[19] 中华人民共和国国家知识产权局

[51] Int. Cl.
G06F 9/38 (2006.01)
G06F 9/46 (2006.01)

[12] 发明专利说明书

专利号 ZL 200410018198.2

[45] 授权公告日 2006 年 9 月 6 日

[11] 授权公告号 CN 1273890C

[22] 申请日 2004.4.30
[21] 申请号 200410018198.2
[71] 专利权人 浙江大学
地址 310027 浙江省杭州市西湖区浙大路 38 号
[72] 发明人 陈文智 谢 铖
审查员 郑宗玉

[74] 专利代理机构 杭州求是专利事务所有限公司
代理人 张法高
权利要求书 4 页 说明书 17 页

[54] 发明名称

ARM 处理器架构的微内核设计方法

[57] 摘要

本发明公开了一种 ARM 处理器架构的微内核设计方法。 涉及一个针对 ARM 架构设计的完全可抢占的微内核。 它用于支持下一代面向服务的嵌入式操作系统。 该内核支持软件模块以服务线程的形式构造系统，服务器与客户以及服务器内部使用相同的线程间通信技术，线程间通信只使用 ARM 通用寄存器组发送和接收数据，动态更改通信结构则实现服务动态替换、参数化或者分层叠加。 内核执行并管理这些服务器进程，使之高效的协同工作。 内核支持在线程间通信中共享地址空间，用于服务器进程间传输更多数据。 内核使用自卷上下文结构实现完全可抢占性，支持硬实时中断响应。

1、一种 ARM 处理器架构的微内核设计方法，其特征在于方法的步骤为：

a)将用于构造面向服务的嵌入式操作系统中所有的服务用线程来表示，系统调用都是以线程为操作对象，将服务的地址空间作为线程附带的一部分状态进行管理；

b)用线程间通信方式实现多服务器协作，通信数据用寄存器组进行传递；

c)在线程间通信接口中支持地址空间的传递，用于不同地址空间的服务协作中大量的数据通信；

d)将微内核中无法确定计算量的大任务划分为多个基本的内核指令，进行流水作业；

e)对关键区采用非阻塞式同步访问机制，实现完全可抢占性。

2、根据权利要求 1 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说将用于构造面向服务的嵌入式操作系统中所有的服务用线程来表示：服务用一个 32 位无符号数标示，具有单独的寄存器组，包括指令执行地址、堆栈地址、处理器状态寄存器，创建新服务必须指定合作服务、空间服务及监控服务。

3、根据权利要求 2 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说寄存器组，其中指令地址寄存器和堆栈地址寄存器还用于管理服务的运行、暂停和删除，当服务最初创建时，并不指定指令地址寄存器和堆栈地址寄存器，因此它不处于活动状态，不能被调度执行，当指定了合理的指令地址寄存器和堆栈地址寄存器值后，服务进入活动状态，可以被调度执行，指令地址寄存器值为特殊的 0 值时，则微内核暂停该线程的运行，如果设置指令地址寄存器为 1，则微内核继续运行该服务，在暂停的情况下设置合理的指令地址寄存器值可以使暂停的服务从新的指令地址继续运行，堆栈地址寄存器值为特殊的 0 值时，微内核删除该服务，通过指定一个现有的服务作为合作服务以共享同一个地址空间，否则微内核为新服务创建独立的地址空间，通过指定监控服务和空间服务，由这两个服务控制其运行、暂停或者删除，如果服务的标示号最高位为 0 表明系统服务，如果最高位为 1 表明普通用户服务，服务标示号由创建者设定，创建服务的能力完全由系统服务控制，普通用户服务通过向系统服务发送请求间接创建新服务，所述的空间服务还负责内存分配，当一个服务向系统请求内存管理时，内存请求由对应的空间服务来实现，微内核根据优先

级进行服务调度，缺省情况下，内核上将服务的优先级自高到低分为紧急级、实时级、正常级、后台级四层，总共定义 15 个优先级，最高级为 14，最低级为 0，其中紧急级为 12，实时级为 8，正常级为 5，后台级为 2，其中优先级为 0 的服务是一个比较特殊的空闲服务，它通常不参与调度过程，只有当所有其他服务都不处于活动状态的时候，微内核才会运行这个服务，该服务管理系统的能耗，当持续运行一段时间后，则关闭外围器件并降低处理器主频。

4、根据权利要求 1 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说用线程间通信方式实现多服务器协作：微内核基于通信顺序过程处理模式，以线程间通信方式实现相同或不同地址空间的服务之间的数据通信，从而实现服务接口的调用，任何一个线程间通信都分为发送阶段和接收阶段，如果一个线程间通信只发送数据，那么其接收阶段为空，同样如果一个线程间通信只接收数据，那么其发送阶段为空。

5、根据权利要求 4 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说数据通信：数据通信用寄存器组进行传递，发送阶段和接收阶段只使用通用寄存器组，服务需要发送/接收的数据完全通过通用寄存器组来传递，微内核完成一次线程间通信的数据传递，只需要将数据从发送服务的寄存器组拷贝到接收服务的寄存器组，其发送阶段必须指明接收服务；而线程间通信的接收阶段则不一定需要指明从哪个服务接收数据，分为三种模式，一是接收来自任何服务的数据，二是接收来自同地址空间的服务的数据，第三种情况是针对的多服务器架构设计的客户/服务器模式，当服务 A 调用服务 B 时，服务 A 充当客户角色，服务 B 充当服务器角色。服务 A 进入发送阶段，当服务 B 接收数据之后，服务 A 直接进入接收阶段的阻塞状态，直到再次收到来自服务 B 的数据，在此过程中，任何其他服务如果向服务 A 发送数据，其发送阶段都不能执行，必须等到客户/服务器模式完成为止，服务通常采用阻塞式调用策略，当服务 A 对服务 B 发送数据时，如果服务 B 并没有处于接收阶段，或者虽然处于接收阶段，但是服务 A 并不在其接收范围内，那么服务 A 将成为阻塞状态，直到服务 B 接收数据为止，服务也可采用非阻塞式调用策略，如果发送/接收阶段不能立即执行，服务将得到一个错误值，而不会进入阻塞状态，微内核还提供一种特殊形式的“抢占式”线程间通信，当服务处于发送/接收阶段的阻塞状态时，其它服务可以通过向其发送“抢占式”线程间通信，打断阻塞状态，此时服务得到的结果有两种：第一种情况是服务原先处于发送阶段的阻塞状态，那么服务将得到一个错误返回值，下一次调用只有接收阶段的线程间通信时，将得到这

