A User-Space Device Driver Framework

William M. Grim *
Commodities Systems
Morgan Stanley
New York, NY, USA
grimwm@gmail.com

Stephen A. Blythe
Department of Computer Science
Southern Illinois University - Edwardsville
Edwardsville, IL, USA
sblythe@siue.edu

Abstract

Most modern operating systems have the tendency to be complicated and error prone because device drivers operate in close proximity to the operating system kernel. This paper implements and tests a method for making the execution of device drivers more robust by placing them in a controlled environment and strives to set an example for how modern operating systems should approach this task.

With drivers in a controlled environment, they are simpler for software developers to implement since they will no longer need to worry about unintentionally corrupting the operating system kernel. The advantage to this is that drivers should no longer be the primary cause of operating system freezes that cost users time and money.

1 Introduction and Prior Work

Operating systems are one of the core components to a computer. In fact, operating system research dominated much of the early history of computer science, with much of the research involving monolithic kernels that are used in operating systems such as GNU/Linux or BSD. However, a newer breed of kernels known as microkernels delivers a new set of promises and problems.

Microkernels involve multiple design paradigms, including principles from computer architecture, operating systems, and distributed systems. While modern microkernel performance now roughly matches monolithic kernel performance on most levels, one lingering problem with microkernels is keeping device drivers inside a protected memory region known as user-space without sacrificing performance. Prior work on these types of device driver frameworks indicates that poor performance is the result of poor design. However, related work in distributed systems indicates that even with a good design, the overall problem is inherently difficult because it involves communication amongst many nodes, creating a fully-connected graph of communication links [2, 7]. Therefore, the central issue is to find an optimal compromise between functional perfection and responsiveness.

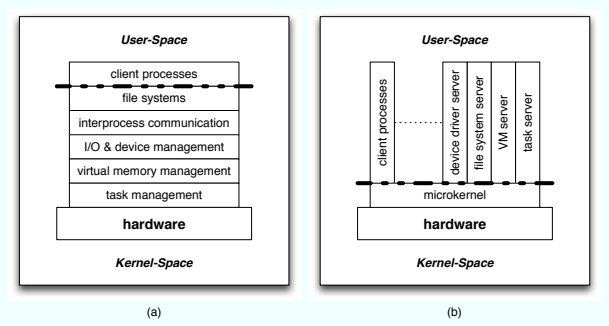

Figure 1: Kernel design in (a) monolithic kernels and (b) microkernels.

When operating systems were first created, the most easily understood design was that of the monolithic kernel. In the monolithic kernel, all primary services and many secondary services are integrated into a single, shared address space that has full privileges to the CPU and its hardware [4]. Some of the services offered by a monolithic kernel include scheduling, file systems, networking, device drivers, and memory management [6].

Since all the code for a monolithic kernel executes as an executive[1] on a CPU [3], it has full access to all kernel data structures currently within the CPU's address space. Due to the fact that monolithic kernels execute as privileged code in an area commonly referred as kernel-space, the obvious, primary advantage is that it is easier to leverage the performance of hardware, because there is almost no interpro-

*Work completed while author was at Southern Illinois University-Edwardsville

[1]Set of software instructions executed on a CPU in a priviledged mode, giving the software full hardware access.

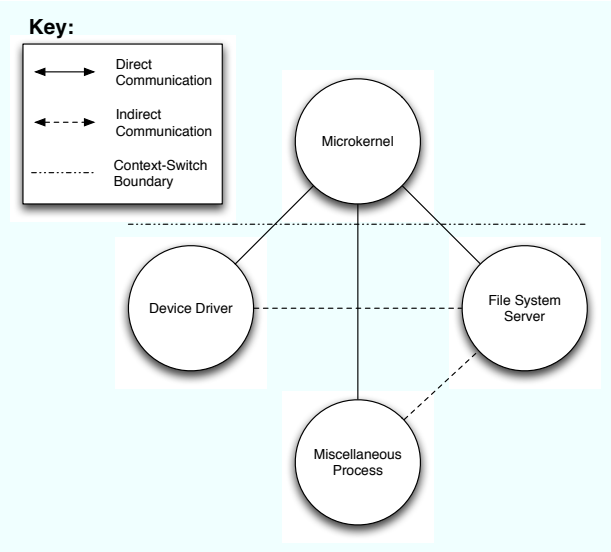

Figure 2: Microkernels exhibit client/server behaviour.

