



[12] 发 明 专 利 说 明 书

[21] ZL 专利号 98119611.X

[45] 授权公告日 2003 年 8 月 27 日

[11] 授权公告号 CN 1119746C

[22] 申请日 1998.9.17 [21] 申请号 98119611.X

[30] 优先权

[32] 1997. 9. 18 [33] JP [31] 253889/1997

[71] 专利权人 索尼公司

地址 日本东京都

[72] 发明人 村田诚二

[56] 参考文献

US5632032 1997.05.20 G06F9/44

US5666523 1997.09.09 G06F9/46

审查员 林松岭

[74] 专利代理机构 北京市柳沈律师事务所

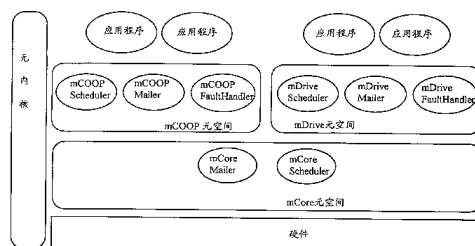
代理人 马 莹

权利要求书 1 页 说明书 21 页 附图 14 页

[54] 发明名称 数据处理方法

[57] 摘要

一种数据处理方法，用于采用纳内核的操作系统，该方法大大减少调度程序调用次数而不影响系统操作。每当发生从一线程到另一线程转接的上下文转接时，该操作系统记录上下文转接历史。当控制线程执行序列的调度程序被调用时，由调度程序所管理的队列被更新回所记录历史。



-
1. 一种数据处理方法，包括如下步骤：
当发生上下文转接以转接所执行线程时，记录该上下文转接的历史；以及
5 当控制线程执行序列的调度程序被调用时，通过跟踪回所记录的上下文转接历史更新由所述调度程序管理的队列。
 2. 按照权利要求 1 所述的数据处理方法，其中更新队列的步骤包括：
在引起要求在所述队列中改变序列的上下文转接的线程上设置一个预定标志；以及
10 当引起一新上下文转接的线程具有所述标志时响应于所述新上下文转接的发生调用所述调度程序改变所述队列中的序列。
 3. 按照权利要求 2 所述数据处理方法，其中所述其上设有标志的线程不同于所述调度程序的调用的类型。

数据处理方法

5

技术领域

本发明涉及数据处理方法，详细涉及由操作系统实现的调度管理机构。本发明还涉及存有此种操作系统的记录介质，以及采用此记录介质的数据处理装置。本发明一些特征的细节在本发明被授权人于 1996 年 7 月 12 日提交的名称为“数据处理方法及装置”的欧洲专利申请号为 0,753,811A1 的专利申请的中有所说明，在日本 1997 年 7 月 14 日申请的号码为 178625/95 的专利申请中提出了对该申请的优先权要求。其全部公开内容在此引作参考。

背景技术

15 操作系统可以被分类成如下三类：采用大内核(fat kernel)的操作系统；采用微(micro)内核的操作系统；和采用纳(nano)内核的操作系统。

在大内核操作系统中，所有的系统功能都装在一个大内核中，参看图 12，当应用程序即以改变线程状态的方式请求操作系统提供一项服务时，系统调用便以内核中调用对应于所请求服务的模块，如“邮件程序”或“同步”。相应地为了使此模块改变线程的状态，执行一个所谓的功能调用以调用“调度程序”，调度程序控制着线程的执行序列以管理这些线程的调度，这样改变线程状态的操作被执行。

为了把各种功能都放入使用大内核的操作系统，程序员必须具有足够的内核知识，因为所有的功能都必须放入内核中。另外由于这些功能都放入了内核，这些功能的改变不易进行。这样，使用大内核的操作系统通常具有有限的通用性。

现在再来看一下使用微内核的操作系统。微内核仅仅容纳基本的如硬件控制功能之类的功能，而其它的操作系统的功能都以外部方式实现。这种类型的操作系统因而不存在大内核型操作系统所特有的有限通用性的问题，因为只有选择的功能被装入内核中。

如图 13 所示，微内核操作系统不仅允许应用程序还允许提供操作系统功

能的模块如“同步”“邮件程序”和“调度”等模块运行在微内核中。这样，这些提供了操作系统功能的模块也在装入到微内核中的调度程序的调度控制下。

在微内核类型的操作系统中，应用程序和模块之间的交互(interaction)作用是由微内核提供的通讯机构“邮件程序”来完成的，同时提供操作系统功能的模块通过微内核中提供的接口间接调用微内核中的调度程序控制应用程序的执行。

现有参照图 14，在纳内核型操作系统中，仅仅从事上下文转接(context switch)的上下文转接机构从微内核的内容中被提取。该被提取的上下文转接机构形成一个纳内核。

这样，在纳内核型操作系统中，依赖于 CPU 类型等的上下文转接机构被提取出来并被抽象化。这便提供了很大的编程的方便性，以及高的内核灵活性。

为了获得增强的操作系统的多用性，以及高内核灵活性，和编程的方便性，微内核型操作系统优于大内核类型操作系统，纳内核型的操作系统优于微内核型操作系统。

然而，微或纳内核型操作系统要求大量的上下文转接。这是因为微内核的通讯机构和调度程序必须在任何一个运行实体的每一次运行中被调用，这些运行实体可以是在该操作系统上运行的应用程序，也可以是提供系统功能的对象等等。这样，微或纳内核类型操作系统的整个性能低于大内核类型操作系统的性能。

使用纳内核系统的操作系统甚至还要求用于调用通讯机构和调度程序的上下文转接。因此这种类型的操作系统由于需要使用大量上下文转接进一步增加了成本，使成本高于微内核类型操作系统的成本。

这样，纳内核的使用由于增加了上下文转接数量不良地影响了整个操作系统的性能。解决这个问题的办法是在执行上下文转接时抑制调度程序的调用。然而仅仅抑制调度程序的调用影响了调度管理以及系统操作。

更进一步地，调度程序的调用的省略使调度程序不能接收调度管理所需要的信息。导致实际系统状态和调度程序的所掌握的状态的不一致。这样，调度程序便不能执行正常的调节管理。调度程序调用的省略还带来如下问题。一般来说，在调度程序的调度下的线程中的队列改变时，调度程序执行再调

度以使较高优先权的处理被优先地执行。这样的再调度在调度程序的调度省略时不被执行，这样高优先级的处理或许被不希望地延迟。

发明内容

5 因此，本发明的一个目的是提供一种使用纳内核的数据处理方法，其中、调用调度程序的数量达到最小而不影响采用此方法的系统的操作。本发明的另一个目的是提供一种记录有实现该方法的程序的记录介质，以及使用该记录介质的数据处理装置。

 为达到这些目的，按照本发明的一个方面，本发明提供了一种数据处理
10 方法，其中，当上下文转接发生以转接执行的线程时，上下文转接的历史被记录。之后，当控制线程执行序列的调度程序被调用时，由调度程序管理的队列按记录的上下文转接历史更新回来。

 按本发明的另一个方面，本发明提供一种包含以计算机可读形式出现的程序的记录介质，其中，所述程序使得计算机能够实现如下步骤：当发生上
15 下文转接以转接所执行线程时，记录该上下文转接的历史；以及当控制线程执行序列的调度程序被调用时，通过跟踪回所记录的上下文转接历史更新由所述调度程序管理的队列。

 按本发明的另一个方面，本发明提供一种包含以计算机可读形式出现的程序的记录介质，其中，所述程序使得计算机能够实现如下步骤：当发生上
20 下文转接以转接所执行线程时，记录该上下文转接的历史；以及当控制线程执行序列的调度程序被调用时，通过跟踪回所记录的上下文转接历史更新由所述调度程序管理的队列，其中更新队列的步骤包括：在引起要求在所述队列中改变序列的上下文转接的线程上设置一个预定标志；以及当引起所述新上下文转接的线程具有所述标志时响应于新上下文转接的发生调用所述调度
25 程序改变所述队列中的序列。

 按本发明的另一个方面，本发明提供一种包含以计算机可读形式出现的程序的记录介质，其中，所述程序使得计算机能够实现如下步骤：当发生上
 下文转接以转接所执行线程时，记录该上下文转接的历史；以及当控制线程执行序列的调度程序被调用时，通过跟踪回所记录的上下文转接历史更新由
30 所述调度程序管理的队列，其中更新队列的步骤包括：在引起要求在所述队列中改变序列的上下文转接的线程上设置一个预定标志；以及当引起所述新

上下文转接的线程具有所述标志时响应于新上下文转接的发生调用所述调度程序改变所述队列中的序列，并且其中，所述其上设有标志的线程不同于所述调度程序的调用的类型。

- 按本发明的另一个方面，本发明提供一种数据处理装置，包括：记录部分，当发生上下文转接以转接要执行的线程时，记录该上下文转接的历史；以及队列更新部分，当控制线程执行序列的调度程序被调用时，通过跟踪回所述上下文转接的所述历史更新由所述调度程序管理的队列。

附图说明

- 10 参照附图进行说明后本发明的目的和特征及优点更清楚。

图 1 是使用本发明的 TV 系统的配置概要说明图；

图 2 是使用本发明的操作系统层次结构说明图；

图 3 是一般调度管理系统机构说明图，详细说明从对象 A 发送信息到对象 B 的过程，及操作系统层次结构图；

- 15 图 4 是一般调度管理机构的说明图，以时间图表方式说明了对象 A 与 B 之间的交互使信息被从对象 A 发送至对象 B；

图 5 是一个使用本发明的调度管理机构说明图，详细说明从对象 A 向对象 B 发送消息的过程，以及操作系统的层次结构图；

- 20 图 6 是一个本发明的调度管理机构说明图，以时间图表的方式说明了对象 A 与对象 B 之间的相互联系以便从对象 A 到对象 B 发送消息；

图 7 是线程状态变化框图；

图 8 是表示延时队列机构的算法流程图；

图 9 是在延时队列机构中设置“当前线程”的算法流程图；

- 25 图 10 是通过异步调度程序调用机构从对象 A 到对象 C 发送消息的过程说明图；

图 11 是一个时间图表，说明通过异步调度程序调用机构对象 A 和对象 C 之间的交互以实现从对象 A 向对象 C 发送消息；

图 12 是使用大内核的操作系统说明图；

图 13 是使用微内核的操作系统说明图；

- 30 图 14 是使用纳内核的操作系统说明图。

具体实施方式

以下将结合附图对本发明的优选实施例进行说明。

图 1 以举例的方式说明了本发明所采用的硬件结构。虽然下面专门讲述的是一个 TV 系统，应能理解，本发明可以被应用于各种各样的使用操作系统的数据处理装置中，例如，本发明可以被有效地使用于声像系统、即所称的 AV 系统，办公室用具、计算机等装置，以及如下要说明的电视系统。

参照图 1，做为本发明的数据处理装置的实施例的 TV 系统通过空中(天线)或通过电缆从台站接收信号。TV 系统按所接收到的信号在 CRT 或液晶显示装置上显示图像，同时从扬声器向外输出声音信号。