个“抢占式”线程间通信所附带的数据，第二种情况是服务原先处于接收阶段的阻塞状态，那么服务将得到另一个错误返回值，以及这个“抢占式”线程间通信所附带的数据。

6、根据权利要求 1 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说在线程间通信接口中支持地址空间的传递，用于不同地址空间的服务协作中大量的数据通信：传递地址空间是线程间通信的一种增强模式，当服务 A 向服务 B 发送较多数据时，以内存块的形式调用，一个参数是该内存块的地址 Address_Src，另一个参数则是该内存块的字节大小，微内核把服务 A 所在的地址空间中 Address_Src 对应的物理内存区域直接映射到服务 B 的地址空间中，服务 B 在调用线程间通信接收数据时，由微内核指定读取数据的内存地址 Address_Dst，该值指向实际数据块所在的物理内存地址，在线程间通信进行期间，服务 B 可以直接访问映射的内存区域，当服务 B 再次向服务 A 发送数据时，这块内存区域归还给服务 A，此后服务 B 不能再访问该内存区域，上述的增强模式必须在“客户/服务器”模式的线程间通信上才能实现。

7、根据权利要求 1 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说将微内核中无法确定计算量的大任务划分为多个基本的内核指令，进行流水作业：流水线结构中的每个任务提供的功能是由若干条基本的内核指令组合完成，大多数简单的任务只需要一条内核指令即可完成，而复杂的任务分解为依次执行的多条基本的内核指令，所有基本的内核指令都必须在固定的短时间内完成，需要连续执行的一系列内核指令则放入一条流水线中，定时器中断从流水线取得内核指令并执行。

8、根据权利要求 7 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说需要连续执行的一系列内核指令则放入一条流水线中：一条内核指令在执行期间，可以产生更多的内核指令，它们都被放入流水线中，并由定时器中断依次执行内核指令，当一个任务的最后一条 内核指令完成之后，把结果写回服务的寄存器组，服务就可以继续执行下去，内核指令用于分解复杂的线程间通信任务，一组服务$\{t_1,t_2,...,t_{n-1},t_n\}$，其中$t_1$不处于线程间通信接收阶段，对于$2\le i<n$，服务$t_i$调用的线程间通信包括发送和接收阶段，向服务$t_{i-1}$发送数据，然后接收任何服务的数据，服务$t_n$只有线程间通信发送阶段，此时$\{t_2,t_3,...,t_{n-1},t_n\}$服务都处于阻塞的发送阶段，在某时刻微内核调度$t_1$执行，当$t_1$调用线程间通信接收$t_2$的数据后，微内核应将$t_2$转入线程间通信接收阶段，由于$t_3$正处于线程间通信发送阶段，因此服务$t_2$立即接收$t_3$的数据，并进入活动状态，以此类推，$\{t_2,t_3,...,t_{n-1},t_n\}$服务都将依次进入活动状态，通过内核指令的分解，微内核调度服

务t_i进入接收阶段，将立即接收t_{i+1}的数据，但是t_{i+1}不会立即进入接收阶段，微内核发射一条“内核指令”，该指令的功能是令t_{i+1}服务进入接收阶段，在下一个“内核指令”的时钟周期到来时，微内核将执行这条“内核指令”，并且在执行完毕的时候，又发射新的一条指令来令t_{i+2}服务进入接收阶段。

9、根据权利要求 1 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说按照在前述步骤中确定的关键区采用非阻塞式同步访问机制，实现完全可抢占性：为了让中断响应抢占内核任务，将关键区改造为非阻塞式同步，任何一段关键区内的任务都自行管理上下文环境，在执行之前预先设置好上下文环境以便在任意时间被中断抢占，当中断响应处理对该关键区重入时，首先恢复到被抢占的上下文环境完成该关键区任务，然后再继续中断处理。

10、根据权利要求 9 所述的一种 ARM 处理器架构的微内核设计方法，其特征在于所说任何一段关键区内的任务都自行管理上下文环境：关键区内的任务用于所有内核例程，包括系统任务、普通定时器和硬中断三部分，其中硬中断的例程是不可抢占的，它可以抢占系统任务和普通定时器，系统任务和普通定时器互相之间不可抢占，普通定时器在进入中断例程后改为与系统任务相同的处理器模式，并且使用相同的一个内核堆栈和内核环境区，当被硬实时中断抢占时，则它们的上下文保存到内核环境区中，如果是用户代码被硬实时中断抢占，则用户代码的上下文保存到对应服务的环境区的空间中，对硬实时中断而言，只是将被抢占指令的上下文保存到全局环境区中，硬实时中断使用另外一个堆栈，系统任务和普通定时器的进入和返回步骤包括，首先将被抢占的用户代码保存到全局环境区中，然后将全局环境区指向内核例程的上下文，然后屏蔽普通定时器中断，接下来打开处理器的中断屏蔽模式，允许 IRQ 中断，然后进入可以被抢占的内核例程中，执行完之后，首先进入安全模式，屏蔽 IRQ 中断，然后恢复普通定时器中断，设置全局环境区指向为被抢占的用户代码的上下文，最后切换回原用户代码的执行指令环境，硬中断的进入和返回步骤包括，首先将被抢占的代码保存到全局环境区中，然后执行中断例程，最后再恢复执行。

