内部

计算机操作系统学术交流会

论文选集

Jisuanjicaozuoxitongxueshujiaoliuhui

Lunwenxuanji

计算机操作系统学术交流会秘书组
中国科学院计算技术研究所
一九七八年十月

计算机操作系统学术交流会
论 文 集

编 辑　计算机操作系统学术交流会秘书组
　　中国科学院计算技术研究所

印 刷　合 肥 新 华 印 刷 厂

一九七九年二月出版　2.00

出 版 说 明

为了响应党中央、华主席向科学技术现代化进军的伟大号召，认真落实全国科学大会的精神，中国电子学会于一九七八年六月二十二日至七月三日在江西庐山召开了“计算机操作系统学术交流会”。

会议贯彻“百花齐放，百家争鸣”的方针，充分发扬学术民主，就计算机操作系统的科研成果及结构理论等，进行了广泛的交流和学术讨论，收到了良好效果。

会议期间共收到学术报告和交流资料80余篇。根据到会同志及有关专业人员的要求，选其中部分论文编成会议文集，供参考。因出版时间仓促，错误难免，欢迎读者批评指正。

“计算机操作系统学术交流会”秘书组

一九七八年十月

目　　录

国外操作系统的若干研究动向

张尤腊　周巢尘

（中国科学院计算技术研究所）

一、同步机构

进程之间共享资源时，则发生同步问题。同步问题是导致操作系统复杂性的重要原因之一。因此当前花相当大的力量来研究这一问题，特别是扩充已有的同步机构和探索新的同步机构。目前同步机构主要有以下几种。

1.1信号量

Dijkstra 在1968年前后提出了信号量概念及P、V操作，并应用于THE系统〔Dij 68B〕，获得了普遍重视，从此开始了这类机构的各种研究。

最初提出的P、V操作有三种。一种是二值P、V操作，即信号量的值只允许为0和1。第二种是一般P、V操作，即信号量可以取任意整数值。第三种是并行P、V操作，即在多个信号量上同时进行P、V操作〔Dij71〕。

72年提出了带测试参数的P、V操作〔Van72〕。带测试参数的P操作表示成P（S_1，a_1；…；S_n，a_n），其含义是，若对一切i（$1\leqslant i\leqslant n$）有$S_i\geqslant a_i$，则令$S_i:=S_i-a_i$（$1\leqslant i\leqslant n$），否则进程被封锁。相应的V操作表示成V（S_1，a_1；…；S_n，a_n），其含义是，对全部i（$1\leqslant i\leqslant n$），令$S_i:=S_i+a_i$。这类V操作的解封方面有不同的方法，有的是释放全部被封锁者，有的是在进入封锁队列时带有参数值，解封时，按一定的调度原则及当前的信号量值，解封一些可能的解封者。

75年提出了既带有测试参数又带有改变量的P、V操作〔Pre75〕。P（S_1，a_1，δ_1；…；S_n，a_n，δ_n）的定义是，若对一切i（$1\leqslant i\leqslant n$），$S_i\geqslant a_i$，则令$S_i:=S_i-\delta_i$；否则封锁进程。V（S_1，δ_1；…；S_n，δ_n）的定义是，对全部i（$1\leqslant i\leqslant n$），$S_i:=S_i+\delta_i$。

用以上各种P、V操作实现进程的协同时，经常要使用公用的控制变量（称为全局变量）以及对全局变量的赋值、判断转移等。77年Agerwala 提出了一种不使用全局变量就可以完成各种同步要求的P、V操作，称之为完全的P、V操作〔Age77〕。完全P操作P（S_1，S_2，…，S_n，$\overline{S}_{n+1}$，…，$\overline{S}_{n+m}$）的定义是，若对任一i，j（$1\leqslant i\leqslant n$，$1\leqslant j\leqslant m$）$S_i>0$且$S_{n+j}=0$，则对一切i（$1\leqslant i\leqslant n$），令$S_i:=S_i-1$，否则进程被

封锁。完全V操作V（S_1，S_2，…S_n）的定义是，对全部i（$1 \leqslant i \leqslant n$），$S_i := S_i + 1$。为证明这种操作的完全性，作者先假设图灵机为进程间协同的一般机构。

1.2 消息缓冲

Hansen 于1973年提出了消息缓冲同步机构〔Han 73 B〕。发送消息的进程把消息放在发送区中，用 send message 原语发送消息。该原语把消息从发送区搬至缓冲区，若接收者正在等待消息，则唤醒之。接收者用 wait message 原语接收消息，该原语把消息从缓冲区搬至接收区，若没有消息，则封锁接收者。接收者把消息处理完后，用send answer原语回答发送者，发送者用wait answer原语等待回答。

在实际中常常用改进了的消息缓冲同步机构，例如邮箱通信机构等。

1.3 管　程

Dijkstra 在71年提出将进程间的同步控制集中起来，用一种叫“秘书”的程序结构来实现同步控制〔Dij 71〕。Hansen〔Han 73 A〕和Hoare〔Hoa 74〕把“秘书”思想发展为管程的概念。