10 此 TV 系统除一般的功能外还具有接收从外部输入的程序和数据的功能。特别是图 1 所示 TV 系统具有通过总线/I/O 桥 1 连接至总线 2 上的 TV 功能部件 3、通过总线/存储器桥 4 连接到总线 2 上的处理器 5、同样通过总线/存储器桥 4 连接到处理器 5 上的 ROM(只读存储器)6 和 RAM(随机存储器)7、连到总线 2 上的操作面板 8 和也连到总线 2 上的外部存储设备 9 和通讯设备 10。

15 TV 功能部件 3 具有根据空中或电缆接收的信号产生图像和声音的功能，并且被通过总线/I/O 桥 1 连接至总线 2 上以和系统其它的设备进行通讯。

处理器 5 控制着构成 TV 系统的各种设备的操作。为达到这个目的，处理器 5 通过总线/存储器桥 4 连至总线 2 上。ROM6 和 RAM7 也通过总线/存储器桥 4 连接至处理器 5 上。

20 ROM6 中存储着用以使处理器 5 执行控制的操作系统、应用程序等。在本实施例中，操作系统使用了一个纳内核并且是面向对象的。这样，操作系统也是一个面向对象的纳内核操作系统。

RAM7 被用做是一个处理器 5 的工作区。这样，处理器 5 当执行操作系

统和应用程序时将 RAM7 用作一个工作区，这样便控制了各种构成 TV 系统的设备的操作。

操作面板 8 起到输入设备的作用，它接收用户输入的操作指令，如用于通道转接或音量控制的指令。特别是操作面板 8 具有一个如带有多个键钮和如鼠标器等指点设备的控制台的输入设备。通过操作面板 8 输入的信号通过总线 2 和总线/存储器桥 4 传送至处理器 5。按照通过操作面板 8 输入的信号，处理器 5 执行各种计算以控制部件设备。

可以是硬盘设备的外部存储设备 9，存储着通过通讯设备 10 从外部系统上下载下来的图像数据、控制数据和应用程序，这样，通讯设备 10 起到 I/O 部件的作用，承担着与外部系统信号转接的作用，例如它可以包含，MODEM 或终端适配器。

TV 系统不仅执行由 TV 功能部件 3 提供的一般的 TV 功能还具有各种其它的功能。例如，TV 系统可以接收通过通讯设备 10 来自外部系统的程序和数据。这个功能使操作系统和设备驱动程序的版本的更新简单只要通过通讯设备 10 从外部网络接收新的软件模块即可。

TV 系统在处理器 5 的控制下执行存储在 ROM6 中的操作系统，并且允许存储在 ROM6 或外部存储设备 9 中的应用程序运行在操作系统上，这样，控制了部件设备。这样 ROM6 起到计算机可读存储介质的作用。它存储着操作系统和驱动程序。然而操作系统可以被存储在 RAM7 中或外部存储设备 9 中。当需要重写操作系统和设备驱动程序时，最好采用 RAM7 或外部存储设备 9 中的存储。

以下将说明 TV 系统的软件环境。

该 TV 系统中使用的操作系统为一采用了纳内核的纳内核类型操作系统。如图 2 所示，其中使用了总核来作为负责上下文转接的纳内核。

此纳内核操作系统是面向对象的，这样，在做为纳内核的总内核上的运行整体是对象。对象由消息传递来实现交互。

此操作系统同时地提供多种程序运行环境，在下面说明中，由此操作系统提供的程序运行环境将被称为是元空间(meta-space)。如图 2 所示，此操作系统提供 3 种元空间，即，元空间 mCOOP、元空间 mDrive 和元空间 mCore。每个元空间由多个对象构成。在下面的说明中，每一构成元空间的元空间被称为“元对象”。

元空间 mCOOP 被用来实现面向对象的如实现控制操作面板 8 的图形用户界面的程序的应用程序的运行。表示 mCOOP 的首字符“m”是元空间的指示符，COOP 是“当前面向对象编程”的缩写。

5 元空间 mDrive 用于负责控制硬件设备的设备驱动程序的运行。表示 mDrive 的首字符“m”是元空间的指示符“Drive”意为此元空间用于设备驱动程序的运行。

元空间 mCore 用于构成元空间 mCOOP 和 mDrive 的元对象的运行。这样元空间 mCore 对应于微内核。表示 mCore 的首字符“m”是元空间的指示符，“Core”指示此元空间构成了操作系统的一个“核”。

10 这样，操作系统的层次结构便如图 2 所示，元空间 mCOOP 和 mDrive 运行其上的元空间 mCore 运行在如 CPU 的硬件设备上，面向对象的应用程序和设备驱动程序分别运行在元空间 mCOOP 和元空间 mDrive 上。这些上下文之间的上下文转接由总核控制。

除元空间 mCOOP 和 mDrive 外操作系统还提供其它的能运行在元空间 mCore 上的元空间，如过程应用程序，例如使 TV 功能部件显示活动图像的应用程序。然而，由于本发明可被等同地应用于各种运行在元空间 mCore 上的元空间，元空间 mCOOP 和 mDrive 以外的元空间的解释在下面的说明中省略。这样下面将以举例方式详细说明元空间 mCOOP 和 mDrive。

如图 2 所示，有构成元空间 mCOOP 的三种类型的元对象：即
20 mCOOPScheduler 对象、mCOOPMailer 对象和 mCOOPFaultHandler 对象。mCOOPScheduler 对象即是所说的“调度程序”，它是一个元对象，它负责运行在元空间 mCOOP 上的应用程序的调度管理。mCOOPMaier 对象是一个元对象，它执行运行在元空间 mCOOP 上的应用程序之间的信息传递。mCOOPFauleHandler 对象是一个用于例外处理的元对象。除以上专门提到的
25 元对象外，元空间 mCOOP 还具有各种元对象成份。

参照图 2，有三种构成元空间 mDrive 的元对象，即对象 mDriveScheduler 和对象 mDriveMailer 和对象 mDriveFaultHandler。对象 mDriveScheduler 是所说的“调度程序”，他是一个元对象负责运行在元空间 mDrive 上的设备驱动程序的调度管理。对象 mDriveMailer 是一个元对象，它运行在元 mDrive
30 上上。对象 mDriveFaultltandler 是一个意外处理元对象。元空间 mDrive 除上述元对象外有各种元对象做为组成成份。

参照图 2，有两种构成元空间 mCore 的元对象：即 mCoreScheduler 对象和 mCoreMailer 对象。mCoreScheduler 对象即是所谓的调度程序，它是一个负责运行在元空间 mCore 上的对象的调度管理元对象。mCoreMailer 对象是一个元对象，它执行运行在元空间 mCore 之上的对象之间的消息传递。元空间 mCore 除以上专门提到的两类元对象外还具有各种元对象成份。

虽然所说明的实施例中，每一元空间采用其自己的元对象做为调度程序，但这并不是必须的，还可以将调度元对象设置成一个被所有元空间共同使用的元对象。在这样的设置中，例如，一个调度元对象在被元空间 mCOOP 使用时做为一个属于该元空间 mCOOP 的元对象。相似地，该调度元对象在被元空间 mDrive 使用时做为一个属于该元空间的对象以及该调度元对象在被元空间 mCore 使用时做为一个属于该元空间的对象。

在元空间上运行的对象在下面的说明中将被称为“基对象”(base-object)，用于使基对象获得元对象提供服务的元对象的调用被称之为“元调用”。这样，元调用等同于大内核类型操作系统的系统调用。基对象和元对象之间的上下文转接由对应于纳内核的总核来完成。

在下面的说明中，“调度程序”这一术语集中代表调度元对象，包括 mCOOPScheduler 对象、mDriveScheduler 对象、和 mCoreScheduler 对象。

在进行实施例说明之前，需要说明一下纳内核型操作系统中所采用的一般调度管理方法。参照附图 3 和 4 将进行说明，附图 3 和 4 说明了对象 A 和对象 B 之间发送信息的操作，对象 A 是一个运行在元空间 mCOOP 之上的应用程序，对象 B 也是一个运行在同一个 mCOOP 之上的应用程序。图 3 在说明操作系统层次结构的同时还说明了从对象 A 到对象 B 发送消息的过程。图 3 中省略了元空间 mDrive 的示出。图 4 是一个时间图表，说明了执行从对象 A 至对象 B 的消息邮递的对象 A 和对象 B 之间的对象交互。

其过程如下：

(1)对 mCOOPMaier 元调用

首先，由对象 A 调用在元空间 mCOOP 上的 mCOOPMaier 对象以发出元调用。这样，为使对象 A 到对象 B 传递消息，负责对象间消息传递的 mCOOPMaier 对象被调用。

更详细地，为调用负责处理从对象 A 发出的元调用的 mCOOPMaier 对象，由箭头 A1 指示了一个对元空间 mCore 上的对象 mCoreScheduler 的调

用，在元空间 mCore 上运行着指定的 mCOOPMaier 对象。这样，在元空间 mCore 上的对象便处于 mCoreScheduler 对象的调度控制之下。在这种状态下，在元空间 mCOOP 上的指定的对象 mCOOPMaier 被调用，如箭头 2 指示。

(2)向 mCOOPScheduler 发送信息。

- 5 之后，为了启动对象 B，对象 mCOOPMaier 向元空间 mCOOP 上的对象 mCOOPScheduler 发送消息。更详细地，对象 B 的启动要求运行在元空间 mCOOP 之上的对象的调度管理。上述提到的用于启动对象 B 的消息因此被发送给执行元空间 mCOOP 上对象的调度管理的对象 mCOOPScheduler。

- 10 为详细解释，从对象 mCOOPMaier 到对象 mCOOPScheduler 的消息发送开始于由箭头 A3 指示的对做为元对象的负责运行在元空间 mCore 上的对象间消息传递的 mCoreMailer 对象的调用。相应地，如箭头 A4 所指示，由于消息目的地对象 mCOOPScheduler 运行在元空间 mCore 上，做为元对象的负责运行在元空间 mCore 之上的对象的调度管理的 mCoreScheduler 对象被调用。这样，在元空间 mCore 上的对象便处于由 mCoreScheduler 对象执行的调度控制之下。在这个调度控制之下，mCOOPScheduler 调度程度对象被启动，如箭头 A5 指示并且，上述用于启动对象 B 的消息被发往 mCOOPScheduler 对象。

(3)回答 mCOOPMaier

- 20 接收到消息后，mCOOPScheduler 对象立刻按所接收消息改变对象 B 的状态，并且对此消息发一回答消息至 mCOOPMaier 对象。

- 25 为详细解释，此回答操作执行如下。为从 mCOOPScheduler 对象发送回答至 mCOOPMaier 对象，由箭头 A6 指示了一个调用，调用负责运行在元空间 mCore 之上的对象间的消息传递的 mCoreMailer 对象。之后，为使回答目的地对象 mCOOPMaier 工作在元空间 mCore 之上，由箭头 A7 指示发出一个调用，调用做为元对象的负责运行在元空间 mCore 之上的对象的调度管理的 mCoreScheduler 对象。在由 mCoreScheduler 对象执行的运行在元空间 mCore 之上的对象的调度管理之下，由箭头 A8 所指示，mCOOPMaier 对象被启动，并且由 mCOOPScheduler 对象发出的回答被发往启动的 mCOOPMaier 对象。