ARM 处理器架构的微内核设计方法

技术领域

本发明涉及一种操作系统微内核，尤其涉及一种 ARM 处理器架构的微内核设计方法。

背景技术

随着网络技术的发展，采用 ARM 处理器的嵌入式设备联网的趋势日渐明显。嵌入式操作系统的运行环境也从孤立的单机环境发展为复杂的网络分布式环境，应用软件也出现了向以服务为单位发展的趋势。但是针对 ARM 架构设计的嵌入式操作系统往往是单内核系统，难以采用面向服务的软件技术进行开发，由此带来了软件开发周期长，软件难以重用、维护和更新、系统不易于用户定制等缺点。随着嵌入式设备硬件结构日趋复杂，功能日益增强，对嵌入式操作系统技术和嵌入式软件开发技术提出了新的挑战。这主要包括：支持软件快速开发和升级、支持嵌入式软件的高度可配置性、轻便的移动应用和多媒体的信息处理。为控制功能的复杂性、减少开发难度、保障软件质量和缩短开发周期，嵌入式应用软件的开发需要操作系统和强大的开发工具的支持。目前国内外被广泛应用的商业嵌入式实时操作系统大部分是传统的嵌入式操作系统。国外嵌入式实时操作系统有：VxWorks, Microsoft Windows CE, QNX, NuClear, eCos 等等。在国内，凯思集团的“女娲 Hopen”是拥有自主产权的嵌入式操作系统，应用于数字电视、股票机顶盒、VOD 视频点播机顶盒等方面。嵌入式 Linux 方面，中软总公司以数控平台为背景，推出了中软 Linux3.0，主要应用于数控机床等工业控制领域。中科红旗公司把工控和信息家电作为主要的发展领域，开发出了针对工控领域的红旗 ControLinux。商用嵌入式操作系统如表 1 所示：

表 1 商用嵌入式操作系统

公司名称	OS 名称	备注
QNX Software Systems	Neutrino RTOS	
Wind River Systems	VxWorks	应用于实时领域
Palm	PalmOS	应用于 PDA
Microsoft	WinCE	
Red Hat	eCos	开放源码

MontaVista Software	MontaVista Linux	实时嵌入式 Linux
凯思昊鹏软件工程技术有限公司	Hopen	
中软网络技术股份有限公司	中软 Linux	工业控制领域
北京中科红旗技术有限公司	ControLinux	工业控制领域

在这些流行的嵌入式操作系统中，基于面向服务的微内核技术构建的嵌入式操作系统还不多。而在内核上支持多服务器技术特点的嵌入式操作系统目前还不存在。基于本发明所实现的嵌入式操作系统是新一代的嵌入式操作系统，能够满足新一代嵌入式应用的需求。

发明内容

本发明的目的是提供一种 ARM 处理器架构的微内核设计方法。

方法的步骤为：

a)将用于构造面向服务的嵌入式操作系统中所有的服务用线程来表示，系统调用都是以线程为操作对象，将服务的地址空间作为线程附带的一部分状态进行管理；

b)用线程间通信方式实现多服务器协作，通信数据用寄存器组进行传递；

c)在线程间通信接口中支持地址空间的传递，用于不同地址空间的服务协作中大量的数据通信；

d)将微内核中无法确定计算量的大任务划分为多个基本的内核指令，进行流水作业；

e)对关键区采用非阻塞式同步访问机制，实现完全可抢占性。

本发明克服 ARM 架构上嵌入式操作系统在单内核体系结构上的缺陷，本发明提供一个针对 ARM 处理器优化设计的完全可抢占的微内核，该内核用于支持下一代面向服务的嵌入式操作系统。在架构设计上重点考虑安全性、可扩展性和实时性能，以服务的形式构造上层应用，实现了高度的模块化结构和稳定的系统性能，为开发复杂的嵌入式操作系统提供了坚实的基础。

本发明围绕服务的概念设计微内核，每个服务独立为一个线程，该模块包含了相关的内部数据和硬件资源，操作系统是多个服务器线程的组合，服务器与客户以及服务内部线程使用相同的线程间通信技术，动态更改通信结构则实

现服务动态替换、参数化或者分层叠加。微内核以线程和线程间通信为基础执行并管理服务器进程，使之高效的协同工作。线程间通信只使用 ARM 通用寄存器组发送和接收数据，内核完成一次线程间通信的数据传递，只需要将数据从发送进程的寄存器组拷贝到接收的寄存器组即可。内核支持在线程间通信中共享地址空间，用于服务器进程间传输更多数据。内核使用自卷上下文结构实现完全可抢占性，支持硬实时中断响应。

具体实施方式

线程是 Pcanel 支持的唯一的活动主体，Pcanel 的系统调用都是以线程为操作对象。一个线程代表一个单独的服务，具有单独的寄存器组，包括指令执行地址、堆栈地址、处理器状态等寄存器，Pcanel 对多服务器的支持直接体现在多线程管理上。

所说将用于构造面向服务的嵌入式操作系统中所有的服务用线程来表示：服务用一个 32 位无符号数标示，具有单独的寄存器组，包括指令执行地址、堆栈地址、处理器状态寄存器，创建新服务必须指定合作服务、空间服务及监控服务。通过指定一个现有的服务作为合作服务以共享同一个地址空间。

Pcanel 中的线程基本上和 POSIX 线程功能相当，运行在同一个地址空间的所有线程则构成 POSIX 的进程概念。当用户创建一个新的线程 A 时，可以通过指定一个现有的线程 B 作为“同事”，这样，线程 A 与线程 B 就共享同一个地址空间；如果用户没有指定“同事”，则 Pcanel 为新线程创建一个独立的地址空间。

每个线程用一个 32 位无符号数表示，0 表示空，若最高位为 0 表明系统服务线程，可以调用一些特权 API，如果最高位为 1 表明普通用户线程，不可以调用 Pcanel 的特权服务。比如创建线程就是特权服务。线程 ID 号由创建者设定，创建线程的能力完全由特权线程控制，普通线程可以通过向特权线程发送请求间接创建线程。

Pcanel 支持基于优先级的分时调度。缺省情况下，内核上将线程的优先级自高到低分为紧急级、实时级、正常级、后台级四层，总共定义了 15 个优先级，最高级为 14，最低级为 0，其中紧急级为 12，实时级为 8，正常级为 5，后台级为 2。紧急级一般是用于系统紧急事件的处理，实时级用于实时系统的用户事件处理，正常级则用于分时系统的任务调度，后台级一般用于显示一些调试信息等可以延缓处理的后台服务。优先线的数目以及四个预定义优先级在内核信息表中存取，通过启动时更改内核信息表可以定制线程的优先级数目。

