Next Generation Operating System based on Resource Governance Model

Abstract

The kernel is the core part of the operating system. At present, in operating system domain, the kernel is divided into four categories: monolithic kernel, micro kernel, hybrid kernel and exokernel, each of which has its own advantages: the monolithic kernel reduces the switching between user mode and kernel mode by abstracting fixed modules such as virtual memory, IPC, hardware driver, and IO model in the kernel to ensure the performance of task processing. The microkernel only retains core system abstractions such as IPC and interrupt in the kernel to realize the high scalability of modules such as hardware driver and IO model[3]. The hybrid kernel combines the former two[4]. The exokernel allow user to implement the hardware operation rules (IO, IPC, interrupt processing, etc.) through user-mode LibOS to customize performance optimization of the application[5]. These kernel architectures are either too heavy, like the monolithic kernel, put too much modules into the kernel, or too light, like the exokernel, it's too flexible, or have no rules, like the hybrid kernel, the kernel modules organization is disordered. Therefore, in order to solve these problems, the author proposes resource governance model that places resources such as application, hardware, and rules into model governance. Through unified governance, the organization of application, hardware, and rules is more orderly. It also ensures the security of the key resources of the operating system and high performance of task processing.

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

1 Introduction

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

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 wants to write data to a file, it needs to call the Unix kernel function through the POSIX interface. Then, the kernel function synchronously writes the memory page, asynchronously flushes the dirty page to the hard disk. Because only the kernel function can directly to operate the hard disk and the user process can't be access to operate the hard disk, so the key data is protected. At the same time, due to the mechanism of synchronous writing of memory and the asynchronous disk flushing is implemented in the kernel, which reduces the switching between user mode and kernel mode, it guarantees the performance of IO processing.

The microkernel operating system represented by L4[6] minimizes the kernel to only IPC, interrupt, thread scheduling, etc., and improves the communication performance between the user process and the kernel process 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 [5] operating system minimizes the kernel functions to only security bindings, and assigns IPC, thread scheduling, and interrupt to the user to implement, providing 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 replacement strategy of the operating system will have a performance impact on their data access, so P.Cao etc.[7] implemented that the application directly controls the file cache, it reduced the program running time by 45%. At the same time, the application is securely bound to the hardware through LibOS, which prevents the application from accessing untrusted hardware resources.

Every kernel architecture is trying to maximize security, performance and scalability. However, we also found that no matter which kernel architecture is used, the invocation between application and hardware is implemented through hard coding. This action exists the following questions:

The code is tightly coupled and has poor portability. For example, if we want to migrate the LRU memory replacement strategy of the Unix system to exokernel operating system, we must strip the relevant strategy from the Unix kernel and migrate it into the exokernel operating system, or directly re-implement this strategy in exokernel operating system. The migration cost is too high.

The cost of implementing cross-language invocation is expensive. For example, if LibOS is developed using C in exokernel operating system, then a Java application invokes this LibOS, it must package LibOS into a JNI, which increases the development cost of LibOS .

Therefore, the author proposes the concept of resource governance, designs a resource governance model and places application, hardware and rules into governance. Through exposing standard rules in the model, it reduces development cost of invocation between multi-language applications, and through the high abstraction of the model, it improves the portability of resource modules within different kernel architectures. In this paper, Section 1 will discuss and analyze the current situation of different kernel architectures proposing the value and challenges of resource governance. Seciton 2 will introduce the core concepts of resource governance model. Section 3 will describe the design of resource governance model. Section 4 will share the core implementation details of resource governance model. Sections 5 and 6 will share the relationship between the resource governance model and current kernel architecture OS and cloud operating system.

2 The Value of Resource Governance

2.1 Monolithic Kernel OS

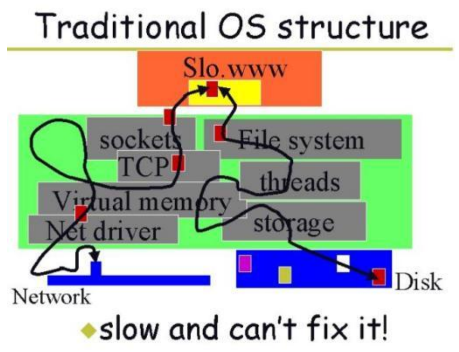

Figure 1: Monolithic Kernel OS Architecture

Figure 1 shows the architecture of the Unix system. This is an example of opening a web page. When we open the web page linked to Slo.www, the kernel needs to go through socket -> TCP -> VM -> Net driver -> NIC to complete the network transmission, and then go through File System -> threads scheduling -> VM -> storage -> Disk to complete the reading of web page data. In the two paths, the coupling between the modules in the kernel is very tight and modules cannot be customized.