- 30 (4)恢复基对象

当由 mCOOPMaier 对象执行的处理完毕时，对对象 A 和 B 的处理开始。

首先如箭头 A9 所示该处理进行 mCOOPScheduler 对象的调用。在 mCoreScheduler 对象执行例如重调度的处理之后，如箭头 A10 所示，做为基对象的对象 B 被调用，就此启动了处理。但应当注意，若优先级设定为对象 A 的处理优先于对象 B 的处理，处理先从对象 A 开始，即箭头 A10 调用对象 A。

在一般调度管理机构下的从对象 A 到对象 B 的消息发送过程已被说明。

可以看出，一般的调度管理机构要求执行相当大量的对象转接，即大量的上下文转接。在所说明的系统中，这些上下文转接由做为纳内核的总内核控制和执行。这些大量的上下文转接不良地影响了系统的整体性能。这样改善使用需要大量上下文转接的一般的调度管理机构的操作系统的整体性能便很困难。

如下将说明本发明所采用的调度管理机构。

通常，对象的交互具有相互依赖性。这样，便存在这样一些情况，即不论调度是否执行，对象之间执行的转换的序列不变。在这样的情况下，按照本发明，调度操作由有意地阻止调度程序的调用而省去。靠有效地利用对象执行转换序列调度程序的调用频度被减小。这样便减小了相应的上下文转接的数量，改善了系统的整体性能。

下面将参考附图 5 和 6 以实施例说明按照本发明的调度管理过程。为了便于比较本发明的调度管理过程和一般的调度管理过程，如下的说明是根据下面这样一种情况，如同在说明一般的过程中，消息从对象 A 发往对象 B，对象 A 是一个运行在元空间 mCOOP 上的应用程序，对象 B 是运行在元空间之上的另一个应用程序。图 5 中，相似于图 3，说明了从对象 A 到对象 B 发送消息的过程以及操作系统的层次结构，图 6 中，与图 4 相似，说明了在对象 A 至对象 B 发送消息的过程中对象交互的时间图表。

该过程如下

(1)对 mCOOPMaier 元调用

首先由对象 A 发出元调用以调用运行在元空间 mCOOP 之上的 mCOOPMaier 对象。这样，为使消息从对象 A 发至对象 B，在元空间 mCOOP 之上的负责对象间消息传递的 mCOOPMaier 对象被调用。

此时 mCOOPMaier 对象如箭头 B1 所示由从对象 A 的元调用直接被调

用。对象 A 必须等待由 mCOOPMaier 对象所做处理完成。这样，在从对象 A 到 mCOOPMaier 对象的运行转换中存在一个序列，因而不用象在一般调度管理过程中那样为了调度管理调用 mCoreScheduler 对象，直接地调用了 mCOOPMaier 对象。

5 (2)发送消息到 mCOOP 调程序。

之后，为了启动对象 B，对象 mCOOPMaier 向运行在元空间 mCOOP 之上的 mCOOPScheduler 对象发送消息。详细地说，对象 B 的启动要求运行在元空间 mCOOP 之上的对象的调度管理。因此上述的用于启动对象 B 必需的消息被发送至做为元对象的负责在元空间 mCOOP 上的对象调度管理的 mCOOPScheduler 对象。

10 为详细解释，从 mCOOPMaier 对象至 mCOOPScheduler 对象的消息发送开始于由箭头 B2 所指示的对做为元对象的负责运行在元空间 mCore 之上的对象之间消息传递的 mCoreMailer 对象的调用。之后，如箭头 B3 所指示，mCOOPScheduler 对象被直接调用，并且上述用于启动对象 B 必需的消息被
15 发送至 mCOOPScheduler 对象。

应当理解，mCOOPMailer 对象必须等待 mCOOPScheduler 对象所做处理的完成，这样在从 mCOOPMailer 到 mCOOPScheduler 的运行转换中存在序列。因此，调用 B3 直接调用 mCOOPScheduler 对象，而不用象一般的调度管理过程中那样为了调度管理调用 mCoreScheduler 对象。这样，与本发明
20 一致，在同一元空间上的对象间的消息传递当存在运行转换序列时执行跳过调用调度程序的步骤，这种消息传递的方式在下面的说明中被称为“FastSend”。

(3)回答 mCOOPMaier

25 接到消息后，mCOOPScheduler 对象按所收消息改变对象 B 的状态，并以 FastSend 的方式发送回答此消息的消息至 mCOOPMaier 对象。

为详细说明，这种回答操作如下执行。为使 mCOOPScheduler 对象发送回答至 mCOOPMaier 对象，由箭头 B4 指示发出一个调用，调用负责运行在元空间 mCore 上的对象间消息传递的 mCoreMailer 对象。之后，由箭头 B5 指示，mCOOPMaier 对象被直接调用，并且从 mCOOPScheduler 对象的回答
30 发往 mCOOPMaier 对象。在从 mCOOPScheduler 对象到 mCOOPMaier 对象的运行转换中存在序列。因此，调用 B5 被发出以使回答直接发至

mCOOPMaier 对象，而不用象一般的调度管理过程那样为调度管理调用 mCoreScheduler 对象。

(4)恢复基对象。

当由 mCOOPMaier 对象执行的处理完成后，开始对象 A 和 B 的处理。

- 5 由箭头 B6 所指示此处理开始于对 mCoreScheduler 对象的调用。当 mCoreScheduler 对象执行了如重调度之类的处理后，做为一个基对象的对象 B 被调用如箭头 B7 指示，这样处理被启动。但应注意，如果所设优先级为在对象 A 中的处理优先于在对象 B 中处理时，处理首先从对象 A 开始，即箭头 B7 调用对象 A。

- 10 从上面说明中应能理解，本发明通过使用实现直接元调用的 FastSend 功能和机构省略了调用调度程序的步骤。这样，本发明减少了所要执行的上下文转接数量，比使用一般的调度管理机构的系统具有系统整体性能上的改进。

- 15 虽然在前面的说明中，消息传递处理被举例详细地说明成具有跳过调用调度程序的优点的处理，此优点也可以实现于其它的处理之中。例如，调用调度程序的省略也可以在一中断处理中实现。在这种情况下，中断等待时间可以因消除了用于调用调度程序的时间而被缩短。这样，调用调度程序的省略可适用于需要快速执行的处理，例如执行响应于中断的处理。

- 20 但仅省略调用调度程序的步骤会造成对象实际状态和调度程序所掌握状态的不一致。这样带来了系统操作出错的风险。因此，为与本发明一致，采用了一种机构以确保对象实际状态和调度程序所掌握状态的一致。这个机构在本说明中将被称为是“延迟队列机构”。

在进行延迟队列机构的说明之前，图 7 给出了一个参考，示出了对象的线程状态。

- 25 每一个线程被分配给一个对象，由调度程序执行的调度管理以线程为基础来被执行。在此假定线程可以有取三种状态之一，“睡眠”，“等待”和“运行”。睡眠状态意为由于相关的对象无活动而此线程不处于激活状态。等待状态意为由于相关对象正等待某一事件而此线程不处于激活状态。运行状态意为对象正被执行。这样，运行状态指明线程是激活的或线程是在调度程序控制下的队列里并且正等待 CPU 处理时间的分配。调度程序在接到另一对象的请求时改变对象的线程状态并且更新在线程中的队列。
- 30

延迟的队列机构将在下面说明。

延迟队列机构的使用要求为每一线程分配一结构。当从一线程向另一线程(分别称为“源线程”和“目标线程”)的上下文转接产生运行转换时,上下文转接的历史被存于结构中。上下文转接的历史为每一线程存放了运行转换的性质以及做为运行变换结果激活每一线程的线程。这样,上下文转接的历史包括标明了恰在运行变换之前即源线程运行的线程的信息以及标明了触发目标线程例如元调用或 FastSend 的原因。

当调度程序被调用时,延迟队列机构找回记录在结构中的历史以保持对象实际状态和调度程序的所掌握状态匹配。更详细地,当调度程序被调用时,延迟队列机构被启动并参照记录在结构中的信息跟踪回上次调用调度程序所执行的上下文转接历史。延迟队列机构按记录历史与实际对象状态一致地改变由调度程序识别的对象状态,同时更新线程中的队列,以此达到对象实际状态和调度程序所掌握对象状态的匹配。

这样,放置于延迟队列机构的操作下的线程便是那些省略了调用调度程序步骤而已被触发的线程。更详细地,这些线程包括一系列线程,从恰在调度程序的当前调用之前运行的线程(此线程被称为“当前线程”)到调度程序为响应上一次调用而触发的线程(此线程为“激活线程”)。这样,由调度程序触发的响应于上一次调用的“激活线程”即是由调度程序识别的当前运行的线程。

延迟队列机构执行和线程状态有关的由调度程序识别的信息的更新,这个更新按下面的表 1 所示规则进行。

表 1

	源线程状态		目标线程状态	
	上下文转接之前	上下文转接之后	上下文转接之前	上下文转接之后
元调用	运行	等待	睡眠	运行
FastSend	运行	运行	睡眠	运行

这样,其做为由元调用产生的上下文转接的结果启动了目标线程,源线程在上下文转接之后被设成等待状态并且目标线程在上下文转接上后被设成运行状态。相反,若目标线程做为由 FastSend 产生的上下文转接结果被启动,源线程在上下文转接之后被设成运行状态并且目标线程在上下文转接之后也被设成运行状态。源线程和目标线程的状态在本质上是睡眠状态不论上

下文转接是由元调用还是由 FastSend 引起。

下面将参照图 8 的流程图详细说明延迟队列机构的算法。在这个流程图中的以及在图 9 的流程图中，以面向对象的编程语言 C++ 的表达式来描述每一步骤所执行处理。

- 5 延迟队列机构以在调度程序中描述的“Scheduler::Delayed Queuing()”的方法实现。换言之，当调度程序被调用时，最好用执行“Scheduler::Delayed Queuing()”的方法来激活延迟队列机构。

在本方法“Scheduler::Delayed Queuing()”的第一步中，恰在调度程序被调用之前已被激活的线程“CurrentThread”被分配给在延迟队列机构中使用的变量“pThread”。这样，当第一步结束时，变量“pThread”表示当调度程序被调用时已被激活的线程。

10

在第二步“pThread”被用来与“active Thread”进行比较，“active Thread”是由调度程序为响应上一次对调度程序的调用而激活的线程，即被调度程序识别为当前激活的线程。若“pThread”与“active Thread”不同，即当“pThread”没有指示出上一次由调度程序激活的线程时，处理向前至步骤 3。然而，若“pThread”与“active Thread”相同时，即当“pThread”指示出上一次由调度程序激活的线程时，处理跳至步骤 10。

