(19)中华人民共和国国家知识产权局

(12)发明专利

(10)授权公告号 CN 102779024 B
(45)授权公告日 2016.09.28

(21)申请号 201210105283.7
(22)申请日 2005.03.31
(65)同一申请的已公布的文献号
申请公布号 CN 102779024 A
(43)申请公布日 2012.11.14
(30)优先权数据
10/816103 2004.03.31 US
(62)分案原申请数据
200580010053.0 2005.03.31
(73)专利权人 英特尔公司
地址 美国加利福尼亚州
(72)发明人 E·T·格罗乔夫斯基 H·王
J·P·沈 P·H·王
J·D·科林斯 J·P·赫尔德
P·坎杜 R·莱维亚坦
T·-F·魏
(74)专利代理机构 中国专利代理(香港)有限公司 72001
代理人 朱海煜
(51)Int.Cl.
G06F 9/30(2006.01)
G06F 9/38(2006.01)
(56)对比文件
US 5485626 A,1996.01.16,
EP 0947926 A2,1999.10.06,
CN 1384431 A,2002.12.11,
JP 平4-60864 A,1992.02.26,
审查员 梁岩

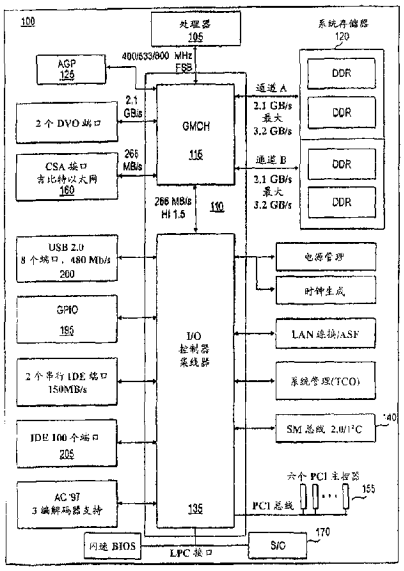
权利要求书1页 说明书26页 附图6页

(54)发明名称

提供用户级多线程操作的方法和系统

(57)摘要

本发明的名称是“提供用户级多线程操作的方法和系统”。公开一种用于提供用户级多线程操作的方法和系统。根据本发明技术的方法包括接收编程指令以通过指令集体系结构(ISA)来执行一个或多个共享资源的线程(Shred)。通过ISA来配置一个或多个指令指针;并利用微处理器同时执行这一个或多个Shred,其中该微处理器包括多个指令定序器。



CN 102779024 B

1.一种在处理器中用于提供用户级多线程操作的设备，包括：

第一专用状态寄存器，用于为将由第一非特权用户指令创建的第一共享资源的线程shred保存按照共享资源的线程的专用应用状态；

包括所述第一专用状态寄存器中保存的专用应用状态的副本的第二专用状态寄存器，用于为将由第一非特权用户指令创建的第二共享资源的线程保存按照共享资源的线程的专用应用状态；和

至少可由所述第一共享资源的线程通过第二非特权用户指令直接访问以及所述第二共享资源的线程通过第三非特权用户指令直接访问的共享的寄存器，用于在所述第一共享资源的线程和所述第二共享资源的线程之间提供通信。

2．一种用于执行多个线程的计算系统，包括：

处理器，包括用于提供用户级多线程操作的设备，所述设备包括：

第一专用状态寄存器，用于为将由第一非特权用户指令创建的第一共享资源的线程shred保存按照共享资源的线程的专用应用状态；

第二专用状态寄存器，用于为第二共享资源的线程保存按照共享资源的线程的专用应用状态，所述第二专用状态寄存器包括所述第一专用状态寄存器中保存的专用应用状态的副本，其中所述第二共享资源的线程将由所述第一非特权用户指令的第二非特权用户指令实例来创建；和

共享寄存器，用于提供所述第一共享资源的线程和所述第二共享资源的线程之间的通信，其中所述共享寄存器至少可由所述第一共享资源的线程通过第三非特权用户指令直接访问以及可由所述第二共享资源的线程通过第四非特权用户指令直接访问，以用于在所述第一共享资源的线程和所述第二共享资源的线程之间提供所述通信。

3．一种用于执行多个线程的方法，包括：

在第一专用状态寄存器中为将由第一非特权用户指令创建的第一共享资源的线程shred存储按照共享资源的线程的专用应用状态；

在第二专用状态寄存器中存储所述第一专用状态寄存器中存储的专用应用状态的副本，作为用于将由所述第一非特权用户指令的第二非特权用户指令实例创建的第二共享资源的线程的按照共享资源的线程的专用应用状态；以及

在至少可由所述第一共享资源的线程通过第三非特权用户指令直接访问以及所述第二共享资源的线程通过第四非特权用户指令直接访问的共享寄存器中，提供在所述第一共享资源的线程和所述第二共享资源的线程之间的通信。

提供用户级多线程操作的方法和系统

[0001] 本申请是申请日为2005年3月31日、申请号为200580010053.0、发明名称为“提供用户级多线程操作的方法和系统”的专利申请的分案申请。

技术领域

[0002] 本发明的实施例涉及计算机系统领域。具体来说，本发明的这些实施例涉及用于提供用户级多线程操作的方法和系统。

背景技术

[0003] 多线程操作是程序或操作系统一次执行多于一个指令序列的能力。对程序或系统服务的每个用户请求(其中用户也可以是另一个程序)是作为具有各自身份的线程来被持续跟踪的。当程序代表对该线程的初始请求而工作并由其他请求中断时，代表该线程的工作的状态被持续跟踪，直到该工作完成为止。

[0004] 计算机处理的类型包括单指令流单数据流，这是包括单指令流的常规串行冯·诺伊曼计算机。第二种处理类型是单指令流多数据流处理(SIMD)。该处理方案可以包括多个算术逻辑处理器和单个控制处理器。每个算术逻辑处理器在锁定步骤中对数据执行操作，并由控制处理器对其同步。第三种类型是多指令流单数据流(MISD)处理，这包括通过执行不同指令流的处理器的线性阵列处理相同的数据流。第四种处理类型是多指令流多数据流(MIMD)处理，它使用多个处理器，每个处理器执行各自的指令流以处理馈送到每个处理器的数据流。MIMD处理器可以具有多个指令处理单元、多个指令定序器以及由此具有多个数据流。

[0005] 今天的多线程微处理器采用的编程模型与传统共享存储器的多处理器相同：多个线程被编程为好像它们运行在独立的CPU上一样。线程之间的通信通过主存储器来执行，并且线程创建/摧毁/调度由操作系统执行。多线程操作不是以程序员可以直接访问线程的结构可视方式提供的。

发明内容

[0006] 根据第一实施例，本发明提供了一种在处理器中用于提供用户级多线程操作的设备，包括：

[0007] 第一专用状态寄存器，用于为将由第一非特权用户指令创建的第一共享资源的线程(shred)保存按照共享资源的线程的专用应用状态；

[0008] 包括第一专用应用状态的副本的第二专用状态寄存器，用于为将由第一非特权用户指令创建的第二共享资源的线程保存按照共享资源的线程的专用应用状态；和

[0009] 至少可由所述第一共享资源的线程通过第二非特权用户指令直接访问以及所述第二共享资源的线程通过第三非特权用户指令直接访问的共享的寄存器，用于在所述第一共享资源的线程和所述第二共享资源的线程之间提供通信。

附图说明

[0010] 结合附图根据下文的详细说明，将更全面地理解和认识本发明的实施例，其中：

[0011] 图1图示根据本发明一个实施例利用本发明方法和设备的示范计算机系统的框图；

[0012] 图2图示根据本发明一个实施例的示范芯片级多处理器；

[0013] 图3图示根据本发明一个实施例的示范同时多线程处理器；

[0014] 图4图示根据本发明一个实施例的示范非对称多处理器；

[0015] 图5图示根据本发明一个实施例用于提供用户级多线程操作的示范执行环境；

[0016] 图6图示根据本发明一个实施例的Shred与共享存储器的线程之间的示范关系；以及

[0017] 图7图示根据本发明一个实施例的用户级多线程操作的示范过程的流程图。

具体实施方式

[0018] 公开一种用于提供用户级多线程操作的方法和系统。根据本发明技术的方法包括接收编程指令以通过指令集体系结构(ISA)来执行一个或多个共享资源的线程(Shred)。通过ISA来配置一个或多个指令指针；并且利用微处理器同时执行这一个或多个Shred，其中该微处理器包括多个指令定序器。

[0019] 在下文描述中，出于解释的目的提出特定的术语。但是，对于本领域技术人员来说，显然这些特定的细节并非是必不可少的。下文详细描述中的一些部分是依据对计算机存储器内的数据位进行操作的算法和符号表示来给出的。这些算法描述和表示是数据处理领域的技术人员向本领域的其他技术人员最有效地表达其工作内容的方式。在本文中以及一般来说，算法被构想为是促成期望结果的独立操作序列。这些操作是需要以物理方式操纵物理量的那些操作。通常但不一定，这些量采用能够被存储、传送、组合、比较和以其他方式操纵的电信号或磁信号的形式。以位、值、元素、符号、字符、术语、数字等提及这些信号时常被证明是方便的，主要是因为常用。

[0020] 但是切记，所有这些和相似的术语均与适合的物理量相关联，并且仅仅是应用于这些量的便于使用的标号。除非明确说明，否则从上面的讨论中来看显然会意识到，整个说明书中，使用诸如“处理”、“计算”、“演算”、“确定”、“显示”等术语的讨论涉及计算机系统或类似电子计算装置的动作和/或处理，这些装置把在计算机系统的寄存器和存储器中的表示成物理(如电子)量的数据处理和/或转换成在计算机系统的存储器、寄存器或其它此类信息存储、传输、或显示装置内的同样表示成物理量的其它数据。

[0021] 所提出的本发明实施例还涉及一种用于执行本文的操作的设备。该设备可以是针对所需目的专门设计的，或它可以包括由计算机中存储的计算机程序选择性激活或重新配置的通用计算机。此类计算机程序可以存储在计算机可读存储介质中，例如但不限于包括软盘、光盘、CD-ROM和磁光盘的任何类型的磁盘、只读存储器(ROM)、随机存取存储器(RAM)、EPROM、EEPROM、磁卡或光卡、或适于存储电子指令的任何类型的介质，并且它们均耦合到计算机系统总线。

[0022] 本文提出的算法和显示并不固有地与任何特定的计算机或其他设备相关。多种通

用系统可以结合根据本发明原理的程序来使用，或构造更专用的设备来执行所必需的方法可能证明是便利的。根据下文描述，将明了用于各种此类系统的必需的结构。此外，本发明的一个实施例不是参考任何特定的编程语言来描述的。将认识到，可以使用多种编程语言来实施本文所描述的本发明实施例的原理。

[0023] 在本说明书中所用的“用户”描述用户级软件，如应用程序、非特权代码和相似的软件。用户级软件与操作系统或相似的特权软件是不同的。根据本发明的一个实施例，以下描述适用于如上所述的MIMD处理器。

[0024] 图1图示根据本发明一个实施例利用本发明方法和设备的示范计算机系统100的框图。计算机系统包括处理器105。芯片组110为系统100提供存储和I/O功能。更具体地来说，芯片组110包括图形和存储器控制器集线器(GMCH)115。GMCH 115作为与处理器105通信的主控制器，还作为主存储器120的控制器。根据本发明一个实施例，处理器105使多线程操作能够扩展到用户级。GMCH 115还提供至与之耦合的高级图形端口(AGP)控制器125的接口。芯片组110还包括执行大量I/O功能的I/O控制器集线器(ICH)135。ICH 135耦合到系统管理总线(SM总线)140。

