北京理工大學

本科生毕业设计（论文）外文翻译

外文原文题目： Managarm: A Fully Asynchronous Operating System

中文翻译题目： Managarm：完全异步的操作系统

北京理工大学本科生毕业设计（论文）题目

The Subject of Undergraduate Graduation Project (Thesis) of Beijing Institute of Technology

学　　院：计算机学院

专　　业：计算机科学与技术

班　　级：07112005

学生姓名：董若扬

学　　号：1120202944

指导教师：陆慧梅

Managarm：完全异步的操作系统

摘　要

在本文中，我们概述了 Managarm 的系统架构，这是一个基于微内核的开源操作系统。Managarm 的目标是在用户空间运行特权驱动程序和服务器，构建一个通用的操作系统，同时仍然提供与现有的 POSIX 和 Linux 应用程序源级兼容性。为了在驱动现代支持高并发度的硬件时最小化上下文切换，Managarm 专门依赖于一种异步的 IPC 机制，它可以在执行上下文切换之前提交任意数量的独立 IPC 请求。另外，由于现有的 POSIX 和 Linux 应用程序并不总是设计为异步操作，我们提供了对 POSIX 和 Linux API 的用户级模拟。我们的模拟足以运行各种 Linux 应用程序，包括现代桌面环境。

关键词：微内核、异步 I/O、操作系统、进程间通信、POSIX 模拟

目 录

第 1 章 引言

Managarm 是一个面向 x86-64 和 aarch64 平台的自由开源操作系统，从 2014 年开始作为一个社区努力的成果开发。Managarm 基于微内核架构，采用基于能力的设计，使非特权用户空间驱动程序和服务器能够提供大多数常见于主流内核（如 Linux）的功能。

在许多最新的微内核设计中（如 NOVA[1]、seL4[2]或其他 L4 风格的内核），IPC 通常通过从一个执行上下文（即线程）调用到另一个来进行同步执行。虽然这些现有的通信机制已经被高度优化以实现低延迟的 IPC 调用，但对于同步 IPC 来说，有效地处理高并发的输入和事件可能是具有挑战性的。例如，在处理高带宽网络接口和能够达到每秒数十万次 I/O 操作（IOPS）的非易失性存储数据时就会出现这个问题。

相比之下，Managarm 的 IPC 机制完全围绕着异步通信设计。我们的微内核通过共享内存中的并发通知队列与用户空间线程进行交互。在 Managarm 中，线程很少被阻塞，即使在提交了尚未收到响应的 IPC 请求后也是如此。相反，它们保持在用户空间，并继续在并发运行的任务上取得进展。在实践中，这些用户空间任务使用 C++20 协程来实现。只有当任何用户空间协程中没有更多的工作要做时，Managarm 才会阻塞线程。这使我们能够使用最多一个线程每个 CPU 核心来操作大多数应用程序，实现最大的并行性，而无需额外的线程开销。

为了仍然支持在我们的新 IPC 层之上运行的现有应用程序，Managarm 在用户空间使用 POSIX 模拟层。这个模拟层支持许多 Linux API，使 Managarm 能够运行来自 Linux 生态系统的众所周知的应用程序，如基于 Wayland 的桌面环境。

操作系统概述。Managarm 操作系统的组件可以大致分为三类：（i）微内核、（ii）直接运行在微内核之上的服务器，以及（iii）运行在 POSIX 模拟层之上的应用程序（它本身也是一个服务器）。这个架构在图 1 中显示。

