Rhodes: A Next-Generation OS based on Resource Governance Model

Changsheng Wang

Abstract

The kernel is the core part of an operating system. At present, four types of kernels are widely available: monolithic kernels, microkernels, hybrid kernels, and exokernels. These kernel architectures are either too tightly-coupled, like monolithic kernels, put too many modules into the kernel, or too free, like exokernels, allow users to freely develop core features such as IPC, Virtual Memory, etc, or have no rules, like hybrid kernels, kernel modules organization is uncontrolled. In recent years, to solve these problems, resource management concept frequently appears in industry and academia. Although resource management concept makes resources in OSes orderly, no existing OSes or software systems can modify and extend management rules conveniently and dynamically.

This paper proposes a resource governance model that places resources such as applications and hardware into the model. Through unified rule governance, the organization of resources is more orderly and scalable. At the same time, by separating rules from resources, rules in the model are also scalable and flexible. Based on the model, we built Rhodes OS that allows users to modify and extend resource management rules by governance center which separates rules from resources. In addition, Rhodes also fits all kinds of OSes that are optimized for specific scenarios such as mobile, big data, IoT and AI. We implemented Rhodes on x86-64 and evaluated it. Our evaluation results show that applications performance in Rhodes is equivalent to Linux in native mode. So we believe that OSes will enter resource governance age from resource management age in the near future.

Key words monolithic kernel; microkernel; hybrid kernel; exokernel; resource governance

Ⅰ Introduction

Modern operating systems always constantly balance security, performance, and scalability, striving to find the perfect solution to achieve the best of the three.

The monolithic kernel operating system represented by Unix uses the POSIX interface to implement that only trusted kernel primitives can operate hardware resources. For example, if a user process writes data to a file, it needs to call Unix kernel functions through the POSIX interface. Then, kernel functions synchronously write the memory page and asynchronously flush the dirty page to the hard disk. Because only kernel functions can directly operate the hard disk and user processes can't be accessed to do it, key data is protected. At the same time, due to the mechanism of synchronous writing of memory and the asynchronous disk flushing implemented in the kernel, it reduces the switching between user mode and kernel mode to guarantee the performance of IO processing.

The microkernel operating system represented by L4 [5] minimizes the kernel to only IPC, interrupt, thread scheduling, etc., and improves the communication performance between user processes and kernel processes through asynchronous IPC and IPC hardware. At the same time, minimizing the kernel provides a basis for the diversified customization of upper-layer applications. In addition, through the security inspection of IPC, key data is effectively protected.

Exokernel [4] operating system minimizes kernel functions to only security bindings and lets users implement IPC, thread scheduling, and interrupt. The design provides a basis for customized optimization of application performance. For example, relational databases and garbage collectors sometimes have predictable data access patterns, and the general LRU page policy of operating systems had a performance impact on their data access, so P.Cao, etc. [6] implemented that applications directly control the file cache, it finally reduced the program running time by 45%. In addition, applications are securely bound with the hardware through LibOS, which prevents the application from accessing untrusted hardware resources.

Each kernel architecture is trying to maximize security, performance, and scalability. However, we also find that no matter which kernel architecture is used, it still exists the following problems:

The code is tightly coupled and has poor portability. For example, if we want to migrate the LRU policy of the page in a Unix to an exokernel OS, we have two solutions. One is we must strip the strategy from the Unix kernel and migrate it into the exokernel OS, the other is to directly re-implement this policy in the exokernel OS. Both of them have a too-high migration cost.

It seems that the LRU policy can be easily migrated from microkernel OSes to exokernel OSes because microkernels are layered clearly. However, if microkernel OSes invoke the LRU policy differently from exokernel OSes, it will increase the migration cost of the LRU policy between them.

The cost of implementing cross-language invocation is expensive. For example, if LibOS is developed using C in an exokernel OS, then a Java application invokes this LibOS, you must package the LibOS into a JNI, this will increases the development cost of LibOS. Similarly, if a Python application calls the LibOS, you need to use cdll in Python. Thus, a library must be implemented using different cross-language protocols. The cost of implementing cross-language invocation will be increased.

Resource management recently is too popular. Software and hardware disaggregation architecture based on resource management frequently appears in papers such as LegoOS [7] and Multikernel [8] at top conferences. Unfortunately, existing system design cannot modify the resource management rule dynamically in their system. LegoOS proposed a two-layer resource management mechanism to coordinate resource usage of components. Multikernel designed monitors which coordinate system-wide state and encapsulate much of the mechanism and policy that would be found in the kernel of a traditional OS to manage resources in OSes. While both rules have to be redeployed when they are modified due to harding code instead of extension protocol. Thus, it will limit the reusability and portability of these good management rules.

New optimizations for resource operations appear in industry and academia every year. Recently, researchers proposed XRP [30] and Memliner [31]. The former used user-defined BPF functions in the NVMe driver to reduce Linux kernel storage stack overhead. The latter used “lining up” memory accesses to improve the performance of far-memory systems. However, both of them have to modify

the existing drivers or the virtual machine. Thus, as well as resource management, it also will limit the reusability and portability of these optimization rules because of the intrusive approach.

Therefore, we propose a concept of resource governance, designs a resource governance model, and places applications, hardware, and rules into governance. By exposing standard rules in the model, it reduces the development cost of invocation between multi-language applications, and through the high-level abstraction of the model, it improves the portability of resources among different kernel architectures.

Using the model, we built Rhodes, which consists of agents and a governance center. Through binding agents with resources, Rhodes decouples the communication among resources such as applications and hardware. It makes developers focus more on resources, not communication protocol. In the governance center, users can modify and extend more rules, and then embed rules into agents. Thus, users can optimize the communication among resources dynamically.

We evaluated Rhodes with microbenchmarks and two unmodified applications, Bzip2 [24] and WhiteDB [25]. Our evaluation results show that compared to Linux, Rhodes is only 1.6× slower with wasm rule scripts. But Rhodes's performance is equivalent to Linux in native mode.

Overall, this work makes the following contributions:

- We propose the concept of resource governance model, a new OS architecture that fits both Stand-alone OSes and Distributed OSes.
- The concept of resource governance model also fits most optimization in App, VM, and OS layers.
- We built Rhodes, the first OS based on the resource governance model.
- We propose a new architecture to cleanly separate resources and management rules to make rules extend conveniently, while preserving the performance of monolithic OSes such as Linux.

The rest of the paper is organized as follows: In Section 2, we discuss and analyze current situations of different kernel architectures, and then propose motivation and challenges of resource governance. We introduce the core concepts of resource governance model in Section 3. Section 4 describes the design of resource governance model. Section 5 shares core implementation details of resource governance model. In Sections 6, we evaluates the performance of Rhodes. Section 7 shares the relationship between resource governance model and current kernel architecture OSes/distributed OSes. Finally, Section 8 concludes the paper.

II Motivation for Resource Governance

This section motivates the resource governance model and discusses the challenges in managing resource organization in OSes.

A Monolithic Kernel OS

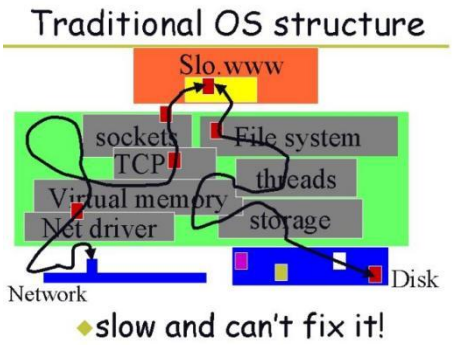

Figure 1: Monolithic Kernel OS Architecture

Figure 1 shows the architecture of a Unix system [1]. This is an example of opening a web page. When we open the web page linked to Slo.www, the kernel in a client needs to go through socket ->

TCP -> VM -> Net driver -> NIC to complete the network request, and then in a server, go through File System -> threads scheduling -> VM -> storage -> Disk to complete the reading of web page data. In the two paths, the coupling among the modules such as VM in the kernel is very tight and modules cannot be customized.

