微内核架构多线程机制的形式化设计研究

钱振江[1,2,3] 卢 亮[1,2] 黄 皓[1,2]
(南京大学软件新技术国家重点实验室 南京 210046)[1] (南京大学计算机科学与技术系 南京 210046)[2]
(常熟理工学院计算机科学与工程学院 常熟 215500)[3]

摘 要 微内核架构因其有效的模块隔离性而成为操作系统方面研究的热点,多线程机制是微内核架构需要解决的关键性能问题。有不少的工作对微内核架构多线程机制进行了研究,但存在频繁的系统地址空间切换和实现复杂度高的问题。采用形式化的方式对微内核架构多线程和安全机制进行描述和设计,提出一个微内核线程分层对象语义模型,用以设计多线程机制的线程间通信、调度和互斥同步方案。在已实现和验证的微内核操作系统VTOS中对多线程功能和性能进行了测试,结果表明VTOS有效地实现了多线程机制,并具有很好的系统性能。

关键词 微内核,多线程,操作系统,形式化描述,形式化设计
中图法分类号 TP316 **文献标识码** A

Research on Formal Design of Multi-thread Mechanism Based on Microkernel Architecture

QIAN Zhen-jiang[1,2,3] LU Liang[1,2] HUANG Hao[1,2]
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210046, China)[1]
(Department of Computer Science and Technology, Nanjing University, Nanjing 210046, China)[2]
(School of Computer Science and Engineering, Changshu Institute of Technology, Changshu 215500, China)[3]

Abstract Microkernel architecture has become a hot topic in the research area of operating systems because of its effective isolation for modules. The multi-thread mechanism is a critical issue for the performance of the microkernel architecture. Many works research into the multi-thread of microkernel operating systems, but there are some problems such as frequent switching of system address space and high degree of implementation complexity. We used formal methods to describe and design the multi-thread and security mechanism, and proposed a hierarchical object semantics model. With the object semantics model, we formally designed the mechanism of inter-thread communication, thread scheduling, mutual exclusion and synchronization. Meanwhile, we used our self-implemented and verified microkernel operating system-VTOS as an example to test, and the results show that VTOS achieves multi-thread effectively and has a good system performance.

Keywords Microkernel, Multi-thread, Operating system, Formal description, Formal design

1 引言

操作系统(Operating System,OS)是重要的系统软件,其可靠性、安全性和性能对于整个应用环境的正常高效运行至关重要[1]。OS的架构主要分成宏内核和微内核[2]框架。相比于宏内核OS,微内核架构OS将尽可能减少运行于特权模式的代码量,其拥有最小的可信计算基(Trusted Computing Base,TCB),能够提供很好的策略与机制的隔离,同时也便于验证。微内核中仅含有OS的基本功能,如中断机制、消息处理、进程/线程调度等,而其它许多功能模块被实现为用户态的服务程序[3]。操作系统的服务以客户/服务器(Client/Server)的方式提供给应用程序。根据微内核提供的消息通信机制(Inter-Process Communicating,IPC),用户进程通过发送和接收消息与系统服务进程进行通信,请求操作系统的服务。

相比于单一执行流的系统,在多线程环境下,可以将I/O消耗型工作和CPU消耗型工作划分为不同的线程,从而提高外部设备与处理器的并行效率。同时,在多核处理器平台上,多线程机制可以充分利用多核的并行处理能力。

微内核架构OS的多线程机制的研究工作正处于发展阶段。早期由卡内基梅隆大学开发的微内核系统Mach[4]支持内核级线程,其设计过程最早提出了"任务"和"线程"的概念,系统中的进程由多个线程构成,从而将系统资源分配的单位和处理器调度执行的单位区分开来。在线程调度方面,Mach采用多优先级反馈队列,针对每个处理器(核)均有一个局部就绪线程队列。尽管Mach的微内核架构带来了很多技术优势,但由于相比于传统的UNIX宏内核而言,Mach在运行过

到稿日期:2012-06-17 返修日期:2012-10-10 本文受国家高技术研究发展计划(863)(2011AA01A202),江苏省"六大人才高峰"高层次人才项目(2011-DZXX-035),江苏省高校自然科学研究项目(12KJB520001)资助。
钱振江(1982-),男,博士生,讲师,主要研究方向为操作系统安全与形式化验证、嵌入式系统,E-mail:zhenjiang.qian@gmail.com;卢 亮(1986-),男,硕士,主要研究方向为信息安全、操作系统安全;黄 皓(1957-),男,博士,教授,博士生导师,主要研究方向为系统软件、信息安全。

程中有较多的任务切换，导致频繁的地址空间切换，从而影响了整个系统的性能[5]。

由德国国家信息技术研究中心的 Liedtke J 主导设计并实现的 L4[6] 是第二代微内核的标准。L4 系统内核支持线程机制和虚拟地址空间的概念。L4 也是一个通信内核，它提供 IPC 机制，使得线程可以通过消息来进行相互通信。L4 在内存管理方面采用特有的缺页异常协议进行处理，策略和机制有效分离[7]，但带来的问题是复杂度较高，对高层应用程序员来说，编写多线程程序有额外的约束[8]。

为了准确地设计和描述微内核架构的线程机制以及安全策略和属性，本文采用形式化方法提出一个微内核框架的线程对象模型，并使用一阶逻辑(First-Order Logic，FOL)[9] 系统的命题公式形式化地描述多线程模型的安全机制，这些安全机制贯穿整个设计过程。本文是我们实现的可信操作系统(Verified Trusted Operating System，VTOS)的多线程机制的形式化描述和设计部分。

