第 4 章　特权态微虚拟机的组件化操作系统设计与性能优化

4.1　本章引言

除了绕过内核、消除系统调用的上下文切换开销外，专用的操作系统也是微虚拟机提升性能的重要途径。专用化（specialization）通过对功能进行按需实现，以及针对不同的场景选用不同的实现，形成极简的、可定制的操作系统，以获得最佳性能。

然而，现有的微虚拟机操作系统，在专用化的效果与效率上存在不足。它们一般采取以下三种方法：一是在已有通用操作系统的基础上进行裁剪与改造[49,61-62,69-70]，但是因为原系统模块的紧耦合性，专用化程度不够高；二是从头开发新的专用操作系统[19,26,162-163]，但是需要大量的专家知识与开发成本，效率不高；三是采用组件化的设计方法对功能进行组合[22,28,46-47]，但是在组件的易用性与可重用性上仍有待提高。

另一方面，现有微虚拟机操作系统在接口设计上难以同时兼顾性能与兼容性。POSIX 接口为应用程序提供了最大程度的兼容性，但在单地址空间的微虚拟机场景下存在许多冗余的调用路径，无法获得最佳性能。而使用针对性能进行优化的接口，又会导致难以兼容已有应用。

本章认为，是编程语言阻碍了操作系统的专用化发展。现有的通用操作系统或微虚拟机操作系统，大多采用 C 语言编写，导致功能模块紧耦合，难以裁剪、改造、替换与重用。近年来新兴的 Rust 语言，具有诸多高级语言的特性，以及与 C 相媲美的性能，非常适合用于实现专用化的微虚拟机操作系统。然而，目前虽然也有不少用 Rust 语言实现的微虚拟机操作系统[22,164]，但是它们并没有充分发挥 Rust 以及微虚拟机的优势，在性能上反而不如用 C 编写的系统。

为了更好地利用专用化提升微虚拟机的性能，本章提出了 ArceOS 微虚拟机操作系统。ArceOS 利用 Rust 语言的特性，针对微虚拟机的特点，在组件化设计与接口设计两方面进行了优化。在组件化设计中，ArceOS 通过将组件划分为操作系统相关与操作系统无关两类，来减少组件对操作系统设计理念上的依赖，以降低耦合性，提升可重用性（第 4.3.2 节），同时也借助 Rust 语言提供的机制实现了操作系统功能的灵活定制（第 4.3.3 节）。ArceOS 还利用组件化设计带来的灵活性，不仅可以作为一个 Unikernel 运行于特权态微虚拟机中，还能形成宏内核、虚拟机管理程序等多种内核形态（第 4.3.5 节）。

在接口设计中，ArceOS 借用了 Rust 标准库的接口，提供了快速路径，避免了

微虚拟机场景下使用 POSIX 接口带来的冗余路径，提升了性能。同时还能直接支持已有的 Rust 应用，提供了足够的兼容性（第 4.3.4 节）。

目前，ArceOS 提供了 30 多个可重用组件与 10 多个松耦合模块，实现了多核、多线程、网络、文件系统、图形显示等基本内核功能，支持 x86_64、RISC-V、AArch64 等主流处理器架构，QEMU/KVM、x86 服务器、树莓派、黑芝麻等多种硬件平台，以及 Virtio、Intel 82599 万兆网卡等设备驱动（均可根据应用需求进行灵活定制），总代码量达 3 万多行。实验结果表明，ArceOS 的组件化设计方便了操作系统功能的定制，ArceOS 的 API 快速路径设计比使用 POSIX 的接口快 50% 以上（文件读写）。ArceOS 同时兼容已有 C 应用程序，与 Linux 相比，使 Redis 的延迟降低了 33%。

本章的主要贡献如下：

1. 提出了一种基于设计理念相关性的组件化操作系统设计方法，能够降低组件间的耦合性，并提供良好的可定制性与可重用性。
2. 提出了一种基于 Rust 语言特性的 API 快速路径设计，专门针对微虚拟机场景进行了性能优化，同时也不失兼容性。
3. 实现了一系列可重用的组件，能够快速搭建出满足应用需求的多形态操作系统，支持已有的 C 或 Rust 典型应用。
4. 通过实验展示了组件化以及 API 快速路径的优化效果，并与 Linux 以及已有的类似工作进行了细致的对比。

4.2　研究动机

专用化是提升操作系统性能的重要途径。本章先介绍利用微虚拟机实现专用化的基本方法，以及面临的挑战，然后讨论了如何利用组件化的设计方法以及新型编程语言来应对这些挑战，最后给出了本章系统的设计目标。

4.2.1　操作系统专用化

微虚拟机只运行一个应用程序，而且可以在应用中实现部分操作系统的功能，非常适合根据应用与场景的特点，对操作系统的功能进行深度定制，从而获得最大的性能。这种专用化主要体现在两方面：

（1）极简性，即只实现必须要实现的功能，去除任何不需要的功能。这不仅可以减小镜像大小、提升启动速度、减小攻击面，还能避免过多地考虑通用性，简化实现方式，带来更多优化机会。例如，对于只运行单个应用的微虚拟机，应用与内核的特权级隔离、地址空间隔离都是多余的；对于事件循环驱动的单线程服务器

应用，抢占与调度是多余的；对于基于 UDP 的低延迟网络应用，完整的 TCP 协议栈是多余的，而且还可以通过消除 VFS 层的多态性来缩短调用路径[70]。

（2）可定制性，即提供同种功能的多种实现。即使是同一功能，在不同的场景下，使用不同的算法或实现方式，也会有不同的性能表现，这使得应用程序能够根据需要选择最佳的实现。例如，本文第 3 章重点关注的任务调度，在不同的任务负载下有各自适合的调度策略；不同的内存分配算法也对应用的性能有很大影响[28]；对于设备驱动，也可以根据应用的需要进行配置，在性能更高的轮询模式与 CPU 利用率更高的中断模式之间进行取舍。

4.2.2　实现专用化的挑战

专用化的终极目标是能够以很小的代价为每个应用快速生成出一个专用的操作系统。然而，即便是在微虚拟机这样的轻量场景下，实现以上目标也绝非易事。本节分析了其中的一些关键挑战。

操作系统模块紧耦合。为了实现专用化，操作系统的功能模块需要能被方便地移除或替换。然而，在以宏内核为代表的传统操作系统设计中，各个功能模块间存在错综复杂的函数调用与数据访问关系[28]。如果要删除其中一个模块，或替换为另一种实现，必须仔细考虑这些依赖关系，确保其他模块不受影响。因此难以对它们进行专用化改造。

现有的一些微虚拟机操作系统基于对已有通用操作系统的改造[49,61,69-70]，但因为原系统模块的紧耦合性，专用化程度不够高，例如只能做到去除特权级分离与多地址空间，其余大部分功能仍保留原系统的设计。

操作系统模块难以重用。另一条实现专用化的途径是从头开发新的操作系统。但是操作系统的开发门槛较高，不仅需要掌握相关软硬件技术，还要考虑诸多实现细节，这对于只为了运行某个应用的普通用户来说难以接受。另一方面，虽然开发操作系统的工作量大，但是过程具有重复性。例如，几乎所有的操作系统都要实现页表操作、内存分配等基本功能，或是链表、平衡二叉树等基础数据结构。在开发新操作系统时理应可以重用其他系统中的相关代码，以减少开发代价。

然而，由于传统操作系统的自包含性，其功能模块只考虑了为自己服务，很难被其他软件重用。再加上传统操作系统模块的紧耦合性，如果要将一个模块拆出给其他软件使用，就不得不处理模块间复杂的依赖关系。

已有一些微虚拟机操作系统，重用了成熟操作系统的部分代码[27,62,68]，但是这种重用的粒度较大，如完整的网络协议栈或驱动程序，使得其中仍保留大量原系统的影子，专用化程度不够高。还有一些工作采用组件化的设计方法，通过对内核组件的组合来形成满足需求的专用操作系统[22,28,46-47]，从而减少开发代价，但

是这些组件与原系统的设计较为耦合，难以被广泛重用。

接口设计难以兼顾性能与兼容性。另一个阻碍专用化效果的因素是接口的设计。现有的许多微虚拟机操作系统[27-28,48-49]，为了能够兼容已有的应用程序，选择使用 POSIX[165] 这样的原通用操作系统中的接口。然而，这些接口原本是为宏内核而设计的，考虑了诸多宏内核下的特点，如地址空间隔离，在微虚拟机场景下存在冗余的调用路径，从而限制了从应用到内核端到端的功能专用化。但如果使用面向性能的专用接口[19,22,26,32]，又会面临应用兼容性的问题。

4.2.3　编程语言的考虑

为解决以上挑战，采用组件化的操作系统设计方法仍是一个值得深入研究的方向。为了更好地实现一个组件化的操作系统，需要重新思考操作系统的架构设计，乃至使用的编程语言。先前的工作难以实现专用化，或存在一些不足，在很大程度上可以通过使用更现代的编程语言代替原来的 C 语言而解决。近年来，Rust[96] 语言越来越流行，而且在编写操作系统等底层软件上很有吸引力[21-23,97-98,164]。本文认为 Rust 语言非常适合用于实现微虚拟机场景下的专用操作系统。本节先分析 C 语言在操作系统专用化方面的不足，然后介绍 Rust 语言在该方面的优势。

C 语言的不足。C 语言具有简洁、高效、灵活等特点，被认为是编写操作系统的首选语言。然而，也正是因为 C 语言的高度灵活性，导致这些操作系统中存在许多混乱的、不利于维护的代码，例如类型强制转换、随意访问其他模块的函数或数据等。正是这种过度的灵活性导致了软件模块的紧耦合，特别是对于操作系统这样的大型软件来说。此外，C 语言也没有提供一些现代高级语言的特性，如面向对象、模块封装、泛型、包管理器等，使得用 C 语言编写的功能模块难以被其他软件重用。C 语言的简单数据类型也限制了模块间接口的设计。更不必说 C 语言在类型、内存、并发等方面的不安全性，容易导致缓冲区溢出、use-after-free、数据竞争等种种安全漏洞[166]。

Rust 语言的优势。 Rust 语言最具特色的机制是基于所有权的内存安全管理，使得无需垃圾回收等运行时开销，就能实现自动内存管理与安全检查，从而提供与 C/C++ 相当的性能。此外，Rust 还具有许多高级语言的特性，如类型安全、泛型与接口、函数式、枚举模式匹配等等，可以带来极大的开发便利性。同时，Rust 对底层代码的控制能力，使得它非常适合用于编写操作系统等底层软件。下面主要介绍 Rust 有助于实现专用的微虚拟机操作系统的几大特性。