优先级为 0 的线程是一个比较特殊的空闲线程，它并不参与调度过程，只有当所有其他线程都不处于活动状态的时候，Pcanel 才会运行这个线程。该线程一般用于能耗管理，比如当持续运行 1 分钟后，可以关闭一些外围器件或者降低处理器主频。

任何一个线程创建的时候，都需要指定监控线程（Keeper）和宿主线程（Spacer），只有这两个线程才能控制其运行、暂停或者删除。其中 Spacer 线程还负责内存分配，也就是说，当一个线程调用 malloc()这样的内存分配函数时，该内存分配请求将由 Spacer 来实现。

线程的执行地址（IP）和堆栈地址（SP）是两个最重要的值。当线程最初创建时，并不指定 IP 和 SP，因此它不处于活动状态，不能被调度执行。当用户指定了合理的 IP 和 SP 值后，线程进入活动状态，可以被调度执行。IP 值为特殊的 0 值时，表明暂停该线程，如果设置 IP 为 1 表明继续运行该线程；在暂停的情况下设置合理的 IP 值可以使暂停的线程从新的地址继续运行。SP 值为特殊的 0 值时，表明删除该线程。这样，用户通过设置 IP 和 SP 的值，可以实现对线程的全面控制。

线程间通信

Pcanel 基于通信顺序过程处理模式，以线程间通信方式实现相同或不同地址空间的服务之间的数据通信，从而实现服务接口的调用，Pcanel 的主要功能就是实现线程间通信（线程间通信）。线程间通信既可以用于相同地址空间的线程，也可以用于不同地址空间的线程。

线程间通信两阶段

任何一个线程间通信都分为两个阶段：发送阶段和接收阶段。如果一个线程间通信只发送数据，那么其接收阶段为空，同样如果一个线程间通信只接收数据，那么其发送阶段为空。Pcanel 支持阻塞式的线程间通信调用，也就是说，当线程 A 对线程 B 发送数据时，如果线程 B 并没有处于接收阶段，或者虽然处于接收阶段，但是线程 A 并不在其接收范围内，那么线程 A 将成为阻塞状态，直到线程 B 接收数据为止。Pcanel 也支持非阻塞的线程间通信调用，也就是说，如果发送/接收阶段不能立即执行，线程将得到一个错误值，而不会进入阻塞状态。

线程间通信的发送/接收阶段只使用通用寄存器组，也就是说，发送/接收的数据完全通过通用寄存器来实现，现代 RISC 处理器都具有较多的通用寄存器，一般可以支持 6 个 32 位整数的线程间通信数据内容，根据对一些操作系统的系

统调用参数的统计情况看，数据内容平均在 3~4 个 32 位整数之间。因此使用通用寄存器来实现线程间通信数据的传递，其效率很高。以 ARM 处理器的函数调用为例，函数的参数通过 R0、R1、R2、R3 传递，其余更多的参数则放到堆栈中，因此通常 Pcanel 完成一次线程间通信的数据传递，只需要将数据从发送线程的寄存器组拷贝到接收线程的寄存器组即可，并不涉及线程的地址空间、堆栈等，因此能够更高效的发挥处理器一级缓存的作用。

线程间通信的发送阶段必须指明接收线程；而线程间通信的接收阶段则不一定需要指明从哪个线程接收数据，具体分为三种情况，一是接收来自任何线程的数据，二是接收来自同地址空间线程的数据，第三种情况较为特殊，是针对 Pcanel 的多服务器架构设计的，称为客户/服务器模式。

客户/服务器模式

线程间通信的一个重要用途是向客户端程序提供服务器调用接口。通常情况下，线程 A 要调用线程 B 的服务，必然是先向线程 B 发送数据，然后等待来自线程 B 的回复，这两个阶段是紧密相连的，线程 A 不希望在等待线程 B 回复的过程中收到来自其他线程的数据，因此 Pcanel 设计了一种“客户/服务器”模式的线程间通信：当服务 A 调用服务 B 时，服务 A 充当客户角色，服务 B 充当服务器角色，即线程 A 进入发送阶段，当线程 B 接收数据之后，线程 A 直接进入接收阶段的阻塞状态，直到再次收到来自线程 B 的数据，在此过程中，任何其他线程如果向线程 A 发送数据，都将被阻塞。

抢占式线程间通信

Pcanel 还提供了一种特殊形式的“抢占式”线程间通信，它有些类似于 POSIX 的信号机制。当线程处于发送/接收阶段的阻塞状态时，用户可以通过向其发送“抢占式”线程间通信，打断阻塞状态。此时线程得到的结果有两种：第一种情况是线程原先处于发送阶段的阻塞状态，那么线程将得到一个错误返回值，下一次调用只有接收阶段的线程间通信时，将得到这个“抢占式”线程间通信的数据；第二种情况是线程原先处于接收阶段的阻塞状态，那么线程将得到另一个错误返回值，以及这个“抢占式”线程间通信的数据。与 POSIX 信号机制不同的是，如果“抢占式”线程间通信发生时，线程并没有处于调用线程间通信的阻塞状态，那么这个“抢占式”线程间通信将保存起来，直到线程下一次调用线程间通信的时候才接收到。

“抢占式”线程间通信往往用于发送一些紧急事件，比如对线程间通信的超时处理、硬件中断或者其他异常情况的发生等。

传递地址空间

当线程间通信传递的数据较多时，可以通过共享地址空间来完成。对此，Pcanel 实现了线程间通信的一种增强模式，支持地址空间的传递。

当线程 A 向线程 B 发送较多数据时，往往是以某内存块的形式调用，一个参数是该内存块的地址 Address_Src，另一个参数则是该内存块的字节大小，而线程 B 在调用线程间通信接收数据时，则由内核指定某内存地址 Address_Dst，Pcanel 将把线程 A 所在的地址空间中 Address_Src 对应的物理内存区域直接映射到线程 B 的地址空间中，即虚拟地址与物理地址的值相同。在共享期间，线程 B 可以直接访问这块虚拟地址空间；当线程 B 再次向线程 A 发送数据时，这块虚拟地址空间将归还给线程 A，线程 B 自然也就不能再访问该地址空间了，上述的增强模式必须在“客户/服务器”模式的线程间通信上才能实现。