15

步骤 3 判定对由“pThread”指示的线程的调用是由元调用发出的还是由“FastSend”发出的。当步骤 3 判定为调用由元调用发出的则过程前进至步骤 4，否则，过程前进至步骤 7。这样，当在步骤 3 中判定调用由“FastSend”发出时，执行步骤 7。

20

由“pThread”指示的线程被元调用调用的事实意为此线程为源线程，即从基对象到元对象产生上下文转接的源线程。因此，在步骤 4 中，在此线程上设置了一个队列更新标志。在图 8 中队列更新标志被表示为“HOOK flag”。由异步调度机构使用的此队列更新标志，后面将说明。在步骤 4 中设置了标志之后，过程前进至步骤 5。

25

这样，由于由“pThread”指示的线程是由元调用所调用而执行步骤 5。这样在步骤 5 中，恰在由“pThread”指示的线程之前已运行的线程的状态被设置成等待，这与表 1 所示一致。步骤 5 中完成处理之后，过程前进至步骤 6。

30

在步骤 6 中，为了跟踪回上下文历史，“pThread”的内容被改变以使

“ pThread ” 指示恰在由 “ pThread ” 所指示的线程之前运行的线程。然后处理返回至步骤 3 以重复已述步骤。

在步骤 7 中，在由 “ pThread ” 所指示的线程中设置队列标志，如在步骤 4 中的情况。当步骤 7 的处理完成后，过程前进至步骤 8。

- 5 这样，由于由 “ pThread ” 所指示线程被 “ FastSend ” 调用而步骤 8 被执行。因而在步骤 8 中，恰在由 “ pThread ” 所指示线程前运行的线程的状态被设置成 “ 运行”，这与表 1 所示规则一致。步骤 8 的处理完成之后，过程前进至步骤 9。

- 10 步骤 9 中，恰在由 “ pThread ” 所指示的线程前运行的线程，即其状态在步骤 8 中被设定为是 “ 运行” 的线程被加入到在调度程序管理之下的队列中。步骤 9 的处理结束后，过程前进至步骤 6。

如前已述，步骤 6 中，为了跟踪回上下文历史，“ pThread ” 的内容被改变，以使 “ pThread ” 指示恰在由 “ pThread ” 已经指示的线程之前运行的线程。然后处理返回至步骤 2 以重复已述步骤。

- 15 在步骤 7 中，如同步骤 4 的情况，在由 “ pThread ” 所指示的线程中设置队列标志。在步骤 7 的处理完成后，过程前进至步骤 8。

- 20 这样，由于由 “ pThread ” 所指示线程被 “ FastSend ” 调用而步骤 8 被执行。因而在步骤 8 中，恰在由 “ pThread ” 所指示的线程前运行的线程的状态被设置成 “ 运行”，这与表 1 所示规则一致。步骤 8 的处理完成后，过程前进至步骤 9。

在步骤 9 中，恰在由 “ pThread ” 所指示的线程前运行的线程，即其状态在步骤 8 中设定为是 “ 运行” 的线程被加入到在调度程序管理之下的队列中，步骤 9 的处理结束后，过程前进至步骤 6。

- 25 如前所述，在步骤 6 中，为了跟踪回上下文历史，“ pThread ” 的内容被改变以使 “ pThread ” 指示恰在由 “ pThread ” 所指示线程前运行的线程。然后处理返回至步骤 2 以重复已述步骤。

- 30 如前所述，若在步骤 2 中进行的比较指示 “ pThread ” 指示的是上次由调度程序激活的同一线程，则过程直接跳至步骤 10。在步骤 10 中，“ CurrentThread ” 被分配给 “ active Thread”。“ active Thread ” 是被调度程序识别为正激活的线程，“ CurrentThread ” 是恰在调度程序被调用之前活动的线程。这样，当在步骤 10 中的处理完成之后，被调度程序识别为是正

激活的线程便是恰好在调度程序被调用前激活的线程。

可以看出，在开始时，仅当“CurrentThread”和“active Thread”一样时，即当调用调度程序的线程与上一次由调度程序调用的线程一样时，才发生从步骤 2 跳至步骤 10 越过步骤 3 至 9。

- 5 当在上下文转接历史中没有省略调用调度程序的上下文转接时，正调用调度程序的线程便与由调度程序上一次调用的线程是同一线程。相反，当上下文转接历史包括一个省略调度程序调用的上下文转接时，正调用调度程序的线程便与上一次由调度程序调用的线程不是一个线程。

- 10 这样，若没有省略调用调度程序的上下文转接，则在过程开始时，“CurrentThread”和“active Thread”是相同的。在这种情况下，该过程在步骤 1 之后跳过步骤 3 至步骤 9 而从步骤 2 至步骤 10。在步骤 10 中没有实际的执行被执行，因为“active Thread”与“CurrentThread”已经是相同的。相反，当存在不调用调度程序而被执行的上下文转接时，“CurrentThread”和“active Thread”便不相同。这样，执行下面的步骤 3 以跟踪回上下文转接历史中以定位不调用调度程序而执行的上下文转接。
- 15

因此完成在上次上下文转接时省略的线程和队列的改变和更新，这样，达到到对象实际状态和调度程序所掌握对象状态的一致。

- 20 本技术领域人员可以看出如上说明的延迟队列机构还可被用于因需要加急处理如中断而省略调度程序调用的情况中，虽然在如上说明的实施例中，延迟队列机构被用于调度程序的调用在消息传递的过程中被省略的情况。

- 25 当与调度程序一样的元空间的对象间的上下文转接序列不受由延迟队列机构执行的对象状态的改变的影响时，在由“Scheduler::Delayed Queue”所执行的执行之前如图 9 流程所示“CurrentThread”可以由方法“Scheduler::Check Invoke()”所设置。

方法“Scheduler:: Check Invoke()”开始于步骤 21，将“Scheduler Thread”指定给“PThreda”。“Scheduler Thread”的意思是调度程序的线程。这样，步骤 21 完成后，“pThread”指向调度程序的线程。

- 30 步骤 22 判定是否由“pThread”所指示线程的调用由“FastSend”产生。若是，即如果“FastSend”调用了此线程，过程前进至步骤 23。

由“pThread”所指示线程由“FastSend”调用的事实是指此线程和恰

在此线程前运行的线程是与调度程序同一元空间的元对象线程。这样，步骤 23 仅当由 “ pThread ” 所指示线程与恰在其前运行的线程是与调度程序相同的元空间的元对象的线程时被执行。因此这种情况下，使延迟队列机构工作的 “ CurrentThread ” 跟踪回恰在 “ CurrentThread ” 前运行的线程。因此在 5 步骤 23 中，“ pThread ” 被更新以指示恰在由 “ pThread ” 所指示线程前运行的线程。然后过程返回步骤 22 以重复步骤 22 和 23 。

相反，当在步骤 22 中判定由 “ pThread ” 所指示线程的调用不是由 “ FastSend ” 产生时，过程跳至步骤 24 。可看出做此跳步时恰在由 “ pThread ” 所指示线程前运行的线程不是与调度程序同一元空间上的元对 10 象的线程。

在步骤 24 中，“ pThread ” 被分配给 “ CurrentThread ”，这样，由 “ pThread ” 所指示线程被设为由 “ CurrentThread ” 所指线程。因此，当步骤 24 的操作被完成时，恰在由 “ CurrentThread ” 所指示线程前运行的线程不是与调度程序一样的元空间的一个对象的线程。

15 当在步骤 24 中执行的 “ CurrentThread ” 的设定完成时，方法 “ Scheduler::CheckInvoke() ” 的处理被完成且过程前进至上述方法 “ Scheduler::delayed Qneuing() ” 的处理。

这样，在 “ CurrentThread ” 的设定之后执行方法 “ Scheduler::Delayed Qneuing() ” 的处理。这种处理序列靠跟踪回转接上下文历史而减少了调度程 20 序状态更新的处理，这样，改善了处理的效率。

当由延迟队列机构改变队列时设置队列更新标志，按标志异步地调用调度程序执行再调度。更详细地说，当延迟队列机构被激活时，为触发需要再调度的上下文转接的源转接设置队列更新标志。当此上下文转接被激活时，调度程序被调用而执行再调度此机构被称为 “ 异步调度程序调用机构 ”。

25 由异步调度程序调用机构所执行的处理的过程如下：

(1) 当队列的更新由延迟队列机构实现时，队列更新标志被设置于触发从元对象到基对象的上下文转接的源线程。此标志的设置如上参照图 8 说明的流程图 中步骤 4 和步骤 7 中执行。

(2) 当发生从元对象到基对象的上下文转接时，在激活目标线程即基对象 30 线程前，检查是否在源线程上即元对象上设置了队列更新标志。当在源线程上这样的标志被证实了后，调度程序被调用而取代调用目标线程。调度程序

在从元对象到基对象的上下文转接之前执行重调度。

队列更新标志的设置如下进行，取决于调度程序改变队列的条件。

(1)当延迟队列机构被对象“mCore”调度程序执行时，队列更新标志被设置在引起从构成元空间mCore的对象到运行在元空间mCore上的对象的上下文转接的线程上。当调度程序做为一个元空间mCore的元对象以执行延迟队列机构时，调度程序被所有元空间公用的情况下，也能进行这个设置。

(2)当延迟队列机构被由元空间mCore的元对象调用的mCOOPScheduler对象已经执行时，队列更新标志被设置在引起从构成元空间mCOOP的对象到运行在元空间mCOOP上的对象的上下文转接的线程上。当调度程序做为一个元空间mCOOP的元对象执行延迟队列机构时，调度程序被所有元空间公用的情况下，也能进行这个设置。通过使用图9所示方法“Scheduler::Check Invoke()”改变启动延迟队列机构的线程而执行这个标志的设置。

下面将详细说明采用如上所述队列更新标志的异步调度程序调用机构。此说明将以举例方式说明从运行在元空间mCOOP上的对象A向运行在元空间mDrive上的对象C发送消息的情况。假定调度程序被所有元空间公用，即通过公用“Scheduler”对象在所有元空间上执行调度管理。

图10和11说明了从对象A到对象C传递消息的过程。具体地说，如图3和图5的情况，图10说明了此过程及操作系统的层次结构。图11是一个时间图表。如4和6情况，说明了从对象A到对象C消息的传递期间执行的对象的交互。

此过程如下：

(1)对mCOOPMailer的元调用。

对象A用元调用调用元空间mCOOP的mCOOPMailer对象。这样，负责由运行在元空间mCOOP之上的对象所执行消息传递的mCOOPMailer对象被调用，以使消息从对象A传至对象C。

在这种情况下，如箭头C1所示，mCOOPMailer对象由来自对象A的元调用调用。这样，因为在从对象A到mCOOPMaier对象的运行转换中存在序列，mCOOPMaier被直接调用而不必为了调度管理而调用调度程序对象。