（1）良好的包管理机制。Rust 通过包（crate）的形式组织代码模块，每个包都有一个配置文件，指定了其所依赖的其他包。开发者还可以从官方的包发布平台

crates.io① 中寻找与下载自己所需的包，或将自己开发的包发布出来供其他软件使用。目前 crates.io 中已有超过十万个包[167]。Rust 的包管理机制为组件化的操作系统设计提供了便利，可以将操作系统的功能模块以包的形式组织，从而显式指定它们的依赖关系，避免随意调用，降低耦合性。此外，包发布平台也方便组件的重用（第 4.3.2 节）。

（2）灵活的条件编译。Rust 可在配置文件中，为每个包指定一组特性（feature）。特性可在编译时选择是否启用，从而决定是否编译某段代码。特性还能用于配置包的依赖关系，或者通过包的依赖关系进行传递。对于专用操作系统，这种灵活的条件编译机制，既能方便根据需求选择性启用某些功能，又能方便在同种功能的多种实现间进行切换（第 4.3.3 节）。

（3）基于高级类型的库函数接口。Rust 在 `std` 标准库中提供了对操作系统级功能的访问，例如线程、文件、网络等。与 C 的文件描述符不同，Rust 的标准库接口使用专门的数据类型对这些系统资源进行封装，并通过资源获取即初始化（RAII）机制，由编译器保证这些资源在其生命周期结束时被释放。这种接口既避免了文件描述符转换的开销，又避免了资源泄漏或多次释放。然而，在 Linux 等通用操作系统上，Rust 仍会依赖 libc 库（使用 POSIX 接口）实现对系统级资源的访问，反而增加了开销。但在微虚拟机场景下，可以完全利用这套接口代替传统的 POSIX 接口，既能提供更好的性能与安全性，又能直接支持已有 Rust 应用（第 4.3.4 节）。

4.2.4　本章设计目标

本章的总体目标是能够快速定制出满足应用需求的高性能高安全微虚拟机操作系统。为此，本章认为，使用组件化的设计方法是实现这一目标的关键。为了解决已有组件化设计存在的问题，本章希望能够基于 Rust 语言的优势，结合微虚拟机场景的特点，重新思考组件化操作系统的设计方法。本章具体的目标包含以下三方面：

1. **组件可重用**：组件不应该与特定内核的设计和实现绑定，需要能够方便地被其他应用或内核重用。
2. **组件灵活定制**：应用程序需要能够对组件的功能与组合方式进行方便地配置，以实现专用化所需的极简性与可定制性。
3. **接口专用化**：应用程序调用内核的接口也需要专用化以提供最佳性能，但同时还要保证足够的兼容性。

① https://crates.io

4.3　系统设计

本节先介绍 ArceOS 微虚拟机操作系统的整体架构，然后介绍 ArceOS 的设计细节，包括基于操作系统设计理念相关性的可重用组件化设计、灵活的组件功能定制方法、基于 Rust 语言特性的 API 快速路径优化，最后演示了利用组件化设计实现的另一特色功能：构建多种形态的内核。

4.3.1　整体架构

ArceOS 以特权态微虚拟机作为基本形态，即 Unikernel，仅支持一个应用程序，应用与内核在构建时被链接在一起生成单个镜像，共同运行在客户机特权态，可以直接管理特权硬件资源，而依赖底层的微虚拟机管理程序进行安全隔离。通过不同的配置选项，ArceOS 可以构建为其他形态的内核，如宏内核或虚拟机管理程序，直接运行在裸机上（见第 4.3.5 节）。不失一般性地，ArceOS 的设计思想与优化技术也可用于用户态微虚拟机（如第 3 章介绍的 Skyloft 库操作系统）。

图 4.1 描述了 ArceOS 的系统架构。根据依赖关系，自底向上，ArceOS 可划分为以下几个层次：

- **元件层**：与操作系统的设计无关的部分，可方便被其他系统软件复用。
- **模块层**：与操作系统的设计相关的部分，它们与 ArceOS 的设计较为耦合，不太容易被其他系统软件复用。元件层与模块层集中了 ArceOS 的大部分组件，可认为是传统意义上的“内核”，用来管理硬件并为应用提供服务。
- **API 层**：将模块层的功能封装为 API 的形式，以供应用程序调用。ArceOS 为 Rust 和 C 语言编写的应用程序分别提供了不同的 API，并针对语言的特点做了快速路径优化。
- **用户库层**：通过对 API 层的进一步封装，提供对已有用户库的兼容（如 Rust 标准库、libc 库），以便应用程序的移植。
- **应用程序层**：目前 ArceOS 提供了对 Rust 和 C 这两种语言编写的应用程序的直接支持。

4.3.2　可重用组件化设计

传统的操作系统，虽然大多也采用“模块化”的思路进行功能单元的组织。但这种模块的划分粒度较大，并没有清晰的边界，使得模块间可以随意调用，耦合程度严重。而且这些模块也难以被其他软件重用。例如，为 Linux 实现的设备驱动大量依赖 Linux 的内核 API，如果要将其移植到其他系统，需要在其他系统中也支持这些 API，这几乎覆盖了 Linux 的所有子系统。

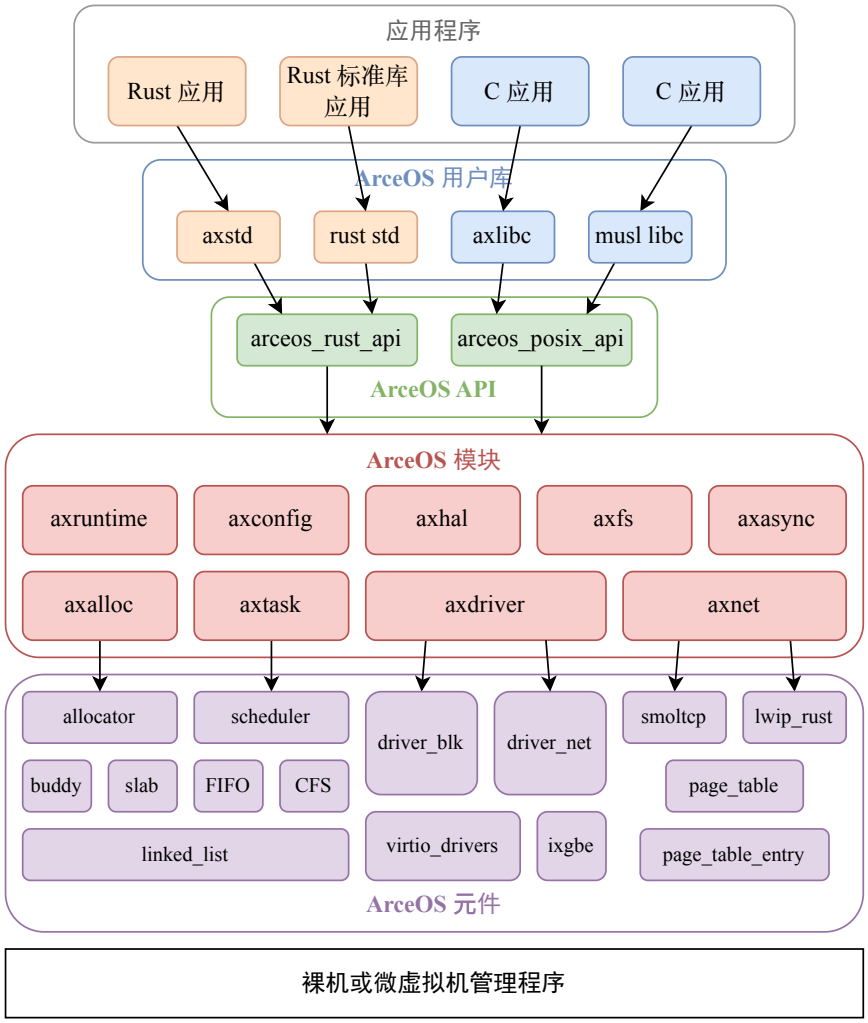


图 4.1　ArceOS 系统架构

为了让操作系统的功能单元具有更广泛的适用范围，需要从模块的划分方式入手重新设计。ArceOS 的基本设计思路是按照更小的粒度，以及是否与操作系统的设计理念相关这两个依据进行功能划分。本文将这些粒度较小的、为了可重用性设计的功能单元称为组件。

操作系统无关与操作系统相关。ArceOS 为提升组件可重用性的重要方法是根据与操作系统的设计理念的相关性进行组件划分。操作系统虽然种类繁多，实现各不相同，但是有的功能单元是几乎所有的操作系统都需要实现的，而且无论操作系统具有什么形态与设计理念，都不会对这些功能单元做太多调整，本文称它们为操作系统无关的组件。这些组件位于依赖关系的底层，对其他组件的依赖较少，但可能会有大量其他组件依赖它。例如，像链表这样的基础数据结构，或各种调度算法的实现。除此之外，本文称剩余的功能单元都是操作系统相关的组件，它们为某个特定的操作系统而打造，并与其设计理念绑定，当用于其他操作系统时可能需要进行重大修改，因此可重用性相对较差。这些组件位于依赖关系的较

高层，一般是对其他组件的组合，以实现更复杂的功能。例如，实时操作系统有自己特有的任务管理方式，以满足高实时性要求，但不一定适用于其他操作系统。表 4.1 总结了这两类组件的区别，并例举了一些组件示例。

表 4.1　ArceOS 组件类别

类别	操作系统相关/无关	依赖关系层次	可重用性	举例
元件	无关	底层	好	基础数据结构与算法 硬件相关操作 具体设备驱动 具体文件系统 网络协议栈
模块	相关	高层	差	系统配置与初始化 异常与中断处理 任务管理与调度 虚拟文件系统抽象 设备驱动层抽象

在 ArceOS 中，把操作系统无关的组件称为*元件*，把与操作系统相关的组件称为*模块*，元件层与模块层共同构成传统意义上的内核，提供操作系统的基本功能。ArceOS 利用 Rust 语言的包管理机制来实现组件化，组件都以 Rust crate 的形式提供，具有清晰的边界与依赖关系。此外，Rust 语言的单一所有权机制，以及对可变全局变量的使用限制（除非使用 `unsafe` 代码），使得状态都是组件私有的，不会在组件间共享，容易实现组件的拆分与替换。通过 Rust 的包管理平台 crates.io，可以方便实现组件的重用与被重用。

元件层在设计上需要更多地考虑可重用性，仔细设计对外的接口，减少对其他组件的依赖，并避免与 ArceOS 的设计产生耦合。ArceOS 的最终目标是让整个操作系统都是由可重用的元件构成的，然而，由于元件设计上的复杂性，一些功能暂时无法与操作系统的设计理念解耦，就需要暂时以模块的形式存在。因此，一种循序渐进的开发方式是，逐步将操作系统的功能从模块拆解到元件中，减少模块的数量，增加元件的数量。