2.2 Microkernel OS

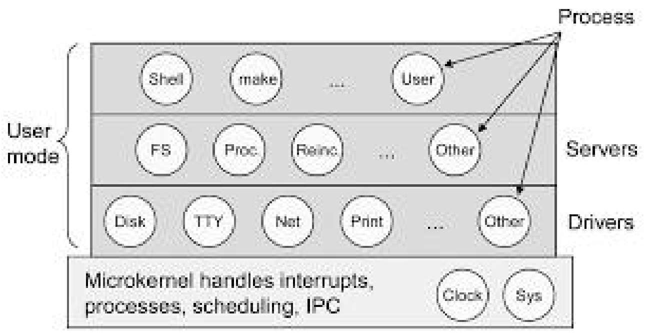

Figure 2: Microkernel OS Architecture

Figure 2 shows the microkernel operating system architecture [3]. 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 clearly structured and the modules are clearly 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 it may appear that one module is coupled to another .

2.3 Hybrid Kernel OS

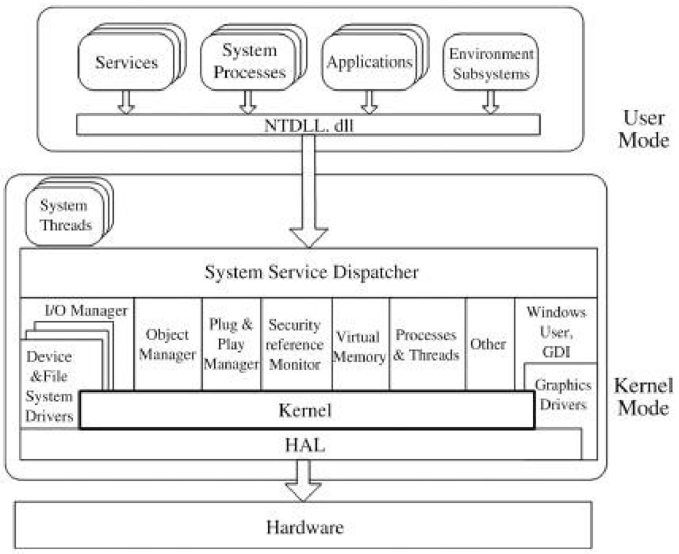

Figure 3: Hybrid Kernel OS Architecture

Figure 3 shows the architecture of WindowsNT [4], which is a hybrid kernel operating system. We found 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 the complexity of plug-in management is shielded for users through placing plug-in management in the kernel mode and plug-in management provides users with a unified plug-in extension standard, so it makes WindowsNT better application scalability. At the same time, it also retains the most basic functions of the monolithic kernel to ensure the application processing performance. However, it also has a similar problem with the microkernel operating system. Microsoft can place plug-in management into the kernel, then other os manufacturers can also place plug-in management into user mode. Therefore, what modules do each os manufacturer want to place is their own decision, which eventually leads to the blossoming of various kernel architectures, which is difficult to reuse.

2.4 Exokernel OS

In 1995, Dawson Engler, a professor at MIT, proposed the concept of exokernel [5] to solve the problem of application performance optimization in specific scenarios .

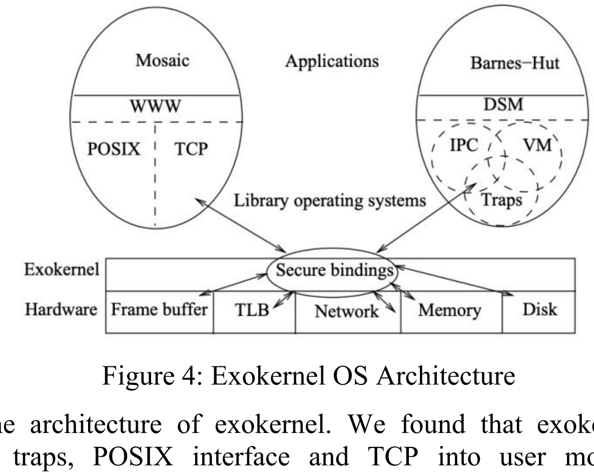

Figure 4: Exokernel OS Architecture

Figure 4 shows the architecture of exokernel. We found that exokernel places 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 to the hardware by LibOS. Since LibOS is developed by users, exokernel also has the same problem encountered by microkernel operating systems: How to control the high-quality architectural design of LibOS ?

2.5 Problem of Current Kernel Architecture

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

Code organization is uncontrollable. The monolithic kernel OS places TCP, IO, driver, interrupt, scheduling, etc. in the kernel. In a hybrid kernel OS , users can arbitrarily put their own 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 user learning costs. Microkernel and exokernel OS allow users to customize common functions such as TCP and IO, which makes it difficult for us to ensure that such functions developed by users are reusable. Therefore, all these kernel-based operating systems carry the risk of uncontrollable code development.

Multi-language application development is not supported. Regardless of the operating system of the kernel architecture, it is currently developed in a single language, which makes it impossible for users to write new applications in the operating system using a programming language different from the operating system.

2.6 The Value of Resource Governance

Resource governance is one of the effective solutions to solve the above two problems.

First, resource governance places various resources of the operating system into governance model, and unifies the organization of various resources by the governance center, such as inter-resource invocation, thread scheduling, etc. Through unified organizational management, the relationship between various resources is more controllable and orderly.