B Microkernel OS

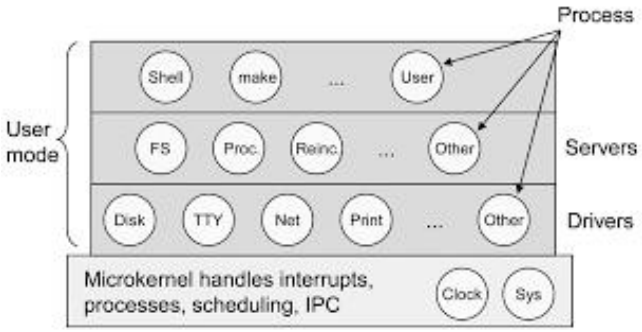

Figure 2: Microkernel OS Architecture

Figure 2 shows a microkernel OS architecture [2]. Only interrupts, scheduling, IPC, etc. are reserved in the kernel mode. In the user mode, from bottom to top, the first layer is Disk, Net driver, etc, and the second layer is FS, Proc (Process Management), and other services, the third layer is Shell, make, and other applications. The modules of each layer appear in the form of processes. It seems that the architecture is structured and the modules are divided. However, the user-mode program is customized by the user, so how can we ensure that a large number of developers have strong architectural capabilities and do a good hierarchy design? So one module may be coupled with another.

C Hybrid Kernel OS

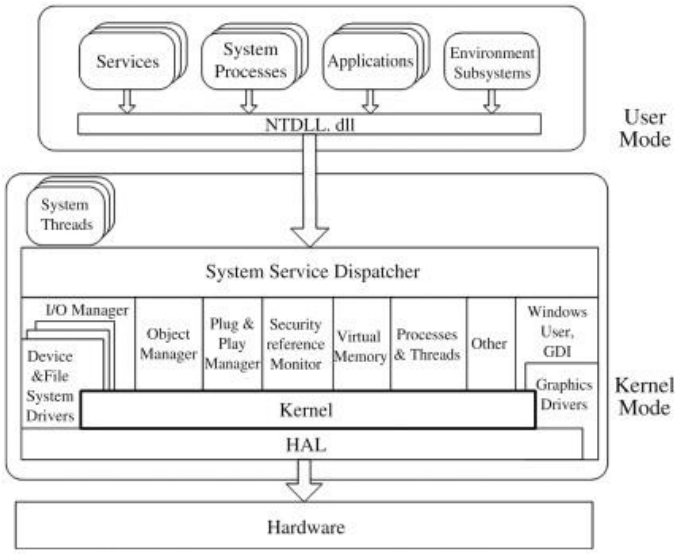

Figure 3: Hybrid Kernel OS Architecture

Figure 3 shows the architecture of WindowsNT [3], which is a hybrid kernel OS. We find that in its kernel space, there are not only basic kernel functions, such as virtual memory, graphics driver, file driver, etc. but also customized functions, such as object management, plug-in management, etc. Because of placing plug-in management in the kernel mode, the complexity of plug-in management is shielded for users. Thus, it makes WindowsNT better application scalability by a unified plug-in extension standard. In addition, it also retains the most basic functions of the monolithic kernel to ensure the application processing performance. However, Microsoft can place plug-in management

into the kernel, then other os manufacturers can also place plug-in management into user mode. So, what modules each os manufacturer wants to place is their own decision, and it will eventually bring security risks for key functions such as plug-in management.

D Exokernel OS

In 1995, Dawson Engler, a professor at MIT, proposed exokernels [4] to solve the problem of application performance optimization in specific scenarios.

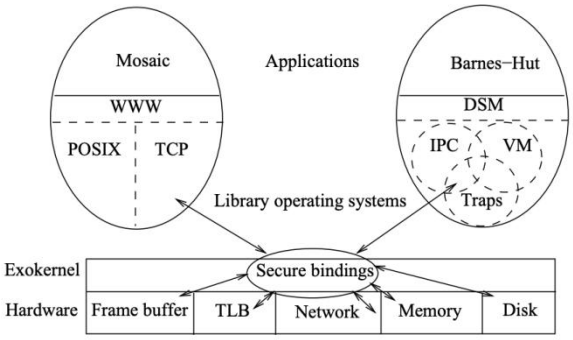

Figure 4: Exokernel OS Architecture

Figure 4 shows the architecture of an exokernel. We find that exokernels place IPC, virtual memory management, traps, POSIX interface, and TCP into user mode. It defines them as applications, which can directly operate the hardware after they are bound with the hardware by LibOS. Because LibOS is developed by users, exokernels also have the same problem encountered by microkernel operating systems: How to control the high-quality architectural design of LibOS?

E Problems of Current Kernel Architecture

Whether it is a monolithic kernel, microkernel, hybrid kernel, or exokernel, they all have similar problems:

Code organization is uncontrollable. Monolithic kernel OSes place TCP, IO, driver, interrupt, scheduling, etc. in the kernel. In hybrid kernel OSes, users can arbitrarily put their modules into the kernel. The former couples non-operating system core functions such as TCP and IO with operating system core functions such as interrupt and scheduling, making it difficult for users to customize non-core functions. The latter allows OS developers to customize kernel functions, resulting in the emergence of various operating systems in the market and increasing application developer learning costs. Microkernel and exokernel OSes allow users to customize common functions such as TCP and IO, so it is difficult to ensure that such functions developed by users are reusable. Thus, all these kernel-based operating systems will bring the risk of uncontrollable code development.

The cost of cross-language application development is high. If Java calls a C library, you need to use JNI to implement the C library. If Python calls a C library, you need to use cdll in Python. Thus, a library must be implemented using different cross-language protocols. It will increase the cost of cross-language application development.

F OSes for Resource Management

Many teaching and research institutions are trying different resource management solutions. Among them, the most closely related one is LegoOS [7] designed by Purdue University. It divides the operating system into three types of components: pComponent, mComponent, and sComponent, and designs a two-layer resource management mechanism to coordinate resource usage of components. Its goal is to build a flexible and scalable distributed hardware management model. It looks like a good resource management solution, but unfortunately, LegoOS is designed for data center scenarios, so it exists two problems:

Hardware expansion is limited. LegoOS only abstracts three types of components, which manage CPU, memory, and storage. However, a complete operating system not only includes these resources but also input and output devices such as keyboard and network card. The communication between

network card and CPU is different from the communication between memory and CPU, so LegoOS model abstraction will limit the scalability of hardware.

It is difficult to dynamically change the code. The two-layer resource management mechanism in LegoOS is implemented by hard coding. If the resource scheduling algorithm is optimized, the code must be changed and redeployed.

So, to solve the above problems, we have made a higher-level abstraction of operating systems and placed software and hardware into unified governance to maximize the flexibility and scalability of software and hardware organizations. The goal of the abstraction is not only to manage resources but also to modify and extend management rules dynamically. To achieve this goal, we will face the following challenges :

- How to organize software and hardware in an operating system to make them run in order?
- How to build a more flexible and scalable management of software and hardware in the operating system?
- How to ensure more efficient communication between software and software, and between software and hardware in the operating system?
- How to support existing applications in traditional OSes?

III OSes for Resource Governance

In 1992, Steven McCanne and Van Jacobson proposed BPF [32] which is an interface that allows users to offload a simple function to be executed by the kernel. Afterwards, Linux presents eBPF (extended BPF) [33] based on BPF, which is commonly used for filtering packets (e.g., TCPdump), load balancing and packet forwarding. The appearance of BPF/eBPF exactly makes us use different languages to customize the optimizations in OSes. It also allows OSes to discover the change for optimization dynamically. But BPF/eBPF still has weak portability between various applications. For example, we have implemented LRU policy (BPF funciton) for page replacement. If we hope to migrate the policy to the local cache management developed in Java, we must set the probe in the local cache management program and modify LRU policy according to eBPF specification. Thus, it will limit the portability of excellent concepts and technologies.

By the high-level abstraction of operating systems, the author proposes a resource governance model, which places both operating system applications and hardware resources into a governance model. Through unified governance, we maximize the flexibility and scalability of resource organization and the performance of task processing. The current governance model is divided into four parts:

Governance Center. The governance center is a system that is separated from operating system resources. It is responsible for managing and maintaining general rules that application and hardware resources used. It can be deployed in a node together with application and hardware resources. It also can be deployed in a node separated from application and hardware. Through the automatic registration and discovery mechanism, the governance center can push rules to resources, and then related resources can run in order according to the established rules. In addition, we can customize rules through the separation of rules and resources. It ensures the scalability and flexibility of rules in the governance center.

Agent Process. By introducing the agent process and binding it with the resource process, the communication between resources is transformed into inter-agent communication, which makes users can use different programming languages to develop applications or hardware that communicate with each other with the same cross-language protocols instead of different protocols such as JNI, cdll, etc.

Static rule discovery. The model defines a rule which controls the order of the invocation among resources. Because programs only can use this rule and cannot change it, we define this kind of rule as a static rule. By the publish-subscribe model, an agent can listen to the changes of static rules stored in the governance center. When the rules are changed, the governance center will push the changes to the agent in real time, and then the agent will update its rules in the local cache after receiving the changed rules. This design ensures the efficiency of processing static rules by agents bound with applications or hardware running locally.