有序的组件依赖。影响组件可重用性的另一因素是组件间的依赖关系。这种依赖关系包括在另一组件中实现的函数、定义的数据结构。由于在重用一个组件时，需要同时包含其依赖的所有组件，混乱的依赖关系容易导致引入过多无关紧

要的组件，不仅违背了专用化的极简性原则，也导致了“牵一发而动全身”的情况。例如 C 语言没有提供语言层面的组件概念，可以通过外部函数声明的形式调用任意函数，造成了无序的跨组件调用，难以重用。为此，ArceOS 规定组件间需形成有序的依赖关系，即遵循以下原则：

1. *明确定义依赖关系*。一个组件可调用的组件列表，以及可使用的功能（函数或数据结构）需要被明确给出，不能随意调用不在列表中的功能。
2. *无反向依赖*：下层组件不能反向调用上层组件（如元件层不能调用模块层）。
3. *无循环依赖*：组件不能相互依赖，整体的依赖关系需要形成有向无环图。

在 ArceOS 中，组件的依赖关系通过 Rust crate 的配置文件（Cargo.toml）指定，只能调用在依赖列表中的组件的功能（或 Rust 核心库 `core`），可以避免随意进行跨组件调用。ArceOS 通过规定元件只能调用元件，模块只能调用元件或模块来避免反向依赖，防止重用下层组件时需要同时包含上层组件。此外，Rust 的包管理机制也明确禁止了循环依赖，使得依赖关系清晰。

然而，由于操作系统固有的复杂性，有些时候仍然难以避免一些反向或循环的组件依赖。例如，自旋锁的实现与操作系统的设计关联不大，应该被放到元件层。一个实现正确的自旋锁需要在进入临界区前关中断或关抢占，并在出临界区时重新打开，以防当前线程在临界区被调度走而影响正确性。但是，“关抢占”这一操作无法在元件层单独实现，因为需要依赖模块层中的线程管理功能，而线程管理模块与操作系统的设计理念较为相关，不适合移入元件层中。

为此，ArceOS 专门提供了一个元件 crate_glue，用于在不得已时提供*接口定义明确*的反向或循环调用。该元件将依赖双方分为功能的调用者与实现者，位于下层的调用者会声明一套所需功能的接口，位于上层的实现者会对其声明的接口进行具体实现。两者只需依赖 crate_glue 元件，而无需直接形成反向或循环的依赖。这种方式，虽然在本质上还是存在组件的循环依赖或反向依赖，但对这些调用关系进行了特殊标记，避免了开发者的滥用与不经意的使用，有利于日后的分析与检查。表 4.2 列出了目前 ArceOS 使用 crate_glue 解决反向或循环依赖的情况。

4.3.3　组件灵活定制

专用化要求能根据应用的需求构建出最简的、可定制的操作系统。ArceOS 利用 Rust 的“特性”（feature）机制，提供组件的可选择性与可替换性，以实现专用化。

基于“特性”的组件定制。图 4.2 展示了通过特性对组件定制的一个例子。在 ArceOS 的 allocator 组件（内存分配）的配置文件中，可以指定三种特性，分别对应不同的内存分配算法（TLSF[168]、slab[169]、伙伴系统[170]）。内存分配算法的具体

表 4.2　ArceOS 组件间不可避免的反向或循环依赖关系

功能调用者	功能实现者	调用关系	功能描述
axlog（日志输出）	axruntime（运行时）	模块 → 模块	输出带时间的日志
axhal（硬件抽象层）	axruntime（运行时）	模块 → 模块	中断与异常处理例程
axhal（硬件抽象层）	应用程序*	模块 → 应用	宏内核形态的系统调用
axfs（文件系统）	应用程序*	模块 → 应用	应用自定义的文件系统
自旋锁	axtask（任务管理）	元件 → 模块	线程抢占的关闭与开启
互斥锁	axtask（任务管理）	元件 → 模块	线程的睡眠与唤醒

* 指运行于内核态的 ArceOS 应用程序层，而非传统的运行于用户态的应用程序（见第 4.3.5 节）。

实现在 allocator 的依赖组件中，并可通过启用不同的特性来进行选择。在 allocator 的代码实现中，也会根据特性来条件编译，分别使用相应的依赖组件实现不同内存分配算法间的切换。这些依赖组件都是可选的，只要不启用相应的特性就不会对它们进行编译，从而不被包含进最终的镜像中。此外，特性还可以通过组件的依赖关系进行传递，如在图 4.2 的第 5 行可以通过 allocator 组件的特性来启用其依赖的 rlsf 组件的特性。

```
1 [package]
2 name = "allocator"
3
4 [features]
5 alloc_tlsf = ["rlsf/foo"]
6 alloc_slab = ["slab_allocator"]
7 alloc_buddy = ["buddy_system_allocator"]
8
9 [dependencies]
10 rlsf = { version = "0.2", optional = true, features = ["foo"] }
11 slab_allocator = { path = "../slab_allocator", optional = true }
12 buddy_system_allocator = { version = "0.9", optional = true }
```

图 4.2　ArceOS 基于 “特性” 的组件定制

Rust 的编译工具链会根据所启用的特性，选择相应的组件（同时进行条件编译）进行组合，来形成最终的操作系统镜像。为了提供最佳性能，组件的选取与组合在编译期间静态完成，无运行时的开销。Rust 编译器会对软件栈上的所有组件进行编译，包括应用程序、用户库、被选择的模块与元件，生成一系列静态库（rlib 格式）。最后，由链接器将这些组件静态链接，生成最终的统一镜像，并支持链接时优化（LTO）。

基于组件灵活定制的专用操作系统演示。下面以最简单的 helloworld 应用程序为例，演示 ArceOS 是如何使用最少的组件，组装出一个仅支持输出功能的最简

```rust
fn main() {
    println!("Hello, world!");
}
```

```
[dependencies]
axstd = { path = "../../ulib/axstd" }
```

(a) 最简形式

```rust
fn main() {
    println!("{}",
        String::from("Hello,")
        + " world!");
}
```

```
[dependencies]
axstd = { path = "../../ulib/axstd", features = [
    "alloc",
    "alloc_slab", # "alloc_tlsf", "alloc_buddy"
] }
```

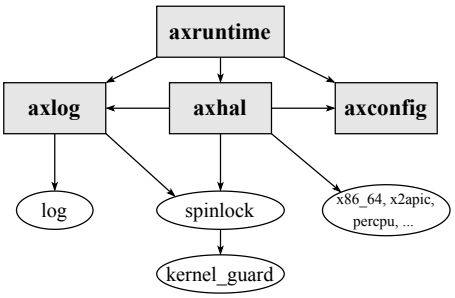
(b) 动态内存分配

```rust
fn main() {
    std::thread::spawn(|| {
        print!("Hello,");
    }).join().unwrap();
    println!(" world!");
}
```

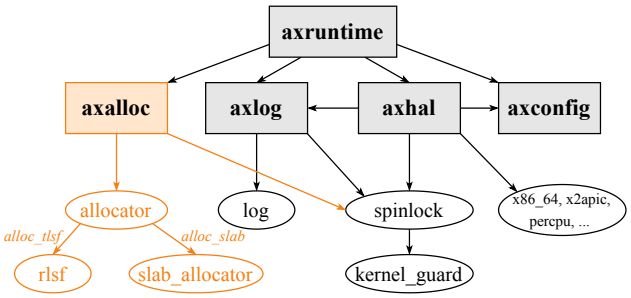
```
[dependencies]
axstd = { path = "../../ulib/axstd", features = [
    "multitask",
    "sched_cfs", # "sched_fifo", "sched_rr"
] }

```

(c) 多线程

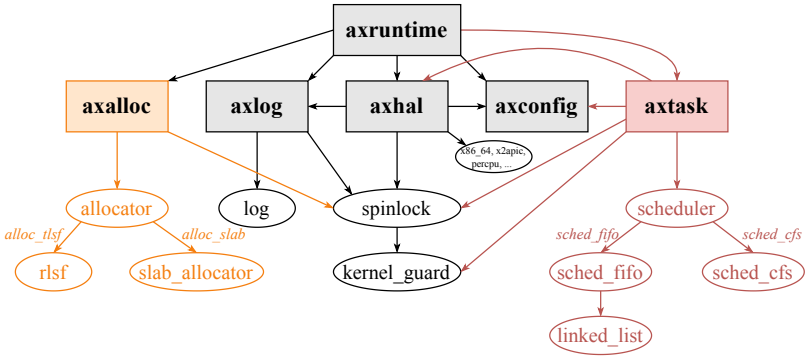
图 4.3　不同功能需求下的 helloworld 应用程序及其配置文件



(a) 最简形式



(b) 增加动态内存分配功能（启用 “alloc” 特性）



(c) 增加多线程功能（启用 “multitask” 特性）

图 4.4　不同功能需求下 helloworld 应用程序的组件依赖

操作系统。然后演示当应用的功能增加时，如何通过修改配置文件，快速构建满足应用需求的最简操作系统。

图 4.3(a) 是一个最简单的 helloworld 应用程序，仅完成输出一行字符串的功能。为了支持该应用程序的运行，需要针对运行的平台准备相应的执行环境，并提供 “输出” 的功能（如串口）。如图 4.4(a) 所示，ArceOS 最少需要四个模块层的组件来支持该应用的运行，这四个模块也是支持应用运行的最小组件集合。其中，最主要的是模块是 axruntime 与 axhal。axruntime 用于为应用建立执行环境，对引入的各组件进行初始化。axhal 是硬件抽象层，为各种硬件相关操作提供了统一的接口。此外，axconfig 定义了平台相关常量与内核参数，如物理内存范围、内核加载基地址、栈大小等。axlog 用于输出内核日志消息。

模块层的组件还会调用元件层的组件。如 axhal 会使用 x86_64、x2apic 这样的对具体硬件进行操作的组件；axhal 和 axlog 都会调用 spinlock 来使用自旋锁；spinlock 还会调用 kernel_guard 进行关中断或关抢占①。

当应用程序的功能需求增加时，会引入更多的组件，如图 4.3(b) 中的应用需要动态内存分配功能才能完成字符串拼接操作。为此，ArceOS 提供了一个 “alloc” 特性，应用在配置文件中启用该特性后，会引入一些额外的组件来实现动态内存分配功能（如图 4.4(b) 中的 axalloc 等）。类似地，如果应用需要使用多线程功能（图 4.3(c)），则可以启用 “multitask” 特性，并额外引入 axtask 等组件（图 4.4(c)）。其他可选组件还有设备驱动（axdriver）、网络（axnet）、文件系统（axfs）等。