Secondly, the governance center is separated from system resources, so that users can freely change or expand the rules of resource organization through the governance center, which ensure the flexibility and scalability of resource governance.

Finally, in the governance model, by introducing an agent process, resources are bound to agents, then invocation between different resources are transformed into invocation between agents, which decouples the language dependencies between resources and enables users to use different programming language to implement the resource module function.

2.7 Current Governance Situation

Many teaching and research institutions in the industry are trying resource management solutions, such as LegoOS [8] designed by Purdue University, which divides the operating system into three types of components: pComponent, mComponent and sComponent, and coordinates resource usage of components through a two-layer resource management mechanism. Its goal is to build a flexible and scalable distributed hardware management model. It looks like a good governance solution, but unfortunately, LegoOS is designed for data center scenario, so it exists two problems:

- LegoOS only abstracts three types of components, which manage CPU, memory and storage respectively. However, a complete operating system not only includes these resources, but also input and output devices such as keyboard and network card, and the communication between network card and CPU is different from communication betwween memory and CPU, so the limitations of the LegoOS model abstraction will affect scalability for new types of hardware.
- The two-layer resource management mechanism designed in LegoOS is implemented by hard coding. If the resource scheduling algorithm is to be optimized, the mechanism code needs to be changed. Thus, the dynamic replacement of the algorithm cannot be realized.

Therefore, in order to solve the above problems, we have made a higher abstraction of the operating system, and placed software and hardware into unified governance to maximize the flexibility and scalability of software and hardware organizations. In order to achieve this goal, we will face the following challenges :

- How to organize the operating system software and hardware so that it runs in an orderly

manner in the operating system?
- How to ensure more flexible organization 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?

3 Resource Governance Model

Through the high abstraction of the operating system, the author proposes a resource governance model, which places both operating system applications and hardware resources into the governance model. Through unified governance, we maximize the flexibility and scalability of resource organization and the performance of task processing. The current governance model covers four core concepts:

Governance Center. The governance center is a system independent of operating system resources. It is responsible for managing and maintaining general rules which application and hardware resources use. It can be deployed in a node or independently deployed on other nodes with application and hardware. Through the automatic registration and discovery mechanism of the model, the governance center can push the rules to resources, so that related resources can be invoked and run in an orderly manner according to the established rules. At the same time, the separation of the governance center from resources ensures the scalability and flexibility of the rule definition in the governance center.

Agent Process. By introducing the agent process and associating the agent process with the resource process, the communication between resources is transformed into the inter-agent communication, which helps the user to use different programming languages to develop two applications or hardwares that communicate with each other .

Static rule discovery. The model stipulates that different types of resources must obey the order in which type A calls type B. When a resource of type B calls a resource of type A, this invocation rule is not allowed. Since the program only uses this rule and cannot change it, we define this type of rule as a static rule. Through the publish-subscribe model, the model allows the agent process associated with the resource to monitor the changes of the static rules stored in the governance center. When the rules change, the governance center will push the changes to the agent process in real time, and the agent process will update the local cache after receiving the changed rules. This design ensures the efficiency of application or hardware processing static rules.

Dynamic rule discovery. The model allows users to customize the communication strategy between resources. For example, users can implement shared memory strategy and apply it to the communication between resources. Since application or hardware will pass its own parameters to the rule policy, then the rule policy will use the parameter to execute the policy, that is, the rule caller will affect the rule policy, so we define this type of rule as a dynamic rule. Through Wasm technology, the model allows dynamic rules in the governance center to be embedded into agents related resources, and executes the rules in the agents locally. For that are not suitable for remote communication such as IPC, thread scheduling, interrupt, etc., the performance of rule execution is greatly guaranteed .

4 Resource Governance Model Design

The resource governance model is a high abstraction of the operating system, while traditional operating systems provide many core functions, such as thread scheduling, IPC, interrupts, and virtual memory management. Therefore, this section first describes the process of model abstraction in combination with some core functions of the operating system, and then describes the design of the core functions of the model in detail.

4.1 Model Abstraction

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

4.1.1 Hierarchical Management

Although we recommend that users follow the hierarchical rule when developing application or hardware, it is difficult for us to ensure that all developers follow the hierarchical rule to develop application or hardware. So the governance center maintains the hierarchical rule. When a user submits an application to the operating system, the governance center will check whether the application invocation meets the hierarchical rules. Only applications that meet the hierarchical rule can be submitted normally.