cess communication (IPC) overhead inside the kernel. However, there are some serious drawbacks to the monolithic approach, including the lack of transparency between kernel services, the lack of fault tolerance, and the overwhelming complexity of having all kernel services in the same, unprotected address space, making it virtually impossible to verify the overall correctness of a kernel [6].

1.1 The Microkernel

In contrast to monolithic kernels, which place operating system specific code directly in kernel-space, pure microkernels put only essential operating system functions into kernel-space; all other services are placed outside the kernel. Microkernels replace the vertical scaling of monolithic kernels with horizontal scaling [6], as shown in Figure 1. An example of how microkernels represent a client/server architecture is shown in Figure 2.

A microkernel operating system is a set of user-space servers built on top of an existing microkernel [6], such as Mach or L4, with one critical service being the device driver framework. The device driver framework is the backbone service for all devices in the system, providing a common way for device drivers to access devices and input/output (I/O) buses on behalf of requesting clients. This allows microkernel operating systems to be easily extended into several application domains, such as hard and soft real-time systems, high-availability systems, high performance systems, or any combination of these and others.

As operating system history, and in general, software development history has unfolded, it has become clear that object-oriented approaches to large-scale problems are often beneficial to reducing the complexity of a given system. In fact, operating system researchers once theorized that a similar approach to kernel design could alleviate the shortcomings of the monolithic kernel by splitting primary and secondary services into two groups. The set of primary services, such as IPC and kernel thread scheduling would remain in kernel-space with the microkernel, and everything else would be placed in user-space, where memory addressing is protected. In this sense, a microkernel approach can also be viewed as object-oriented in nature.

1.2 Microkernel Performance

While early versions of microkernels suffered from performance issues due to complex message passing and Remote Procedure Call (RPC) issues [6] [4], modern approaches have worked to rectify these problems. To mitigate the performance problems that first generation microkernels were suffering, two branches of development took place. The first branch decided to keep the same generation of microkernels but move some critical servers and device drivers back into the kernel, based on empirical observations of good performance in monolithic operating systems; this led to a breed of kernels known as hybrid kernels. The other branch of microkernel development led to a more radical redesign of microkernels, aiming to decrease the size of the microkernel as much as possible, thus eliminating enough interprocess communication overhead to make them practical for general use; this branch of development has given way to the second generation of microkernels [4].

Both hybrid kernels and second generation microkernels had tradeoffs. In hybrid kernels, the tradeoffs included better performance, a larger, less flexible kernel, and more interfaces rather than fewer. In contrast, second generation microkernels completely eliminate performance issues; however, it remains an open question as to whether second generation microkernels primitives are flexible enough for modern demands [6].

2 Building Blocks of the Proposed Framework

Our design for a user-space device driver framework is built upon work seen in deva/fabrica [1] and is implemented on Minix 3.0. Deva/fabrica is a user-space device driver framework implemented for the GNU

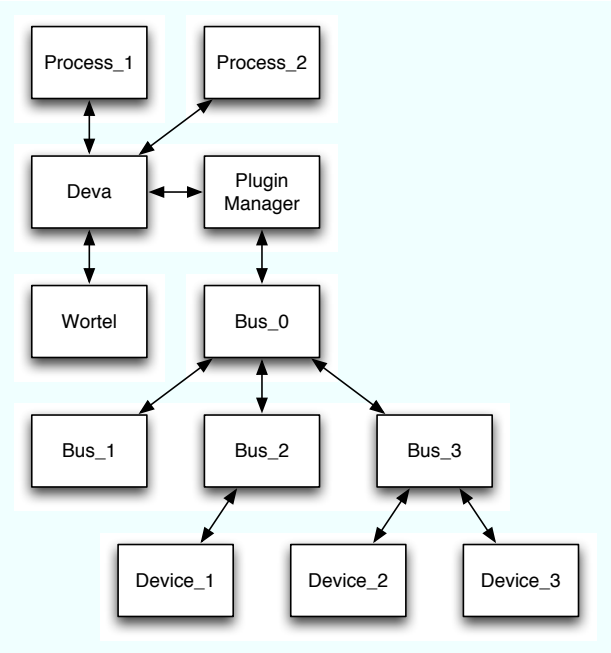

Figure 3: Deva/fabrica layout.