应用程序通过指定不同的特性，来选择同一功能的不同实现。如图 4.3(b) 右侧，应用可在配置文件里启用不同的特性来指定相应的内存分配算法，并引入相应的组件依赖（图 4.4(b)）。图 4.3(c) 和 4.4(c) 也展示了应用对不同调度算法的配置及相应的组件依赖。

4.3.4　API 快速路径

ArceOS 利用了 Rust 的高级语言特性，为 Rust 应用程序调用内核的功能提供了快速路径，以缩短传统接口的调用路径，提高性能。

传统接口的不足。传统的通用操作系统，或一些微虚拟机操作系统，为了兼容已有应用，仍使用 POSIX 这样的接口作为应用程序调用内核的接口。POSIX 接口为宏内核而设计，考虑了诸多宏内核下的特点，但在微虚拟机场景下限制了灵活性，也存在许多冗余的调用路径。例如，宏内核的应用与内核处于不同地址空间不同特权级，应用需要通过系统调用访问内核。由于隔离性，系统调用的参数不能通过栈来传递，因此对参数的数量有限制（一般不超过 6 个），使得接口的参数

① 为了简洁性，这里省略了用于处理反向与循环依赖的 crate_glue 组件。

与数据类型不能太复杂。此外，宏内核对安全性的要求也使得内核不得直接访问存储在应用内存中的数据，因此内核在给应用返回一个对象时，需要使用一个整数编号作为抽象（如线程编号 pid 或文件描述符 fd），带来编号分配与查找的开销。另一方面，为减少系统调用次数，许多 POSIX 接口都提供了批处理的版本（如向量读写 `readv`、`writev`），这在微虚拟机场景下也是不必要的。反之，如果完全抛弃 POSIX 接口，重新设计一套针对微虚拟机优化的接口，又会导致无法兼容已有应用，带来不小的移植代价[19,22,26]。

以文件读写为例，如图 4.5(a) 所示，在 Linux 上 C 语言编写的应用程序要读取一个文件的内容时，使用文件描述符作为内核文件对象的中间表示。这不仅需要内核为每个进程维护一份文件描述符表，还会在文件描述符的分配与查找时带来不小开销，特别是在多核场景下严重影响可扩展性[8]。

```c
int fd = open("foo.txt", O_RDWR);
int len = read(fd, buf, 100);
close(fd);
```

(a) C

```rust
let mut f = File::open("foo.txt")?;
let len = f.read(&mut buf)?;
// drop(f);
```

(b) Rust

图 4.5　C 和 Rust 语言的文件操作接口

针对 Rust 应用的 API 快速路径。 ArceOS 希望能够提供一套接口，既能针对微虚拟机的特点进行优化，又能带来足够的兼容性。为此，ArceOS 借用了 Rust 的标准库 API，作为 Rust 应用访问内核服务的接口。如图 4.5(b) 所示，Rust 应用在进行文件操作时，采用了资源获取即初始化（RAII）风格的接口，在打开文件时会创建一个文件对象，并在其生命周期结束后自动释放以关闭文件。这不仅消除了文件描述符带来的间接开销，还让开发者无需自行考虑文件的关闭，避免了资源泄漏或多次释放。

在传统的宏内核下，难以将此类接口直接作为应用访问内核的接口。因此，现有的通用操作系统或微虚拟机操作系统，对 Rust 应用的支持都是通过 libc 实现的，Rust 标准库在内部仍会调用 libc 的库函数，使用传统的 POSIX 接口来访问内核，反而使 Rust 应用的调用路径比 C 还长。但微虚拟机让直接使用这样的接口成为可能。由于应用程序与内核处于同一地址空间，能够相互调度与访问各自的所有函数与内存，因此可以让 `File::open` 直接返回内核的文件对象，保存在应用的栈上，允许应用直接对其进行操作。

除了能消除 POSIX 接口带来的冗余路径外，使用 Rust 标准库的 API 还能利用 Rust 的众多高级语言特性。仍以图 4.5(b) 中的文件操作为例，Rust 的所有权机制确保了该文件对象只能在同一个线程中被访问，因此可以在编译期间检查出并发访问的缺陷，无需在内核中进行额外的运行时检查。此外，可以使用闭包、泛型等

高级数据类型作为接口的参数，以及使用带 async 关键字的异步函数[153]，为应用提供更多灵活性，并且无接口转换的开销。使用 Rust 风格的函数作为 API，还能实现从应用程序到用户库，再到内核的端到端 Rust 调用，中间不存在外部语言接口（FFI）的转换，不会丢失参数的类型信息，便于 Rust 编译器进行全局优化与安全检查[22]。

使用 Rust 标准库的接口还能一定程度解决应用兼容性的问题，因为可以直接支持已有的 Rust 应用程序。由于现在越来越多的新应用开始使用 Rust 语言编写，也有不少旧应用使用 Rust 语言进行重写[171]，这种方式可以带来足够的兼容性。

针对 C 应用的 API 快速路径。除了面向 Rust 应用的快速 API 外，ArceOS 也支持 POSIX API，以兼容已有的 C 应用程序。ArceOS 为 C 语言也提供了一定程度的快速路径，但由于 POSIX 接口的限制，效果不如 Rust。

以线程创建为例，当 C 应用程序要创建线程时，在 Linux 或一些 POSIX 兼容的微虚拟机操作系统上都是通过 libc 的 pthread_create 接口实现的，其内部会先使用 sys_mmap 系统调用分配线程的私有内存（栈与 TLS），再调用 sys_clone 完成线程的最终创建。然而，这些系统调用为了通用性，其参数类型与 pthread_create 存在较大差异，使得需要对传入的参数进行层层处理，延长了调用路径，带来了额外的开销（见第 4.5.2 节的实验评估）。

ArceOS 通过使用库函数级别的接口（API）代替系统调用级别的接口（ABI），以缩短 C 应用程序对内核服务的调用路径。ArceOS 的用户库提供了与 pthread_create 接口一致的 sys_pthread_create，来直接创建一个线程，而不再使用 sys_clone 系统调用。库函数级别的接口还有助于简化用户库的实现。例如，对于 libc 中的 malloc 接口，可通过 sys_alloc 直接调用内核的

表 4.3　ArceOS API 与 Linux 系统调用的差异

	Linux	ArceOS C 应用	ArceOS Rust 应用
创建线程	sys_clone 创建内核线程，libc 仍需维护自己的线程结构	提供与 pthread_create 一致的接口，用户库无复杂实现	提供与 Rust 标准库一致的接口，用户库无复杂实现
动态内存分配	libc 自行维护分配器，通过系统调用扩充容量	提供与 malloc 一致的接口，用户库无复杂实现	注册全局分配器后，无需显式接口与用户库实现
互斥锁	sys_futex 创建内核 futex 结构，等待时需进内核查找对应结构	提供与 pthread_mutex_* 一致的接口，用户库无复杂实现	内核返回等待队列结构，等待时直接调用
打开文件/套接字	内核返回文件描述符	内核返回文件/套接字结构，用户库维护文件描述符表	内核直接返回文件/套接字结构
使用文件/套接字	系统调用使用文件描述符作为参数，内核查找对应结构	将文件描述符作为参数，用户库根据描述符查找对应结构	将文件/套接字结构直接作为参数

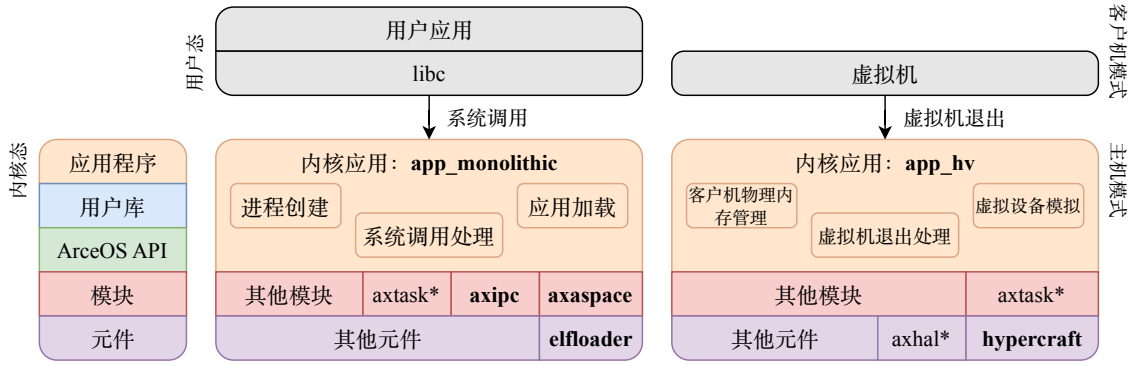
分配器实现动态内存分配，而无需在用户库中自行实现堆的管理（使用 `sys_brk` 或 `sys_mmap`）。

在本节的最后，总结了 ArceOS API（分别针对 C 与 Rust 应用）与 Linux 系统调用接口的几处典型差异，如表 4.3 所示。

4.3.5　多内核形态构建

组件化设计带来的灵活定制能力，可以让 ArceOS 不止是一个微虚拟机操作系统（Unikernel），还能形成其他形态的内核或系统软件，比如具有用户内核隔离的宏内核，或是能运行虚拟机的虚拟机管理程序。本节将对这两种内核形态的支持进行详细介绍。

由于微虚拟机单特权级的特点，ArceOS 采用以应用为中心的设计思想，通过在应用程序层（仍处于内核态）实现特权功能，以扩展为不同形态的特权软件，同时避免对原有架构与性能的破坏。为了便于区分 ArceOS 的应用程序层与宏内核中运行在用户态的应用程序，本节称 ArceOS 的应用程序层为*内核应用*（仍为内核的一部分），运行用户态的应用程序为*用户应用*。本节先分析这两种内核与 Unikernel 的差异，然后分别提出相应的内核应用设计方案。图 4.6 给出了利用 ArceOS 的组件实现的这三种内核的架构。



(a) Unikernel　(b) 宏内核　(c) 虚拟机管理程序

图 4.6　ArceOS 扩展为其他内核形态①

宏内核。 ArceOS 宏内核的目标是能二进制兼容已有的 Linux 应用程序。与 Unikernel 不同，宏内核*不信任应用程序*，因此需要采取种种措施，提供应用与内核以及应用间的隔离，以防应用程序破坏内核与其他应用。其次，宏内核一般也需要提供*多应用*的支持②。具体来说，宏内核与 Unikernel 的差异主要体现在以下几方面：

