The Current State of Library OSes

2.1 Introduction

Over the past few years, software called library OSes (operating systems) has been attracting a lot of attention. In this report, we examine the significance of this type of software, and discuss the software under development at the IIJ Research Laboratory.

2.1.1 Why Do We Need a New OS Now?

OSes were invented as computers evolved, and their performance and convenience have been improved since then. There are now various types of OS for each usage scenario, and they serve as the heart of all software in computers. So why would we need a new or different OS when the technology has already matured to this level?

The unit of ownership for computer resources has changed from multiple people to individuals. And with the pseudo-computers made possible by virtualization technology today, each service or session can own a computer and OS instead of each user. These virtual computers require an OS with functions that are as slimmed down as possible, rather than a generic OS that has variety of functions. Some anticipated forms of use may include situations where the boot time is required to be as fast as several milliseconds, or where the OS only runs for a few seconds, and is shut down once a specific task is completed.

Additionally, for the programs used in data centers and the high-performance computing (HPC) used in scientific computation, the functions and programs run are limited in scope in the first place, so to achieve specific performance improvements they require an OS dedicated to boosting the efficacy of certain computational processes, instead of a generic OS.

To implement functions like these, the library OSes that were researched and developed in the 1990s are being considered again. Library OSes are based on mechanisms called exokernels, which are distinct from the monolithic kernels adopted by many current OSes.

As shown in Figure 1, by having the kernel that serves as the core part of the OS handle only the minimum necessary functions, and incorporating the majority of functions in the library OS portion. These functions can be added easily, and are reusable over different kernels. By keeping the kernel size to a minimum, exokernels support the design of OSes able to withstand flexible applications.

In the next section, we introduce the features of library OSes that have been proposed around the world as actual examples of software with these kinds of characteristics.

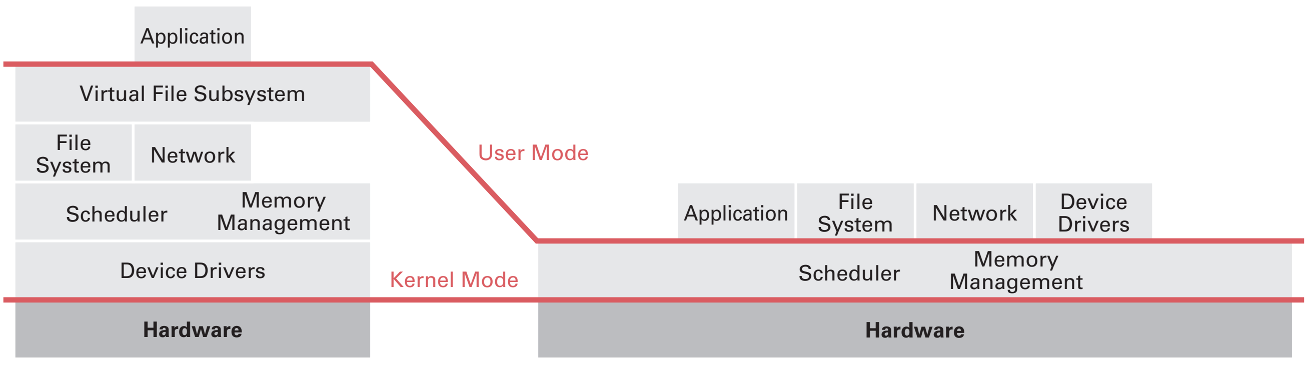

Figure 1: Comparing the Concepts of Monolithic Kernels and Exokernels*1

*1 Source: https://commons.wikimedia.org/wiki/File:OS-structure.svg.

2.2 Related Research Projects

2.2.1 MirageOS

MirageOS, which was presented at USENIX HotCloud in 2010, originally began research and development as an OCaml runtime that runs on a Xen hypervisor without the intervention of a regular OS. This research and development was carried out mainly by researchers at the University of Cambridge in the United Kingdom.

Their greatest contribution was introducing the concept (name) of unikernels for this scaled down software that does not depend on a specific technology such as OCaml or Xen*[2].

2.2.2 DrawBridge

Around the same time, a technology called DrawBridge developed by researchers at Microsoft Research was presented at ACM ASPLOS*[3]. DrawBridge implemented an application-specific kernel by porting over the Windows kernel source code as a library that can be directly used in user space. Their contributions are now used in the Windows virtualization function*[4].

2.2.3 Rump Kernel

The rump kernel was proposed by Antti Kantee and others as an enhancement to the NetBSD kernel. It was designed based on a concept called anykernel. The main characteristic of anykernel is that it enables source code to be reused as a user-space library or a unikernel instead of being just a monolithic kernel.

Research and development for various other library OSes is being carried out, and technologies such as library OSes and unikernels are highly anticipated. In the next section we take a look at a library OS based on the Linux kernel, also known as the Linux Kernel Library (LKL), the development of which I am also involved in.