Hurd-L4 microkernel. It makes many practical decisions but was never completed and suffers from some design complexities. The goals of this *device driver framework* (DDF) are to be comparable in speed to existing monolithic systems, portable, flexible, consistent, tolerant of critical device driver failures, and to provide a convenient interface. To help meet the goals set forth by the deva/fabrica project, our DDF needs to consider three primary components are: bus drivers, device drivers, and services.

In Figure 3, it can be seen that deva/fabrica models the physical hardware layout of a system. For example, there is one entry point , Bus_0, to all primary busses. Bus_0 is a virtual bus driver with information about how to access all directly-connected children busses. At the lowest level of the driver hierarchy are device drivers, which are directly attached to bus drivers.

2.1 Drivers

The bus drivers watch I/O busses for insertion and removal events and notify plugin managers of these events, providing them with the necessary information to load the correct device driver. The bus drivers also provide metadata about a device to its driver, such as the bus addresses to which its attached, the interrupt request (IRQ) numbers, the DMA channel IDs, etc. Also, bus drivers provide a set of communication primitives for use between the drivers and devices, such as send/receive and memory (un)mapping. Support for rescanning busses to check for devices whose drivers are not loaded is also supported [1]; this particular ability is especially useful during bootstrapping when some devices are not loaded because their drivers depend on other devices to be running first, such as printers depending on parallel port drivers.

All I/O operations that need use of memory buffers require the user-space application provide the device driver with the memory containing the I/O to be done. This is in contrast to monolithic kernels where certain devices use kernel-space memory buffers. The advantages to having user-space memory buffers is that it avoids a memory-to-memory copy of data [1, 5, 8] and prevents user memory from entering protected memory.

2.2 Driver Classes

In the Deva/fabrica specification, device drivers may be one of nine device classes, including character, block, human input, packet switched network, circuit switched network, framebuffer, streaming audio, streaming video, and solid state storage [1]. Each of these classes has a specific type of interface for communicating with the I/O bus to which they're attached. As mentioned earlier, the bus drivers themselves will provide this interface to the device drivers.

Deva/fabrica's nine device classes can be simplified to just two classes: *character* and *block*. Character devices cover five of deva/fabrica's device classes: human input, packet and circuit switching networks, and streaming audio and video. All other aforementioned device classes fall under the block devices category.

2.3 Services

In Figure 3, the *Wortel* subsystem is a system-central IRQ logic server that runs in kernel-space, and its primary task is to enable IRQ rerouting [1]. When IRQs are shared amongst multiple drivers, it is their responsibility to pass IRQs to peers.

The *plugin manager* found in Figure 3 is a process where bus events are sent to ask deva to (un)load drivers. Each bus driver is associated with exactly one of the potentially many running plugin managers. Also, the plugin manager maintains a list of currently loaded drivers and provides this information upon request.

The *deva* module found in Figure 3 supports interaction with the host operating system. Its primary functions are to provide support for thread creation

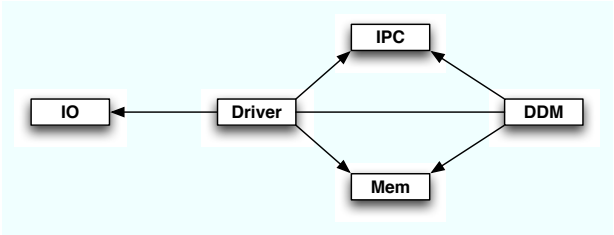

Figure 4: Device driver framework layout.

and deletion, memory (de)allocation, I/O port mapping, driver (un)loading, and bootstrapping.

