Monolith[1]: A Device Driver Framework for Microkernel Operating Systems

William M. Grim

January 19, 2006

Abstract

One particular problem with microkernels is keeping device drivers inside a protected memory region known as user-space without sacrificing performance. Analysis on recent work in these types of device driver frameworks indicates that poor performance is the result of poor design. However, the literature on 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. Therefore, the central issue is to find an optimal compromise between functional perfection and practical usability. *XXX: Please note that this abstract is a verbatim copy of a paragraph in my introduction. I really like this paragraph and am not sure in which place it belongs.*

1 Introduction

Operating systems are one of the core components to a computer. In fact, operating system research has 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 Microsoft©Windows or BSD. However, a newer breed of kernels known as microkernels delivers a new set of promises and problems.

Microkernels have always offered the promise of improved extensibility, flexibility, reliability, and others; however, the largest failure in early microkernel designs was performance. To counteract this failure, years of research have considerably improved microkernel performance to the point where microkernels are now a viable alternative to monolithic kernels.

Even though 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. Analysis on recent work in these types of device driver frameworks indicates that poor performance is the result of poor design. However, the literature on 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. Therefore, the central issue is to find an optimal compromise between functional perfection and responsiveness.

When a compromise between functional perfection and responsiveness has been reached for a device driver framework, its design must be implemented and analyzed. Analysis will include mathematical proofs to demonstrate the design's correctness and its expected performance. Later, after a full implementation of the framework, empirical analysis on it will be performed, demonstrating that, the framework works in a practical setting. Also, the empirical analysis will demonstrate its performance against other frameworks used in operating systems such as GNU/Linux and FreeBSD, demonstrating that the framework is responsive.

2 Background

A microkernel operating system is a set of user-space servers built on top of an existing microkernel 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) busses on behalf of requesting clients. However, to better grasp a general understanding of microkernel operating systems, a brief background on their development is required.

2.1 The Monolithic Kernel

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 [6]. Some of the services offered by a monolithic kernel include scheduling, file systems, networking, device drivers, and memory management [7].

[1] In *2001: A Space Odyssey*, an object known as the "monolith" gave mankind the ability to use tools, leading to their present superiority over the world. The word is also intended as an antonym to "microkernel", in much the same way that Unix is to Multics.

DRAFT

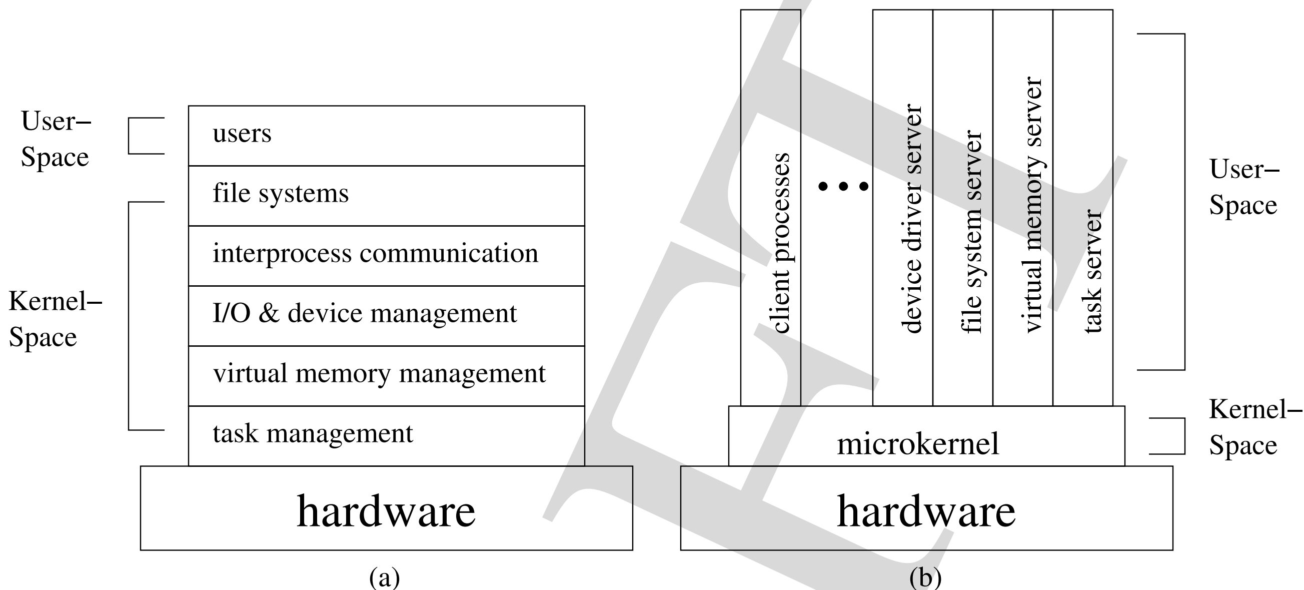


Figure 1: Kernel design in (a) monolithic kernels and (b) microkernels [7].

Since all the code for a monolithic kernel executes as an executive[2] on a CPU [5], 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 relatively no interprocess 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 [7].

2.2 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 [7], as shown in Figure 1. An example of how microkernels represent a client/server architecture is shown in Figure 2.

The microkernel's unique design pattern maintains several advantages over monolithic kernels; see Figure 3. All the unique features referenced in Figure 3 interact to essentially allow 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 [7].