(2)向mDriveMailer发送消息。

作为消息目的地对象的对象C，位于元空间mDrive上。这样，要求向

对象 C 传递消息的消息被发往做为元对象负责由运行在元空间 mDrive 上的对象的消息传递的 mDriveMailer 对象上。

具体地说，为使消息从 mCOOPMaier 对象到 mDriveMailer 对象之间传递，如箭头 C2 所指示，执行对对象 mCoreMailer 的调用，该 mCoreMailer 是一个元对象负责运行在元空间 mCore 上的对象间的消息传递。相应地，如箭头 C3 所指示，直接调用对象 mDriveMailer。应注意，对象 mDrive Mailer 被直接调用，而不必为调度管理而调用调度程序，因为在从对象 mCOOPMailer 到对象 mDriveMailer 的运行转换中存在序列。换言之，FastSend 实现了从对象 mCOOP Mailer 到对象 mDrive Mailer 的消息传递。

10 (3)向调度程序发送信息。

之后，对象 mDrive Mailer 向对象 Scheduler 发送消息，以激活对象 C。对象 C 的激活要求调度管理。为达到此目的，用于激活对象 C 必需的。消息被发至做为元对象负责调度管理的对象 Scheduler。

具体地说，为使消息从对象 mDrive Mailer 传递至对象 Scheduler，如箭头 C4 所示，调用被执行以调用做为元对象负责运行在元空间 mCore 上的对象间消息传递的对象 mCore Mailer。之后，由箭头 C5 指示，直接调用对象 Scheduler，并且用于激活对象 C 的必需消息被发往对象 Scheduler。应注意，对象 Scheduler 被直接调用而不必调度步骤，因为在从对象 mCOOP Mailer 到对象 Scheduler 的运行转换中存在序列。这样，从对象 mDriveMailer 到对象 Scheduler 的消息传递由 Fast Send 完成。

(4)更新调度队列。

之后，对象 Scheduler 按所收到消息改变对象 C 的状态，并且使延迟队列机构改变队列。

因要改变对象 C 的状态对象 Scheduler 被调用，并且起到元空间 mDrive 的元对象的作用。这样，对象 Scheduler 在激活为恢复运行在元空间 mDrive 上的对象的执行的上下文转接或为恢复运行在元空间 mCOOP 上的对象的执行的上下文转接的线程上设置队列更新标志，其中该 mCOOP 元空间与元空间 mDrive 具有同一层次结构。具体地说，对象 Scheduler 在对象 mCOOP Mailer 的线程上设立一个队列更新标志。

30 这一操作可以如下具体实现，用方法“Scheduler::CheckInvoke()”的算法来跟踪下列序列的对象执行转换，之后执行从对象 mCore Mailer(2)开始的

延迟队列机构：对象 Scheduler、对象 mCoreMailer(1)、对象 mDriveMailer、对象 mCoreMailer(2)、对象 mCOOPmailer、对象 A。

对象 mCoreMailer(1)和对象 mCore mailer(2)是同一对象。后缀(1)和(2)是用来说明此对象被调用的时间上的区别。

5 (5)之后,为响应由 Fast Send 从对象 mDriveMailer 发送给对象 Scheduler 的消息, 对象 Scheduler 向对象 mDriveMailer 发送回答。具体地说, 为使回答从对象 Scheduler 发送至对象 mDrive, 如箭头 C6 所示执行调用以调用做为元对象负责运行在元空间 mCore 之上的对象间消息传递的对象 mCore Mailer。之后, 如箭头 C7 所示直接调用对象 mDriveMailer 并且从对象
10 Scheduler 的回答被发往对象 mDrive。回答被直接发往对象 mDrivemailer 而不用进行调度, 因为在从对象 Scheduler 到对象 mDriveMailer 的运行转换中存在序列。对象 mDriveMailer 的执行这样被恢复。

(6)回答 mCOOPMailer。

之后,为响应由 Fast Send 从对象 mCOOPMailer 发送的消息,对象 mDrive
15 Mailer 向对象 mCOOP Mailer 发送回答。

具体地说, 为使回答从对象 mDrive 发至对象 mCOOPMailer, 如箭头 C8 所示执行调用以调用做为元对象负责运行在元空间 mCore 上的对象间消息传递的对象 mCoreMailer, 对象 mCOOPMailer 如箭头 9 所示直接被调用, 并且从对象 mDrive Mailer 的回答被发送至对象 mCOOP Mailer。此回答直接
20 被发至对象 mCOOPMailer 而不用调度, 因为在从对象 mDrive Mailer 到对象 mCOOPMailer 的运行转换中存在序列。对象 mCOOPMailer 的执行被恢复。

(7)恢复基对象

由对象 mCOOP Mailer 所执行的处理完毕后, 为恢复对象 A, 上下文转接处理被执行以转接上下文到对象 A。但是, 应理解, 队列更新标志被设置在做为上下文转接的源线程的对象 mCOOP Mailer 的线程上。这样, 实际
25 上执行了异步调用机构的激活而不是到对象 A 的上下文转接。即不执行对象 A 的恢复而调用对象 Scheduler, 如箭头 C10 所指示。

(8)重调度

这样重调度被对象 Scheduler 执行。假定在此队列的调度被更新以使对
30 象 C 在对象 A 的执行前被执行。这样重调度便被对象 Scheduler 执行以使对象 C 被接下来执行。因此对象 C 如箭头 C11 所示被调用并且执行。

可以看出调度程序即对象 Scheduler 的调用次数可以被明显地减少。这是因为在从对象 A 到对象 B 的发送消息的消息传递操作中使用了异步调度程序调用机构。具体地说,在所述实施例中,对运行在元空间 mCOOP 或 mDrive 上的对象调度被延迟,并且当发生上下文转接以恢复对象 A 的线程时,调度程序被以异步方式调用。可以理解,与执行回答 Fast Send 消息的上下文转接每次调用调度程序相比,调度程序的调用次数减少了 2 次。

在如上说明的实施例中,在调度程序的控制下的队列更新只对运行在元空间 mCOOP 和元空间 mDrive 上的对象实施。这样,更新一点不影响运行在元空间 mCore 上的对象。以延迟对运动在元空间 mCOOP 和 mDrive 上的对象的调度为代价的调度程序调用次数的减少不产生任何对整系统的不良影响。