应用程序和服务器都在用户模式下运行。它们不能直接访问硬件（例如硬件寄存器、DMA 和中断处理），除非通过内核提供的能力。

在本文的其余部分，我们将详细介绍我们操作系统的各个组件。第 2 节概述了内核与用户空间交互，重点介绍了我们的并发通知队列机制。第 3 节和第 4 节分别

介绍了 Managarm 的 IPC 机制和内存管理设计。第 5 节介绍了我们的 POSIX 模拟层设计。

第 2 章 内核与用户空间交互

Managarm 的用户空间使用系统调用告知内核启动异步操作。这些操作包括 IPC、等待事件（如硬件 IRQ 或定时器）或各种内存管理操作。在启动操作后，内核立即返回到用户空间（即，不等待操作完成）。操作完成后，内核将操作结果写入由用户空间线程消费的共享内存通知队列中。通常，每个用户空间线程都使用自己的通知队列来实现这一目的。由于用户空间不一定以内核发布的顺序消耗异步操作的结果，Managarm 不使用单个环形缓冲区来存储所有结果。相反，使用多个独立的内存区域；用户空间可以使用引用计数或类似的机制来在完全处理后回收这些内存区域。

Managarm 用于通信异步操作结果的并发队列数据结构由两部分组成：（i）一个或多个通过从零开始的连续整数进行索引的块，以及（ii）一个索引队列。

块是存储实际结果数据的内存区域。通过系统调用创建块时确定的块的字节大小。块由内核写入并由用户空间读取。内核通过将新结果追加到先前写入的数据来发布新结果；即，块从偏移零到更高偏移处被填充。Managarm 中异步操作的结果可以是可变长度的；然而，在启动时可以确定结果的最大长度。这允许我们的微内核预先确定通知队列是否有足够的空间来存储异步操作的结果。

一旦一个块已满（即内核需要发布的下一个结果不再完全适合当前块的剩余空间），内核需要确定下一个数据应写入的块。为此，使用索引队列。索引队列由用户空间写入并由内核读取。它存储块索引（即整数）的环形缓冲区；内核按索引队列确定的顺序写入块。用户空间可以通过将完全处理的块重新追加到索引队列中来回收块。

在乐观情况下，我们的并发队列数据结构操作是无锁的。然而，内核端如果写入的块用尽，则仍然需要暂停，而用户空间端如果处理的结果用尽，则需要阻塞。与其他内核（如 Linux 或 Zircon）类似，Managarm 使用 futexes 进行阻塞。无论是内核端还是用户空间端，如果需要暂停或阻塞，它们都会等待 futex；一旦有进展，它们就会分别被对方唤醒。在内核端暂停向特定通知队列传递通知时，不会影响其他通知队列，确保线程未能处理通知不会影响系统的其他组件。

第 3 章 进程间通信（IPC）

为了促进系统各个组件（如 POSIX 服务器、驱动程序等）之间的通信，Managarm 的 IPC 机制基于消息传递方法，允许在两个对等方之间交换任意数据、能力和认证消息。与纯粹的共享内存队列相比，消息传递机制还允许不相信彼此的对等方进行通信。

主要的 IPC 概念是流，它是一个具有两个端点（称为 lane）的双向通信通道。流端点没有固定的角色（没有读端点和写端点）。相反，双方需要针对每个消息个体确定其角色。

为了交换消息，对等方向它们的 lane 提交操作（IPC 操作）。这些操作被排队，并且只有在两个端点都提交了补充操作之后才会被调度（例如，在一个 lane 上发送字节的操作和在另一个 lane 上接收字节的操作）。我们的微内核只排队操作，而不排队它们的相关数据（例如，应该传输的内存缓冲区或能力）。相反，发送方负责延长缓冲区的生命周期，直到它们被复制到接收方的地址空间。这种提交模型允许线程高效地处理任意数量的并发请求，但需要内核内存分配和记账来维护每个 lane 的待处理 IPC 操作队列。

3.1 支持的 IPC 操作概览

操作代表对等方可以执行的操作，例如发送和接收数据、传输能力或对启动线程进行认证。

传输数据。通过其中一方提交可用的“发送”操作（例如，对于在虚拟内存中连续的缓冲区的 SendBuffer 操作，或对于散布收集语义的 SendBufferSg 操作），另一方提交可用的接收操作（例如，RecvBuffer）来完成字节的传输。作为小消息的优化，我们还支持 RecvInline 操作，它将接收的消息直接嵌入到内核与用户空间通知队列中。所有可用的发送操作都可以与所有接收操作匹配。