本文第 2 节说明多线程机制在 VTOS 的整体框架中的关系和基本概念；第 3 节阐述 VTOS 的线程对象模型；第 4 节在线程对象模型的基础上阐述 VTOS 的线程安全机制；第 5 节介绍 VTOS 内核级线程的关键技术；第 6 节阐述 VTOS 多线程机制的实验和性能分析；最后对本文的工作进行总结，并对未来的工作方向进行展望。

2 VTOS 系统架构

本节说明 VTOS 的整体框架以及本文设计的多线程机制在 VTOS 中的关系和基本概念。

2.1 VTOS 整体框架

VTOS 是按照我们预定的安全目标而设计的微内核 OS。VTOS 拥有安全核(Secure Kernel)和访问控制安全机制，实现了虚拟内存管理(VMM)、多线程管理(MTM)和多核调度管理(MCSM)等功能。VTOS 整体框架如图 1 所示。

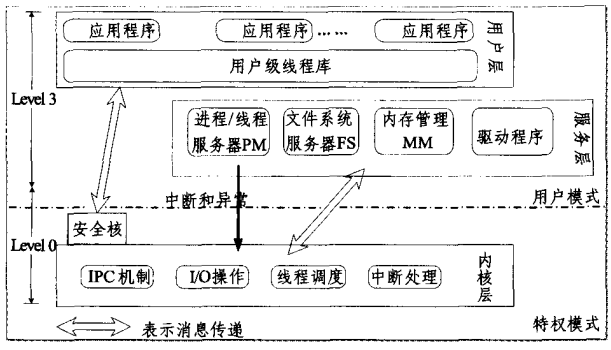

图 1 VTOS 系统架构

2.2 VTOS 线程机制

本文设计的线程机制采用了策略与机制分离的方式。VTOS 中内核级线程之间的关系由进程/线程管理服务器(PM)维护，进程之间的信号传递也在 PM 服务器中完成。微内核负责内核级线程的调度等工作。所以，本文的 VTOS 内核级线程机制的设计与 Linux 等宏内核操作系统中的设计是不同的，这种设计上的不同是由于 VTOS 操作系统的微内核架构决定的。

VTOS 内核级线程机制由微内核与 PM 共同完成：微内核负责维护系统中的内核级线程的状态，进行调度与切换工作，提供 IPC 机制；PM 负责维护内核级线程之间的关系，为信号提供策略注册与传递工作，并且 PM 也是用户进程与 VTOS 交互的接口。

本文在用户态设计了一个用户级线程库，它以 N 对 N 的线程模式[10] 向应用程序提供用户级线程服务。N 对 N 的模式是指程序中的用户级线程以一对一映射的方式绑定到轻量级进程，轻量级进程与内核线程是以一对一的关系存在，如图 2 所示，从程序看来，每个用户级线程均占用一个独立的内核级线程。N 对 N 线程模式的优点在于，由于用户级线程与内核级线程是一一对应的，因此用户级线程的调度实质是由内核来完成的，而且内核级线程可以被统一管理；同时，在多处理器或多核处理器的硬件平台上，不同的内核级线程可以在不同的核上并行地计算，也就使得应用程序中的不同用户级线程真正并行地运行。

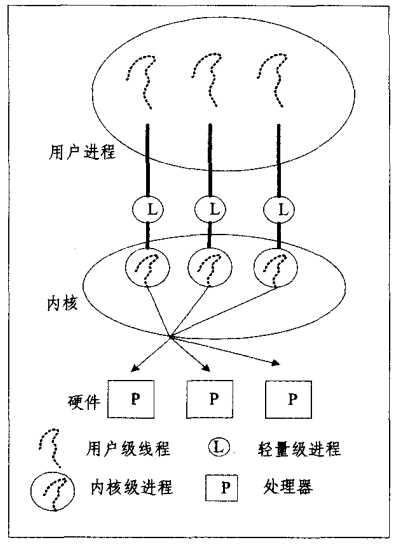

图 2 N 对 N 的线程模式

3 VTOS 线程模型

本节对 VTOS 线程机制进行抽象描述，给出一个线程对象模型。同时，在线程对象模型的基础上，使用一阶逻辑系统的命题公式形式化地描述多线程模型的安全机制。

3.1 VTOS 线程对象模型

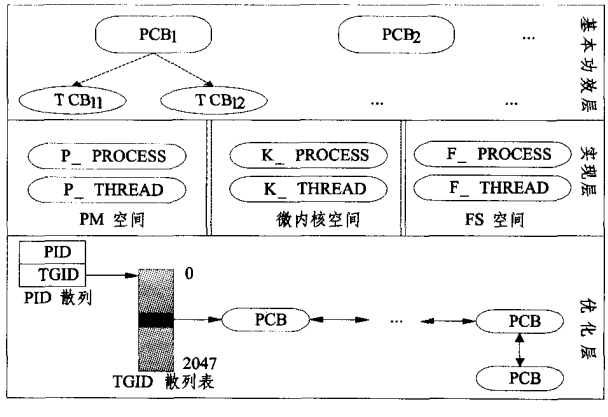

图 3 VTOS 线程对象模型

为了准确地设计和描述 VTOS 线程机制，本文给出一种分层的线程对象模型。从 3 个层次来对 VTOS 线程机制进行描述：基本功效层、实现层和优化层。基本功效层描述多线程机制的高层视图，是高层用户程序和 VTOS 多线程机制的