系统中的各种硬、软资源用数据结构抽象地表示，资源管理用对数据结构的操作表示，数据结构以及在数据结构上的操作集中起来，构成了管程。一个管程的数据只能由该管程存取，外界不能存取。每个管程管理着一种资源，当进程需要使用该资源时，就调用该管程。

一个管程由管程名字、局部于该管程的数据的说明、若干个过程、给局部数据赋初值的语句这四部分组成。

实现管程时必须解决三个问题。第一，必须有两个基本同步操作 wait 和 signal。当一个进程想获得一个资源而该资源又暂时不能供它使用时，管程用 wait 操作延缓该进程的运行。当别的进程通过调用该管程释放有关资源时，管程执行signal操作，解封被延缓的某个进程。有待研究的一个问题是，signal 操作是否一定可以安排在管程过程的最末尾。若能做到这一点，则管程的实现可以简化〔Hoa 74〕。Robert等人则更进一步，提出把同步机构中的处理部分和控制部分分开〔Rob 77〕。他们认为，同步的主要困难是复杂地使用同步原语，即同步原语散布在处理代码的各处而引起的，因此两者分离对解决复杂性会有好处。第二，各进程互斥进入一个管程，即在一个时刻只能有一个进程真正成功地进入一个管程，其它调用者必须暂停。有了互斥执行，就有可能避免某些很荫蔽的与时间有关的编码错误。第三是解决管程的嵌套调用问题。这一问题所引起的实现上的困难是，在嵌套调用过程中内层管程执行 wait 操作时，是释放当前管程的互斥呢，还是释放这次嵌套调用所牵涉到的所有管程的互斥。

管程这一同步机构的同步能力并不比其它同步机构弱，其它的同步机构，例如信号量和邮箱，皆可用管程实现。除此以外，用管程作同步手段还有两点好处。第一，由于共享数据和在数据上的操作集中起来，并且限制外界对管程局部数据的存取，编译程序进行检查以保证数据局部性，因此减少了同步复杂性，增加了可了解性，提高了可靠性。第二，由于上述的集中，使模块独立，功能单一，层次分明，减少重复，因此有利

于系统模块化。

用管程的概念来构造操作系统，尚属试验阶段。Hansen〔Han 77A，77B〕已用管程、类程和进程的概念实现了“单用户操作系统（SOLO）”、处理小作业的“作业流处理系统”、用于过程控制的“实时调度程序”三个小系统。SOLO 系统是在1975年4月间用27次调试性运行就调试完毕。自那时以来每天都在运行，没有故障。其它两个系统分别用10次和21次调试性运行调试完毕。Hansen 指出，一个一千行的并发程序，它的每个成分只须二、三次编译，接着一次调试性运行就可交付使用了。

1.4 路径表达式

Habermann 等人于1974年提出了不同于管程的实现“秘书”的方法〔Cam 74〕，它使用SIMULA67的概念，使用一种叫做类的结构。类也由若干个过程组成，进程通过调用类中的过程实现进程间的同步。一个类中各个过程之间的同步控制是由一种称为路径表达式的机构实现。

路径表达式由下列几种运算组成，其中p，q，r为同一类的不同过程：

（1）顺序运算p；q；r——表示过程被调用时，必须严格地按此顺序进行，即任意进程调用q时，必须等待它进程先调用p。

（2）选择运算p，q，r——表示每次只能按某种合理的随机次序，调用其中一个。

（3）重复运算Path p_1，(p_2；（p_3，p_4））end——表示路径表达式按p_1，（p_2；（p_3，p_4））重复执行，即先由p_1和p_2中任选一个，若选中p_1，p_1调用结束，就可重复执行p_1，（p_2；（p_3，p_4））。若选中p_2，则p_2调用结束后，可执行p_3或p_4，一旦p_3或p_4调用结束后，方可重复执行p_1，(p_2；（p_3，p_4））。

（4）同时运算{p}——表示可有多个进程同时调用p。

每个类中包括有各种过程、数据说明和控制调用的路径表达式。Habermann 等还讨论了路径表达式的实现问题。

二、操作系统的可靠性

目前在运行的大型操作系统中，普遍存在着比较严重的可靠性问题。IBM 360的OS系统，每一个新版本都隐藏着1000个错误。在一个拥有二百万条指令的实时系统中，平均每天发现一个错误。美国APOLLO载人空间飞行的软件系统是世界上调试得最彻底的程序之一，然而在APOLLO 8，11、14中仍然发现了软件故障〔Boe 77〕。

影响操作系统可靠性的因素很多，最重要和最实质性的是操作系统的并发性和共享。由于这一特性，当系统出现一个错误时，错误结果往往不能重现，因此想通过动态调试来找出错误极为困难。第二个因素是现代操作系统很庞大，一个通用系统可多达几百万条指令。当几百人共同完成这样一件紧密相关的工作时，在接口上和在每个人的本体工作中不免会出现错误。第三个因素是采用不适当的设计方法。低劣的设计方法可能导致在设计上隐藏着较多的逻辑性错误，并且造成调试和修改都比较困难。