地址空间

MMU 是 ARM 的协处理器，分为 First-Level（L1）和 Second-Level（L2）两层映射表。不同的 L1 表代表了不同的地址空间。Pcanel 通过直接管理地址空间来实现多服务器之间的保护机制。

采用地址空间后，内存访问地址都是虚拟地址，其映射的物理地址由 Pcanel 指定并控制访问权限，一般分为不可访问、只读、读写三种权限，在个别微处理器平台上，还可以设置“可执行”权限。任何一个服务总是运行在某一个地址空间内，不同地址空间的服务可以对同一物理地址进行映射，因此数据块可以在不同地址空间传递，而不需要重复拷贝。

Pcanel 没有提供单独的数据结构和接口来管理地址空间，而是将其作为线程的一部分进行内部维护。

流水线内核

有一些系统调用需要完成复杂的功能，不能在短时间或者说可估计的时间内完成，这样势必影响系统的实时响应性能。在下面这个案例中系统调用的执行时间无法确定。

一组线程$\{t_1,t_2,...,t_{n-1},t_n\}$，其中$t_1$不处于线程间通信接收阶段，对于$2 \le i < n$，线程$t_i$调用的线程间通信包括发送和接收阶段，向线程$t_{i-1}$发送数据，然后接收任何线程的数据，线程$t_n$只有线程间通信发送阶段，此时$\{t_2,t_3,...,t_{n-1},t_n\}$线程都处于阻塞的发送阶段。在某时刻 Pcanel 调度t_1执行，当t_1调用线程间通信接收t_2的数据后，Pcanel 应将t_2转入线程间通信接收阶段，由于t_3正处于线程间通信发送阶段，因此线程t_2立即接收t_3的数据，并进入活动状态，以此类推，$\{t_2,t_3,...,t_{n-1},t_n\}$线程都将依次进入活动状态。

这里所有的行为都是由于t_1进入线程间通信接收阶段这一事件引起的，如果Pcanel连续执行所有的行为，则其执行时间依赖于n的值，属单调递增关系，执行时间显然不确定。Pcanel采用了流水线技术来解决这一问题。

Pcanel将物理上的处理器划分为虚拟的“内核指令”处理器和“普通指令”处理器，每个系统调用提供的功能是由若干条基本的“内核指令”组合完成，而用户态程序执行的是“普通指令”，系统调用的开始相当于取指令，大多数简单的系统调用只需要一条“内核指令”即可完成，而复杂的系统调用类似于超长指令，可以分解为依次执行的多条简单指令。所有基本的“内核指令”都可以在固定的短时间内完成，而需要连续执行的多条“内核指令”则放入一条流水线中，定时器中断将激发“内核指令”取指器从流水线取得“内核指令”并执行，这就是“内核指令”的时钟周期。一条“内核指令”在执行期间，可以产生更多的“内核指令”，它们都被放入流水线中。当一条（超长）“内核指令”完成之后，把结果写回线程的寄存器组，线程就可以继续执行下去。

这样的内核结构称之为流水线内核，所有“内核指令”的最大延时就是整个系统的最大中断响应延时。从用户态程序的角度看，系统调用相当于虚拟的特权指令，而复杂系统调用则相当于虚拟的超长特权指令。系统调用的参数包含在线程的寄存器组中，执行结果也通过寄存器组返回给线程。

就前述案例而言，线程t_i进入接收阶段，将立即接收t_{i+1}的数据，但是t_{i+1}不会立即进入接收阶段，Pcanel将发射一条“内核指令”，该指令的功能是令t_{i+1}线程进入接收阶段。在下一个“内核指令”的时钟周期到来时，Pcanel将执行这条指令，并且在执行完毕的时候，又将发射新的一条指令来令t_{i+2}线程进入接收阶段。在具体实现中，Pcanel可以在一条“内核指令”中令多个线程依次进入接收阶段，以提高效率。

“内核指令”的概念是简单的，其实现难点在于，每一条“内核指令”的实现必须是完全独立的过程，无论当前内核处于何种状态，都能够正确执行。Pcanel的体系结构是不可重入的，也就是说，Pcanel执行内核态代码的工作是不能被中断的。通过仔细实现系统调用，可以得到一个较短的单位时间，所有“内核指令”都能在这个时间内完成。

SRC结构

Pcanel在关键区内的指令采用自卷上下文（SRC）结构来保证最短的中断响应延时，实现内核的完全可抢占性。SRC 结构就是任何一段关键区指令都自行管理上下文环境，在执行之前预先设置好上下文环境以便在任意时间被抢占，

当关键区重入时，自动恢复被抢占上下文环境并切换运行的主体。SRC 结构实现了完全可抢占的调度能力和非阻塞的同步能力，用于内核的并行计算及同步，保证了 Pcanel 的硬实时性能。通过 SRC 结构，关键区采用非阻塞式同步访问机制，实现完全可抢占性：为了让中断响应抢占内核任务，将关键区改造为非阻塞式同步，任何一段关键区内的任务都自行管理上下文环境，在执行之前预先设置好上下文环境以便在任意时间被中断抢占，当中断响应处理对该关键区重入时，首先恢复到被抢占的上下文环境完成该关键区任务，然后再继续中断处理。

SRC 结构涉及系统调用、普通定时器和硬中断（高精度时间中断或者其他要求硬实时的中断）三部分。其中硬中断的例程是不可抢占的，它可以抢占系统调用和普通定时器，系统调用和普通定时器互相之间不可抢占，普通定时器在进入中断例程后改为与系统调用相同的处理器模式，即 SVC 模式，并且使用相同的一个 KERNELBIN_STACK 和 KERNEL_CONTEXT，用 Kernel_PreviousContext 指向被它们（通称为内核例程）抢占的用户代码的 CONTEXT，而当前的 EXCEPTION_CURRENTCONTEXT 则指向 KERNEL_CONTEXT，因此，如果被硬实时中断抢占，则它们的 CONTEXT 将保存到 Kernel_Context 空间中，而如果是用户代码被硬实时中断抢占，则用户代码的 CONTEXT 保存到对应线程的 Context 空间中，对硬实时中断而言，只是将 Context 保存到 EXCEPTION_CURRENTCONTEXT 中而已。硬实时中断使用另外一个堆栈。所有的堆栈都分配在 0xff00 之前的一小段空间。

