分类号 TP3 密级

UDC 编号

中国科学院研究生院

硕士学位论文

基于进程迁移的地址空间复用和可信进程间通信组在 x86 处理器中的研究及实现

李勇

指导教师 杜晓黎 研究员 中科院计算所

Markus Rex 高级工程师 Linux Foundation

申请学位级别 工程硕士 学科专业名称 软件工程

论文提交日期 2008-10-10 论文答辩日期 2008-10-14

培养单位 中国科学院研究生院计算与通信工程学院

学位授予单位 中国科学院研究生院

答辩委员会主席

独 创 性 声 明

本人郑重声明：所提交的学位论文，是本人在指导教师的指导下，独立进行研究工作所取得的研究成果。尽我所知，文中除特别标注和致谢的地方外，学位论文中不包含其他人或集体已经发表或撰写过的研究成果，也不包含为获得中国科学院研究生院或其它教育机构的学位或证书所使用过的材料。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

签名：_____日期：_____

关于学位论文使用授权的说明

本人完全了解中国科学院研究生院有关保管、使用学位论文的规定，其中包括：①学校有权保管、并向有关部门送交学位论文的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存学位论文；③学校可允许学位论文被查阅或借阅；④学校可以学术交流为目的，复制赠送和交换学位论文；⑤学校可以公布学位论文的全部或部分内容。

（涉密的学位论文在解密后应遵守此规定）

签名：_____导师签名：_____ 日期：_____

摘要

Jochen Liedtke 于 1995 年提出通过段切换来实现进程地址空间切换的途径，可以在奔腾处理器上大幅提高微内核消息传递的性能。但十几年后，在面对 64 位 x86 处理器、多处理器和多核处理器的逐渐普及，该方法显露出明显的局限性。基于段切换的方法无法使用在 64 位的 x86 处理器平台上，也没有考虑到多处理器和多核处理器环境中对进程间通信的优化。论文的目标就是突破 Jochen Liedtke 原先工作的局限性，在现代的 64 位 x86 处理器与多处理器多核处理器体系结构中延续地址空间复用的思想，提高微内核进程间通信性能。

论文通过线性地址重定位实现了基于进程迁移的地址空间复用。进程地址在动态链接重定位后，可以迁移到另一个进程的地址空间中，复用内核态或用户态地址空间。这种地址空间复用基于页式内存管理，可以同时应用在 32 位和 64 位的 x86 处理器中。当若干个进程经过充分测试验证不会访问超出自身范围的地址空间后，可以将这些进程迁移到某一个相同的地址空间中，这些进程就组成了一个可信进程间通信组。如果彼此通信的两个或多个进程属于相同的可信进程间通信组，则不需要切换地址空间，即可进行消息传递，从而提高性能。在多处理器或多核处理器环境下，如果调度器尽量使相同可信进程间通信组的进程运行在相同的处理器或处理器核上，就能够通过提高局部处理器缓存命中概率而保持较高的进程间通信性能。

为了在实践中验证基于进程迁移的地址空间复用和可信进程间通信组的可行性和实际效果，论文工作也包括开发一个原型微内核操作系统 MLXOS，并在该操作系统上针对 32 位 x86 处理器架构实现了基于地址空间复用的进程迁移和可信进程间通信组。在对 MLXOS 的实际性能测试中，当采用内存复制的方式来传递消息时，运行在相同处理器的两个进程如果属于同一个可信进程间通信组，进程间通信性能可以提高 15%～19%。

关键词： 操作系统，微内核，进程间通信，地址空间复用，可信进程间通信组

Abstract

In 1995, Jochen Liedtke introduced a method to switch process address space using segment switching. This method is said to improve the performance of message passing on microkernels dramatically on Pentium processors. However, thirteen years later, when 64-bit versions of the x86 processors were introduced, and multi-processor and multi-core processors became more prevalent, limitations of the Jochen's method became obvious. Segment switching based context-switching does not apply to 64-bit x86 processors at all. There was no optimization for inter-process communications on multi-processor or multi-core processors either. So, the goal of this thesis is to propose a method to overcome the limitation of the Jochen Liedtke's method, to spread the idea of multiplexing address space on both 32-bit, and 64-bit x86 processors, and to improve the performance of inter-process communications on microkernels.

The thesis implements an address space multiplexing by process migration and linear address relocation. The process can be migrated into other process' address space, multiplexing kernel or user address space. The multiplexing of address space is based on the paging memory management subsystem, applicable to both 32-bit, and 64-bit x86 processors. If a group of processes are verified by sufficient testing, these processes can be migrated into a single address space, and form a Trusted Processes Inter-process communication Group (TPIG). If inter-process communication happens within the processes of the same TPIG, address space switching can be avoided, and thus improves the performance of message passing. In a multi-processor or multi-core processor environment, the scheduler will try its best to place processes of the same TPIG on the same processor or the same processor core, and as a result, has the ability to maintain high performance of the inter-process communication, and to improve the local processor cache hit rate.

A prototype microkernel operating system, MLXOS, was written to proof the feasibility, and the performance of process migration based address space multiplexing, and TPIG, on a 32-bit x86 processor. According to the benchmarks performed, when performing message passing via a memory copy, inter-process communication between

two processes of the same processor of the same TPIG, executing on the same processor, can be improved by 15% ~ 19%.

Key Words: operating system, microkernel, inter-process communication, address space multiplexing, trusted process inter-process-communication group

目录

第一章　绪论

1.1 研究背景与意义

1.1.1 操作系统架构的宏内核与微内核之争

操作系统内核是整个计算机操作系统最为核心的程序，它控制应用程序的执行，并作为硬件抽象层为应用程序提供访问系统资源的抽象接口。为了使内核能够安全的为多个应用程序提供资源共享，需要从硬件级别上提供一定的保护机制。

硬件必须至少提供内核模式和用户模式两种保护级别。在内核模式下，程序可以访问所有硬件资源；在用户模式下，程序无法执行某些特权指令，也无法访问所有的硬件。因此根据保护级别的不同，程序的运行空间被划分为两部分：内核空间和用户空间。操作系统的内核运行在内核空间，普通应用程序运行在用户空间。

传统的 UNIX 系统内核是宏内核架构，即操作系统的进程管理、内存管理、网络、文件系统、设备驱动程序都运行在内核空间，应用程序通过系统调用来访问这些系统服务。

微内核是在宏内核之后出现的架构，主要思想是仅仅在内核空间运行必不可少的功能，譬如进程管理、进程间通信、内存管理和设备管理的基础部分，其它系统功能，诸如完善的进程调度、内存管理、驱动程序、文件系统、网络协议等，都运行在用户空间。

微内核系统根据提供系统服务进程的数量又可分为单服务进程系统和多服务进程系统。单服务进程系统就是仅由一个运行在用户空间的服务进程完成进程调度、内存管理、驱动程序、文件系统、网络协议等系统功能；多服务进程系统就是将上述系统功能划分由多个不同的用户空间服务进程协同完成。

无论是单服务进程系统，还是多服务进程系统，当应用程序调用系统服务时，都存在服务进程和应用程序之间的进程间通信。尤其是在多服务进程系统中，在多个服务进程之间存在频繁的进程间通信。

宏内核架构具有较高的性能，系统调用进入内核后，内核各功能模块之间的通信都是通过函数调用完成的，额外的进程间通信开销极少。宏内核的缺点则是由于大量的驱动程序、协议栈代码运行在内核空间，一旦出现错误，很容

易导致内核崩溃从而使整个系统无法工作。此外，调试内核空间程序的过程本身也很繁琐，难以提高程序开发效率。

微内核架构的系统服务以用户空间进程的形式运行在受保护的独立地址空间内，一旦某个功能模块发生错误，不会破坏其它功能模块的正常工作，整个系统还能继续运行。此外在微内核下调试驱动程序或者协议栈时，可以使用大量已有的用户空间程序调试工具，可以显著提高程序开发效率。而微内核的主要缺点在于系统的功能模块都是用户空间进程，多个功能模块之间的交互必须通过进程间通信完成，通信开销导致系统性能相对宏内核较低。

宏内核与微内核哪一个更为先进，近 20 年来一直都是争论热点。现在比较普遍的观点是，宏内核与微内核根据应用环境和需求目标的不同，各有优劣。

1.1.2 微内核架构的优缺点

微内核是一种中立于各种操作系统的高度模块化的抽象集合，基于该抽象集合之上可以构建各种操作系统服务。目前比较普遍接受的抽象包括了：任务、线程、内存对象、消息和端口。这些抽象提供了管理虚拟内存、调度和进程间通信的机制，基于这些机制可以实现更为复杂的系统服务。

微内核提供的抽象运行在内核空间内，其它服务都可以运行在独立的用户态地址空间内，这种设计具有如下优势：

系统整体更加健壮

如果某一个服务出现了错误，只需要重新起动该服务即可，不会导致其它服务错误，也不需要重新启动整个系统。运行在内核空间的代码量大大减少，内核代码更容易被人理解，更容易进行代码审查，遗漏致命错误的几率更低。

运行环境更加安全

可以配置特定的服务访问特定的系统设备，这样即使某一个服务进程被恶意控制，入侵者也无法彻底控制整个系统。

系统服务可配置

服务可以在线修改、加载而无须重新启动整个系统。

核心子系统开发更便捷

调试内核空间代码必须使用特定的硬件设备和调试软件，并且可以提供的调试功能也非常有限。当服务运行在用户空间时，可以像调试普通用户态程序那样，使用丰富的功能成熟的调试工具，编写程序更加高效、便捷。

常驻内存代码减少

除微内核本身和极少数服务外，多数运行在用户空间的服务进程都不需要常驻内存。因此相比宏内核而言，微内核架构常驻内存量大大减少。

更容易实现高性能实时系统

减少了在内核中关闭中断的几率，实时进程可以比其它系统进程获得更优先的处理机会，更容易实现高性能的实时处理。

支持多处理器/多核更简洁

传统的宏内核在支持多处理器/多核架构时，需要在众多内核线程之间维护大量的互斥和同步设施，这是非常复杂和难于调试的。微内核可以将众多的互斥同步机制实现在用户态服务进程之间，内核中的多线程互斥同步数量大为减少，内核代码的处理更为简洁。

内核复杂度降低

系统服务被划分到各自独立的进程中实现，模块化程度更高。和硬件相关的接口也可以独立实现在不同的服务中。这种模块化设计在硬件和协议栈支持数量增加的情况下，相对于宏内核而言，在整个系统代码复杂度急剧增加的情况下更容易管理整个系统的质量和规模。

从上世纪八十年代，世界各地众多大学和研究院所展开了对微内核架构操作系统的研究热潮。其中比较著名的系统有美国卡耐基梅隆大学的 Mach，GNU 组织的 GNU Hurd，以及荷兰 Vriji 大学的 Minix 系统等。尤其 Mach 系统做了大量微内核系统的开创性工作，学术界和工业界有很多研究开发工作都是在 Mach 操作系统基础上开展的。进入九十年代后人们普遍认为微内核架构的主要缺点在于进程间通信性能相比较于宏内核而言较低。在早期 Mach 3.0 上的 UNIX 单进程服务系统，性能比直接运行 UNIX 要慢 10%，而对于更新的硬件平台，性能损失接近 50% [17]。

1.1.3 提高微内核进程间通信性能的意义

由于微内核架构与生俱来的安全性和稳定性，虽然它的进程间通信性能相对宏内核而言较低，但一直都是系统软件领域的研究热点，人们提出了大量的方法来改进微内核通信效率，并取得了不错的效果。每一次对提高微内核进程间通信性能的尝试，对于促进微内核架构的发展都是非常有意义的。本论文的

目的，就是进一步改进前人提出的方法，尝试在新的硬件环境中提高微内核进程间通信的性能。

1.2 本课题的研究进展

在最初微内核架构被提出时，进程间通信性能就是研究的热点。第一代微内核的原型系统多多少少保留有最初宏内核架构的痕迹，并且非常注重内核代码的可移植性。但可移植性的代价就是无法在不同的硬件平台上进行针对性的特殊优化，并且“支持不同硬件”也产生了大量的冗余代码。

改进第一代微内核进程间通信性能的大量研究工作和工程实践，主要思路多集中在：

- 使用共享内存来传递消息，减少内存复制。
- 将服务进程运行在内核空间，减少进程调度开销。
- 使用寄存器来传递短消息，降低访存几率。
- 针对消息传递的发送、接收者优化进程调度。

比较典型的工程实践有 Mach、Hurd、Minix、Chorus 等微内核架构操作系统项目。这些系统都具有较好的可移植、可配置性，但进程间通信性能比宏内核至少要慢 10% [1]，而在实际运行中性能更低。

也有人提出在 64 位处理器上使用单一地址空间，即系统中所有的进程和内核共同一个 64 位的巨大地址空间 [16]，国内也有类似的研究工作 [7]。但由于在单一地址空间下，所有的进程和内核运行在不同的线性地址范围，实现程序动态链接和采用虚拟化技术时会遇到很多繁琐而困难的细节问题，因此在 2000 年后该方向上的研究工作就不那么活跃了。

开创性的工作是由 Jochen Liedtke 发起的，他认为第一代微内核架构进程间通信性能较低，并不是微内核思想的问题，而是微内核开发中的不良设计导致。正是由于在设计时没有仔细考虑影响微内核进程间通信的实质原因，到导致微内核进程间通信性能明显低于宏内核。

1.2.1 影响微内核进程间通信性能的原因

过多的系统调用

在主频 2992 MHz 的 Intel Core Duo 处理器上，对 2.6.24 版本 Linux 内核运行 getpid 系统调用，需要 394.7 纳秒。getpid 系统调用只是从内存中将当前

进程 pid 返回给调用者，这个系统调用的时间可以几乎看作是进入和退出系统调用的时间。在主频 2992MHz 的处理器上，394.7 纳秒意味着即使什么工作都不做，处理器也必须消耗 1180 个指令周期。由此可见系统调用的开销是非常大的。微内核的模块块间消息传递越频繁，导致的系统调用就越多，相应的系统开销就越大。

消息复制

传统微内核进程间通信的消息传递需要在不同地址空间之间传递消息，消息的复制过程是：进程 A 地址空间→内核地址空间→服务进程 B 地址空间，消息需要被复制 2 次。而在宏内核中消息只需要从进程地址空间复制到内核地址空间即可，内核功能模块之间的数据传递可以直接通过栈或地址引用来实现。

进程调度代码未优化

由于微内核的进程调度可能比宏内核更为频繁，所以有必要对内核的进程调度代码进行进一步的优化。传统微内核的进程调度代码，还有进一步优化的提升空间。

进程调度的策略未充分优化

目前常见的优化是，一旦消息发送者向内核提交了消息，内核会立刻将消息接收者调入处理器运行。这种方法的确提高了两个进程之间的通信性能。但是，当多个进程试图与一个接收者通信时，这个策略有可能使其它发起通信的进程必须等待前一个进程完成通信并释放处理器后，才能继续进行通信。这种粗粒度的进程调度策略无法满足整个系统进程间通信的公平性和整体性能。

通信接口未充分优化

由于消息中可能包含多种数据类型，因此接收方接收到消息数据后，首先要进行必要的类型检查。如果内核可以将某个进程要接收的消息按照数据类型分组，那么相应的类型检查、消息分析就可以大大简化。

参数传递不灵活

传统微内核为了强调兼容性，在参数传递的实现上灵活度不够。譬如，在通用寄存器较多的平台下（多数 RISC 处理器），完全可以将短消息数据直接放在寄存器中传输。但为了兼顾寄存器较少的平台（譬如 i386，只有 4 个通用寄存器），参数传递只有少数直接通过寄存器进行，多数仍然要通过内存复制。

过多的参数检查

传统微内核中多个服务进程之间的参数检查有很多是冗余和重复的。这些参数检查不仅增加了执行指令和访问内存的数量，由参数检查发起的跳转指令还有可能刷新处理器流水线，从而导致进程间通信的速度更慢。

TLB[①] 失效

在诸如 32 位或 64 位 x86 处理器上，TLB 项是没有标签的，因此多个进程的 TLB 项无法同时存在于 TLB 中。当地址空间切换时，必须将整个 TLB 刷新。现在的 32 位和 64 位 x86 处理器支持全局页，标志为全局页的对应 TLB 项在地址空间切换时不会被刷新，所以多数系统会将内核页设置为全局页。但是这样只能避免地址空间切换时不会刷新内核对应的 TLB 项，重新加载用户态地址空间 TLB 项的代价仍然是非常昂贵的。尤其是当多个服务进程彼此切换地址空间时，TLB 失效导致的开销远比宏内核大的多。

1.2.2 第二代微内核架构思想

要从根本上提高微内核进程间通信的性能，就要摈弃原先从宏内核设计中引入的不利因素，重新设计微内核架构。因此 Jochen Liedtke 提出了第二代微内核架构的思想，首次概括出提高微内核进程间通信性能的设计原则 [19]：

- 进程间通信的性能是首要问题。
- 所有的设计都需要针对性能进行讨论。
- 如果某部分性能很低，就要考虑新技术。
- 所有的模块要协同起来考虑性能。
- 设计时要考虑整个系统的各个层面，从体系结构一直到编码。
- 设计要有稳固的基线，哪些模块是依赖于特定硬件的，哪些模块是通用的，这些都要明确。
- 设计一定要制订确定的性能目标。

在他看来，QNX、L4 都属于采用新设计思想的第二代微内核架构，微内核进程间通信性能都得到了显著提升。第二代微内核的研究者们提出，要获得更好的进程间通信性能，首先要放弃微内核的可移植性，必须针对特定的硬件来优化微内核架构，有时候甚至要重新设计整个内核。但应当保持通信接口的一致

[①] TLB 全称为 Translate Look-aside Buffer，也称为旁视缓冲器。它是处理器的内存管理单元在页式内存管理模式下用来保存线性地址到物理地址映射的缓存。附录 B 对 TLB 做了概要的介绍，更详细的内容请参看 [28，29]。

性，这样可以确保用户程序在代码级的兼容性。在这个基础上，如下的改进措施逐渐得到了大家的共识[①]：

- 添加新的系统调用
- 丰富消息的结构
- 通过临时的地址空间映射避免额外的消息复制
- 减少进程调度的开销
- 优化进程调度策略
- 采用虚拟寄存器技术
- 避免不必要的检查
- 减少 TLB 失败

更多的改进措施可以在［19］中看到。当然这些措施并不是已知的所有改进。有一些更为激进的设计，诸如 exokernel[②] [10，13]，由于不属于本论文课题要讨论的范围，因此不再赘述。

在这些改进措施中，有一个提高 TLB 命中率的方法非常精妙。该方法由 Jochen Liedtke 于 1995 年提出，主要思想是在进程切换时，用段切换代替页表切换，并通过复用用户态地址空间，来尽可能避免进程切换时的 TLB 刷新，从而提高 TLB 命中概率。本课题就是在该方法的基础上所作的进一步改进工作。因此首先这里将这个方法进行概要介绍。

1.2.2 通过复用用户态地址空间改进消息传递性能

目前 Intel x86 和 x86_64 处理器的 TLB 都是没有标签的，任何一个时刻，TLB 中所有表项都只能确定的属于一个进程的地址空间。在进程切换时，必须将 TLB 全部清空。当新的进程访问内存的时候，会根据访存区域再次加载 TLB。以下是在多种 Intel 处理器上测试访存时数据 TLB 命中和失效的性能数据，单位是处理器周期：

表 1-1 TLB 失效数据统计

处理器型号	TLB 失效	TLB 命中	代价

[①]这些观点由 Jochen Liedtke 提出，详见［19］中 5.6 节 Summary of Techniques。

[②] Exokernel 思想最早由 MIT 在 1994 年提出并开发了原型，这是一个小尺寸、针对特定硬件的全新内核系统。它的设计者认为受保护的系统资源开销较大而且缺少灵活性，系统应该在一种安全的方法下直接复用裸硬件，这样才能达到最高的性能。

Intel(R) Pentium(R) M processor 1.86GHz	145	48	97
Intel(R) Pentium(R) M processor 1700MHz	152	47	105
Intel(R) Core(TM)2 Duo CPU E6850 @ 3.00GHz	468	101	367

由于内存访问速度的提高远不如处理器速度提高的快，因此处理器的速度越快，加载 TLB 项时损失的指令周期就越多。从上述数据可以看出，在 3.0GHz 处理器上加载一个 TLB 带来的开销是在 1.86GHz 处理器上开销的 3.2 倍。如果当前进程的工作区有 8 个页的话，在 3.0GHz 处理器上总计加载对应的 8 个 TLB 项需要额外的 2936 个指令周期，约等于 1 微秒。在相同的机器上，这个时间足够 Linux 完成 4 次 getpid 系统调用了。因此如果能够在地址空间切换时避免不必要的 TLB 刷新，则可以大幅度降低加载 TLB 的开销，从而提高进程间通信时消息传递的性能。

针对奔腾处理器，Jochen Liedtke 提出了透明复用用户地址空间的方法，来提高地址空间切换的性能 [11]。

在奔腾处理器上，通常进程会将操作系统内核也映射到自己的地址空间中来。以 Linux 为例，32 位平台下，在进程的 4GB 线性地址空间中，通常划分出 3GB 供进程的用户态部分使用，称为用户态地址空间，余下的 1GB 供进程的内核态部分使用，称为内核态地址空间。

图 1-1 传统 4G 地址空间划分

对于运行在用户地址空间的系统服务，譬如某个协议栈，或者驱动程序，本身的工作集是非常小的，在整个进程寿命周期中，所能用到的地址空间远远小于划分给普通应用程序的 3GB。多数服务进程在整个生命周期中用到的最大的地址空间不超过 1MB，极少有服务进程能用到超过 64MB 的地址空间。因此对于这些服务进程，无需将它们放置在 3GB 的大地址空间中，而可以将它们限定在最大不超过 64MB 的小地址空间里。

由于 32 位 x86 处理器[①]可以通过设置段基址来确定段式映射产生的线性地址，并可以通过设置段长度限制防止访问超过段长度的数据或代码。因此通过

① 附录 A 描述了关于 32 位 x86 处理器保护模式的基本内容，更详细的内容可以参看 [30]。

适当的设置小地址空间对应的段基址和段长度限制，就可以将这段小地址空间放置在起始地址非 0 的线性地址中。如果将这段线性地址空间对应的全局页目录和页表项在所有大地址空间的进程之间共享，则这段小地址空间就可以映射到其它大地址空间进程的地址空间中。

譬如，可以将原先 3G 的普通进程地址空间缩小为 2.5GB，从 2.5GB 到 3GB 的地址空间专门用来放置小地址空间。小地址空间的大小可以从 1MB 开始到 64MB 不等。如果某个系统服务所需的地址空间不得不超过 64MB，那也可以将它放置在大地址空间中运行。

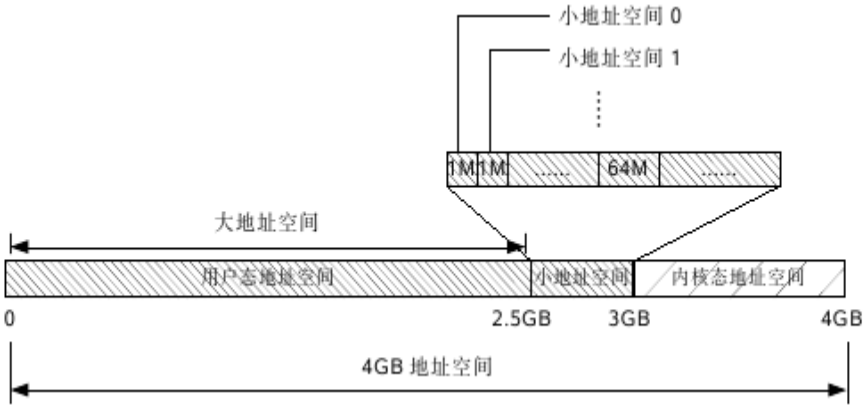


图 1-2 带有小地址空间的 4G 地址空间划分

当一个大地址空间的进程要切换到小地址空间的进程时，只需要切换段寄存器到小地址空间即可，不需要更换整个页式地址空间。同样，当小地址空间进程之间进行切换时，也只需要进行段寄存器的切换即可。当一个小地址空间进程要切换到大地址空间进程时，如果该大地址空间进程和上一次切换到小地址空间的大地址空间进程相同，那么也仅仅切换段寄存器即可，不需要将整个 TLB 刷新。这样一来，当在普通应用程序和系统服务进程之间因为传递消息而切换进程和地址空间时，就可以降低因刷新 TLB 而重新加载 TLB 表项的开销[①]。根据 Jochen Liedtke 的性能基准测试，当进程的工作集大小为 2 页时，基于段的

[①]在 Pentium Pro 处理器后，在页目录和页表项中引入了 Global 标记。被标记为 Global 的页表在刷新 TLB 时不会被刷新掉。这样一来，如果将所有小地址空间对应的页表都设置为全局页表，那么所有从小地址空间进程切换到大地址空间进程的情况下也都不需要刷新 TLB 了。这样一来，只有大地址空间进程之间切换的时候才，需要执行刷新 TLB 的操作。

地址空间切换比基于页的地址空间切换性能提高 1 倍；当进程工作集大小为 64 页时，基于段的地址空间切换比基于页的地址空间切换性能提高 2.5 倍。

1.2.2 硬件的发展带来新的挑战

距离 Jochen Liedtke 第一次提出复用用户态地址空间思想到今天，已经过去 12 年了。在这 12 年中，软件和硬件都有巨大的发展，且整个计算机系统的应用模式和 12 年前相比也更为复杂。通过复用用户地址空间来提高地址空间切换性能的方法面临着新的挑战，而这些挑战是最初设计者所无法预料的。