Dynamic rule discovery. The model allows users to customize the communication strategy

between resources. For example, users can implement a shm policy. An application can pass its parameters to the policy, then the policy will be run using the parameters. Because the policy need application parameters to run, we define this kind of rule as a dynamic rule. Through Wasm technology, the model allows dynamic rules in the governance center to be embedded into agents and run them in the agents locally. For those that are not suitable for remote communication such as IPC, thread scheduling, interrupt, etc., the performance of rule execution is greatly guaranteed.

Ⅳ Rhodes Design

Traditional operating systems provide many core functions, such as thread scheduling, IPC, interrupts, and virtual memory management. *Rhodes* makes a high-level abstraction of them based on the resource governance model. This section first describes the process of model abstraction based on these core functions and then describes the design of the model in detail.

A Model Abstraction

The goal of resource governance model is to design a resource-organized, scalable, and high-performance operating system. Focusing on this goal, this part combines the following core functions to explain why Rhodes can realize the orderly and extensibility of resource organization and the high performance of resource operation.

Hierarchical Management. Although we recommend that users follow the hierarchical rule when developing applications or hardware, it is difficult for us to ensure that all developers follow it to develop applications or hardware. So the governance center maintains the hierarchical rule. When an application is registered to OS, the governance center will check whether the application invocation meets the hierarchical rule. Only applications that meet the hierarchical rule can be registered successfully.

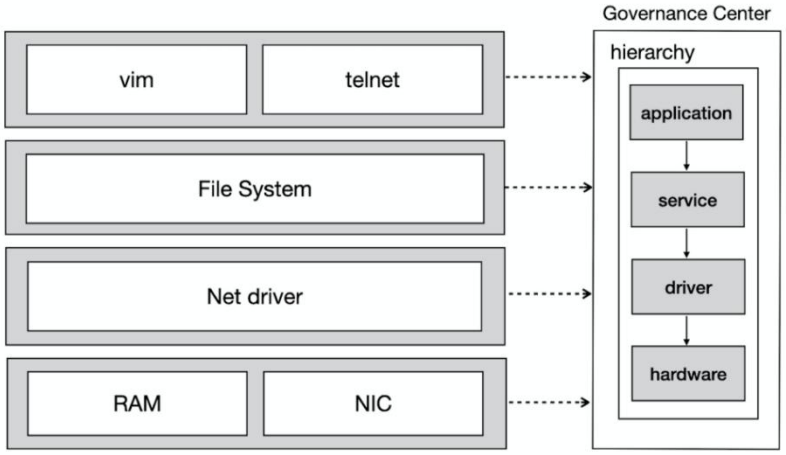

Figure 5: Hierarchical management

In Figure 5, the governance center maintains the relationship among layers: application -> service -> driver -> hardware, we define this rule as a hierarchical rule.

Before an application or hardware is registered to the governance center, the governance center checks its reference relationship according to the hierarchical rule. If it meets the hierarchical rule, registration is allowed; otherwise, not allowed. For example, in Figure 5 a Net driver is registered into the governance center, and the governance center will use the hierarchical rule to check its reference relationship. If it calls NIC, it meets the hierarchical rule and allows registration because the type of Net driver is the driver and the type of NIC is hardware. If Net driver calls File System, due to File System being a service, so the reference of Net driver doesn't meet the hierarchy rule and it is not allowed to be registered.

The governance center uses the hierarchical rule to check whether the reference relationship between applications and hardware is reasonable, ensuring orderly invocation among user-defined

applications and hardware.

Cross-Language Invocation. If there is a Java application that wants to directly call a network card driver to send data packets, but the network card driver is implemented in C language, and this driver is not JNI, then, is there any way to make this application directly call the network card driver to send data packets?

Rhodes designed an agent process that shields the difference invocations between applications in different languages. It transforms the invocation between applications/hardware into an invocation between agents. Let's take a look at the calling process:

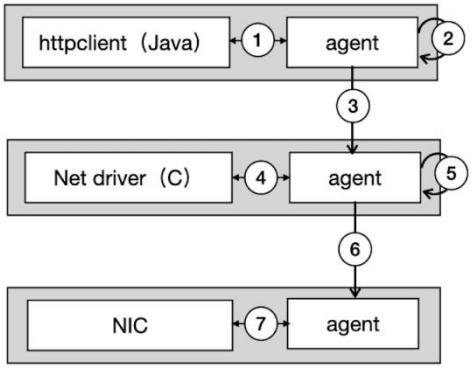

Figure 6: Cross-language invocation

Figure 6 shows the process of sending requests by httpclient library. Httpclient is implemented in Java, and Net driver is implemented in C. The process is as follows:

① Httpclient passes the calling function and function input parameters to its agent process.

② The agent process converts the function and input parameters into a message queue protocol: *msgqueue function name, the type of input parameter 1 | input parameter 1, the type of input parameter 2 | input parameter 2, return type*. In Figure 6, httpclient calls a function in Net driver to send data packets, then the agent will convert the function into a message queue protocol.

③ The agent process of httpclient executes the protocol to send data packets to the agent process of Net driver.

④ The agent process of Net driver passes the protocol and data packets to Net driver to execute sending packet logic locally. When the logic call NIC, Net driver passes the calling function and input parameters to its agent process.

⑤ The agent process converts function and input parameters into message queue protocol.

⑥ The agent process of Net driver executes the protocol to send data packets to the agent of NIC.

⑦ The agent process of NIC passes packets to NIC to perform packet transmission.

If the invocation between httpclient and Net driver is changed from message queue to IPC-based shared memory, how does the agent of httpclient library perceive it?

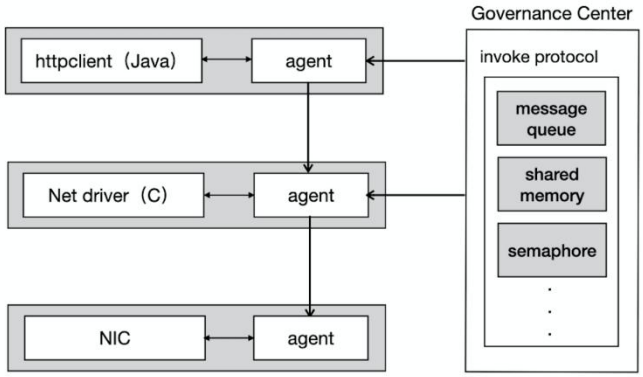

Figure 7: Protocol Discovery

In Figure 7, the invocation protocols in the governance center include message queue, shared memory, and semaphore, etc. For example, in the above example, the invocation between httpclient and Net driver uses message queue to send data packets. When the protocol is now changed to shared memory, the governance center will push shared memory protocol to the agent of httpclient and Net driver to operate shared memory. In addition, new invocation protocols can also be extended in the governance center, such as rpc and rest, etc.

Thread Scheduling. If application scale is not large, rules can be deployed in a node with applications, otherwise, rules must be deployed to other nodes separated from applications. For example, we can deploy a thread scheduling policy in a node separated from applications. But if we do this, frequent thread scheduling will drive much remote communication between the governance center and applications, which will inevitably lead to a huge overhead of performance. So Rhodes introduces an agent process and embeds such rules into the agent that is deployed in a node with applications. Thus, it will transform remote scheduling into local scheduling. The performance of thread scheduling will be improved.

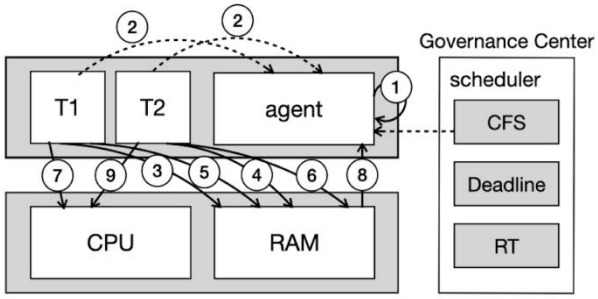

Figure 8: Thread scheduling

Figure 8 shows the process that schedules two concurrent threads using CFS. The governance center first embedded the thread scheduler script into the agent, and then the agent process starts to schedule T1 and T2:

① The agent starts to schedule periodically.
② T1 and T2 are handed over to the agent when they are started.
③ The agent allocates stack and writes TCB in memory for T1.
④ The agent allocates stack and writes TCB in memory for T2.
⑤ The agent updates the virtual runtime of T1 and adds T1 to a runnable queue implemented by a red-black tree in memory.
⑥ The agent updates the virtual runtime of T2 and adds T2 to the runnable queue.
⑦ The agent finds that CPU is idle at this time, and hands the T1 to CPU for execution because the virtual runtime of T1 is minimum.
⑧ After T1 is executed, the agent frees the stack and TCB of T1, then takes T2 from the runnable queue in the next tick.