从上述说明中可以理解,本发明最小化了调度程序的调用次数,同时保证了整个系统的正常操作,这样实现了使用纳内核时的整个系统性能的改善。

虽然通过优选形式描述了所描述的实施例,但应理解在不偏离所附权利要求所限定的范围内可以对本发明做各种修改和改变。

图 1

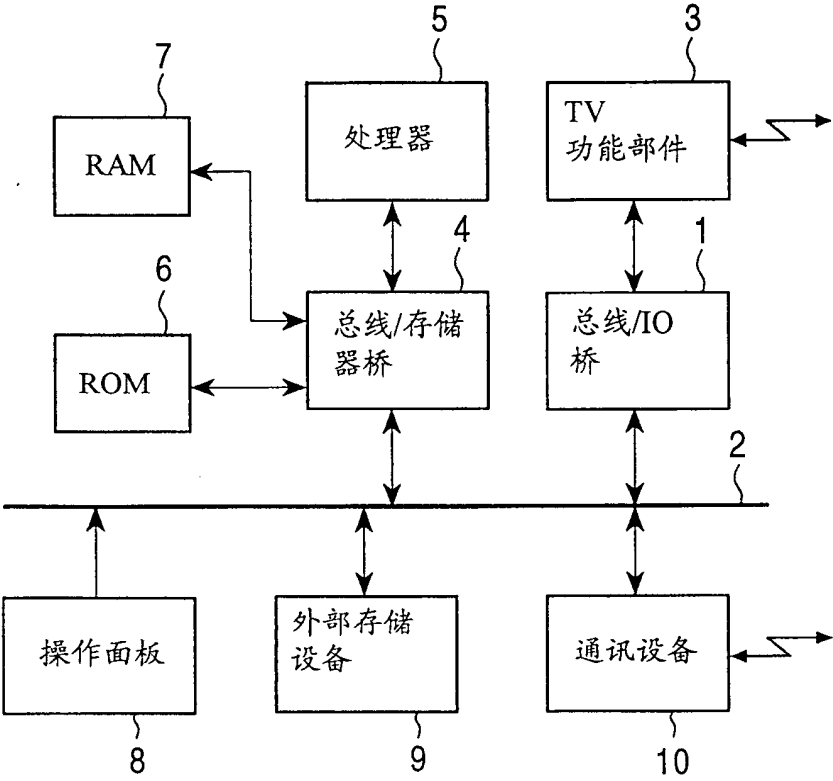


图 2

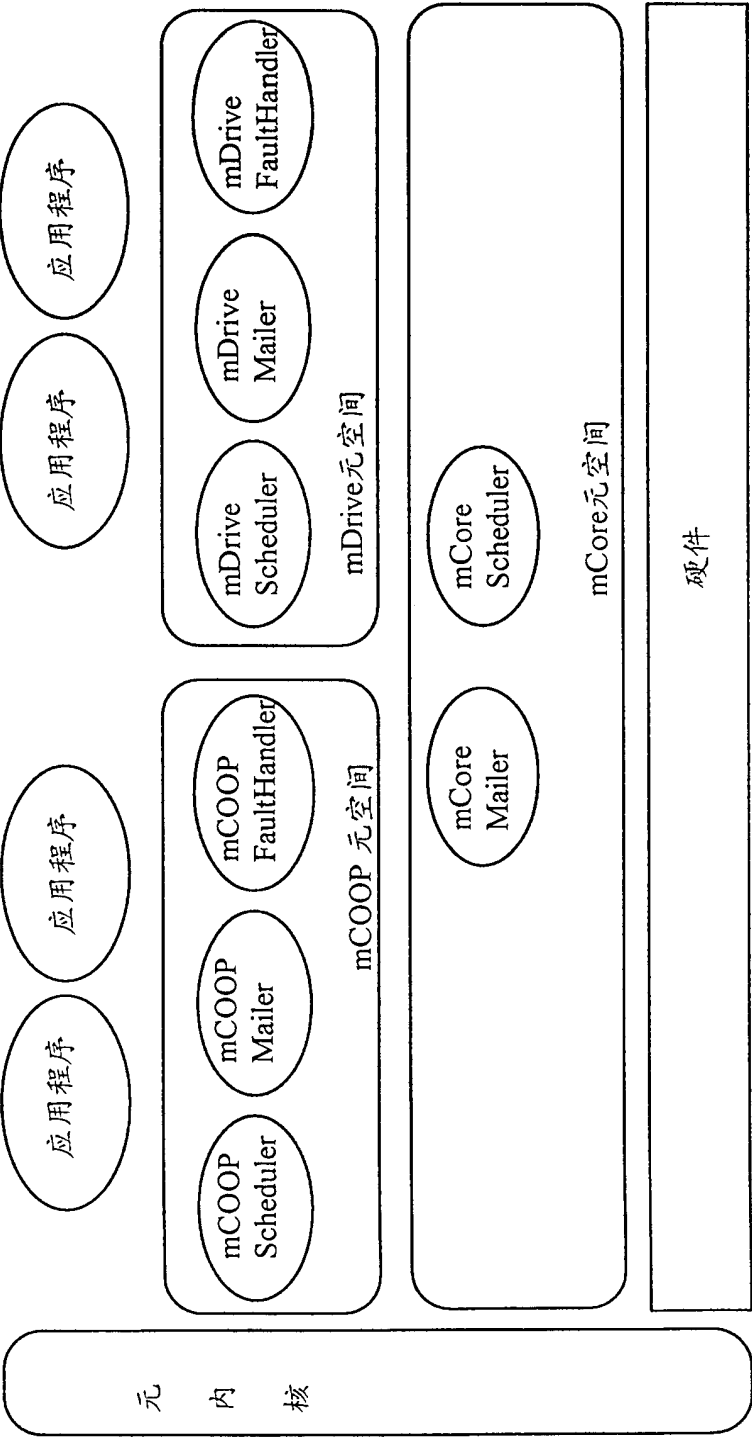


图 3

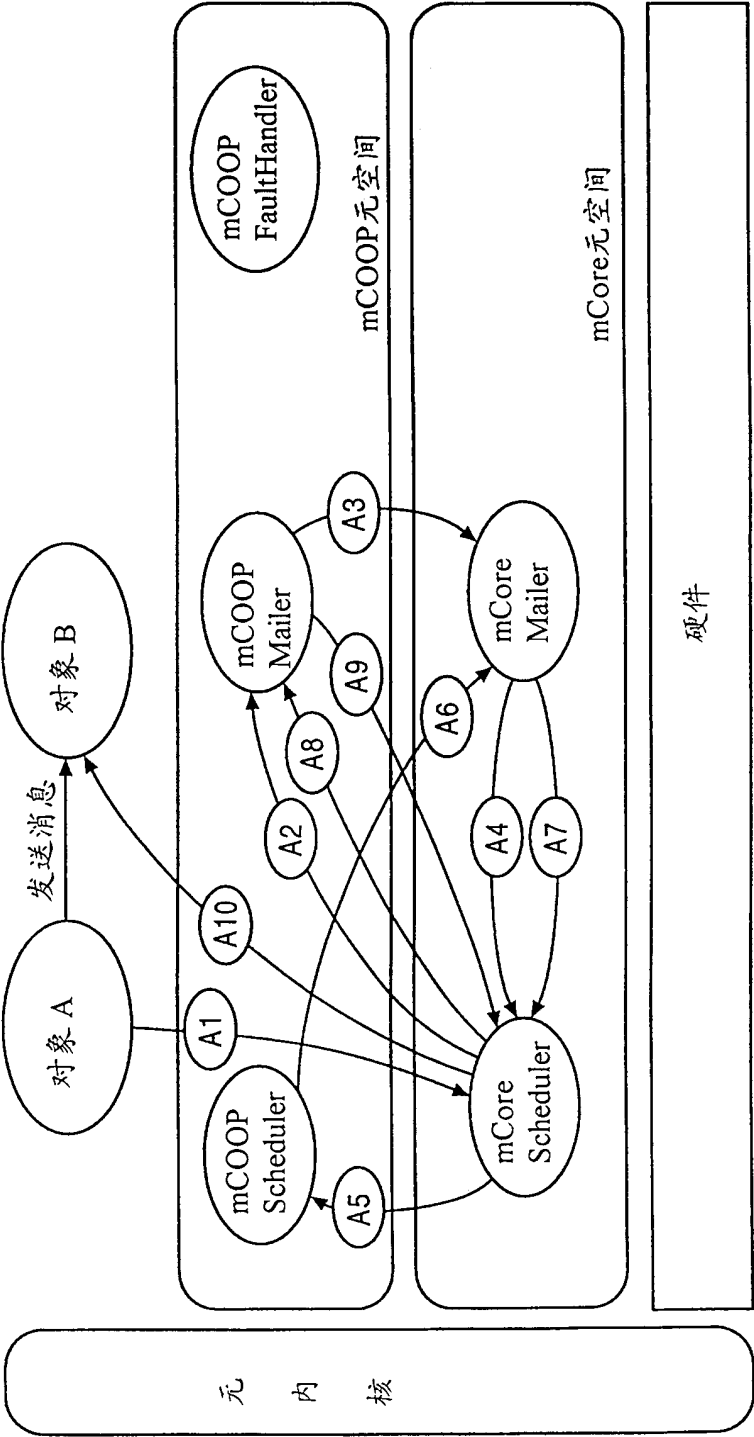