64 位处理器的普及

目前各大通用处理器厂商均推出了 64 位的 x86 处理器产品，通常被称作 x86_64 或 64 位 x86 处理器。在 x86 架构的服务器和桌面个人电脑领域，64 位的 x86 处理器将逐渐成为主要平台，尤其是多核 64 位 x86 处理器。而传统的 32 位 x86 处理器则仍然会继续应用在 x86 架构的低端入门级产品中。

在 64 位 x86 处理器中，内存管理的基本原理同 32 位处理器是相近的，但是在诸如段式内存管理和保护方面有重要差别。在 32 位处理器中，线性地址是由段基地址和段内偏移量之和决定的。段基地址可以灵活的设置于 4GB 地址空间内任意 4 字节对齐的位置上。因此二进制程序中，相同的段内地址可以根据不同的段基地址得到不同的线性地址，如下图所示。这也是透明复用用户地址空间方法的根本。

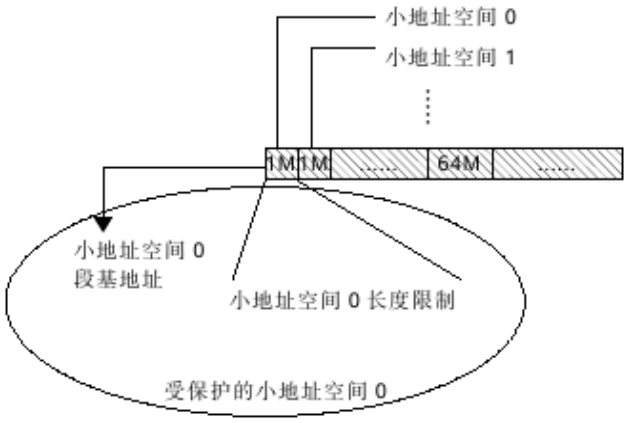


图 1-3 受保护的小地址空间

但是这种方法在 64 位 x86 处理器上不再适用[①]。当处理器运行在完全的 64 位模式下时，段式寻址功能基本上（但不完全）被禁止了，即基于段基址和段长度限制的寻址、保护基址都不再有效。

在完全 64 位模式下，处理器会将 CS，DS，ES，SS[②]段寄存器基址直接当作 0 来对待，并创建 64 位的线性地址空间，而程序中的有效地址就是线性地址。FS 和 GS 寄存器是例外，这两个寄存器可以在线性地址计算时当作额外的基址寄存器来使用。这两个寄存器的功能是寻址局部数据或某些操作系统相关数据结构。并且在 64 位模式下，处理器不再进行段长度限制的保护检查。处理器仅仅检查地址格式是否合理，而不再进行段长度检查。在不同模式之间切换时，不会改变段寄存器或相应描述符的内容。同样在执行 64 位模式时，它们的内容也不会改变，除非显式的加载不同的段属性。为了兼容应用程序的模式，对段的加载指令（MOV，POP）仍然是可以执行的，并且相应的段属性也会被加载到段寄存器的不可见部分。但是多数属性在寻址时都会被忽略掉，只有优先级信息会被使用到。

简单的说，就是在 64 位模式下，段式保护中除了优先级（ring0/1/2/3）仍然被保留外，其它功能都在处理器的寻址和保护过程中被忽略了。除了优先级之外，其余的保护均通过页式寻址和保护机制实现。

因此，Jochen Liedtke 提出的透明复用用户地址空间来提高地址空间切换的方法，在 64 位 x86 处理器模式下将不再适用。

多核处理器和多处理器平台

由于随着处理器主频的不断提高，功耗和散热的问题无法彻底克服，现在的处理器厂商已经开始向多处理器和多核处理器的方向来开发自己的产品了。现在的趋势是，在一台计算机上，会有多个处理器，在每一个处理器封装中，又会有多个处理器核，而每个处理器核内也可能存在多个逻辑处理器。这种情况，普通进程和服务进程在任何一个逻辑处理器上运行都是有可能的。如果进程间通信的两个进程运行在不同的逻辑处理器上，消息传递的性能会根据两个逻辑处理器关联关系的不同而差别巨大。

[①]此处关于 x86_64 处理器寻址模式的内容，主要参考和引用自［30］，这里为部分内容的中文翻译。
[②]CS 为代码段寄存器，DS 为数据段寄存器，ES 为扩展段寄存器，SS 为栈段寄存器。

如果一个 Intel 处理器使用了超线程（Hyper-Threading）[①]技术，则该处理器中的两个逻辑处理器具有各自独立的机器状态（寄存器组），但共享处理器算术逻辑单元 ALU、数据和指令的 cache，如下图 1-4（a）所示。

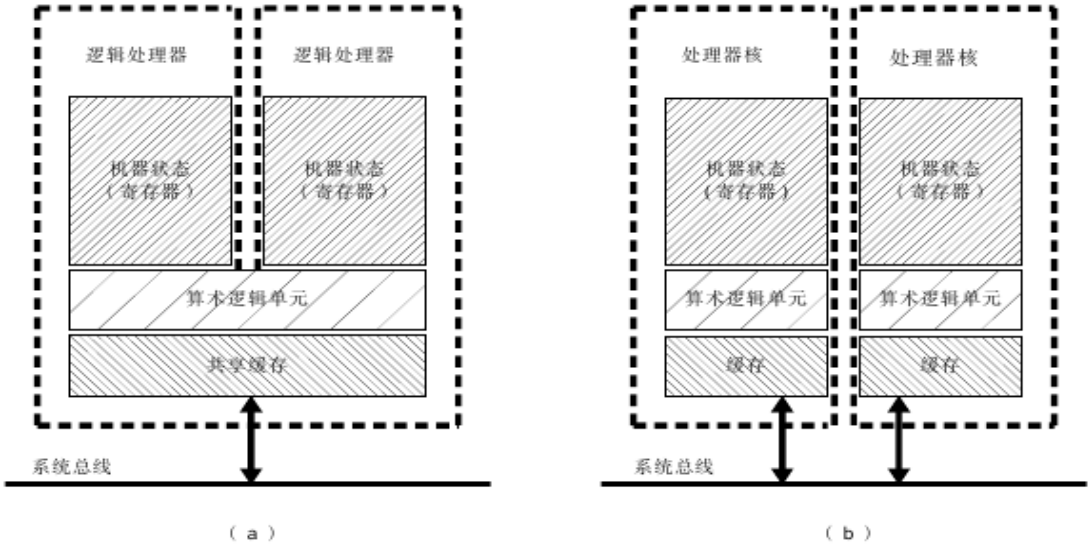


图 1-4 Hyper-Threading（a）和双核（b）处理器架构

对于 Hyper-Threading 架构，当相同的内存数据由不同的逻辑处理器来处理时，由于逻辑处理器共享 cache，所以不会发生 cache 失败。因此当进程调度到不同的逻辑处理器上时，不会因为 cache 失效而影响进程运行的性能。

而多核处理器的架构与 Hyper-Threading 有所不同，两个处理器核都具有自己的算术逻辑单元 ALU 和 cache，如上图 1-4（b）所示的双核处理器。在多核处理器的不同核之间，由于不共享 cache，当数据从一个核调度到另外一个核执行时，首先会发生 cache 失败，当数据从内存中传递到二级和一级 cache 后，该处理器核才能继续运行[②]。

[①]超线程（Hyper-Threading）技术基于这样的事实：当一个指令被处理器执行时，并不会占用所有的处理器部件，那些没有被占用的部件就有可能被并行利用起来。在超线程处理器中增加了逻辑处理单元，可以尽可能的在一条指令被处理器执行的同时，用剩余的空闲部件来执行另外一条指令。这样在软件看来，似乎是运行在两个处理器上，但实际达不到两个独立处理器的性能。

[②]目前已有在多核处理器间共享 L2 cache 的处理器产品，但 L1 cache 在不同处理器核间是不共享的，因此导致的性能损失仍然是存在的。

因此,在共享 cache 的两个逻辑处理器之间传递数据,性能要高于独立 cache 的两个处理器核。如果计算机系统中存在多个处理器封装（socket），那么在不同的封装之间传递数据还要经过封装之外的一系列电气连接，速度要比在同一个封装内传递数据慢。因此在同一个封装内不同处理器核的 cache 之间传递数据速度要高于不同封装处理器核的 cache 之间传递数据的速度。

因此为了提高进程在超线程、多处理器或多核情况下的进程间通信性能，还应当考虑到：

- 对于传递消息的若干个进程：在不影响系统整体性能的前提下，首先尝试使它们处于相同的逻辑处理器中，其次尝试使它们处于相同的处理器核中，然后再尝试使它们处于相同的处理器封装内。
- 对于某个特定的进程，当再次给它分配处理器运行时，应当尽量分配上一次运行过的处理器给它。

为了尽量避免服务进程在多个逻辑处理器之间迁移带来的开销，可行的办法之一就是将服务进程多线程化，然后在每一个逻辑处理器上运行一个线程实例，专门负责所在逻辑处理器上与之相关的通信。微内核下的一些设备驱动程序在响应内核发送的硬件中断消息时经常会采用这种优化方法。如果可以预见某些进程彼此会非常频繁的通信，譬如某个应用程序的一组相关进程，还可以将这些进程组成一组，尽量调度在相同的处理器上运行。这些优化方法 Jochen Liedtke 在最初提出复用用户地址空间时也没有考虑过。

1.2 本文主要研究内容

本文的研究目标是试图解决 Jochen Liedtke 提出的复用用户态地址空间方法随着硬件的发展所面临的新问题：

- 是否可以改进 Jochen Liedtke 的“小地址空间”的方法，使得复用地址空间的思想可以同时运用在 32 位和 64 位 x86 上。
- 是否有可能为进程间通信频繁的多个进程之间复用地址空间，在多核处理器进程调度策略的基础上进一步提升微内核进程间通信的性能。

论文工作就是围绕上述两个问题，提出解决办法和具体实现，并根据实际性能测试来评价论文提出方法的可行性和实际效果。

论文主要内容是研究并实现了基于进程迁移的地址空间复用和可信进程间通信组。为了在实践中验证基于进程迁移的地址空间复用和可信进程间通信组的可性能和实际效果，论文工作同时也开发了一个原型微内核操作系统 MLXOS，并在该操作系统上针对 32 位 x86 处理器架构实现了论文所研究的主要内容。MLXOS 中运行的进程可以通过线性地址重定位实现基于进程迁移的地址空间复用。进程地址在地址动态重定位后，可以迁移到另一个进程的地址空间中，复用内核态或用户态地址空间。这种地址空间复用基于页式内存管理，可以同时应用在 32 位和 64 位的 x86 处理器中。当若干个进程经过充分测试验证不会访问超出自身范围的地址空间后，可以将这些进程迁移到某一个相同的地址空间中，这些进程就组成了一个可信进程间通信组。如果彼此通信的两个或多个进程属于相同的可信进程间通信组，则不需要切换地址空间，从而可以提高消息传递的性能。在多处理器或多核处理器环境下，如果调度器尽量将相同可信进程间通信组的进程运行在相同的处理器或处理器核上，就能够通过提高局部处理器缓存命中概率而保持较高的进程间通信性能。在对 MLXOS 的实际性能测试中，当采用内存复制的方式来传递消息时，运行在相同处理器的两个进程如果属于同一个可信进程间通信组，进程间通信性能可以提高 15%～19%。

论文组织结构：

第一章 绪论。本章简要介绍本论文的研究背景、当前相关领域的研究现状、研究目标和主要研究内容。

第二章 基于进程迁移的地址空间复用。本章阐述了基于进程迁移的地址空间复用方法的概念和原理。

第三章 可信进程间通信组。本章在前一章基础上阐述了论文提出的可信进程间通信组的概念和原理。

第四章 设计与实现。本章以 MLXOS 原型操作系统为实例，阐述论文是如何设计并编码实现了基于进程迁移的地址空间复用和可信进程间通信组，包括关键数据结构和伪代码示例说明。

第五章 性能测试与评估。本章介绍了实际性能测试的方案、方法，并结合性能测试结果对论文研究内容的实际效果进行评估。

第六章 结论与展望。本章主要介绍了本课题研究结论和下一步工作展望。

第二章　基于进程迁移的地址空间复用

由于 64 位 x86 处理器不支持基于段的地址重定位，因此要复用地址空间，必须采用 64 位 x86 处理器支持的基于页式内存管理的地址重定位。32 位和 64 位 x86 都支持基于页的内存寻址模式，因此基于页的地址重定位可以适用于 32 位和 64 位的 x86 处理器。

需要说明的是，为了在有限的时间内快速开发实现验证用的原型操作系统 MLXOS，并避免同时维护两种处理器代码所带来的额外工作量和复杂度，论文工作只在 32 位 x86 处理器上实现了基于进程迁移的地址空间复用。对于 64 位 x86 处理器的代码实现将会在 MLXOS 后续开发工作中进行，但在本文中不会涉及。

2.1 地址空间复用策略

地址空间复用并不是新名词，该技术早已被众多微内核或者宏内核操作系统所采用。如下图 2-1（a），是传统 32 位 x86 处理器上 Linux 操作系统复用 4G 字节地址空间的策略。

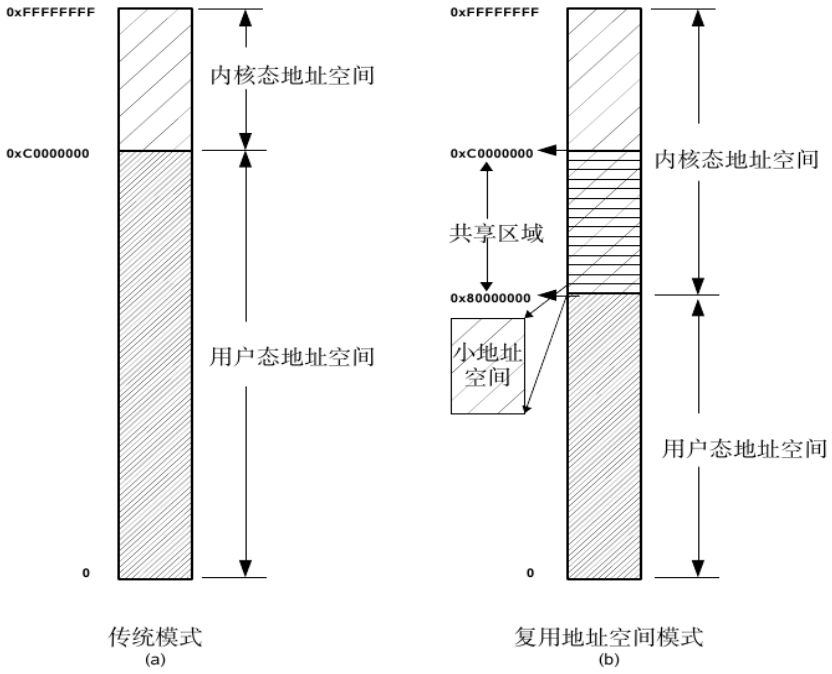


图 2-1 传统地址空间模式和复用地址空间模式

其中最高的 1G 地址空间（0xC0000000 ~ 0xFFFFFFFF）划分给了内核，所有应用程序 4G 地址空间的最高 1G 都会共享这段内核地址空间，进程的页全局目录表中对应 3G 到 4G 的表项内容是相同的，而对应于内核空间的页表，均标识为全局页。该策略的好处是，在进程切换后，不需刷新内核空间对应的 TLB 表项，有利于在系统调用中提高访问内核代码和数据的速度。

论文提出的地址空间复用策略，在 4G 地址空间中引入了一个新的区域。如上图 2-1（b），在原型系统 MLXOS 中，该区域位于 4G 地址空间的 2G 到 3G 区域，称为地址空间共享区域。在原型系统中，共享区域的地址空间具有内核优先级，并将对应的页表标识为全局页。因此普通优先级的用户态进程无法访问和修改该区域内的内容，并且在进程切换时，TLB 中对应于该区域的表项不会被刷新掉。这样一来，除了映射给系统内核的区域之外，共享区域也可以在进程地址空间中复用。

对于多数进程，尤其是服务进程，在整个生命周期内占用和访问过的地址空间远远达不到 2G 或者 1G 字节，甚至不会超过 512K 字节。因此 1G 字节的共享区域可以分成多个小地址空间供多个进程使用。共享区域可以划分为从 512K 字节到 64M 字节不等的多个小地址空间，可以根据实际迁移的进程的需求，将其重定位到对应大小的小地址空间中。在原型系统 MLXOS 中，为了简化起见，每个小地址空间的大小统一设定为 4M 字节，如图 2-1（b）所示。

系统内核的内存管理模块需要将服务进程在小地址空间中访问的页都设置为全局，这样在进程切换时，服务进程地址空间对应的 TLB 表项不会被刷新掉。同样系统内核的内存管理模块还要将小地址空间对应的页设置为系统权限，这样服务进程的内容就不会被普通进程所修改。

2.2 运行时的动态进程迁移

由于服务进程在迁移到共享区域的小地址空间后，将具有与内核相同的最高优先级（在 x86 处理器中称为 ring0 级别，见附录 A），必须确保服务进程的代码足够稳定和可信，才能以内核权限运行。未经充分测试的服务进程应当运行在独立的用户态地址空间中，以避免由于软件错误破坏内核或其它运行在小地址空间中的服务进程。

当一个服务进程在用户权限的独立地址空间中经过充分测试和验证后，就可以在共享区域的小地址空间中以内核权限运行。很可能因为某些原因，不允许将服务进程停止，或者重新编译、配置和启动内核来将服务进程重定位到小地址空间运行，例如：

- 系统正在运行重要业务，不允许更改内核或重新引导系统。
- 当前服务进程正在工作，不能停止。

同样当共享地址空间无法容纳新的服务进程时，如果必要还需要将已经运行在小地址空间中的服务进程移出，将更重要的服务进程移入共享区域中的小地址空间。如果这种移动需要停止所有运行在共享区域的服务进程，或者需要重新启动系统，那都是无法接受的。

因此，将服务进程移入或移出共享区域中的过程必须在运行时动态进行，不能影响其它进程的运行，也不能重新启动系统。这就需要运行时的动态进程迁移。如下图 2-2，本文实现的动态进程迁移流程分为以下五个步骤：

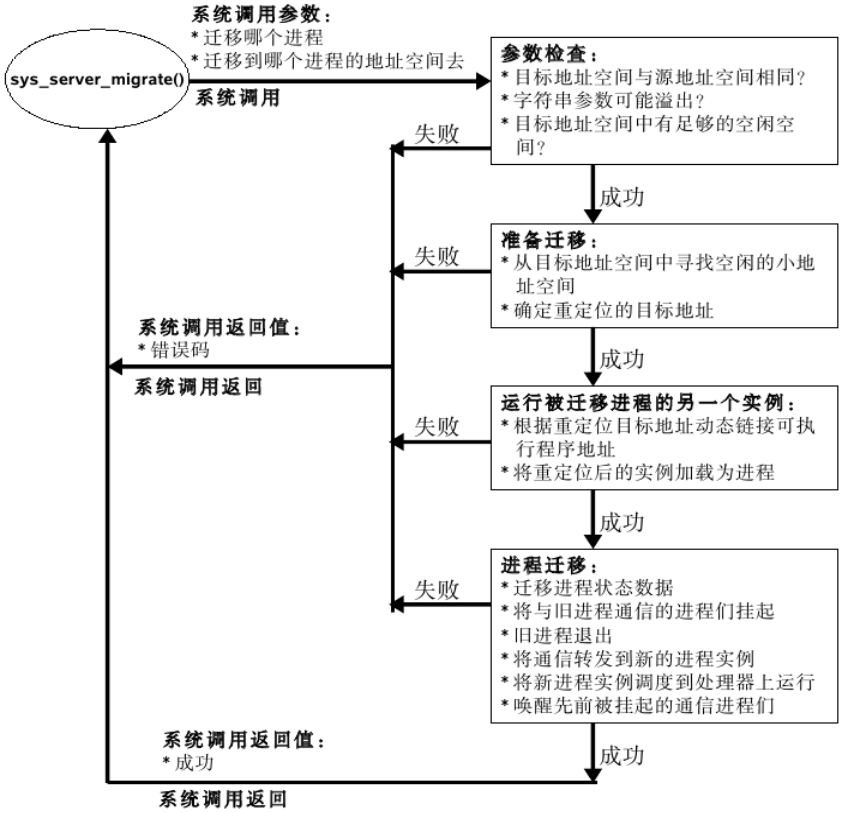


图 2-2 进程迁移五个步骤

- 服务进程管理程序调用 sys_server_migrate()系统调用

  在 MLXOS 中有一个名为 serverroot 的进程，该进程的任务就是监视所有服务进程，并发起服务进程的运行时迁移操作。内核提供了 sys_server_migrate()系统调用，serverroot 进程可以使用该系统调用开始对指定服务进程的迁移操作。

- 检查系统调用参数

  sys_server_migrate()系统调用主要的参数是：指定迁移的服务进程 ID 和目标地址空间服务进程 ID。目前服务进程 ID 是根据服务进程的能力唯一标识的，例如提供文件系统能力的服务进程的标识符为 FS，除正在进行迁移外，任何一个时刻，系统中只有一个服务进程可以提供 FS 的能力。当迁移的服务进程 ID 为 0 时，则表示将被迁移进程迁移到具有内核优先级的小地址空间中。

  由于 MLXOS 是微内核，因此系统调用不是由内核本身来处理，而是由运行在内核优先级的 system_task 内核线程来处理。system_task 从接收的消息中读取系统调用参数，检查是否合法。如果不合法，则将返回系统调用失败的消息给 serverroot 进程。

- 进行准备工作

  system_task 要进行的准备工作主要是检查目标地址空间是否拥有空闲区域可容纳被迁移的服务进程，当找到一个空闲区域后，要根据该区域的线性地址确定重定位服务进程时的目标地址。

- 运行指定进程的重定位后实例

  当确定了重定位的目标地址后，system_task 将被迁移服务进程在内存中的可重定位二进制代码重定位到指定的地址空间中[①]。然后根据重定位后的代码创建一个新的运行实例（进程），并加入到运行队列中准备调度。如果目标地址空间在 2G 以上，则将进程加载到具有内核优先级的小地址空间中，否则加载到用户态的目标区域。

- 在不同地址空间中迁移进程

[①]鉴于论文工作的时间限制，为了简化细节，对位置无关代码的地址重定位工作目前并没有在 MLXOS 中实现。而是借助于用户态的工具链工具来完成地址重定位的。当前的简化方法是为每个服务进程都预先约定好在目标地址空间中的目标地址，并在编译生成服务进程时生成起始地址为 0 和从预定小地址空间起始的两份程序。当调用 sys_server_migrate()系统调用后，system_task 根据系统调用参数，从内存中将预先加载的对应服务进程可执行二进制代码进行分析和映射，来直接创建重定位后的进程。

开始进行进程迁移时，system_task 向原服务进程发消息，原服务进程将当前服务状态数据返回给 system_task。system_task 在接收到状态数据后，终止原服务进程，并挂起所有正在和原服务进程通信的进程。然后调度新服务进程运行，新服务进程首先向 system_task 获取原服务进程的状态数据，然后转入工作状态。此后 system_task 将所有对原服务进程的通信都转发到新服务进程，并恢复所有被挂起进程。至此服务进程迁移操作结束。

当进程从复用地址空间迁移到独立的用户态地址空间时，操作流程大致与上述步骤相同，差别在于：

- 不需要再检查是否有可用的小地址空间
- 原服务进程退出时注销所占用的小地址空间
- 新进程以用户态优先级运行

通过以上步骤，就可以在运行时完成对服务进程的地址重定位，而重定位后的服务进程，就可以复用其它进程 4G 字节的地址空间。

2.3 基于进程迁移的地址空间复用

基于进程迁移的地址空间复用方法，不涉及进程段属性的改变，可以同时运行在 32 位和 64 位 x86 处理器平台上，在进程切换时同样可以避免不必要的 TLB 刷新。同宏内核操作系统相比，用户可以在进程间通信的性能和安全性之间作出灵活的选择，决定是否将某个服务进程迁移到其它进程的地址空间中，或进一步迁移到内核地址空间的共享区域中。

2.3.1 可以同时运行在 32 位和 64 位 x86 平台上

基于段的地址空间切换只能用于 32 位的 x86 处理器上。在 64 位的 x86 处理器上因为段基址总是为 0，且不再检查段长度限制，因此无法采用基于段的地址空间切换。而基于进程迁移的地址空间复用，是通过进程地址重定位实现的，对进程地址空间的保护是基于页式保护来实现的，在 x86_64 处理器上仍然有效。

相比于基于段的地址空间切换，进程迁移必须要进行程序地址的重定位，操作过程比较复杂，迁移过程较慢，但是这是目前唯一可以同时应用在 32 位和 64 位 x86 处理器上的地址空间复用方法。

2.3.2 同样可以避免 TLB 刷新

在传统内存管理模式中，内核空间的页表设置为全局页，在进程切换中刷新 TLB 时可以避免刷新内核空间相关的 TLB 表项。在复用地址空间时，服务进程被重定位到共享区域的小地址空间中。将具有内核优先级的共享区域对应的页表项设置为全局，在某些进程切换的情况下，同样有可能避免 TLB 刷新[①]。