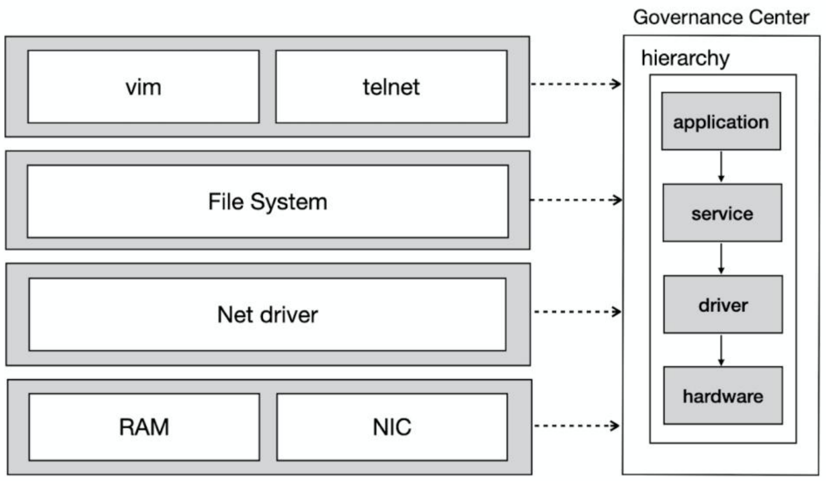

Figure 5: Hierarchical management

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

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

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

4.1.2 Cross-Language Invocation

If there is a Java application that wants to directly call the 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?

4.1.2.1 Agent Process

We designed an agent process, through which communicate with the application, shielding the difference in calls between applications in different languages. Let's take a look at the specific calling process:

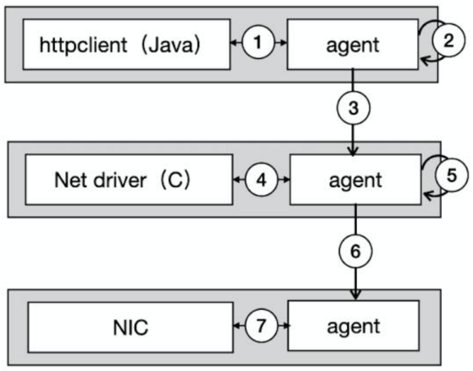

Figure 6: Cross-language invocation

Figure 6 shows the function of sending requests is implemented in the httpclient library, which is implemented in Java, and Net driver is implemented in C, the process as follows:

① The httpclient library passes the calling function and function input parameters to its agent process.

② The agent process converts the function and input parameters into a system call protocol: syscall function name, input parameter 1 type | input parameter 1, input parameter 2 type | input parameter 2, return type. For example, the http client library calls Net driver function netdev_tx_t igb_xmit_frame_ring(struct sk_buff *skb, struct igb_ring *tx_ring) to send data packets, then the agent will convert the function into a system call protocol such as syscall igb_xmit_frame_ring, sk_buff|skb, igb_ring | tx_ring, netdev_tx_t .

③ The agent process of httpclient executes syscall igb_xmit_frame_ring, sk_buff|skb, igb_ring | tx_ring, netdev_tx_t to send the data packets to the agent process of Net driver.

④ The agent process of Net driver passes the data packet to Net driver to execute sending packet logic. 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 system call protocol.

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

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

4.1.2.2 Protocol Discovery

If the way of sending data packets between httpclient and Net driver has changed, from system call to IPC-based message queue call, how does the agent of the httpclient library perceive it?

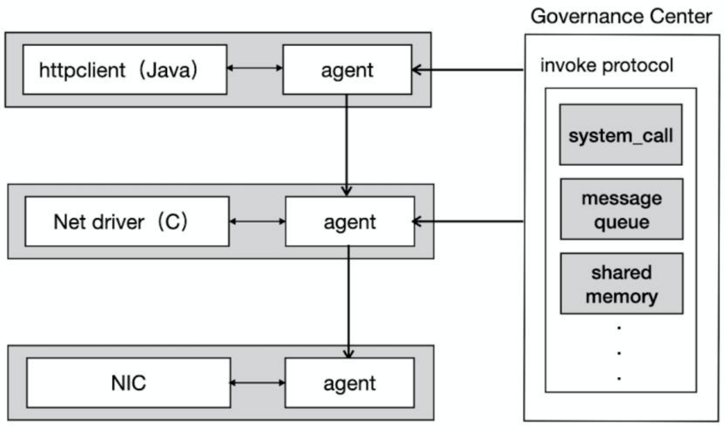

Figure 7: Protocol Discovery

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

By introducing an agent process, which is associated with an application or hardware, it decouples the invocation between applications and hardwares and transforms the invocation between them into invocation between agents, so that users can use different programming languages to develop applications and hardwares that call each other. At the same time, the invocation protocol is separated from applications and hardwares, which ensures the scalability of the protocol.

4.1.3 Thread Scheduling

If the application scale is not large, the governance center can be deployed in a node with the application, otherwise, the governance center must be deployed to other nodes. If the governance center is physically separated from the application, in scenarios such as thread scheduling, since the scheduling rules are in the governance center, frequent thread scheduling will cause frequent remote communication between the governance center and application, which will inevitably lead to huge cost of performance. Therefore, we use the agent process and inject such rules into the process in the node where the application is located, and transform the remote scheduling to local scheduling to ensure the performance of thread scheduling.