⑨ The agent hands the T2 to CPU for execution.

Interrupt. Interrupts are the most frequently executed function in an OS. Considering that the governance center can be deployed separated from applications, to ensure the performance of interrupt processing, Rhodes embeds the soft interrupt processing script from the governance center into the agent process. Because the agent process and the application are deployed on the same node, the soft interrupt scheduling is performed by the agent locally. Thus, it will improve the performance of interrupt schedulers.

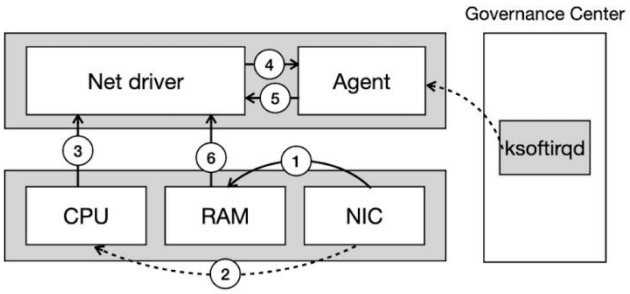

Figure 9: Softirq processing

In Figure 9, this is an interrupt process of a network card receiving data packets after the soft interrupt processing script is embedded into the agent process. The process is as follows:

① NIC writes packets to ring buffer in memory via DMA.
② NIC notifies CPU, initiates a hard interrupt, and notifies CPU that a network packet arrives.
③ CPU call back Net drvier.
④ Net driver triggers NAPI and set sofitirq status to pending.
⑤ The softirqd thread in the agent is woken up when CFS schedule at the next tick, and then call back Net driver to perform packet pulling.
⑥ Net driver pulls packets from ring buffer in memory.

B Resource Governance Model

Through the abstraction of hierarchical management, cross-language invocation, thread scheduling, and interrupt processing, we obtain a general resource governance model.

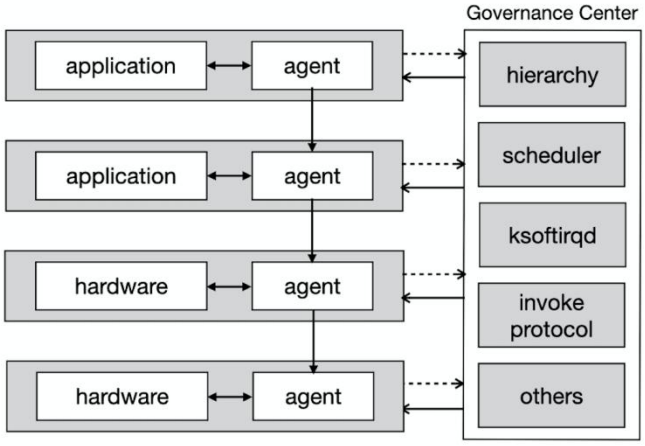

Figure 10: Resource governance model

Figure 10, this is a resource governance model:

① The model contains 4 elements: application (including application, service, and driver), agent, hardware, and governance center.
② Governance center maintains the rules that applications and hardware use. These rules

contain hierarchical rules, scheduler, soft interrupt, cross-language invocation protocols, etc.

③ Registration and Discovery: applications and hardware register themselves into the governance center. The governance center can push rules to the agents bound with applications and hardware.

C Static Rule Discovery

To allow static rules to be pushed to the agent process in real-time, Rhodes implements the publish-subscribe model between the agent and governance center.

The agent implements CWatchManager internally, which is used to save and manage watch events inside the agent. The governance center implements SWatchManager to save and manage watch events within the governance center. When the OS starts, the agent will run the subscription process, and when the rules are changed, the governance center will run the publishing process. Taking the change hierarchical rule as an example, let's take a look at the process of watch event subscription and publishing:

Subscription process

① The agent embeds the namespace of hierarchical rule, agent-side address, and namespace listener into watch events.

② The agent sends a watch event registration request to the governance center.

③ After the governance center receives the watch event, it saves the event to SWatchManager.

④ The governance center sends a registration success response to the agent.

⑤ After the agent receives the response, it saves the watch event to CWatchManager.

Publishing process

① The user changes the hierarchy rule in the hierarchical namespace through the governance center.

② The governance center uses SWatchManager to find the watch event associated with the hierarchy namespace.

③ SWatchManager pushes the changed hierarchical rule to the associated agent in the watch event.

④ After the agent receives the changed hierarchical rules, it uses CWatchManager to refresh the hierarchical rules in the local cache.

At present, Rhodes has provided two ways for the communication between the agent and the governance center: shm and PubSub. The former is used in the scenario where the agent and the governance center are deployed on the same node, and the latter is used in the scenario where the agent and the governance center are deployed separately.

D Dynamic Rule Discovery

Since dynamic rules such as cross-language invocation protocol, and thread scheduling are pushed into the agent by the governance center, it is particularly important to let the agent discover the changes in these rules in time. Therefore, we introduce WebAssembly technology to achieve the dynamic discovery of the rules.

WebAssembly (Wasm) [9] is a portable instruction format written in multiple languages that executes at near-native speed. Although WebAssembly was originally born as a client-side technology, the Wasm community is standardizing the "WebAssembly System Interface Interface" (WASI) [10] in the W3C, which provides an OS-like abstraction for Wasm programs, so we can use different languages to implement Wasm extensions, and then embed the compiled extensions into the agent through WASI.

Therefore, the agent we developed supports WASI using Wasmer [11] with LLVM [12], and through WASI, rules developed in different languages can be embedded into the agent in the form of scripts and then executed.

However, at present, Wasm programs only can be run in user space. So we decide to use eBPF to run dynamic rules in the kernel. Developers can implement dynamic rules in C/Rust, compile them into bytecode and attach them to tracepoints, kprobes/uprobes, and perf events.

eBPF makes rules run faster by reducing the logic execution path, while Wasm provides the possibility to implement complex rules. So Rhodes supports both of them to allow developers can

select a suitable mode (eBPF/Wasm) to implement dynamic rules. But at present, only Linux, Windows, MacOS support eBPF, so, in the future, we will optimize eBPF to allow developers to customize both kernel events not dependent on OSes and event orchestration rules.

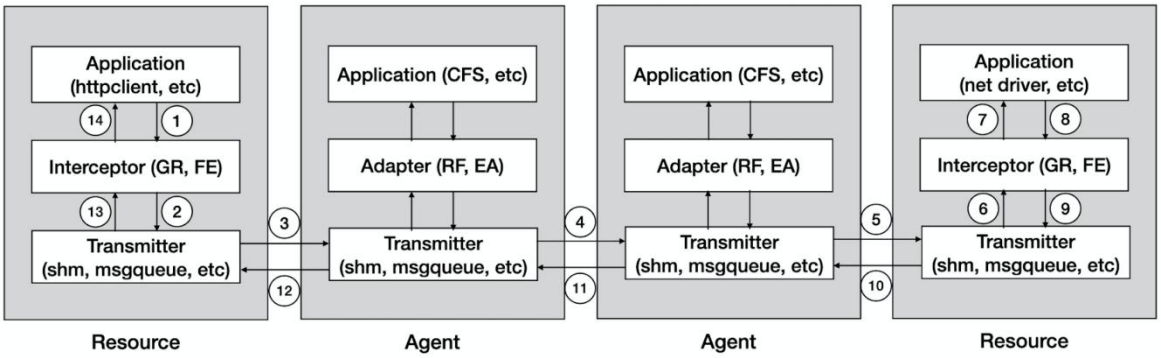

Figure 11: The Communication among Resources and Agents based on 3-layer model for Dynamic Rules

At present, Rhodes has designed a 3-layer model to implement the communictaion among resources and agents (Figure 11). This model contains four components: application, interceptor, adapter and transmitter.

Application. The business logic is implemented in the application. For example, the application in httpclient implements request body converting and request sending, etc. The application in the agent implements the dynamic rule logic.

Interceptor. The interceptor includes generic reference (GR) and function extractor (FE). The former calls the application by generalization using the function name and call params. The latter extracts the callee name, call function name and call params from the invocation handler.