2.3.3 可以在性能和安全之间进行选择

如图 2-1（b）所示，位于小地址空间中的服务进程以内核优先级运行，因此具有修改系统内核代码或数据的权限。因此，只有安全性和稳定性可信的进程才可以被迁移到内核优先级的地址空间中。当某一个服务进程经过充分测试后，如果需要更高的进程间通信性能，就可以将该服务进程迁移到具有内核优先级的小地址空间中。如果某些进程彼此之间频繁通信的话，也可以将这些进程迁移到同一个用户态地址空间中，则这些进程之间仍有可能破坏彼此的数据或代码，但不会影响系统整体的安全。如果对服务进程的稳定性和安全性尚未充分测试的话，则仍可以让服务进程像普通应用程序那样运行在独立的地址空间中。

因此，基于地址迁移的进程地址空间复用可以让用户在性能和安全性之间进行选择[②]，使用户有了一个在两者之间得到平衡的机会。

2.3.4 不需重新启动内核，可提供不间断服务

进程在不同地址空间之间的迁移是在系统运行时进行的，而且不会影响其它正在运行的进程。由于在进程迁移过程中，服务进程的状态数据也会迁移，因此迁移后的服务进程可以继续之前的工作而不需要重新初始化其它应用程序或整个系统。对于一些状态数据无法同步的服务进程，迁移后的服务进程可以向后续请求返回-EAGAIN 错误，这样应用程序可以再次向服务进程发出相同的请求以得到正确响应。而实际中，诸如 GNU libc 的系统调用库函数会在返回失败

[①]在第六章中，提出可信进程间通信组的概念。如果涉及进程切换的两个用户态应用程序同属于一个可信进程间通信组，那即使两个进程对应的页都不是全局页，也不需要刷新 TLB。在第六章中将会对进程切换时不需要刷新 TLB 的各种情况进行详细阐述。

[②]在可信进程间通信组中，任何两个用户态进程也可能修改彼此的代码和数据，这种情况下，也是存在性能和安全性的选择与权衡的。

时重复调用多次，因此从应用程序层面几乎感觉不到系统请求的接收方已经迁移到不同的地址空间了。

2.3.5 灵活分配共享地址空间

由于服务进程迁移时，不影响其它服务进程和应用程序，因此可以根据需要来调整重定位地址的位置。当目标地址空间无法容纳被移入的进程时，如果可用的空闲地址空间总和大于该服务进程所需的地址空间，那么就可以通过运行时地址空间迁移的办法，对目标地址空间中可复用的地址空间进行“碎片整理”。最终仍有可能整理出一块连续的足够大的可复用地址空间，将服务进程移入其中。

运行在复用地址空间中的服务进程，也可以在运行时被迁移到独立的地址空间中作为普通应用程序来运行。当可复用的地址空间全部被占用，而需要移入更重要的进程时，就可以选定一些服务进程迁移到独立地址空间中，以让出有限的地址空间。

基于进程迁移的地址空间复用的灵活性不仅体现在服务进程可以迁移到内核权限的小地址空间，一个普通进程也可以迁移到另一个进程的用户态空间来复用地址空间。两个或多个进程，如果他们都经过了充分测试，信任彼此不会破坏其它进程的数据或代码，那它们就可以组成一个可信进程间通信组。通过进程迁移的方法，可以让可信进程间通信组中的多个进程迁移到某一个进程的地址空间中，使得多个进程复用一个用户态地址空间。当某一个处理器上的进程切换发生在可信进程间通信组的任何两个进程之间时，就可以不加载新的页全局目录地址，从而避免 TLB 的刷新。

2.4 小结

本章阐述了基于进程迁移的地址空间复用的概念和原理，这是实现可信进程间通信组的基础。在运行时进程迁移于不同地址空间的基础上，系统可以实现多种更为复杂的可信进程间通信组策略，使得用户可以在多个用户进程之间的通信性能和安全性之间作出更为灵活的权衡。下一章将会对可信进程间通信组的概念和原理进行详细说明。

第三章　　可信进程间通信组

在基于进程迁移的进程地址空间复用的基础上，属于同一个可信进程间通信组的进程可以复用同一个地址空间，当进程切换发生在同一个可信进程间通信组的进程之间时，可以不加载新进程的全局页目录表地址，从而避免 TLB 刷新，进而提高了进程间通信的性能。

3.1 什么是可信进程间通信组

如果两个或多个进程，当他们运行在同一个地址空间时，在整个生命周期内都不会访问不属于自己的地址空间范围，那么这些进程就不会影响和破坏运行在相同地址空间的其它进程。在这种情况下，他们彼此是可信的。如果两个和多个彼此可信的进程之间会发生频繁的进程间通信，为了提高他们的进程间通信性能，可以让他们运行在同一个进程的地址空间中，这些进程就构成了一个可信进程间通信组。最初拥有该地址空间的进程称为该可信进程间通信组的主进程。该可信进程间通信组由主进程（在进程表中的编号）来标识，所有组内进程复用主进程的地址空间。

如下图 3-1，进程 P_1，P_2，P_3之间存在频繁的进程间通信（消息传递）。虽然在 32 位 x86 上每个进程的用户态空间有 2G，但是整个生命周期内，这三个进程所使用到的地址空间范围加起来也远远不到 2G。如果这三个进程经过充分测试是稳定的，那么就可能将它们迁移到相同的地址空间中来复用地址空间。

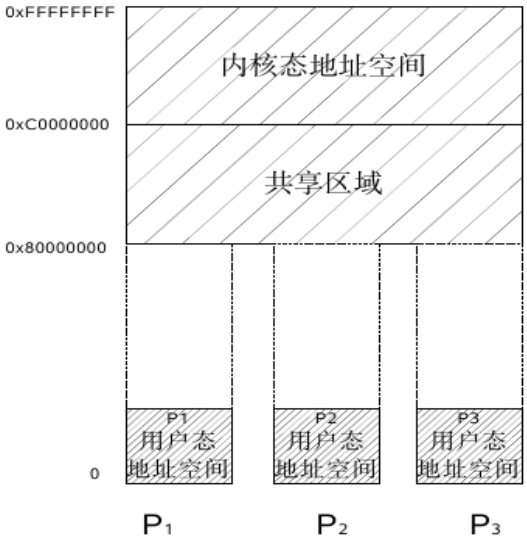


图 3-1 组成可信进程间通信组前

这时如果将进程 P_2和 P_3迁移到 P_1的地址空间中，那么 P_1，P_2，P_3就构成了一个可信进程间通信组。如下图 3-2，进程 P_1是由 P_1，P_2，P_3组成的可信进程间通信组的主进程，所有的用户态进程复用主进程 P_1的 2G 用户地址空间。

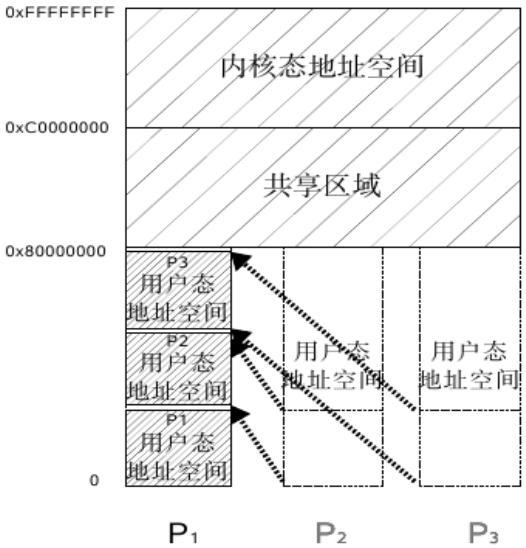


图 3-2 组成可信进程间通信组时的示例

所有运行在共享区域中具有内核优先级的服务进程组成一个特殊的可信进程间通信组。该组可以是任何其它用户态可信进程间通信组的子组，但是在任

何一个时刻，在每个处理器核或处理器上，只能有一个父组。如下图 3-3，当运行在同一个逻辑处理器上时，由运行在共享区域中的服务进程 S_1、S_2 组成的可信进程间通信组可以是由用户态进程 P_1、P_2、P_3 构成的可信进程间通信组的子组。

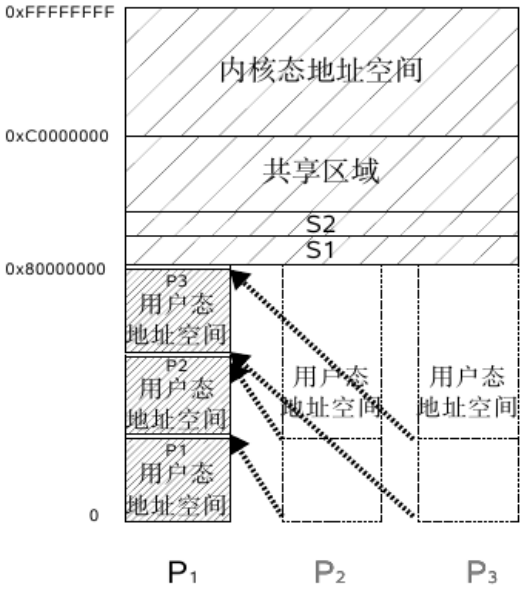


图 3-3 包括服务进程的可信进程间通信组

当可信进程间通信组中的某一个用户态进程切换到运行在共享区域的内核态服务进程时，后者所在的可信进程间通信组自动成为前者所在可信进程间通信组的子组。

3.2 可信进程间通信组基本策略

可信进程间通信组的构成具有以下基本策略：

- 形成可信进程间通信组的多个进程之间必须复用同一个地址空间。
- 运行在小地址空间、具有内核优先级的所有服务进程组成一个特殊的可信进程间通信组，该可信进程间通信组的主进程是在共享区域中线性地址最低的服务进程。只有该组是任何其它可信进程间通信组的子组，但在每个处理器上任何时刻只能有一个父组。
- 一个运行在用户空间的进程，只能唯一的属于一个确定的可信进程间通信组。

- 当一个可信进程间通信组的用户态进程切换到共享区域的小地址空间中的服务进程时，用户态进程所在的可信进程间通信组自动成为小地址空间中服务进程所在的可信进程间通信组的父组。
- 每个运行在独立地址空间中的进程，构成一个自身为主进程的可信进程间通信组。

当两个运行在独立的地址空间的进程发生进程切换时，由于操作系统不会为不同地址空间的相同线性地址分配相同的物理地址（共享内存除外），如果不刷新 TLB 中的非全局表项，那么处理器在执行新进程的代码时，就可能发生 MMU 寻址错误。为了避免这个情况，x86 处理器提供了多种刷新 TLB 表项的方法，譬如在进程切换时通过重载新进程的页全局目录表地址到 CR3 寄存器[①]中来刷新所有非全局的 TLB 表项。

根据上述基本策略，可以发现，属于同一个可信进程间通信组的每个进程，在整个生命周期内，所访问的地址空间范围都不会彼此发生重叠。因此，TLB 表中不会存在相同的线性地址映射到不同的物理地址的情形。因此，当某个处理器在调度进程时，如果发现进程切换发生在一个可信进程间通信组的两个进程之间，那么操作系统调度器就可以不进行刷新 TLB 的操作。因此，可信进程间通信组内的进程之间进行通信的性能，要高于普通进程之间的通信性能。

在 32 位和 64 位 x86 的多处理器或多核处理器系统中，TLB 属于处理器的一级访问缓冲寄存器，每个处理器核都会有自己的指令或数据 TLB 表。因此，为了提高进程间通信的性能，在多核处理器的系统上，操作系统应当尽可能将可信进程间通信组的所有进程都调度到相同的处理器核上运行。当必须要将同一组的进程迁移到其它处理器核上运行时，也要首选选择相同封装内共享二级缓冲的处理器核。这种针对多核处理器的可信进程间通信组的调度策略，也是与当前业界共识的多核任务调度策略相同的。

3.3 符号标识约定

在对可信进程间通信组的特性和应用做进一步阐述之前，首先就一些符号标识的约定进行说明。这样在后面进行伪代码示例时，就清晰多了。

[①]关于 CR3 寄存器的更多内容请参看附录 A 或参考文献［30］。

3.3.1 进程

可信进程间通信组中运行在用户地址空间的进程以 p 标识，当存在多个进程时，加数字下标。譬如 p_i 标识某个进程组中的第 i 个用户态进程。所有的用户态进程构成的集合标识为 P。

可信进程间通信组中运行在内核地址空间的服务进程以 s 标识，当存在多个服务进程时，加数字下标。譬如 s_k 标识进程组中的第 k 个内核态服务进程。所有的内核态服务进程构成的结合标识为 S。

3.3.2 可信进程间通信组

主进程为 p_i 的可信进程间通信组标识为 TPIG(p_i)，其中 i 为进程 ID。

如果进程 p_m 属于可信进程间通信组 TPIG(p_i)，则记为 $p_m \in$ TPIG(p_i)。其中 m，i 为进程 ID。

3.3.3 全部由内核态服务进程构成的可信进程间通信组

所有运行在小地址空间中的服务进程组成一个特殊的可信进程间通信组，当该可信进程间通信组不属于任何其它可信进程间通信组时（即没有其它用户态进程运行时），它就是系统中唯一的可信进程间通信组。该可信进程间通信组标识为 TPIG（S），S 为所有内核态服务进程的集合。

3.3.4 可信进程间通信组的尺寸

可信进程间通信组中的进程个数称为该可信进程间通信组的尺寸，标识为 SIZE(TPIG)，根据定义，可知任何一个可信进程间通信组的尺寸永不为 0。

则对于主进程为 p_i 的可信进程间通信组 TPIG(p_i)，当 SIZE(TPIG(p_i)) == 1 时，进程 p_i 运行在独立的用户态地址空间中，没有任何其它进程复用它的地址空间，这就是传统操作系统的进程地址空间使用方式。

3.4 可信进程间通信组特点

基于可信进程间通信组的地址空间复用和进程调度，具有一些其它方法和实现不具有的特点，这些特点并不能彻底解决微内核中进程间通信和多核处理器上进程调度的性能难题，但颇为有趣，值得一谈。

3.4.1 更清晰的判断进程切换时是否需要刷新 TLB

在 Jochen Liedtke 的透明复用用户态地址空间思想中，进程调度在以下情况下不需要刷新 TLB：

- 普通进程切换到小地址空间中的进程。
- 小地址空间中的进程之间切换。
- 一个普通进程切换到小地址空间中的进程后，小地址空间中的进程又切换回该进程。

该判断方法的判断条件较为复杂，无法处理地址空间复用发生在非共享区域的情况，没有考虑多处理器平台上进程调度的情况。相比之下，基于可信进程间通信组来判断是否在进程切换时刷新 TLB 的条件就非常简单，用文字描述是一句话：

在同一个处理器或处理器核上进行两个进程的切换，当且仅当这两个进程属于同一个可信进程间通信组时，不需要刷新该处理器或处理器核的 TLB。

如下伪代码 3-1 描述了调度器中和刷新 TLB 相关的处理流程，其中 p_{prev} 和 p_{next} 分别是要被调出和调入处理器的进程：

```
void update_context( p_prev ∈ TPIG(p_k),  p_next ∈ TPIG(p_m) ):
        ... ....
10      if  k == m  or  TPIG(p_k) ∈ TPIG(p_m)
11              then    return;
12              else    if  p_prev ∉ S  and  p_next ∈ S
13                              then  TPIG(p_prev) ← TPIG(p_prev)  ∪ TPIG(S);
14                      fi;
15              else    flush TLB
16      fi;
        ... ...
```

伪代码 3-1 可信进程间通信组在进程上下文切换中的处理流程

10～11 行伪代码表明同一个可信进程间通信组中的两个进程在切换时不需刷新 TLB。12～14 行代码是将小地址空间中的进程构成的可信进程间通信组设置为被调出进程所在可信进程间通信组的子组。只有所有条件都不满足时，update_context 函数才会执行 15 行伪代码刷新 TLB。可见基于可信进程间通信组来判断是否需要刷新 TLB，更为简洁。

3.4.2 复用地址空间，而非共享地址空间

在多线程程序的运行中，一个进程的多个线程共享整个进程地址空间。也就是说，任何一个线程都可以访问整个进程空间中所有的全局变量和函数，每个线程都具有自己的运行时栈来保存执行上下文和局部数据。

而可信进程间通信组中的进程之间，是复用地址空间，而不是共享地址空间。虽然可信进程间通信组中的进程之间在切换时，和同一进程内的线程切换类似，不需要刷新 TLB，但是具有以下不同：

- 本质上这些进程原本都是运行在各自独立的地址空间中的，不能通过全局符号来访问其它进程地址空间的全局变量或函数。因此复用地址空间后，彼此之间不可以访问自己地址空间之外的内容（程序出错的情况除外）。
- 线程之间共享进程的页全局目录，而可信进程间通信组的进程各自具有独立的页全局目录。这些进程基于操作系统对地址空间的特殊管理，通过设置相应的页全局表项和页表来实现地址空间的复用。
- 当线程是通过共享同一地址的多个轻量级进程实现时，线程切换就是进程切换，这时线程切换的性能和可信进程间通信组中进程切换的性能相近。当线程是通过用户空间线程库实现的，那线程切换将在用户空间完成，不会调用内核态代码，这种情况下可信进程间通信组中进程切换的性能要低于线程切换。

3.4.3 源代码级透明的地址空间复用

多线程编程中，多个线程共享进程的地址空间是通过调用线程库例程来实现的。而可信进程间通信组中的进程复用地址空间，是通过操作系统内核对内存管理的支持实现的。因此，可信进程间通信组中进程之间的地址空间复用是源代码透明的。也就是说，在复用地址空间时，不需要修改每个进程的源程序。而多线程程序，如果要增加新的线程，改变线程共享地址空间的行为，是需要修改程序源代码、重新构建可执行代码的。

当可信进程间通信组中的一个进程需要运行在独立的地址空间时，该进程可以在运行时被迁移到独立的地址空间中。但基于调用线程库实现的多线程地址空间共享，就无法如此灵活的将一个线程迁移到一个独立地址空间中作为进程运行了。

3.4.4 与生俱来的模块化特性

在多线程编程中，由于多个线程可以直接访问全局的变量和函数，因此在多线程并发访问时的同步和互斥就需要格外小心。在多线程编程模式下，函数的调用效率会比较高，但是由此带来的问题是，大量代码相互交错的引用，功能模块之间无法确保强制性的隔离，程序员很容易因为不小心调用了其它功能模块的代码而导致不能预料的错误后果。

而可信进程间通信组中的进程，原本运行于独立的地址空间之中，彼此之间没有任何办法相互直接调用对方上下文中的全局变量或函数。因此，程序员必须使用操作系统提供的标准进程间通信接口来通信，所以这些进程之间具有与生俱来的模块化特性。

虽然多线程程序内的线程间通信性能更高，但可信进程间通信组的进程间通信与生俱来的模块化特性，有助于提高参与通信的各个模块的总体稳定性和安全性。

3.5 小结

本章基于第二章阐述的复用地址空间的方法，提出和阐述了可信进程空间通信组的概念与原理，并进一步说明可信进程间通信组对于提高进程间通信性能的作用，以及相比于传统地址复用方法的特点。在下一章中，将基于论文原型操作系统 MLXOS 来说明，如何设计 MLXOS 各个子模块以支持上述特性，并以伪代码的形式来描述具体的编码实现。

第四章　设计与实现

基于进程迁移的地址空间复用，和可信进程间通信组，需要内核和系统库同时支持才能工作。在前两章概念和原理介绍的基础上，本章对 MLXOS 中相应的设计和实现进行详细阐述。本章首先列出内核和系统库必须提供的功能点。然后基于这些功能点，阐述基本设计思想。最终基于设计思想，使用伪代码或流程图来说明具体设计。详细的代码实现，读者若有兴趣可以参照 MLXOS 实现代码，有关如何获得源代码和构建实验环境的具体步骤，请参看附录 C。

4.1 原型操作系统 MLXOS

总体来讲，操作系统要支持基于进程迁移的地址空间复用和可信进程间通信组，必须考虑以下若干方面：

- 内存管理，包括内核空间的虚拟内存管理和用户空间库中的内存管理。
- 地址空间分配，包括内核态内存管理子系统和用户态系统库中对地址空间的分配。
- 进程间通信框架和接口。
- 进程地址空间迁移的系统调用接口和操作流程。
- 内核中的进程管理
- 进程调度对基于进程迁移的地址空间复用和可信进程间通信组的支持。
- 进程切换对于可信进程间通信组的支持。

论文会说明涉及到内核态和用户态的相应修改，以求尽量全面的阐述论文工作的想法，尽可能有利于读者理解。

对于操作系统设计和开发略有经验的读者可以感觉到，对一个成熟操作系统进行上述各方面的修改，并使之可以正确工作的工作量是巨大的，修改出一个符合课题设计的操作系统，其工作量甚至远大于重新设计一个原型操作系统。

在论文工作的初期，经过仔细研究认为，目前可能作为论文工作的系统原型，开放源代码可以自由修改的主要有三个：Linux，Minix 和 L4。其中 Linux 非常成熟，涉及代码量巨大，也很难保证在论文课题的时间限制之内完成上述所有功能模块的修改,因此不考虑基于 Linux 直接修改完成课题工作。对于 Minix

现在最新版本是Minix3,它的内存管理是基于32位x86处理器的段式内存管理，既不支持页式内存管理，也不支持基于页的虚拟内存管理。要基于 Minix 的代码来修改，来完成上述各方面工作，就必须要为Minix3添加基于页的内存管理和虚拟内存的支持，而这些修改涉及到Minix3各个子模块，几乎是重新设计和实现 Minix3，工作量仍然巨大。最后考虑的是被称为第二代微内核操作系统的L4，但L4仍然无法直接作为开发原型。主要原因是，L4的精华在于就微内核系统设计中的一些先进思想和在各个平台上都统一的进程间通信编程接口。而 L4本身并不具备完整成熟的操作系统功能，多数传统操作系统的功能，诸如进程管理、内存管理、设备驱动程序等，都是由运行在L4用户态的Linux服务进程L4Linux来完成的。L4Linux项目的初衷是将Linux内核修改后，作为运行在L4系统的用户态服务进程，来满足L4系统上其它应用程序的系统请求。该策略的好处是，L4 的开发人员在工程化方面的工作（譬如设备驱动程序）都可以直接借用Linux内核的工作成果，这样节省了大量的人力和物力。

因此，最终认为，工作量最小的办法，就是重新设计开发一个原型操作系统，在该原型操作系统上实现基于进程迁移的地址空间复用和可信进程间通信组。因此就有了MLXOS原型操作系统。为了保证代码质量和项目进度，MLXOS借鉴、引用和修改了大量开放源代码项目的成果。譬如在内核中，在思想上借鉴了很多L4的设计，在实现上大量借鉴了来自于Linux和Minix的代码。同样在用户空间系统库中，也大量借鉴了 uClibc 项目和 Minix3 libc 的源代码[①]。实践证明，这种策略可以保证快速的、高质量的开发出原型系统，在毕业论文要求的期限内完成必须的开发工作。

MLXOS采用微内核架构，内存管理、进程管理和中断处理均以内核线程的形式运行在内核空间运行，设备驱动程序以普通进程形式运行在用户空间。驱动程序与内核线程或其它应用程序之间的通信均通过消息传递进行。同样 MLXOS用户态的 C 函数库也是针对微内核设计的，所有的系统调用都被包装成消息包传递给内核或其它消息接收者。

[①]由于 Linux 内核采用的许可证是 GPL2，而 Minix3 采用的许可证是 BSD，因此将这两种不同许可证的源代码混合成一个系统是需要技巧的，否则就很容易引起许可证不兼容和冲突的问题。经过咨询有经验的开放源代码程序开发人员，最终的解决办法是：整个系统采用 GPL2 和 BSD 混合许可证策略，对于所有从 GPL2 许可的代码中引用过来的源代码，均在源代码中表明采用 GPL2 许可，同样将所有原先采用 BSD 许可的代码也在源代码中详细列出采用的是 BSD 许可。这样的结果是，即遵守了 GPL2 许可，也遵守了 BSD 许可，但结果是代码中会有大量和许可协议相关的注释文字。

4.2 内核与库的支持

下面介绍 MLXOS 就上述若干方面是如何考虑和设计的，如果可能，内核空间和用户态库函数都会进行阐述。需要留心的是，用户态库函数虽然绝大多数情况下是运行在用户空间的，但是当某一个服务进程被迁移到内核态的小地址空间时，库代码也会运行在内核态。MLXOS 的 C 库和传统类 UNIX 系统的 C 库略有不同。

4.2.1 内存管理

动态的可用地址空间

在复用地址空间后，被迁移的服务进程的可用地址空间就不再是 0－2G，而是 2G－3G 之间的某个小地址空间。因此，需要动态的确定当前进程的可用地址空间。在 MLXOS 的进程控制块数据结构 task_struct 中有一个名为 mm 的成员，该成员为数据结构 mm_struct，它的作用就是处理绝大多数和当前进程内存管理相关的细节。在 mm 结构中定义两个成员 unsigned long base 和 unsigned long limit，这两个成员在运行时动态确定了当前任务的可用地址空间，即[base, base + limit)这段线性地址空间。这个区域的确定可以帮助内核在处理和内存映射相关的系统调用时分配正确的地址空间范围，但在程序错误的访问区域外地址时，并不能总是触发缺页中断。