接口，主要包括多线程机制的关键数据对象和语义函数。实现层阐述为实现 VTOS 多线程机制的具体功能而引入的相关数据对象和语义函数。优化层阐述为优化多线程机制而引入的新的数据对象和语义函数。整个线程对象模型的框架如图 3 所示。

下面将分别阐述这 3 个层次的具体描述和设计。

3.2 基本功效层

3.2.1 基本功效层的元素集

1)常量的集合：C={MAX_PROCESS, MAX_THREAD, MAX_FILE, MAX_SLICE, MAX_VPAGE, MAX_PPAGE}，其中 MAX_PROCESS 为最大进程数，MAX_THREAD 为最大线程数，MAX_FILE 为最大文件描述符数，MAX_SLICE 为最大时间片数，MAX_VPAGE 为最大虚拟地址页，MAX_PPAGE 为最大物理地址页。

2)线程标示符 Tid 的取值集合：TIDset={1,2,…,MAX_THREAD}，我们使用 TID 表示当前所有线程标示符的集合，使用 TID(p)表示进程 p 拥有的线程的标示符集合。

3)线程的运行时间片的取值集合：RunTime={0,1,…,MAX_SLICE}，线程当前时间片的剩余滴答数目的取值的集合：LeftTicks={0,1,…,t|t∈RunTime}。

4)线程状态的取值集合：TState={*ready*, *running*, blocked, dead}。

5)处理器的模式：CPULevel={UserMode, KernelMode}。

6)线程的寄存器对象集合 TRegs。

7)线程控制块对象：TCB={Tid, TRegs, RunTime, LeftTicks, SigMask}，其中 SigMask 表示线程当前屏蔽的信号集合。当前所有线程控制块对象的集合记为 TCBset，使用 TCB(t)表示线程 t 的线程控制块对象。

8)进程标示符 Pid 的取值集合：PIDset={0,1,…,MAX_PROCESS−1}，使用 PID 表示当前所有进程标示符的集合。

9)进程的寄存器对象：PRegs。

10)进程地址空间虚拟地址页的集合：VM={a_0, a_1, …, $a_{MAX\_VPAGE-1}$}。进程的虚拟地址页的集合为 VM 的幂集，即 2^{VM}。我们也用 VM 表示函数映射：Pid→VM，VM(p)表示进程 p 的已经分配的虚拟地址页的集合。

11)进程地址空间物理地址页的集合：PM={p_0, p_1, …, $p_{MAX\_PPAGE-1}$}。进程的物理地址页的集合为 PM 的幂集，即 2^{PM}。我们也用 PM 表示函数映射：Pid→PM，PM(p)表示进程 p 的已经分配的物理地址页的集合。

12)所有页表的集合：PT={pt | pt ∈ $2^{VM\times PM}$, ∀(a_1, m_1), (a_2, m_2)∈ pt. $a_1\neq a_2 \wedge m_1\neq m_2$}。我们也用 PT 表示函数映射：Pid→PT，PT(p)表示进程 p 的页表。

13)文件描述符 Fd 的取值集合：FDset={f_0, f_1, …, f_{MAX_FILE}}，我们用 FD(p)表示进程或者线程 p 打开的文件描述符集合。

14)进程对各种信号的处理策略 SigAction，使用 default-SigAction 表示系统缺省的信号处理策略。

15)进程控制块对象：PCB={Pid, PT, PRegs, SigAction, FD, TID, subPIDset}，其中 TID 为进程中所有线程的标示符组成的集合；subPIDset 是进程当前拥有的子进程的标示符集合。当前所有进程控制块对象的集合记为 PCBset，我们用 PCB(p)表示进程 p 对应的进程控制块。

16)系统中当前进程对象的集合：P_{curr}，系统已分配的物理页集合：$PM_{curr}=\bigcup_{p\in Pcurr} PM(p)$，当前空闲的物理页对象集合：FreePM=PM\PM$_{curr}$。

3.2.2 基本功效层接口函数的语义

下面阐述基本功效层接口的语义，主要包括 fork_process, fork_thread, exec, thread_exit 和 process_exit 函数。

(1) fork_process 语义函数用于创建新的进程。假设进程 p_i 中的线程 t_j 调用此接口来创建新的进程 p_k，fork_process 将复制一个与父进程 p_i 完全相同的子进程 p_k。这两个进程有各自独立的页表 PT 对象。新进程 p_k 中仅含有新线程 t_m。线程 t_m 的寄存器对象集合中的对象取值与创建者进程 p_i 中的线程 t_j 的相应寄存器对象取值是相同的，只是所属的进程标示符不同。fork_process 的操作语义可以通过其对基本功效层的对象集的操作来说明，我们用谓词公式来表示：

PCBset=PCBset∪PCB(p_k)

PCB(p_i). subPIDset=PCB(p_i). subPIDset∪{PCB(p_k). Pid}

fork_process 安全约束条件是子进程具有独立的地址空间、寄存器环境和文件系统环境，其初始时是对父进程资源的复制，我们用谓词公式来表示：

PCB(p_i). PT≅PCB(p_k). PT

∧PCB(p_i). PRegs≅PCB(p_k). PRegs

∧PCB(p_i). FD≅PCB(p_k). FD

其中“A≅B”表示 A 和 B 是不同对象但两者的值相等。

同时，新线程 t_m 拥有自己独立的时间片资源：

TCB(t_m). RunTime≅TCB(t_j). RunTime

∧TCB(t_m). LeftTicks≅TCB(t_m). RunTime

(2) fork_thread 创建一个新线程，新线程与创建者同属一个进程。同一个进程中的所有线程共享进程的资源。假设进程 p 中的线程 t_i 创建一个新线程 t_j，线程 t_j 和 t_i 同属进程 p。fork_thread 的操作语义可以用如下的谓词公式表示：

