Toward Reusing Device Drivers through a Portable and Minimal Driver Interface

Yu Lu[1] Takahiro Shinagawa[1]

Abstract:
Developing new operating systems (OSes) is often hindered by the lack of device drivers. Prior approaches, such as virtualization and OSKit-like frameworks, support limited driver reuse but suffer from runtime overhead, high development effort, and poor portability. We aim to make new and experimental OSes run easily and efficiently on non-virtualized environments. To this end, we propose a portable and minimal driver interface that enables a new OS to access device drivers of a commodity OS without virtualization. Our design reduces runtime overhead by avoiding hypervisor intervention, lowers development effort through a simple message-based API, and improves portability by isolating driver access from OS-specific dependencies. In this design, the new OS and the commodity OS kernel run on separate cores, and the commodity OS kernel handles all driver-level I/O operations. The two systems communicate through a lightweight message-passing mechanism implemented over shared memory and inter-processor interrupts. We implemented a prototype using Linux on x86, and it successfully boots a new OS through our interface. We are now extending the implementation to support several devices.

Keywords: Operating System, Driver reuse, OS interface

1. Introduction

Operating systems (OSes) have long been an important field of research, driven by the need to explore new hardware architectures, improve efficiency and security, and simplify system design and maintenance. In practice, however, building a new OS from scratch remains a challenging and time-consuming task, primarily due to the lack of device drivers. Modern computing platforms include a wide variety of peripheral devices, and developing or adapting drivers for all of them requires significant effort. As a result, many new or experimental OSes cannot fully run on real hardware and are often limited to emulated environments.

Porting device drivers is particularly difficult because drivers rely on kernel-specific interfaces, interrupt handling mechanisms, and memory management schemes. Even minor differences in kernel APIs can lead to incompatibility, forcing developers to reimplement similar functionality repeatedly. This process not only consumes development resources but also introduces a high risk of errors, such as race conditions or memory corruption. Consequently, many research OSes remain incomplete or untested on actual hardware. Reducing the engineering cost of driver development while maintaining performance and compatibility is therefore a key challenge in OS research.

Several studies have explored ways to reuse existing device drivers. Virtualization-based approaches [1], [2] reuse host OS drivers through virtual or paravirtual interfaces, but they rely on hypervisors and hardware virtualization, which cause runtime overhead and make them unsuitable for bare-metal environments. OSKit-like frameworks [3], [4], [5] modularize kernel components to facilitate OS construction, yet they are difficult to adapt across different programming languages and kernel architectures. Other systems [6], [7], [8] execute drivers in separate domains or kernels and communicate via message passing, improving reliability but adding complexity and communication overhead. These limitations motivate a lightweight, portable, and non-virtualized approach to driver reuse.

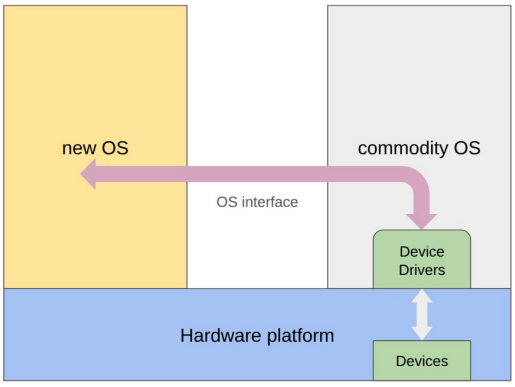

Fig. 1 Our portable and minimal driver interface for reusing device drivers

We propose a portable and minimal driver interface that enables a new OS to access device drivers of a commodity OS without virtualization (see **Fig. 1**). In this design, the new OS and the commodity OS kernel run on separate processor cores; the latter manages all driver-level I/O operations. They communicate through a lightweight cross-OS interface implemented using shared memory and inter-processor interrupts. This interface provides a simple message-based API for sending I/O requests and receiving responses, which eliminates hypervisor involvement and reduces runtime overhead. Because the interface abstracts driver access at the message level, it minimizes kernel dependency and improves portability across different OS designs

[1] The University of Tokyo

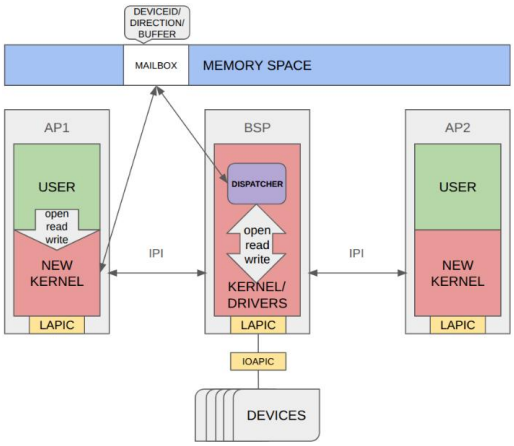

Fig. 2 Architecture for reusing device drivers

and programming languages. Furthermore, its minimal design lowers implementation effort and allows new OSes to run and interact with real devices easily and efficiently, leveraging unmodified commodity OS drivers with low overhead.