[0025] ICH 135耦合到外设组件互连(PCI)总线155。超级I/O(“SID”)控制器170耦合到ICH 135，以提供至诸如键盘和鼠标175的输入装置的连接性。通用I/O(GPIO)总线195耦合到ICH 135。如图所示，USB端口200耦合到ICH 135。可以在该总线上将如打印机、扫描仪、游戏杆等的USB设备添加到系统配置。集成驱动器电子部件(IDE)总线205耦合到ICH 135，以将IDE驱动器210连接到计算机系统。在逻辑上来说，ICH 135看上去像是单个物理组件内的多个PCI设备。

[0026] 包括在处理器105中的是指令集体系结构。指令集体系结构(ISA)是诸如处理器105的微处理器的一种抽象模型，它由状态单元(寄存器)和对那些状态单元操作的指令组成。指令集体系结构通过为程序员和微处理器设计人员提供微处理器行为的抽象规范而作为软件与硬件之间的分界。

[0027] 硅芯片上可提供的晶体管数量上的发展已经能够将多线程操作引入到通用微处理器。多线程操作可采用两种不同的方式来实施：芯片级多处理器(CMP)和同时多线程处理器(SMT)，二者都可以用作处理器105。

[0028] 图2图示根据本发明一个实施例的示范芯片级多处理器。在如处理器200的芯片级多处理器中，多个CPU核210-213被集成到单个硅芯片200上。CPU核210-213的每一个都能够完成一个独立线程220-223的执行，即使一些资源(如高速缓存)可能为CPU核210-213中的一个以上所共享。

[0029] 图3图示根据本发明一个实施例的示范同时多线程处理器300。处理器105可以是如处理器300的同时多线程处理器。在同时多线程处理器300中，单个CPU核310能够完成多个线程的执行。CPU核310通过以极精细的粒度共享CPU资源使得对于软件来说看上去像是存在两个或两个以上处理器(常常基于逐个时钟来确定哪个线程利用每个资源来处理)。

[0030] 图4图示根据本发明一个实施例的示范非对称多处理器400。处理器105可以是如多处理器400的非对称多处理器。构建芯片级多处理器400、其中CPU核410-427具有不同的微体系结构但是具有相同的ISA，这是可能的。例如，可以将小量的高性能CPU核410-411与大量的低功率CPU核420-427集成。这种类型的设计可以实现高合计吞吐量以及高标量性

能。这两种类型的CPU核可以对于软件来说像是常规共享存储器的线程或Shred或二者的某种组合。指令集体系结构(ISA)是诸如处理器105的微处理器的一种抽象模型，它由状态单元(寄存器)和对那些状态单元操作的指令组成。ISA通过为程序员和微处理器设计人员提供微处理器行为的抽象规范而作为软件与硬件之间的分界。本发明的编程模型使应用程序能够直接控制多个非对称CPU核。

[0031] 共享存储器的编程模型

[0032] 现有技术的多线程微处理器采用与现有技术的共享存储器的多处理器系统相同的编程模型。该编程模型如下文所述。微处理器向操作系统提供执行的多个线程。操作系统使用这些线程来并发运行多个应用(“进程”)，和/或并发运行来自单个应用的多个线程(“多线程”)。在这两种情况中，对于软件来说线程看上去像是独立的CPU。主存储器为所有线程所共享，线程之间的通信经由主存储器来完成。CPU内的硬件资源也可以被共享，但是微体系结构将这种共享对软件隐藏。

[0033] 虽然传统的共享存储器的多处理器编程模型广泛地被多种操作系统和应用程序所理解和支持，但是这种模型具有多个缺点。它们是：

[0034] 1) 线程之间的通信通过主存储器来完成，因此极慢。缓存可以缓解一些等待时间，但是通常必须将高速缓存行从一个CPU核传给另一个，以便利于共享。

[0035] 2) 线程之间的同步使用基于存储器的信号量(semaphore)来完成，因此极慢。

[0036] 3) 创建、摧毁、挂起和恢复线程需要操作系统介入，因此极慢。

[0037] 4) 微处理器供应商无法提供最有效的多线程操作方法，因为CPU多线程操作中的改进被上面描述的存储器等待时间以及操作系统等待时间所抵消。

[0038] 多线程操作体系结构扩展

[0039] 出于有关现有技术系统的上述原因，本发明的方法和系统通过多线程操作体系结构扩展将处理器体系结构扩展为包括结构可视的多线程操作。提供在单个处理单元内的多个同时执行的线程、多个指令指针以及某种应用状态(寄存器)的多个副本。执行的多个线程不同于现有的共享存储器的线程，它们称为Shred或共享资源的线程。

[0040] 本发明的多线程操作体系结构扩展(它的一个示例在下文称为“MAX”)包括现有体系结构功能，另外还支持多个同时的Shred，它们均具有自己的指令指针、通用寄存器、FP寄存器、分支寄存器、预测寄存器和某些应用寄存器。创建非特权指令以创建并摧毁Shred。除共享的存储器外，Shred之间的通信还通过共享的寄存器来完成。对信号量的需求将减少，因为本发明的多线程操作体系结构扩展会保证对共享的寄存器的原子访问。此外，本发明的多线程操作体系结构扩展可以结合32位体系结构、如Intel®公司的32位体系结构，或64位体系结构、如Intel®公司的64位体系结构，或甚至16位体系结构来使用。

[0041] 根据本发明一个实施例，下表中示出常规共享存储器的多处理器线程与Shred之间的比较。

[0042]

操作	共享存储器的多处理器线程	多线程操作体系结构扩展Shred
创建、摧毁	操作系统调用	非特权指令
通信	共享的存储器	共享的寄存器和存储器
同步	存储器信号量	寄存器和存储器信号量。共享的寄存器保证原子更新。
系统状态	对于每个线程唯一的系统状态	对于所有Shred共享的系统状态

[0043] 表1

[0044] 应该注意，本发明的多线程操作体系结构扩展在根本上不同于现有技术的体系结构扩展。虽然现有技术的体系结构扩展提供更多的指令和更多的寄存器(状态)，但是多线程操作体系结构扩展提供更多的执行单元。

[0045] 应用和系统状态

[0046] 程序员可视的CPU状态可以划分成两类：应用状态和系统状态。应用状态由应用程序和操作系统共同使用和控制，而系统状态由操作系统独占地控制。

[0047] 图5图示根据本发明一个实施例用于提供用户级多线程操作的示范执行环境。执行环境600包括下表中汇总其应用状态的寄存器：

[0048]

32 位体系结构应用状态	名称	宽度
通用寄存器 605	EAX、EBX、ECX、EDX、 EBP、ESI、EDI、ESP	32 位
浮点寄存器 625	ST0-7	80 位
段寄存器 610	CS、DS、ES、FS、GS、 SS	16 位
标志寄存器 615	EFLAGS	32 位，某些位是 应用
指令指针 620	EIP	32 位
FP 控制和状态寄存器 626-631	CW 626 SW 627 TW 628 FP 操作码 629 指令指针 630 运算数指针 631	16 位 16 位 16 位 11 位 48 位 48 位
MMX 寄存器 635	MM0-7	64 位，别名为 ST0-7
SSE 寄存器 640	XMM0-7	128 位
MXCSR 寄存器 645	MXCSR	32 位

[0049] 表2

[0050] 下文将更详细地描述用户级多线程操作寄存器650-665。

[0051] 下表汇总32位体系结构系统状态。

[0052]

32位体系结构系统状态	编号	宽度
控制寄存器 626	CR0-CR4	32位
标志寄存器 615	子集EFLAGS	32位，子集
存储器管理寄存器	GDTR、IDTR	48位
本地描述符表寄存器、任务寄存器	LDTR、TR	16位
调试寄存器、	DR0-DR7	32位
模型专用寄存器 650	MSR0-MSRN	64位 包括用于时间戳计数器、APIC、机器检查、存储器类型范围寄存器、性能监视的寄存器
共享的寄存器 655	SH0-SH7	32位
Shred控制寄存器 660	SC0-SC4	32位

[0053] 表3

[0054] 对于每个Shred来说，应用状态划分成两类：按照Shred的应用状态和共享的应用状态。本文所述的MAX编程模型提供按照Shred的应用状态的唯一实例，而共享的应用状态在多个Shred之间共享。系统状态仅有一个副本，对应于给定线程的所有Shred共享相同的系统状态。下表中给出了应用和状态的大致划分：

[0055]

状态	类型
通用寄存器(可编程子集) 浮点寄存器(可编程子集) SSE 寄存器(可编程子集) 指令指针 标志(应用子集)	按照 Shred 的专用状态
通用寄存器(可编程子集) 浮点寄存器(可编程子集) SSE 寄存器(可编程子集) 共享的寄存器(新的) 标志(系统子集) 存储器管理寄存器 地址转换(TLB) 当前特权级 控制寄存器	在多个 Shred 之间共享，专用于每个线程
主存储器	在多个线程之间共享

[0056] 表4

[0057] 本发明的多线程操作体系结构扩展提供大多数应用状态的可编程共享或保密，以便软件可以选择最佳分区。利用位向量来执行编程，由此可以将各个寄存器选为共享的或专用的。硬件重命名器可以按位向量所指定的从共享池或专用池中分配寄存器。

[0058] MAX的整个存储需求小于现有技术的同时多线程处理器和芯片级多处理器的需求。在MAX中，仅复制按照Shred的专用应用状态，而在实施传统共享存储器的多处理器编程模型的同时多线程处理器或芯片级多处理器中，必须复制整个应用和系统状态。

[0059] Shred/线程层次结构

[0060] 每个共享存储器的线程由多个Shred组成。Shred和共享存储器的线程构成两级层次。在备选实施例中，可以根据共享存储器的MAX处理器簇构建三级层次。这些簇使用消息传递来通信。操作系统处理线程的调度，而应用程序处理Shred的调度。在任何给定的Shred视其他Shred为本地的或远程的意义上来说，Shred是非一致的。对于每个Shred，复制按照Shred的应用状态。共享的应用和系统状态对于本地Shred是公用的，并且为每个共享存储器的线程复制它。存储器状态仅有一个副本。

[0061] 图6图示根据本发明一个实施例在Shred与共享存储器的线程之间的示范关系。对于每个Shred，复制按照Shred的应用状态510。共享的应用和系统状态520对于本地Shred是

公用的，并且为每个共享存储器的线程复制它。存储器状态530仅有一个副本。

[0062] 因为系统状态520在MAX编程模型中在多个Shred之间共享，所以这些多个Shred属于相同的进程。本发明的多线程操作体系结构扩展旨在供多线程应用、库和虚拟机使用。MAX编程模型为这种类型的软件提供对它的Shred前所未有程度的控制以及利用上述共享存储器的多处理器编程模型无法实现的性能潜力。

[0063] 在Shred之间无需任何保护检查，因为它们全部以相同的特权级运行并共享相同的地址转换。因此，可以在Shred间通信期间避免传统保护机制。

[0064] 由于共享的系统状态的原因，MAX编程模型无法用于在相同线程上运行不同的进程。为此，MAX和现有技术的共享存储器的编程模型共存于相同的系统内。

[0065] 因为给定的CPU提供有限数量的物理Shred，所以软件采用与硬件线程的虚拟化相似的方式对该数量的可用Shred虚拟化。虚拟化产生有限数量的并行运行的物理Shred以及潜在无限数量的虚拟Shred。