TCBset=TCBset∪{TCB(t_j)}

PCB(p). TID=PCB(p). TID∪{TCB(t_j). Tid}

fork_thread 安全约束条件是新线程与创建者线程的寄存器环境除段寄存器 SS、栈指针寄存器 ESP 和 EAX 不同外，其他寄存器相同；新线程和创建者线程分享当前的处理器时间片资源，谓词公式表示如下：

TCB(t_i). TRegs\{SS, ESP, EAX}≅TCB(t_j). TRegs\{SS, ESP, EAX}

TCB(t_i). LeftTicks≅TCB(t_j). LeftTicks=TCB(t_j). LeftTicks/2

(3) exec：更换进程的执行文件镜像。假设进程 p_i 中的线程 t_j 要求更换进程的执行文件，首先将进程 p_i 中除线程 t_j 外的其他线程从系统中退出，根据更换的执行文件镜像构建进程 p_i 的虚拟地址空间 newVM，为进程 p_i 建立页表对象，并重新分配处理器时间片资源。exec 操作语义表示如下：

PCB(p_i). TID={TCB(t_j). Tid}

PT(p_i)={⟨va, pa⟩ | va ∈ newVM}

TCB(t_j). LeftTicks=TCB(t_j). RunTime

(4) process_exit：进程退出。假设进程 p_i 退出，首先将进程 p_i 中所有线程退出，然后将进程 p_i 退出，操作语义为：

TCBset=TCBset\{TCB(t_k) | t_k∈PCB(p_i). TID}

PCBset=PCBset\{PCB(p_i)}

(5) thread_exit:线程退出。假设进程 p_i 中的线程 t_j 从系统中退出,如果线程 t_j 不是进程 p_i 的最后一个线程,那么仅有此线程 t_j 退出,操作语义为:

TCBset=TCBset\{TCB(t_i)}

如果线程 t_j 是进程 p_i 的最后一个线程,那么进程 p_i 也退出,操作语义为:

PCBset=PCBset\{PCB(p_i)}

3.3 实现层

3.3.1 对象描述

VTOS 微内核对线程的支持只负责一些基本的机制,例如线程的调度。进程的文件资源则是由用户态的 FS 系统服务器提供的,进程的其它资源管理与策略决策均由用户态的 PM 系统服务器实现。在微内核空间中的"进程"和"线程"对象,使用 K_Process 和 K_Thread 表示,其集合用 K_PROCESS 和 K_THREAD 表示;对应地,在文件服务器 FS 中,使用 F_Process 表示"进程"在文件服务器中的部分,集合用 F_PROCESS 表示,使用 F_Thread 表示"线程"在文件服务器中的部分,集合用 F_THREAD 表示;在进程管理 PM 中,使用 P_Process 和 P_Thread 表示,集合用 P_PROCESS 和 P_THREAD 表示。为此,与基本功效层相对应,进程 p 的控制块对象 PCB(p)=K_Process(p)∪F_Process(p)∪P_Process(p)。相应地,线程 t 的控制块 TCB(t)=K_Thread(t)∪P_Thread(t)。

线程 t 使用的用户态栈占用的虚拟内存页面集合记为 userStack_vp(t),这些虚拟页面对应的物理页面对象的集合为:

$$\text{userStack\_pp}(t)=\{m \mid \exists a \in \text{userStack\_vp}(t).\ \langle a,m\rangle \in PT(t)\}$$

对应地,线程 t 使用的内核态栈占用的虚拟内存页面集合记为 kernelStack_vp(t),这些虚拟页面对应的物理页面对象的集合为:

$$\text{kernelStack\_pp}(t)=\{m \mid \exists a \in \text{kernelStack\_vp}(t).\ \langle a,m\rangle \in PT(t)\}$$

本文将线程 t 使用的栈的虚拟页面集合记作:Stack_vp(t)=userStack_vp(t)∪kernelStack_vp(t)。将线程 t 使用的栈的物理页面集合记作:Stack_pp(t)=userStack_pp(t)∪kernelStack_pp(t)。

3.3.2 接口函数的语义

本节在基本功效层的基础上,给出实现层的语义描述。

(1) fork_process 在实现层表现为对微内核、PM 和 FS 中对象的操作,主要包括进程控制块、线程控制块、用户态栈和内核态栈的对象。假设进程 p_i 中的线程 t_j 调用此接口来创建新的进程 p_k,fork_process 将复制一个与父进程 p_i 完全相同的子进程 p_k。这两个进程有各自独立的页表 PT 对象。新进程 p_k 中仅含有新线程 t_m。操作语义表示成如下的逻辑公式:

K_PROCESS=K_PROCESS∪{K_Process(p_k)}

P_PROCESS=P_PROCESS∪{P_Process(p_k)}

F_PROCESS=F_PROCESS∪{F_Process(p_k)}

K_THREAD=K_THREAD∪{K_Thread(t_m)}

P_THREAD=P_THREAD∪{P_Thread(t_m)}

F_THREAD=F_THREAD∪{F_Thread(t_m)}

fork_process 在实现层的安全约束条件在基本功效层安全条件的基础上进一步限定为进程 p_i 与进程 p_k 的信号处理策略相同,即:

PCB(p_i). SigAction≅PCB(p_k). SigAction

(2) fork_thread:假设进程 p 中的线程 t_i 创建一个新线程 t_j,线程 t_j 和 t_i 同属进程 p,fork_thread 在实现层的操作语义可以用以下的逻辑公式来描述:

K_THREAD=K_THREAD∪{K_Thread(t_j)}

P_THREAD=P_THREAD∪{P_Thread(t_j)}

F_THREAD=F_THREAD∪{F_Thread(t_j)}

由于线程 t_i 和 t_j 同属进程 p,因此在 PM 的视图中,线程 t_i 和 t_j 共享信号处理,用如下的谓词公式表示 fork_thread 在实现层的关于 PM 的安全约束条件:

t_i SHARE_SIGACTION t_j

同时,在微内核的视图中,线程 t_i 和 t_j 共享虚拟地址空间,用如下的谓词公式表示 fork_thread 在实现层的关于微内核的安全约束条件:

t_i SHARE_VM t_j

类似地,在 FS 的视图中,线程 t_i 和 t_j 共享文件资源,用如下的谓词公式表示 fork_thread 在实现层的关于 FS 的安全约束条件:

t_i SHARE_FILES t_j

(3) exec:在实现层,假设进程 p_i 中的线程 t_j 要求更换进程的执行文件,将 exec 的操作语义细化如下:

K_THREAD=
  K_THREAD\
    \{{K_Thread(t_k)|TCB(t_k). Tid ∈ PCB(p_i). TID}
      {K_Thread(t_j)}}

P_THREAD =
  P_THREAD\
    {{P_Thread(t_k)|TCB(t_k). Tid ∈ PCB(p_i). TID}
      \{P_Thread(t_j)}}

F_THREAD =
  F_THREAD\
    {{F_Thread(t_k)|TCB(t_k). Tid ∈ PCB(p_i). TID}
      \{F_Thread(t_j)}}

PCB(p_i). TID={TCB(t_j). Tid }

PT(p_i)={⟨va, pa⟩ | va ∈ newVM}

PCB(p_i). FD=ϕ

(4) process_exit:假设进程 p_i 退出,首先将进程 p_i 中所有线程退出,然后将进程 p_i 退出,process_exit 在实现层的操作语义细化为:

K_THREAD =
  K_THREAD\
    {K_Thread(t_k)|TCB(t_k). Tid ∈ PCB(p_i). TID}

P_THREAD =
  P_THREAD\
    {P_Thread(t_k)|TCB(t_k). Tid ∈ PCB(p_i). TID}

F_THREAD =
  F_THREAD\
    {F_Thread(t_k)|TCB(t_k). Tid ∈ PCB(p_i). TID}

K_PROCESS=K_PROCESS\{K_Process(p_i)}

P_PROCESS=P_PROCESS\{P_Process(p_i)}

F_PROCESS=F_PROCESS\{F_Process(p_i)}

$PM_{curr}=PM_{curr}\backslash PM(p_i)$

$FreePM=FreePM\cup PM(p_i)$

(5) thread_exit:假设进程 p_i 中的线程 t_j 从系统中退出,thread_exit 在实现层的操作语义细化为:

$K\_THREAD=K\_THREAD\backslash\{K\_Thread(t_j)\}$

$P\_THREAD=P\_THREAD\backslash\{P\_Thread(t_j)\}$

$F\_THREAD=F\_THREAD\backslash\{F\_Thread(t_j)\}$

如果此时进程 p_i 中没有任何线程存在,则进程 p_i 退出:

$K\_PROCESS=K\_PROCESS\backslash\{K\_Process(p_i)\}$

$P\_PROCESS=P\_PROCESS\backslash\{P\_Process(p_i)\}$

$F\_PROCESS=F\_PROCESS\backslash\{F\_Process(p_i)\}$

3.4 VTOS 线程机制的优化层

为了能够通过线程标示符或进程标示符来快速地检索到线程控制块或进程控制块,本文采用了散列表数据结构[11]来快速定位线程控制块或者进程控制块,如图 4、图 5 所示。

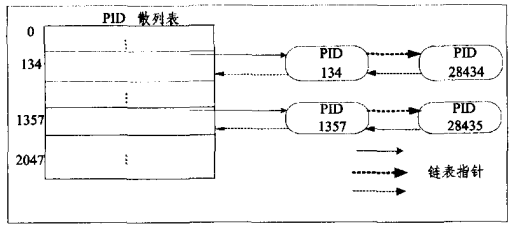

图 4 PID 散列表结构

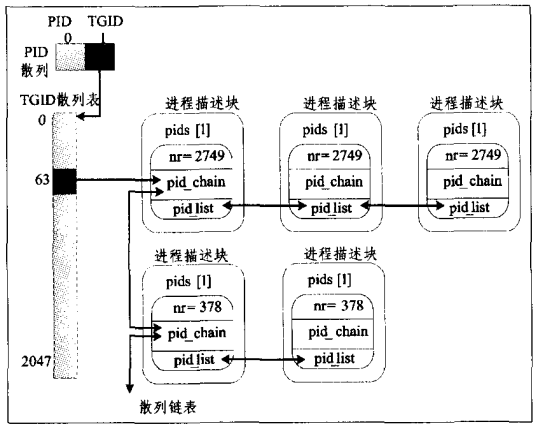

图 5 TGID 散列和链表结构

VTOS 微内核定义了 2 个全局的散列表,分别对应 2 种散列查找类型:进程的标示符(PIDTYPE_PID)和线程组领头进程的标示符(PIDTYPE_TGID)。图 4、图 5 显示了 PIDTYPE_PID 和 PIDTYPE_TGID 两者散列方式。其中 nr 字段代表组标示符,pid_chain 字段用于链接冲突元素,pid_list 字段将所有相同组标示符的进程链接在一起。同时,我们定义任务(task)结构,其中主要包含 pids 字段。pids 是一个数组结构 pid,其成员是 nr、pid_chain 和 pid_list。散列表的链接关系通过 pids 结构完成。