Adapter. The adapter includes rule finder (RF) and run-mode adapter (EA). The former gets the dynamic rule from the dynamic rule configuration cached locally. The latter calls the application to run the dynamic rule with the mode from the dynamic rule configuration cached locally.

Transmitter. The transmitter implements the data transmission among resources and agents. It uses shm to implement the communication between resources and agents. Furthermore, it uses invocation protocol among applications configured in the governance center to implement the communication between agents. The invocation protocol includes eBPF, IPC and RPC, etc.

We take request sending as an example. We assume that the invocation protocol between httpclient and net driver is configured as msgqueue in the governance center and the protocol has been cached locally. First, httpclient calls the interceptor passing the invocation handler (① in Figure 11). Then, the interceptor extracts the callee name, call function and params from the invocation handler and calls the transmitter passing the data extracted before (②). Next, the transmitter creates a shared memory named “httpclient-shm” according to the process name and writes the extracted data into the shared memory (③).

In the httpclient agent, the transmitter reads the call function and params from the shared memory named “httpclient-shm” in the local configuration according to the “httpclient” param that is passed into the agent when booting the agent (③). After the transmitter gets the msgqueue protocol from the dynamic rule configuration cached locally according to the “httpclient” param and the callee name from the shared memory, the transmitter combines the “httpclient” param and the callee name, runs the msgqueue protocol to create a queue named “httpclient-netdriver-queue” using the combination and add then the call function and params into the queue (④).

In the net driver agent, first, the transmitter scans the dynamic rule list according to the “net driver” param that is passed into the agent when booting the agent to get the msgqueue protocol, extracts the queue name from the msgqueue protocol like "httpclient-netdriver-queue" and monitors the “httpclient-netdriver-queue”. Then, the transmitter reads the call function and params from the

queue. Then, the transmitter creates the shared memory named “netdriver-shm” according to the “netdriver” param that is passed into the agent when booting the agent and writes the call function and params into the shared memory (⑤).

In the net driver, the transmitter reads the call function and params from the shared memory named “netdriver-shm” according to the process name. Then, the transmitter passes the call function and params to the interceptor (⑥). Next, the interceptor generically calls the application using the function and params (⑦). Afterwards, the application runs the function and passes the result to the interceptor (⑧). The 3-layer model uses a similar mechanism in transmitters mentioned above to send the result across the transmitter in net driver, the transmitter in the net driver agent, the transmitter in the httpclient agent and the transmitter in httpclient (⑨, ⑩, ⑪ , ⑫). At last, the transmitter in httpclient uses a similar way in net driver to pass the result to the application (⑬ , ⑭).

For the dynamic rule that works in a single application instead of cross-applications, the adapter will call the application to run the rule. Rhodes provides native and wasm modes to run the rule. The native mode allows rules to be embedded into the agent by eBPF. The wasm mode uses WASI [10] to embed rules into the agent. For example, the adapter can receive a key like "cfs" from httpclient by shm and then get the CFS rule from the dynamic rule configuration cached locally according to the application name bound with the agent and the key. At last, the adapter calls the application to run CFS using params from httpclient in the mode configured in the CFS rule.

Thus, a new dynamic rule can take effect immediately without modifying applications and agents.

E Agent Process

First of all, in order to implement the publish-subscribe mode, the agent process implements the consumer of publish-subscribe mode.

Second, in the interrupt processing scenario, the communication between agent processes and applications is frequent. Thus, the probability of concurrent operations on the same memory area between the application and the associated agent is very high. So we decided to use shm to implement the communication between the application and the associated agent to ensure the efficiency of the communication.

F Dynamic Rules

There are many core functions in OSes. But we only implemented two functions in Rhodes: CFS and Tasklet based on “as needed” referring to Linux. Both of them can be embedded in agents in native and wasm modes.

CFS scheduler. We implemented CFS dynamic rule in Rhodes. We use a red-black tree to store the virtual runtime of threads. To avoid lock contention, we also use atomic primitive in the critical section to implement context switches between threads.

Tasklets. Softirqs are statically-registrable, and you must protect shared data with spinlock since the same softirq will run simultaneously on more than one CPU. Tasklets are dynamically-registrable, and they also guarantee that any tasklet will only run on one CPU at any time, although different tasklets can run simultaneously. So we stripped tasklets from Linux as a wasm script and embedded it into the agent to make drivers extend callback functions registered into tasklet conveniently.

G Supporting Unmodified Applications Migration

We have organized functions that need to intercept. At the same time, we also developed a hook to modify different language codes. For example, the hook can scan bytecodes and intercept function calls between applications by bytecode enhancements such as ASM [13], Javassist [14], and ByteBuddy [15] for Java applications. Afterwards, the hook extracts call functions and parameters, then passes them to agents by shm. For non-virtual-machine languages such as Clang and Rust, the hook can do this by modifying compilers. Thus, all existing applications can be migrated from existing OSes such as Linux, MacOS, and Windows to Rhodes without modification.

Ⅴ Rhodes Implementation

A Resource Location

In order to accurately push the hierarchical rules in the governance center to an agent of application or hardware, Rhodes designed resource protocols, resource collection, and invocation matrix.

Resource Protocol. Rhodes defines a protocol for the application and hardware resources registered to the governance center. The protocol contains 4 attributes: type , reference, privilege and agent. The details are as follows:

type: The type of registered resource, divided into application, service, driver and hardware. For example, vim's *"type = application"*.

reference: Who was called by the registered resource. For example, Net drvier call NIC to complete the packet transmission, then Net driver need to pass *"reference = nic"* when registering the governance center.

privilege: The privilege mode of the registered resource. It contains kernel mode and user mode.

agent: The agent bound with application or hardware. It includes pid and name of the agent.

Resource Collection. Rhodes categorizes each resource registered to the governance center, dividing them into 4 sets :

Application Set: denoted as $\boldsymbol{A}=\{x \mid type\ (x)=application\}$, indicating that the type of all elements x in set is application. For example, two applications *vim* and *telnet* are registered to the governance center, then they are in set A.

Service Set: denoted as $\boldsymbol{S}=\{x \mid type\ (x)=service\}$, which represents the resource set of all elements x whose type is service. For example, *File System* service is registered to the governance center, then File *System* is in the set S.

Driver Set: denoted as $\boldsymbol{D}=\{x \mid type\ (x)=driver\}$, which represents the resource set of all elements x whose type is driver. For example, Net driver is registered to the governance center, then Net driver is in set D.

Hardware Set: denoted as $\boldsymbol{H}=\{x \mid type\ (x)=hardware\}$, which represents a resource set of all elements x whose type is hardware. For example, RAM and NIC are registered into the governance center, then RAM and NIC are in set H.

Invocation Matrix. In order to accurately push the rules in the governance center to an agent of application or hardware, Rhodes designs an invocation matrix and save the matrix in the governance center. It defines the matrix as follows:

① A union of above four sets is denoted as $\boldsymbol{O} = A \cup S \cup D \cup H$.

② Suppose $\boldsymbol{R} \subseteq O \times O$.

The following matrix is obtained:

$$M_R=(r_{ij})_{n\times n}$$
$$r_{ij}=1,\ o_i \mathrm{R}\ o_j$$
$$r_{ij}=0,\ o_i \not\mathrm{R}\ o_j$$

Suppose we define 4 collections as follows:

$$A = \{vim, telnet\}$$
$$S = \{file\ system\}$$
$$D = \{net\ driver\}$$
$$H = \{ram\ ,\ nic\}$$

Then, we obtain the union $\boldsymbol{O} = \{vim, telnet, file\ system, net\ drvier, ram, nic\}$, so vim call File System can be expressed as a matrix:

$$M_R = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Before the resource is registered to the governance center, the governance center compares the type of the caller resource with the type of the reference resource to check if the hierarchical rule is met, and decides whether to create an element in the matrix. For example, vim calls File System, the type of vim is application, and the type of File System is service, which meets the hierarchical rule, and then the corresponding element in the matrix $\boldsymbol{M_R}$ is set to 1, otherwise, the corresponding element is set to 0.

With the resource collection and invocation matrix, we can push the hierarchical rule to the application or hardware accurately.

Taking the above matrix $\boldsymbol{M_R}$ as an example, suppose we configure the invocation protocol between vim and File System as msgqueue in the governance center, then the governance center will find the element $\boldsymbol{r_{02}}$ in $\boldsymbol{M_R}$, and push the protocol to $\boldsymbol{vim(O_0)}$. But the protocol is directly pushed to the agent of vim according to the agent attribute of vim. After the agent of vim receives the protocol, it can call File System to perform file operations by msgqueue.