根据重定位地址加载进程

由于进程可能被迁移到任何可能的地址空间中，因此 MLXOS 需要在加载时确定进程的可用地址空间。在重定位服务进程时，进程的大小会初始化为 SMALL_TASK_SIZE，程序的入口地址被链接到代码的最低地址处，而进程的可用地址空间的起始地址就是包含入口地址的对齐 SMALL_TASK_SIZE 的地址。通过如下定义的宏

```
#define SMALL_TASK_BASE(entry)  ((entry) & ~(SMALL_TASK_SIZE - 1))
```

就可以根据进程的入口地址确定进程可用地址空间的起始地址，而地址空间长度就被初始化为 SMALL_TASK_SIZE。目前 SMALL_TASK_SIZE 定义为 64MB，但实际上对于绝大多数服务进程 8MB 的地址空间都是绰绰有余的。

动态确定内存映射地址空间范围

在传统类 UNIX 操作系统中，内存映射的有效地址空间范围是相对于 0 起始地址计算的。当进程被迁移到地址空间的其它位置后，就不能够以 0 为起始地址寻找空闲的地址空间范围了。诸如处理 mmap 之类的系统调用时，内核需要从当前进程的地址空间中寻找未被映射过的地址空间，然后将该地址空间作为返回给 mmap 调用者的可用指针。在 MLXOS 中将进程地址空间的最低 4MB 空间作为栈空间，4MB 以上 512KB 作为进程的参数空间，再向上是进程的代码和数据段等。MLXOS 的 C 库没有进行复杂的内存管理，malloc()/free()函数直接通过 mmap()/munmap()向系统申请空闲内存。因此内核不能从 4MB+512KB 的位置开始直接搜索空闲地址空间，如果将 4MB+512KB 用宏 TASK_UNMAPPED_BASE 来代替，那么寻找空闲地址空间的起始位置应当是，

```
start_addr = mm->base + TASK_UNMAPPED_BASE
```

这样当进程被迁移到地址空间的不同位置时，内存映射仍然可以正确的工作。

用户态内存管理

传统的类 UNIX 系统的 C 库都有自己的用户态内存管理代码，譬如 malloc()/free()，并不是每次调用都会访问系统调用，而是尽可能从用户态的内存堆中分配和释放内存。这样一来，关于用户态内存管理就存在状态数据——在进程迁移时，这些状态是否也要考虑呢？在 MLXOS 的设计中，当进程迁移到另外一个位置时，先前的用户态内存状态不需要迁移到新的实例中。在 MLXOS 的进程迁移中，只迁移工作状态，新的实例通过该服务进程的标准接口转移状态数据时，如果需要分配内存，则会在新的实例的地址空间中再次分配地址空间，当服务进程的工作状态迁移完成后，也就相应的完成了用户态内存状态数据的重建。目前 MLXOS 的用户态 C 库没有实现复杂的 malloc()/free()函数族，而是直接通过内核的内存管理完成的。每次应用程序调用 malloc()函数时，C 库都直接根据请求的内存块大小调用 mmap()系统调用，用内核的内存管理系统来决定将内存空间映射到请求进程的某个地址空间范围。同样，当调用 free()函数时，C 库函数直接调用 munmap()系统调用，由操作系统内核来取消相应的映射区域。这种设计每次分配和回收内存都需要调用内核代码，性能较低，但它的好处是不需要在用户空间维护，实现简单，比较适合构建原型操作系统。

设备驱动内存管理

和设备驱动相关的内存管理，有时候会有一个特殊情况。譬如在 MLXOS 中实现的 tty 驱动，负责终端信息显示。在 IBM PC 规范中，终端字符显示的显存位于物理地址的 0xB8000。由于 tty 驱动程序是以应用程序的形式运行在用户空间的，因此不能直接访问独立地址。在这种情况下，内核就需要提供设备驱动的内存管理功能。在 MLXOS 中，tty 驱动可以调用 maptty()系统调用将显存的物理地址映射到自己的用户态地址空间中。tty 驱动并不需要知道显存的物理地址，只需要将系统调用返回的地址作为显存缓冲区的起始地址使用即可。同样，内核在处理 maptty()时需要根据当前进程的 base 和 limit 来确定返回给进程的线性地址范围。该内存是不能使用 free()函数来释放的，而是要通过 unmaptty()来释放。因为显存区域是特殊的，只能通过共享内存的方式分配给请求者，不能够释放到普通内存区当作普通内存来使用。

4.2.2 进程间通信框架和接口

遵循 POSIX 标准

在实现用户态库函数时，应当遵循 POSIX[①]标准，虽然下层封装系统调用时，会将参数封装为消息包，但是上层编程接口，应当遵循 POSIX 标准。这样在将来移植其它程序到 MLXOS 系统上时，就会大大的减少移植工作量。

系统调用的消息格式

不同于类 UNIX 操作系统，MLXOS 在系统调用中传递的参数不会超过 3 个。基于消息的系统调用接口，只有三个参数，分别是消息发送/接收者，消息包地址，消息传递功能号。其中消息接收者表明该消息发送给哪个进程或从哪个进程接收，消息包地址是要发送的消息包在当前进程地址空间中的线性地址，消息传递功能号表明进程是要发送消息或接收消息。具体 POSIX 规定的系统调用号等，都包装在消息包内。根据系统调用中涉及到的参数个数和参数数据类型，可以采用不同的消息格式。而该消息的接收方，因为清楚自己要处理的是什么请求，因此可以按照实现约定来解析消息包，得到相应数据并进行相应处理。在后面章节将会介绍 MLXOS 的消息格式实现。

① POSIX 全称为可移植操作系统接口（Portable Operating System Interface），是目前操作系统中被业界广为支持的系统编程接口事实标准之一。包括各种 UNIX，Linux 和 Windows 操作系统，均声称支持 POSIX 标准。因此 MLXOS 在开发时，也力求遵循 POSIX 接口标准。

消息传递

消息包从一个进程传递给另外一个进程，实际上就是将消息从一个进程的地址空间，复制到另一个进程的地址空间。有时候，消息在不同地址空间中的传递可以通过内存映射来完成，这样可以避免内存复制的开销。为了简单的实现消息传递框架，目前 MLXOS 的消息传递是基于内存复制的。由于 MLXOS 不支持虚拟内存交换到磁盘上，因此所有进程的内存都存在于物理内存之中，不会因为当前访问内存被交换到磁盘上而触发缺页中断。因此 MLXOS 的内存复制过程就比较简单，仅仅是在两个不同的地址空间中复制内存。

内核在传递消息前，已经可以确定发送方发送消息的地址和接收方接收消息的地址，根据各自进程的页全局目录和相应的消息地址，就可以解析出对应于内核的线性地址，这个工作是由 uaddr_to_kaddr()函数完成的。然后内核就可以用发送方和接收方缓冲区对应的内核线性地址来复制内存。需要注意的是，当消息跨越内存页框边界时，由于用户态连续的页在内核态并不一定是连续的，因此需要再次调用 uaddr_to_kaddr 获取新的内核线性地址。这个跨地址空间的消息复制操作，是由_cross_uspace_copy_memory()函数完成。但内核服务线程通常不会直接调用_cross_uspace_copy_memory()函数，而是调用它的封装函数_copy_message()。

系统调用框架

MLXOS 采用微内核架构，系统调用不会被内核直接处理，内核只负责根据消息的接收者将消息传递给对应的接收进程。如果发送方有权限发送进程但接收方没有准备好接收消息，那么内核会将发送进程挂起休眠。当接收方处理完毕发送方的消息后，内核会再次将发送方唤醒。同样当一个进程准备接收消息时，如果没有消息到来，该进程也需要休眠，直到消息到来时被唤醒。内核要能够正确的将某些任务转入休眠，并及时的将某些任务唤醒转入运行。

当系统初始化完成后，内核代码就不会主动运行了，而是像库函数那样被其它执行流程调用。譬如，当系统调用发生时，中断处理程序会调用内核代码将消息包发送给运行在内核空间的服务线程 system_task。system_task 线程处理与进程管理、内存管理相关的系统服务，在适当时会将内核代码像库函数那样来调用。完成消息处理后的消息返回和接收方唤醒行为虽然都是由内核代码

完成的，但是这些行为都不是内核主动发起的，而是由服务进程驱动的。在后面的实现部分，会用伪代码来说明和系统调用框架相关的数据结构和相应处理过程。

4.2.3 进程管理

虽然多数人认为操作系统中最重要、最复杂的部分是进程管理部分，但实际并不完全如此。MLXOS 的进程管理，尤其是进程调度器和调度函数，相当的简单朴素。目前为止，这个调度器工作的很好，而且显示了它的灵活性。在进程管理功能中支持基于进程迁移的地址空间复用和可信进程间通信组，并没有涉及大量代码的修改，而且代码流程非常清晰。

进程创建

MLXOS 目前没有磁盘驱动，因此服务进程都是依靠 GRUB[①]的模块方式从磁盘上加载到内存中的。服务进程重定位的工作是在用户空间完成的，为了简便起见，目前 MLXOS 中的重定位工作是借助 GNU 的工具链预先完成的。因此系统引导时会将从 0 开始的服务进程和重定位到某个预定小地址空间的服务进程一并加载到内存中。这样做的目的是快速开发原型系统，避免将大量时间花费在实现重定位和动态链接的细节中。

当系统启动后会搜索所有被 GRUB 加载到内存中的可执行程序镜像，并调用 load_multiboot_module()函数依次将模块加载。加载过程大致流程如下：

- 分析可执行程序镜像文件中各个段，找到可以加载的那些段，然后在内存中为其分配物理页框，为进程设置相应的页表，并通过内核的虚拟内存模块为它们设置好可用地址空间链表。
- build_driver_process()函数会在进程的栈中初始化最初运行的上下文，包括各种通用寄存器的初始值和第一条执行的指令等，并将进程控制块链接到系统任务的运行队列上。
- register_procs_table()函数将进程注册到进程表中以确定进程的唯一标识，在进程间通信时内核就可根据发送/接收者的标识来传递消息。

[①] GRUB 全称为 Grand Unified Bootloader，是 GNU 项目中的一个子项目。在 BIOS 初始化完毕后可以将 GRUB 加载到内存中，之后 GRUB 可以将操作系统内核、引导驱动模块和其它必须模块加载到内存中，然后转而跳转到加载好的内核开始引到系统。有了 GRUB 引导加载器后，操作系统的初始化就大大简化，不需要再考虑内核未加载到内存之前的繁琐处理了。

- 在 register_procs_table()函数的最后会调用 set_default_tpig()函数设置进程缺省的可信进程间通信组属性。

当这些完成后，该进程就已经被加载完毕。只要内核的进程调度器从运行队列中选中该进程，它就可以执行了。对于内核线程的创建则与进程创建有所不同，由于内核线程并不需要为它们专门分配物理页面，因此使用kernel_thread()函数来创建。

进程消亡

当用户程序的 main()函数执行完毕后，代码会返回到 C 库中。在 C 库做完相关清理操作后，最终会调用 exit()系统调用。当内核接收到该系统调用后，会根据消息包的接收者标识将消息转发给 system_task 内核线程来处理。system_task 会调用 do_exit()执行如下操作：

- 将请求进程的当前状态设置为 TASK_EXIT，避免其它服务进程再对该进程进行操作。
- 将该进程从系统运行队列中去除。
- 释放该进程代码、数据占用的内存页，释放该进程页表、页全局表等占用的内存页。
- 唤醒等待该任务接收消息的其它进程，并将其它任务从该进程的接收等待队列中去除。
- 释放进程控制块所占用的内存空间。

当完成这些工作后，该进程就从系统中被注销了，由于系统调用的请求者已不再存在，因此 system_task 不需要返回消息，而是继续等待新的消息到来。同样，内核线程的终结也调用 exit()系统调用[①]。

任务管理和进程调度

目前 MLXOS 的调度尚未支持多处理器或多核处理器[②]，虽然任务队列可以在编译时确定支持的处理器数量，但目前只针对一个处理器。

[①]目前 MLXOS 的内核线程均是运行在内核空间的服务任务，例如 clock_task，hardware_task，system_task 和 idle_task。它们在系统整个运行期间都不会退出，因此目前还没有实现内核线程退出的代码。从概念上讲，内核线程的退出和普通进程的退出相似，但不需要释放代码、数据占用的空间。

[②]虽然在代码实现上尚未支持多处理器架构，但是在内核中所有涉及锁保护的地方，都已经通过基于内存的 spin_lock/spin_unlcok 来进行互斥保护了。因此，代码框架在设计之初，已经为多处理器做了准备。

目前采用的调度策略是多级反馈队列调度。系统中根据优先级从高到低存在多个运行队列，调度器总是尝试从优先级最高的运行队列上调度任务，只有当高一级队列为空时才尝试调度低一级的队列。在经过一个时间间隔，或者除最低优先级的队列之外其它队列均为空时，就会重新按照每个任务的初始优先级将每个任务重新放置在最初的运行队列上。多级反馈队列的调度代码实现起来相当简单，从不同优先级中选择合适的任务准备调度，只需要不超过 10 行 C 代码即可实现。如下面伪代码 4-1 所示，其中 NR_SCHED_QUEUES 是多级反馈队列的级数，目前为 4 级。内核服务线程的缺省运行队列为 0 级，普通服务进程的缺省运行队列为 1 级，普通用户进程的缺省运行队列为 2 级，系统空闲线程 idle_task 运行在第 3 级队列上。当 0 到 2 级队列为空而 3 级队列上存在除 idle_task 以外的任务，内核就会根据每个任务的缺省运行队列将所有反馈队列重新布局。

```
for (i = 0; i < NR_SCHED_QUEUES; i ++)
{
        queue = &run_queues[i];
        if (!list_empty(queue))
        {
            np = queue->next;
            next_task = list_entry( np, struct task_struct, run_queue);
            break;
        }
}
```

伪代码 4-1 多级反馈队列调度 1

由于 MLXOS 是支持内核级可抢占[①]的微内核，因此当中断处理返回时，也有可能发生任务调度。在这种情况下，如下面伪代码 4-2 所示，只需将当前任务移到当前任务队列的末尾。

[①]在传统的类 UNIX 操作系统上，如果有一个更高优先级的进程需要运行时，内核会将正在运行的低优先级进程移出 CPU 让高优先级的进程运行，这称为抢占。但传统的类 UNIX 操作系统只有当进程运行在用户空间时才能抢占，如果进程通过系统调用运行在内核空间时，必须等低优先级的进程再次从内核空间返回用户空间后才能抢占，这就是内核非抢占。而 MLXOS 的设计可以在任何中断处理结束时考虑是否进行任务切换，而不仅仅是在进程从系统调用中返回时。因此，就可以在进程尚运行在内核空间时进行任务抢占。这样做的好处是可以提高内核处理任务的实时性，确保高优先级的任务尽快执行。

```
if (!list_empty(&cur_task->run_queue))
{
        /* current process is interrupted */
        queue = &run_queues[cur_task->priority];
        list_move_tail(&cur_task->run_queue, queue);
}
```

伪代码 4-2 多级反馈队列调度 2

有时候，任务运行在内核态时也需要主动放弃处理器让别的任务调度到处理器上运行，这时候内核会调用 schedule()函数来调用调度器主动发起任务切换。如果这时在系统中可运行的任务超过一个，那么就可以切换到其它可运行任务上。MLXOS 的 schedule()函数的实现比较特殊，多数操作系统都会有专门的任务切换代码，但是 MLXOS 没有。MLXOS 的任务切换是借用中断返回时的任务切换代码完成的，这样做的好处就是 schedule()函数的实现非常简单，只有一行汇编代码。相应的弊端就是每次任务切换都需要经历一次中断处理代码，效率较低。为了让 MLXOS 的代码尽可能简单，并容易让更多的初学者理解进程调度的实现细节，因此采取了性能向可读性让步的策略。如下面伪代码 4-3 所示，即为 MLXOS 的 schedule()函数实现，可以看到这个 C 函数中嵌入了一段汇编代码，这段汇编代码的功能就是调用 IRQ 0 中断，然后在中断例程返回时调用恢复中断现场前的任务切换代码，而避免了再实现一套代码。

```
inline void schedule(void)
{
#if (0x50 == IRQ0_VECTOR)
        __asm__ __volatile__(
                "int $0x50\n\t"
        );
#else
        #error "IRQ0 is not 0x50, see the source code."
#endif
}
```

伪代码 4-3 schedule()函数实现

由于 MLXOS 将时钟中断 IRQ 0 编程到了 Intel 处理器的 0x50 号中断上，因此指令“ int $0x50 ”就是主动触发 IRQ 0 对应中断，并在中断例程处理完毕返

回时发生进程切换。只用一行汇编代码来实现 schedule()函数，除了 MLXOS 之外恐怕很少有操作系统这么做。

进程切换时机

一般在三种情况下会切换进程，

- 时钟 IRQ 中断发生，当前任务时间片运行完毕
- 硬件 IRQ 中断发生，有更高优先级的任务需要执行时
- 当前任务主动调度其它任务

在传递较大尺寸的消息时，很有可能正好会碰到时钟中断被触发。虽然时钟中断的处理代码不会影响消息传递过程，但是会占用一定的处理器时间，因此会降低消息传递的效率。为了尽可能准确的测量 MLXOS 上进程间通信的效率，目前的内核采用的是无时钟中断的进程调度。而普通的进程间通信中，其它硬件中断发生也会导致进程间通信性能的降低，因此 MLXOS 目前在测试进程间通信性能时，将其它硬件 IRQ 中断的处理也关闭了。因此在目前应用进程同 tty 服务进程通信的过程中，没有任何的硬件 IRQ 中断发生的（不包括程序通过 schedule()函数触发 IRQ 0 的相应代码）。

因此在测试进程间通信性能时，MLXOS 中的进程只会在主动放弃处理器时才会发生任务切换。虽然进程切换只在这一种情况下发生，但是这样可以让处理器时间都尽可能多的耗费在进程间通信的进程中，可以更准确的测量进程间通信的性能数据[①]。从目前的运行状况来看，这种进程切换策略可以正确工作，任务的休眠、唤醒，消息在多个任务之间的传递，都可以很流畅的进行，传递大尺寸消息的速度比打开时钟中断时要快。

进程休眠和唤醒

休眠和唤醒是比较形象的说法，实际上休眠就是将特定任务从运行队列中移出，这样该任务就不会再被调入处理器运行；而唤醒就是将特定任务再次移入运行队列，这样该任务就有可能再次被调度器调入处理器运行。在 MLXOS 中进程间通信的消息是同步传递的，也就是说，只有当消息的发送和接收方都准

[①]在某些科学计算系统中，为了让运算任务尽可能多的消耗处理器时间，也会采取这种无中断的调度策略，关闭时钟中断，并禁用不相关的硬件中断。而最新版本的 Linux 2.6 内核也引入了 tickless 特性，即在进程调度中使用其它计时器来代替时钟中断，以减少时钟中断消耗的处理器时间，这样既可以提高运算任务的性能，也可以在系统空闲时节省处理器的电源消耗。

备好后，消息传递才会发生，之前消息的发送方或接收方都需要休眠等待，当消息传递完成后，相应的进程还会被唤醒。

在 MLXOS 中，进程或内核线程在以下情况下可能会进入休眠：

- 消息接收方未准备好接收消息，消息发送方转入休眠
- 消息接收方未接收到消息，消息接收方转入休眠

如面伪代码 4-4 所示，当发送方发送同步消息是转入休眠的情形，内核发现该消息传递是阻塞型的，便会调用 dequeue()函数将发送方从当前运行队列中移出。需要注意的是，将任务移出运行队列后，需要将它休眠在接收消息任务的 rcvm_queue 队列上，否则接收方在可以接收消息时找不到消息发送方，也无法唤醒该休眠任务。

```
int mini_send(struct task_struct *caller_task,    int dst_e, message *m_ptr, unsigned long flags)
{
      ... ...
      } else if (!(flags & NON_BLOCKING)) {
            ... ...
            if (caller_task->rts_flags == 0)
                  dequeue(caller_task);
            ... ...
            list_add_tail(&(caller_task->sndm_waiton), &(dst_task->rcvm_queue));
      } else {
            ... ...
      }
      return 0;
}
```

伪代码 4-4 发送消息时任务休眠

下面的伪代码 4-5 则展示了接收方转入休眠的情形，当没有任何消息到来时，内核会将接收消息的任务从运行队列中移出，并将它的 rts_flags 设置为 RECEIVING，这样当有任务向它发送消息时，就知道它准备好接收消息了。马上就可以看到休眠的任务是如何被唤醒的。

```
int mini_receive(struct task_struct *caller_task, int src_e, message *m_ptr, unsigned long flags)
{
        ... ...
        if (!(flags & NON_BLOCKING))
        {
                ... ...
                if (caller_task->rts_flags == 0)
                        dequeue(caller_task);
                caller_task->rts_flags |= RECEIVING;
                ... ...
        }
        ... ...
}
```

伪代码 4-5 接收消息时任务休眠

进程或内核线程在以下情况下会被唤醒：

- 如果发送方只是发送消息，则接收方接收消息后唤醒发送方。
- 发送方发现接收方在等待消息，则发送消息后唤醒接收方。
- 接收方等待消息时，有发送方发送消息。

第一种情况通常发送在内核收到 IRQ 中断后，发送一个通知消息给驱动服务进程，这种情况下，消息是非阻塞的，发送方发送消息后即可返回，不需要等待接收方被唤醒。第二种情况发生在同步消息传递中，如下面伪代码 4-6 所示，当发送方发送消息时，如果接收方正在等待接收消息，则内核首先复制消息，然后如果接收方正在休眠，则将其插入到其当前所在运行队列。这样的结果就是唤醒了接收方任务——稍后该任务将可能被调入处理器运行。

```
int mini_send(struct task_struct *caller_task, int dst_e, message *m_ptr, unsigned long flags)
{
        if ((dst_task->rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
          (dst_task->getfrom_e == ANY || dst_task->getfrom_e == caller_task->endpoint))
        {
                _copy_message(caller_task, m_ptr, dst_task, dst_task->messbuf);
                if ((dst_task->rts_flags &= ~RECEIVING) == 0)
                        enqueue(dst_task);
        } else if (!(flags & NON_BLOCKING)) {
                ... ...
        }
        ... ...
}
```

伪代码 4-6 发送方唤醒接收方

而下面伪代码 4-7 则展示了发送方休眠等待接收方接收消息的情况，当接收方调用 receive()系统调用接收消息时，会查看自己的休眠队列 rcvm_queue，如果不为空，则从休眠队列中的第一个发送方处复制消息。如果发送方在休眠中，则将发送方加入到相应的运行队列中，这样发送方就会在稍后被调入处理器运行。

```
int mini_receive(struct task_struct *caller_task, int src_e, message *m_ptr, unsigned long flags)
{
        ... ...
        while(list_p != &caller_task->rcvm_queue)
        {
                task_ptr = container_of(list_p, struct task_struct, sndm_waiton);
                if (src_e == ANY || src_p == task_ptr->proc_slot)
                {
                        _copy_message(task_ptr, task_ptr->messbuf, caller_task, m_ptr);
                        if ((task_ptr->rts_flags &= ~SENDING) == 0)
                                enqueue(task_ptr);
                        list_del_init(list_p);
                        return 0;
                }
                list_p = list_p->next;
        }
          ... ...
}
```

伪代码 4-7 接收方唤醒发送方

4.2.4 可信进程间通信组

基于进程迁移的地址空间复用

通过基于进程迁移的地址空间复用，才能够让多个进程复用同一个线性地址空间，这是构成可信进程间通信组的前提条件。迁移进程的请求可以由任何的应用程序提出，但最终由 serverroot 服务进程决定是否进行。serverroot 服务进程如果决定进行进程迁移，就会调用系统调用 server_migrate()指明哪个进程要如何迁移。由于内存管理模块可以动态标识进程可用地址空间，因此在迁移过程中，实际的操作者 system_task 要做的事情主要就是正确的设置被迁移进程的地址空间基址和长度，以及相应的可信进程间通信组标识。内存管理相关细节在前面段落已经陈述，下面会详细说明对可信进程间通信组的标识和识别。

可信进程间通信组标识和表示

可信进程间通信组由主进程在进程表中的编号标识。

所有服务进程构成一个可信进程间通信组，该组的主进程是所有运行在共享区域的小地址空间中线性地址最低的那个。

由数据结构 tpig_table_entry 来标识一个可信进程间通信组，所有的 tpig_table_entry 结构成系统中的可信进程间通信组表，称为 tpig_table。下面的伪代码 4-8 展示了 tpig_table_entry 和 tpig_table 的结构定义。

```
struct tpig_table_entry
{
        int nr;
        pgd_t *pgd;
        unsigned long long pgd_version;
};

struct tpig_table_entry tpig_table[NR_TASKS + NR_PROCS];
```

伪代码 4-8 tpig_table_entry 和 tpig_table 结构定义

由于任何一个进程都可能作为某一个可信进程间通信组的主进程，因此可信进程间通信组表中的表项个数和进程表相同，均为 NR_TASKS + NR_PROCS 个（其中 NR_TASKS 是内核中内建服务线程个数，目前为 4，NR_PROCS 是包括服务进程在内的其它进程最大数量，目前为 4096）。在 32 位系统上，tpig_table_entry 占用 8 个字节内存，因此整个 tpig_table 大小为 4100 项时占用内存 32800 字节（约 32K）[①]。