[0066] 系统调用

[0067] 可以通过将控制从应用程序转移到操作系统并执行上下文切换来以常规方式处理操作系统调用。就MAX体系结构而言，一个关键的不同是，在任何Shred上调用操作系统都将挂起与给定线程相关联的所有Shred的执行。操作系统负责保存并恢复属于相同线程的所有Shred的状态。

[0068] 由于附加的状态，上下文切换开销增加。上下文切换存储器占用与Shred的数量成比例的增长。但是，上下文切换时间不会增加太多，因为每个Shred可以与其他Shred并行地保存/恢复它的状态。上下文切换机制允许使用多个定序器来实施并行状态保存/恢复。操作系统本身利用多个Shred。

[0069] 因为调用操作系统的成本增加，所以过去由操作系统执行的某个功能迁移到应用程序。该功能包括某些异常和中断的处理和线程维护。

[0070] 执行系统调用的一个备选实施例是基于如下的观察：线程正变得便宜，而上下文切换正变得昂贵。在该实施例中，一个线程专用于运行操作系统，而第二线程专用于运行应用程序。当应用程序Shred执行系统调用时，它将消息发送到操作系统Shred(通过共享的存储器)并等待响应消息。以此方式，消息传递和等待机制替代了常规控制转移和上下文切换机制。无需对任何线程的地址转换做出更改。其好处是，一个Shred向操作系统发送的消息不会干扰其他本地Shred。

[0071] 异常

[0072] 在现有技术的体系结构中，异常会挂起应用程序的执行，并调用操作系统异常处理程序。在MAX编程模型下，该行为是不合乎要求的，因为挂起一个Shred来调用操作系统会导致所有Shred(与给定线程相关联的)也被挂起。

[0073] 为了解决该问题，我们引入一种新的用户级异常机制，它为应用程序提供服务多种类型异常的第一次机会。用户级异常机制是基于如下的观察：很少现有异常类型最终是由应用本身来服务的。

[0074] 对于用户级异常机制，如何报告异常和如何服务异常是不同的。异常可以划分成如下三种类别。

[0075] 1．报告给应用程序并由应用程序服务的异常。例如，除以零异常被报告给导致异

常的应用，并由该应用来服务。无需或不期望任何操作系统介入。

[0076] 2．报告给应用程序然后必须调用操作系统来服务的异常。应用产生的页故障可以报告给该应用，但是应用程序必须调用操作系统来换入该页。

[0077] 3．必须报告给操作系统并由操作系统服务的异常。出于安全性原因，硬件中断必须报告给操作系统。系统调用(软件中断)显然必须报告给操作系统。

[0078] 下表说明这三种类别的每一种中的异常。提供作为与本发明的一个实施例相关的异常类型的“高速缓存未命中的加载异常”和“精细粒度化的定时器”异常类型。

[0079]

异常类型	报告给	服务提供
除以零、溢出、边界、FP异常	应用	应用
对齐检查	应用	应用
无效操作码	应用	应用
高速缓存未命中的加载异常	应用	应用
精细粒度化的定时器	应用	应用
栈段故障	应用	系统
一般性保护	应用	系统
页故障	应用	系统
双重故障	应用	系统
设备不可用	应用	系统
硬件中断	系统	系统
不可屏蔽中断	系统	系统
软件中断(INT n)	系统	系统

[0080] 表5

[0081] 报告给应用程序的异常在应用内选择性地被处理，或被传递到操作系统以进行处理。在后一种情况中，应用程序响应异常(例如页故障)执行系统调用来明确地从操作系统请求服务。这与操作系统隐含地代应用执行此类服务的传统方法形成对照。为了避免嵌套的异常，提出特殊的规定以确保将异常中继到操作系统的应用代码自己不会引起附加的异常。该用户级异常机制在影子寄存器组中保存最小数量的CPU寄存器，并且处理器引向到(vector to)固定位置。

[0082] 虚拟机

[0083] 虚拟机和本发明实施例的多线程操作体系结构扩展彼此对对方施加约束，因为无论何时只要软件尝试访问正被虚拟化的资源，虚拟机就会产生异常，并且异常处理对Shred有重大性能影响。

[0084] 在虚拟机中，特权指令的执行或对特权处理器状态的访问产生异常。该异常必须报告给虚拟机监视器(并由它服务)。在MAX中，由操作系统(以及虚拟机监视器)服务的异常导致与给定线程相关联的所有Shred被挂起。虚拟机监视器了解多个Shred的存在。虚拟机体系结构将非特权指令和处理器资源上产生的异常的数量降至最小。

[0085] 死锁

[0086] 在MAX体系结构中死锁防止是复杂的，因为Shred可能被其他本地Shred挂起。应用

软件确保，如果一个Shred引起OS服务的异常或系统调用，从而导致所有本地Shred被挂起，则将不会发生死锁。

[0087] 本地(Shred间)通信和同步不同于远程(线程间)通信和同步。使用共享的寄存器655(如图5所示)或共享的存储器来执行本地通信。使用共享的存储器来执行远程通信。使用原子寄存器更新、寄存器信号量或存储器信号量来执行本地数据同步。使用存储器信号量来执行远程数据同步。

[0088] 本地和远程Shred控制(创建、摧毁)都使用MAX指令来执行。Shred控制不调用操作系统来执行wait()或yield()，因为这可能对在给定的线程上挂起所有Shred产生非期望的影响。通过对用户级Shred库的调用来替代线程维护所用的操作系统调用。继而Shred库调用操作系统来按需要创建并摧毁线程。

[0089] Shred和纤程

[0090] Shred不同于现有技术的操作系统中实施的纤程。下表汇总了这些不同：

[0091]

特征	纤程	Shred
创建	一个线程可以创建多个纤程	一个线程可以创建多个Shred
并行性	一个线程在任何时间瞬间只能运行一个纤程	一个线程可以同时运行多个Shred
调度	由软件使用协作式多任务机制来调度纤程	由硬件使用同时多线程操作或芯片级多处理来调度Shred
状态	每个纤程具有它自己的专用应用状态	每个Shred具有它自己的专用应用状态
状态存储	当前运行的纤程的状态被存储在寄存器中。不活动的纤程的状态被存储在存储器中。	每个当前运行的物理Shred的状态被存储在芯片内寄存器中。不活动的虚拟Shred的状态被存储在存储器中。
状态管理	操作系统在上下文切换上保存/恢复当前运行的纤程的状态	操作系统在上下文切换上保存/恢复所有Shred的应用状态

[0092] 表6

[0093] 硬件实施

[0094] 支持多线程操作体系结构扩展的微处理器的实施可以采用芯片级多处理器(CMP)和同时多线程处理器(SMT)的形式。现有技术的CMP和SMT处理器试图将CPU资源的共享对软件隐藏，而当利用本发明实施例的多线程操作体系结构扩展来实施时，处理器公开作为体系结构的一部分的共享。

[0095] 为了将MAX处理器实施为芯片级多处理器，使用一种广播机制以在CPU核之间的同步时保存系统状态的多个副本。引入快速通信总线用于共享的应用和系统状态。因为片内通信相对比片外存储器快速，所以这些通信总线赋予MAX处理器超过共享存储器的多处理器的性能优势。

[0096] 将MAX处理器作为同时多线程处理器来实施是可能的，因为硬件已经提供资源的必要共享。在多线程32位处理器上几乎完全以微代码形式实施MAX是可能的。

[0097] 根据一个实施例，本发明的方法和系统允许在多个Shred之间进行系统调用和异常(报告给OS)的优先级设置，以便在任何时间瞬间仅服务一个Shred的请求。一个请求的优先级设置和选择是必要的，因为系统状态一次只能够处理一个OS服务请求。例如，假定

Shred 1和Shred 2同时执行系统调用。优先级设置器确保仅Shred 1的系统调用被执行，而Shred 2的系统调用尚未开始执行。为了公平性考虑，优先级设置器采用循环选择算法，显然也可以使用其他选择算法。

[0098] 可伸缩性

[0099] MAX编程模型的可伸缩性由如下因素确定：

[0100] 1）可在上下文切换上保存/恢复的状态的数量；

[0101] 2）上下文切换期间因挂起与给定线程相关联的所有Shred而导致的并行度的降低；

[0102] 3）Shred间的通信。

[0103] 随着Shred数量的增加，必须在上下文切换上保存/恢复的状态的数量也增加，并且因挂起所有Shred而导致丢失的潜在并行度也随之增加。这两个因素将限制Shred的实际数量。

[0104] Shred间通信还将限制可伸缩性，因为该通信使用片内资源来执行。相比之下，传统的共享存储器的多处理器模型的可伸缩性受限于片外通信。

[0105] 共享的分类信息(Taxonomy)

[0106] 下表给出Shred的软件使用、实施和体系结构上的各种自由度的分类信息：

[0107]

属性	选项 1	选项 2	选项 3
指令集体系结构	同构 - 所有 Shred 实施相同的指令集体系结构	异构 - Shred 实施不同的指令集体系结构	
微体系结构实施	对称 - 所有 Shred 运行在相同的硬件微体系结构上	非对称 - Shred 运行在不同的硬件微体系结构上	
应用并行度	顺序 - 常规顺序代码	并行 - 并行化的代码	
Shred 生成	程序员生成的 - Shred 由程序员明确地创建	编译的 - Shred 由编译器自动创建	固定的功能 - Shred 专用于特定功能，如垃圾收集
体系结构正确性	体系结构 - 所有 Shred 都对程序的体系结构正确性给予贡献	提示(hint) - 一些 Shred 对体系结构正确性给予贡献，而另外的 Shred 仅对性能给予贡献	
输入/输出	计算。Shred 仅执行计算。	I/O。除了计算外、Shred 还执行输入/输出。	

[0108] 表7

[0109] 两种不同类型的MAX体系结构区分为:同构和异构。同构Shred与同构多处理器相似,所有Shred都执行相同的指令集。异构Shred也可能与异构多处理器相似。例如,异构Shred可以构造于如下两种处理器之间:

[0110] · 32位处理器和网络处理器。

[0111] · 32位处理器和64位处理器。

[0112] 相似地,底层微体系结构可以是对称的或是非对称的。后者的一个示例是包含少量大且高性能的CPU核和大量小且低功率的CPU核的芯片级多处理器,如图4所示的处理器。

[0113] 使用模型

[0114] 下表汇总了本发明的多线程操作体系结构扩展的实施例的多个使用模型:

[0115]