B Wasm

In a distributed environment, dynamic rules can be deployed separately from the agent. If Wasm technology is used, these rule scripts will be allowed to be embedded in the agent in instruction format to execute locally in a distributed environment. It ensures the efficiency of rule execution. At present, the agent in Rhodes already supports WASI [10] using Wasmer [11]. Through WASI, the agent can call rule scripts embedded in it directly and run them locally. However, unfortunately, the programming language SDK provided by the Wasm community does not include Verilog which is a hardware development language. Therefore, in the future, we will implement Verilog [16] SDK including the wasm compiler and hardware agent to support communication between hardware agents.

In the deployment of rule scripts, Poseidon has implemented the publishing of rules in CLI mode. The command will automatically upload the script to a node that agents are deployed on by executing the command.

C IPC

In a highly concurrent scenario, it produces frequent communication between applications and agents, so we decide to use shm to implement communication between them in Rhodes to ensure communication efficiency.

Considering that agent process is implemented in Rust, we found and investigated ipcd [18], which is an IPC component developed by the team who developed Redox [19]. Because ipcd is developed in Rust and is very light, we choose it to implement communication between applications and agents. In addition, we also use this component to ensure communication efficiency between the governance center and the agent deployed on the same node.

To avoid locking shared memory in a concurrent scenario, we optimize shm, which allocates an independent memory for each thread. Thus, it makes shared memory for each thread small enough to greatly improve communication efficiency between applications and agents.

D PubSub

If the governance center and the agent are deployed separately, we use publish-subscribe model to push static and dynamic rule configurations to agents in Rhodes. Although the frequency of rule changes is too low, considering that the agent process will collect statistics information of invocation between applications in the future, in order to ensure the performance of the collection, we implement publish-subscribe model ourselves and use a queue to support the communication between subscriber and publisher, based on it, we can achieve high-performance remote communication between the governance center and the agent process. In addition, by customizing the pubsub model, the agent size

becomes very small, so that one node can deploy many agents.

E Rule Governance

Rhodes uses Rocket [20] to implement backend services for rule governance in the governance center. At present, users can only call API to add/change rules based on restful protocol.

We can maintain static rule configurations through backend services. For example, if you'd like to change the hierarchical rule, you need to call the API to update the rule configuration in the governance center.

We also can maintain dynamic rule configurations through backend services. For example, if you'd like to change thread scheduling for sending request from CFS to Deadline, you need to push the Deadline policy script into the agent bound with httpclient via CLI mentioned above, and then call the API to update the policy configuration in the governance center.

Static and dynamic rule configurations are stored in MySQL to make the query of rule configurations convenient. Rhodes uses Diesel [21] which is an ORM framework to implement the data operation for database interaction.

F Experience and Discussion

For non-virtual-machine languages, we started to modify their compilers to intercept function calls in applications to send the invocation protocol to agents. However, some languages such as Golang don't support plugin mode at compile time. We spent our engineering efforts on an "as needed" base and decided to implement interceptors for languages such as Clang and Rust which provide extension at compile time by plugins.

Interceptor for GCC. The callback in the interceptor is invoked after finishing parsing a function in GCC by monitoring PLUGIN_FINISH_PARSE_FUNCTION event [22]. Then, the callback parses the event-specific AST to extract the function call entry and passes the call function to the agent by shm.

Interceptor for RustC. The late lint pass in the interceptor is invoked at a late stage of compilation in rustc by lint plugins [23]. Then, the late lint pass parses the AST to extract the function call entry and passes the call function to the agent by shm.

We have implemented the interceptor in Golang compiler by harding code. We also will strip the interceptor from the compiler and discuss how to extend features at compile time together with language-related communities such as Golang in the future.

Considering large and frequent syscalls in existing applications in Linux, we have to implement syscalls using the same language as Rhodes to reduce cross-language invocation overhead. When we started to do it, we found there are so many ABIs in Linux. Fortunately, Rust supports glibc, so we can intercept Linux ABIs in an application and then forward ABIs to the agent. At last, the agent finds Rust std functions through Linux ABI and Rust libstd [24, 25] mappings and runs them. All of these can be implemented by modifying compilers. Thus, it makes us run application performance benchmarks without modifying existing applications.

VI Evaluation

This section presents the performance evaluation of Rhodes using micro-benchmarks and two unmodified applications. We ran all experiments on a machine, with four Intel Xeon CPU E5-1620 3.5GHz processors and 32GB DDR4. Considering that enabling JIT compilation with LLVM can make the performance of wasm application execution close to native mode by machine code cache, all wasm modules in Rhodes such as thread scheduling enable JIT based on LLVM.

A Micro-benchmark Results

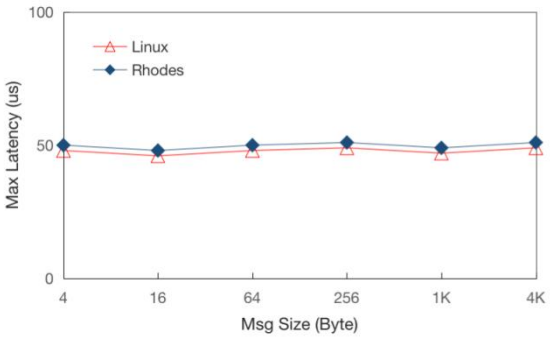

Figure 12: Max Latency with shm

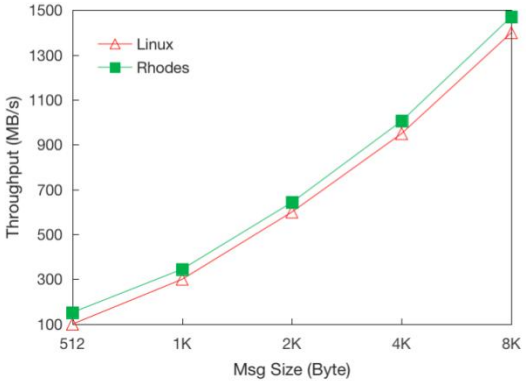

Figure 13: Throughput with shm

Shared memory performance. Figure 12 shows the max latency of IPC in shm between Rhodes and Linux. We found that when the message size is less than 4k and the cache misses, the maximum latency of shm in Rhodes is about 50us, which is close to Linux.

Figure 13 presents the throughput of IPC using shm between Rhodes and Linux. We use atomic operations for reference-counting pointer to make shared object thread-safe in Rhodes and Linux. When the message size is larger, the message throughput of shm increases exponentially, and the throughput in Rhodes is better than in Linux because of avoiding the high overhead of atomic operations by independent memory for each thread in Rhodes. Because each thread in applications and agents uses small enough shared memory independently, we believe that the performance impact of shm communication between applications and agents is negligible.

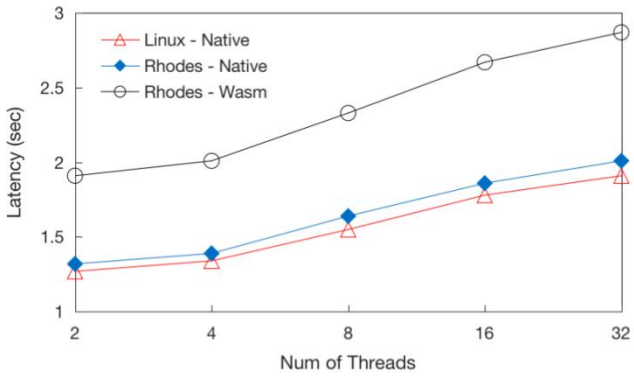

Figure 14: CFS Latency

Thread scheduling performance. Figure 14 compares Rhodes' CFS performance with Linux. We vary the number of concurrent threads from 2 to 32 to show the latency of the same task in Rhodes and Linux. In this experiment, Rhodes' CFS performs as well as Linux equivalent before the number of threads reaches four in native mode. Afterwards, the latency of Rhodes' CFS increases by 1.2× compared to Linux. This is because Rust enforces checking borrowing rules at runtime with mutable type [26, 27] in spinlock when context switch occurs in the critical section. The latency of Rhodes' CFS in Wasm mode is 1.5× compared to Linux. When calling functions, Wasmer runtime checks that the target is a valid function and that the function's type is the same as the type specified at the call site. So we attributed the worse performance to the frequent function checks when calling functions.

Thus, for common and frequently executed rules such as thread scheduling, we recommend implementing a native library in unsafe languages such as Clang to embed it into the agent by eBPF, avoiding large performance penalty. For the rules of asynchronous scenarios such as Softirqs mentioned above, we can use a language different from OS, compile the rules into Wasm files, and embed them into the agent for execution.