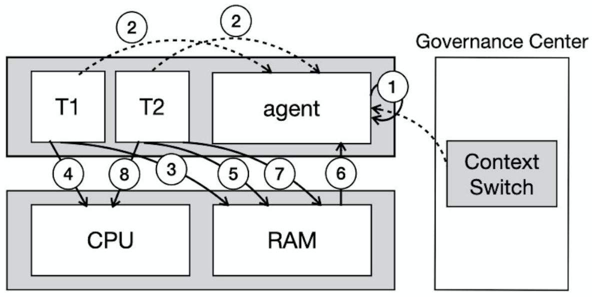

Figure 8: Thread scheduling

Figure 8 shows the process that schedules two concurrent threads using CFS. The governance center first injects the thread context switching rule script into the agent, and then the agent process starts to schedule two concurrent threads 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 applies stack and write TCB for T1 in memory.

④ The agent finds that the CPU is idle at this time, and hands the T1 thread to the CPU for execution.

⑤ The agent finds that the CPU is busy at this time and cannot process the T2 thread, so it writes the T2 thread to the local memory queue and waits for it to wake up.

⑥ The agent cleans stack and TCB of T1, then takes T2 threads from the local queue in next tick.

⑦ The agent applies stack and write TCB for T2 in memory.

⑧ The agent hands the T2 thread to the CPU for execution.

4.1.4 Interrupt

Interrupt is the most frequently executed function in the operating system. Considering that the governance center can be deployed separately from the application, in order to ensure the performance of interrupt processing, we inject the soft interrupt processing script from the governance center into the agent process. Since the agent process and the application are deployed on the same node, therefore, the soft interrupt scheduling is performed by the agent process to ensure the performance of interrupt processing.

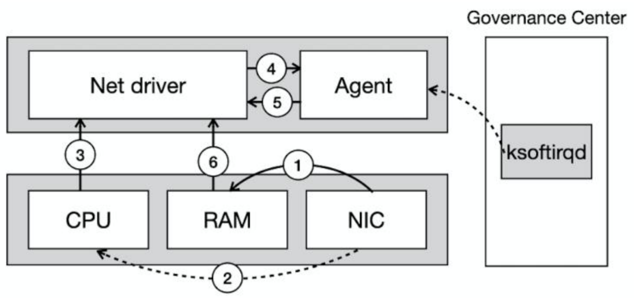

Figure 9: Soft Interrupt processing

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

① NIC writes packets to memory via DMA.

② NIC notifies CPU, initiates a hard interrupt, notifies the CPU that a network packet arrives.

③ CPU callback Net drvier.

④ Net driver to start NAPI and initiate a soft interrupt to the agent process.

⑤ The agent process performs soft interrupt processing and callback Net driver to perform packet pulling.

⑥ Net driver pulls packets from memory.

4.1.5 Resource Governance Model

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

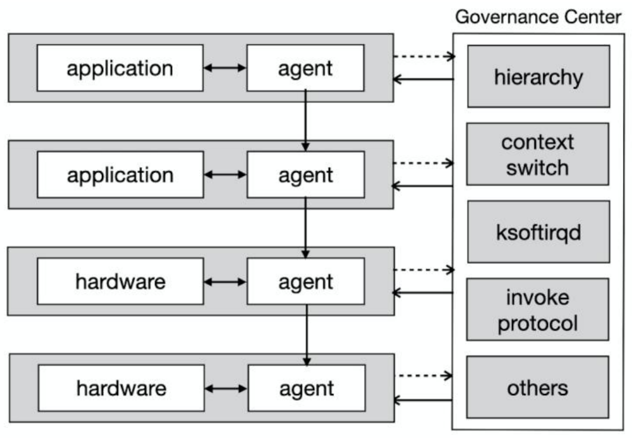

Figure 10: Resource governance model

Figure 10, this is the resource governance model:

① Model contains 4 elements: application (including application, service and driver), agent, hardware and governance center .

② Governance center maintains the rules that application and hardware use: hierarchical rule, context switching, soft interrupts, cross-language invocation protocols, etc.

③ Registration and Discovery: applications and hardware register themselves into governance center, which pushes rules to the agents associated with application and hardware.

4.2 Static Rule Discovery

In order to allow static rules to be pushed to the agent process in real time, we implement the publish-subscribe model between 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 operating system starts, the agent process will initiate the subscription process, and when the rules are changed, the governance center will initiate 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 binds 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 User change the hierarchy rule in hierarchy namespace of the Governance Center.

② The governance center calls the SWatchManager to check whether there is a binding relationship between the hierarchy namespace and its listener in the watch event, and if so, triggers the publishing of the change rule.

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 hierarchy namespace which in the local cache.

At present, we have implemented two ways for the communication between agent process and the governance center: shm and event-driven. 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.

4.3 Dynamic Rule Discovery

Since the governance center pushes dynamic rules such as cross-language invocation protocol, thread scheduling, and soft interrupt processing strategy to the agent, and then the agent executes the relevant logic, it is particularly important to let the agent process dynamically discover the changes of such rules. Therefore, we introduce WebAssembly technology to achieve dynamic discovery of rules.

WebAssembly (Wasm) [9] is a portable bytecode 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) [10,11] in the W3C, which provides an OS-like abstraction for Wasm programs, so we can use different languages to develop Wasm extensions, and then, the compiled extensions are embedded into the agent through WASI.

