ResearchCoders' For Programmers Series
The Design and Implementation of Microdrivers
V.1

Mohammed Q. Hussain
mqh@reseachcoders.dev

January 2019

This document is one of "For Programmers" series, a part of ResearchCoders project. It explains the ideas of [1] for programmers to help them implement them. Please visit our website for more information: http://www.reseachcoders.dev

1 Introduction

Device drivers can be considered as an extension of the operating system's kernel, they work in kernel-mode where they have privileges similar to the kernel itself. Furthermore, to write device drivers, the programmer cannot use user-mode development tools. Although running device drivers in kernel-mode is good for performance, but the drawback is that they decrease the reliability of the operating system. For example, one device driver that is written in a bad way can cause the kernel of operating system to crash, which means the whole machine will stop. According to the paper 89% of Windows XP crashes are caused by device drivers, while in Linux driver code has 2 to 7 times the bug density of other kernel's parts.

To mitigate the reliability problem of device drivers, the authors of the paper "The Design and Implementation of Microdrivers" [1] proposed *Microdrivers* which is an idea that is obviously inspired by *microkernel* design. A microdriver has two parts, the first one called *k-driver* which is a portion of device driver's code that works in kernel-mode due to its performance requirements, the second part is *u-driver* which is the other portion of device driver's code that work in user-mode as a process. In this way, most of device driver's code can be moved to the userspace, which serves the reliability of the operating system.

1.1 Microdriver's Benefits

The architecture of Microdrivers has been designed to gain the following benefits. First, since a large portion of the driver will be moved to the userspace

(u-driver) then the normal userspace programming tools can be used with u-drivers and they can be considered as normal applications. Second, since the code that should work in high performance remain in the kernel and work in kernel-mode, then Microdrivers give a performance that's comparable to the traditional architecture of device drivers and better performance than device drivers that work fully in userspace. Third, if there are bugs in u-driver, this will not cause the whole system to crash. Finally, microdrivers aim to be compatible with current operating systems, and the existing device drivers can be converted to microdrivers, which saves all the time invested in writing current device drivers.

1.2 DriverSlicer

To convert existing device drivers from traditional drivers architecture to Microdrivers architecture automatically, the authors proposed a tool called *DriverSlicer* which has two parts. The first part is the *splitter* which decides which portion of driver's code should remain in kernel (k-driver) and which portion should be transferred to userspace (u-driver), this decision is taken based on performance criteria. The second part is the *code generator* which generates the code that moves the data between the components of microdriver (u-driver and k-driver).

2 Microdrivers Architecture

As mentioned before, a device drivers in Microdrivers architecture are divided into two parts, the first part is *k-driver* which runs in kernel, the other part is *u-driver* which runs in userspace as a process. For efficiency, multiple k-drivers may use the same u-driver. The kernel always calls the k-driver, and the k-driver can call u-driver. These calls from k-driver for u-driver appear as local calls to the k-driver, but in reality, an RPC-like mechanism is used to realize such calls. Microdrivers architecture should be compatible with current device drivers, so kernel's interface for the device drivers should not be changed.

2.1 Dividing the Code

To decide which portion of code should be in k-driver and which should be in u-driver, the authors chose performance as a criteria. According to the authors, the device drivers usually contains two types of code, either *data path* code or *control path* code.

The data path code is responsible for transmitting and receiving data, these operations are performance-critical, so, they should be in k-driver for better performance. The control path code usually perform non-critical operations such as initializing and configuring the device and handling the errors, this type of code should be moved to u-driver. The general rule is to keep any performance-critical code in k-driver while move the other to u-driver.

2.2 Runtime

In Microdrivers architecture, runtime layers should be presented in both kernel and userspace to provide the following functions:

- **Communications**: The k-driver and u-driver need to communicate somehow to call each other's functions and to transfer data.
- **Object tracking**: After dividing a driver into u-driver and k-driver, the data structures that hold the shared information between the two components will be available in two versions, user-mode version for u-driver and kernel-mode version for k-driver. Object tracker's responsibility is to synchronize these data structures between the two components. For example, when some changes are performed in a data structure of u-driver, these changes should be reflected also in k-driver and vice versa.
- **Recovery**: One of Microdriver's goal is to improve the reliability of device driver's architecture. Hence, one of runtime responsibilities is to restore the system's operations after a failure in u-driver. For example, unload the k-driver when u-driver crashes.

3 Epilogue

This document aimed to show the general architecture of Microdrivers for the programmers. Referring to the original work [1] is necessary for the programmers who would like to implement Microdrivers on some existing kernel since the details of *DriverSlicer* are not mentioned in this document for the sake of simplicity. Also, the authors have implemented Microdrivers in Linux kernel and they wrote in their paper about this experiment and how they managed to implement this architecture to achieve its goals.

References

[1] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178. ACM, 2008.