图 4

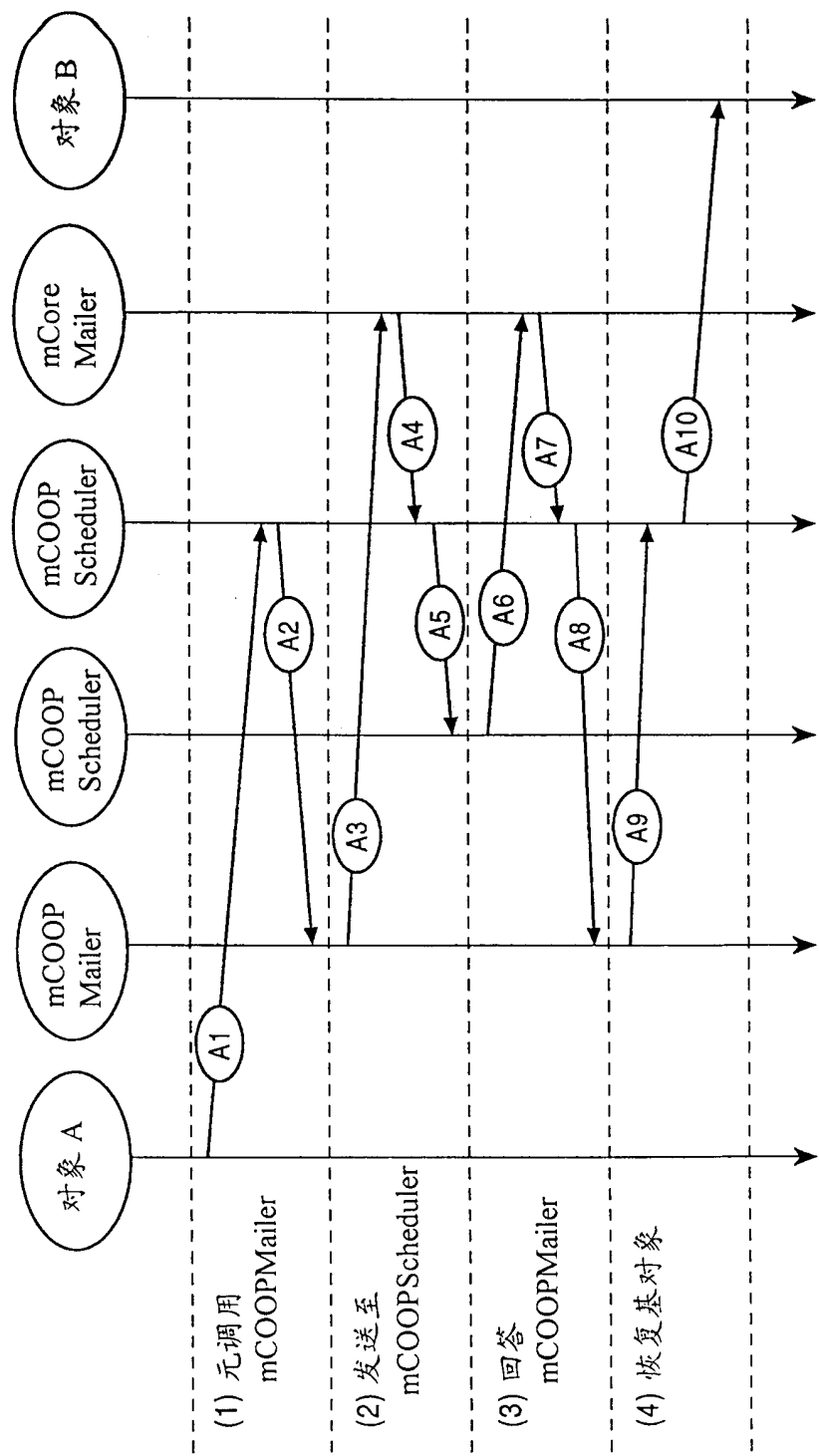


图 5

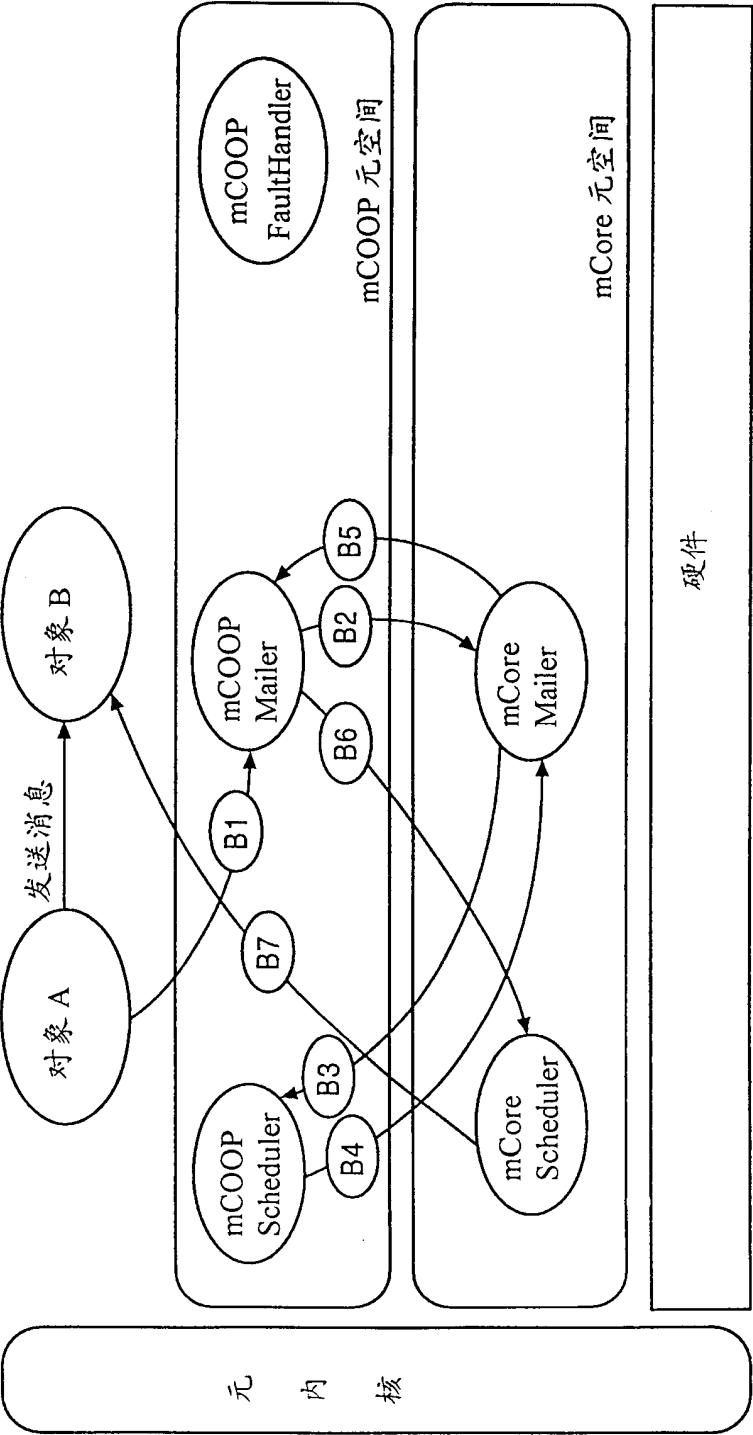


图 6

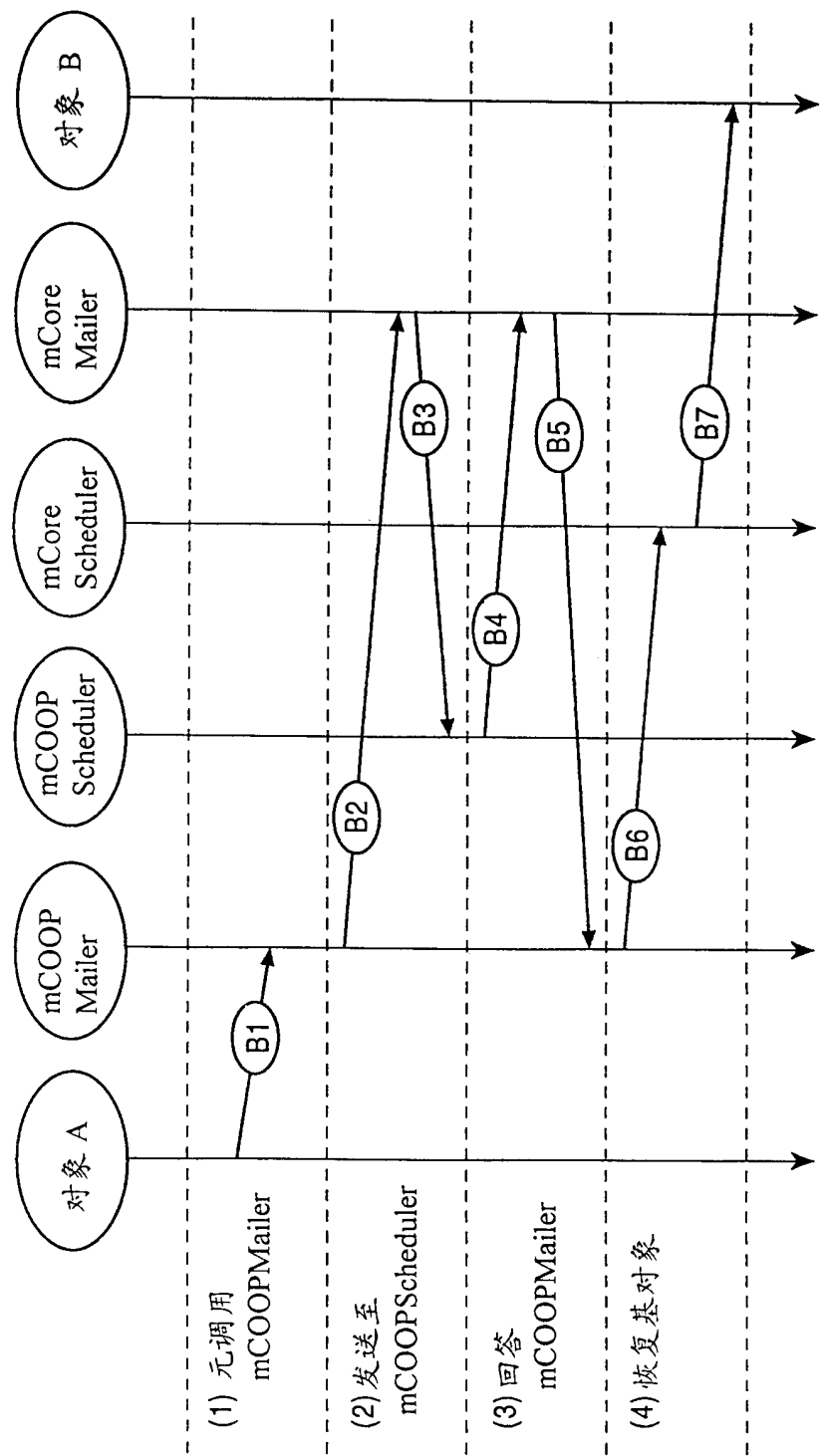


图 7

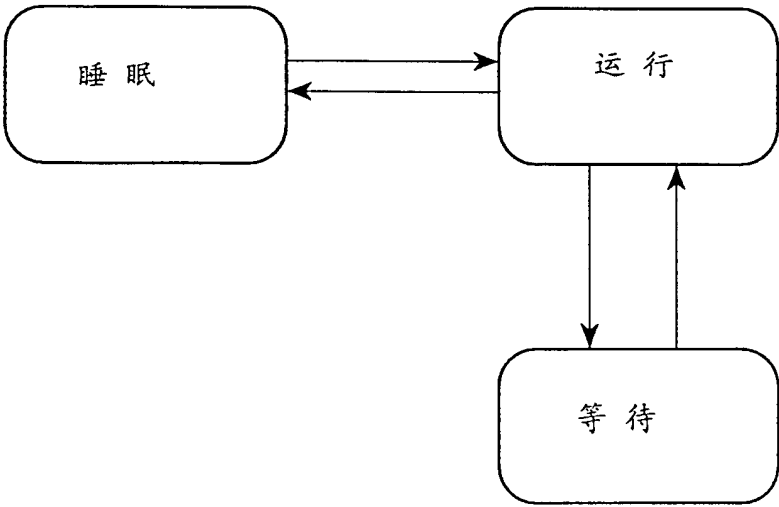


图 8

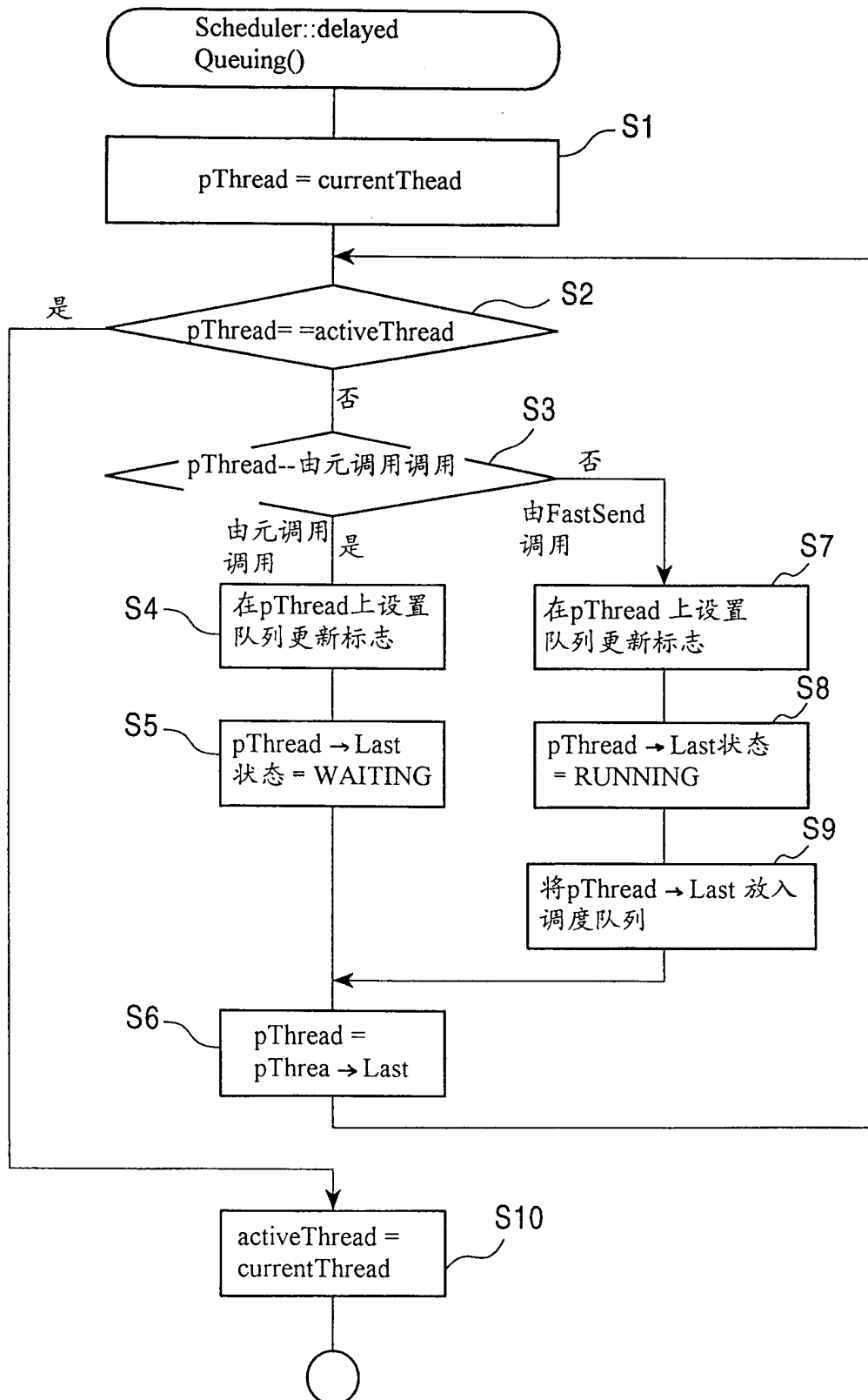