① 粗体的为新增组件，带 “*” 的为需少量修改的组件。
② 虽然这在一些 Unikernel[49,64,113,172] 中也得到了实现。

- *接口*。Unikernel 中，应用程序通过函数调用直接访问内核服务，而宏内核中需要通过系统调用，存在特权级的切换。使用系统调用接口的优势是容易实现对已有应用的二进制兼容，而不需要基于专门的用户库重新编译。不过也因此无法使用第 4.3.4 中介绍的 API 快速路径。
- *进程*。Unikernel 中没有进程的概念，或者内核和应用整体可视为一个进程，而宏内核一般提供多进程的抽象。进程可认为是共享相同资源的线程集合，多进程间需要有资源的隔离（例如地址空间、打开的文件等）。这些资源在 Unikernel 中可全局只维护一份，但在宏内核中需要为每个进程都维护一份。此外，宏内核还需提供进程间通信（IPC）的机制，例如信号、共享内存等。
- *地址空间*。宏内核的应用程序不再与内核共享同一地址空间，需要为每个进程创建单独的地址空间，并在进程切换时进行切换。多地址空间还影响应用程序的内存布局与系统调用时参数的传递，例如需要额外将用户参数拷贝到内核地址空间以防 TOCTOU（time-of-check to time-of-use）攻击。
- *应用程序加载方式*。为了支持运行多个应用，需要实现动态加载，而不能再将内核与应用静态链接在一起作为一个单独镜像。此外，在为新应用准备资源时，可能需要利用写时复制、延迟分配等虚拟内存技术来加速应用启动。

ArceOS 提供了一个内核应用 app_monolithic，其中包含了宏内核相比于 Unikernel 需要额外实现的部分，例如进程创建、用户应用加载、系统调用实现等（图 4.6(b)）。对于系统调用，由于宏内核无法利用 ArceOS 提供的 API 快速路径，需使用模块层提供的功能接口对 Linux 系统调用进行忠实实现。此外，原有的模块也需新增一些接口来实现与 Linux 兼容的系统调用，如增加线程克隆接口以实现 `sys_clone`。对于进程，ArceOS 宏内核沿用了 Unikernel 形态的线程管理组件（axtask），只是在线程控制块中多了一个额外的字段（以下称*资源控制块*），用于存放地址空间、文件描述符表等不与其他进程共享的资源（图 4.7），并在切换线程时进行切换。此外，还需新增一些组件用于实现宏内核特有的功能，如各种 IPC 机制、虚拟内存管理等。

虚拟机管理程序。本节讨论的是类型一的 hypervisor，即 hypervisor 管控所有硬件资源，拥有最高特权级，无需依赖主机操作系统。此时的 hypervisor 可认为是一种特殊的内核，只是需要额外考虑虚拟机的状态管理与事件处理，包括：

- *处理器状态*。hypervisor 通常使用传统内核中的线程来包装虚拟处理器（vCPU），以支持多核虚拟机。因此，需要在线程上下文中维护 vCPU 的状态，并实现相应的切换流程。
- *虚拟机抽象*。类似于宏内核中的进程，hypervisor 需要为每个虚拟机维护一

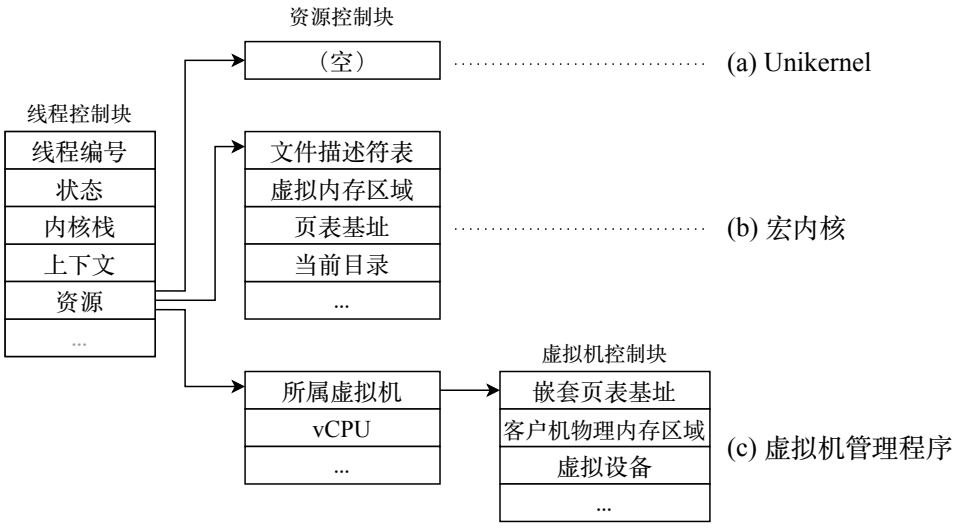


图 4.7　ArceOS 不同形态下的线程控制块结构

个数据结构，即同一个虚拟机使用的资源，可在该虚拟机的所有 vCPU 间共享，但在不同虚拟机间相互隔离。例如地址空间（GPA 到 HPA 的映射）、虚拟机设备列表等。

- 虚拟机退出的处理。类似传统内核的异常处理，hypervisor 需要注册一个虚拟机退出处理例程，该例程主要用于实现虚拟设备的模拟，包括 I/O 指令的模拟、中断注入等。

ArceOS 提供了一个元件层组件 hypercraft，其中封装了基本的虚拟化功能，例如不同架构下的 vCPU 抽象、嵌套页表实现、虚拟机退出例程的注册等。其余与 hypervisor 的功能紧密相关的部分实现在应用程序层（app_hv），例如客户机物理内存的管理、虚拟机退出的处理、虚拟机设备的模拟等（图 4.6(c)）。与宏内核类似，此时的线程控制块仍需维护额外的资源控制块，其中包含 vCPU 的状态、线程所属的虚拟机等不在虚拟机间共享的资源（图 4.7）。ArceOS 通过将资源控制块的具体结构交给其他组件来定义（这里指内核应用 app_monolithic 和 app_hv），使得可以不改模块层的 axtask 的代码，只改应用程序层的代码，就能扩展出满足不同内核需求的线程控制块。

4.4　系统实现

本节将详细介绍 ArceOS 中几个核心模块与元件的设计与实现。ArceOS 目前实现了多核、多线程、网络、文件系统、图形显示等基本内核功能，支持 x86_64、RISC-V、AArch64 等主流处理器架构，QEMU/KVM、x86 服务器、树莓派、黑芝麻等多种硬件平台，以及 Virtio、Intel 82599 万兆网卡等设备驱动（均可根据应用

表 4.4　ArceOS 组件列表

层次（组件总数）	组件	代码行数[b]	描述
元件层（33）[a]	linked_list	445	支持常数时间删除的链表[c]
	crate_glue	167	提供跨组件调用机制
	kernel_guard	175	创建关中断或关抢占的临界区
	spinlock	301	内核自旋锁
	percpu	213	CPU 私有数据的定义与访问
	scheduler	388	具有统一接口的多种任务调度算法
	allocator	551	具有统一的接口的多种内存分配器
	slab_allocator	423	基于 slab 的内存分配器
	page_table	401	针对不同硬件架构的通用页表结构
	page_table_entry	423	多种硬件架构下的页表项定义
	driver_virtio	316	各类 Virtio 设备的驱动程序
	axfs_vfs	404	虚拟文件系统接口
	axfs_ramfs	349	内存文件系统
	…	…	…
	小计	7,724	—
模块层（11）	axruntime	284	应用运行时
	axhal	3,318	硬件抽象层
	axalloc	208	全局内存分配器
	axtask	917	任务管理模块
	axsync	187	同步互斥模块
	axdriver	895	设备驱动模块
	axnet	1,203	网络模块
	axfs	1,529	文件系统模块
	…	…	…
	小计	8,882	—
API 层（2）	arceos_rust_api	532	为 Rust 应用程序提供的 API
	arceos_posix_api	3,662	为 C 应用程序提供的 API（POSIX 接口）
用户库层（2）	axstd	1,066	为 Rust 应用程序提供的精简版标准库
	axlibc	9,460	为 C 应用程序提供的精简版 libc 库
合计（48）	—	31,326	—

[a] 不包括其他开发者提供的已发布到 crates.io 中的组件。
[b] 不包括配置文件。
[c] Rust 核心库提供的链表不支持该操作。

需求进行灵活定制），总代码量达 3 万多行。表 4.4 给出了 ArceOS 目前在各个层次的核心组件列表及代码量[①]。

4.4.1　核心组件

本节从组件的角度，介绍 ArceOS 中一些最核心的、几乎会被所有应用都使用的组件（模块层）。

应用运行时（axruntime）。该组件负责对内核所启用的几个功能模块进行统筹管理与初始化，会在系统及硬件平台初始化完成后（由 axhal 实现），进入应用程序的主函数之前被执行。axruntime 会根据应用指定的特性，选择性依赖其他模块层组件，并通过条件编译选择性对它们进行初始化。此外，由于它作为所有应用都必须包含的模块，管控着所有其他模块，所以也会注册一些全局处理例程，如向其他模块分发中断与异常，注册 Rust 的崩溃（panic）处理例程等。

硬件抽象层（axhal）。该组件封装了不同平台的硬件相关操作，并向上提供了统一的接口，使得其他模块在调用时无需关心具体的硬件细节，只需实现硬件无关的代码。ArceOS 支持多种硬件平台，不同平台有着不同的启动流程、处理器状态以及平台相关设备[②]，均在 axhal 中完成配置与初始化。当系统启动后，首先会在 axhal 中执行硬件相关的初始化流程，再跳转到 axruntime 中执行硬件无关的初始化流程。此外，axhal 也提供了对架构相关指令与结构的封装，如开关中断、切换页表等指令，以及页表、异常与中断的向量表与上下文等结构，并实现了发生异常与中断时的状态保存与恢复。

由于硬件平台的多样性以及硬件操作的繁琐性，axhal 可能因此变得十分臃肿与复杂。幸运的是，其他开发者已经提供了众多封装好的硬件相关操作（通过 crates.io 发布），例如 x86_64、x2apic、aarch64-cpu 等 Rust 包，也包括 ArceOS 元件层的一些组件，axhal 可以重用这些组件而大大简化自身的实现。

动态内存分配（axalloc）。该组件主要包含了两个内存分配器：页分配器与字节分配器。其中页分配器只能返回一个或多个连续的且对齐的内存页（4KB），用于物理页帧或 DMA 内存的分配。字节分配器即一般的堆分配器，可以返回一块任意大小的内存。axalloc 的页分配器基于位图（bitmap）来实现，而字节分配器根据应用程序指定的特性，可选 slab、TLSF、伙伴系统等多种分配算法。此外，axalloc 还提供了查询各分配器已用的内存、剩余的内存等接口。