4 VTOS 线程安全机制

在上节 VTOS 线程对象模型的基础上,本节给出 VTOS 线程安全机制。

· 进程空间隔离性

进程之间保持地址空间的隔离性,即各自的页表交集为空,可以用以下的谓词公式表示:

$\forall p_i,p_j\in P_{curr}.\ PT(p_i)\cap PT(p_j)=\phi$

· 线程栈空间隔离性

相同进程中的线程之间保持栈空间的隔离性,可以用以下的谓词公式表示:

$\forall p\in P_{curr},t_i\in TID,t_j\in TID$

$PHasThread(p,t_i)\wedge PHasThread(p,t_j)$

$\rightarrow Stack(t_i)\cap Stack(t_j)=\phi$

其中 PHasThread 谓词表示线程和进程的从属关系,Stack 谓词定义线程的栈空间。

· 线程资源共享性

线程资源主要是指 I/O 资源,如打开文件的描述符,同一进程中的线程共享这些资源:

$\forall p\in P_{curr},t_i\in TID,t_j\in TID$

$PHasThread(p,t_i)\wedge PHasThread(p,t_j)$

$\rightarrow FD(t_i)\doteq FD(t_j)$

其中"A≐B"表示 A 和 B 是相同对象。

· 寄存器隔离性

不同线程的寄存器对象的隔离性:

$\forall t_i\in TID,t_j\in TID.\ \neg\ (TCB(t_i).\ TRegs\doteq TCB(t_j).\ TRegs)$

5 VTOS 内核级线程的关键技术

本节阐述 VTOS 内核级线程机制的关键技术。

5.1 VTOS 线程间通信

VTOS 中执行主体之间的通信功能 IPC 是由内核提供的。虽然名称是"进程间通信 IPC",但实质上通信的双方主要是线程。

VTOS 中的 IPC 机制采用微内核提供的消息转发服务方式。有以下几种 IPC 操作服务:

(1)SEND:调用者向指定线程或进程发送消息;

(2)RECEIVE:调用者从指定线程或任意线程接收消息;

(3)SENDREC:SEND 与 RECEIVE 的原子封装;

(4)NOTIFY:调用者向指定线程或进程发送通知消息。

5.1.1 SEND、RECEIVE 和 SENDREC 服务的设计

这 3 个 IPC 服务都是针对普通消息的。普通消息的发送与接收是同步进行的,也就是说如果一个线程 A 向线程 B 发送一个消息,则必须在线程 B 处于"等待线程 A 的消息"的前提下才能完成消息转发工作。同样地,如果一个线程 A 想从线程 B 接收到消息,则必须在线程 B 处于"发送消息到线程 A"的前提下才能完成消息转发工作。

VTOS 线程机制中,线程在请求 SEND、RECEIVE 服务的过程中,可能被阻塞而进入睡眠状态。线程被阻塞的原因主要有:线程如果无法向指定目标发送消息,则被阻塞,并记其状态中有 SENDING 标志;线程如果无法从指定目标接收到消息,则被阻塞,并记其状态中有 RECEIVING 标志。这两个阻塞的原因说明了在 SEND、RECEIVE 服务中可能会阻塞请求者线程。

SENDREC 服务是将 SEND 与 RECEIVE 两种操作进行原子性封装的结果。假设线程 A 针对线程 B 调用 SENDREC 服务,可能会出现的情况如下:

(1)A 无法向 B 发送出消息,从而被阻塞,所以 A 的状态中有 SENDING 标志。又由于服务的原子性,立即判断出 A 无法从 B 接收到回复消息,因此 A 的状态中又会有 RECEIVING 标志。若 A 的状态中同时有 SENDING 和 RE-

CEIVING 标志，则表示出现这种情况。

(2)A 成功向 B 发送出消息后，如果 A 无法从 B 接收到回复消息，则 A 被阻塞并且其状态中仅有 RECEIVING 标志。

(3)A 成功向 B 发送出消息后，如果 A 能够从 B 接收到回复消息，则 A 不会被阻塞。

5.1.2 NOTIFY 服务的设计

NOTIFY 服务主要针对通知消息(Notification Messages)。VTOS 的线程机制中，当通知消息的发送者是微内核时，消息的内容是"硬件中断号"；当通知消息的发送者是内核任务(System Task)时，消息的内容是"信号 ID"。在线程机制设计上，VTOS 中的系统服务进程均有权限来请求 NOTIFY 服务，并且采用异步模式，通知消息的发送者不会被阻塞，即使通知消息的接收者当时并没有等待发送者的消息。

5.2 VTOS 线程调度

VTOS 的线程调度算法是在多级反馈队列算法的基础上增加了"线程集中"调度策略，尽量减少地址空间的切换，从而提高效率。"线程集中"调度策略的核心思想是尽量优先选择与当前线程同属的进程中的其它线程占有处理器资源，但又不失各线程占用处理器资源的公平性。