为了提高系统可靠性，目前在操作系统设计、实现和调试各个阶段都采取了一些措施。

2.1 设计阶段

Boehm 指出，三分之二的软件错误都是发生在设计阶段〔Boe 77〕。实现阶段的错误相对说来比较少，一个职业程序员平均每年才出现一个错误〔Mil 75〕。因此设计质量对系统可靠性有举足轻重的影响。下面介绍几项在设计阶段减少错误的措施。

1.改善设计方法

1968 年 Dijkstra 提出了层次设计方法，这种方法大大提高了系统可靠性〔Dij 68 B〕。层次设计最基本的特性是把系统分解成若干层，各层之间单向依赖，减少循环。由于这一特性，带来了一些好处。第一，系统内部结构简单，有利于减少设计上的逻辑错误。Dijkstra 在介绍第一个层次系统（THE 系统）时说，平均每 500 条指令中才发现一个编码错误。第二，系统易了解，对整个系统的了解变成对各层的局部了解。第三，由于减少循环，因此减少了因为调用关系、通信关系等引起的死锁。第四，易于调试，对系统的调试变成了对自底向上的逐层调试，这无疑提高了调试的彻底性。

管程不仅是一个同步机构，而且是一个重要的结构概念。由于管程最基本的特性是把共享数据以及在共享数据上的操作集中起来，而且对共享数据作严格的管理，因此为提高系统可靠性创造了有利条件。第一，引进管程概念之后，系统结构清晰，大大避免了与时间有关的错误。第二，管程本身的逻辑正确性可以通过形式或非形式证明来保证。第三，由于一个管程的局部变量只能由该管程的过程存取，因此编译程序可以检查出违反这一规则的错误。

2.建立设计分析工具

自动设计分析工具包括两个方面。一是说明语言，它被设计者用来说明其设计方案，即是说，设计者用说明语言来表达系统的输入、输出和处理特性。显然，若要对设计工作进行自动检查，那么必须按一定的规格对设计方案作描述，说明语言就是被提供来作这一用途的。二是一套分析程序，它被用来对设计方案进行检查，即检验输入、输出特性，以便断定数据是否有缺损和多余，数据使用上是否有矛盾。目前已有一些实验性的设计分析系统在运行。例如ISDOS〔Tei 74〕、LOGOS〔Ros 72〕都是包含了这种自动设计工具的系统。当前的设计分析系统往往是专用系统，例如 ISDOS 主要是用来帮助设计商业数据处理系统。

当前也往往用自动模拟工具来帮助设计。

2.2 实现阶段

操作系统程序编写完毕之后，在上机调试之前，可以通过下列手段发现程序错误。

1.程序正确性证明

1)程序正确性

程序正确性是讨论一个正确程序应具有的特性。

状态空间是指程序变元和控制变元（例如程序记数器）取值的值域。程序的每个

语句的执行将一个状态变为另一个状态。因此给定一个初始状态，程序从开始语句执行，就可得到一个或多个状态序列，即执行序列。程序的执行序列代表着程序的含义，甚至刻划了程序运行的环境。因此程序的各种特性一般都可由执行序列的特性来规定。

关于顺序程序的正确性，一般要研究如下特性。第一研究部分正确性，即，若程序的初始状态满足性质P，则程序的终止状态一定满足性质Q。虽然程序中包括终止语句，但是部分正确性并不保证程序必定达到终止状态。第二研究终止性，即自一定的初始状态出发，程序一定达到终止状态。第三研究完全正确性。既终止又部分正确的程序称为完全正确。此外，还研究两个程序的等价和同构。

并行系统除了讨论程序变元所具有的特性之外，还研究其它两种特性。第一是研究加工结果的决定性，即自一定的初始状态出发，无论出现那种可允许的执行序列，都能达到具有同一特性的相同结果。第二是研究系统的控制或调度特性，如无死锁问题，是否满足实时要求问题，调度策略的评价问题等等。

2)程序特性的数学陈述

1967年Floyd 提出用归纳断言方法陈述和证明程序正确性后，迄今多数都使用这种方法。所谓断言，就是有关变元的一个关系（或称谓词）。顺序程序的特性一般可用程序变元的断言来描述，而并行系统的特性涉及控制、调度等，必须使用辅助变元。所谓辅助变元是一些不在原来程序中出现的变元，在添入后的程序中，它可被赋值，但不能影响原有程序变元的取值，也不能影响原来程序的执行序列的先后顺序〔How 76〕、〔Kel 76〕、〔Owi 76 A、B〕。

3)程序正确性证明

证明方法可分形式化演绎系统和非形式的数学推理两种。

1969年，Hoare在Floyd方法的基础上提出了证明顺序程序正确性的演绎系统〔Hoa 69〕。设程序由赋值、复合、如果则、当等四种语句组成。与这些语句相对应，建立了若干公理和推理规则，对被证明的程序反复应用这些规则，从而证明其正确性。