内核例程（Kernel_ServiceEntry 和 KernelTimer_Handler）首先将被抢占的用户代码保存到 EXCEPTION_CURRENTCONTEXT 中，然后先执行 Kernel_SaveContext 操作，然后屏蔽 OSTIMER0 中断，接下来进入 preemptmode，即 SVC_MODE，允许 IRQ 中断，然后进入可以被抢占的内核例程中了；执行完之后，首先进入 safemode，即 SVC_MODE，屏蔽 IRQ 中断，然后恢复 OSTIMER0 中断，最后设置当前 Context 为被抢占的用户代码上下文（Kernel_SwitchToPreviousContext），这之后就可以进行上下文切换了。

在内核中，还有 Thread_Schedule 两处使用 SRC 结构，一处是线程的调度确定之后，应当将 Kernel_PreviousContext 设置到该线程对应的 Context，另一处是 Memory_Switch 中先进入 safemode，然后切换 mmu，再重新进入 preemptmode。

对于硬中断例程，首先将被抢占的用户代码（也可能是内核例程）保存到 EXCEPTION_CURRENTCONTEXT 中，然后执行中断例程，最后再恢复执行，

过程比较直接。

内核例程使用 SRC 结构如下：

```
ldr   sp, exception_currentcontext
stmia sp!, {r14}
mrs   r14, spsr
stmia sp!, {r14}
stmia sp, {r0-r14}^
nop
ldr   sp, =KERNELBIN_STACK
ldr   r0, exception_currentcontext
ldr      r1, [r0]
ldr   r0, [r1, #-4]
   bic      r0, r0, #0xFF000000
ldr   r2, exception_currentcontext
str   r2, Kernel_PreviousContext
ldr   r3, =Kernel_Context
str   r3, exception_currentcontext
mov   r1, #0x40000004
add   r1, r1, #0xd00000
ldr   r2, [r1]
bic   r2, r2, #0x04000000
str   r2, [r1]
mrs   r4, cpsr
bic   r4, r4, #0x80
msr   cpsr, r4
bl Kernel_ServiceEntry
mrs   r4, cpsr
orr   r4, r4, #0x80
msr   cpsr, r4
mov   r1, #0x40000004
add   r1, r1, #0xd00000
ldr   r2, [r1]
```

```
orr   r2, r2, #0x04000000
str   r2, [r1]
ldr   r2, Kernel_PreviousContext
str   r2, exception_currentcontext
 swi_recover_context:
ldr   sp, exception_currentcontext
ldmia sp!, {r14}
ldmia sp!, {r0}
msr   spsr, r0
ldmia sp, {r0-r14}^
nop
movs  pc, r14
```

引导

Pcanel 的引导程序 Initer 首先进行硬件初始化，设置一些环境参数，然后将操作系统内核加载运行。引导程序 Initer 实现了一个基本的文件系统，它把 Flash 存储空间分配给两个映像文件，第一个映像文件存放 Pcanel 相关的系统模块，包括初始化程序、内核模块、资源服务器、文件系统服务器、核心动态链接库等，第二个映像文件占据其余的 Flash 存储空间，为应用程序提供完整的文件系统服务。

Initer 将 Flash 存储系统映射到物理地址 0x04000000 开始的 32M 空间，由于使用了支持 Execute-in-Place（XIP）的静态内存，处理器可以直接从 Flash 执行代码。Initer 进行硬件初始化之后，直接跳转到第二部分初始化程序的开始地址执行。

内核运行环境的地址映射表如下：

虚拟地址	物理地址	用途
0x00000000	0xA0000000	内核管理的内存空间（1MB）
0x04000000	0x04000000	Flash 存储地址空间（32MB）
0x40100000	0x40100000	FFUART Register
0x40A00000	0x40A00000	OS Timer Register
0x40D00000	0x40D00000	Interrupt Control Register
0x40E00000	0x40E00000	GPIO Register

注：寄存器映射范围均为 1MB。

初始化程序的作用是完成与内核运行相关的所有初始化工作，其任务分为四个部分。第一步，初始化 MMU，建立一些硬件寄存器的虚拟地址映射，其中动态内存的物理地址 0xA0000000 开始的 1M 空间映射到虚拟地址 0x0，这部分内存空间由 Pcanel 内核管理，存放内部数据，包括线程信息、MMU 的映射表等。第二步，建立内核的运行环境，初始化程序将把内核模块从 Flash 拷贝到虚拟地址 0x0 开始的 32K 空间内，在 0xFF00 开始的 256 个字节空间内建立起 Kernel 信息表，该表存放系统相关的各种基本参数，包括内存容量、时钟频率等。第三步，初始化程序将资源服务器、文件系统服务器和核心动态链接库加载到内存，并直接调用内核中的系统函数创建线程和分配地址空间。第四步，初始化时钟中断等硬件部分。

所有系统相关的初始化工作完成后，内核将掌管系统，并调度资源服务器线程开始运行。

异常处理

ARM 架构具有标准的异常产生机制。有些异常是系统硬件引起的，比如定时器中断，有些是用户线程运行过程中引起的，比如堆栈溢出，大多数异常都会引发线程调度。用户线程可以在 Pcanel 中注册接收系统的异常（Exception），当该异常发生的时候，Pcanel 通过进程间通信（线程间通信）将相关信息发送给这个线程。

异常事件

Pcanel 将 ARM 处理器架构中可能产生的异常事件分为四类：

■ Software interrupt：用户线程执行“SWI”指令，就会产生软件中断异常事件。Pcanel 的系统调用接口就是通过 SWI 指令实现，SWI 指令的低 24 位用来表示系统调用的索引。

■ Prefetch/Data Abort：指令预取或者数据访问的时候，由内存系统产生的异常事件。Pcanel 采用内存管理单元（MMU）来管理内存，当发生该异常时，一般是由于线程运行出错访问非法地址，例外的情况是堆栈溢出，此时 Pcanel 将事件发送给内存管理器，由它自动扩展该线程的堆栈空间来解决异常。

■ Undefined instruction：线程执行非法指令。这种异常发生的时候，Pcanel 将停止线程的运行，并发送事件给相应的内存管理器，它将收回分配给该线程的地址空间，然后将线程删除。