2. Design and Implementation

Our system is designed to enable a new OS to reuse existing device drivers of a commodity OS without virtualization. The overall architecture is illustrated in **Fig. 2**. Modern multi-core processors include one Bootstrap Processor (BSP) that performs system initialization and multiple Application Processors (APs) that can execute independent tasks. In our design, the BSP runs the commodity OS kernel, which manages all devices through its existing drivers, while the APs run the new OS. Hardware interrupts from peripheral devices are routed exclusively to the BSP so that all driver-level I/O operations are handled by the commodity OS. The two OSes communicate through a lightweight cross-OS interface implemented using shared memory and inter-processor interrupts (IPIs).

The proposed interface provides simple primitives for message-based communication. Two basic functions, `request` and `reply`, are used for sending and receiving messages, respectively. A dispatcher within the BSP kernel receives each request from the new OS, invokes the corresponding device driver, and sends back a reply once the operation completes. This message-based abstraction eliminates hypervisor involvement and avoids complex virtualization layers, thereby reducing runtime overhead and simplifying implementation.

To perform I/O operations, the new OS prepares a message containing essential information such as `DEVICE_ID`, operation `DIRECTION` (e.g., read or write), and `BUFFER_ADDRESS`. This message is stored in a predefined shared memory region called the `MAILBOX`, and the new OS sends an IPI to the BSP to signal a request. Upon receiving the IPI, the dispatcher retrieves the message, executes the corresponding driver in the commodity OS kernel on behalf of the new OS, and writes back the result to the shared memory.

We implemented a prototype using the Linux kernel as the commodity OS running on the BSP. The new OS runs on an AP and boots without requiring BIOS or UEFI functions; instead, it relies on the Linux-based loader provided by the BSP. Memory regions are partitioned to define areas for shared communication (`MAILBOX`), device I/O mappings, and memory spaces for each OS. The prototype successfully boots a new OS through this interface, confirming the feasibility of driver reuse on bare-metal systems without virtualization. We are extending the implementation to support additional device classes and improve system stability.

3. Summary and Future Work

We presented a portable and minimal driver interface that enables new operating systems (OSes) to reuse existing device drivers of a commodity OS on bare-metal environments without virtualization. Our design separates processor cores to run the new OS and the commodity OS kernel, which manages all driver-level I/O operations. The two OSes communicate through a lightweight message-passing mechanism implemented over shared memory and inter-processor interrupts (IPIs). We implemented a prototype on x86 using Linux as the commodity OS and demonstrated that a new OS can successfully boot and access devices through this interface.

In future work, we plan to extend our implementation to support additional device classes, such as network and storage controllers, and to evaluate the performance of our approach.

Acknowledgments This work was supported by JST, CREST Grant Number JPMJCR22M3, Japan.

References

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, p. 164–177 (online), DOI: 10.1145/945445.945462 (2003).

[2] LeVasseur, J., Uhlig, V., Stoess, J. and Götz, S.: Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines, *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04 (2004).

[3] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A. and Shivers, O.: The Flux OSKit: a substrate for kernel and language research, *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, p. 38–51 (online), DOI: 10.1145/268998.266642 (1997).

[4] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C. and Eggers, S.: Extensibility safety and performance in the SPIN operating system, *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, p. 267–283 (online), DOI: 10.1145/224056.224077 (1995).

[5] Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. and Hyden, E.: The design and implementation of an operating system to support distributed multimedia applications, *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 7, pp. 1280–1297 (online), DOI: 10.1109/49.536480 (1996).

[6] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems, *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, p. 29–44 (online), DOI: 10.1145/1629575.1629579 (2009).

[7] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R. and Hunt, G. C.: Rethinking the library OS from the top down, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, p. 291–304 (online), DOI: 10.1145/1950365.1950399 (2011).

[8] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., Williamson, M. et al.: Safe hardware access with the Xen virtual machine monitor, *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (2004).