使用模型	分类信息	描述	优点

预取	同构ISA、顺序代码、编译器生成的、提示、计算	辅助器(helper)线程在主线程之前预取存储单元。辅助器线程由编译器生成。	加速标量代码，高速缓存未命中中花费大量时间
常规线程的替代	同构ISA、并行代码、程序员生成的、体系结构、计算	使用Shred替代常规的共享存储器的线程。库提供线程服务，以替代操作系统。	加速线程化的代码。线程原语变得加快多个数量级。
编译器的专用执行资源	同构ISA、顺序代码、编译器生成的、体系结构、计算	编译器根据标量源代码创建多个Shred。	编译器具有对Shred的直接控制。
受控的运行时环境的专用线程	同构ISA、固定的功能、体系结构、计算	Shred专用于受控的运行时功能。例如，可以使用专用的Shred来执行准时转换和垃圾收集。	转换和垃圾收集Shred变得实质上免费
并行编程语言	同构ISA、并行代码、程序员生成的，体系结构、计算	程序员创建编译成多个Shred的并行代码。	线程原语足够快速，以用作指令。
具有集成I/O功能的CPU	异构ISA、并行代码、程序员生成的、体系结构、输入/输出	I/O功能直接由应用程序执行。例如图形和网络处理。	允许将I/O功能直接集成到CPU体系结构中。
同时多ISA CPU	异构ISA、非对称uarch、程序员生成的、体系结构、计算	单个CPU实施多个ISA，例如32位体系结构和64位体系结构。每个ISA都可供程序员用作Shred。	可能受关注，但是可能没有用。
非对称核多处理器	同构ISA、非对称uarch、体系结构、计算	CMP实施核的混合，例如高性能与低功率。	实现良好的标量和吞吐量性能。

[0116] 表8

[0117] 预取

[0118] 在预取使用模型中，主线程产生用于从主存储器预取高速缓存行的一个或多个辅助器线程。辅助器线程是响应主线程上的高速缓存未命中产生的。因为主存储器访问需要数百到上千个CPU时钟来完成，所以标量代码的执行实际将在主存储器访问期间停止，除非对高速缓存未命中并进入到主存储器的加载故障进行体系结构规定。

[0119] 常规线程的替代

[0120] Shred可以被多线程应用用作常规线程的高性能替代。提供用户级软件库以执行先前由操作系统执行的Shred管理功能(创建、摧毁等)。该库利用Shred指令并且按需要调用操作系统以请求附加的线程。调用软件库远比调用操作系统快速，因为无需上下文切换。

[0121] 编译器的专用执行资源

[0122] 编译器可以按它使用其他处理器资源(如寄存器)的相同方式使用Shred。例如，编译器可以将处理器视为具有8个整数寄存器、8个浮点寄存器、8个SSE寄存器和4个Shred。通过将Shred视为资源，编译器以类似于寄存器分配的方式分配Shred。与寄存器一样，需要某种机制以在应用程序需要比硬件所提供的虚拟Shred更多的虚拟Shred的情况中将Shred溢出/填充到后备储存。在现有技术的体系结构中，常常不将控制流视为是处理器资源，因为只有一个。

[0123] 受控的运行时环境的专用线程

[0124] 在受控的运行时环境中，Shred专用于诸如垃圾收集、准时编译和成形

(profiling)的功能。Shred实质上“免费”执行此类功能，因为Shred是作为指令集体系结构(ISA)的一部分提供的。ISA是对于程序员或编译器编写者可视的处理器的一部分。ISA用作软件和硬件之间的分界。

[0125] 并行编程语言

[0126] MAX直接支持并行编程语言和硬件描述语言。例如，iHDL或Verilog编译器直接生成用于多个Shred的代码，因为源代码明确地是并行的。

[0127] 通过芯片级多处理器使线程的扩散成为可能，这促成多线程操作的语言支持。这种支持通过对操作系统和运行时库的调用来提供。对多线程操作的语言支持迁移到主流通用的编程语言。

[0128] 具有集成I/O功能的CPU

[0129] Shred用于实施如网络协处理器的I/O功能。作为Shred实施的网络协处理器之间的一个重要不同之处是，它看上去像是CPU的一部分，而非I/O设备。

[0130] 在现有技术的系统中，当应用程序需要I/O时，该应用程序使用API(应用程序接口)调用操作系统。继而操作系统调用将该请求发送到I/O设备的设备驱动程序。操作系统负责将来自多个应用程序的I/O请求排队或串行化，以确保I/O设备一次仅处理一个(或有限个)请求。这是必要的，因为I/O设备的状态对于系统是全局的，而CPU状态在多个应用之间是时分多路复用的。

[0131] 在实施为异构Shred的I/O设备中，I/O设备的状态被视为是CPU的应用状态的扩展。应用程序直接控制CPU的应用状态和I/O设备状态。应用状态和I/O状态都由操作系统在上下文切换上保存/恢复。I/O设备构造为使它的状态可以在多个应用之间时分多路复用，而不会有负面影响。

[0132] 同时多ISA CPU

[0133] 64位体系结构定义为通过一种称为“无缝”的机制包括32位体系结构应用体系结构以及新的64位指令集。与32位体系结构指令集的兼容性使64位体系结构处理器能够运行现有32位体系结构应用，也可以运行新的64位体系结构应用。

[0134] 在当前定义下，64位体系结构CPU在任何时间瞬间运行64位体系结构线程或32位体系结构线程。这两种ISA之间的切换通过64位体系结构br.ia(分支到32位体系结构)和32位体系结构jmpe(跳至64位体系结构)指令来实现。32位体系结构寄存器被映射到64位体系结构寄存器，使得只需要状态的一个副本。

[0135] 可以创建多ISA CPU，其中在任何时间瞬间多于一个指令集体系结构在运行。这可以通过使用64位体系结构ISA的Shred和32位体系结构ISA的第二Shred来实现。与同构Shred一样，必须为64位体系结构的Shred和32位体系结构的Shred提供相异的应用状态。64位体系结构的Shred和32位体系结构的Shred同时运行。

[0136] 描述了通过上述多线程操作体系结构扩展来提供用户级多线程操作的本发明方法和系统的特征之后，下文给出32位系统的实施例。

[0137] 32位体系结构实施例

[0138] 虽然是参考IA-32体系结构描述的，但是读者理解，本文所描述的方法和系统可以应用于其他体系结构，如IA-64体系结构。此外，读者回顾图5，还理解根据本发明一个实施例的示范执行环境。将少量指令添加到IA-32 ISA以及多个寄存器650-660，以对IA-32提供

用户级多线程操作的能力。

[0139] 该多线程操作体系结构扩展由如下状态构成：

[0140] · 模型专用寄存器650(MAX_SHRED_ENABLE)，供操作系统或BIOS用于启用/禁用这些扩展。

[0141] · CPUID中的三个位扩展特征信息，特征信息指示处理器是否实施扩展以及可用的物理Shred的数量。

[0142] · 大多数应用状态的复制(EAX、EBX等)，使得每个Shred具有它自己的应用状态专用副本。

[0143] · 一组共享的寄存器SH0-SH7 655，它们可以用于Shred之间的通信和同步。

[0144] · 一组控制寄存器SC0-SC4 660，用于Shred管理。

[0145] 该多线程操作体系结构扩展由如下指令构成：

[0146] · Shred创建/摧毁：forkshred、haltshred、killshred、joinshred、getshred

[0147] · 通信：向共享的寄存器655 mov/从共享的寄存器655 mov，向共享的寄存器655同步mov/从共享的寄存器655同步mov。

[0148] · 同步(信号量)：cmpxshgsh、xaddsh、xchgsh

[0149] · 发信号：signalshred

[0150] · 到/从多Shred化模式的转换：entermsm、exitmsm

[0151] · 状态管理：shsave、shrestore

[0152] · 其他：向Shred控制寄存器mov/从Shred控制寄存器mov。

[0153] 此外，IA-32机制设有如下功能。

[0154] · IA-32异常机制退出多Shred化模式，并保存有关异常的所有Shred状态(当适用时)。

[0155] · IA-32 IRET指令恢复所有Shred状态并返回到多Shred化模式(当适用时)。

[0156] · 引入用户级异常机制。

[0157] 配置

[0158] 使用模型专用寄存器(MSR)650来启用多线程操作体系结构扩展。下文描述MSR。

[0159]

寄存器地址(十六进制)	寄存器地址(十进制)	寄存器名称字段和标志	共享的/唯一的	位描述
1F0H	496	MAX_SHRED_ENABLE	共享的	位0启用多线程操作体系结构扩展。复位时初始化至0。操作系统或BIOS必须通过将1写入该寄存器来明确启用MAX。

[0160] 表9

[0161] 仅在特权级0写和读如Shred MSR 650的模型专用寄存器。如果未启用多线程操作体系结构扩展，则将遗留代码的执行限于Shred编号0。

[0162]

初始 EAX 值	有关处理器提供的信息
1H	EAX 版本信息(类型、系列、模型和步进 ID) EBX 位 7-0: 种类索引 位 15-8: CLFLUSH 行大小。(值 8 = 以字节计的高速缓存行大小) 位 23-16: 每个物理处理器的逻辑处理器的数量。 位 31-24: 本地 APIC ID ECX 扩展的特征信息 EDX 特征信息

[0163] 图10

[0164] CPUID

[0165] 将IA-32 CPUID指令修改为返回该处理器支持多线程操作体系结构扩展的指示以及所提供的物理Shred的数量的计数。这通过将三个位(NSHRED)添加到ECX中返回的扩展的特征信息中来实现。下表给出CPUID指令返回的信息：

[0166]

初始 EAX 值	有关处理器提供的信息
1H	EAX 版本信息(类型、系列、模型和步进 ID) EBX 位 7-0: 种类索引 位 15-8: CLFLUSH 行大小。(值 8 = 以字节计的高速缓存行大小) 位 23-16: 每个物理处理器的逻辑处理器的数量。 位 31-24: 本地 APIC ID ECX 扩展的特征信息 EDX 特征信息

[0167] 表11

[0168] ECX寄存器中返回的扩展的特性信息具有如下形式：

[0169]

位 #	助记符	描述
18:16	NSHRED	指示硬件支持的物理 Shred 的数量的三个位： 000: 1 个 Shred/线程 001: 2 个 Shred/线程 010: 4 个 Shred/线程 011: 8 个 Shred/线程 100: 16 个 Shred/线程 101: 32 个 Shred/线程 110: 保留 111: 保留

[0170] 表12

[0171] 如果未启用多线程操作体系结构扩展(通过MAX_SHRED_ENABLE MSR)，则扩展的特性信息对于NSHRED返回000的值。

[0172] 体系结构状态

[0173] 该多线程操作体系结构扩展将所有状态置于三个类别的其中之一中。

[0174] · 对于每个Shred专用的

[0175] · 在本地Shred之间共享的

[0176] · 在所有Shred之间共享的

[0177] 上表2中示出了IA-32状态至每种类别的分类。每个Shred复制Shred的专用状态一次。Shred专用状态对于每个Shred是完全专用的。确切地来说，该体系结构不提供个别地从一个Shred读或写另一个Shred的专用寄存器的任何指令。该体系结构提供shsave和shrestore指令以统一地将所有Shred的专用状态写和读到存储器，但是这些指令仅以单Shred化模式执行。上图3示出Shred的共享状态。

[0178] 将一组共享的寄存器SH0-SH7 655用于Shred之间的通信和同步。通过向共享的寄存器MOV和从共享的寄存器MOV的指令来读和写这些寄存器655。SH0-SH7寄存器655存储32位整数值。根据一个实施例，通过主存储器共享80位浮点625和128位SSE数据640。

[0179] 设有一组Shred控制寄存器SC0-SC4 660。这些寄存器如下定义。

[0180]

寄存器	名称	描述
SC0	Shred运行寄存器	SC0包含每个Shred一个位的位向量。位0对应于Shred0；位1对应于Shred1，依此类推。每个位指示相关联的Shred当前是在运行还是暂停。当通过MAX_SHRED_ENABLE MSR禁用多线程操作体系结构扩展时，SC0包含指示仅Shred0是活动的值1。