在初始化时，axruntime 会向 axalloc 提供所有可用的物理内存段，先将它们全部用于页分配器，并将字节分配器设为空。当后续按字节分配失败时，会请求页

① 基于该版本统计：https://github.com/rcore-os/arceos/tree/ba8f1fc7a18ee1ac4711afa4f0d984498534f114。
② 这里指串口、时钟、中断控制器等简单但必需的设备，而不包括网卡、磁盘等复杂且可选的设备。

分配器分配一段连续的物理页，将它们添加进字节分配器，然后再次尝试字节分配直到成功。字节分配器每次扩充的容量下限为当前容量，使得每次扩充都会让容量至少翻一倍。

axalloc 还负责将字节分配器注册为 Rust 的全局内存分配器，使得应用程序与内核共用一个分配器。其他所有用到动态内存分配或释放的操作（如使用 `Box<T>`、`Vec<T>` 等类型），均会被重定向到全局分配器中。

4.4.2　任务管理

ArceOS 的任务管理功能实现在 axtask 组件中。在微虚拟机操作系统的设计中，一般只有线程而没有进程的概念（事实上可将整个微虚拟机看做一个进程），ArceOS 将线程作为最基本的调度单元，并称为一个任务①。

该组件实现了多任务的调度以及任务生命周期的管理（创建、退出、睡眠、唤醒、切换），并支持通过特性指定不同的任务调度算法。最简单的是 FIFO 协作式调度，只有当任务主动让出时才会发生任务切换，适合基于事件循环的运行到完成式的应用。其他支持的调度算法还有时间片轮转、完全公平调度等，它们都是抢占式的调度算法，适合对延迟或公平性有要求的应用。该组件会从 axruntime 接收时钟中断，并决定是否要进行抢占。此外，在 axtask 中还实现了等待队列，用于为上层模块提供基本的线程同步机制。例如 axsync 基于 axtask 的等待队列实现了互斥锁、信号量、条件变量等更多高级同步原语。

axtask 也可用于本文第 3 章提出的库操作系统，作为 Skyloft 用户级线程库的实现，以进一步提高组件的可重用性。

4.4.3　设备驱动

ArceOS 的 axdriver 组件中实现了对设备驱动层的抽象。该组件提供了对各类设备的抽象与封装，目前 ArceOS 支持三种设备的抽象：网络设备、存储设备、图形显示设备，它们分别具有一组统一的接口，用于实现类型相同但型号不同的设备驱动组件。例如，对于存储设备，接口主要是数据块的读写，对于网络设备则是数据包的收发以及数据包缓冲区的分配与释放。

axdriver 会注册几种支持的设备型号，应用程序可通过特性在已注册的设备中选择要使用的具体设备。axdriver 只负责将选定的设备驱动进行组合，而驱动的具体实现会拆分到元件层的组件中。表 4.5 列出了 ArceOS 目前支持的设备型号，其中大部分是 Virtio 设备[84]。此外，应用程序也可通过特性指定设备的探测方式，如通过 PCI 总线还是设备树。

① ArceOS 的宏内核形态可基于任务扩展出进程结构（第 4.3.5）节。

如果应用程序指定了多个同一类型的设备，axdriver 需要将它们放入一个列表中，并通过统一的接口进行访问。然而，不同的设备在实现时会被组织为不同的数据类型，在进行功能调用时会涉及数据类型的动态分发[173]（例如通过虚函数表），带来一定的调用开销，且无法使用内联优化。为此，axdriver 提供了一种特性配置，可通过限制同一类型只能有一个设备，从而使用静态分发以提供最佳性能。如不启用该特性则仍为多设备动态分发。

表 4.5　ArceOS 设备驱动列表

设备类型	设备型号	描述
存储	ramdisk	内存模拟磁盘
	virtio-blk	Virtio 块设备
	bcm2835-sdhci	树莓派 SD 卡控制器
网络	virtio-net	Virtio 网卡
	ixgbe	Intel 82599 万兆网卡
显示	virtio-gpu	Virtio 显卡

4.4.4　网络栈

ArceOS 的网络功能实现在 axnet 组件中。该组件不仅是对网卡驱动与网络协议栈组件的组合，还为应用程序提供了基于套接字（socket）的网络接口。ArceOS 引入了已在嵌入式与实时系统中被广泛使用的 smlotcp[174]（Rust 编写）和 lwip[175]（C 编写）作为网络协议栈，两者都以元件层组件的形式提供。这两个组件都提供了各自的网卡设备抽象，通过为 axdriver 中的网卡驱动实现该接口，可将驱动与协议栈结合。axnet 在这两个协议栈提供的套接字接口上再封装了一层，使得接口统一且与 POSIX 接口更接近，例如提供 TCP 套接字的 `connect`、`bind`、`listen`、`accept`、`send`、`recv` 等操作。除了 TCP，axnet 还提供了 UDP 套接字、DNS 解析等多种网络功能。

为了给应用提供最佳性能，目前 ArceOS 的网络访问是基于轮询的。在每次调用套接字操作后，都会轮询网卡，将收到的数据包传入协议栈以推进状态机，或发送已在协议栈发送缓冲区的数据包。此外，网卡驱动也使用一组零拷贝的接口进行数据包的收发。例如，在接收数据包时，会预先分配一堆数据包缓冲区，并放入网卡的接收队列。当网卡收到数据包后，会直接将该缓冲区传递给协议栈以避免拷贝，协议栈处理完成后将缓冲区再次放入接收队列。

4.4.5　文件系统

文件系统也是操作系统中一个容易被解耦的功能，因此它们需要尽可能被实现为一个元件，便于被其他系统重用。ArceOS 通过一个元件层组件 axfs_vfs 定义了虚拟文件系统层，提供对文件系统、文件节点的接口定义。在实现具体的文件系统时，只需为其包装一层该组件提供的接口，就可被 ArceOS 使用。ArceOS 目前以这种方式实现了多种文件系统组件，例如内存文件系统 ramfs、设备文件系统 devfs、进程文件系统 procfs 等。此外，ArceOS 还通过重用他人编写的组件 rust-fatfs[176] 支持 FAT 文件系统。

在模块层中的 axfs 中，会根据应用程序提供的配置，静态选择所需的文件系统，进行初始化并挂载。与 ArceOS 的设备驱动一样，静态配置也减小了虚拟文件系统层的动态分发开销（见第 4.5.3 节的实验）。此外，axfs 在内核文件对象之上，为应用程序提供类似 Rust 的标准库接口，作为实现 API 快速路径的基础，如 `File::open`、`File::read` 等，

4.5　实验与评估

本节通过一系列实验评估基于组件化设计的 ArceOS 在几大内核基础功能上性能表现，包括线程管理、文件与网络操作，以及针对 Rust 与 C 应用的 API 快速路径的优化效果。最后，本节评估了 ArceOS 在真实世界应用上的性能。

4.5.1　实验环境

本章的实验在一台桌面计算机上进行，CPU 型号为 4 核 AMD Ryzen 3 5300G，主频为 4.0 GHz，内存为 16 GB，并且关闭了超线程与动态频率调节。实验机器运行 Ubuntu 18.04 操作系统，内核版本为 Linux 5.11.0，并使用 QEMU/KVM 作为虚拟机管理程序运行 ArceOS 及其他 Unikernel。

实验以 Unikraft[28]（v0.15.0）这一目前最先进的 Unikernel，以及 Linux 作为基线。其中 Unikraft 和 ArceOS 一样运行在虚拟机中，Linux 运行在裸机上，以演示微虚拟机技术相对于主机原有应用程序的性能提升。由于 Unikraft 依赖 musl libc 库[177] 才能支持大部分 POSIX 操作，为公平比较，实验在评估 Linux 上的应用程序时也链接了相同版本的 musl libc 库。

4.5.2　线程操作性能

本实验分别使用 C 语言和 Rust 语言编写应用程序，评估 ArceOS 与基线系统上的几种线程操作的用时，包括线程的创建、切换与同步，以及在这些操作中利用

API 快速路径的优化效果。

线程创建。由于线程创建对内存分配较为敏感，因此实验将所有系统上的线程栈大小均设为 4KB，并将 ArceOS 与 Unikraft 均配置为使用 TLSF[168] 分配算法，而 Linux 因难以修改内核的内存分配算法仍使用默认的。创建的线程均为空线程，不执行任何操作直接退出。

图 4.8 给出了在各个系统上使用 C 与 Rust 语言创建一个空线程所需的时间，以及时间的主要组成部分。在 Linux 与 Unikraft 上，C 应用通过 POSIX 接口 `pthread_create` 创建线程，其开销主要来自 `sys_mmap` 与 `sys_clone` 系统调用，分别用于分配线程私有内存（栈与 TLS）以及最终的线程创建，此外还有很大一部分用户库的开销，用于为这几个系统调用准备参数。

在 Unikraft 中，主要因为避免了系统调用时的上下文切换而使开销大大减小。然而，由于 Unikraft 注重对 POSIX 接口的兼容性，使得在该简单场景下仍有大量不必要的操作。图 4.8 中的 Unikraft-opt 为经过笔者优化的版本，消除了多余的内存清零与 POSIX 线程结构的初始化，可以使开销进一步降低。但由于仍使用低效的 `sys_clone` 接口以及实现上的问题，每次线程创建仍需多次内存分配与拷贝，导致开销依然可达数百纳秒。

此外，对于 Rust 应用，在 Linux 和 Unikraft 上均需依赖 libc 库以提供系统级功能支持，使其最终仍会调用 `pthread_create` 创建线程，比 C 还多了一部分 Rust 标准库的开销。

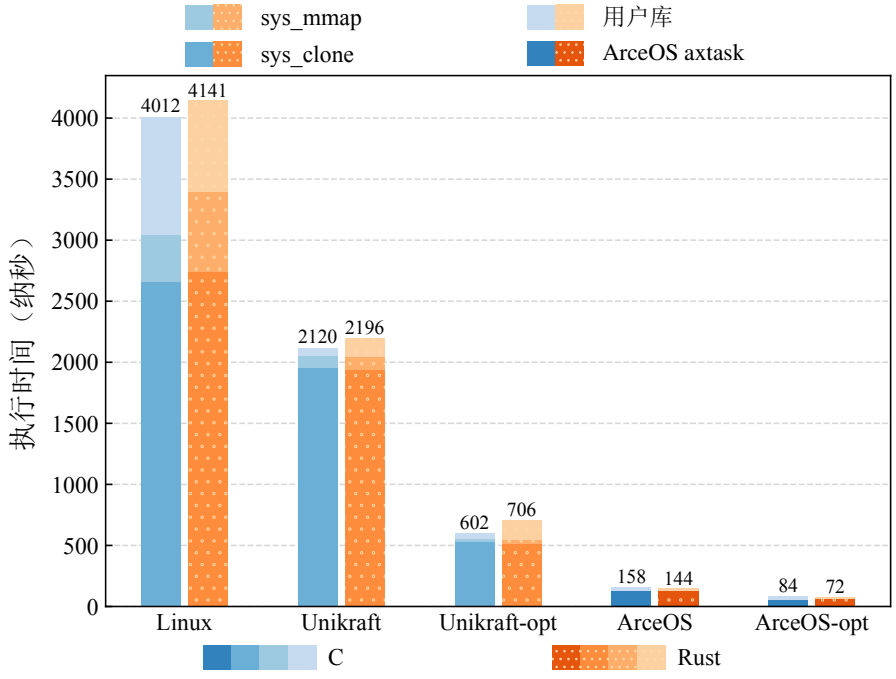


图 4.8　几种系统的线程创建性能剖析

ArceOS 的线程创建开销非常小，主要原因是使用了 API 快速路径，简化了用户库以及底层模块（axtask）的实现，而无需经过 `sys_clone` 的转换。同时，Rust 比 C 应用具有更小的开销，因为 Rust 用户库的接口与底层模块的接口更接近。此外，对于此类简单场景，ArceOS 还允许配置为不启用浮点数与 TLS，从而免去线程创建时浮点上下文与 TLS 结构的内存分配与初始化，使得线程创建的开销可以低至几十纳秒（图 4.8 中的 ArceOS-opt），与用户级线程[95,153]相媲美。