可信进程间通信组中所有进程均共享主进程的页全局目录，对应的 tpig_table 表项中的 pgd 指针指向主进程页全局目录所在物理地址。对于独立的进程，它们构成只有一个进程的可信进程间通信组，对应 tpig_table 表项中的 pgd 成员同样也指向该进程自己的页全局目录表物理地址。在每个进程的 task_struct 结构中，有一个成员 int tpigid 表明该进程所属可信进程间通信组在可信进程间通信组表中的位置。

由于同一个可信进程间通信组中的进程共享相同的页全局目录表，因此组内进程切换时不需要重载页全局目录表地址。不同组的进程间切换时，MLXOS 不

[①] 目前 MLXOS 设置系统中最多可同时运行 4096 个进程，这是为以后运行在多处理器多核环境下做准备。在目前的实际环境中，一个系统同时运行的进程通常不超过 256 个，因此如果在编译内核时适当设置最大进程个数，tpig_table 的尺寸会减小很多。

像传统类 UNIX 操作系统那样重载调入进程的页全局目录表物理地址，而是将调入进程所在可信进程间通信组对应的 tpig_table 表项中的 pgd 成员加载到 CR3 寄存器中。这样做实质上加载的是可信进程间通信组主进程的页全局目录物理地址，但从形式上看，是该进程组共享的页全局目录物理地址被加载到 CR3 中了。

可信进程间通信组在进程切换中

在进程切换时，MLXOS 内核调度器 scheduler()首先决定将哪个进程调入处理器。在调度器决定了下一个运行进程后，它会调用 update_context()来尝试加载新的页全局目录。在第三章的伪代码 3-1 中，论文曾经给出了基于 TPIG 集合模型的抽象伪代码。在下面的伪代码 4-9 中，将更加具体的使用 C 语言风格的伪代码展示 update_context()函数中可信进程间通信组在进程切换中的处理流程。

```
Void update_context(struct task_struct *prev, struct task_struct *next)
{
        ... ...
10      if (prev_tte->pgd_version == kernel_pgd_version &&
11          next_tte->pgd_version == kernel_pgd_version)
12      {
13          if (prev->tpigid == next->tpigid)
14              return;
15          if (IS_SMALL_TASK(prev))
16          {
17              if (!IS_SMALL_TASK(next) &&
18                  small_tasks_parent_tpigid == next->tpigid)
19                  return;
20          } else {
21              if (IS_SMALL_TASK(next))
22              {
23                  if (small_tasks_parent_tpigid != prev->tpigid)
24                      small_tasks_parent_tpigid = prev->tpigid;
25                  return;
26              }
27          }
28      }
29
30      lazy_update_kspace_pagetables(prev_tte, next_tte);
31      load_cr3(tte->pgd);
}
```

伪代码 4-9 update_context()函数伪代码示例

如上伪代码中，10～11 行是分别比较调入和调出进程所在可信进程间通信组的页全局目录版本是否和当前内核的页全局目录版本相同。这是因为 MLXOS 采用了懒惰的页全局目录更新策略（在后面会进行说明）。当调入进程和调出进程的页全局目录版本和当年内核页全局目录版本一致后，13~14 行伪代码判断被调出进程和调入进程是否属于同一个可信进程间通信组，如果是，则 update_context()函数直接返回，不刷新 TLB。15～19 行如果调入调出进程不属于同一个可信进程间通信组，则判断被调入进程所属的可信进程间通信组是否是被调出进程所属可信进程间通信组的父组[①]，则 update_context()函数直接返回，也不需要刷新 TLB。伪代码 13～19 行的操作实际上就对应伪代码 3-1 中如下语句：

```
10 if  k == m  or  TPIG(p_k) ∈ TPIG(p_m)
11        then   return;
```

伪代码 4-9 中接下来的 21～25 行是判断调入进程所在可信进程间通信组是否由运行在共享区域的具有内核优先级的服务进程组成（即 TPIG(S)），如果判断成功则将被调出进程所属的用户态优先级的可信进程间通信组设置为 TPIG(S)的父组，然后 update_context()函数同样直接返回，仍然不需要刷新 TLB。这 4 行代码实际上对应的就是伪代码 3-1 中如下语句：

```
12        else   if  p_prev ∉ S  and  p_next ∈ S
13               then  TPIG(pprev) ← TPIG(pprev)  ∪ TPIG(S);
14               fi;
```

如果伪代码 4-9 中以上判断都失败，则执行 30～31 行代码。29 行调用 lazy_update_kspace_pagetables()函数来懒惰升级调入和调出进程所在可信进程间通信组地址空间中具有内核权限部分的页全局目录表项。31 行调用 load_cr3()函数加载调入进程所属可信进程间通信组的地址空间，该操作将刷新 TLB 中所有非全局的表项。而伪代码 4-9 中的这两行代码实际对应了伪代码 3-1 中的如下一行：

```
15        else   flush TLB
```

[①]在 MLXOS 系统中，只有由所有运行在共享区域具有内核优先级的服务进程构成的可信进程间通信组可以作为其它可信进程间通信组的子组。因此首先判断被调出进程是否为 TPIG(S)，如果判断失败，则不需再进行父子进程组的判断。

通过将 C 风格的伪代码 4-9 和抽象表达式风格的伪代码 3-1 进行主行比对，就可以了解可信进程间通信组的思想是如何设计和实现在 MLXOS 原型操作系统中。

由伪代码 4-9 中可以看到，当调出进程和调入进程属于同一个可信进程间通信组，或调入进程所属可信进程间通信组是调出进程所属可信进程间通信组的父组时，update_context()函数直接返回，而不会刷新 TLB。同样在某个可信进程间通信组切换到 TPIG(S)时，在设置 TPIG(S)的父组之后，也不需刷新 TLB。由于普通进程的页都不是全局的，在进程切换中引入可信进程间通信组的结果就是，同一个可信进程间通信组中的进程之间切换时，不会切换页全局目录表地址，已经访问过的<线性地址，物理地址>映射关系仍然保留在 TLB 中。由于 TPIG(S)共享内核的页全局目录表 swapper_pg_dir，组内所有进程的内存页，都在共享的页表中设置为全局页。由于其它所有可信进程间通信组的地址空间中对应内核态空间的页全局目录表项与 swapper_pg_dir 对应部分相同，因此当其它可信进程间通信组中的进程切换到 TPIG(S)中的进程时,同样不需要刷新 TLB，只设置 TPIG(S)的父组即可。这样的结果就是大大提高了处理器 MMU 命中 TLB 的几率，从而提高了整个可信进程间通信组内的进程间通信性能。

TPIG(S)可以是任何一个可信进程间通信组的子组，目前在 MLXOS 设计中，也只有 TPIG(S)可以作为某个可信进程间通信组的子组。当其它可信进程间通信组中的进程切换到 TPIG(S)中的某个进程时，不需要重新加载 TPIG(S)的页全局目录表地址，被调出进程所在的可信进程间通信组自动成为 TPIG(S)的父组。当任务调度发生在 TPIG(S)和其它可信进程间通信组的进程间时，如果前者是后者的子组，也同样不需要重新加载 CR3 寄存器。这是因为 TPIG(S)和它父组中的进程彼此在整个生命周期内访问的线性地址空间不会有重叠，从父组调度到 TPIG(S)后，仍使用父组的页全局目录，之前进程在 TLB 中的表项不会被污染，当重新调度回父组后,没有必要再次加载相同的页全局目录。伪代码 4-9 中 17~19 行就是对这种情况的处理。

回想 Jochen Liedtke 提出的地址空间复用方法中，曾经说过如果某一个普通进程切换到运行在小地址空间中的进程后，当再次从某个小地址空间中的进程切换回普通进程时，如果该进程恰好是之前那个普通进程的话，则不需要重新加载 CR3 寄存器。TPIG(S)在进程切换中的处理策略受 Jochen Liedtke 思想的启发，针对多（核）处理器环境有所改进：

- Jochen Liedtke 的方法只是针对小地址空间中的服务进程切换到某一个普通进程的情况进行优化。可信进程间通信组对 TPIG(S)和其父组中多个进程的切换情况进行了优化。
- 在多核处理器情况下，如果某个进程在切换到小地址空间的进程后，被迁移到其它处理器上运行，那么当其它处理器上的小地址空间进程切换到该进程时，为了避免 MMU 在 TLB 中命中混乱，必须将进程的页全局目录加载到 CR3 中。因此如果继续沿用 Jochen Liedtke 在单处理器上的方法，就会出现严重错误。而采用可信进程间通信组后，首先调度器会尽量将相同组的进程调度到同一个处理器上运行。其次在每个处理器上，TPIG(S)中的进程切换到其父组中的进程，仍然可以简洁的通过父子组关系来决定是否避免重载 CR3 寄存器。

因此在可信进程间通信组的设计中，必然会考虑到多处理器或多核处理器的情况。这就是为什么即便目前 MLXOS 只支持一个处理器，仍然实现了 spin_lock 自旋锁的支持，并在声明重要数据时都考虑到多处理器的情况[①]。

懒惰的页全局目录更新

在 MLXOS 中，内核态的线性地址被所有进程共享，因此进程页全局目录中对应 2G~4G 的表项与内核页全局目录表（swapper_pg_dir）中对应位置是相同的。这样做的好处是发生用户态和内核态的优先级切换时只需要加载新的段属性即可，而在内核态实现进程间切换时也更加简洁。因此，当为新创建进程分配页全局目录表时，MLXOS 内核会连带将内核空间对应的页全局目录表项一并复制。

但是在 MLXOS 中，由于服务进程可以被迁移到内核态的共享区域（2G～3G）中，当迁移发生时，MLXOS 就会修改相应位置的页全局目录项。这时，内核的页全局目录就与系统中所有进程的页全局目录出现不一致，当普通进程切换到运行在内核态小地址空间中的服务进程时就会触发错误的缺页中断。因此需要更新进程的页全局目录表中对应内核空间的部分。但是当系统中运行的进程数量

[①]例如在声明记录当前处理器 TPIG(S)组的父组标识号时，采用 int TPIG_S_PARENT [CPU_NR]的形式，而引用时采用 TPIG_S_PARENT [current_processor()]的形式。虽然 CPU_NR 被定义为 1，而 current_processor 也被定义为 0，但今后一旦支持多处理器时，整个代码框架不需要进行太大改动。

较多时，就要更新所有进程的页全局目录，如果这种更新频繁发生，那性能损失是相当大的。

解决的办法就是采用懒惰的页全局目录更新策略。在两次更改内核空间页全局目录表项的间隔中，系统中的所有进程并不会都被调度运行一次，因此没有必要在内核空间页全局目录更改后就立刻将系统中所有进程的页全局目录更新，这就是懒惰更新策略的依据。MLXOS 实现的懒惰页全局目录更新方法如下：

在可信进程间通信组表的成员结构 struct tpig_table_entry 中定义了 pgd_version，用来标识当前可信进程间通信组内所有进程复用的地址空间所对应的页全局目录表中内核部分的版本。同时在内核中有一个全局变量 kernel_pgd_version，来维护内核页全局目录表 swapper_pg_dir 的版本。这两个变量均为 64 位，即使在主频 100GHz 的处理器上（现在还没有这么高主频的处理器产品上市）最短也可以提供超过 8 千年的时间间隔，目前看是足够了。在创建新的可信进程间通信组时，会设置初始的 pgd_version 为 0。而每次内核页全局目录表 swapper_pg_dir 的内容发生变化时，并不立刻更新进程的页全局目录，而是将更新工作延迟到某一个进程被调度运行时，这就是懒惰更新这个名字的由来。当某一个进程被选择调入处理器运行时，参看伪代码 4-9 在 update_context()函数中编号 10 和 11 两行中会比较调出和调入进程的 pgd_version 是否和 kernel_pgd_version 相等，如果不相等，则调用 lazy_update_kspace_pagetables()函数对不相等的页全局目录表进行更新。

这样一来，只有涉及到进程调度的进程才会在被调度时更新页全局目录，就大大减少了更新的范围，避免了由不必要的页全局目录表更新而导致的额外开销。

4.2.5 可迁移进程的状态迁移

无论是服务进程或普通进程，在迁移到不同的地址空间时，为了不中断对外的通信状态，必须同时对进程的状态数据进行迁移。这里所说的进程状态数据，主要是指在进程迁移过程中，为了不中断进程对其它进程的服务，所必须同步迁移的内部数据结构。譬如对于 MLXOS 中的 tty 驱动来说，必须迁移的状态数据就是当前 tty 终端的显示内容、光标位置等。

每一个服务进程的具体实现不同，很难设定统一的状态数据同步方法，甚至连定义一个标准的系统调用接口都是非常困难的。因此具体某一个服务进程

在地址空间迁移过程中如何同步状态数据，应当由该服务进程的开发人员自己灵活实现。由于状态数据的迁移仅仅发生在同一个服务在不同地址空间的两个运行实例之间，因此其过程和接口可以在服务进程内私有，不必公开。

因此 MLXOS 在迁移进程状态时只定义了标准调用接口 GET_MIGSTATE。serverroot 管理进程在对某个进程进行迁移操作前，首先会向该进程发送消息 GET_MIGSTATE，如果返回消息无效，或不具有进程迁移能力，那么 serverrroot 进程将不再继续进程迁移操作。如果被迁移进程返回可以迁移，那么消息中会同时附带保存有被迁移进程状态数据的内存地址和长度。serverroot 将状态数据保存后，即可将该进程迁移到新地址空间中的新运行实例中。新的运行实例启动后，会首先向 serverroot 发送 GET_MIGSTATE 消息获取缓存的状态数据，进一步初始化并运行。

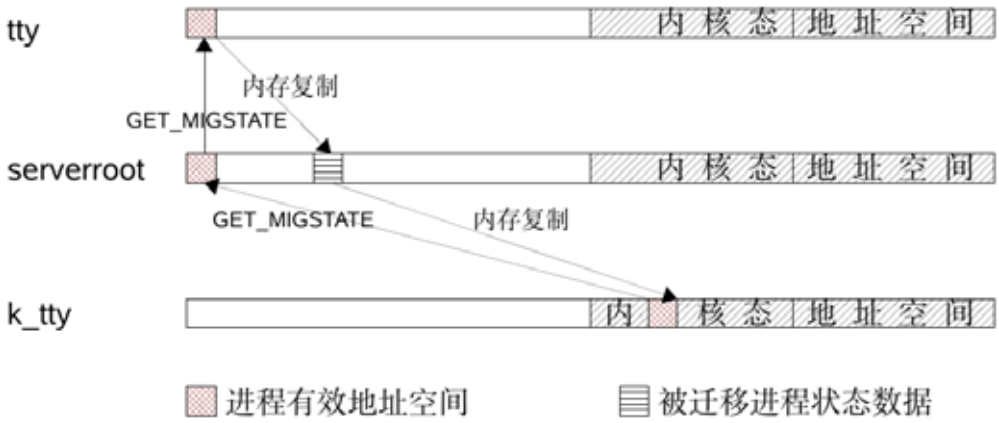


图 4-1 可迁移进程状态迁移

上图说明了 tty 服务进程在迁移过程中进程状态数据迁移的主要步骤。其中 tty 进程从用户空间迁移到共享区域的小地址空间中，迁移后的进程实例名为 k_tty 以示区别。serverroot 在迁移前向 tty 发送 GET_MIGSTATE 消息，并将返回的状态数据复制到图中横线阴影表示的区域中。当 tty 进程被迁移到实例 k_tty 后，k_tty 进程会再次向 serverroot 发送 GET_MIGSTATE 消息将之前实例的状态数据同步到自身。serverroot 在返回 GET_MIGSTATE 请求后将 tty 进程的状态数据内存释放。

4.3 基于 tty 服务的情景示例

本章前部分详细说明了对基于进程迁移的地址空间复用和可信进程间通信组的设计思想和伪代码实现。为了让读者更清晰连贯的理解上述内容，本节以 tty 服务为对象，通过情景示例的方法，来说明基于进程迁移的地址空间复用和可信进程间通信组的主要运行流程。相关具体代码实现，可以在附录 C 给出的地址中得到。

这里描述的情景示例是 serverrrot 将 tty 进程迁移到自己的内核态地址空间中，这样由 serverroot 构成的可信进程间通信组在进程间通信时就有可能成为由运行在内核空间的 k_tty 进程构成的 TPIG(S)的父组。这样在进程间通信过程中进行消息传递时，就可以避免刷新 TLB。

该情景示例展示了在进程迁移过程中，在普通应用程序看来，TTY 服务并没有中断。serverroot 进程在 TTY 服务从 tty 进程迁移到 k_tty 进程后，前后发送的打印消息都可以正确的显示在字符终端上。

4.3.1 tty 进程迁移到 serverroot 的内核态地址空间

将 tty 进程迁移到 serverroot 进程的用户地址空间的过程如下，

- serverroot 获取 tty 进程状态数据
  serverroot 调用 get_migstate()函数向 tty 进程发送消息，获取其状态数据。并将得到的状态数据保存在自己地址空间中的缓存区里。
- serverroot 发起系统调用 server_migrate()
  该系统调用格式为 int server_migrate(int tpig_id, int server_e)，其中 tpig_id 是服务进程所要迁入的可信进程间通信组标识号，server_e 是被迁移进程的端点号[①]。在这里 tpig_id 是 idle_task 在系统中的端点号，server_e 是 tty 服务进程在系统中的端点号。该系统调

[①]MLXOS 中进程间通信的接收方和发送方是通过进程在进程表中的位置（position）确定的，但是考虑到某一个进程终结后新的进程被分配在相同位置的情况，因此在进程间通信中标识接收方和发送方时不直接使用进程在进程表中的位置，而是在这个数值上附带一个版本（generation）号，这个合成的数值被称为端点号（endpoint）。当用 p 代表进程在进程表中的位置，g 代表版本，e 代表端点号时，这三者的关系可用如下伪代码标识：

```
ENDPOINT(g, p)    ←    (g) * ENDPOINT_GENERATION_SIZE + (p)
ENDPOINT_G(e)     ←    (e) / ENDPOINT_GENERATION_SIZE
ENDPOINT_P(e)     ←    (e) % ENDPOINT_GENERATION_SIZE)
```

用的结果就是 serverroot 将 tty 服务迁移到共享区域的小地址空间中以内核态优先级运行。

- 内核接收到请求后将消息转发给 system_task 内核服务线程
- system_task 加载 k_tty 进程

  为了简化系统开发工作，k_tty 进程的 ELF 文件事先已经被重定位到预定的线性地址上，这样就避免了在 MLXOS 上实现复杂的地址重定位和动态链接的工作。k_tty 进程的 ELF 文件通过 GRUB 引导器在系统启动时预先加载到内存中。system_task 调用 find_migrate_mb_mod()函数从内存中搜索以 GRUB module 模式加载的 k_tty 的 ELF 文件，如果搜索到 k_tty 的 ELF 文件，就调用 do_exit()函数将前一个 tty 进程终止，否则返回系统调用失败给 serverroot。接着 system_task 调用 load_multiboot_module()函数根据 k_tty 的 ELF 文件创建新的 k_tty 进程。
- system_task 设置 k_tty 进程的 tpigid

  在内核中内建了 idle_task、system_task、clock_task 和 hardware_task 四个内核线程，虽然这 4 个内核线程直接运行在内核空间而不是共享区域的小地址空间中，MLXOS 也将它们作为可信进程间通信组 TPIG(S)中的进程来看待。由于目前 clock_task 和 hardware_task 还没有加入 MLXOS 系统中，当 TTY 服务迁移到 k_tty 进程后，就构成了以 idle_task 为主进程、成员数量为 3 的 TPIG(S)。system_task 需要将 k_tty 进程和 serverroot 进程的端点号作为参数调用 set_tpigid()函数，来设置 k_tty 进程的可信进程间通信组。

  set_tpigid()函数的操作为，(1) 将 k_tty 进程的 task_struct 中的 tpigid 设置为 idle_task 进程的端点号；(2) 将 k_tty 进程对应于自己 mm_struct 中的 base 到 base+limit 之间的页全局目录表项复制到 TPIG(S)的页全局目录表对应位置，并释放 k_tty 原先页全局目录占用的内存页，并将自己 mm_struct 中的 pdg 指向 TPIG(S)的页全局目录物理地址；(3) 将 TPIG(S)在 tpig_table 表对应项中的 nr 计数器加 1，表示当前组中有 2 个进程，并将内核空间的页全局目录版本变量 kernel_pgd_version 加 1。
- k_tty 进程获取状态数据

当 system_task 返回后，serverroot 得到 server_migrate()调用成功的返回值，即可向新的 k_tty 进程发消息了，由于 k_tty 仍然使用原先 tty 进程在进程表中的位置，因此 serverroot 在与 k_tty 进程通信时不需要调整发送接收方[①]。在完成进程迁移后，serverroot 重新进入等待接收新消息的状态。当 serverroot 休眠后，k_tty 进程得到调度，完成初始化后，会向 serverroot 发送 GET_MIGSTATE 消息。serverroot 接收到消息后，将保存的之前 tty 进程的状态数据返回给 k_tty 进程，然后将缓存区释放，并再次进入等待新消息的状态。k_tty 在得到状态数据后，将这些数据更新到当前环境中，然后进入工作状态。如果 k_tty 无法从 serverroot 处得到状态数据，则进行缺省的环境初始化（如清屏、光标复位等），然后进入工作状态。

经过上述步骤后，k_tty 进程就被迁移到了共享区域的小地址空间中以内核态运行，属于 TPIG(S)中的一员。

4.3.2 serverroot 和 k_tty 的进程间通信

serverroot 与 tpig_tty 通信的消息传递过程中，会发生进程切换。整个消息传递的过程如下：

- serverroot 创建消息

  serverroot 在自己的地址空间开辟一块内存区域存放要发送的消息，由于 MLXOS 中的消息是同步传递的，因此这段内存区域可以从栈中分配。分配好空间后，可以根据进程间通信的需要填充相应的消息内容。

- serverroot 发送消息

  根据消息的类型，serverroot 可以使用不同的系统调用来发送消息，譬如 SEND 仅仅发送消息，SENDREC 在发送消息后还要接收 k_tty 返回的消息，NOTIFY 发送消息后无论接收方是否准备好都会立刻返回。本例中 serverroot 调用 SEND 向 tty 发送消息，消息的内容为一个字符串“hello，world”，操作为写入。

- 内核转发消息

[①]由于进程迁移而在进程表中相同位置处注册新进程是一个例外，在这种情况下，该进程的版本号不会增加。否则内核会因为端点号中的版本不匹配而认为消息接收方或发送方无效。

当内核捕捉到系统调用后，会调用 sys_call()函数。sys_call()函数根据消息中的进程间通信操作（SEND、SENDREC、NOTIFY 或 RECEIVE）进行相应操作。在示例中 serverroot 使用的是 SEND 操作，则接下来会将消息的发送方设置为 serverroot[①]，然后检查 k_tty 进程是否准备好接收消息。根据 k_tty 进程的状态不同，处理方法不同：(1) k_tty 进程准备好接收消息，则内核会调用 copy_message()函数将消息从 serverroot 地址空间中直接复制到 k_tty 地址空间中等待接收消息的缓存中,然后唤醒 k_tty 进程 ;(2)k_tty 进程正忙,则内核会将 serverroot 进程 task_struct 结构中的 messbuf 赋值为消息地址，然后将 task_struct 中的 sndm_waiton 队列节点加到 k_tty 进程 task_struct 结构中的 rcvm_queue 队列中，然后将 serverroot 任务挂起（即将 serverroot 进程休眠在 k_tty 进程的消息等待队列上)。在第 (2) 中情况下，一旦 k_tty 可以接收消息，内核会从 k_tty 进程的 rcvm_queue 队列中摘取 serverroot，调用_copy_message()函数从该进程 messbuf 指向的地址将消息复制到 k_tty 进程的消息接收缓存中。

- 从 serverroot 进程切换到 k_tty 进程

  一旦内核将消息复制到 k_tty 的地址空间，并将其唤醒时，调度器 scheduler()就会选择 k_tty 进程调入处理器。这时就会发生进程调度。调度过程在 7.2.4 中已经阐述过这里不再赘述。一旦 k_tty 调入处理器后，就会执行接收消息返回后的代码，对 serverroot 发送来的消息进行处理。

- k_tty 处理消息

  在 MLXOS 中，TTY 服务的工作非常简单，就是将接收到的消息内容直接从当前光标位置写入终端显存，这样相应的内容就可以从显示器上被显示出来。当 k_tty 进程接收到 serverroot 发送来的消息后，由于消息包只是描述消息的结构，本身并不包含消息内容，因此需要根据消息包中的地址将消息内容从 serverroot 的地址空间中复制到 k_tty 的地址空间中[②]。k_tty 进程中的 get_request_buf()函数会调用系统调用函数

[①]由于应用程序自己无法知道进程当前自身的端点号，因此设置消息发送方的操作必定是在内核中进行的。

[②]虽然 k_tty 和 serverroot 共享一个地址空间，但是这两个进程各自并无法确定自己是运行在独立的地址空间中还是复用其它进程的地址空间。因此在接收到消息包后，必须将消息的内容复制到自己可见的地址空间中。

memcpy_cross_address_space() 由 system_task 将消息内容从 serverroot 的可见地址空间复制到 k_tty 可见的地址空间中。当消息内容复制完成后，k_tty 进程调用_tty_echo()函数将消息内容复制到映射在 k_tty 可见地址空间中的显存区域中。由于消息内容是“ hello world ”字串，这时就可以从显示器的终端上看到“ hello world ”这行英文字符。

