要反复测试条件，以了解什么时候可以进入关键区。

现在我们来设计用于互斥的程序。

假设有两个进程 P 和 Q，共享资源 R。我们可以设想一个标志，例如一个布尔量，它的一种状态表示资源正在被使用，另一种状态表示资源是空闲的。如果正在被使用，其他进程不能使用，如果空闲，则可以使用。按照这一思路，并假设 P 和 Q 都是反复要求使用资源 R 的，则可以写出以下的程序：

```
var free : boolean;
begin
   free := true;
   cobegin
     process P; do forever;
       repeat until free;
       free := false;
       use resource;
       free := true; end;
     process Q; do forever;
       repeat until free;
       free := false;
       use resource;
       free := true;  end
   coend
end
```

这个程序能够正常工作吗？我们知道，在 cobegin 和 coend 之间的两个进程，是并发执行的。一开始，标志 free 已经置为 true，因为标志说明资源是空闲的，程序中又没有加入其他的限制，因此如果 P 和 Q 同时要进入 R，则是允许的。但这样一来，互斥就不能实现了。因此，不能简单地靠这一个标志达到互斥存取的功能。

是不是把标志加以修改，就可以改变上述的缺点呢？例如使标志为真时，进程 P 可以使用资源 R，标志为假时，进程 Q 可以使用资源 R。显然，这样做可以实现互斥。以 F 代表这一标志，可以写出以下的程序：

```
var F : boolean;
begin
   F := true;
   cobegin
     process P; do forever;
       repeat until F;
        use R;
```

```
        F:=false; end
      process Q; do forever;
        repeat until not F;
        use R;
        F:=true; end
    coend
end
```

在这个程序中，当 F 为真时，进程 P 可以使用 R，进程 Q 不能使用；F 为假时，则 Q 可以使用 R 而 P 不能，互斥得到了保证。但这个程序使用很不方便，因为两个进程使用资源 R，只能以 P，Q，P，Q…的方式轮流进行。当其中一个进程不使用时，另一个进程也不能使用，虽然它企图再次使用。这样做显然不能满足应用的要求。

下面介绍 Peterson 于 1981 年提出的一个较好的解法。解的格式如下：

```
process P;
   loop
      entry protocol;
      critical section;
      exit protocol;
      noncritical section
      end
   end;
process Q;
   loop
      entry protocol;
      eritical section;
      exit protocol;
      noncritical section
      end
   end
```

其中入口协议entry protocol包括：企图进入关键区的布尔量，用 enter 1 和 enter 2表示；优先级的名次，用 turn 表示；还有根据以上两个条件应采取的动作的语句。出口协议 exit protocol 比较简单，只包括 enter，并把它置成假。举例如下：

```
program mutex;
   var enter 1, enter 2 : boolean initial(false, false);
       turn : integer initial("P1"); {or "P2"}
   process P1;
      loop
         entry-protocol;
            enter 1:=true;
            turn:="P1";
            while enter 2 and turn="P2"
                  do skip;
            critical section;
```

```
      exit-protocol;
          enter 1:=false;
      noncritical section
      end
    end;
  process P2;
    loop
      entry-protocol;
          enter 2:=true;
          turn:="P2";
          while enter 1 and turn="P1"
            do skip;
          critical section;
          exit-protocol;
            enter 2:=false;
          noncritical section
      end
    end
  end
```

在程序中的 do skip，是当另一进程企图进入关键区，而名次又属于那个进程时，则让那个进程进入关键区，自己则处于等待状态。这样的互斥机制，比较安全，避免了死锁，而企图进入关键区的进程，迟早总能进入，不会永远等待。

互斥是利用共享变量的一种同步方式，并发的进程只能在关键区访问共享变量，或对它们实现操作，所以这里所讲的互斥，也就是关键区的互斥。概括以上的论述，关键区互斥所具有的特点是：

(1) 当关键区内没有进程在的时候，一个进程可以立即进入关键区。

(2) 当关键区内已经进入一个进程，其他企图进入关键区的进程必须被延时。

(3) 当一个进程退出关键区，同时有多个其他进程企图进入时，只能允许这些进程中的一个进入关键区。

(4) 能够在有限的时间内决定某个进程进入关键区。

(5) 一个进程在关键区内的时间是有限的。也就是说在关键区内对共享变量进行操作的语句一定能在有限的时间内结束。

(6) 如果对于被延时的进程进入关键区的选择带有优先权，那么必须保证不使某个进程受到无限期的延时。

如果不同的进程所访问的变量是不同的，那么，可以同时进入各自的关键区。例如 v 和 w 是两个不同的变量，V 和W是它们的类型，可以作以下的安排：

```
var v: shared V; w: shared W;
cobegin
    critical section v do…;
    critical section w do…;
coend
```