■ Interrupt：ARM 处理器产生的中断分为 IRQ 和 FIQ（快速中断）两种模式，Pcanel 统一使用 IRQ 模式的异常处理例程。对于定时器中断，Pcanel 进行抢占

式的分时线程调度；对于其他中断，Pcanel 将该事件发送给注册接收的线程。

异常的进入和退出

ARM 处理器具有七种不同的运行模式（usr、fiq、irq、svc、abt、und、sys），Pcanel 的用户线程总是运行在 usr 模式，前面列举的四类异常分别使处理器进入 sys、abt、und、irq 模式。所有处理器模式共享通用寄存器组 R0~R7，各自配备独立的 R8~R14 寄存器组。

为简化中断处理例程的代码，Pcanel 总是切换到 svc 模式运行，因此内核运行的堆栈寄存器（R13）就是相同的。当进入异常处理例程后，Pcanel 总是将处理器的通用寄存器组 R0~R7 都保存到当前运行线程的控制块（TCB）中，作为线程的上下文（Context）的一部分，而当 Pcanel 处理完异常，可采取两种方式退出，一是如果没有发生线程切换，依然返回到刚才被抢占运行的线程，由于线程是在 usr 模式运行，而内核是在 svc 模式运行，不会更改线程的 R8~R14 寄存器组，因此只要将线程的 R0~R7 寄存器组重新恢复到处理器中即可；二是如果发生线程切换，则首先要保存被抢占线程的 R8~R14 寄存器组，然后将新线程的 R0~R14 寄存器组全部恢复到处理器中。下面是一个 DataAbort 处理例程。

```
ldr   sp, exception_currentcontext
stmia sp!, {r14}
mrs   r14, spsr
stmia sp!, {r14}
ldr   r14, =KERNEL_MODE
msr   cpsr, r14
stmia sp, {r0-r14}^
mrc p15, 0, r0, c5, c0, 0
mrc p15, 0, r1, c6, c0, 0
ldr   sp, =KERNELBIN_STACK
bl data_abort_handler
ldr   sp, exception_currentcontext
ldmia sp!, {r14}
ldmia sp!, {r0}
msr   spsr, r0
ldmia sp, {r0-r14}^
movs  pc, r14
```

内存管理

Pcanel 通过直接管理地址空间来实现多服务器之间的保护机制。MMU 管理表，分为 First-Level（L1）和 Second-Level（L2）两层映射表。MMU 使用不同的 L1 表，就可以切换到不同的地址空间。

采用地址空间后，内存访问地址都是虚拟地址，其映射的物理地址由 Pcanel 指定并控制访问权限，具有不可访问、只读、读写三种权限。任何一个服务器总是运行在某一个地址空间内，不同地址空间的服务可以对同一物理地址进行映射，因此数据块可以在不同地址空间传递，而不需要重复拷贝。

Pcanel 管理着一些动态的数据结构，它们都在内核空间（缺省为 1M 字节）进行分配和维护，因此需要具备一套内部使用的内存管理算法。内核数据结构的大小与处理器关系密切，算法必须简单快速。

Pcanel 内部分配的数据结构按照大小可以分为如下两类：

16K：Pcanel 使用 16K 字节的内存作为 MMU 的 L1 表，要求开始地址与 16K 对齐。

1K：Pcanel 使用 1K 字节的内存作为 MMU 的 Large Page Table（LPT），要求开始地址与 1K 对齐。Pcanel 还使用 1K 字节的内存作为线程控制块（TCB）。

另外，Pcanel 还需要维护一些全局的数据，例如标记内存块分配/释放的位图数组、线程调度队列等，它们的大小与系统的具体配置有关。

总体考虑内存分配的需求，Pcanel 采用 Bit Maps 算法管理内存空间的分配情况，单位块大小是 1K，采用 Linked List 算法管理可用的内存块，具有 16K、4K 和 1K 三种规格。

在嵌入式系统中，内存空间较少，应用程序的内存分配的复杂度较低，因此 Pcanel 统一采用 Coarse Page Table 规格，表中共有 256 个页描述项，每一项对应一个 Small Page，或者连续 16 项对应同一个 Large Page，每一项具有四个访问权限单元，分别控制四分之一的子空间，对于 Small Page，其访问权限单元 AP0、AP1、AP2、AP3 分别控制 1K 子空间。Pcanel 通过线程间通信传递地址空间，由于两个地址空间的 Page Table 规格相同，就能够迅速完成传递工作，代码量也很少，这对于保证 Pcanel 的性能起了非常重要的作用。

内核接口

线程

✓ 线程的创建

```
unsigned int Thread_Create(unsigned int dst, unsigned int coagent, unsigned int keeper, unsigned int spacer);
```

参数：

dst 新创建的线程的 id 号

coagent 伙伴线程 id，伙伴进程间使用同样的地址空间，拥有同一张 L1TABLE

keeper 监护线程 id，负责处理线程运行时出现的各种执行异常

spacer 地址空间管理线程，负责创建线程的地址空间分配、再分配、映射等工作

✓ 读取线程的 Reg 值

```
unsigned int Thread_GetRegs(unsigned int dst, unsigned int mask, unsigned int
* registers);
```

参数：

dst 线程号

mask Reg 掩码，通过置不同的位区分各个 Reg，具体见 1.1.2

registers 依次表示要读取的 Reg 值

✓ 设置线程的 Reg 值

```
unsigned int Thread_SetRegs(unsigned int dst, unsigned int mask, unsigned int
* registers);
```

参数：类似读取线程的 Reg 的情形

dst 线程号

mask Reg 掩码，通过置不同的位区分各个 Reg，具体见 1.1.2

registers 依次表示要读取的 Reg 值

寄存器掩码

在参数 mask（Reg 掩码）中设置不同的位来标志这 4 个 Reg：

```
#define THREAD_REGSMASK_SP        (1 << 0)
#define THREAD_REGSMASK_IP        (1 << 1)
#define THREAD_REGSMASK_STATUS    (1 << 2)
#define THREAD_REGSMASK_CUSTOM    (1 << 3)
```