传输能力和认证。通过发送方提交 PushDescriptor 操作（给出要传输的能力的句柄），接收方提交 PullDescriptor 操作来执行能力的传输。

如上所述，对等方还可以交换认证消息。这是通过发送方提交 ImbueCredentials 操作，接收方提交 ExtractCredentials 操作来完成的。这些操作指示内核将唯一的内

核控制的线程标识符从发送方传输到接收方。由于 lane 能力可以在不同的进程之间自由交换（即，lane 不绑定到线程），这是确定对等方身份的唯一方法。验证此身份有时对于 POSIX 模拟是必要的，例如，因为某些 POSIX 文件描述符的行为取决于访问它们的进程，即使它们引用相同的基础对象。

提供和接受。除了上述的基本操作外，Offer 和 Accept 操作创建了一个附属流，可用于进行进一步的通信。这种功能通常用于请求-响应循环，其中客户端使用 Offer 操作打开，然后发送实际请求，并在附属流上接收响应（类似地，服务器在主流上提交 Accept 操作，然后在附属流上接收请求并发送响应）。这种机制确保多个客户端可以通过单个共享流与服务器通信，而不会相互干扰或混淆服务器。示例通信模式如图 2 所示。

3.2 提交和完成

将操作提交到一个通道会将它们加入队列，如果在流的另一个通道上有匹配的操作，消息交换将异步开始。一个线程可以在一个系统调用中提交多个操作，这有助于避免用户空间线程和内核之间不必要的上下文切换。

作为围绕 Offer 和 Accept 构建的请求-响应模型的优化，在一次性提交多个操作时，随后的操作可以标记为使用新的流，避免等待原始操作完成以获得新的通道来使用。

一旦消息交换完成，两个对等方都会通过通知环缓冲区收到关联操作的完成通知。发布的完成提供了对等方的状态（成功、由于无效参数失败等）、关联信息（接收到的数据大小、接收到的能力句柄等）以及初始提交操作时给定的用户指定值。

在特殊情况下，内核可以在只有一个对等方提交了操作时发布完成。例如，如果其中一个对等方关闭了它的通道，则任何未完成的操作将以错误完成。

此外，如果在两个通道上发布的操作不兼容（例如，在一个通道上发送数据，而在另一个通道上接收能力），数据传输是不可能的（例如，因为目标缓冲区太小），或者消息通过 Dismiss 操作被拒绝（用于简化处理协议违规，例如消息内容格式错误），则消息不会交换，并且两个对等方都会收到错误通知。

3.3 与其他微内核的比较

许多当代微内核（如 seL4[2]、NOVA[1]、Minix[3]）主要利用同步方法进行通信，其中接收消息（通常也包括发送消息）会阻塞线程直到操作完成。特别是这些内核侧重于 IPC 延迟，而 Managarm 则强调 IPC 带宽。

一个具有异步 IPC 支持的操作系统示例是 Fuchsia[4]。在 Fuchsia 中，如果没有消息排队，接收消息会阻塞，它提供了类似“select”的界面，可以等待多个响应，以及类似 Managarm 的通知环缓冲区接口，完成事件被发布到一个称为“端口”的队列中。

另一个值得注意的异步 I/O 示例是 Linux 的 io_uring[5]。io_uring 的提交和完成模型类似于 Managarm，主要区别在于：提交通过环形缓冲区和通知内核的系统调用完成，并且用户可以将 I/O 操作提交到同一个提交队列中的任何文件描述符，相比之下，Managarm 的模型将所有提交的操作与一个通道关联起来。与 Managarm 相比，io_uring 的一个缺点是它不与 Linux 的系统调用接口集成，必须重新实现操作才能在 io_uring 中使用。

第 4 章 内存管理

Managarm 提供了内存管理功能，可以有效地实现和简化 POSIX 接口，例如内存映射文件或共享内存的写时复制（Copy-on-Write，CoW）语义。

内存管理接口围绕两个核心组件展开：地址空间和内存视图。地址空间代表线程对地址空间的视图（但与线程没有直接关联，线程在创建时会被分配一个地址空间），而内存视图则代表可以映射到地址空间中的各种类型的内存（例如匿名内存、硬件内存、CoW 内存等）。