Hoare 自己首先将顺序程序的形式演绎系统推广至包括并行语句和条件临界区语句的并行程序〔Hoa 72〕，增加了并行语句和条件临界区语句两条推理规则。但是Owicki证明了Hoare系统的不完全性，给出了一个完全的演绎系统，这是有关并行程序的第一个完全系统。系统中除了并行语句、条件临界区语句推理规则外，还增加了一条强有力的隐变元推理规则〔Owi 76 A〕。1974年Hoare又对管程概念作了精确刻划。并提出了相应的演绎系统〔Hoa 74〕1976年Howard又改进了Hoare关于管程的演绎系统〔How 76〕。

用演绎系统证明程序特性时，如何建立中间断言，特别是不变式断言，这是证明中最富创造性部分，是证明的难点。

关于非形式数学推理，一般认为有三种方法：归纳、穷举和抽象。归纳法者归纳于执行序列的长度。在使用归纳法证明时，必须寻找归纳假设，也就是执行序列的任一点上恒成立的不变式。这是归纳法的难点。穷举法为穷举执行序列的各种可能。抽象法即

抽去和所论性质无关的细节。抽象法对于并行程序正确性的证明尤为重要。一个并行程序自给定的初始状态出发，由于并发现象，可能产生多种甚至无穷多种执行序列，因此一个实际可行的证明方法，必须有强有力的抽象工具。〔Lip 75〕、〔Doe 76〕是证明系统无死锁时所用的抽象方法。但是这方面的发展尚为不足。

4)程序正确性证明的应用

程序正确性证明的研究起自 Von Neumann。1967年 Floyd 的文章〔Flo 67〕发表以来，顺序程序正确性的研究更有了蓬勃发展。已产生各种自动验证系统，据宣布已用此类系统证明了一个由两千条高级语言语句组成的程序。但是目前尚属试验阶段。并行程序正确性的研究开始得更晚，到实际应用还有很长距离。文末列有近年来有关文献〔Ash 75〕、〔Bel 74〕、〔Doe 76〕、〔Gri 76〕、〔Hoa 74〕、〔How 76〕、〔Kel 76〕、〔Lam 77〕、〔Lip75〕、〔Owi 76 A B〕、〔Ros 75〕。

系统完全性的证明从理论上证明了不变式的存在，并抽象地构造了不变式，但是这种构造是很难实际使用的，因此出现了有关断言的自动综合和其它自动证明的研究〔Lon 75〕、〔Mor 77〕、〔Weg 77〕。目前有实际应用价值的是一些程序运行时的验证系统。如Stucki建立的PET 断言验证系统〔Stu 73〕，程序员把程序连同插入在程序中的断言输入系统，并执行程序，PET 自动地在程序执行过程不断地检查断言是否成立。

2.程序的自动分析和检查

一般说来，在系统程序编写完毕后，由程序员对程序进行人工的静态检查。但是这种检查必须非常细致，才能产生效果。另方面，对于大型操作系统来说，由于检查量过大，人工往往难于胜任。因此产生了一些自动检查系统，把检查工作交由计算机本身去完成。检查系统在不执行代码的情况下对源代码进行分析和检查。自动检查系统由若干工具构成，每个工具负责一种检查职能〔Ram 75〕。主要工具有：

1)代码分析。它完成源程序的语法分析，以便找出有错误的程序结构。它与编译系统的全面的语法检查不同，它测重于某些语法分析方面，目的在于检查我们感兴趣的程序结构，寻找结构上的错误。

2)程序结构检查。结构检查包括三个方面。第一产生程序图。程序图是一个有向图，节点代表语句，弧代表控制流。程序图反映各语句之间控制转移关系。程序图实际上是用连接矩阵存贮起来的。第二是检查结构，即检查程序图，寻找结构上的缺陷，例如不合理的循环嵌套，无引用的标号，不可到达的语句，没有后继者的语句等等。第三是循环终止检查。

3)模块接口检查。代码分析和结构检查是局部性检查，即只检查各个模块本身。模块接口检查是全局性检查，即检查各模块之间数据说明的不一致性，各模块之间连接的不合理性。例如，被访问的模块和访问模块的参数个数应相同，参数类型应相同。

4)事件序列检查。它从程序代码中抽取出所关心的各事件，然后把一事件序列与固有的序列比较，发现程序的次序性错误。例如使用文件的过程应是按建立、打开、读

写、关闭、撤消这一顺序进行，若违反这一规定，则使用顺序有错。

用自动检查工具来检查操作系统程序是有好处的。首先它是一种排错手段，能够发现程序中一些简单的错误。〔Ram 75〕中指出了自动检查系统的使用对软件错误发现率的影响，这一影响可用图 2.1 表示。其次，由于操作系统的程序量大，若对它进行人工检查，则很费时间，且容易遗漏。由自动检查系统来做这一工作，可以加速检查过程，使程序员少做琐碎的重复性工作，使他们能够集中精力于疑难问题的解决上。目前已有一些自动检查系统在运行，例如PACE、RSVP等〔Ram 75〕。