Therefore, the agent we developed supports WASI, and through WASI, rules developed in different languages can be embedded in the agent in the form of scripts and executed. At present, we have only implemented CLI to publish and embed the rule script into the agent on the node where the agent is located. In the future, we will develop the mechanism that the rule scripts can be published and embedded into the agent remotely through the governance center.

4.4 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 process and application is frequent. But, since each application is associated with an agent, the probability of concurrent operations on the same memory area between the application and the associated agent is very low. So we decided to use shm to implement the communication between the application and the associated agent to ensure the efficiency of the communication. We also have developed SDK for applications. When the application calls other resources, it only needs to use the shm functions in the SDK to pass data to the agent.

5 Resource Governance Model Implementation

5.1 Resource Location

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

5.1.1 Resource Protocol

We define 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 related application or hardware. It includes pid and name of the agent.

5.1.2 Resource Collection

We categorize each resource registered to the governance center, dividing them into 4 sets :

Application Set: denoted as ***A****={x | 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 ***S****={x | 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 the File System is in the set S.

Driver Set: denoted as ***D****={x | 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 ***H****={x | 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.

5.1.3 Invocation Matrix

In order to accurately push the rules in the governance center to related agent of application or hardware, we design invocation matrix and save the matrix in the governance center. We define the matrix as follows:

① 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:

$$M_R = (r_{ij})_{n \times n}$$
$$r_{ij} = 1,\ o_i \mathrm{R}\, o_j$$
$$r_{ij} = 0,\ o_i \cancel{\mathrm{R}}\, o_j$$

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 see if the hierarchical rule are 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 satisfies the hierarchical rule, and the corresponding element in the matrix $\boldsymbol{M_R}$ is set to 1 , otherwise, the corresponding element is set to 0 .

5.1.4 Rule Distribution

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 system_call in the governance center, then the governance center will find the element $\boldsymbol{r_{02}}$ in $\boldsymbol{M_R}$, and push *"O_2 = file system"* corresponding to the subscript 2 of $\boldsymbol{r_{02}}$ to *"O_0 =vim"* corresponding to the subscript 0 of $\boldsymbol{r_{02}}$, that is, pushes the File System to vim. But the File System is directly pushed to the agent of vim instead of being pushed to vim according to the agent attribute of vim. After the agent of vim receives the data, it can syscall the File System to perform file operations.

By checking the invocation before resource registration, the forced layering of resources is realized.

5.2 Wasm

In distributed environment, thread scheduling and interrupt processing strategy can be deployed separately from the agent associated with the application. If Wasm technology is used, these rule scripts will be allowed to be embedded in the agent in bytecode format to excute locally in distributed environment. These rules are executed in this way that ensures the efficiency of rule execution. At present, the agent we have implemented already supports WASI [10, 11]. Through WASI, the agent can call the rule script embedded in it and run the rules locally. However, unfortunately, the programming language SDK provided by the Wasm community does not include Verilog that is hardware development language. Therefore, in the future, we will implement Verilog [12] SDK including wasm compiler and hardware agent to support communication between hardware agents.

In the deployment of rule scripts, we only implement the local publishing of rules in the CLI mode, that is, by executing commands on the node where the agent is located, the rule script is embedded into the agent. In the future, we will also implement remote publishing and embedding of the rule script through the governance center.

5.3 IPC

In the interrupt handling scenario, there is frequent communication between the agent process and the application. However, since each application is associated with an agent, the probability of concurrent operating sharing memory between the application and the associated agent is very low. So we decided to use shm to implement the communication between the application and the associated agent to ensure the efficiency of the communication.

Considering that the agent process is implemented in Rust, we investigated and found the component ipcd [13], which is the IPC component developed by the team who opened Redox [14]. It is a lightweight component. Because it supports the shm mechanism and is very light, we choose this component to implement the communication between application and agent. At present, for the communication between the governance center and agent deployed on the same node, we also use this component to ensure the efficiency of communication.

5.4 Event Driven

We use the publish-subscribe model to push static rule between the governance center and the agent. When the governance center and the agent are deployed separately, static rule push has to be implemented by remote invocation. Although the frequency of rule changes is not high, considering that the agent process will collect statistics information of invocation between applications in the future, in order to ensure the efficient collection of statistics information, we choose tokio (a Event-driven framework) [15], based on it, we can achieve high-performance remote communication between the governance center and the agent process.