2.3 The Linux Kernel Library

The Linux Kernel Library (LKL) is an extension of the Linux kernel that began development from around 2007. It implements an abstraction layer atop the original Linux kernel source code that hides low-level resources such as the OS or hypervisor of the operating environment, allowing Linux-based user-space programs to link to and use it, or enabling Windows-based programs to use it as well.

2.3.1 Why Was LKL Developed?

LKL was initially developed to access files in file systems on a disk image created using another OS. Linux uses ext4 as a file system, and people can develop software that understands this file system if we need to access those files on a different OS. Although implementing this software is all that is needed to access files, this is actually quite tricky to accomplish. Because file systems are fairly sophisticated components that involve an enormous amount of source code, the cost of porting is not small. And even if it is ported, the tracking of software enhancements on a daily basis becomes even more challenging.

It is possible to run an OS on another OS through virtualization technology, but this is not efficient because resource utilization for the virtual environment is generally too large. For example, it only takes a few seconds to boot a large number of OSes if the OS size is minimal, while it can take from 10 to 20 seconds per instance when launching a general OS (such as Linux), so it is difficult to start OSes quickly*[5].

Libraries like LKL work well in such cases. Since LKL can use the kernel source code “as-is,” there is no need to worry about the cost of porting. Quick instantiation of OSes is also possible because you can build another OS environment using a single and minimized program, without the unused code path that is required if we use a general purpose OS over virtual machines.

*2 Madhavapeddy et al., “Unikernels: Library operating systems for the cloud.” In Proceedings of ACM ASPLOS 2013, ACM, pp. 461-472.
*3 Porter et al., Rethinking the Library OS from the Top Down. In Proceedings of ACM ASPLOS 2011, ACM, pp. 291-304.
*4 Yuma Kurogome, The Front Line of Microsoft and Library OSes, IIJ Lab Seminar, July 2015 (in Japanese)
*5 Kanatsu et al., Start-Up Time Comparison and Classification for Cloud Environment-Oriented Library OSes, Internet Conference 2016 (in Japanese)

This characteristic is largely due to the benefits of anykernel, which is also implemented in the aforementioned rump kernel. This new form of use, in which only some of the OS functions are utilized, enables the intended functions to be achieved without a wasteful re-implementation process. Generally, improving reusability by increasing the level of abstraction incurs some kind of penalty, leading to issues such as decreased performance or increased complexity, but with LKL these shortcomings have been ignored to give priority to abstraction. In the next section, we examine the kinds of abstraction that have been incorporated into the design.

2.3.2 Design and Implementation

LKL is implemented as a hardware architecture within the Linux kernel source code tree. This architecture was originally introduced to abstract the differences among different processors (CPUs), but LKL, although it is not a type that absorbs CPU differences, uses this mechanism to define the roles of intermediaries so that operating environments can be controlled by external programs. This enables a Linux kernel implemented to operate on certain CPU architecture to be used in user-space programs, as well as programs for other OSes.

Meanwhile, abstraction layers added in this way generally introduce overhead. One example is the packet scheduler, a function that runs on an underlying OS to control the timing of packet transmission. Because this also exists within LKL, it is possible that individual packets will be controlled multiple times. For virtualization technology like LKL, an appropriate design that takes into account such unnecessary duplicated functions is required.

As a basic requirement of this intermediary layer, it is necessary to have the three resources, clock management, program scheduling, and memory management depend on external programs. This enables environment-independent operation using a single piece of software.

The program also performs the relay of input and output (I/O) mechanisms for exchanging data with external resources on this layer, such as accessing data on a disk drive or via a network device. In Linux, a protocol called virtio provides a mechanism for using virtual devices such as these, and LKL also performs I/O operations using (or reusing) this protocol. Reuse not only eliminates the time and effort for an implementation, it also contributes to benefitting various useful functions such as acceleration and availability that utilize virtio "as-is."