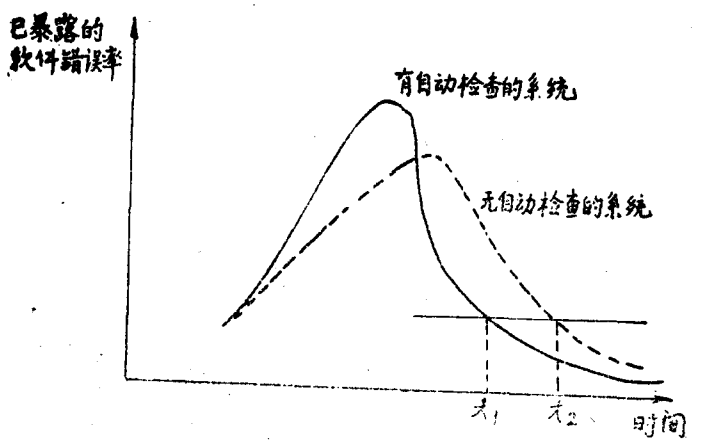

图 2.1

尽管自动检查系统有许多好处，但是想建立一个实用的系统，也并非轻而易举的事。首先，程序的结构、各模块之间的关系等都比较复杂，因此建立起一个行之有效的、易于实现的检查系统就较难。例如在代码分析方面，如果要求对源代码作完全的语法分析，则代码分析程序就会象编译程序本身一样复杂。其次，由于存贮器和 CPU 时间的限制，往往使自动检查系统不适合于检查包含很多语句的大型程序。例如在结构检查中，其复杂性与语句个数的平分成正比。

2.3 调试阶段

调试与实现阶段的程序正确性证明和自动检查工具不同，后者是在不执行代码的情况下对源代码进行正确性验证，而调试则通过程序的实际执行来发现错误。调试包括检测和排错两方面，检测是通过程序的执行发现错误现象，而排错则是确定错误位置和修正错误。

调试手段有一定的局限性，因为它难于发现和纠正全部错误。但是在程序证明、自动工具还不成熟的情况下，调试仍是保证系统可靠性的最主要手段。调试办法主要有：

1.传统调试

当前最基本的调试办法仍是传统调试。所谓传统调试就是选择适当的调试数据作为被调试程序的输入值，然后执行程序，得到计算结果，从结果值判断程序正确与否。目前的研究重点是，如何选择调试数据，使得程序能执行所有的语句和穿越所有可能的执行路径，或者暴露全部错误。

1)自动产生调试实例。可以建造一些自动工具来帮助程序员产生调试实例，以满足上述要求。Miller 和Paige〔Mip 74〕已建立一种算法，它主要是确定穿越一个执行

通路的条件，以此自动找出满足以上要求的最小调试数据，并提供给程序员。该算法是以程序图的分析为基础。

2)调试数据选择理论。Goodenough 提出了这一理论〔Goo 75〕。他定义了完全调试、可靠调试和有效调试三个抽象概念，并且设计了一种具体的调试办法。这一办法想要达到的目标是确定一种选择调试数据的方法，使得程序如果正确地处理这些调试数据，则也将正确地处理所有数据。为了达到这一目标，必须让调试数据对准被调试程序的所有错误。Goodenough 认为程序出错的主要原因是程序不能正确地处理条件和条件的组合。因此他的调试办法的最主要方面就是调试谓词，它阐述了与程序的正确操作有关的条件和条件组合。换言之，调试谓词指出了一个程序的那些方面要被调试。调试谓词从决定表中得到。决定表的每一列代表一种调试谓词，即程序在执行过程中出现的条件组合。而决定表又是从程序的说明和实现中得到的。Goodenough 证明了他的方法是完全的、可靠的和有效的。

2.符号调试

传统调试是数据调试，即向被调试的程序提供样品数据，该程序执行后产生数据结果，若结果与预想的相同，则程序被认为是正确的。〔Kin 75〕中提出符号调试的办法。在符号调试时，向被调试的程序提供的不是输入数据，而是符号，程序输出的计算结果不是数值，而是符号公式。符号调试与传统调试一样是通过程序的执行来进行的，但是符号调试只是进行一次“符号执行”，这一次执行就相当于多次的（通常是无限次的）传统的调试性运行，因此它比传统调试更有效。

为了使程序能够接收输入符号和产生符号公式，首先必须提供一种程序设计语言，并且必须对“程序的执行”赋予新的含义。“程序的执行”包括两方面，一方面是通常意义下的执行，另一方面是扩充该语言的语义，使得能够定义符号执行。因此，在执行中在不涉及到符号的情况下按通常的意义执行，在涉及到符号时按符号执行的定义进行。其次要求在调试时不修改源程序本身，即要求编译程序把源程序编译成调态或算态形式，或者调算统一的形式。