With *deva*, thread creation and deletion, memory (de)allocation, and I/O port mapping are done by asking the host operating system to perform these tasks on its behalf. Thus, *deva* provides an operating system abstraction so drivers do not have to be modified for each operating system.

during the handling of bus events. It is also used by user-space processes that wish to manually load drivers. The reason why *deva* loads drivers instead of having the plugin manager so is simply due to the fact that *deva* is asking the host operating system's file system for a stored driver to load. This is more consistent with using *deva* as an abstraction to the operating system.

3 Device Driver Framework Design

The *device driver framework* (DDF) design contains three primary components: an interprocess communication (IPC) library that provides common methods for DDF communication, a *device driver manager* (DDM) that monitors device drivers registered with it, and a keyboard snooping driver that tests the correctness of the framework. Figure 4 presents an overview of the DDF. An in-depth discussion can be found in the following subsections.

It is very important to realize that the DDF's primary purpose is not to improve the raw performance (speed) of existing frameworks, but merely to enhance the safety of driver execution and simplify the driver implementor's task. Performance is an important secondary goal, so the DDF should either maintain or exceed current performance levels seen in existing second-generation microkernel OS's.

3.1 Device Classes

In our framework, there is no distinction amongst whether a driver is block, character, or raw; in the context of our framework, these classes are only relevant to higher level subsystems (such as the file system) which do not affect driver operation. In fact, drivers are each free to interact with the hardware in any way they see fit.

3.2 Device Scheduling

Device drivers each have a first-in-first-out (FIFO) queue where requests to use the device are sent. This allows the drivers to pull new requests from a single location one at a time while clients are allowed to add new requests in parallel.

The framework is not designed to allow client processes to directly lock drivers for exclusive use. The reasons for this include avoiding a performance penalty that would have occured by always having to check bits to see if the calling client had appropriate access rights and because it was intended that security be appropriately handled by other OS subsystems. In fact, the primary responsibility of this DDF is to provide "utility" to driver writers and to leave security to dedicated subsystems. An example of a subsystem that could properly handle security for device driver access is a device file system similar to FreeBSD's devfs[2]. This both simplifies the design and makes it more robust by concentrating on one task only.

3.3 User Presentation

To the user, the DDM represents the drivers scattered in memory as a coherent hierarchy. In Figure 5, busses are internal nodes while devices are leaves. The DDM provides facilities for searching such a tree structure in order to make finding a driver or set of drivers easier. For example, a client can ask the DDM for a list of drivers attached to the PCI bus, and the DDM will return the set of drivers claiming to belong to it.

All top-level independent drivers must attach to a special *null bus* supported by the DDM. The null bus provides a central access point into the tree so that when a client wants a listing of all drivers on the system, it can begin by enumerating all drivers on the null bus. Mostly bus drivers will be connected to the null bus. Other drivers that do not directly depend on bus drivers for operation, such as a keyboard driver, can also be found on the null bus.

3.4 Framework Abstraction

In this design, abstraction libraries are the core components needed to enable communication among the

[2]Provides a file system-level interface for client processes to access device drivers.

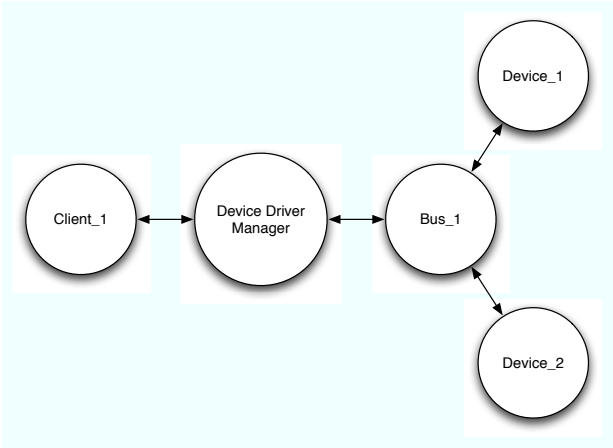

Figure 5: Device driver manager's user presentation.

operating system, device drivers, hardware, DDM, and clients. The libraries need to be efficient; otherwise, the computational overhead could get in the way of time-critical operations. Therefore, a rich set of primitives that can be used to pass raw data quickly are provided. The libraries created for this DDF are *libMem, libIO, libIPC, libDDM*, and *libDriver*, as discussed below.