4.1 写时复制 (CoW)

内核提供了对写时复制内存的支持。内核公开了系统调用，允许创建给定内存视图的写时复制视图，并创建一个写时复制内存视图的分支，这会创建一个新的写时复制视图，其区别在于对父写时复制视图所做的更改在子视图中不可见。

在内核中实现写时复制简化了内核内存管理接口，并且与在用户空间中实现相比，具有速度优势，例如，由于写入而导致的页面错误不需要切换到处理内存映射的服务器的上下文。

虽然大多数现有的微内核不实现复杂的内存管理原语（超出物理页面的映射和解除映射），但一些微内核也选择实现写时复制。例如，Fuchsia 实现了允许使用写时复制语义创建内存对象副本的功能。

4.2 页缓存

许多当代操作系统使用透明缓存来处理文件内容，称为页面缓存。页面缓存由页面组成，这些页面在需要时按需分配，并在必要时填充底层文件内容。页面缓存的内存用于处理读取和写入，通过向其写入，并处理内存映射文件，这是通过将组成页面缓存的页面映射到用户地址空间来实现的。

在 Managarm 中，页面缓存使用“托管内存”实现。托管内存是一个内存视图，其页面由内核分配，但内容由用户空间线程管理。内核要求用户空间线程初始化页面，并执行回写（当页面首次分配时将其填充为文件内容，并在页面即将从内存中逐出时将页面写回磁盘）。例如，当线程访问尚未获取的文件的页面缓存部分时，内核会分配页面并向管理给定页面缓存的服务器发送通知，告知应初始化页面。管理

线程完成此请求（例如，通过从存储加载数据），并通知内核已经初始化，并且随后内核恢复由于页面错误而挂起的线程。

其他微内核中也存在类似的解决方案。例如，Fuchsia 操作系统提供了类似于 Managarm 托管内存的“页式存储器”。L4 系列内核通常提供在用户空间安装页面错误处理程序的能力。这可用于模拟页面缓存，但与由内核调节的机制相比，会产生更高的开销。

第 5 章 POSIX 模拟

尽管我们的微内核为用户空间服务器和应用程序提供了构建块，但其接口在 Managarm 操作系统之外并不具备可移植性。同样，由我们的设备驱动程序和服务器暴露的异步 API 也是如此。鉴于许多现有程序使用 POSIX API，Managarm 在一个名为 posix-subsystem 的指定服务器中实现了这个接口。这个服务器充当了一个仿真层，用于在从 POSIX 移植的用户空间程序和 Managarm 系统的其余部分之间进行通信。

虽然实现的大部分与等价的单体内核中的 POSIX 基础设施的实现相似，但由于在用户空间实现 POSIX 仿真功能而产生了一些独特的设计选择。最大的区别在于进程与 POSIX 实现进行通信的方式。对于单体内核，这在很大程度上是通过系统调用完成的，类似于对内核的库调用。在 Managarm 中，这种通信是通过异步 IPC 请求和所谓的超级调用来实现的。本节将简要概述这些方法以及它们的使用情况。

5.1 异步 POSIX 请求

假设一个用户程序想要打开一个文件。在 POSIX 中，通过调用 C 库的 open 函数来完成这个操作。在 Managarm 的 C 库中，这被转换为对 posix-subsystem 的异步请求。以下是执行此请求的代码。此过程中的几个元素是大多数提交给 POSIX 的请求中共有的，并进一步进行了审查。

首先，exchangeMsgsSync 是一个包装函数，它使用 Managarm 内核接口在给定的 IPC 通道上提交 IPC 调用，并且只有在对等方发送其响应或发生错误条件时才返回。特别地，由于 C 库接口没有使用协程暴露，因此需要将 Managarm 的异步调用转换为同步调用。我们注意到也可以以异步方式执行相同的请求；但是，这需要应用程序适当的支持，并且不由 C 库处理。