6 Kernel Architecture Evolution

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

6.1 Monolithic Architecture Evolution

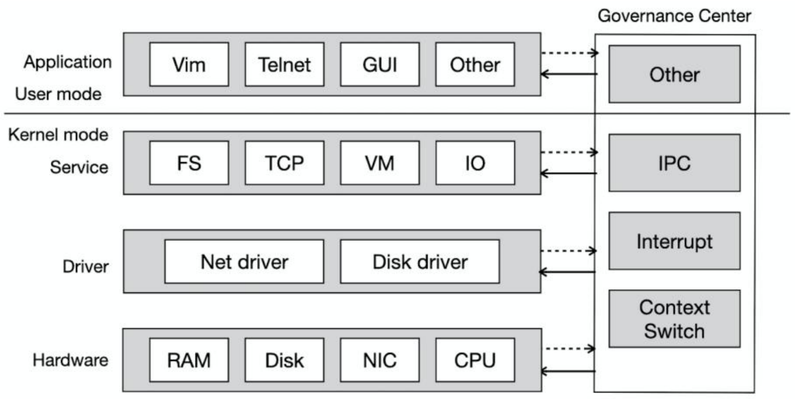

Figure 11: Monolithic Kernel Architecture based on Governance Model

Figure 11, this is the monolithic kernel operating system architecture based on the evolution of resource governance model. The upper part is the user mode and the lower part is kernel mode.

① The user mode includes applications such as Vim, Telnet, and GUI.

② Kernel mode includes services, drivers, hardware and some rules, as follows :
Services: FS (file system), TCP, VM (virtual memory), etc.
Driver: Net drvier, Disk driver etc.
Rules: IPC, Context Switch, Interrupt etc.

6.2 Microkernel Architecture Evolution

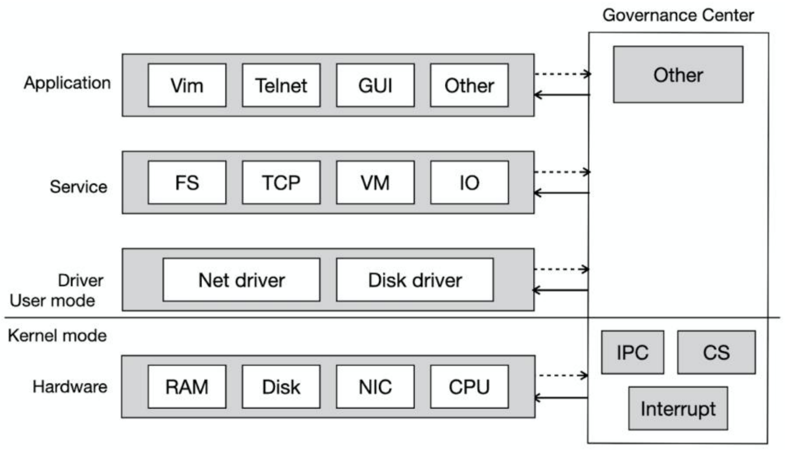

Figure 12: Microkernel Architecture based on Governance Model

Figure 12, this is the microkernel operating system architecture based on the evolution of the resource governance model. The microkernel operating system is similar to the monolithic kernel operating system architecture. The only difference is that the user mode and the kernel mode are divided differently. In microkernel operating system, Services and drivers are divided into user mode, and kernel mode only retains hardware, as well as IPC, context switching (CS), and interrupt processing (Interrupt).

6.3 Exokernel Architecture Evolution

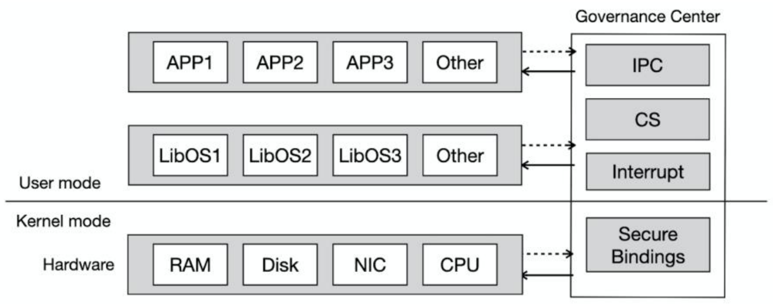

Figure 13: Exokernel Architecture based on Governance Model

Figure 13, this is the exokernel operating system architecture based on the evolution of the resource governance model. Exokernel operating system is also divided into two parts: user mode and kernel mode. User mode includes applications and LibOS libraries, as well as IPC, context switching (CS), Interrupt rules, the kernel mode only contains security binding rules, the application is securely bound to the hardware through LibOS, and after binding, the application can directly operate the hardware.

Since the resource governance model can easily evolve into various kernel architectures, it brings great convenience to the migration of different kernel architectures to the governance model, and it also provides the possibility for the rapid migration of applications and rules between different kernel architectures.