SC1	中断Shred运行寄存器	当从多Shred化转换到单Shred化模式时，将SC0的内容复制到SC1，并且当从单Shred化转换到多Shred模式时，将SC1的内容复制到SC0。
SC2	Shred状态保存/恢复指针	SC2指向存储器中的Shred状态保存/恢复区。该存储区用于保存和恢复上下文切换上的所有运行Shred的状态。
SC3	共享的寄存器空/满位	SC3包含共享的寄存器的空/满位。位0对应于sh0；位1对应于sh1，依此类推。
SC4	用户级中断表基地址	SC4指向用户级中断表的基地址。

[0181] 表13

[0182] 存储器状态为所有Shred所共享。下表中示出EFLAGS寄存器615的分类。

[0183]

位	类型	复制	助记符	描述
0	状态	Y	CF	进位标志
2	状态	Y	PF	奇偶校验标志
4	状态	Y	AF	辅助进位标志
6	状态	Y	ZF	零标志
7	状态	Y	SF	符号标志
8	系统	Y	TF	陷阱标志
9	系统	N	IE	中断启用标志
10	控制	Y	DF	方向标志
11	状态	Y	OF	溢出标志
13:12	系统	N	IOPL	IO特权级
14	系统	N	NT	嵌套任务
16	系统	N	RF	恢复标志
17	系统	N	VM	虚拟86模式
18	系统	N	AC	对齐检查
19	系统	N	VIF	虚拟中断标志
20	系统	N	VIP	虚拟中断未决
21	系统	N	ID	ID标志

[0184] 表14

[0185] 以每个Shred为基础来复制标记为“Y”的标志。标记为“N”的标志具有由所有Shred共享的一个副本。

[0186] 32位EFLAGS寄存器615包含一组状态标志、控制标志和一组系统标志。在处理器105的初始化(通过断言RESET引脚或INIT引脚)之后，EFLAGS寄存器615的状态是00000002H。该寄存器615的位1、3、5、15和22至31保留。软件不应使用或依赖于这些位的任何一个的状态。

[0187] 可以使用专用指令直接修改EFLAGS寄存器615中的一些标志。没有指令允许直接检查或修改整个寄存器。但是，如下指令可用于将标志集合移到过程栈或EAX寄存器以及移动来自过程栈或EAX寄存器的标志集合：LAHF、SAHF、PUSHF、PUSHFD、POPF和POPFD。在将EFLAGS寄存器615的内容传送到过程栈或EAX寄存器之后，可以使用处理器的位操纵指令

(BT、BTS、BTR和BTC)来检查和修改这些标志。

[0188] 当挂起任务(使用处理器的多任务处理实用程序)时,处理器自动将EFLAGS寄存器615的状态保存在正在挂起的任务的任务状态段(TSS)。当将自己绑定到新任务时,处理器以来自新任务的TSS的数据加载EFLAGS寄存器615。

[0189] 当对中断或异常处理程序过程发出调用时,处理器自动将EFLAGS寄存器615的状态保存在过程栈上。当利用任务切换来处理中断或异常时,EFLAGS寄存器615的状态被保存在正在挂起的任务的TSS中。

[0190] Shred创建/摧毁

[0191] 可以使用forkshred指令创建Shred。其格式是:

[0192] forkshred imm16, 目标IP

[0193] forkshred r16, 目标IP

[0194] 提供两种形式,具有Shred编号作为立即运算数的一种形式和具有Shred编号作为寄存器运算数的第二种形式。对于这两种形式,指定目标IP作为立即运算数,其值是相对于代码段的开头(标称为0)而不是相对于当前IP的。

[0195] forkshred imm 16, 目标IP编码与far jump指令相似,其中以Shred编号替代16位选择器(selector)并且以目标IP替换16/32位偏移量。

[0196] forkshred指令在SC0中设置适合的运行位,并在指定的地址开始执行。不同于Unix fork()系统调用,forkshred指令不复制父Shred的状态。新的Shred利用更新的EIP以及所有其他专用寄存器的当前值开始执行。可预期的是,新的Shred应通过加载ESP来初始化它的栈,并从共享的寄存器或存储器中检索输入参数。forkshred指令不自动传递参数。

[0197] 如果目标Shred已经在运行,则forkshred产生#SNA(Shred不可用)异常。这是一个用户级异常,如下文所述。软件确保它不尝试启动已经运行的Shred,或者提供暂停现有Shred并返回以重新执行forkshred的#SNA处理程序。如果Shred编号大于硬件支持的Shred的最大数量,则产生#GP(0)异常。

[0198] 为了终止当前Shred的执行,使用haltshred指令。haltshred清除SC0中的当前Shred的运行位并终止当前Shred的执行。Shred的专用状态即使被暂停时仍被保存。因为不存在任何机制可供一个Shred用来访问另一个Shred的专用状态,所以无法看到暂停的Shred的专用状态。但是,该状态一直存在,并在Shred通过forkshred又开始执行时变得可被看到。

[0199] 为了提前终止另一个Shred的执行,引入killshred指令。其格式为:

[0200] killshred imm16

[0201] killshred r16

[0202] 根据一个实施例,Shred编号是16位寄存器或立即运算数。killshred清除SC0中的指定Shred的运行位并终止该Shred的执行。虽然暂停,但是该Shred的专用状态被保存。

[0203] 如果目标Shred未在运行,则killshred无响应地被忽略。该行为对于避免killshred与正常终止的Shred之间的竞争(race)是必要的。在执行killshred之后,软件得以保证目标Shred不再运行。允许Shred自己杀死自己,而代替执行haltshred。如果Shred编号大于硬件支持的Shred的最大数量,则产生#GP(0)异常。

[0204] 为了等待直到指定的Shred已经终止(如被清除的SC0位所指示),引入joinshred

指令。其格式为：

[0205] joinshred imml6

[0206] joinshred r16

[0207] 如果目标Shred未在运行，则joinshred立即返回。该行为避免了joinshred与正常终止的Shred之间的竞争。在执行joinshred之后，软件得以保证该目标Shred不再运行。对于Shred来说对自己执行joinshred是合法的(但是无用的)。如果Shred编号大于硬件支持的Shred的最大数量，则产生#GP(0)异常。joinshred指令不自动传递返回值。为了允许Shred确定它自己的Shred编号，引入getshred指令。其格式为：

[0208] getshred r32

[0209] getshred返回当前Shred的编号。getshred可以用于访问根据Shred编号索引的存储器阵列。getshred对16位Shred编号执行左侧补零以写入目的地寄存器的所有位。

[0210] 对于所有Shred创建/摧毁指令，可以将Shred编号指定为寄存器或立即运算数。可预期的是，立即形式的执行会比寄存器形式的执行快，因为Shred编号在解码时间而非执行时间是可用的。利用立即形式，编译器指定Shred编号。利用寄存器形式，使用运行时指定。

[0211] 下表给出Shred创建/摧毁指令的汇总。

[0212]

指令	描述
forkshred imml6, 目标 IP forkshred r16, 目标 IP	在指定的地址开始 Shred 执行
haltshred	终止当前 Shred
killshred imml6 killshred r16	终止指定的 Shred
joinshred imml6 joinshred r16	等待直到指定的 Shred 终止
getshred r32	返回当前 Shred 的编号

[0213] 表15

[0214] forkshred、haltshred、killshred、joinshred和getshred指令可以在任何特权级上执行。haltshred是非特权指令，而现有的IA-32 hlt指令是特权指令。

[0215] 执行killshred或haltshred导致零个运行Shred是可能的。该状态(SC0中为0)不同于现有的IA-32暂停状态。在创建用户级定时器中断之前，SC0=0是合法状态，但是无用。

[0216] 通信

[0217] Shred通过现有共享的存储器并通过为此目的专门引入的一组寄存器来彼此通信。共享的寄存器SH0-SH7 655可被属于相同线程的所有本地Shred访问。SH0-SH7寄存器655可以用于将输入参数传递到Shred，传送来自Shred的返回值，并且执行信号量操作。软件约定针对每个目的指定特定共享的寄存器655。

[0218] 每个共享的寄存器655在SC3中具有对应的空/满位。为了写和读共享的寄存器655，使用向共享的寄存器655 MOV和从共享的寄存器655 MOV的指令。这些指令汇总如下：

[0219] mov r32, sh0-sh7

[0220] mov sh0-sh7, r32

[0221] 这些指令编码与现有的向控制寄存器660 MOV/从控制寄存器660 MOV和向调试寄存器MOV/从调试寄存器MOV的指令相似。向共享的寄存器MOV/从共享的寄存器MOV的指令可以在任何特权级上执行。这些指令假定软件使用附加指令明确地执行同步。向共享的寄存器MOV/从共享的寄存器MOV的指令既不检查也不修改SC3中的空/满位的状态。

[0222] 可预期的是，向共享的寄存器655 MOV和从共享的寄存器655 MOV的等待时间将低于对共享的存储器加载和存储的等待时间。硬件实施可能推测性地读共享的寄存器655并监听其他Shred写。硬件必须在对共享的寄存器655写时确保等效的强排序。在备选实施例中，可以创建屏障指令来访问共享的寄存器655。

[0223] 一个体系结构特征保持共享的寄存器排序和存储器排序彼此分开。因此，如果Shred对共享的寄存器655写，然后对存储器120写，则不能保证共享的寄存器655的内容会在共享的存储器的内容之前可被看到。这种定义的原因是为了在共享的寄存器655中启用循环计数器的高速访问/更新，而不创建不需要的存储器屏障。如果软件要求共享的寄存器655和存储器上的屏障，则软件应该执行共享的寄存器信号量以及存储器信号量。除了作为屏障外，存储器信号量是冗余的。

[0224] 为了提供快速通信以及同步，使用向共享的寄存器同步mov/从共享的寄存器同步mov的指令。这些指令汇总如下：

[0225] syncmov r32, sh0-sh7

[0226] syncmov sh0-sh7, r32

[0227] 这些指令编码与现有的向控制寄存器660 MOV/从控制寄存器660 MOV和向调试寄存器MOV/从调试寄存器MOV的指令并行。向共享的寄存器655同步mov与它的对应的异步指令相似，只是它等到空/满位指示空之后，才对共享的寄存器655写入。在对共享的寄存器655写入之后，空/满位被设为满。从共享的寄存器655同步mov与它的对应的异步指令相似，只是它等到空/满位指示满之后，才从共享的寄存器655读取。在从共享的寄存器655读取之后，空/满位被清除为空。

[0228] 可以利用向SC3移动来初始化空/满位，如下文所述。向共享的寄存器同步MOV/从共享的寄存器同步MOV的指令可以在任何特权级上执行。这些共享的寄存器通信指令汇总如下：

[0229]

指令	描述
mov r32, sh0-sh7	从共享的寄存器移动。
mov sh0-sh7, r32	向共享的寄存器移动
syncmov r32, sh0-sh7	从共享的寄存器同步移动
syncmov sh0-sh7, r32	向共享的寄存器同步移动

[0230] 表16

[0231] 同步

[0232] 一组同步原语对共享的寄存器655操作。同步原语与现有的信号量指令相似，只是它们对共享的寄存器655操作，而非对存储器操作。这些指令如下所示：

[0233]

指令	描述
cmpxchgsh sh0-sh7, r32	将共享的寄存器与r32比较。如果相等，则设置ZF并将r32加载到共享的寄存器中。否则，清除ZF，并将共享的寄存器加载到EAX中。
xaddsh sh0-sh7, r32	将共享的寄存器与r32交换。然后将r32加到共享的寄存器。该指令可以与LOCK前缀一起使用来启用原子操作。
xchgsh sh0-sh7, r32	将共享的寄存器与r32交换。该指令始终是原子的。

[0234] 表17

[0235] 同步原语在任何特权级上执行。这些指令既不检查也不修改SC3中的空/满位的状态。.