VTOS 线程调度多级反馈队列算法的原理是：系统中有 16 个调度队列(Q_1，Q_2，…，Q_{16})，其中各个队列对于处理器的优先级是不一样的。每个队列仍以进程为单位组织，线程的优先级与所属的进程的优先级相同。Q_1 队列的优先级最高，并按照顺序优先级依次减小。Q_1 队列中主要是时钟任务和系统任务，位于 Q_1 中的任何一个线程都要比 Q_2 中的任何一个线程相对于处理器的优先级高，也就是说，Q_1 中的线程一定要比 Q_2 中的线程先被处理器调度，其它的队列依次类推。由于采用"线程集中"的调度策略，处理器在没有比当前线程更高优先级的线程的情况下，首先调度与当前线程同属的进程中的其它线程，如果不存在这样的线程，则调度优先级最高的队列中的线程，若高优先级中队列中已没有调度的线程，则调度次优先级队列中的线程。每个队列的内部遵循时间片轮转法，例如位于队列 Q_n 中有 M 个进程，它们的运行时间是通过 Q_n 这个队列所设定的时间片来确定的，每个进程中的线程时间片的总和等于该进程分得的时间片。同时对于同队列中的进程采用"先来先服务(FCFS)"策略进行调度。VTOS 中这 16 个队列的时间片是不一样的，各个队列的时间片是随着优先级的增加而减少的，也就是说，队列的优先级越高，线程的时间片就越短。VTOS 的线程就绪队列如图 6 所示。

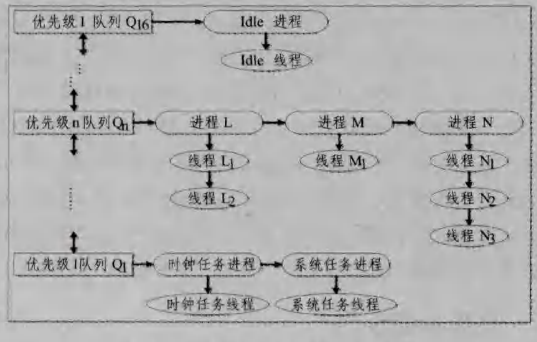

图 6 VTOS 线程就绪队列示意图

5.3 线程间的互斥和同步

同一进程或者不同进程中的多个线程存在同时访问同一个逻辑地址或者同一个对象的情况，为此线程间存在互相干扰的情况，即线程间互斥；同时，线程间存在着相互依赖的情况，如线程 A 的运行需要线程 B 的支持，即存在线程间同步。本文在 VTOS 线程机制中提供一些接口给用户级线程使用，从而能够保证系统中线程间互斥地访问共享对象或同步地进行配合工作。本文在 VTOS 线程机制的互斥和同步方面设计并实现了线程级的 Futex(Fast Userspace Mutex)锁机制[12]，Futex 是一种用户态和内核态相配合的同步机制。同时，本文将 Futex 作为互斥(mutex)和同步(semaphore)的基础。

线程之间的许多同步操作是无竞争的，当一个线程 t 试图进入临界区 A 的时候，并没有其他线程要求进入或已经处于临界区 A。在这种情况下，试图加锁的线程 t 就没有必要调用系统调用，这样可以减少特权级切换的开销。

6 实验和性能分析

本文阐述的线程机制在 VTOS 原型系统上实现。本节阐述对 VTOS 线程机制的实验和性能分析。

6.1 功能验证

测试平台是 Studio XPS 9100 台式电脑，2.8GHz Intel i7 930 处理器，3GB 内存。

通过一个多线程售票程序来验证 VTOS 线程机制的功能。多线程售票程序的逻辑为：主线程初始化总票数 ticks 为 5，然后创建出两个线程来一起对全局量 ticks 进行"读取-判断-改写"操作。图 7 显示了没有使用互斥机制的卖票程序的运行情况。可以看出，程序运行逻辑出现了严重错误。图 8 显示了使用 mutex 互斥字后的卖票程序的运行情况。

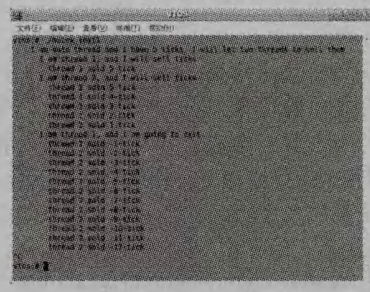

图 7 未使用线程互斥机制的运行情况

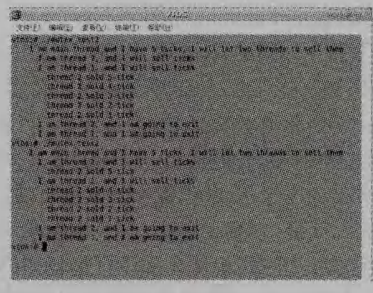

图 8 使用线程互斥机制的运行情况

VTOS 线程机制实现了线程的互斥同步功能。

6.2 性能分析

使用性能测试工具 SysBench[1] 来测试 VTOS 的线程性能，SysBench 是模块化的系统性能基准测试工具。测试平台为 lenovo thinkpad W510 系列，i7 Q720 CPU，4GB 内存。

(下转第 163 页)

1) SysBench. http://sysbench.sourceforge.net/

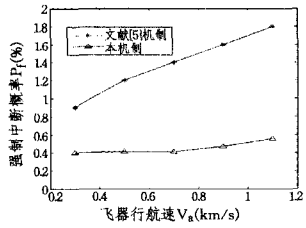

图 7　不同飞行器速度条件下对 P_f 的影响

通过仿真实验可以看出，本文提出的安全切换机制的强制中断概率不受基站通信负载及接入节点飞行速度的影响，能够适应临近空间应用场景。

结束语　本文提出了一种面向临近空间浮空器基站的安全切换机制。通过基于多普勒频移的切换基站预测算法，确定切换基站；通过上下文机制预先将节点认证信息与会话密钥传递给切换基站，避免了切换过程中的认证时延与通信中断。最后，通过性能分析与仿真实验证明了本机制的可靠性。