7 Resource Governance and Cloud Operating System

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

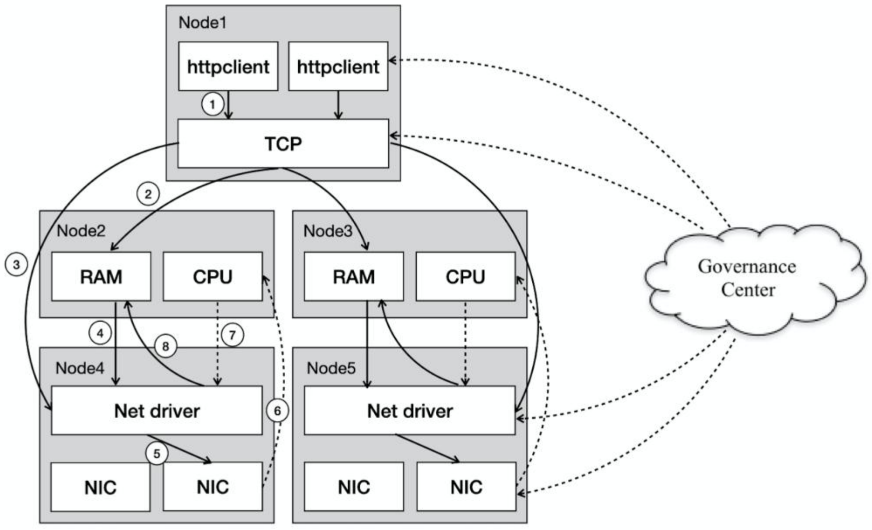

Figure 14: Packet sending in a distributed environment

Figure 14, this is the packet sending process in a distributed environment. Related applications and hardware are deployed on different nodes, http client and TCP are deployed on Node 1, RAM and CPU 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 messages in the figure. Let 's take the process for sending request messages on the left as an example to see the process:

① httpclient calls TCP protocol to send request message.

② The TCP protocol copies the corresponding data packet of the message to the skb in the memory.

③ TCP calls Net driver through the network device subsystem.

④ Net driver selects available RingBuffer elements from memory and associates them with skb.

⑤ Net drvier calls NIC to send packets.

⑥ After NIC complete to send 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 clean up the RingBuffer.

In the process, the rules of communication of applications/hardware within or between nodes is pushed by the governance center cluster.

We found that when we applied the resource governance model to the cloud operating system, the hardware, such as CPU and NIC, which had become the performance bottleneck in a single machine, was deployed in a distributed manner, which greatly improved the throughput of network data processing. At the same time, through separate deployment of applications and hardware, compared with the traditional whole-os deployment of distributed nodes, it avoids the waste of single-machine resources.

8 Discussion and Conclusion

The prototype of the current resource governance model OS implements the registration and discovery of application and hardware, the local and remote discovery of static rule, and the local discovery of dynamic rule. But the remote discovery of dynamic rule and the hardware agent mechanism have not yet been implemented.

Availability and data consistency of governance center after distributed deployment are all work that we need to improve in the future. Therefore, in the future, we need to continuously improve the prototype to make the whole system actually available for production.

In order to avoid the disorder of organization among the resources of the operating system, this paper proposes a resource governance model. We find that the operating system based on the resource governance model is better than the centralized kernel operating system (including monolithic kernel, microkernel, hybrid kernel, and exokernel). , which has 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 . The communication between resources is no longer limited to the two modes: system call and IPC, and users can customize more communication rules.
- By forcing layering, the quality of resource module development is guaranteed .
- Through the cross-language invocation mechanism, the cost of multi-language application development is reduced.
- The high abstraction of the governance model makes the model more universal, which can be used for both stand-alone operating systems and cloud operating systems, showing great flexibility and scalability .

Security, high performance and scalability have gradually become important standard 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, there is reason to believe that the operating system based on resource governance model will become a new trend in the development of operating systems, leading the next generation of operating systems .

References

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

[2] Pablo Pessolani - Gustavo Weisz – Marisa Bardus – César Hein. 2008. Enhancing MINIX 3.X Input/Output Performance. http://sedici.unlp.edu.ar/handle/10915/21574

[3] Andrew Tanenbaum, Raja Appuswamy, Herbert Bos, Lorenzo, Cristiano Giuffrida, Tomáˇs Hub´y, Jorrit Herder , Erik van der Kouwe, and David van Moolenbroek. 2010. MINIX 3: status report and current research. https://www.usenix.org/system/files/login/articles/61781-tanenbaum.pdf

[4] Hongbiao Jiang, Wanlin Gao, Manwei Wang, Simon See, Ying Yang, Wei Liu, Jin Wang. 2010. Research of an architecture of operating system kernel based on modularity concept. https://www.sciencedirect.com/science/article/pii/S0895717709003409

[5] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. https://dl.acm.org/doi/10.1145/224057.224076

[6] Jochen Liedtke. 1995. The Performance of µ-Kernel-Based Systems. https://dl.acm.org/doi/pdf/10.1145/268998.266660

[7] P. Cao, EW Felten, and K. Li. 1994. Implementation and performance of application-controlled file caching. In Proceedings of the First Symposium on Operating Systems Design and Implementation, pages 165–178

[8] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang, Purdue University. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. https://www.usenix.org/system/files/osdi18-shan.pdf

[9] WebAssembly. https://webassembly.org/

[10] WASI. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[11] wasmer-wasi. https://crates.io/crates/wasmer-wasi

[12] Verilog. 1985. https://www.verilog.com/

[13] ipcd. https://gitlab.redox-os.org/redox-os/ipcd

[14] Redox. https://www.redox-os.org/

[15] tokio. https://tokio.rs/