- 返回成功给 serverroot

  当 k_tty 进程将消息内容写入显存后，返回成功给 serverroot 进程。serverroot 进程收到返回消息后，继续进行后续程序。

- 发送较大消息

  当 serverroot 发送给 k_tty 的消息比较大时，就可能需要多次消息传递才能完成数据传送。在消息传递过程中，进程切换的顺序为：serverroot（发送消息）→ k_tty（获取消息后请求数据）→ system_task（返回数据）→ k_tty（返回消息）→ serverroot。并重复上述循环直至数据传递结束。当 serverroot 切换到 k_tty 时，serverroot 所在的可信进程间通信组成为 k_tty 所在的 TPIG(S)的父组，此时不需刷新 TLB。当 k_tty 与 system_task 之间切换进程时，由于二者属于同一个可信进程间通信组 TPIG(S)，所以也不需要刷新 TLB。当 k_tty 进程切换到 serverroot 时，由于 serverroot 所在可信进程间通信组是 k_tty 所在的 TPIG(S)的父组，也不需要刷新 TLB。这样一来，在传递消息的整个过程中都不需要进行地址空间的切换。

经过上述步骤，就完成了一次从 serverroot 进程到 k_tty 进程的进程间通信。

4.4 基于 vfs 服务的情景示例

这里之所以要专门描述基于 vfs 服务的情景示例，是由于在下一章的性能测试方案中会将 VFS 服务迁移到 serverroot 的用户态地址空间中，因此有必要将测试方案中进程间通信的情景进行阐述。需要说明的是，在实际运行环境中，是不会将 VFS 系统服务迁移到 serverroot 的用户态地址空间中的，测试方案是为了简化测试环境，尽可能减少代码开发工作，因此才设计了让 VFS 服务和

serverroot 复用用户态地址空间的方案。虽然接收 serverroot 发送来消息的服务进程名字为 VFS，但是除了与 serverroot 进程交互消息外，该进程目前还没有其它任何功能。该情景示例中着重介绍与之前示例不同之处，而相同之处则会一笔带过。

4.4.1 vfs 进程迁移到 serverroot 的用户态地址空间

对于服务不间断性的示例，在基于 tty 服务的情景示例中已经展示。基于 vfs 服务的示例主要是为了测试可信进程间通信组对进程间通信性能的提升，在编写性能测试的 vfs 服务时，仅仅实现了消息传递部分，没有定义和实现状态数据转移的接口。因此在迁移进程的过程中，省略了迁移状态数据的过程。

进程迁移到 serverroot 进程的用户地址空间的过程如下，

- serverroot 发起系统调用 server_migrate()，tpig_id 参数为 serverroot 进程自己的端点号。作为一个特殊的服务进程，serverroot 在进程表中的位置是在编译时确定的，因此在调用 server_migrate()时就可以用宏常量来指明。server_e 是 vfs 服务进程在系统中的端点号，同样该端点号是在编译时确定的，可以由宏 FS 来引用。系统调用的结果就是将 FS 服务迁移到 serverroot 自己的用户态地址空间中。
- 内核接收到请求后将消息转发给 system_task 内核服务线程
- system_task 加载 tpig_vfs 进程
  当发现 tpig_id 是指向一个用户态进程时，system_task 会尝试去调用被迁移进程的带有 tpig_前缀的程序[①]，即 tpig_vfs。
- system_task 设置 tpig_vfs 进程的 tpigid
  system_task 调用 do_server_migrate()函数在创建了 tpig_vfs 进程后，在函数返回之前会调用 set_tpig()来设置 tpig_vfs 的可信进程间通信组属性。这里所做的操作与 k_tty 迁移到内核空间所做操作基本相同。不同之处在于会将 tpig_vfs 进程的 tpigid 设置为 serverroot 进程的端点号，并且在将 tpig_vfs 的页全局目录和 serverroot 的页全局

[①]同 k_tty 一样，tpig_vfs 是预先链接到预定的线性地址。在 MLXOS 的地址空间分布中，用户态进程的 1~2G 地址空间是预留给迁移到以该进程为主进程的可信进程间通信组的其它进程的。在本示例中，tpig_vfs 被链接到 1G 开始的线性地址处，占用了 64M 字节的地址空间。这样做的原因同样是为了减少工作量，尽快开发原型系统。

目录合并后，不升级全局变量 kernel_pgd_version。之所以不修改 kernel_pgd_version 值是因为地址空间的变化仅仅局限在以 serverroot 为组主进程的可信进程间通信组内，组外进程不会受到影响。

这样 tpig_tty 和 serverroot 进程组成一个可信进程间通信组，并且两个进程复用的是 serverroot 的用户态地址空间，均以用户态优先级运行。

4.4.2 serverroot 和 tpig_vfs 的进程间通信

serverroot 与 tpig_tty 通信过程中的消息传递，与 serverroot 和 k_tty 相近，主要的不同在于消息传递是发生在两个属于同一个可信进程间通信组的用户态进程之间的。

- serverroot 创建消息、发送消息
  serverroot 在自己的地址空间的栈中开辟一块内存区域存放要发送的消息。本例中它调用 SEND 向 tpig_vfs 进程发送消息，消息中的内容指向一个 4K 字节的内存块。
- 内核转发消息
  当内核捕捉到系统调用后，会调用 sys_call()函数。sys_call()函数根据消息中的进程间通信操作（SEND、SENDREC、NOTIFY 或 RECEIVE）进行相应动作。在示例中 serverroot 仍然使用 SEND 操作，内核将消息的发送方设置为 serverroot 后,检查 tpig_vfs 进程是否准备好接收消息。根据接收方状态不同，同样会有两种不同的处理方法：(1) tpig_vfs 进程准备好接收消息，则内核会调用 copy_message()函数将消息从 serverroot 的有效地址空间中直接复制到 tpig_tpig 的有效地址空间中等待接收消息的缓存中，然后唤醒 tpig_vfs 进程；(2) tpig_vfs 进程正忙，则内核会将 serverroot 进程休眠在 tpig_vfs 进程的消息等待队列上。同样在第（2）中情况下，一旦 tpig_vfs 可以接收消息，内核会从 tpig_vfs 进程的 rcvm_queue 队列中摘取 serverroot，调用 _copy_message()函数从该进程 messbuf 指向的地址将消息复制到 tpig_vfs 进程的消息接收缓存中。
- 从 serverroot 进程切换到 tpig_vfs 进程

当 tpig_vfs 接收到消息后，调度器 scheduler()就会选择 tpig_vfs 进程调入处理器，这时就会发生进程调度。一旦 tpig_vfs 调入处理器后，就会执行接收消息返回后的代码，对 serverroot 发送来的消息进行处理。调度过程在 7.2.4 中已经阐述，这里的不同在于消息传递中的进程切换发生在两个属于相同可信进程间通信组的用户态进程之间。

- 返回成功给 serverroot

  当 tpig_vfs 读取消息后，返回成功给 serverroot。这样 serverroot 可以发送后续消息或进行其它工作。

- 发送较大消息

  当 serverroot 发送给 tpig_vfs 的消息比较大时，同样就可能需要多次消息传递才能完成数据传送。在消息传递过程中，进程切换的顺序为：serverroot（发送消息）→ tpig_vfs（获取消息后请求数据）→ system_task（返回数据）→ tpig_vfs（返回消息）→ serverroot。并重复上述循环直至数据传递结束。当 serverroot 切换到 tpig_vfs 时，由于这两个进程属于相同的可信进程间通信组，此时不需刷新 TLB。当 tpig_vfs 与 system_task 之间切换进程时，以 serverroot 为主进程的可信进程间通信组自动成为 system_task 所在的 TPIG(S)组的父组，所以也不需要刷新 TLB。当 tpig_vfs 进程切换到 serverroot 时，同组进程之间仍然不需要刷新 TLB。这样一来，属于相同可信进程间通信组的两个用户态进程由于复用了相同的地址空间，在传递消息的整个过程中都不需要进行地址空间的切换。

经过上述步骤，就完成了一次从 serverroot 进程到 tpig_vfs 进程的进程间通信。比较 serverroot 分别与 k_tty 和 tpig_vfs 通信的情景，可以看到，虽然 tpig_vfs 没有运行在内核空间，但是通过复用地址空间后可以达到相近的性能效果。

4.5 小结

本章对基于进程迁移的地址空间复用和可信进程间通信组的设计与实现做了详细说明，并结合两个情景示例，对进程迁移和可信进程间通信组内进程通信的执行步骤做了进一步的描述。通过本章内容，读者应该可以对论文工作的

主要设计思想，和实现方法，及其在进程间通信中对性能的提高在概念上有了清楚的认识。

到本章为止，论文就 Jochen Liedtke 的地址空间复用方法在现代处理器架构上的局限性，提出了改进思路，并说明了相应的设计与实现。结合 MLXOS 示例代码，应当可以对基于进程迁移的地址空间复用和可信进程间通信组的内在实现有更具体的理解。因此，对于设计和实现方面的阐述到此为止。下一章，我们将看到论文工作开发的微内核操作系统原型 MLXOS，在采用基于进程迁移的地址空间复用和可信进程间通信组方法前后，在进程间通信中的性能差异。论文提出的方法对微内核进程间通信性能的提高，将会从下一章的测试数据中看到。

第五章　实际测试与分析

本章将对可信进程间通信组内的进程之间消息传递的性能进行实际测试和分析。为了对比性能提高的提高效果，同样也会对运行在独立地址空间的两个进程之间消息传递的性能进行测试。

5.1 实验方案与测试方法

5.1.1 测试环境

在测试中记录的时间，不是绝对时间，而是相对时间。这里的相对时间是指处理器周期，即 TSC 寄存器中的数值。TSC 称为时间戳计数器（Time Stamp Count），记录了从机器上电处理器复位后的周期累加数值，每过一个处理器周期就自动加 1。该数值与时间无关，而和处理器主频有关，例如当处理器主频为 1GHz 时，每秒钟 TSC 计数器数值增加为 1,000,000，计数器每增加 1 表示百万分之一秒时间。从奔腾处理器后，Intel 处理器引入了 RDTSC 指令，可以读取 64 位的 TSC 计数器数值。这里的性能测试就是使用 TSC 作为性能计数的工具，所有的性能数据都是以处理器周期为单位的。

测试中使用的计算机相关配置如下：

表 5-1 性能测试硬件配置

处理器	Intel(R) Core(TM)2 CPU U7600
处理器主频	1.20GHz
1 级 cache	大小 32KB，操作模式 Write Back，连接方式 8-way Set-associative
2 级 cache	大小 2MB， 操作模式 Write Back，连接方式 8-way Set-associative
内存总线速度	DDR, 32 位
内存容量	1 x 2048MB

5.1.2 实验方案

由于 MLXOS 的消息传递目前仍然是基于内存复制实现的，因此随着消息尺寸的增加，复制消息的时间也会线性的正比增长。当传送消息尺寸过大时，内存复制所消耗的时间将在整个进程间通信中占据过大的比例从而无法明显得到

地址复用方法对进程间通信性能提升的程度。因此在性能测试中，消息的尺寸最大为 4KB，即一个内存页的大小。

在 vfs 进程发生地址空间迁移之前，进程间通信发生在 serverroot 进程和 vfs 进程之间。当 vfs 进程迁移到 serverroot 的地址空间中以 tpig_vfs 进程为实例运行后，进程间通信就发生在 serverroot 和 tpig_vfs 进程之间。在进行性能测试过程中，除了 serverroot 和 vfs/tpig_vfs 进程外，内核服务线程 system_task 也会参与进来。system_task 负责完成基于内存复制的消息传递。

性能计数从 serverroot 发送第一个消息开始，一直到将最后一个消息发送完毕并且系统调用返回后结束。因此记录的时间是整个进程间通信的时间，包括消息传递、进程切换和在进程间通信中的其它执行流程消耗的时间。

如果进程间通信中涉及的数据传递超过一个消息的尺寸，那么数据将会被拆分成多个消息依次传递。因此，既要测试不同大小的消息传递的性能数据，也要测试相同大小的消息被重复传递多次的性能数据。

测试在实际计算机硬件上进行，而不是使用虚拟机软件进行测试。测试数据是在实际计算机硬件处于最高性能工作状态时获取，对于 MLXOS 来说，就是要让处理器运行在最高工作主频。

5.1.2 测试方法

性能数据通过读取 TSC 获得。

分别测试当消息尺寸为 16、32、64、128、256、512、1024 和 4096 字节时，分别传递 1、2、4、8、10、20、30、40、50、60、70、80、90、100、110、120、130、140、150、160、170、180、190、200、210、220、230、240、250、260、270、280、290、300、400、500、600、700、800、900 和 1000 次相同尺寸消息所花费的处理器周期。

上面的消息传递测试，serverroot 和 vfs 进程迁移到相同的可信进程间通信组之前和之后分别运行一次，这样来比较可信进程间通信组对于性能的提升。

在进程调度上下文更新的重要函数 update_context()中，即便进程间通信发生在不同的可信进程间通信组的进程中，函数代码也会有一个性能优化。因此，为了更清晰的表示可信进程间通信组对于提高进程间通信性能的作用，在性能测试中，将不支持可信进程间通信组代码的内核在相应消息尺寸和传递次数中的性能也进行了测试。

测试结果中的时间消耗，包含了整个进程间通信过程中涉及的所有代码 TLB 和数据 TLB 未命中对性能的影响[①]。

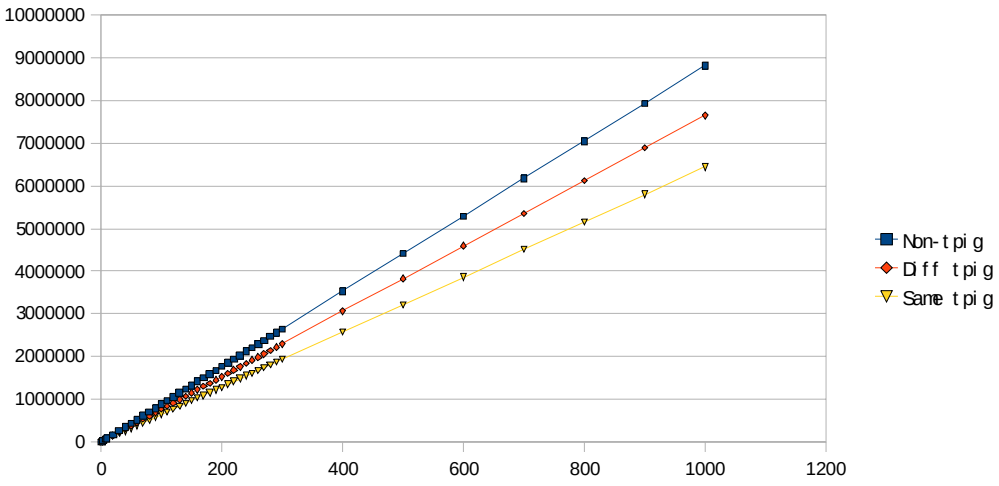
5.2 性能测试数据

具体性能测试结果数据量较大，不在本章节中列出，有兴趣的读者可以查看附录 D。在这里展示测试数据构成的性能图表，从图表上读者可以直观的看到在不同的消息尺寸和传递次数情况中，可信进程间通信组对于进程间通信性能的提高。

后续图中 Non-tpig 所指数据为内核不支持可信进程间通信组时的性能测试结果，在图中以正方形数据点表示；Diff tpig 所指数据是进程间通信的进程不属于相同的可信进程间通信组时的性能测试结果，在图中以菱形数据点表示；Same tpig 所指数据是进程间通信的两个进程属于相同的可信进程间通信组时的性能测试结果，在图中以倒三角形数据点表示。

5.2.1 消息尺寸相同的情况

消息尺寸为 16 字节



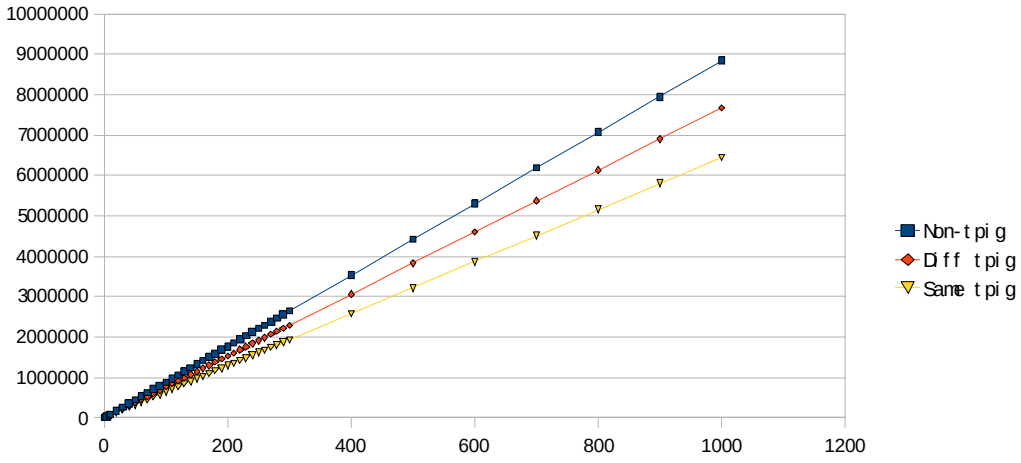
横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-1 消息尺寸为 16 字节时的性能测试结果

[①]由于论文工作时间有限，这里没有编写代码来获取在整个进程间通信过程中更具体的数据 TLB 或代码 TLB 的未命中次数。在将来相关的性能计数器代码将会被加入 MLXOS 中，但在论文进行性能测试时，只能测试所有类型的 TLB 未命中所导致的性能结果。

根据测试数据，当消息尺寸为 16 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 37%，比不同可信进程间通信组的进程间性能提高 19%。

消息尺寸为 32 字节

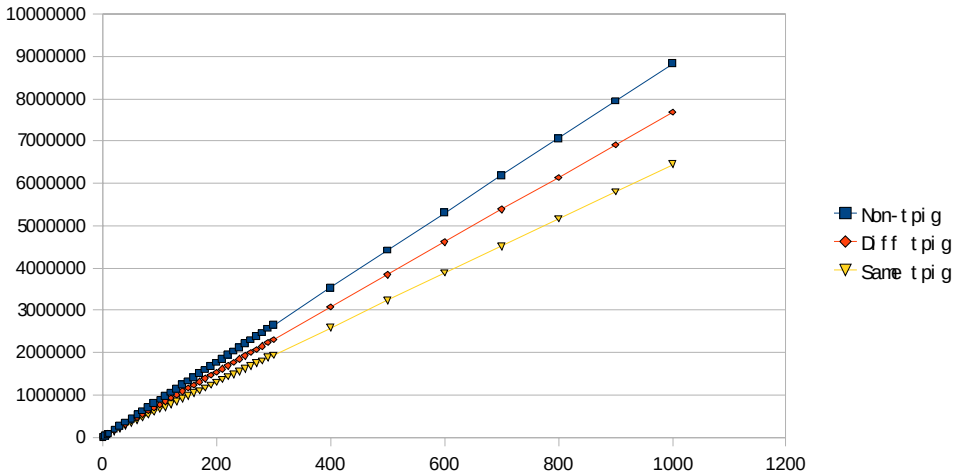


横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-2 消息尺寸为 32 字节时的性能测试结果

根据测试数据，当消息尺寸为 32 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 37%，比不同可信进程间通信组的进程间性能提高 19%。

消息尺寸为 64 字节

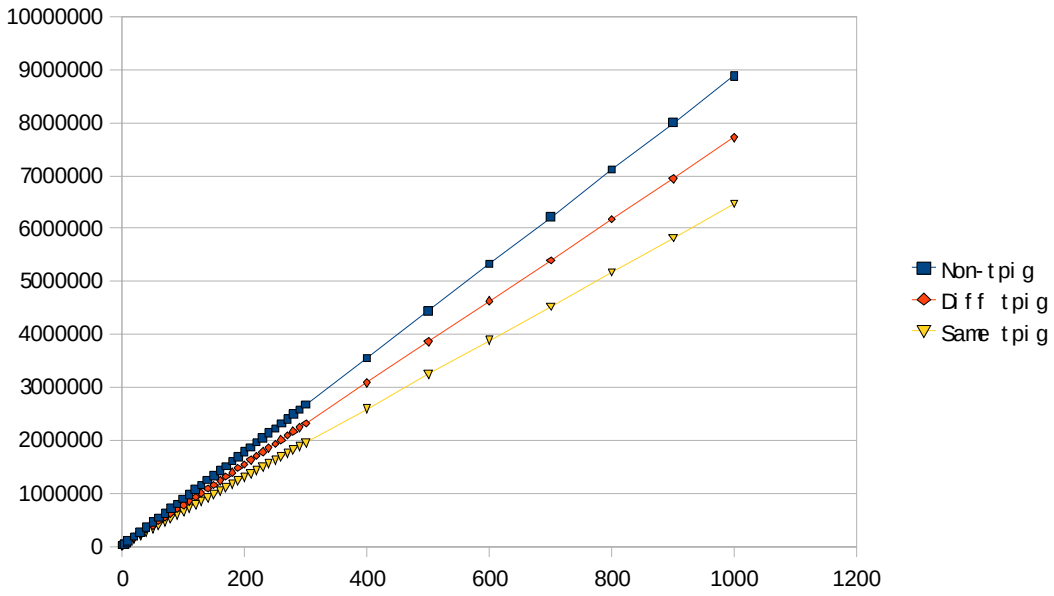


横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-3 消息尺寸为 64 字节时的性能测试结果

根据测试数据，当消息尺寸为 64 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 37%，比不同可信进程间通信组的进程间性能提高 19%。

消息尺寸为 128 字节

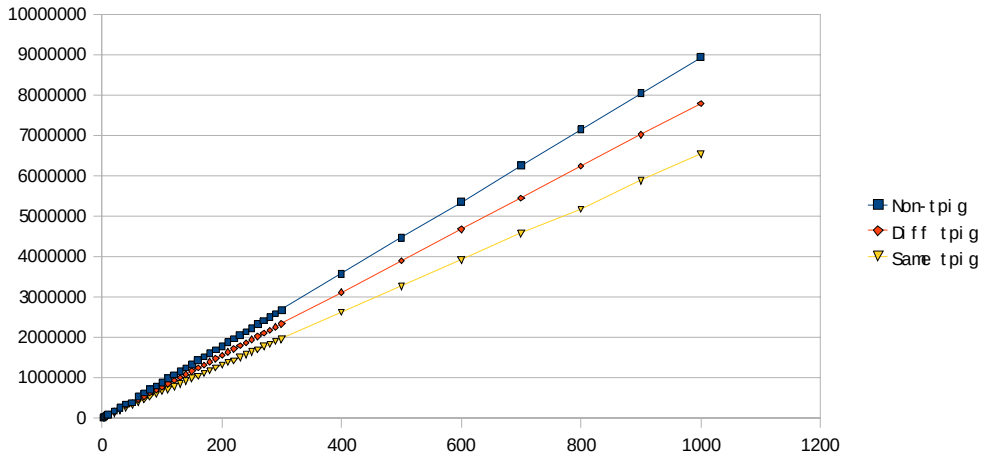


横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-4 消息尺寸为 128 字节时的性能测试结果

根据测试数据，当消息尺寸为 128 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 37%，比不同可信进程间通信组的进程间性能提高 19%。

消息尺寸为 256 字节

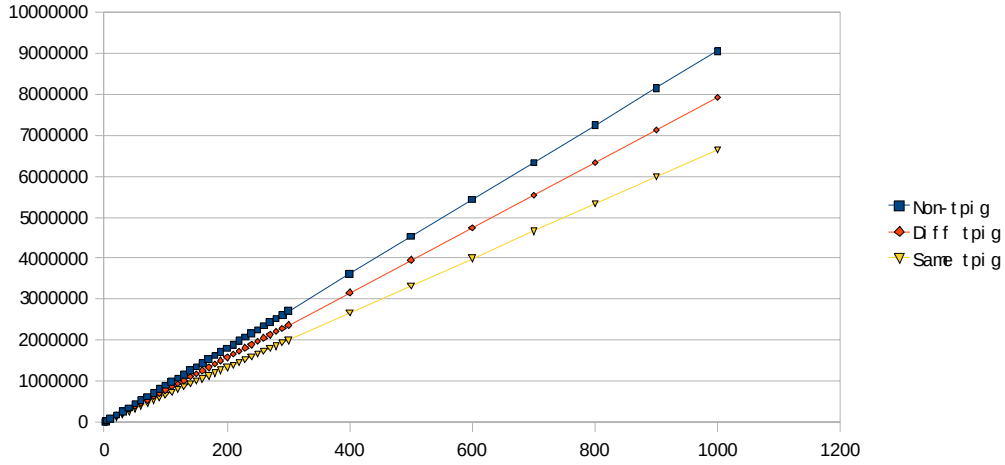


横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-5 消息尺寸为 256 字节时的性能测试结果

根据测试数据，当消息尺寸为 256 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 37%，比不同可信进程间通信组的进程间性能提高 19%。