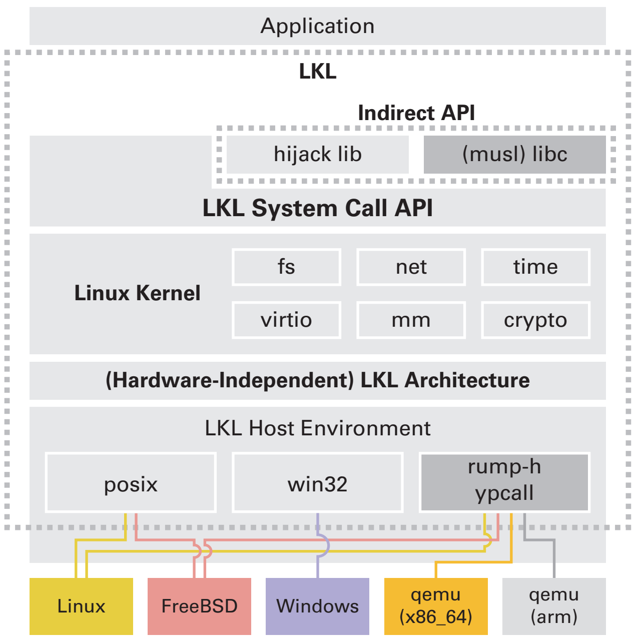

Figure 2: Linux Kernel Library Outlook

LKL is eventually built as libraries that user-space programs can link to and call. It is difficult to use the libraries as they are, so a couple of interfaces are provided as an application programming interface (API). Figure 2 illustrates an overview of the components of LKL. Abstraction layers are added at the top and bottom to enable the Linux kernel source code at the core (and the functions it implements) to be reused across a range of environments and applications. There are currently two available APIs: 1) the system call interface provided by the standard Linux kernel, known as the LKL system call, and 2) the API that uses this indirectly (Figure 2). The indirect APIs include the hijack library that uses LKL system calls by overriding functions when a program is executed, and the standard library implementation, a port of musl libc, that is linked to accordingly when compiling programs. With the latter LKL standard library, it is becoming possible to build unikernel-like applications.

2.3.3 Performance

We attempted to understand the penalties associated with abstraction and virtualization of LKL by measuring packet transmission performance with the traffic generator program called netperf. Although, strictly speaking, this is not a perfect measurement method, it can serve as an index to gain a certain insight into the performance of software.

The measurement was carried out in an environment consisting of two PCs with Intel Core i7-6700 CPUs (8-core, 3.40 GHz), connected via an Intel X540-T2 10 Gbps Ethernet card. For comparison, we used a netperf program that does not use LKL, and runs on the standard Linux kernel (version 4.9.0).

Figure 3 and Figure 4 plot the measurement results for TCP_STREAM and UDP_STREAM results, respectively. The names TCP_STREAM and UDP_STREAM indicate whether netperf uses TCP or UDP. Regarding TCP_STREAM performance, as reported in netdev 1.2*6, the TCP Segmentation Offload (TSO) and checksum calculation processing offload are also implemented in the virtio-net drivers. This means when the size of data to be sent by the application is large (65,535 bytes), and those offload features can be used, then packet transmission processing is carried out efficiently, limiting the drop in throughput to about 10%, and contributing to improved performance. On the other hand, in the current implementation of LKL, this offload processing is not carried out when the packet size is small, so packet transmission processing is performed via software for each packet, which is a cause of slowdown.

Meanwhile, for UDP packets where offloading does not work even in the Linux kernel, LKL still demonstrated up to 47% lower performance than the standard Linux kernel (with a packet size of 1,024 bytes). This result indicates that as additional overhead via virtualization is included when individual packets are sent, there is room for improvement. However, we observed that the transmission performance of LKL and Linux is almost identical when the application specified a large packet size. As with TCP, we believe this is because overhead was reduced by performing bulk packet transmission processing, resulting in improved throughput.

2.4 Conclusion

Although the OS is a mature technology, it is one type of software that will keep evolving, as the number of new uses keeps increasing in response to new requirements. In this report, we discussed the background behind the appearance of these new OS uses, and explained the significance of such software by giving examples of the software under research and development at the IIJ Innovation Institute. We hope this helps people to rethink the role of software and operating systems with an eye toward the future development of the Internet.

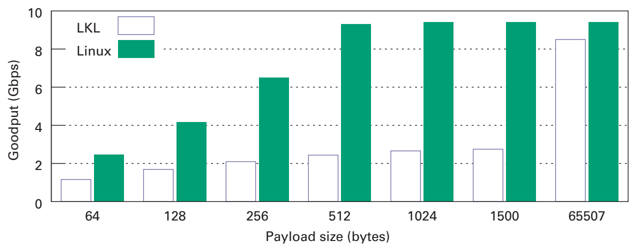

Figure 3: TCP Packet Transfer Performance When Changing Packet Size (Goodput)

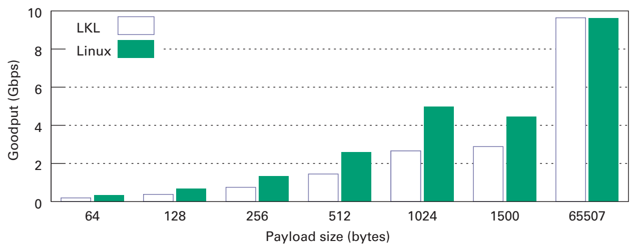

Figure 4: UDP Packet Transfer Performance When Changing Packet Size (Goodput)

Author:
Hajime Tazaki
Senior Researcher, Research Laboratory, IIJ Innovation Institute Inc.

*6 Chu et al., User Space TCP - Getting LKL Ready for the Prime Time. Linux Netdev 1.2 (Oct. 2016)