The memory module, *libMem*, provides a method to copy large slabs of memory from one process to another with the help of the native OS kernel. However, it does not currently provide methods to (de)allocate memory on the heap, because most modern compilers can now dynamically handle this on the stack. Regardless, a future edition of the DDF could handle slab allocation operations for drivers that wish to use the heap for DMA rather than the stack.

Another basic library needed for all higher-level functions in the DDF is the I/O library, *libIO*. This library provides drivers with the ability to read and write various-sized data to computers' I/O ports. This eliminates the need for driver writers to write architecture-dependent assembly to perform the same task.

The interprocess communication library, *libIPC*, provides methods for all low-level interprocess communication in the DDF. It also acts as the basis for all higher-level communication protocols used by the DDM and drivers.

Building on *libIPC* is the DDM proxy, *libDDM*. This library provides easy-to-use methods for communicating with the DDM. Internally, it transforms method calls into IPC messages that are transmitted with libIPC, but externally, access to the DDM is transparent and feels just like a local method invocation. Without a library such as this, clients (typically drivers) would have to understand the exact protocol the DDM uses for communication; that would be an error-prone and mundane task for developers.

The driver library, *libDriver*, is used by both drivers and driver clients. From the driver end, driver writers inherit the classes in libDriver and override the operations they wish to handle. The operations currently supported are *read, write, open, close, scatter, gather, I/O control, initialization*, and *finalization*. Each driver method receives all arguments passed to it in a libIPC message customized to that particular method.

From a client's perspective, *libDriver* acts as a proxy, handling communication with drivers similarly to the way libDDM handles communication between clients and the DDM. This means driver method invocations are as simple as "a local method call" with well-known arguments.

The framework's abstraction libraries are object-oriented, providing a modern approach to development and allowing the framework to be adapted to other platforms more quickly and easily. Currently, the libraries are statically linked to code; however, the DDF can be modified to provide these libraries as shared objects.

3.5 Device Driver Manager

The device driver manager is a server task that manages drivers and allows other tasks to find and gain access to them. Due to the fact that the DDM is the core process in the DDF, the DDM is resilient to failures of devices or their drivers.

Since all drivers reside in their own address space and are assumed to reside on the same system, the DDM does not need extensive fault tolerance. When a process unexpectedly dies and another process tries to communicate with it, the process manager will tell the calling process why the call fails. Likewise, when a failure is recognized, a log message is generated, and it is up to the administrator to resolve the problem. However, since the existing implementation of this design is on Minix 3, Minix's reincarnation server (RS) can reincarnate processes that have died unexpectedly. Such allows for a large degree of fault tolerance.

4 Experimental Results

In the user-space device driver framework, correctness was analyzed via unit tests and mock objects, and performance was proven to be similar to Minix's existing framework. To analyze performance, a single working driver in both our framework and Minix's existing framework was created to generate timing measurements.

Framework	Open	Read	Write	Close
Existing	0.0042	0.0056	0.0050	0.0042
New	0.0054	0.0048	0.0042	0.0062

Table 1: DDF method call timing averages in ticks.

Framework	Open	Read	Write	Close
Existing (0)	4,979	4,972	4,975	4,979
Existing (1)	21	28	25	21
New (0)	4,973	4,976	4,979	4,969
New (1)	27	24	21	31

Table 2: Number of method calls that took 0 or 1 ticks to complete.

4.1 Correctness

The unit testing framework is a custom set of classes that can be fed any type of data. The data fed to the unit tests is always *(actual result, expected result)* tuples. Each unit test is considered valid when the two values match, while a failure is indicated when the two values differ.