[0236] 进入/退出多Shred化模式

[0237] MAX体系结构提供一种机制以在多Shred化和单Shred化模式之间切换。单Shred化模式使处理器能够通过暂停除一个Shred外所有Shred的执行来以有序方式执行上下文切换。SC0指示当前操作模式，如下所示：

[0238] · SC0在任何位位置全包含单个“1”意味着单Shred化模式；

[0239] · SC0在任何位位置全包含不同于单个“1”的任何值意味着多Shred化模式。

[0240] 为了执行上下文切换，需要：

[0241] 1）通过切换到单Shred化模式以挂起除一个Shred外的所有Shred；

[0242] 2）保存Shred的状态；

[0243] 3）加载新Shred的状态；

[0244] 4）通过切换到多Shred化模式以恢复所有Shred的执行。

[0245] entermsm和exitmsm指令分别用于切换到多Shred化模式和单Shred化模式。entermsm用于进入多Shred化模式。在执行该指令之前，必须加载所有Shred的状态。entermsm将SC1中的新Shred运行向量复制到SC0中。然后，entermsm启动指定的Shred。

[0246] SC1的内容导致在执行entermsm之后没有附加Shred在运行是可能的。在此情况中，处理器保持在单Shred化模式。由于执行entermsm，执行了entermsm的Shred不再运行也是可能的。exitmsm用于退出多Shred化模式。exitmsm将SC0中的当前Shred执行向量复制到SC1中。除对应于执行exitmsm的Shred的一个位之外的所有SC0运行位被清除。除执行exitmsm的Shred之外的所有Shred被暂停。这些操作作为原子序列执行。SC0状态指示单Shred化模式。entermsm和exitmsm可以在任何特权级上执行。

[0247] 状态管理

[0248] 指令(shsave和shrestore)分别用于保存和恢复集体Shred状态，将所有Shred专用状态的内容写入到存储器，并且从存储器读取所有Shred专用状态的内容。其格式为：

[0249] shsave m16384

[0250] shrestore m16384

[0251] 存储器保存区的地址被指定为指令中的位移。地址在16字节边界上对齐。存储器保存区是16千字节，以便允许将来扩展。存储器保存区通过添加整数寄存器来扩展现有的FXSAVE/FXRESTOR格式。每个Shred的存储器保存区定义如下：

[0252]

偏移量	寄存器

0-1	FCW
2-3	FSW
4-5	FTW
6-7	FOP
8-11	FIP
12-13	CS
14-15	保留
16-19	FPU DP
20-21	DS
22-23	保留
24-27	MXCSR
28-31	MXCSR_MASK
32-159	ST0-ST7
160-287	XMM0-XMM7
288-351	EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP
352-359	ES、FS、GS、SS
360-367	EIP
368-371	EFLAGS

[0253] 表18

[0254] 所有Shred的内容在如下计算的地址处保存/恢复：

[0255] 地址 = 512 *（Shred编号）+（基地址）

[0256] 存储器保存区包括当前运行的Shred的EIP和ESP。shsave将当前EIP和ESP写入存储器。为了避免分支，shrestore指令不覆写当前Shred的EIP或ESP。当作为IRET的一部分执行时，shrestore函数不覆写当前Shred的EIP和ESP。

[0257] shsave和shrestore可以在任何特权级上但是仅在单Shred化模式中时执行。如果当在多Shred化模式中时试图执行shsave或shrestore，则产生#GP(0)异常。使用所有可用硬件资源来并行地执行shsave/shrestore存储/加载操作，实施起来是不受限制的。

[0258] shrestore无条件地从存储器加载所有Shred的状态。该行为对于确保Shred的专用状态不会从一个任务泄漏到下一个任务来说是必要的。shsave可以无条件或有条件地将所有Shred的状态存储到存储器。一个实施可以维护非体系结构可见的修改位，以在未修改专用状态的情况下跳过一些或所有shsave存储操作。

[0259] shsave和shrestore指令仅保存并恢复Shred的专用状态。操作系统负责保存并恢复共享的寄存器655。

[0260] 向Shred控制寄存器660移动/从Shred控制寄存器660移动

[0261] 提供指令以写和读Shred控制寄存器SC0-SC4 660。这些指令汇总如下：

[0262] mov r32, sc0-sc4

[0263] mov sc0-sc4, r32

[0264] 这些指令编码与现有的向控制寄存器660 MOV/从控制寄存器660 MOV和向调试寄存器MOV/从调试寄存器MOV的指令相似。向Shred控制寄存器MOV/从Shred控制寄存器MOV的

指令可以在任何特权级上执行。提供安全防护以确保恶意应用程序无法通过对Shred控制寄存器写入来影响除了它自己以外的任何进程。

[0265] 应用程序使用forkshred和joinshred，而非直接操纵SC0的内容。exitmsm可采用原子方式从多Shred化模式转换到单Shred化模式。使用从SC0 mov以读当前Shred运行状态，然后使用向SC0 mov以写Shred运行状态，将不会给出期望的结果，因为Shred运行状态可以在读和写之间更改。

[0266] 操作系统异常

[0267] MAX具有用于IA-32异常机制的若干蕴涵。首先，用户级异常机制允许将多种类型的异常直接报告给产生它们的Shred。下文描述该机制。

[0268] 其次，该IA-32异常机制修改为在需要上下文切换的异常存在的情况下妥当地处理多个Shred。现有技术的IA-32异常机制的一个问题在于，它被定义为自动保存并恢复用于正好一个运行线程的CS、EIP、SS、ESP和EFLAGS。

[0269] 现有IA-32异常机制被扩展为包括entermsm、exitmsm、shsave和shrestore指令的功能。当产生需要上下文切换的中断或异常时，该异常机制执行如下操作：

[0270] 1）执行exitmsm以退出多Shred化模式。exitmsm暂停除导致该中断或异常的Shred之外的所有Shred。使用导致中断或异常的Shred进入操作系统；

[0271] 2）通过在SC2给出的开始地址执行shsave来将所有Shred的当前状态保存到存储器；

[0272] 3）按目前所定义的执行IA-32上下文切换。

[0273] 为了返回到多Shred化程序，修改后的IRET指令执行如下操作：

[0274] 1）按目前所定义的执行IA-32上下文切换；

[0275] 2）通过在SC2给出的开始地址执行shrestore以从存储器恢复所有Shred的当前状态。这会覆写IA-32上下文切换中保存的EIP和ESP；

[0276] 3）执行entermsm以进入多Shred化模式。具体根据SC1的状态，entermsm的执行可能导致处理器保持在单Shred化模式中。

[0277] 在执行IRET之前，需要操作系统在存储器中建立Shred状态保存/恢复区并将它的地址加载到SC2中。还要求操作系统保存/恢复SC1、SC3和SC4的状态。

[0278] 多个Shred同时遇到需要操作系统服务的异常是可能的。因为MAX体系结构一次仅能够报告一个OS异常，所以硬件必须在多个Shred中设置OS异常的优先级，刚好只报告一个，并将所有其他Shred的状态设置为产生异常的指令尚未被执行的点。

[0279] 用户级异常

[0280] MAX引入用户级异常机制，它允许某些类型的异常完全在应用程序内处理。无需操作系统介入、特权级转换或上下文切换。

[0281] 当用户级异常发生时，将下一个要执行的指令的EIP推送到栈上，并且处理器引向到指定的处理程序。用户级异常处理程序执行它的任务，然后通过现有RET指令返回。根据一个实施例，未提供任何机制来屏蔽用户级异常，因为假定应用仅在该应用准备服务用户级异常时才会产生它们。

[0282] 提供两个指令以创建前两个用户级异常：signalshred和forkshred。将在下文段落中描述它们。

[0283] 发信号

[0284] signalshred指令用于向指定的Shred发送信号。其格式为：

[0285] signalshred imm16, 目标IP

[0286] signalshred r16, 目标IP

[0287] 可以将目标Shred指定为寄存器或立即运算数。signalshred imm16, 目标IP指令编码与现有的far jump指令相似，其中以Shred编号替代16位选择器，并且以目标IP替换16/32位偏移量。与far jump一样，相对于代码段的开始位置(标称为0)而不是相对于当前IP指定signalshred目标IP。

[0288] 响应signalshred，目标Shred将下一个要执行的指令的EIP推送到栈上并引向到指定的地址。Shred可以向自己发送信号，在此情况中，效果与执行near call指令相同。如果目标Shred未在运行，则signalshred无响应地被忽略。如果Shred编号大于硬件支持的Shred的最大数量，则产生#GP(0)异常。

[0289] signalshred指令可以在任何特权级上执行。signalshred指令不能自动向目标Shred传递参数。未提供任何机制来阻止signalshred。因此，软件可能需要在发出signalshred之前实施阻止机制或提供可以嵌套的signalshred处理程序。

[0290] Shred不可用(SNA)

[0291] 如果程序试图启动已经在运行的Shred，则forkshred产生#SNA异常。软件#SNA处理程序可以对现有Shred执行killshred，并返回到forkshred指令。

[0292] 通过将forkshred指令的EIP推送到栈上并引向到SC4+0给出的地址来处理#SNA异常。SC4+0处的代码应该分支到实际的处理程序。异常向量被置于SC4+16、SC4+32等处。软件保留存储器最多SC4+4095以涵盖256种可能的用户级异常。在后续时间，将存储器/SC4机制中的中断表替换为较清洁的机制。

[0293] 挂起/恢复和Shred虚拟化

[0294] 多线程操作体系结构扩展允许用户级软件使用如下指令来挂起或恢复Shred。为了挂起Shred：

[0295] 1) 初始化存储器中的Shred状态保存区。这是应用程序为挂起操作设置的存储区。它不同于指向SC2的上下文切换Shred状态区。

[0296] 2) 向指向挂起处理程序的Shred发送信号。这通过如下指令完成：signalshred 目标Shred, 挂起处理程序IP

[0297] 3) 挂起处理程序使用现有的mov、pusha和fxsave指令将该Shred的专用状态保存到存储器。

[0298] 4) 挂起处理程序执行haltshred。

[0299] 5) 原代码执行joinshred来等待直到Shred已经暂停。

[0300] Shred可能在挂起操作时已经暂停。在此情况中，signalshred被忽略，从不调用挂起处理程序，并且joinshred不等待。存储器中的Shred状态保存区留存它的初始值，它必须指向立即执行haltshred的伪Shred。为了恢复Shred，执行逆向操作：