Unikraft 虽然提供了众多配置选项，但并不能关闭浮点数与 TLS。因此本实验也说明了，即使对于 Unikraft 这样的目前最先进的、能根据应用需求进行灵活定制的微虚拟机操作系统，其专用化程度仍然不够，仍具有较大的性能提升空间。在 Linux 中实现这种程度的定制则更为困难。

线程切换与同步。为了评估线程切换的开销，本实验创建了两个线程，让它们不断调用主动让出操作（yield），使这两个线程来回切换，并测量单次切换花费的时间。此外，实验还使用条件变量，测量了让当前线程睡眠，唤醒另一线程并切换过去的总时间，以评估线程同步的开销。

实验结果如表 4.6 所示。对于线程让出操作，由于 C 和 Rust 均提供无参数的简单接口，这两种应用的执行时间几乎一样，但 Unikraft 和 ArceOS 的开销比 Linux 低一半以上。对于条件变量操作，在 Linux 和 Unikraft 上，无论是 C 还是 Rust 均通过底层的 `sys_futex` 系统调用直接实现，因此这两种应用的执行时间也差不多，不像线程创建时 Rust 需要间接调用 C 库，而使开销更大。对于 ArceOS，可以使用 API 快速路径避免 `sys_futex` 带来的接口转换，因此开销比 Unikraft 更低。此外，将 ArceOS 配置为禁用浮点数与 TLS，能进一步提高这两类线程操作的性能，例如线程切换的时间减少了近 90%。

表 4.6　几种系统的其他线程操作性能对比

系统	线程让出（纳秒）		条件变量（纳秒）	
	C	Rust	C	Rust
Linux	408	408	1,070	996
Unikraft	199	199	279	274
ArceOS	179	178	212	204
ArceOS（无浮点）	107	109	140	132
ArceOS（无 TLS）	89	89	125	118
ArceOS（无浮点无 TLS）	19	19	45	39

4.5.3　文件操作性能

本节的实验同样对 C 和 Rust 应用分别进行评估。实验先创建了一个 1MB 大小的文件，然后分别测量打开、读取 1 字节与写入 1 字节的执行时间，为了单独评估文件操作的开销，避免磁盘访问的影响，本实验在所有系统上均使用内存文件系统（ramfs）。

表 4.7 给出了实验结果。对于 Linux 和 Unikraft，由于 Linux 系统调用的开销与实现的复杂性，其性能最差。Unikraft 大大优化了文件读写的性能，与 Linux 相比分别有 61% 和 79% 的性能提升，但对于打开文件操作的性能提升不大（12%）。此外，从使用的编程语言来看，Rust 在打开文件操作上的调用路径比 C 更长，速度稍慢，但在文件读写操作上与 C 几乎一样。

表 4.7　几种系统的文件操作性能对比

系统	打开文件（纳秒）		读取单字节（纳秒）		写入单字节（纳秒）	
	C	Rust	C	Rust	C	Rust
Linux	618	686	217	220	421	411
Unikraft	542	597	75	76	75	76
ArceOS	88	51	39	19	39	19

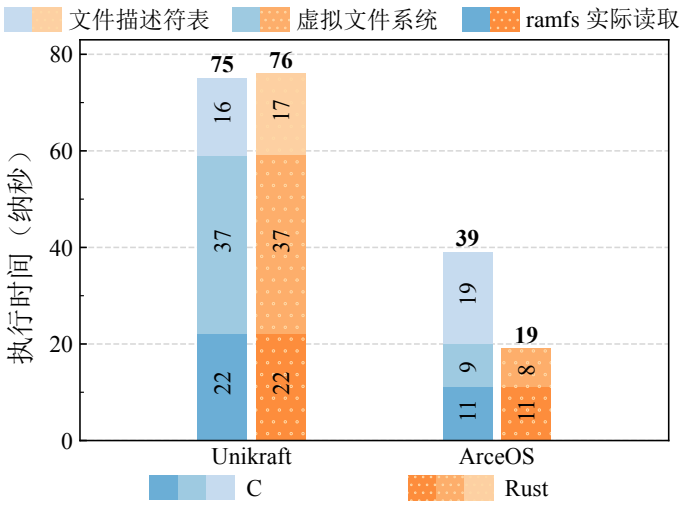


图 4.9　ArceOS 与 Unikraft 文件读取操作的性能剖析

ArceOS 在这三者中具有最佳的性能，相比于 Linux，这些文件操作的性能都提升了 90% 左右。此外，与前两者不同，ArceOS 对 Rust 应用的性能优化效果更明显，Rust 比 C 的读写性能快了 50% 左右。为了进一步分析 ArceOS 性能好的原因，

实验还将 ArceOS 与 Unikraft 的文件读取操作进行分解，分成三个阶段分别测量执行时间，分别为文件描述符表的查询、虚拟文件系统层的分发，以及内存文件系统的实际读取，如图 4.9 所示。可见 Unikraft 的主要开销花费在虚拟文件系统层，这是因为 Unikraft 的普通文件读写与使用 I/O 向量的文件读写共用一个接口，使得需要额外进行一些内存分配与拷贝操作，但在 ArceOS 中则没有这部分的开销。另一方面，对于 Rust 应用，ArceOS 还利用 API 快速路径完全消除了文件描述符表的开销。

此外，实验还评估了读写不同大小的数据块对性能的影响，如图 4.10 所示。对于小数据块（⩽ 1KB），在 ArceOS 中 Rust 相对于 C 的性能提升比较明显，可达 41.5% ∼ 58.5%。但对于大数据块（> 16KB），API 快速路径的优化效果会逐渐被数据拷贝所掩盖，使得 C 和 Rust 的性能差距逐渐缩小（小于 4%）。

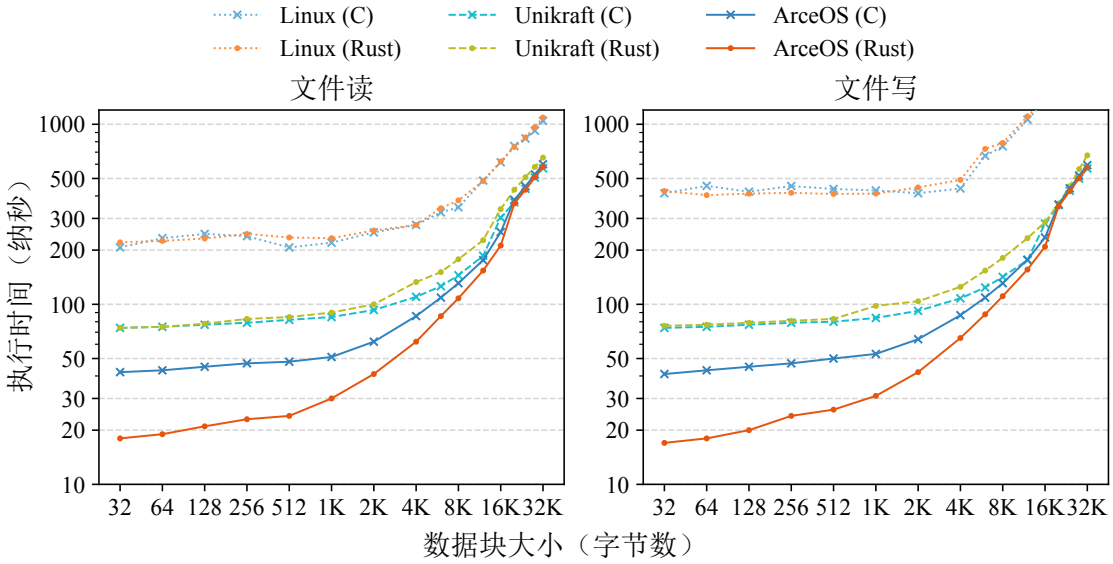


图 4.10　几种系统使用不同大小数据块的文件读写性能对比

4.5.4　网络操作性能

本节的实验在各系统上运行一个常用的网络性能评估工具 NetPIPE[178]。该工具会在客户端与服务器之间收发不同大小的 TCP 消息，测量单向传输延迟与相应的网络带宽。实验另使用一台机器作为客户端，CPU 型号为 32 核 Intel Xeon E5-2683 v4，主频 2.1 GHz。服务器与客户端均通过 Intel 82599ES 10 Gbps 网卡进行网络传输。

本实验的比较对象仍为 Linux、Unikraft 和 ArceOS。其中 Linux 运行在裸机上，使用内核的网络协议栈与 ixgbe 网卡驱动。Unikraft 运行在虚拟机中，由于其不支持物理网卡驱动，只能使用 Virtio 网卡，并通过 Linux 的网桥与物理网卡交换数据包。ArceOS 也运行在虚拟机中，不过可以利用组件化的优势方便重用他人编写的

物理网卡驱动，例如这里使用了 ixy.rs[179] 中的该型号网卡驱动。物理网卡会通过 Linux 的 VFIO[180] 机制直通给 ArceOS 虚拟机，以提供与裸机相当的性能。ArceOS 的网络协议栈使用 smoltcp，其余网络实现细节见第 4.4.4 节。