For most of the DDF's code, unit tests were the obvious choice in proving teh correctness of the implemented framework. One exception to this was the keyboard, which produced non-deterministic output since the data was entered onto a keyboard by a human. In this case, a mock object was used to record whether a method was called or not. When the driver was unloaded, a unit test was executed to validate that all methods were called; this test always succeeded.

4.2 Performance

To ensure performance in the new DDF was similar to that of Minix's existing DDF, timing measurements of several method calls were taken, building information about method call overhead. The methods themselves did nothing and just returned, and these measurements were taken on four typical methods: *open, read, write*, and *close*. The averaged results of 5,000 calls to each method is shown in Table 1.

Averaging the timing results in Table 1 , one can see that the existing framework averages 0.0048 clock ticks per call while our framework averages 0.0052 clock ticks per call. Further, Table 2 shows that most of the calls took no recorded clocks ticks to execute, and the rest of the calls only took one clock tick to execute. Thus, it can be seen that the averages are based on numbers that do not vary greatly, providing strength to the argument that the averages are indeed representative of general method call overhead.

Based on the results of Table 1 and Table 2, it is shown that the new DDF is roughly 6.8% slower than the existing DDF in terms of call overhead, proving that the new DDF's method call speed is similar to Minix's existing device driver framework by less than an order-of-magnitude. However, in general operation, it is believed that method execution times would be much larger than method call overhead, making this slightly degraded performance even less noticeable.

5 Conclusions and Future Work

In the end, the new user-space device driver framework is a success in terms of functionality. With the framework existing in user-space, safety of driver development has improved. When drivers ran in kernel-space, any flaws in the driver could easily bring down an entire computer system by corrupting the operating system kernel. However, with drivers in user-space, failing drivers can only impact processes directly relying on them. This can still result in an entire system grinding to a halt, but the possibility of this is lessened.

The new DDM this new framework provides is able to give users a better presentation of the driver layer by bestowing users with a hierarchical view of the driver tree rather than the flat view given by Minix's process manager.

While this design has improved certain areas of driver development, it still leaves room for enhancement. Proceeding is a listing of potential areas that could be enhanced or extended along with a description of how the enhancement might be accomplished.

Adding DMA memory allocation to *libMem* would be a large step towards making the new DDF more competitive with existing frameworks. Currently, memory allocation is not guaranteed to be contiguous in physical memory, and this guarantee needs to be made for DMA-capable hardware and drivers.

It would also be useful for the DDF's libraries to be shared objects that could be dynamically linked into code using the DDF. This would allow drivers to take advantage of updated libraries as they become available without having to rebuild the drivers.

To improve the fault tolerance of the DDM, the DDM should recognize when a registered driver terminates unexpectedly. It should then unregister the driver and attempt to revive it, logging the problem that occurred.

Adding an access control layer to the DDF is an area that deserves attention. The DDF as currently implemented provides "implementation", but

another layer needs to provide "access". These areas should remain separate, and one way to add the access control layer would be to integrate the DDF with existing file systems. This would have the advantages of being able to use drivers as files while the file system controls access to these special files. However, for this to work, the framework would need to have device classes the file system understand.

References

[1] M. Brinkmann. *Porting the GNU Hurd to the L4 Microkernel*. Free Software Foundation, Inc., August 2003.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 55 Hayward St., Cambridge, MA 02142-1315, USA, 2 edition, 2001.

[3] Intel. *IA-32 Intel Architecture Software Developer's Manual: Basic Architecture*, volume 1. Intel Corporation, Denver, CO 80217-9808, USA, 2004.

[4] J. Liedtke. Toward Real Microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[5] K. M. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, Reading, MA, USA, 1 edition, 1996.

[6] W. Stallings. *Operating Systems*. Prentice Hall, Upper Saddle River, New Jersey 07458, USA, 5 edition, 2005.

[7] A. S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, New Jersey 07458, USA, 1 edition, 2002.

[8] S. A. G. L. Team. *L4 eXperimental Kernel Reference Manual*. Universität Karlsruhe, 6 edition, June 2005.