[0301] 1) 对指向恢复处理程序的Shred执行fork。这通过如下指令完成：forkshred 目标Shred, 恢复处理程序IP

[0302] 2) 恢复处理程序使用现有的mov、popa和fxrestor 指令从存储器恢复该Shred的

专用状态;以及

[0303] 3) 恢复处理程序通过现有RET指令返回到该Shred。

[0304] 当恢复到已经暂停的线程时,恢复处理程序会RET到立即执行haltshred的伪Shred。挂起/恢复能力开启了Shred虚拟化的可能性。在执行forkshred之前,软件可以选择挂起具有相同Shred编号的任何现有Shred。在执行joinshred之后,软件可以选择恢复具有相同Shred编号的任何现有Shred。因为挂起/恢复顺序不是可重入的,所以软件临界区对于确保在任何给定时间对任何给定Shred仅执行一次挂起/恢复是必要的。使用这些机制,应用程序创建它自己的抢先Shred调度程序是可能的。

[0305] 在MAX的备选实施例中,存在一个指令使用第一可用Shred来执行fork(allocforkshred r32),其中利用分配的Shred编号写r32(在forkshred中,r32指定要执行fork的Shred编号)。allocforkshred还返回标志,以指示是否有任何可用的硬件Shred。

[0306] 在另一个实施例中,等待Shred指令提供等待使用共享的寄存器的同步(waitshred sh0-sh7, imm)。该等待指令提供作为指令的等待功能。在没有该指令的情况下,必须使用循环,如:

[0307] loop: mov eax, sh0

[0308] and eax, mask

[0309] jz loop

[0310] 在另一个实施例中,对joinshred提供位掩码以便对多个Shred等待。没有位掩码,一个joinshred等待一个Shred终止。需要多个joinshred才可对多个Shred等待。

[0311] 在备选实施例中,不使用killshred。可以使用signalshred然后使用joinshred来替代killshred。signalshred处理程序由haltshred指令组成。

[0312] 在再一个实施例中,将forkshred与signalshred组合是可能的。forkshred和signalshred唯一不同之处在于有关Shred当前是在运行还是暂停的行为上。如果允许signalshred启动暂停的Shred,则signalshred可以潜在地替代forkshred。

[0313] 图7图示根据本发明一个实施例的用户级多线程操作的示范过程的流程图。假定应用程序或软件程序已启动如下过程。如下过程并非是结合任何特定程序来描述的,而是作为通过上述指令和体系结构来实现的用户级多线程操作的一个实施例来描述的。此外,如下过程结合微处理器的ISA来执行,如16、32、64、128或更高位体系结构的多处理器。多处理器(如处理器105)初始化如上表3的寄存器的共享的寄存器中的值(处理框705)。处理器105执行创建Shred的forkshred指令(处理框710)。通过处理器105执行并行操作。由处理器105执行主(父)Shred(处理框715)。执行joinshred操作以等待新目标Shred完成执行(处理框730)。同时,新目标Shred初始化它的栈,从共享的寄存器和/或存储器中检索输入参数(处理框720)并执行(处理框721)。使用haltshred指令终止当前目标Shred的执行(处理框723)。处理器105将执行结果从存储Shred的执行结果的寄存器返回给程序或应用(处理框735)。一旦返回所有执行的数据,则过程完成(终止框799)。

[0314] 公开一种用于提供用户级多线程操作的方法和系统。虽然本发明的实施例是参考特定示例和子系统来描述的,但是对于本领域技术人员来说,显然本发明的实施例并不限于这些特定的示例或子系统,而是还延伸到其他实施例。本发明的实施例包括符合所附权利要求中所指定的所有这些其他实施例。

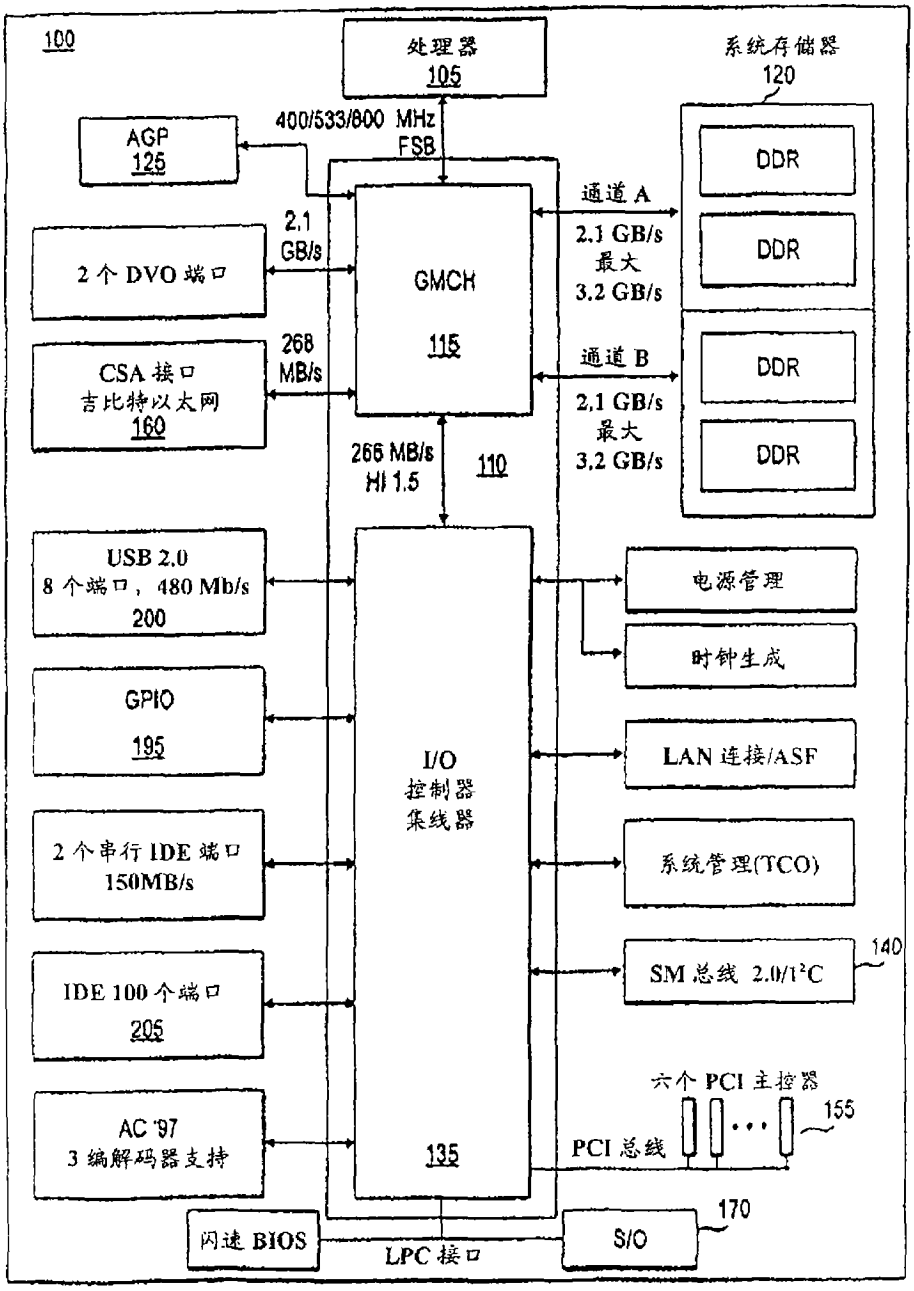

100
处理器
105
系统存储器
120
AGP
125
400/533/800 MHz
FSB
DDR
DDR
通道 A
2.1 GB/s
最大
3.2 GB/s
2.1
GB/s
2 个 DVO 端口
GMCH
115
268
MB/s
CSA 接口
吉比特以太网
160
通道 B
2.1 GB/s
最大
3.2 GB/s
DDR
DDR
266 MB/s
HI 1.5
110
USB 2.0
8 个端口，480 Mb/s
200
电源管理
时钟生成
GPIO
195
I/O
控制器
集线器
LAN 连接/ASF
2 个串行 IDE 端口
150MB/s
系统管理(TCO)
140
SM 总线 2.0/I²C
IDE 100 个端口
205
六个 PCI 主控器
155
AC '97
3 编解码器支持
135
PCI 总线
170
闪速 BIOS
LPC 接口
S/O

图1

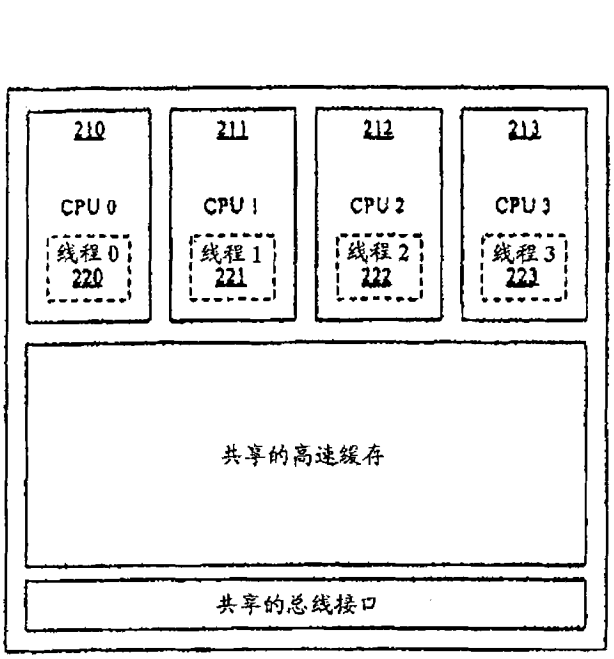

200
210
211
212
213
CPU 0
CPU 1
CPU 2
CPU 3
线程 0
220
线程 1
221
线程 2
222
线程 3
223
共享的高速缓存
共享的总线接口

图2

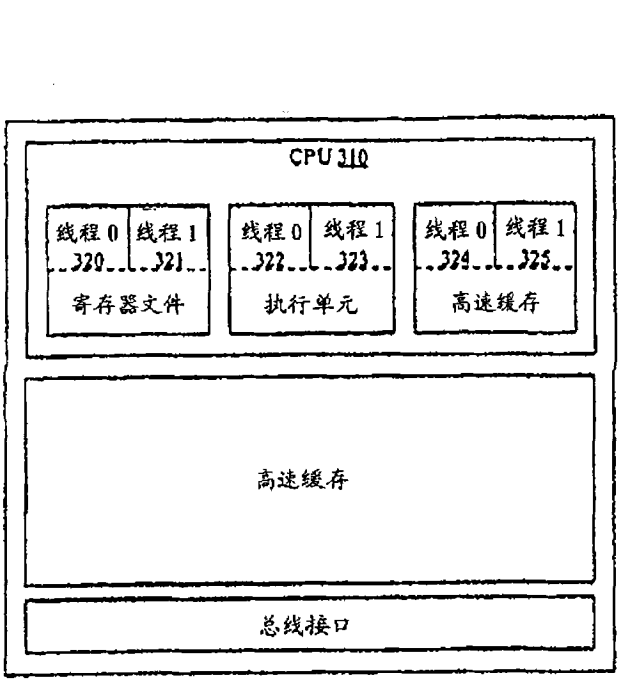

300
CPU 310
线程 0
320
线程 1
321
寄存器文件
线程 0
322
线程 1
323
执行单元
线程 0
324
线程 1
325
高速缓存
高速缓存
总线接口

图3

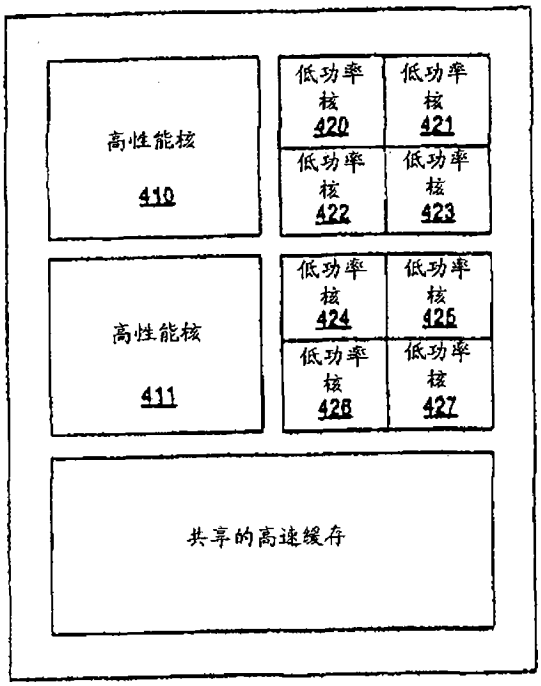

400
高性能核
410
低功率核
420
低功率核
421
低功率核
422
低功率核
423
高性能核
411
低功率核
424
低功率核
425
低功率核
426
低功率核
427
共享的高速缓存