其次，helix 是 Managarm IPC 接口的 C++20 协程实现。它公开了几个辅助函数，允许请求按程序方式组装。在 open 调用的情况下，Managarm 的 C 库发送一个 offer 操作，然后是 sendBragiHeadTail，最后是 recvInline。每个操作的结果都以元组形式返回，允许用户验证 IPC 事务中的每个单独步骤。

处理打开请求。一旦 posix-subsystem 接收到 OpenRequest，将查询内部虚拟文件系统（VFS）以确定负责处理请求的 Managarm 系统的哪个组件。VFS 可能会调用外

部服务器执行路径遍历。一旦找到相应的文件，就会执行打开调用，其行为取决于提供文件服务的文件系统。在 posix-subsystem 中有几种文件系统类型，主要区别因素是是否是外部的（例如，tmpfs）。如果是外部文件系统，则打开调用将导致发送请求到相应的文件系统服务器；否则，它将在内部处理。在这两种情况下，结果都是相同的，即打开文件。打开的文件包含一个新的 IPC 通道，称为透传通道。

透传通道。打开完成后，我们希望避免将与新打开的文件相关的通信路由到 posix-subsystem 服务器。相反，我们通过让应用程序直接与文件系统服务器通信来避免这种开销。

因此，引入了透传通道的概念。当文件系统服务器打开文件时，它创建了一个 IPC 流。这产生了两个通道，其中一个映射到用户进程的地址空间。重要的是要注意，这个 IPC 流不一定是在 posix-subsystem 和用户进程之间建立的。流也可以在文件系统服务器和用户进程之间启动。当用户进程想要对打开的文件执行某些操作时，它能够直接与负责的文件系统服务器通信，而不是让 POSIX 服务器将请求传递给文件系统服务器。

5.2 超调用

POSIX 信号是 POSIX 用户空间的一个重要组成部分，在微内核架构上实现它们比在单片内核上更为复杂。在信号的情况下，仅使用 IPC 进行通信是不够的，因为调用必须更改调用进程的寄存器映像。这在传递同步信号时尤其重要，例如当发生页面错误时。不能安全地在信号处理程序外恢复进程。像在单片内核中那样使用系统调用来实现信号可以避免这个问题。然而，在 Managarm 内核中实现信号违反了核心微内核设计原则。为了正确实现信号，同时仍然遵循微内核设计原则，Managarm 使用了一个名为 supercalls 的机制。

Supercalls 是没有与内核内部关联操作的系统调用，而是导致一个事件被发送到观察用户空间进程。当一个线程由 POSIX 启动时，它会生成一个连续调用 helix::submitObserve 的协程。这是一个异步系统调用，当内核对线程进行新的观察时完成。在内部，内核将调用线程添加到与观察线程相关联的观察者列表中。一旦内核注册了一个可观察的事件（目前包括故障、终止和 supercalls），观察线程被挂起，观察线程被唤醒并被告知观察到的事件。假设用户线程调用了 POSIX 的 kill 函数，它可能导致其自身的寄存器映像发生变化，因此需要使用 supercall 来实现。在 Managarm

的 C 库中，该实现调用了一个特定的系统调用号，该号码落在内核认可的可观察范围内。这完成了对 helix::submitObserve 的调用，触发 POSIX 对 kill 调用的操作。一旦信号逻辑完成，并且寄存器映像被更改，观察到的线程被恢复。

参考文献

[1] Steinberg U, Kauer B. NOVA: A microhypervisor-based secure virtualization architecture[C]// Proceedings of the 5th European conference on Computer systems. 2010: 209-222.

[2] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: 207-220.

[3] Tanenbaum A S, Woodhull A S, et al. Operating systems: design and implementation: vol. 68[M]. Prentice Hall Englewood Cliffs, 1997.

[4] Pagano F, Verderame L, Merlo A. Understanding Fuchsia Security[J]. arXiv preprint arXiv:2108.04183, 2021.

[5] Axboe J. Efficient IO with io_uring[J]. URL https://kernel. dk/iouring. pdf, 2019.