实现符号调试时要解决几个问题。第一是符号输入。向被调试的程序输入符号，以代替真实的数据。其中“输入”是指从外界向程序提供任何信息，包括通过参数得到的信息、全局变量、明显的读语句等等。输入符号不应与被调试程序的变量名混同。输入符号与变量不同，一个变量在程序的执行过程中可以取不同的值，而一个输入符号是代表某种未知的固定值。第二，必须解决在符号执行的情况下，如何进行表达式计算和条件转移。在符号执行情况下，表达式的计算不再是算术运算，而是代数运算。例如，若变量A和B的输入分别为符号α和β时，则执行赋值语句 C：=A+2*B时，得到符号公式（α+2*β）。若接着执行语句 D：=C−A，则执行此语句后D的值是 2*β。对于程序设计语言中的所有计算操作，都可以作类似的符号推广。但是当计算公式太长时，必须用压缩方式表示它。当B为符号情况下遇到条件转移语句

if B then S_1 else S_2

时，则两条通路s_1和s_2都需要执行。为了处理多次执行条件转移的情况，每次执行条件转移时，必须把该语句的执行状态（变量值、语句计数器等）保存起来。可以用执行树的办法实现保存工作。在交互性符号执行系统中，用户可以明显地要求保留执行状态和选择执行通路。

King 等建立了一个试验用的交互性符号执行系统 EFFIGY，它用于调试用简单的PL/I 型程序设计语言编写的程序。该语言的变量类型只限于整值变量和向量（一维数组）。

3.程序的动态分析

可以对程序在运行时的性态进行分析，以帮助调试。例如，可以对程序性态进行监督。在程序的适当处插入一些监督程序，以便收集程序运行时的性态。监督的内容有：范围检查，即变量的取值范围和其值的变化规律；频率监督，即穿过某些代码段的频率；执行通路的跟踪，即把被调试过的通路记录下来。监督程序的插入是自动进行的。

三、保护和容错

操作系统不仅在设计、实现和调试阶段应有可靠性措施，在运行阶段亦应有相应措施。目前在运行阶段的主要措施是使系统具有保护和容错能力。

3.1 系统保护

1.保护的含义

保护的含义是：①在一个多用户的系统中，必须设立一些机构，来标识用户；②保证一个用户不致于有意或无意地干扰和破坏其他用户或系统的信息；③保证系统故障局部化，防止其蔓延。在多道程序出现以前就已有保护措施，但是只在1966年 Dennis等提出“能力”概念之后，才形成一套保护原理的〔Dev 66〕。后来Lampson 把能力概念应用于实际系统中〔Lam 69〕，并提出了一套保护模型。

2.保护原则

Denning〔Den 71〕认为，一个保护系统应由几部分组成。第一部分是一组客体，一个客体是系统中必须受到保护的任一实体，例如段、页、文件等。第二部分是一组主体，主体是系统中可以存取客体的任一实体，例如进程。因为各主体之间也必须互相保护，因此主体本身也是客体。第三部分是一组存取规则，它指明了一个主体可以怎样存取客体。一个保护系统要达到两个目标，一是隔离，即对每个主体进行限制，使它只能使用那些授权于它的客体；二是控制存取，即允许不同主体对同一客体具有不同的存取权。

为了达到隔离和控制的目标，必须有一种描述存取规则的手段。存取规则可以用存取矩阵来说明。矩阵的每一行对应于一个主体，每一列对应于一个客体，矩阵的每一个元素是一个字符串，它指明了有关主体对相应客体的存取权。存取矩阵在表3.1中给出。

每一种客体类型都有一个控制机构与之相联系，而且必须通过这种控制机构才能去

		主体			文件		进程		终端	
		S_1	S_2	S_3	F_1	F_2	P_1	P_2	T_1	T_2
主体	S_1	持有者控制	持有者		读	读持有者	唤醒	唤醒	读写	
	S_2			控制	写	执行				读
	S_3					写	停止		写	

表3.1 存取矩阵

存取相应类型的客体。

3.保护的实现

存取矩阵的存放方式有四种。第一种是按行存放，即每个主体一张表，称为能力表，表中每一项称为能力。当一个主体S企图存取某一客体X时，X的控制机构查询能力表，找出相应存取权。

第二种办法是矩阵按列存放，即每个客体都联系一张表，称为存取控制表。

第三种方法是能力表的一种改进。每一个主体与一串表相联系。每一种可能的存取权对应一张表，主体S相对于存取权α的表记作Sα，且用一个二进制字表示。当且仅当主体S对第i个客体X_i具有存取权α时，该二进制字的第i位才为1。当S企图按α权去存取客体X_i时，控制机构检验Sα的第i位是否为1。

第四种办法是能力表和控制存取表的一种折衷。每个主体S都和一个客体“钥匙表”相联。这个表由许多二元组（x，k）组成，其中x为客体名字，k是一把钥匙。这张表对S来说是不可存取的。每把钥匙是某种特定存取权α的一种编码表示，且只在对应于k的α是在A〔s，k〕中时，才把（x，k）放入到S的客体钥匙表中。与每一个客体x相联系的是一张“锁”表，每把锁是某个α的编码表示。当S企图对x进行存取时，x的控制机构检查S的钥匙中是否有一把与x的某一把锁相配，只在相配时才允许存取。