B Application Performance

We evaluated Rhodes' performance with two unmodified applications, Bzip2 [28] and WhiteDB [29]. Bzip2 is a popular data compression tool that achieves a very high compression ratio. We use it to compress a 120MB data file. WhiteDB is a NoSQL database library written in C, operating fully in main memory. We use it to conduct a set of database operations. Bzip2 is a typical CPU-intensive application, and WhiteDB is an application that largely uses memory. So the experiment of two applications can cover most scenarios in Rhodes except IO-blocking. We ran both applications with eight threads.

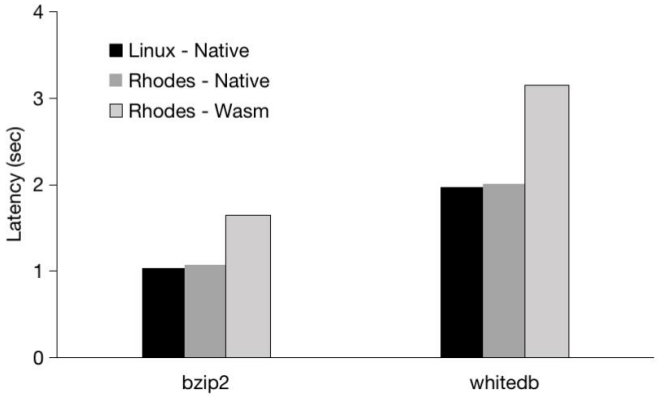

Figure 15: Execution time for Bzip2 and WhiteDB

Impact of Wasm mode on application performance. Figure 15 presents that the execution time of Bzip2 and WhiteDB in Rhodes is 1.6× compared to Linux in Wasm mode. But the applications in Rhode perform as well as Linux equivalent in native mode. We attributed the worse performance to frequent function checks in CFS in wasm mode and frequent borrowing rules checking in Rust.

VII Related Work

Due to the high-level abstraction of the resource governance model, it can easily evolve into different kernel architectures.

A Monolithic OS Evolution

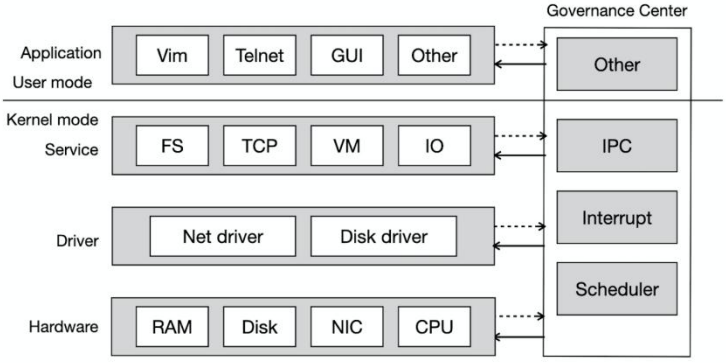

Figure 16: Monolithic Kernel Architecture based on Governance Model

Figure 16 is a monolithic kernel OS architecture based on the evolution of the resource governance model. The upper part is user mode and the lower part is kernel mode.

User mode includes applications such as Vim, Telnet, and GUI. Kernel mode includes services such as FS, and TCP, drivers such as Net driver, hardware such as RAM, and some rules such as IPC, Interrupt, Scheduler, etc in the governance center.

B Microkernel OS Evolution

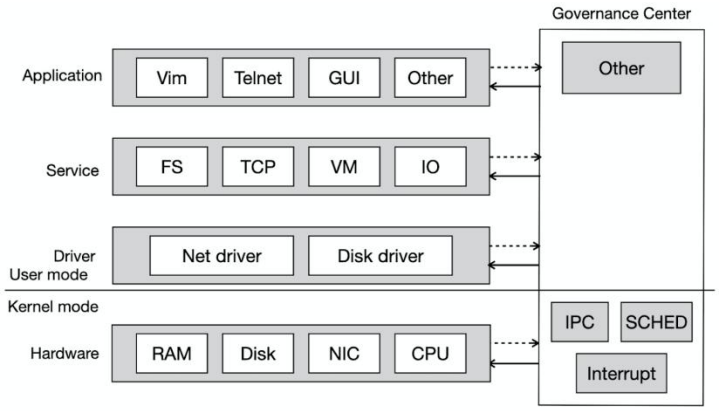

Figure 17: Microkernel Architecture based on Governance Model

Figure 17 is a microkernel OS architecture based on the evolution of the resource governance model. The microkernel OS is similar to the monolithic kernel OS architecture. The only difference is that user mode and kernel mode are divided differently. In the microkernel OS, services and drivers are divided into user mode, and kernel mode only retains hardware, as well as IPC, Scheduler (SCHED), and Interrupt in the governance center.

C Exokernel OS Evolution

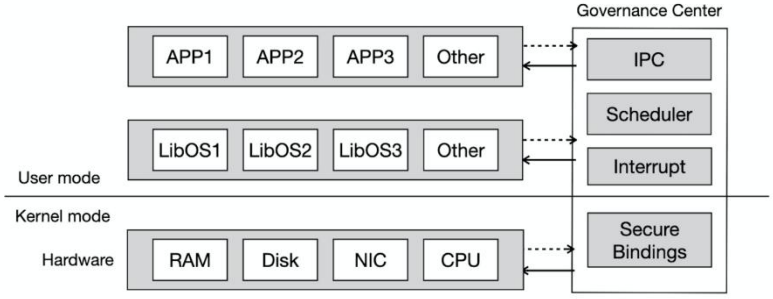

Figure 18: Exokernel Architecture based on Governance Model

Figure 18 is an exokernel OS architecture based on the evolution of the resource governance model. The exokernel OS is also divided into two parts: user mode and kernel mode. User mode includes applications and LibOS libraries, as well as IPC, Scheduler, and Interrupt rules, the kernel mode only contains security binding rules, applications are securely bound with hardware through LibOS, and after binding, applications can directly operate trusted hardware.

Since the resource governance model can easily evolve into various kernel architectures, it will bring great convenience to migrate different kernel OSes to Rhodes based on the governance model, and it also provides the possibility for the rapid migration of applications and rules between different kernel OSes because of a unified model.

D Resource Governance and Cloud OSes

Similarly, based on the resource governance model, we can deploy different applications and hardware on different nodes to implement a manageable cloud OS.

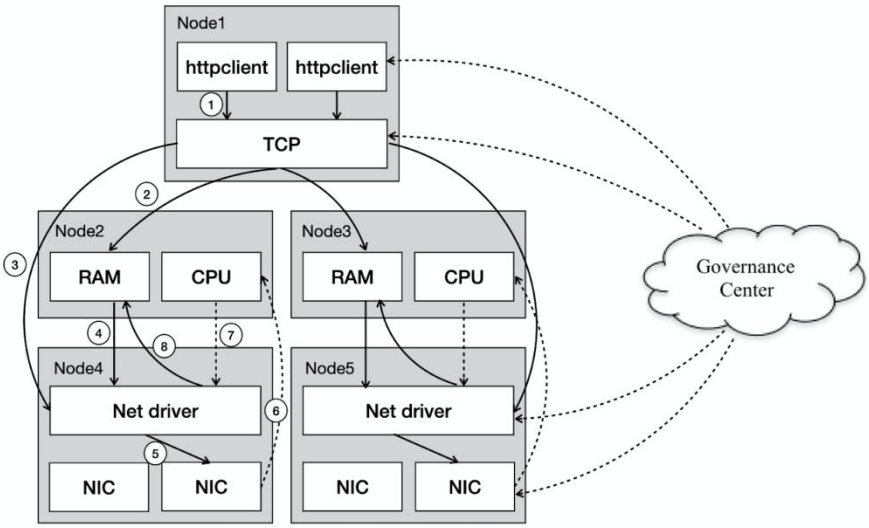

Figure 19: Packet sending in a distributed architecture

Figure 19 is a packet-sending process in a distributed architecture. Applications and hardware are deployed on different nodes. Httpclient and TCP are deployed on Node 1, RAM and CPU are deployed on Node 2 and Node 3, Net driver and NIC are deployed on Node 4 and Node 5. There are two processes for httpclient to send request in the figure. Let's take the process for sending request on the left as an example to see the process:

① Httpclient calls TCP protocol to send request.
② TCP protocol copies the data packet of the request to the skb in the memory.
③ TCP calls Net driver through the network device subsystem.
④ Net driver selects available ring buffer elements from memory and associates them with skb.
⑤ Net driver calls NIC to send packets.
⑥ After NIC completes sending the packet, a hard interrupt is triggered to notify CPU.
⑦ CPU triggers a soft interrupt again to callback Net driver.
⑧ Net driver releases the skb in memory and cleans up the ring buffer.