图4

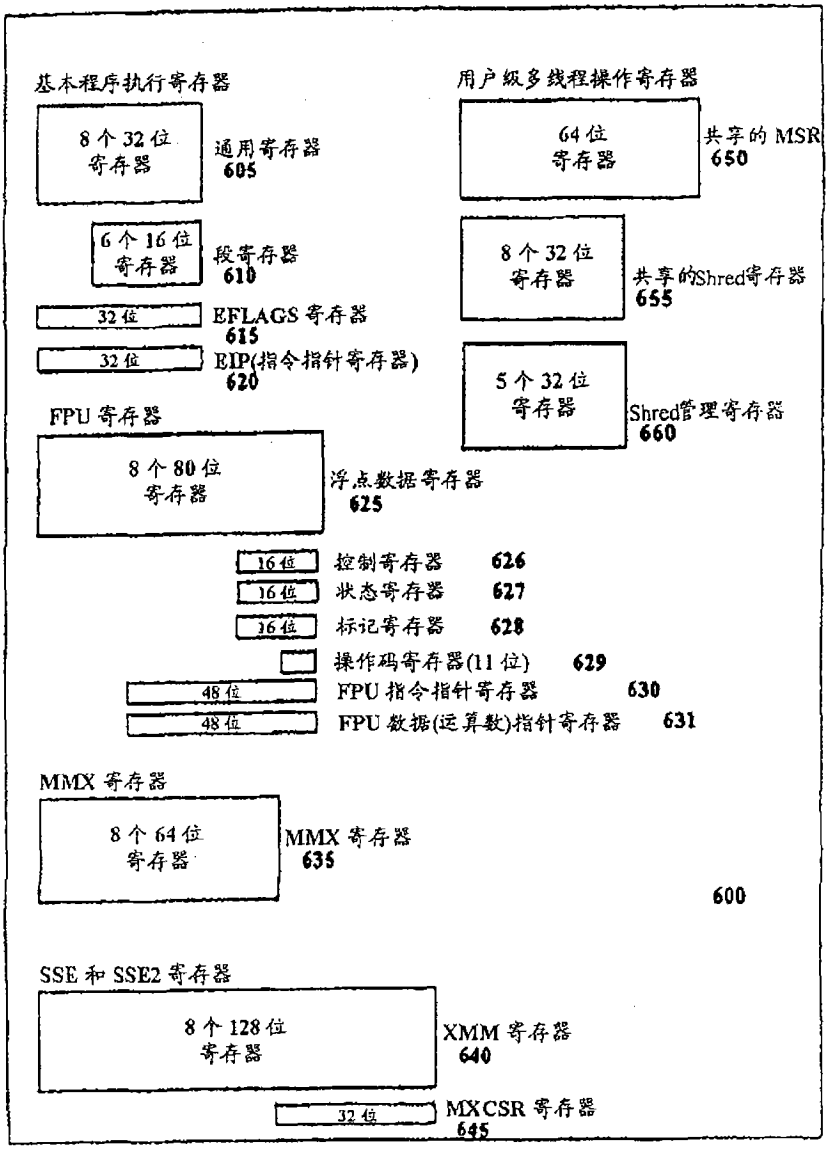

基本程序执行寄存器
8 个 32 位
寄存器
通用寄存器
605
6 个 16 位
寄存器
段寄存器
610
32 位
EFLAGS 寄存器
615
32 位
EIP(指令指针寄存器)
620
FPU 寄存器
8 个 80 位
寄存器
浮点数据寄存器
625
16 位
控制寄存器 626
16 位
状态寄存器 627
16 位
标记寄存器 628
操作码寄存器(11 位) 629
48 位
FPU 指令指针寄存器 630
48 位
FPU 数据(运算数)指针寄存器 631
MMX 寄存器
8 个 64 位
寄存器
MMX 寄存器
635
600
SSE 和 SSE2 寄存器
8 个 128 位
寄存器
XMM 寄存器
640
32 位
MXCSR 寄存器
645
用户级多线程操作寄存器
64 位
寄存器
共享的 MSR
650
8 个 32 位
寄存器
共享的Shred寄存器
655
5 个 32 位
寄存器
Shred管理寄存器
660

图5

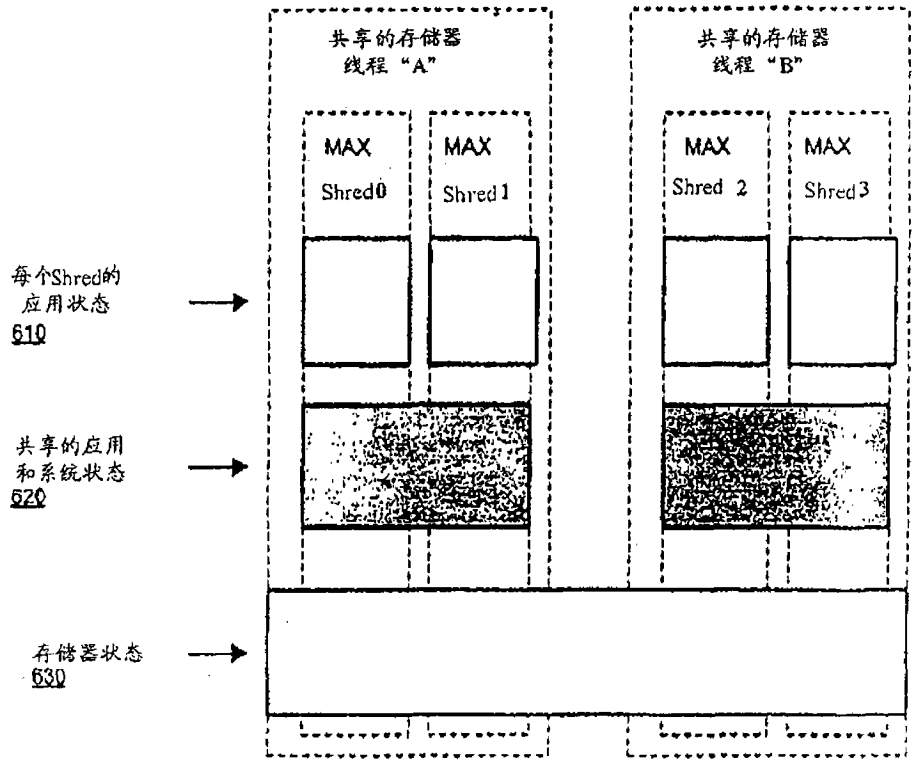

共享的存储器
线程 "A"
共享的存储器
线程 "B"
MAX
Shred0
MAX
Shred1
MAX
Shred 2
MAX
Shred3
每个Shred的
应用状态
610
共享的应用
和系统状态
620
存储器状态
630

图6

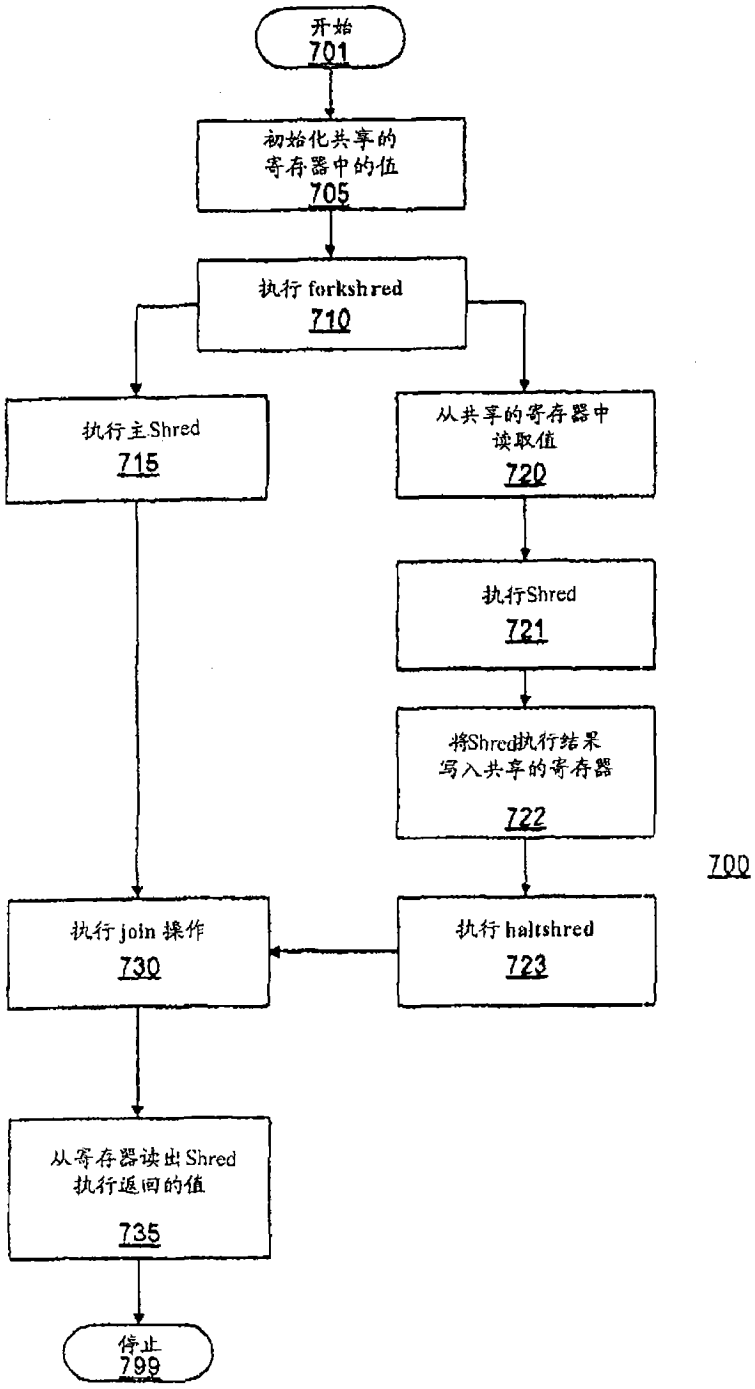

开始
701
初始化共享的
寄存器中的值
705
执行 forkshred
710
执行主Shred
715
从共享的寄存器中
读取值
720
执行Shred
721
将Shred执行结果
写入共享的寄存器
722
700
执行 join 操作
730
执行 haltshred
723
从寄存器读出Shred
执行返回的值
735
停止
799

图7