✓ sp 存放堆栈地址

✓ ip 下一条指令地址

✓ status 存放线程状态，通常是用来读取状态用。

Status 值的设置：

高 16 位标志线程当前工作状态，有 Active、Block、Suspand 和 Signal 这 4 种设置。

```
#define THREAD_STATUS_ACTIVE  (1 << 16)
#define THREAD_STATUS_BLOCK   (1 << 17)
#define THREAD_STATUS_SUSPEND (1 << 18)
#define THREAD_STATUS_SIGNAL  (1 << 19)
```

低 16 位标志优先级，目前有 NORMAL（0x5），REALTIME（0x8）和基本进程（0x0）。其中，基本进程即为 IDLE 状态，可以负责有关功率消耗等问题处理。

✓ custom 用户自定义的 Regs，是扩展出的虚拟寄存器。

地址空间

函数接口

✓ 地址空间的映射

```
unsigned int Memory_Map(unsigned int src_pVAddress, unsigned int ap, unsigned
int dst, unsigned int dst_pVAddress, unsigned int size);
```

参数：

src_pVAddress	Spacer 在自己的地址空间中找到的足够的空间的首地址
ap	子线程对此空间的读写权限（只读、只写、可读可写、无权读写）
dst	子线程 id 号
dst_pVAddress	映射到子线程的空间首地址
size	空间大小

✓ 地址空间释放

```
unsigned int Memory_Unmap(unsigned int dst, unsigned int dst_pVAddress, unsigned int size);
```

参数：与 Map 情形类似

src_pVAddress	Spacer 在自己的地址空间中找到的足够的空间的首地址
dst	子线程 id 号
dst_pVAddress	映射到子线程的空间首地址
size	空间大小

函数说明

线程对自己的空间不能管理，所有的空间映射与释放和权限设置都由线程的 Spacer 来控制。每次线程需要空间，它就向自己的 Spacer 发出请求（IPC 方式），由 Spacer 调用这个函数并实现空间的映射。Spacer 总在自己的空间中找到一块可用且足够的空间给线程，若寻找不到可用的空间，Spacer 将向它的 Spacer，也就是上一级 Spacer 请求空间来满足需要。

这里所涉及的空间操作都是虚存操作。其中的映射工作由内核完成，对外不可见。当操作失败的时候，比如没有合适的空间分配，函数返回异常。如果是空间不足造成的异常，当 spacer 向上级 spacer 申请到空间之后，重新操作满足线程的空间请求。

线程间通信

```
unsigned int Ipc_call(unsigned int option, IpcBuffer * buffer);
```

参数：

option	IPC 调用的方式
buffer	相关的 IPC 信息块地址

备注：

IPC 传输普通数据的 4 种方式（option）：

1、IPC_OPTION_SEND

发送等待直至发出 IPC，或者异常

2、IPC_OPTION_RECV

接收等待直至得到 IPC，或者异常

3、IPC_OPTION_UNBLOCKSEND

无阻塞发送，发送之后不关心结果，即是否可以到达，类似 UDP 协议

4、IPC_OPTION_UNBLOCKRECV

无阻塞接收，查找是否有 IPC 到达，没有就返回

IPC 传输中的 3 种辅助设置（可以与其他 option 任意搭配）：

1、IPC_OPTION_FROMCOAGENT

只接收伙伴线程的 IPC 请求，其他的请求不关心

2、IPC_OPTION_SIGNAL

紧急信号，可以结束所有等待状态，转向其他指定操作

3、 IPC_OPTION_CLIENTSERVER

等待某个特定回复，直到回复到达

IPC 内存地址传输调用的 2 种方式：

1、IPC_OPTION_SENDSPACE

发送一块地址空间，其中接受方可以判断是否接受

2、IPC_OPTION_RECVSPACE

接收一块地址空间，目前尚不支持。

线程运行库接口

```
unsigned int thread_setpriority(unsigned int threadid, unsigned int priority);
unsigned int thread_delete(unsigned int threadid);
unsigned int thread_suspend(unsigned int threadid);
unsigned int thread_resume(unsigned int threadid);
unsigned int thread_create(unsigned int newthread,unsigned int coagent, unsigned int keeper, unsigned int spacer, unsigned int sp, unsigned int ip, unsigned int priority);
unsigned int thread_getip(unsigned int threadid);
unsigned int thread_getsp(unsigned int threadid);
unsigned int thread_getstatus(unsigned int threadid);
unsigned int thread_getcustom(unsigned int threadid);
unsigned int thread_setcustom(unsigned int threadid, unsigned int custom);
unsigned int thread_precreate(unsigned int newthread, unsigned int coagent, unsigned keeper, unsigned int spacer);
unsigned int thread_start(unsigned int threadid, unsigned int sp, unsigned int ip, unsigned int priority);
```

线程间通信运行库接口

```
#define IPC_RECV_ALL    0
#define IPC_RECV_COAGENT   1
#define IPC_RECV_SERVER 2
typedef struct _IPCBuffer
{
   union
   {
```

```
    unsigned int to;
      unsigned int from;
   }id;
   unsigned int data[IPCREGISTERS];
}IPCBuffer;
unsigned int IPC_call(unsigned int option, IPCBuffer * buffer);
```

阻塞模式

```
unsigned int IPC_Send(unsigned int to, unsigned int * data);
unsigned int IPC_Recv(unsigned int * from, unsigned int * data, unsigned
int coagent);
unsigned int IPC_SendRecv(unsigned int to, unsigned int * senddata,
unsigned int * from, unsigned int * recvdata, unsigned int coagent);
```

非阻塞模式

```
unsigned int IPC_UnblockSend(unsigned int to, unsigned int * data);
unsigned int IPC_UnblockRecv(unsigned int * from, unsigned int * data,
unsigned int coagent);
unsigned int IPC_UnblockSend_Recv(unsigned int to, unsigned int *
senddata, unsigned int * from, unsigned int * recvdata, unsigned int
coagent);
```

客户/服务器模式

```
unsigned int IPC_Request(IPCBuffer * IPC);
unsigned int IPC_WaitRequest(IPCBuffer * IPC);
unsigned int IPC_AnswerAndWaitRequest(IPCBuffer * IPC);
unsigned int IPC_Request_Space(IPCBuffer * IPC);
```