参 考 文 献

[1] Tomme L C E B, Phil D. The Paradigm Shift to Effects-Based Space: Near-Space as a Combat Space Effects Enabler[R]. Airpower research institute, 2005

[2] 聂万胜，罗世彬，丰松江，等. 近空间飞行器关键技术及其发展趋势分析[J]. 国防科技大学学报，2012，34(2)：107-113

[3] 钱雁斌，陈性元，杜学绘. 临近空间网络安全切换机制研究[J]. 计算机工程与应用，2008，44(15). 18-21

[4] Qian Yan-bin, Chen Xing-yuan, Du Xue-hui. A Security Context Transfer Method for Integrated space network[C]// 2008 International Symposium on Information Science and Engieering. 2008：276-280

[5] 彭长艳. 空间网络安全关键技术研究[D]. 长沙：国防科学技术大学，2010

[6] Papapetrou E, Pavlidou R N. QoS handover management in LEO/MEO satellite systems[J]. IEEE Transactions on Communications, 2003, 46(3)：309-313

[7] Papapetrou E, Pavlidou F-N. Analytic Study of Doppler-based Handover Management in LEO Satellite Systems [J]. IEEE Transactions on Aerospace and Electronic Systems, 2005, 41(3)：830-839

[8] 陈炳才，韩亚萍，郭黎利，等. 低轨卫星网络支持飞机用户的切换管理算法[J]. 计算机应用，2009，29(8)

[9] Loughney J, Nakhjiri M, Perkins C, et al. Context tranfer protocol[M]. Internet-Draft, August 2004

[10] Jin Zheng-ping, Zuo Hui-juan, Du Hong-zhen, et al. An Efficient and Provably-Secure Identity-Based Signcryption Scheme for Multiple PKGs[C] // Proceedings of the International Conference on Computer Science and Information Technology. Singapore, 2008：189-193

(上接第 141 页)

SysBench 多线程测试运行如下：

```
sysbench --test=threads --num-threads=256
    --thread-yields=100 --thread-locks=2 run
```

参数分别表示创建 256 个线程；每个线程运行“加锁-运行-释放锁”过程 100 次；创建互斥锁 2 个。整个运行过程总时间(total time)在多次测试的情况下平均为 13.34s。

VTOS 的线程调度算法在设计上使得同进程中的线程尽量被集中在局部时间内进行调度运行，这在一定程度上可以减少系统运行时的进程切换次数。因为尽可能地使得前后两次在同一处理器上运行的线程属于同一个进程，这样的线程切换并不涉及到页表 CR3 控制寄存器的更改，从而也就避免了一次 TLB 硬件高速缓存刷新，提高了系统的运行效率。

结束语　本文分析了微内核架构多线程机制的研究现状和存在的问题，提出了一个微内核线程对象分层模型，包括基本功效层、实现层和优化层，并采用形式化的方式对各层进行了描述。在此基础上，本文描述了线程安全机制，包括进程空间隔离性、线程栈空间隔离性、线程资源共享和寄存器隔离性。这一对象模型和安全机制的形式化描述将作为后续的安全性形式化验证的基础。同时，本文设计了多线程机制的线程间通信、调度和互斥同步方案。本文描述和设计的微内核架构多线程机制在微内核操作系统 VTOS 上得以实现，并进行了功能和性能测试。实验结果表明，其有效地实现了 VTOS 的多线程机制，并具有很好的系统性能。

接下来的工作计划将是多线程机制的形式化描述转换为后续的安全性形式化验证；同时，将 VTOS 的多线程机制与多核处理有效地结合，研究新的调度算法，并提供灵活的多线程与多核管理工具。

参 考 文 献

[1] Zhou D. Towards the Formal Modeling of a Secure Operating System [C]// Proc. 23rd National Information System Security Conference. 2000

[2] Liedtke J. On μ-Kernel Construction [C]// Proc. 15th ACM symposium on Operating Systems Principles (SOSP'95). 1995：237-250

[3] Heiser G, Elphinstone K, Kuz I, et al. Towards trustworthy computing systems: taking microkernels to the next level [J]. ACM SIGOPS Operating Systems Review, 2007, 41(4)：3-11

[4] CMU CS. Mach Project [EB/OL]. http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html, 2011-06-05

[5] Accetta M, Baron R, Bolosky W, et al. Mach: A New Kernel Foundation for UNIX Development [J]. Computer, 1986, 39(4864)：1-16

[6] L4 Group. The L4 μ-Kernel Family [EB/OL]. http://os.inf.tu-dresden.de/L4/

[7] Elkaduwe D, Klein G, Elphinstone K. Verified Protection Model of the seL4 Microkernel [C]//LNCS 5295. 2008：99-114

[8] Klein G, Andronick J, Elphinstone K, et al. seL4: Formal Verification of an Operating-System Kernel [J]. Communications of the ACM, 2010, 53(6)：107-115

[9] Smullyan R M. First-Order Logic (Dover Books on Mathematics) [M]. Mineola: Dover Publications, 1995

[10] Stallings W. Operating Systems: Internals and Design Principles [M]. New Jersey: Prentice Hall, 2004

[11] Bovet D P, Cesati M. Understanding the Linux Kernel (3rd) [M]. Sebastopol: O'Reilly, 2005

[12] Franke H, Russell R, Kirkwood M. Fuss, futexes and furwocks: Fast Userlevel Locking in Linux [C]// Proc. Ottawa Linux symposium. 2002