实验将待测系统分别运行在客户端或服务器端，具体配置以及实验结果如图 4.11 所示。比较 ArceOS 与 Linux，对于短消息，例如在 64B 下 Linux 与 ArceOS 的传输延迟分别为 29.0μs 与 8.0μs，Linux 为 ArceOS 的 3.6 倍，这是因为短消息的传输开销主要在于系统调用，ArceOS 利用微虚拟机的单地址空间优势、快速路径优化以及更精简的网络协议栈大大减小了这部分的开销。但对于长消息（> 64KB），ArceOS 的性能开始比 Linux 差，而且最大带宽仅达到 3.6Gbps。因为 ArceOS 虽然在驱动程序中尽量避免了拷贝，但目前在网络协议栈层面的优化不足，还未提供零拷贝接口，对长消息需要多次拷贝而导致性能不如 Linux，这也是 ArceOS 今后需要改进之处。

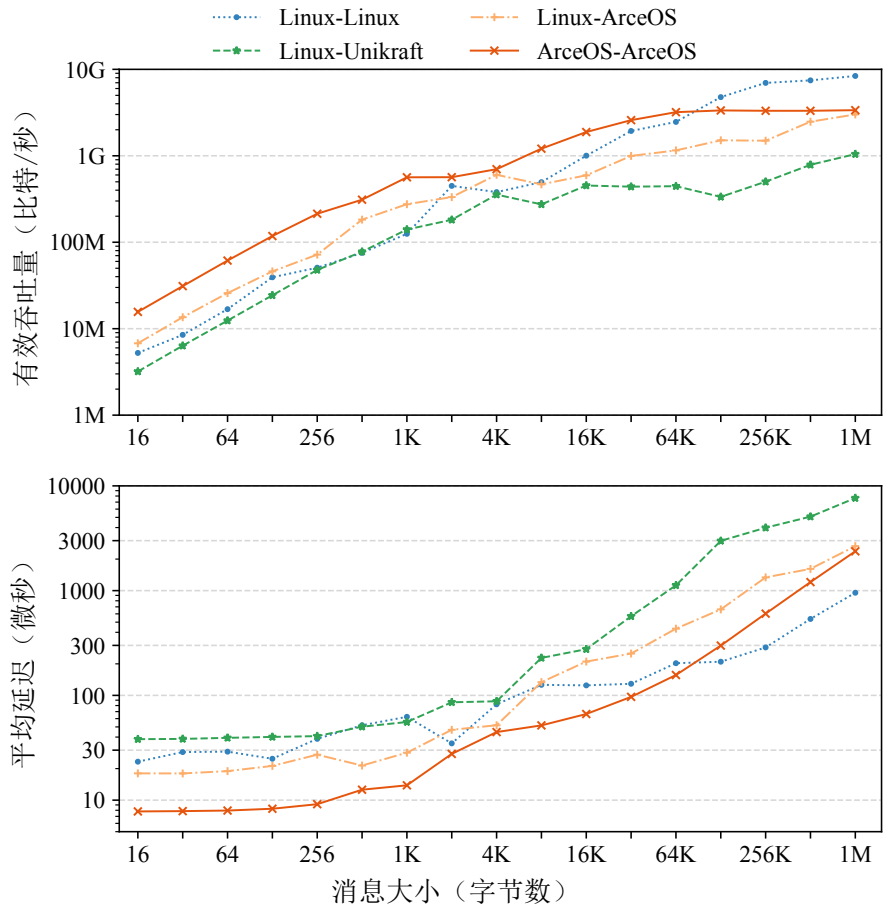


图 4.11　几种系统的网络传输性能对比

为比较 ArceOS 与 Unikraft，实验让客户端均运行 Linux，服务器运行相应的系统。无论是小数据包还是大数据包，Unikraft 比 Linux 的性能都差，这是因为其虽然也消除了系统调用的开销，并同样使用了轻量的 lwip 协议栈，但由于不支持

物理网卡驱动，仍需通过主机 Linux 与物理网卡交换数据包。此外，对于一端运行 Linux 而另一端运行 ArceOS 的情况，与两端都运行 ArceOS 类似，也是小数据包表现较好，大数据包表现较差，不过整体性能仍优于 Unikraft。

4.5.5　综合应用性能

ArceOS 除了直接兼容 Rust 应用外，也为 C 应用提供了基本的 POSIX 支持，能运行一大批已有的复杂 C 应用，并可利用组件化的优势根据场景进行针对性的优化。本节主要评估在 ArceOS 上运行目前被广泛使用的 Redis[181] 数据库（v7.0.12）的性能。为支持运行 Redis，ArceOS 需要依赖大部分模块层组件，包括任务管理、设备驱动、网络、文件系统等。

本实验采用与第 4.5.4 节的网络实验相同的配置。Redis 服务器被配置为内存数据库，客户端使用 Redis 自带的 `redis-benchmark` 工具向服务器发送 GET 请求，采用默认配置（不启用流水线，数据大小为 3 字节），然后不断增加客户端的并发连接数并测量相应的吞吐量与 99% 尾延迟。

实验结果如图 4.12 所示。运行在虚拟机中的 ArceOS 比裸机上的 Linux 具有更

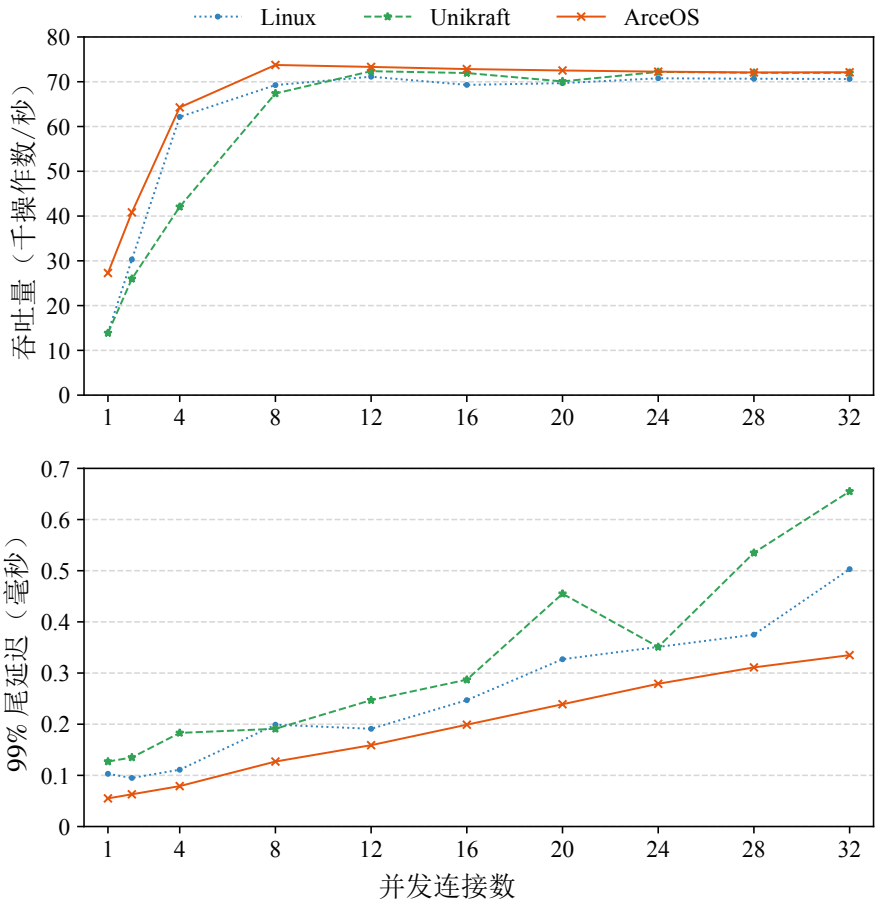


图 4.12　几种系统的 Redis GET 请求性能对比

高的吞吐量与更低的尾延迟，例如在连接数为 32 时比 Linux 的尾延迟低了 33.4%。因为 Redis 的请求都是小数据包，使得 ArceOS 在该场景下表现优异（正如第 4.5.4 节所述）。Unikraft 因不支持物理网卡驱动而表现较差。

本节的实验主要为说明 ArceOS 能根据已有应用的需求快速打造出合适的微虚拟机操作系统，短时间带来性能的提升。虽然 Linux 也可使用 DPDK[31] 等内核旁路技术提供高性能的网卡驱动①，但需要额外实现用户态网络协议栈，提供套接字接口才能运行已有的应用，例如本文第 3 章的 Skyloft 系统。相比之下，ArceOS 通过对组件的组合来支持 Redis 这样的复杂应用，同时具有高性能、兼容性与灵活性三大优势。既提供绕过内核的高性能网络栈，又兼容已有应用，还能方便驱动程序、协议栈等组件的定制与重用。

4.6　讨论

本节简要讨论 ArceOS 的局限性与未来工作。

彻底消除循环依赖。目前，ArceOS 只是利用 crate_glue 明确标记循环的组件依赖，但仍没有完全消除循环依赖（第 4.3.2 节）。当所有直接与间接的循环依赖被消除后，将有助于模块化的测试、调试与形式化验证。例如，一个组件中的缺陷，只可能影响所有依赖它的组件，而不会在整个系统中传播。为彻底消除循环依赖，一种方法是将大组件拆分为几个小组件的方式，不过这可能会使组件间的逻辑关系变得混乱，增大开发难度，也让代码可读性变差。因此，如何提供一种不存在循环依赖，而且不影响开发便利性的组件拆分方式，仍有待进一步研究。一个可能的思路是在编程语言层面引入一些新的特性。

微内核形态支持。第 4.3.5 节讨论了如何将 ArceOS 从 Unikernel 的基本形态扩展为宏内核与虚拟机管理程序。对于微内核形态的扩展，与宏内核基本类似，也需要提供应用程序与内核的地址空间以及特权级隔离，并实现 IPC 机制。不过，由于微内核的各种服务（如设备驱动、网络、文件系统）也运行在用户态，它们之间需要使用专门的 IPC 接口进行通信，而不像 Unikernel 与宏内核那样直接使用函数调用。因此，支持微内核的一个挑战是，如何在不对现有内核功能模块做太多修改的情况下，让它们同时支持 IPC 与函数调用接口，并可通过相应的配置选项进行切换。另一方面，IPC 也会成为微内核的性能瓶颈，需要针对 IPC 机制进行深度优化。

① Unikraft 也可利用 vhost-user 在虚拟机中使用 DPDK。

4.7　本章小结

高性能的微虚拟机操作系统需要能够方便地针对应用程序的需求进行专用化定制。为此，本章提出了一种组件化的操作系统设计方法 ArceOS，通过基于设计理念相关性的组件划分，能够降低组件间的耦合性，并提供良好的可定制性与可重用性，例如构建出多种形态的内核。ArceOS 利用 Rust 语言的标准库接口提供 API 快速路径设计，既获得了比 POSIX 接口更好的性能，又具有足够的兼容性。实验表明，ArceOS 能够为应用快速构建出满足需求的专用操作系统，而且具有比 Linux 等系统更高的性能。