3.2 容错操作系统

1.操作系统中的容错概念

所谓容错是指在硬、软件发生故障的情况下，系统仍然具有继续运行的能力。它往往包含三方面功能。第一是故障约束，即限制过程或进程的动作，使它们不能有比它完成任务所需要的还要多的能力和很大的存取范围，不允许它们在不一致的数据上操作，防止错误被检测出来之前继续扩大；第二是故障检测，即对信息和对过程或进程的动作进行动态检验，以便查看被操作的数据是否一致，动作是否越出规定的范围。最理想的是精确地定出出错位置和错误原因，至少要求确定出错范围；第三是故障恢复，即更换或修正失效的部件(硬件或软件)，把系统恢复到某种状态，并再启动之，让其继续运行。

容错概念是1965年以前提出的，但是一直只应用于硬件的容错设计。由于硬件元件的失效率正在下降，而软件故障却日益严重，因此很自然就要考虑软件容错措施。在

1975年国际软件可靠性会议上 Randell 等讨论了这一问题，1976年 Denning 等人更加明确地提出容错操作系统问题。

2. Denning 提出的容错系统〔Den 76〕

1)容错原理。Denning 提出以能力体系结构为基础的容错原理，它包括四方面。一是进程的隔离。每个进程的能力不能超出为了完成其任务所需的能力范围。二是资源控制。当把一个资源分配给某一客体时，把该资源置成所要求的状态，释放时置成空状态，以防止进程之间通过留在资源内的残余值互相干涉。三是决定的验证。操作系统中有许多决定，这些决定必须受到验证，以便发现是否有偏差。四是错误恢复。错误恢复首先力求修正已发现的错误，若做不到这一点，则对系统进行再配置。

2)进程的隔离。进程的隔离手段是以能力为基础的。图3.1解释了存取内存时的隔离。假定域为 d 的进程用局部名j对 x 段第 w 单元进行A存取。其过程如下。首先，根据域寄存器 D 的内容d和局部地址寄存器 LA 的内容（j，w）找到CL（d）中的（c，x），若要求的存取动作A与规定的权力 c 不匹配，则存取类型出错；其次根据x和CMT找到（p，b，l），其中 p 表示该段是否在内存。若 p = 0，则表示缺段；再次，检查 $w \geq 1$ 是否成立，若成立，则越界存取；最后，对内存地址为b + w的信息完成动作 A。过程之间的相互调用可以用类似的办法实现。

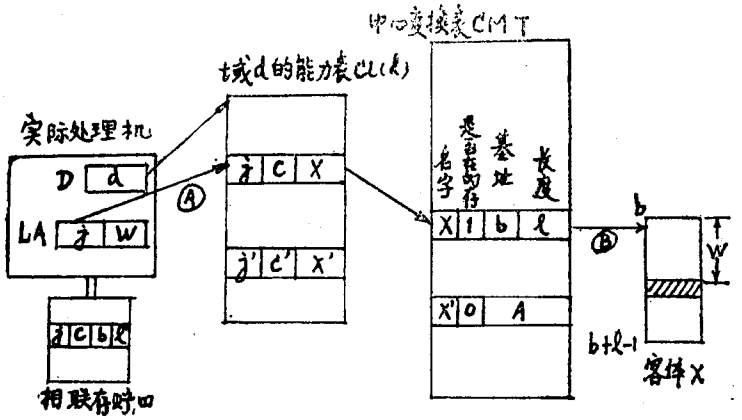

图 3.1存取内存

3)资源控制。进程之间有可能通过资源内部状态中的残留值而进行非法的信息交换。例如，在虚拟存贮器情况下，当一个进程获得一个页面时，可能获得了释放者残留在该页面中的信息。杜绝这一现象的办法很简单。每当释放资源时，把资源的内部状态清除掉。

4)决定的验证。假定进程U作出决定，要在客体V上完成动作A，为此它把消息（A，V）发送给V的控制者。系统把发送者的标识符U加进去，即把三元组（U，A，V）发给控制者。控制者做一致性检查（验证动作A是否与V的状态一致）、来源检查（验证是否允许U用动作A去存取 V）、目标检查（验证V是否为接收该决定的客体）和传输检查（验证已收到的（U，A，V）是有效的）。如果所有的检查都获得通过，则控制者才完成所要求的动作。

5)错误的恢复。错误的恢复以层次操作系统为基础。用 M_0，M_1，…M_k 表示各级抽象机。图3.2解释了第i级。M_i的各个过程P_{i1}，P_{i2}…，P_{il}管理着一些资源和数据，

这些资源和数据用结构D_i描述。对P_{i1}，…，P_{il}的存取受到进入能力（ent）的控制。每一级有一个错误恢复函数E_i当发现一个错误时，经由E_i向第i级报告。E_i递归地处理这一报告：①若已发现的错误有可能已传播到M_{i-1}，则E_i调用E_{i-1}，E_{i-1}把M_{i-1}置成一致的状态；②E_i行D_i进行试验，然后向访问者报告在修正错误方面成功的程度。在第②步骤中，E_i可以安全地使用M_{i-1}的各个过程，因为从①知道，M_{i-1}已处于一致状态。在E_i已报告M_i为一致之前，不允许任何进程执行第i级或更高级的指令。若M_i的一个过程P检测出数据结构D_i的一个不一致性，或者从M_{i-1}收到一个出错报告，它就调用E_i。只当E_i无法修正错误时，P才向它的访问者报告。