消息尺寸为 512 字节

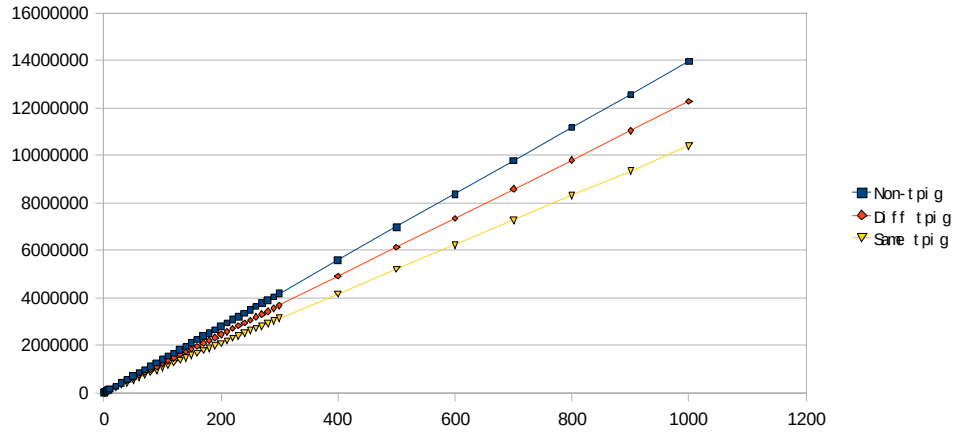


横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-6 消息尺寸为 512 字节时的性能测试结果

根据测试数据，当消息尺寸为 512 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 36%，比不同可信进程间通信组的进程间性能提高 19%。

消息尺寸为 1024 字节

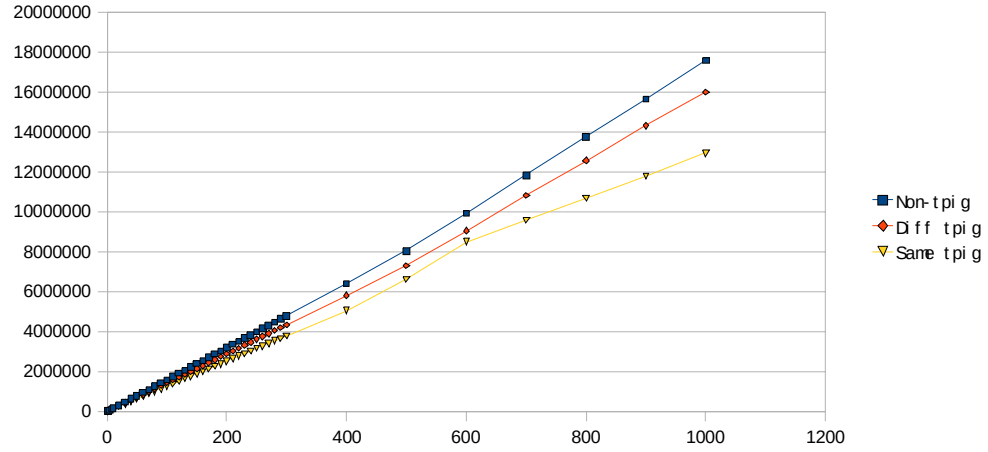


横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-7 消息尺寸为 1024 字节时的性能测试结果

根据测试数据，当消息尺寸为 1024 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 34%，比不同可信进程间通信组的进程间性能提高 18%。

消息尺寸为 4096 字节



横轴：消息传递次数　纵轴：实际消耗处理器周期

图 5-8 消息尺寸为 4096 字节时的性能测试结果

根据测试数据，当消息尺寸为 4096 字节时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 27%，比不同可信进程间通信组的进程间性能提高 15%

5.2.2 消息发送次数相同的情况

发送消息 100 次

表 5-2 发送消息 100 次的性能测试结果

消息大小	Non-tpig	diff tpig	same tpig	对non-tpig提高(%)	对Diff tpig提高(%)
16	885330	767385	646785	36.88	18.65
32	887904	767520	649260	36.76	18.21
64	886779	769986	650583	36.31	18.35
128	890478	772929	649899	37.02	18.93
256	894465	779985	658809	35.77	18.39
512	904491	791190	667251	35.55	18.57
1024	1391688	1223613	1039212	33.92	17.74
4096	1599417	1440810	1263699	26.57	14.02

根据测试数据可以看到，当消息尺寸为 16 字节时，传递次数为 100 时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 36.88%,比不同可信进程间通信组的进程间性能提高 18.65%。当消息尺寸增加到 4096 字节时，相应的性能提高程度则降低到 26.57%和 14.02%。

发送消息 200 次

表 5-3 发送消息 200 次的性能测试

消息大小	Non-tpig	diff tpig	same tpig	对non-tpig提高(%)	对Diff tpig提高(%)
16	1770345	1533141	1279827	38.33	19.79
32	1772991	1533078	1293183	37.1	18.55
64	1770444	1535850	1291599	37.07	18.91
128	1779606	1547946	1296531	37.26	19.39
256	1788084	1559160	1309680	36.53	19.05
512	1809801	1582533	1336779	35.39	18.38
1024	2785455	2445858	2076039	34.17	17.81
4096	3200076	2899224	2524932	26.74	14.82

当消息尺寸为 16 字节时，传递次数为 200 时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 38.33%，比不同可信进程间通信组的进程间性能提高 19.79%。当消息尺寸增加到 4096 字节时，相应的性能提高程度则降低到 26.74%和 14.82%。

发送消息 300 次

表 5-4 发送消息 200 次的性能测试

消息大小	Non-tpig	diff tpig	same tpig	对non-tpig提高(%)	对Diff tpig提高(%)
16	2650158	2296071	1938051	36.74	18.47
32	2653632	2298015	1930266	37.47	19.05
64	2654001	2304009	1938951	36.88	18.83
128	2668905	2318490	1947573	37.04	19.05
256	2682569	2338947	1963917	36.59	19.1
512	2714517	2373624	2002995	35.52	18.5
1024	4179312	3669426	3126798	33.66	17.35
4096	4803687	4355775	3783024	26.98	15.14

当消息尺寸为 16 字节时，传递次数为 300 时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 36.74%，比不同可信进程间通信组的进程间性能提高 18.47%。当消息尺寸增加到 4096 字节时，相应的性能提高程度则降低到 26.98%和 15.14%。

发送消息 400 次

表 5-5 发送消息 200 次的性能测试

消息大小	Non-tpig	diff tpig	same tpig	对non-tpig提高(%)	对Diff tpig提高(%)
16	3535236	3063078	2573199	37.39	19.04
32	3535407	3062511	2579931	37.03	18.71
64	3534390	3074490	2579733	37.01	19.18
128	3557106	3089403	2594430	37.11	19.08
256	3575880	3118239	2620566	36.45	18.99
512	3620070	3164742	2664558	35.86	18.77
1024	5574429	4890006	4145850	34.46	17.95
4096	6405426	5818086	5061924	26.54	14.94

当消息尺寸为 16 字节时，传递次数为 400 时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高

37.39%，比不同可信进程间通信组的进程间性能提高 19.04%。当消息尺寸增加到 4096 字节时，相应的性能提高程度则降低到 26.54%和 14.94%。

发送消息 500 次

表 5-6 发送消息 200 次的性能测试

消息大小	Non-tpig	diff tpig	same tpig	对non-tpig提高(%)	对Diff tpig提高(%)
16	4420233	3829356	3217644	37.37	19.01
32	4420809	3829698	3212262	37.62	19.22
64	4419036	3841137	3230748	36.78	18.89
128	4448628	3864348	3244176	37.13	19.12
256	4471425	3897918	3268710	36.79	19.25
512	4527342	3956625	3329325	35.98	18.84
1024	6967026	6116445	5195682	34.09	17.72
4096	8046657	7309395	6626367	21.43	10.31

当消息尺寸为 16 字节时，传递次数为 500 时，处于相同可信进程间通信组的两个进程传递消息的性能比不支持可信进程间通信组的内核性能提高 37.37%，比不同可信进程间通信组的进程间性能提高 19.01%。当消息尺寸增加到 4096 字节时，相应的性能提高程度则降低到 21.43%和 10.31%。

5.3 分析与评估

最初的性能测试是在 VMware 虚拟机上进行的，测试结果过于乐观，对进程间通信性能的提高达到了 50%左右。为了获取真实的性能数据，在本章节中所列出的数据以及记录在附录 D 中的测试数据全都是直接在计算机硬件上通过 CDROM 启动 MLXOS 系统的实测数据。

当接收方接收到消息后，会在消息所在的每一个页中进行一次读取操作，这样就有可能触发访问消息所在页时的 TLB 缺失。消息所在页对应的 TLB 项未命中，在消息尺寸较小（不超过 4KB）时，不是 TLB 命中率影响进程间通信性能的主要因素。从 TLB 命中率角度影响进程间通信性能的主要部分是进程切换后访问调入进程的代码和数据时，未命中当前进程的指令 TLB 或数据 TLB 导致的性能损失。

下面的数据表是对测试中的消息发送方 serverroot 和接收方 vfs（也代表了 tpig_vfs）的二进制程序在运行时的内存足迹分析[①]，其中页大小为 4KB，可能的 TLB 未命中测试为最恶劣情况下的估计值。

表 5-7 运行时访存足迹

进程	数据页	代码页	动态分配 数据页	运行中访问 代码页数	运行中访问 数据页数	可能的 TLB 未命中
serverroot	1	3	MS*MT/4KB	3	1+ (MS*MT/4KB)	4 + (MS*MT/4KB)
vfs	1	3	512	3	2	5

MS：每次发送消息的大小　MT：总计发送消息个数

由于在测试中发送方 serverroot 最多只发送 4KB 的消息，大于 4KB 的数据需要多次消息传递完成，因此虽然消息接收方的消息接收缓存有 512 个页，但在整个性能测试中只使用了一个页。根据表 8-2 中的分析可以看到，在消息尺寸较小的时候，程序本身代码 TLB 或数据 TLB 的命中率对进程间通信性能的影响要大大超过访问消息页时涉及的 TLB 命中率。

除了磁盘或网络 I/O 外，绝大多数进程间通信的消息数据，都不会超过 4KB。因此在这里测试不超过 4KB 的消息传递，也是具有实际意义的。当涉及到大尺寸数据的传递时，基于内存复制的消息传递机制就不合适了，应当使用基于内存映射的消息传递机制。由于 MLXOS 目前不支持基于内存映射的消息传递，因此不对超过一个内存页的消息传递进行性能测试。

从测试数据中可以发现，随着消息的尺寸从 16 字节增加到 4096 字节，进程间通信性能的降低并不是非常明显。这恰恰验证了当消息尺寸较小时，对消息内容本身的访问并不是影响进程间通信性能的主要因素。

根据测试数据和图表，可以得出如下结论：

- 在系统运行中通过进程地址空间迁移来复用进程地址空间的方法是实际可行的。
- 在微内核架构中，有可能采用可信进程间通信组来提高进程间通信性能。

[①] 通过 readelf 工具可以得到这两个进程的静态内存占用情况，通过 objdump 可以得到这两个进程的静态内存足迹，再结合程序代码就可以得到运行时内存占用和访存足迹。

- 当参与进程间通信的进程属于同一个可信进程间通信组时，在传递小消息时，在 MLXOS 原型系统中进程间通信的性能可以提高 15%~19%。
- 在传递小消息时，可信进程间通信组在提高进程间通信性能方面的主要贡献是提高了进程切换后代码 TLB 和数据 TLB 的命中率，避免了不必要的 TLB 刷新和重新构建 TLB 映射的开销。

第六章　结论与展望

本章对整篇论文进行总结和概括。首先根据前几章内容对本论文主要成果进行简要说明，然后对遗留问题进行讨论并提出今后的下一步工作。

6.1 论文成果

主要的成果是对最初由 Jochen Liedtke 提出的“在奔腾处理器上透明复用用户态地址空间”的想法进行了改进，使地址空间复用的方法可以同时应用在 32 位和 64 位 x86 处理器上并适应了现代的多处理器和多核处理器平台。通过 MLXOS 的实际性能测试可以看到，论文工作提出的改进方法可以进一步提高微内核操作系统的进程间通信性能。

6.1.1 基于进程迁移的地址空间复用

通过基于进程迁移的地址空间复用，可以将一个或者多个进程从各自独立的地址空间迁移到共享区域的小地址空间，或其它进程的地址空间。在 MLXOS 上实现了这个方法，复用相同地址空间的进程之间切换时，就避免了不必要的 TLB 刷新。并且由于该方法是基于页式地址映射方式，不仅在 32 位 x86 平台上可以工作，也支持 64 位的 x86 平台。

6.1.2 可信进程间通信组

在基于进程迁移的地址空间复用之上，为了进一步简化地址空间复用时的进程调度和地址空间切换模型，提出了可信进程间通信组的概念。可信进程间通信组中的进程之间切换时，调度器可以不进行 TLB 刷新，进而提高组内进程间通信的性能。在 MLXOS 中也实现了可信进程间通信组的原型。这种不仅可以同时运行在 32 位和 64 位 x86 处理器上，也对在多核多处理器平台上的进程调度和进程间通信的性能优化进行了考虑。

6.1.3 原型微内核操作系统 MLXOS

为了能够用实际代码来验证论文提出方法的可行性和有效性，论文工作同时构建了一个原型微内核操作系统 MLXOS 来做为参考实现。在针对论文工作的

版本中，MLXOS 共计超过 190 个文件，超过 14000 行 C 代码和 600 行 AT&T 风格汇编代码。MLXOS 不仅仅是论文工作的工程参考实现，今后也可以做为其它微内核技术研究的测试床或原型系统。

6.2 存在不足与展望

时间总是有限的，在论文工作允许的时间内，在 MLXOS 上只完成了对 32 位 x86 单处理器下的原型实现。如何在 64 位 x86 处理器和多核多处理器环境下实现基于进程迁移的地址空间复用和可信进程间通信组，目前的论文工作没有涉及，这些都是未来研究工作需要完善的地方。

6.2.1 实现对 x86 多核和多处理器的支持

下一步首先是要实现在 32 位 x86 处理器下对多核和多处理器的支持，可以使 MLXOS 在多个处理器或处理器核上调度进程。这将是有一个巨大的挑战，也是在 64 位 x86 处理器上支持多处理器或多核处理器的前期准备。

6.2.2 实现对 x86_64 处理器的初步支持

对 64 位 x86 处理器的支持，几乎是对另外一种处理器架构的支持，需要重新设计 MLXOS 的架构和源代码框架。这时就需要考虑代码的硬件抽象层设计了。需要将平台相关代码与平台无关代码隔离开，在编译时通过配置来选定对不同处理器的支持。这将是继支持多处理器后的有一个巨大的挑战。

6.2.3 更多的系统服务

目前 MLXOS 系统中只有 2 个内核服务线程（idle_task，system_task）和 3 个用户进程（serverroot，tty，vfs），除了显示终端信息外无法进行其它更复杂的任务。因此今后还要考虑实现 IDE 驱动、VFS 和只读的 Ext2 文件系统的系统服务，以使得 MLXOS 可以进行简单的 I/O 操作。这样才能在更复杂的工作负载下对论文工作的实效性做进一步的验证。

参 考 文 献

[1] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos, Vrije Universiteit, Amsterdam. Can We Make Operating Systems Reliable and Secure?[J]. in IEEE Computer, May 2006, pp. 44-51.

[2] Andrew S. Tanenbaum and Albert S. Woodhull. Operating Systems: Design and Implementation 3/e[M]. Prentice Hall, 2006.

[3] 王智，李腊元，黄河. 微内核操作系统消息机制分析与评测[J]. 交通与计算机, 2005 年 02 期.

[4] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel 3/e[M]. Oreilly, Nov 2005.

[5] Mel Gorman. Understanding the Linux Virtual Memory Manager[M]. Part of the Bruce Perens' Open Source Series series, Prentice Hall, Apr 29, 2004.

[6] V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of address-space multiplexing on the Pentium[R]. Technical Report 2002-1, University of Karlsruhe, Germany, 2002.

[7] 福岩，尤晋元. 一种提高微内核效率的有效方法[J]. 上海交通大学学报, 2000 年 07 期.

[8] Hermann Hartig, Michael Hohmuth, Jochen Liedtke, Sebastian Schonberg, and Jean Wolter. The Performance of u-Kernel-Based Systems[C]. In Proceedings of the Sixteenth Symposium on Operating System Principles. ACM, Oct 1997.

[9] Jochen Liedtke. u-kernels must and can be small[C]. In Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems, pages 152--161, Seattle, WA, USA, Oct 1996. IEEE.

[10] Jochen Liedtke. Towards real microkernels[J]. Communications of the ACM, 39(9):70--77, Sep 1996.

[11] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces[C]. Technical Report 933, GMD SET-RS, Schlo Birlinghoven, 53754 Sankt Augustin, Germany, Nov 1995.

[12] Jochen. Liedtke. On u-kernel construction[C]. In Proceedings of the 15th ACM Symposium on Operating System Principles. ACM, Dec 1995.

[13] Engler, D., Kaashoek, M. F., and O'Toole, J. Exokernel, an operating system architecture for application-level resource management[C]. In 15th ACM Symposium on Operating System Principles (SOSP), pages 251--266, Copper Mountain Resort, CO, Dec 1995.

[14] Engler, D., Kaashoek, M.F., and O'Toole, J. Exokernel, an operating system architecture for application-level resource management[C]. In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP) (Copper Mountain Resort, Colo., Dec. 1995) ACM Press, 1995, pp. 251–266.

[15] Andrew S. Tanenbaum. A comparison of three microkernels[J]. Journal of Supercomputing,

9(1):7--22, Mar 1995.

[16] Heiser, G., Elphinstone K., Vochteloo, J., Russell S., and Liedtke J. (1998, July). Implementation and Performance of the Mungi Single-Address-Space Operating System[J]. Software: Practice & Experience, 28(9), pp. 901-928. 1994.

[17] M. Condict, D. Bolinger, E. McManus, and D. Mitchell. Microkernel Modularity with Integrated Kernel Performance[R]. Technical report, OSF Research Institute, Cambridge, MA, Apr 1994.

[18] Jochen. Liedtke. A Persistent System in Real Use - Experiences of the First 13 Years[C]. Proceedings of the Third International Workshop on Object-Orientation in Operating Systems, 1993.

[19] Jochen Liedtke. Improving IPC by kernel design[C]. In Proceedings of the 14th ACM Symposium on OS Principles, pages 175--88, Asheville, NC, USA, Dec 1993.

[20] Joseph S. Barrera III. A Fast Mach Network IPC Implementation[C]. USENIX Mach Symposium, pp. 1--18, Nov. 1991.

[21] F.J. Burkowski, G.V. Cormack, and G.D.P. Dueck. Architectural support for synchronous task communication[C]. In 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pages 40--53, 1989.

[22] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, Gyula Szalay. Two years of experience with a u-Kernel based OS[J], ACM Press 1991.

[23] Christopher Browne. Research and Experimental Operating Systems[OL]. http://linuxfinances.info/info/oses.html

[24] Doug Lea. A Memory Allocator[OL]. http://g.oswego.edu/dl/html/malloc.html

[25] Bryan Ford, Erich Stefan Boleyn. Multiboot Specification[OL]. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html

[26] GNU GRUB Manual[OL]. http://www.gnu.org/software/grub/manual/grub.html

[27] QNX's microkernel: IPC via messages[OL]. http://mazsola.iit.uni-miskolc.hu/tempus/parallel/doc/drotos/rtos/microkernel/message.html

[28] Gernot Heiser. Dealing with TLB Tags or I Want to Build a System, What Can L4 Do for Me?[OL]. http://citeseer.ist.psu.edu/heiser01dealing.html

[29] TLBs, Paging-Structure Caches, and Their Invalidation[M/OL]. http://www.intel.com/design/processor/applnots/317080.pdf

[30] Volume 3, System Programming Guide, Intel® 64 and IA-32 Architectures Software Developer's Manuals[M/OL]. http://www.intel.com/products/processor/manuals/

附录 A 32 位 x86 处理器寻址和保护模式

在 x86 平台下，保护模式和内存管理是运行用户态进程的重要基础之一。在 Intel 架构软件开发者手册中，相关章节占据了超过 200 页的篇幅。这里只是将与论文内容密切相关的部分进行基本说明。

A.1 基于段的寻址和保护模式

与实模式下段寻址不同，段属性不再保存在段寄存器中，段寄存器中保存的是段描述符的选择子。通过段描述符选择子可以从全局或进程自己的段描述符表中索引到对应的段描述符。在段描述符中保存着对该段的属性信息，包括段基址，段长度限制，段优先级等等。

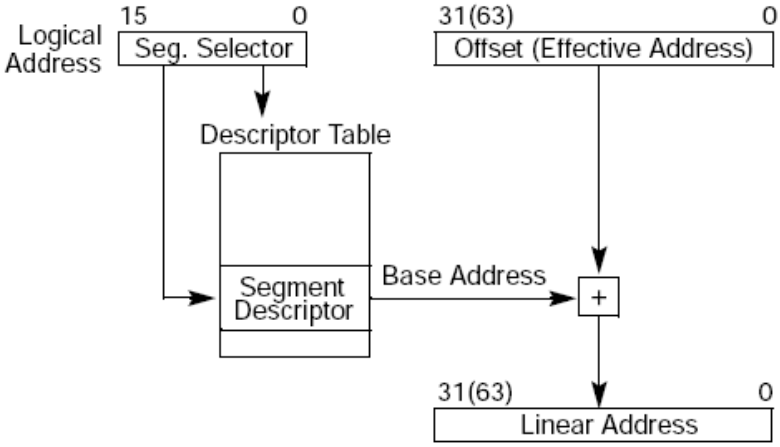


图 A-1 逻辑地址到线性地址的转换[①]

如果不考虑基于页的寻址，那么程序寻址时使用的是逻辑地址，即由<段，段内偏移量>确定的地址。段寄存器中的内容会在程序运行时由者引导代码（如果这个程序是内核）或者操作系统（如果这个代码是应用程序）设置。通过段寄存器中的选择子，可以索引到段描述符表中某一个确定的段描述符，在段描述符中，可以获得目标段的基地址。将该基地址，与逻辑地址中的段内偏移量相加，就可以得到要寻址的线性地址。如果当前寻址模式没有启用页寻址的话，

[①]图 A-1 引用自 Intel® 64 and IA-32 Architectures Software Developer's Manuals 第三卷中图 3-5。

那么得到的线性地址，就是物理地址。整个查表计算过程都在 CPU 的 MMU[①]中进行，无须编写额外代码。

段描述符表的地址和长度由描述符表寄存器确定。段描述符表有两种，一种是全局描述符表 GDT（Global Descriptor Table），一种是局部描述符表 LDT（Local Descriptor Table）。以 GDT 为例，它的地址和长度可以通过将一个 48 位的伪描述符加载到 GDT 寄存器 GDTR 中来设置，该伪描述符结构如下，其中 Base Address 是 GDT 表的 32 位的线性地址，Limit 是该表中表项个数的 8 倍减去 1，单位是字节。

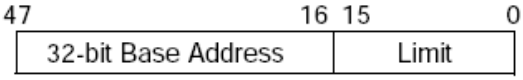
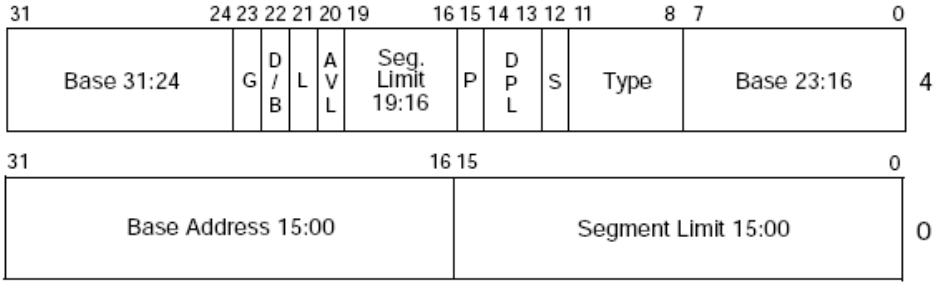


图 A-2 GDTR 伪描述符结构[②]

描述符表是在内存中由一个或多个依次排列的段描述符组成的线性数据结构，每一个段描述符的结构如下：



L　— 64-bit code segment (IA-32e mode only)
AVL　— Available for use by system software
BASE — Segment base address
D/B　— Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL　— Descriptor privilege level
G　— Granularity
LIMIT — Segment Limit
P　— Segment present
S　— Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

图 A-3 段描述符结构[③]

[①]MMU，内存管理单元（Memory Management Unit），处理器中负责地址转换的部件。
[②]图 A-2 引用自 Intel® 64 and IA-32 Architectures Software Developer's Manuals 第三卷中图 3-11。
[③]图 A-3 引用自 Intel® 64 and IA-32 Architectures Software Developer's Manuals 第三卷中图 3-8。

描述符表中每一个段描述符，都可以构建一个特定的段选择子与之对应。通过将指定的段选择子赋值到段寄存器中，就可以对指定的段进行寻址。段选择子的结构如下：

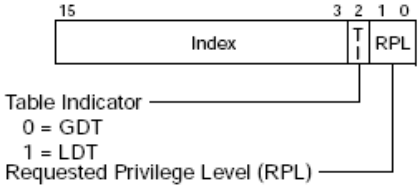


图 A-4 段选择子结构[①]

在 x86 平台下，有 4 个不同的优先级，称为 ring0，ring1，ring2，ring3。其中 ring0 优先级最高，ring3 优先级最低。内核代码一般运行在 ring0 级别，应用程序一般运行在 ring3 级别。选择子中的 RPL，段描述符中的 DPL 都是 2bit，可以表示对应的 4 个优先级。当前段寄存器中的低 2 位代表的权限称为当前优先级 CPL，它和 RPL、DPL 相互配合，完成一系列的权限保护、权限转移过程。