In the process, the rules of communication among applications/hardware and interrupt rules are pushed by the governance center cluster.

In the past, hardware such as CPU and NIC became the performance bottleneck in a single machine. But when this hardware is deployed in different nodes, it will greatly improve the throughput of network data transmission by horizontal scaling. In addition, through the separate deployment of applications and hardware, compared with the traditional whole-os deployment of distributed nodes, it avoids the low utilization of single-machine resources such as CPU and memory.

E Resource Governance and Customized Optimization

In 2022, in order to reduce Linux kernel storage stack overhead with NVMe storage devices, Yuhong Zhong, etc. propose XRP [30] that allows applications to execute user-defined storage functions from an eBPF hook in the NVMe driver, safely bypassing most of the kernel's storage stack. Based on the resource governance model, we can place BPF functions into the governance center in Rhodes by separating rules and resources. Another example is MemLiner [31]. MemLiner is a runtime technique that improves the performance of far-memory systems by "lining up" memory accesses from the application and the GC. We also can put the GC tracing algorithm into the governance center. So we find the resource governance model can fit most of the optimization. This means most optimization rules in OSes, VirtualMachine and App layers can be migrated into Rhodes rapidly.

All of the above show the resource governance model is universal. Due to the universality of the resource governance model, it makes users can build different kinds of OSes by building blocks like "Lego" based on a unified model to reduce the development cost of a new system except for the

portability mentioned above.

VIII Discussion and Conclusion

We presented Rhodes, the first OS based on resource governance model. Rhodes has implemented registration and discovery of applications and hardware, the local and remote discovery of static rules, and the local discovery of dynamic rules. But the remote discovery of dynamic rules and the hardware agent mechanism have not been implemented yet.

Availability and data consistency of the governance center cluster will be also implemented in the future. Therefore, we need to continuously improve Rhodes to make it available for production.

We also find that OSes based on the resource governance model are better than the centralized kernel OSes(including monolithic kernel, microkernel, hybrid kernel, and exokernel)., which have the following advantages:

- Through the unified management of resources, the scalability of different resources is improved, so that computer hardware is no longer limited to CPU, memory, hard disk, network card, etc.
- By separating rules from resources, the scalability of rules is improved. Rules among resources in OSes are no longer limited to traditional rules such as IPC and CFS, etc., and users can customize more rules.
- By hierarchical rule checking, the quality of resource module development is guaranteed.
- Through the cross-language invocation mechanism, the cost of multi-language application development is reduced.
- The governance model is too universal through high-level abstraction, which fits both stand-alone OSes and distributed OSes, as well as customized optimization.

Security, high performance and scalability have eventually become important standards for measuring the quality of an operating system. The resource governance model redefines the operating system architecture and more reasonably combines security, high performance and scalability. Therefore, we have reason to believe that OSes based on resource governance model will become a new development trend for operating system, replacing OSes based on resource management. It will become the next generation operating system.

References

[1] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr, "Exokernels (or, making the operating system just another application library)," March 1998. [Online]. Available: https://pdos.csail.mit.edu/archive/exo/exo-slides/sld003.htm

[2] Andrew Tanenbaum, Raja Appuswamy, Herbert Bos, Lorenzo, Cristiano Giuffrida, Tomáˇs Hrub´y, Jorrit Herder, Erik van der Kouwe, and David van Moolenbroek, "MINIX 3: status report and current research." in ;login: The USENIX Magazine, JUNE 2010, vol. 35, no. 3, pp. 7-13.

[3] Hongbiao Jiang, Wanlin Gao, Manwei Wang, Simon See, Ying Yang, Wei Liu, Jin Wang, "Research of an architecture of operating system kernel based on modularity concept." in Mathematical and Computer Modelling, JUNE 2010, vol. 51, issues. 11-12, pp. 1421-1427, doi: 10.1016/j.mcm.2009.10.006.

[4] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr, "Exokernel: An Operating System Architecture for Application-Level Resource Management." in Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95), 1995, vol. 29, no. 5, pp. 251-266, doi: 10.1145/224056.224076.

[5] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter, "The performance of μ-kernel-based systems." in Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP '97), 1997, vol. 31, no.5, pp. 66-77, doi: 10.1145/269005.266660

[6] P. Cao, EW Felten, and K. Li, “Implementation and performance of application-controlled file caching.” in Proceedings of the First Symposium on Operating Systems Design and Implementation, 1994, pp. 165-178.
[7] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang, “LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation.” in Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18), 2018, pp. 69-87.
[8] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania, “The Multikernel: A New OS Architecture for Scalable Multicore Systems” in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09), 2009, pp. 29-44, doi: 10.1145/1629575.1629579
[9] W3C Community Group, WebAssembly, 2022. [Online]. Available: https://webassembly.org/
[10] WebAssembly Community Group, “Standardizing WASI: A system interface to run WebAssembly outside the web,” 2019. [Online]. Available: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/
[11] Wasmer Inc, wasmer-wasi, 2022. [Online]. Available: https://crates.io/crates/wasmer-wasi
[12] Wasmer Inc, wasmer-compiler-llvm, 2022. [Online]. Available: https://crates.io/crates/wasmer-compiler-llvm
[13] Eric Bruneton, ASM, 2022. [Online]. Available: https://asm.ow2.io/index.html
[14] Shigeru Chiba, Javassist, 2022. [Online]. Available: https://www.javassist.org/
[15] Rafael Winterhalter, Oslo, Norway, Byte Buddy, 2022. [Online]. Available: https://bytebuddy.net
[16] Verilog.com, Verilog, 2022. [Online]. Available: https://www.verilog.com/
[17] dtolnay, syn, 2022. [Online]. Available: https://github.com/dtolnay/syn
[18] Redox OS community, ipcd, 2022. [Online]. Available: https://gitlab.redox-os.org/redox-os/ipcd
[19] Redox OS community, Redox, 2022. [Online]. Available: https://www.redox-os.org/
[20] SergioBenitez, Rocket, 2022. [Online]. Available: https://rocket.rs/
[21] The Diesel Core Team, Diesel, 2022. [Online]. Available: https://diesel.rs/
[22] GCC team, GCC plugins, 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html#Plugin-API
[23] Mozilla, Lint plugins, 2022. [Online]. Available: https://doc.rust-lang.org/beta/unstable-book/language-features/plugin.html
[24] The Rust Team, “Increasing the glibc and Linux kernel requirements,” 2022. [Online]. Available: https://blog.rust-lang.org/2022/08/01/Increasing-glibc-kernel-requirements.html
[25] The Rust Team, The Rust Standard Library, 2022. [Online]. Available: https://doc.rust-lang.org/std/index.html
[26] Mozilla, Interior Mutability (Rust), 2022. [Online]. Available: https://doc.rust-lang.org/reference/interior-mutability.html
[27] Mozilla, Interior Mutability Pattern (Rust), 2022. [Online]. Available: https://doc.rust-lang.org/book/ch15-05-interior-mutability.html
[28] jseward, bzip2, 2022. [Online]. Available: https://sourceware.org/bzip2/
[29] WhiteDB team, WhiteDB, 2022. [Online]. Available: http://whitedb.org/
[30] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman and Asaf Cidon, “XRP: In-Kernel Storage Functions with eBPF.” in Proceedings of the 16th USENIX conference on Operating Systems Design and Implementation (OSDI’22), 2022, pp. 375-393.
[31] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu and Guoqing Harry Xu, “MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime.” in Proceedings of the 16th USENIX conference on Operating Systems Design and Implementation (OSDI’22), 2022, pp. 35-53.

[32] McCanne Steven and Jacobson Van, “The BSD Packet Filter: A New Architecture for User-Level Packet Capture” in Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93), 1993, pp. 2, doi: 10.5555/1267303.1267305

[33] Linux, A thorough introduction to eBPF, 2014. [Online]. Available: https://lwn.net/Articles/740157/

[34] Heyang Zhou, “Running WebAssembly on the Kernel”, 2019. [Online]. Available: https://medium.com/wasmer/running-webassembly-on-the-kernel-8e04761f1d8e

[35] Wasmer Inc, kernel-wasm, 2020, [Online]. Available: https://github.com/wasmerio/kernel-wasm