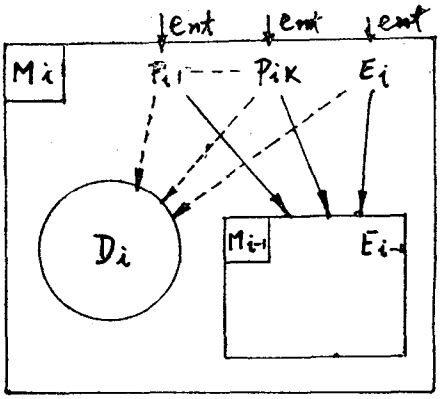

图 3.2

3. Randell 提出的容错系统〔Ran 75〕

Randell 提出了冗余设计的容错方案。此方案类似于硬件的备分设计。在系统运行时，对每个程序成分的运行结果进行“验收试验”。若验收通不过，则换一个备分的成分，以代替出错的成分。这种备分不是主成分的复制品，而是能代替主成分的具有不同逻辑设计的新成分，它能对付得了导致主成分失败的那种环境。

1)顺序进程内部错误的恢复。一个顺序进程被分成若干块，每块称为恢复块。每个恢复块由验收程序和若干个替换块组成。当进入一个恢复块时，保留进入前夕系统的状态。从任何一个替换块退出时，进行验收试验。若验收不合格，则恢复上一次进入恢复块时的系统状态，并且进入下一个替换块。若所有替换块的验收均不合格，则认为整个恢复块失效。若有一个替换块通过了验收，则抛弃该恢复块中此后的替换块，执行下一个恢复块。恢复块允许多重嵌套。

验收试验的目的是要检查恢复块所完成的操作是否达到满意的程度，不是对恢复块所完成的操作作"绝对正确性"检查。它是通过考察恢复块所能存取的变量值来完成的。

第一个替换块是基本块，它完成理想的操作。其它替换块以某种不同的方式（简单可靠）完成所希望的操作。一般说来，越靠后的替换块越简单，其操作结果越不理想，但是仍可接受。

在进入恢复块时系统状态的保存和退出时的恢复完全是自动的。为了减少保留和恢复时的开销，不是保存所有变量值，而只是保留该恢复块要修改的那些全局变量值。

2)交互进程的错误恢复。在进程之间有交互的情况下，例如两个进程传递消息时，其中一个进程的恢复，引起其它进程后援，甚至所有的进程都后援到初始点。这称为多米诺反应。为了在进程交互情况下实现可恢复性，采用会话结构。跨越两个或多个进程的恢复块称为会话块。各个进程可以一致地进入一个会话块。在一个会话块内，这些进程可以自由通讯，但不能与别的进程通讯。在会话块结束处，所有的进程都必须满足它

们各自的验收试验，并且在都满足之前，它们中那一个都不能前进。若其中一个进程验收失败，所有有关进程都必须后援到会话块的起点。

参 考 文 献

〔Age 77〕 Agerwala, T., Some Extended Semaphore Primitive, Acta Informatica 8：3(1977)

〔Ash 75〕 Ashcroft, E. A., Proving Assertions about Parallel Programs, Jcss 10：1 (Feb. 1975)

〔Bel 74〕 Belpaire, G. and Wilmotte, J. P., Correctness of Realization of Levels of Abstraction in Operating Systems, Lecture Notes in Computer Science 16(1974)

〔Boe 77〕 Boehm, B.W., Software and Its Impact：A Quantitative Assessment, Software Design Techniques (Second Edition), Institute of Electrical and Electronics Engineers, Inc., 1977

〔Cam 74〕 Campbell, R. H. and Habermann, A. N., The Specification of Process Synchronization by Path Expression, Lecture Notes in Computer Science 16, 1974

〔Den 71〕 Denning, P. J., Third Generation Computer Systems, Computing Surveys vol. 3, №4, 1971

〔Den 76〕 Denning, P. J., Fault-Tolerant Operating Systems, Computing Surveys vol. 8, №4, 1976

〔Dev 66〕 Dennis,J. B. and Van Horn, E. C., Programming Semantics for multiprogrammed Computations,Comm. ACM 9,3 (March 1966)

〔Dij 65〕 Dijkstra, E. W., Solution of a Problem in Concurrent Programming Control, C. ACM 8：9 (1965)

〔Dij 68 A〕 Dijkstra,E. W.,Constructive Approach,BIT 8 (1968), 174—186.

〔Dij 68 B〕 Dijkstra, E. W., The Structure of the THE Multiprogramming System, C. ACM, 1968, P.341

〔Dij 71〕 Dijkstra, E. W., Hierarchical Ordering of Sequential Processes, Operating System Techniques 1971

〔Doe 76〕 Doeppner Jr, T. W., On Abstraction of Parallel Programs, Proc. of 8th Annual ACM Symp. On Theory of Computing, May 1976.

〔Flo 67〕 Floyd, R. W., Assigning Meaning to Programs, Proc. Symp. Appl. Math., 19, 19—32.

〔Goo 75〕 Goodenough, J. B. and Gerhart, S. L., Toward a Theory of