如果在优先级转移过程中，CPL，RPL 和 DPL 不符号规定的优先级规则，那么就会触发相应的通用保护错误。如果访问的段内偏移量超过了当前段寄存器对应的段描述符中规定的长度限制，也会触发相应的通用保护错误。

在 x86 平台下，段的基址是可以在合理的地址空间范围内移动的。但多数现在的操作系统都会使用扁平寻址模式，即将段基址设置为 0，将段长度限制设置为最大值。这样就可以在段的权限保护的基础上，对线性地址采取基于页的保护和寻址。

A.2 基于页的寻址和保护模式

当通过段式映射得到线性地址后，如果还要使用基于页的寻址和保护，那么处理器就会通过查找页表的方法来完成线性地址到物理地址的映射。对于 32 位线性地址，使用 2 级映射，即地址分为如下 3 部分：

[①]图 A-4 引用自 Intel® 64 and IA-32 Architectures Software Developer's Manuals 第三卷中图 3-6。

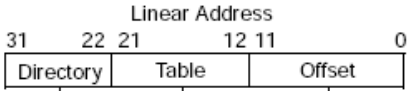


图 A-5 线性地址划分[①]

其中高 10 位用来索引页目录表，中间 10 位用来索引页表项，低 12 位用来在一个物理页内寻址。由于页内偏移量为 12 位，因此通常一个物理页的大小是 4KB。

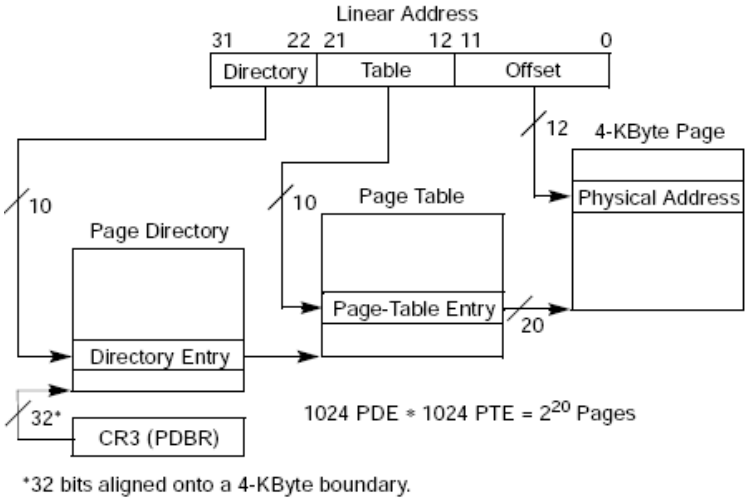


图 A-6 页式映射[②]

页目录表的物理地址存储在 CR3 寄存器中，通过将线性地址在页目录和页表中进行索引，最后由线性地址低 12 位确定页内偏移量，就可以在 4G 的地址空间中完成从线性地址到物理地址的寻址。

通过页式内存管理为应用程序定义一个 4G 的地址空间，以页为单位，提供优先级和读写权限两种保护。页具有管理和用户两种权限，当运行在 ring3 段优先级时，如果用户权限代码访问管理优先级页面，就会触发页错误保护中断；同样当运行在 ring3 段优先级时，如果用户权限代码访问只读页面，也会触发页错误保护中断。基于页优先级和读写权限，就可以实现用户地址空间隔离，按需分页，虚拟内存等功能。

[①]图 A-5 引用自 Intel® 64 and IA-32 Architectures Software Developer's Manuals 第三卷中图 3-12。
[②]图 A-6 引用自 Intel® 64 and IA-32 Architectures Software Developer's Manuals 第三卷中图 3-12。

A.3 虚拟内存和虚拟地址空间

以页为单位更容易实现内存交换、按需加载页面，因此目前多数操作系统上的虚拟内存都是基于页式内存管理来实现的。

所谓虚拟内存，就是将应用程序的虚拟地址空间，和实际占用的物理地址空间分离开来。只将虚拟地址空间中的一小部分和物理地址映射起来，其余部分可以在需要时再加载，或者交换到磁盘中。

当采用虚拟内存管理之后，一个应用程序在启动时不会全部被加载到内存中，而是将虚拟地址空间中的一部分映射到物理内存中。其余部分当需要访问时才加载到内存中，以减少不必要的内存开销，让更多的进程被加载到内存中。

每个进程都有 4GB 的虚拟地址空间，但整个空间并不会都被使用到。绝大多数情况下，在进程的整个生命周期内，只有少部分地址空间会被使用到。而进程在整个生命周期内所使用到的所有虚拟地址空间范围，也不会始终被映射到物理内存中，多数虚拟地址空间在长时间不访问的时候，会被操作系统释放或者交换到磁盘上去。

通过虚拟内存，可以将不同进程地址空间中的区域映射到相同的物理内存之中。这样就可以让两个不同的地址空间共享一段相同的物理内存。在进行进程间通信的两个或多个进程间共享消息内存区，就可以减少因为消息复制产生的额外访存操作。

附录 B　TLB 简介

基于段的寻址和保护，需要访问存储在内存中的段描述符，为了节省访存开销，x86 处理器的段描述符缓冲可以将段寄存器索引的描述符缓冲在寄存器中，大大提高了寻址和保护检查的性能。段描述符缓冲是程序员不可见的。

同样，基于页的寻址和保护，需要访问存储在内存中的页目录和页表。通过将某个页的线性地址和物理地址的映射关系存储在处理器内部的缓冲区中，在寻址时，地址转换和权限检查可以并行进行，大大提高了寻址和保护的性能。该缓冲区称为转换旁视缓冲（Translate Lookaside Buffer），简称 TLB。

在 IA32 处理器下，通常的页式管理是基于 32 位线性地址，和 4KB 大小页框。处理器使用的线性地址是由软件提供的，即在代码生成时确定的。当某一个线性地址经过页表映射到物理页框后，处理器会将这次转换存储在 TLB 中。每一个 TLB 项都是一个有效的转换，可以通过线性地址的页编号索引到物理页框号。TLB 项中包含如下信息：

- 页框地址：用来转换该页号的页表项中保存的物理地址。
- 读/写权限：用于地址转换的所有对应页结构的读写权限位的“逻辑与”结果。
- 管态/目态标识：用于地址转换的所有对应页结构的管态/目态位的“逻辑与”结果。
- 脏位：表明用来进行地址转换的页表项中的脏位是否置 1。
- 禁止执行标识：用户地址转换的所有对应页结构的禁止执行标识的“逻辑与”结果。

TLB 项也可能包含其它信息，而一个处理器中也可能有多个 TLB，譬如指令 TLB 和数据 TLB。而某些用途的 TLB 项也不会使用所有的信息，譬如指令 TLB 中就没有读/写权限标识。

如果进程的某一个页表或页表项被修改了，那么必须刷新整个 TLB 或者将对应的 TLB 项失效。可以通过如下方法使 TLB 失效：

- INVLPG 指令，可以将给定线性地址对应的 TLB 项置为无效。
- 更新 CR3 寄存器，可以将除全局页表之外所有 TLB 中对应项都设为无效。

- 更新 CR4 寄存器中的 PGE 位（bit 7），会将 TLB 中所有项（包括全局页表）均置为无效。

在 x86 处理器上，TLB 内容无法在不同进程之间共享，因此当发生进程切换时，整个 TLB 必须被刷新。

段描述符缓冲和 TLB 是对 x86 平台的寻址和保护检查性能最为密切的两个处理器内部设施。尤其在采用页式内存管理模式下的 TLB 对于进程切换的性能影响显著。

附录 C　编译和运行 MLXOS

C.1 获得源代码

MLXOS 源代码存放与 sourceforge.net 上，可以通过 cvs 工具从服务器端下载和论文工作相对应的源代码版本。以下是在 openSuSE 11.0 操作系统下获取源代码的命令：

```
> cvs -d:pserver:anonymous@mlxos.cvs.sourceforge.net:/cvsroot/mlxos login
> cvs -z3 -d:pserver:anonymous@mlxos.cvs.sourceforge.net:/cvsroot/mlxos co -r
  MLX_0_0_6_TPIG -P mlxos
```

上述命令运行完后，就会在当前目录下出现名为 mlxos 的目录，这就是 MLXOS 的源代码。

C.2 安装 GNU 工具链

在编译 MLXOS 前，要先安装 GNU 的工具链，包括 gcc 编译器，binutils，make 和其它辅助软件包。在 openSuSE11.0 下，选定 developmen 软件包组的缺省配置安装，就可以获得需要的所有软件包。

C.3 编译源代码

MLXOS 的编译非常简单，在 mlxos 目录中，运行“ make ”命令就可以进行整个系统的编译和构建。在编译完成后，运行“ make install ”即可将整个系统所需的文件都安装到光盘镜像文件 cdrom.iso 中。

C.4 在虚拟机或真实硬件上运行

如果在 QEMU 虚拟机上运行 MLXOS 的话，只需要在当前 Linux 系统中安装了 QEMU 软件包，然后在 mlxos 目录下运行“ make test ”命令即可。

如果要在真实硬件下运行 MLXOS 的话，需要 cdrom.iso 刻制到真实的 CDROM 光盘上，并用该光盘引导计算机系统即可。

附录 D　性能测试数据

(均以处理器周期为计时单位)

D.1 消息尺寸为 16 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
16	1	9297	7839	9558
16	2	17865	15885	16218
16	4	35703	31464	29142
16	8	71244	62145	54945
16	10	88794	77562	68202
16	20	177282	154116	132165
16	30	265779	230751	198459
16	40	354573	309483	262746
16	50	442800	384732	326007
16	60	529254	460620	390447
16	70	619785	537381	455346
16	80	708291	615006	518697
16	90	796815	692775	584712
16	100	885330	767385	646785
16	110	973863	844047	711594
16	120	1062396	920484	779211
16	130	1151865	997326	842652
16	140	1239399	1073979	904176
16	150	1327878	1150641	971676
16	160	1416312	1226952	1037664
16	170	1504899	1303902	1098135
16	180	1593495	1380582	1163745
16	190	1682388	1457154	1227213
16	200	1770345	1533141	1279827
16	210	1858869	1609812	1353690
16	220	1947375	1690281	1423512
16	230	2033397	1763154	1486935
16	240	2124414	1840446	1552986
16	250	2212929	1916901	1615797
16	260	2296161	1987866	1681659
16	270	2384703	2065149	1743048
16	280	2473119	2141064	1805445
16	290	2561688	2218536	1872000
16	300	2650158	2296071	1938051
16	400	3535236	3063078	2573199
16	500	4420233	3829356	3217644
16	600	5291325	4593465	3863214
16	700	6176331	5359995	4512708
16	800	7049592	6118929	5155776
16	900	7932366	6884910	5799420
16	1000	8815059	7650108	6443406

D.2 消息尺寸为 32 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
32	1	9090	7884	9486
32	2	17919	15912	16236
32	4	35829	31545	29250
32	8	71316	62199	55098
32	10	89037	77580	68175
32	20	177786	154296	132876
32	30	266544	230931	197550
32	40	355311	307584	262728
32	50	444051	384291	327222
32	60	532818	460962	390951
32	70	621495	537687	455760
32	80	710361	614349	519993
32	90	799731	690822	583254
32	100	887904	767520	649260
32	110	976599	844182	713223
32	120	1065312	920853	778050
32	130	1151685	995571	841860
32	140	1240488	1072233	906417
32	150	1329030	1149084	970155
32	160	1417950	1225647	1034712
32	170	1506771	1304451	1101285
32	180	1595466	1380231	1164654
32	190	1684206	1455912	1231011
32	200	1772991	1533078	1293183
32	210	1861740	1609281	1355481
32	220	1950579	1686393	1421271
32	230	2039283	1763847	1487853
32	240	2128005	1840221	1556172
32	250	2216754	1916757	1612557
32	260	2298555	1991817	1680012
32	270	2387349	2069433	1744119
32	280	2476071	2146230	1803015
32	290	2564910	2222136	1871028
32	300	2653632	2298015	1930266
32	400	3535407	3062511	2579931
32	500	4420809	3829698	3212262
32	600	5305500	4597020	3867876
32	700	6186483	5368311	4510575
32	800	7067052	6131853	5160546
32	900	7948026	6897294	5801544
32	1000	8835957	7663266	6436107

D.3 消息尺寸为 64 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
64	1	9108	7884	9549
64	2	18117	15957	16218
64	4	35757	31518	29196
64	8	71343	62307	55341
64	10	89082	77688	68247
64	20	177822	155322	133488
64	30	266562	231237	198180
64	40	355671	308169	262575
64	50	444033	385335	326610
64	60	532782	461817	391221
64	70	620307	538506	455769
64	80	709047	616896	520659
64	90	797760	692370	584145
64	100	886779	769986	650583
64	110	975510	846063	713736
64	120	1064268	925659	777447
64	130	1149759	999711	844092
64	140	1238445	1076652	908109
64	150	1327221	1153512	972918
64	160	1413810	1230111	1034622
64	170	1504710	1306485	1099539
64	180	1593459	1384020	1167381
64	190	1682271	1460799	1230102
64	200	1770444	1535850	1291599
64	210	1856781	1612476	1357434
64	220	1945224	1689624	1422747
64	230	2033703	1766520	1484757
64	240	2122236	1842417	1554201
64	250	2210751	1919205	1615221
64	260	2298285	1997154	1682370
64	270	2387034	2072988	1739709
64	280	2475819	2151126	1805526
64	290	2564595	2229507	1875834
64	300	2654001	2304009	1938951
64	400	3534390	3074490	2579733
64	500	4419036	3841137	3230748
64	600	5303331	4608306	3875850
64	700	6185826	5379408	4509378
64	800	7067610	6142545	5157576
64	900	7951527	6909669	5801454
64	1000	8834220	7679124	6449409

D.4 消息尺寸为 128 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
128	1	9216	8145	9792
128	2	18126	16218	16416
128	4	36018	31833	29466
128	8	72063	62811	55205
128	10	89775	78174	68454
128	20	179073	155385	133632
128	30	268299	232704	198684
128	40	357318	312534	263376
128	50	446697	387315	328077
128	60	537309	464409	393327
128	70	623844	541863	458559
128	80	712008	619092	521964
128	90	802476	696258	586476
128	100	890478	772929	649899
128	110	979506	850050	715671
128	120	1068534	927270	782262
128	130	1157238	1004580	842534
128	140	1247103	1083132	910818
128	150	1335861	1158525	973278
128	160	1423557	1237140	1038447
128	170	1512954	1314927	1102842
128	180	1602441	1392408	1169892
128	190	1691748	1469133	1238031
128	200	1779606	1547946	1296531
128	210	1868940	1622943	1363032
128	220	1958229	1701180	1426482
128	230	2046060	1777599	1490760
128	240	2134422	1855836	1556190
128	250	2224854	1931913	1621773
128	260	2312802	2009430	1687167
128	270	2402190	2086866	1750635
128	280	2493279	2165229	1816866
128	290	2579418	2240145	1880532
128	300	2668905	2318490	1947573
128	400	3557106	3089403	2594430
128	500	4448628	3864348	3244176
128	600	5339241	4635774	3887874
128	700	6226164	5405850	4532715
128	800	7113888	6175908	5176656
128	900	8003430	6948828	5827347
128	1000	8892783	7722378	6470352

D.5 消息尺寸为 256 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
256	1	9279	8244	9639
256	2	18279	16245	16443
256	4	36162	31968	29745
256	8	72216	63315	55764
256	10	90126	78768	70335
256	20	179802	156618	135108
256	30	269496	234225	199584
256	40	358929	312462	265608
256	50	389952	389826	332550
256	60	537120	467676	395514
256	70	626688	546867	461178
256	80	715716	623889	530730
256	90	805473	702306	590805
256	100	894465	779985	658809
256	110	984267	857700	716967
256	120	1072890	935586	788148
256	130	1162638	1013571	855135
256	140	1252368	1091790	919665
256	150	1343196	1170657	984555
256	160	1430766	1246833	1050570
256	170	1520631	1325493	1113723
256	180	1609758	1403514	1178919
256	190	1699308	1481310	1242513
256	200	1788084	1559160	1309680
256	210	1877733	1637001	1379718
256	220	1967490	1715274	1429344
256	230	2056590	1793295	1508157
256	240	2145717	1870884	1569492
256	250	2235447	1949094	1638198
256	260	2325240	2026665	1695888
256	270	2414322	2104686	1773981
256	280	2503053	2182797	1829700
256	290	2592720	2260953	1898433
256	300	2682569	2338947	1963917
256	400	3575880	3118239	2620566
256	500	4471425	3897918	3268710
256	600	5363289	4677489	3924977
256	700	6256215	5457024	4576563
256	800	7151679	6236082	5176962
256	900	8045469	7016634	5889276
256	1000	8939169	7796187	6534486

D.6 消息尺寸为 512 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
512	1	9729	8892	9765
512	2	18801	16875	16677
512	4	36864	32733	29988
512	8	74277	64575	56997
512	10	91926	80244	70380
512	20	181332	158994	136791
512	30	272916	237636	204624
512	40	361953	316971	271215
512	50	452781	395577	336609
512	60	542313	474561	403695
512	70	633222	553554	473076
512	80	723618	632691	538911
512	90	814203	711810	604305
512	100	904491	791190	667251
512	110	996777	869922	736164
512	120	1086147	950400	802206
512	130	1176075	1028502	868059
512	140	1266354	1107468	938601
512	150	1357146	1186614	1003284
512	160	1447578	1265490	1071801
512	170	1538118	1344915	1133577
512	180	1628775	1423953	1204380
512	190	1719396	1503189	1268640
512	200	1809801	1582533	1336779
512	210	1900386	1661634	1397421
512	220	1990593	1740582	1466064
512	230	2082843	1819782	1528002
512	240	2171754	1898721	1601487
512	250	2262339	1978092	1664397
512	260	2352492	2058651	1729494
512	270	2443248	2137041	1799892
512	280	2533815	2215683	1866627
512	290	2624301	2294487	1936395
512	300	2714517	2373624	2002995
512	400	3620070	3164742	2664558
512	500	4527342	3956625	3329325
512	600	5430132	4747878	3990852
512	700	6336216	5539455	4662126
512	800	7240338	6330528	5327469
512	900	8145297	7121664	5990130
512	1000	9051858	7913394	6647769

D.7 消息尺寸为 1024 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
1024	1	10179	9243	9972
1024	2	19512	17433	17064
1024	4	56736	50409	47016
1024	8	111717	99180	88587
1024	10	139311	123156	109233
1024	20	277299	244125	213489
1024	30	415449	367137	317466
1024	40	554985	490419	422550
1024	50	694062	611748	526374
1024	60	833715	733284	629613
1024	70	972855	856296	735984
1024	80	1112229	977985	835434
1024	90	1251756	1103058	940878
1024	100	1391688	1223613	1039212
1024	110	1530720	1346787	1143423
1024	120	1671165	1467450	1247346
1024	130	1809621	1590147	1353996
1024	140	1949004	1711503	1466604
1024	150	2088063	1834317	1560528
1024	160	2227788	1957104	1668879
1024	170	2367045	2078523	1767492
1024	180	2508777	2200365	1872963
1024	190	2645712	2323287	1969722
1024	200	2785455	2445858	2076039
1024	210	2925837	2568312	2183859
1024	220	3064401	2690856	2285586
1024	230	3203415	2814525	2379132
1024	240	3342897	2934855	2497167
1024	250	3482334	3057561	2598399
1024	260	3621996	3179457	2702520
1024	270	3760956	3302622	2798154
1024	280	3900573	3424059	2907018
1024	290	4039506	3546711	3008169
1024	300	4179312	3669426	3126798
1024	400	5574429	4890006	4145850
1024	500	6967026	6116445	5195682
1024	600	8362701	7337637	6214473
1024	700	9758511	8570025	7260138
1024	800	11151639	9797409	8299602
1024	900	12546423	11029014	9329670
1024	1000	13940361	12259620	10381032

D.8 消息尺寸为 4096 字节

消息大小	传递次数	Non-tpig	Diff tpig	Same tpig
4096	1	19962	17460	19755
4096	2	36531	31743	30447
4096	4	64683	59022	55116
4096	8	127863	117486	106020
4096	10	160155	145062	131445
4096	20	321804	291168	259137
4096	30	482895	434817	385290
4096	40	639864	579627	510876
4096	50	800055	725094	636309
4096	60	960300	865134	764631
4096	70	1120311	1010808	884412
4096	80	1279422	1153818	1016199
4096	90	1442772	1299006	1140633
4096	100	1599417	1440810	1263699
4096	110	1762677	1587276	1392741
4096	120	1921212	1731132	1519551
4096	130	2080143	1876194	1646577
4096	140	2240901	2019654	1769940
4096	150	2401677	2166327	1898253
4096	160	2560329	2309004	2014011
4096	170	2721042	2454030	2145915
4096	180	2882997	2602107	2280564
4096	190	3042189	2749518	2396799
4096	200	3200076	2899224	2524932
4096	210	3361977	3036753	2659095
4096	220	3521502	3184731	2776914
4096	230	3684339	3329388	2902410
4096	240	3838995	3475224	3032955
4096	250	4005612	3621132	3162393
4096	260	4162725	3767211	3284217
4096	270	4322403	3911103	3411648
4096	280	4483431	4063797	3534669
4096	290	4643523	4206852	3647592
4096	300	4803687	4355775	3783024
4096	400	6405426	5818086	5061924
4096	500	8046657	7309395	6626367
4096	600	9935874	9051552	8501931
4096	700	11836512	10808640	9575325
4096	800	13758552	12569364	10691919
4096	900	15650622	14318748	11795643
4096	1000	17577882	15989724	12937635

致 谢

首先感谢我的论文导师杜晓黎和 Markus Rex。杜晓黎研究员目前在联想研究院担任副院长，Markus Rex 目前在 Linux Foundation 担任首席技术官（CTO）。他们在繁忙的日常工作之余，仍然积极推动论文工作的进度，以高度的责任心和专业精神指导我的论文工作。论文工作目前所取得的成果，与两位导师的关心和支持是密不可分的。

在 3 年多前我通过互联网结识了 Zoltan Kovacs。那个时候他 18 岁，但已经拥有了丰富的系统软件开发经验。在接下来的几年里，他鼓励、指导我如何在 x86 处理器上构建一个可以运行的微内核系统，并帮助我解决了很多非常细节的技术难题。他现在是布达佩斯理工学院大学三年级的学生，已被 Nokia Siemens Networks 聘为正式员工，在课余时间仍然从事自己的 giszos 系统开发。正是由于他的支持、鼓励和帮助，我才能在最困难的时候坚持下来完成论文工作中的原型微内核系统 MLXOS。在此我要向他表示忠心的感激之情。

很幸运我的工作单位北京络威尔软件有限公司，允许我使用 10%的工作时间从事和公司业务没有直接关系的创新性研发工作，我才能在紧张工作之余按时完成 MLXOS 微内核原型系统的开发工作。在此也要向建立这个伟大制度的公司管理层，以及对我毕业设计给予热心支持的部门领导和同事表示诚挚的感谢。

在论文完成后，有不少热心的业内前辈和同行在百忙之中帮助我审阅和修改论文。王旭博士（中国移动研究院），钱岭博士（中国移动研究院），胡子明先生（IBM 中国开发中心），涂宏峻女士（IBM 中国开发中心），陈怀临博士（弯曲评论），Eugene Teo 先生（Red Hat Asia-Pacific），特别是吴峰光博士（Intel 开源技术中心），他们以非常严谨的专业态度，一丝不苟的从论文写作、技术观点、文字逻辑等多方面对我的工作提出大量非常宝贵、富有建设性的修改建议。这些前辈和同行们的热心帮助和指导，让我获益菲浅，在学习到很多书本上没有的实践知识的同时，论文质量也得到了明显提高。

最后要感谢我的父母和妻子。在繁忙的工作同时完成硕士学业不是一件容易和轻松的事情，正是他们这些年在生活中对我无微不至的关心和照顾，使我拥有了一段毕生难忘的美妙经历。

个人简历、在学期间发表的论文与研究成果

李勇，1978 年 7 月出生于陕西西安，2001 年毕业于北京邮电大学管理与人文学院管理工程专业，获得工学学士学位。

2001 年 2 月加入联想研究院服务器系统研究室从事与服务器监控管理相关的技术研究与开发工作。2005 年 12 月加入 Novell 公司从事开放源代码软件开发工作。目前就职于 Novell 公司 SuSE 实验室从事 Linux 下文件系统相关的研究、开发和维护工作。

在学期间没有发表论文，相关工程研究成果如下：

- 职务申请国家发明专利 3 项，获得授权 2 项（专利申请号 200410004580，200410004581），均为第一发明人。
- 参与开发和维护多个开放源代码软件项目：OpenIPMI，YaST，e2fsprogs，ocfs2-tools。
- 参与开发和维护 Linux 内核中的 OCFS2、 Ext4 文件系统，目前有 4 个 patch 被 Linux 内核接收。
- 作为 Google Summer of Code 2008 的项目指导人，指导 grub4ext4 项目为 GRUB 引导器添加了 Ext4 文件系统的驱动代码。
- 构建了开放源代码的原型微内核操作系统 MLXOS。