图 9

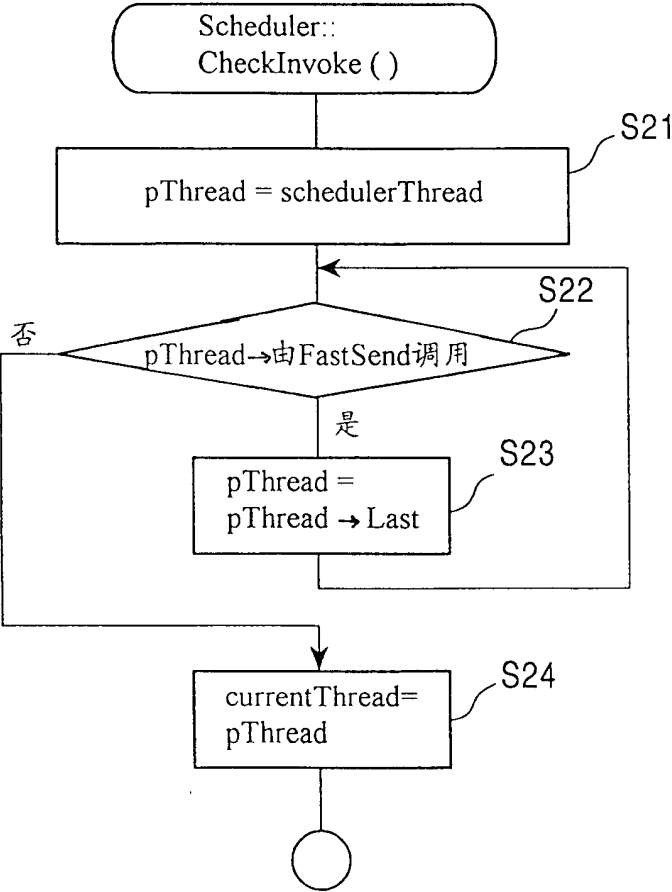


图 10

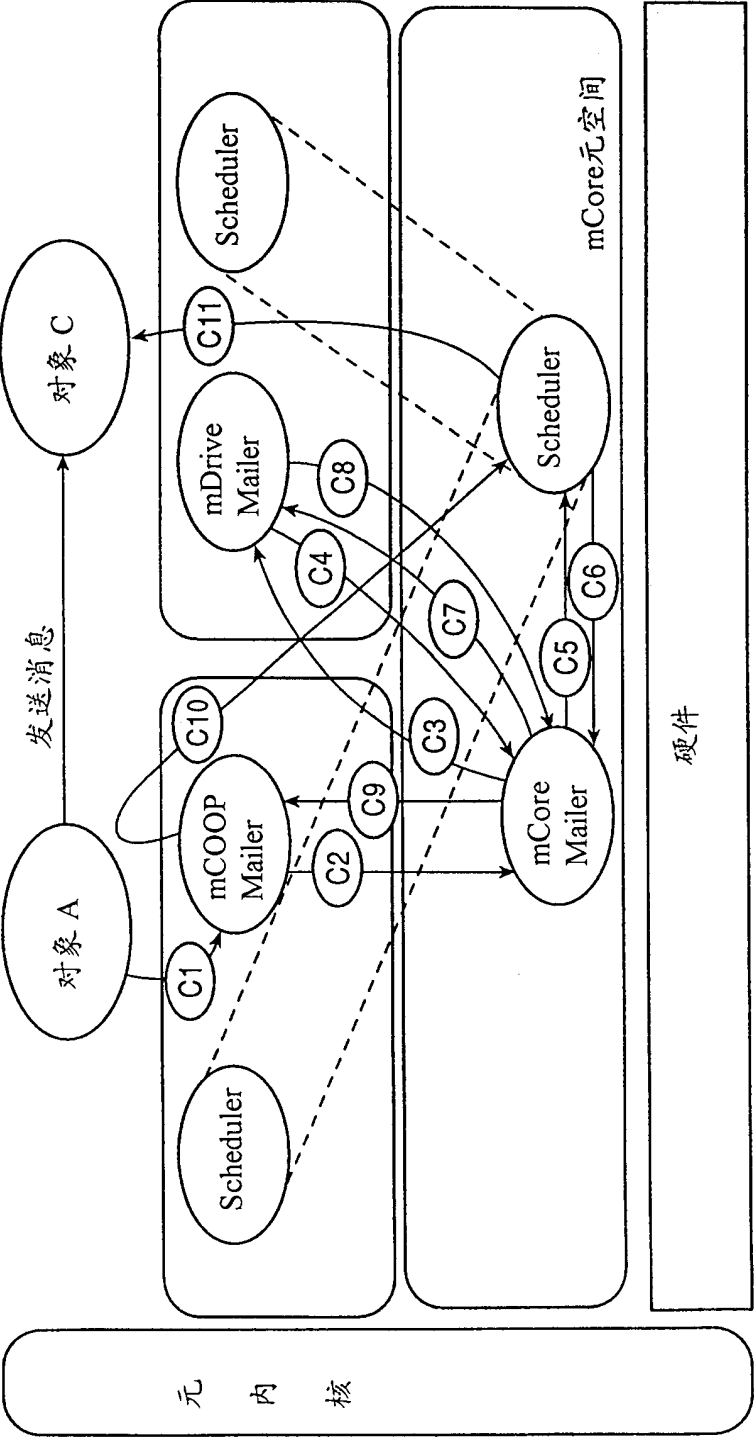


图 11

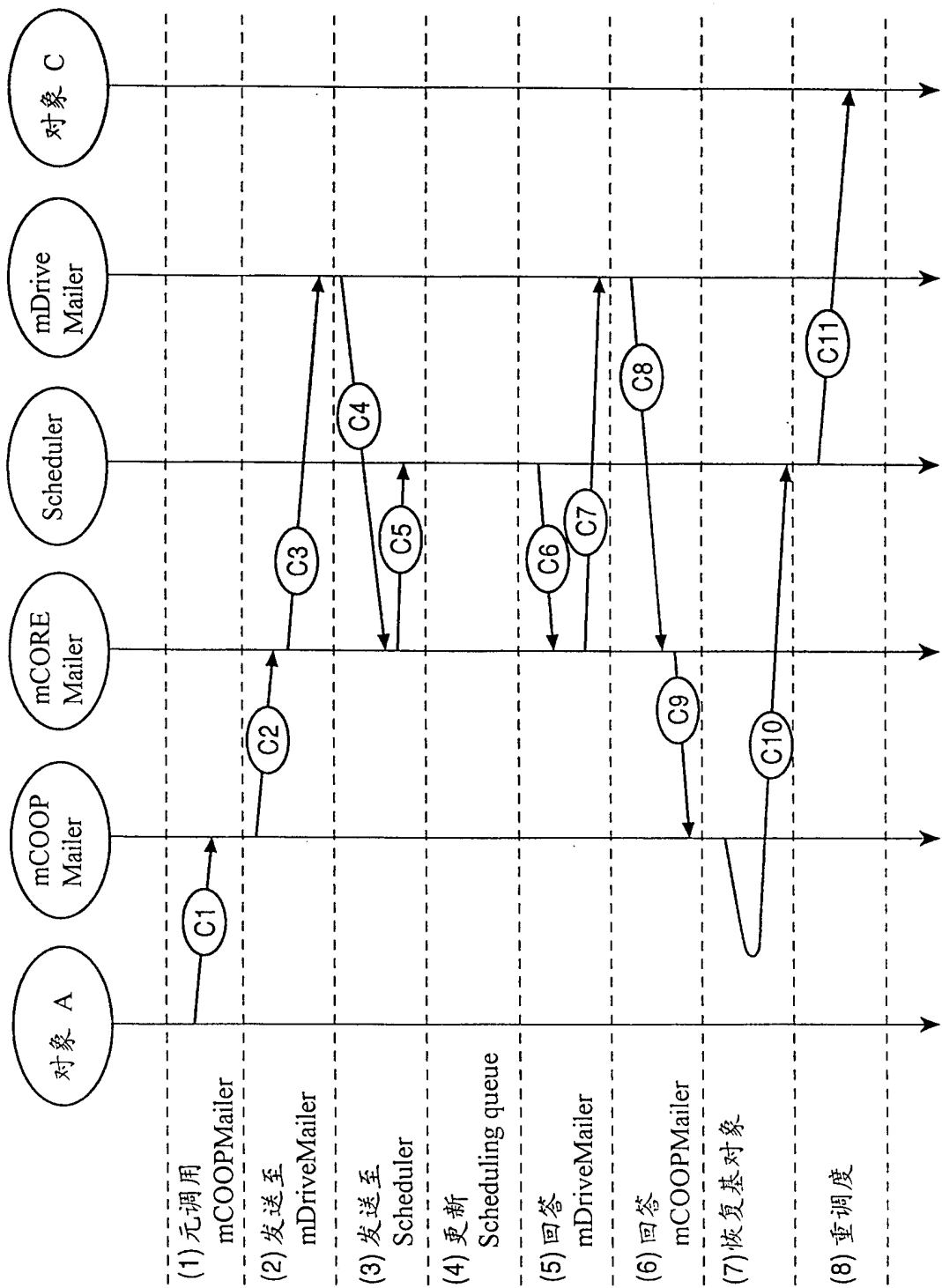


图 12

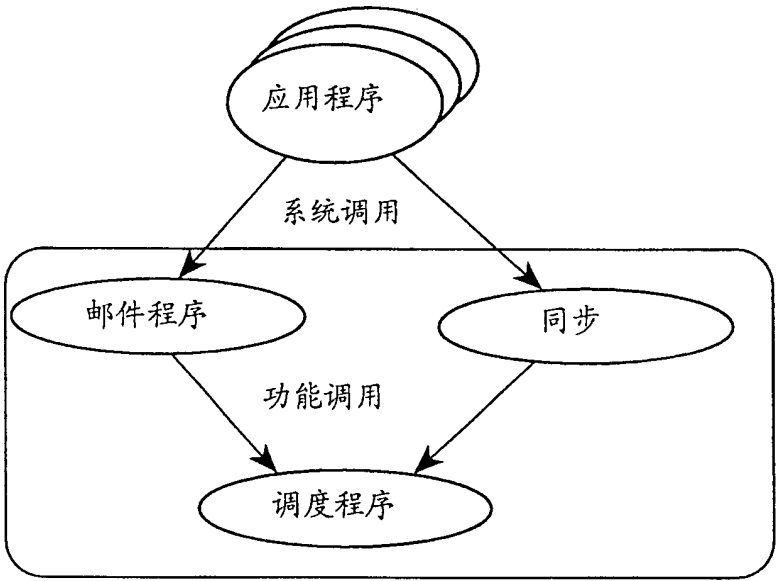


图 13

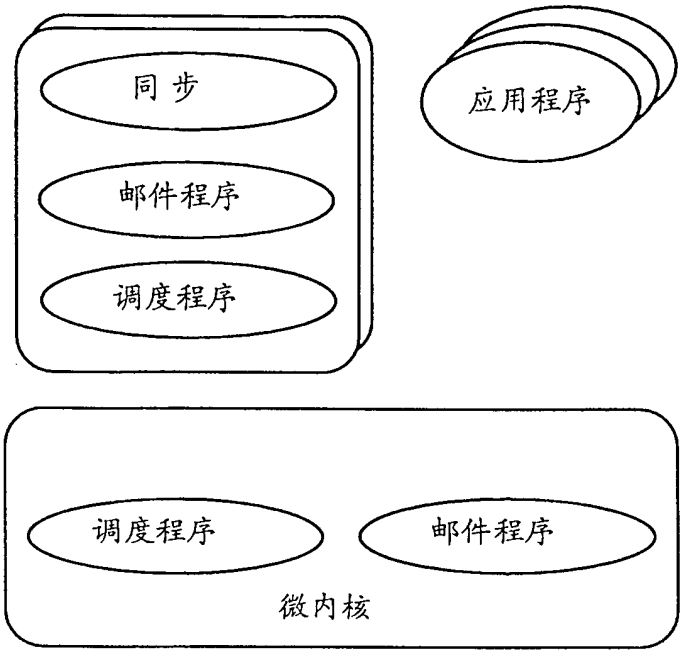


图 14