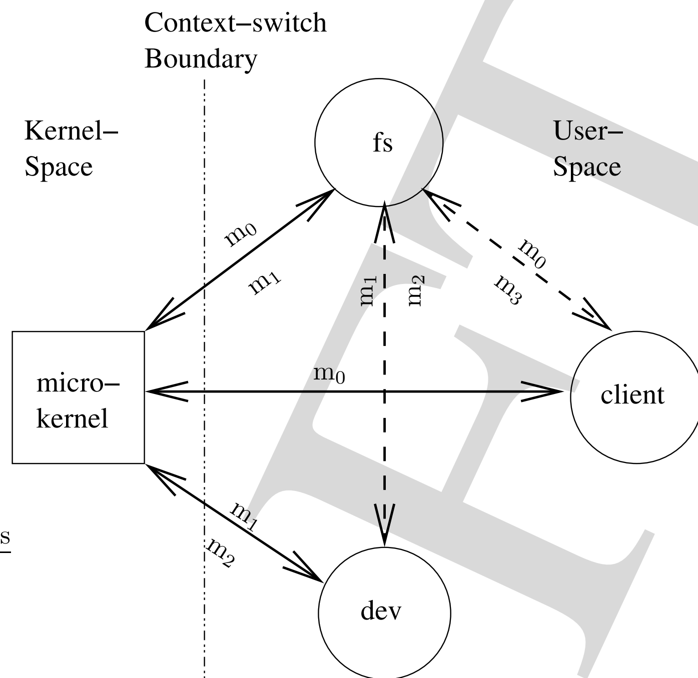
The advantage of a *uniform interface* to all operating system services should be obvious; it provides a consistent mechanism for operating system services to interact [7]. However, perhaps less obvious is the fact that multiple computers could effectively interact with each other using the same message passing interface as tasks on a single system [7]. Another less obvious advantage to this uniform interface is the fact that, depending upon particular design decisions, message passing systems could allow multiple computers to communicate in varying degrees of architecture independence [7].

Also, due to the fact that services are loosely-coupled, microkernels provide *extensibility*, which allows new services to be added or extended relatively easily [7]. This advantage is because of the fact that loosely-coupled code has the tendency to be smaller, making it easier to make changes without needing to review as much code at once [7].

Another feature of microkernel operating systems closely related to *extensibility* is *flexibility*. *Flexibility* affords the ability to easily strip down or change software to meet changing demands [7]. For example, one packaging of the operating system may need to handle high-availability (HA), high-performance computing (HPC), while another packaging may need to run on a PDA or game console. Meeting these changing demands in a monolithic operating

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

DRAFT



PSfrag replacements

Message Number (m_i)	**Message Contents**
0	Ask to open file.
1	Ask for device handle of device with file.
2	Return device handle (or capability, to be discussed later, to it).
3	Return file handle (or capability, to be discussed later, to file).

Figure 2: Microkernels exhibit client/server behaviour. Dashed lines are indirect links, and solid lines are direct links. Messages are denoted by m_i.

system would take much development time and much more testing time than a microkernel operating system.

Likewise, small, loosely-connected services lead to increased *reliability* [7]. For example, all application programming interfaces (API) can be thoroughly scrutinized [7].

The client-server nature of microkernel operating systems exhibited by the message passing architecture means that microkernel operating systems are *inherently distributed systems* [7]. A simple example of this fact is that the process scheduling server will have to interact, via messages, with the virtual memory server as it swaps processes in and out of the CPU and memory.

Finally, microkernel operating systems' client-server nature leads to *object-oriented* or *component-oriented* design [7]. *Object-oriented* methodologies dictate the tight coupling of related data and methods into single entities known as *objects* [7]; likewise, *component-oriented* methodologies dictate the tight coupling of related objects into single entities known as *components* [8]. Both of these methodologies naturally work together in microkernel operating systems, providing a clear context of what everything in the operating system does, easing a programmer's understanding of the system.

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[3].

[3]Memory is securely partitioned so that one unprivileged task cannot access another task's memory via arbitrary addresses. Any requests by an external task to receive another task's memory must be explicitly granted by a memory manager.

DRAFT

1. *Uniform interface to operating systems*: Message passing allows a consistent way to access services from remote servers.
2. *Extensibility*: Services can be added or extended relatively easily without rebuilding or rebooting the operating system.
3. *Flexibility*: Operating systems can be easily stripped down or modified to meet changing demands.
4. *Reliability*: All code can be rigorously tested, since the code base is small.
5. *Portability*: Architecture-dependent code only exists in the microkernel and special user-space servers, leaving the majority of the software architecture-independent.
6. *Inherent distributed system*: If all operating system tasks have globally unique identifiers, then in effect, the operating system is a single system image (SSI) at the microkernel level.
7. *Object-oriented / component-oriented design*: Natural evolution of operating system design as a result of the microkernel's inherently distributed operation via client/server architecture.

Figure 3: Advantages of microkernel-based designs [7].

Microkernel	**Version**	**Size (KB)**	**System Calls**
GNU Mach	1.3	≈300	222
L4Ka::Pistachio	X.2	≈12	12

Figure 4: Microkernel comparison [2, 9].

2.3 Microkernel Performance

Microkernels have a history of performing poorly, because it takes time to build messages, to send them from one end, to receive them at another end, and to decode them. However, the amount of performance penalty is difficult to analyze and depends upon particular implementations of message passing [7].

In fact, first generation microkernels, such as Mach and Chorus, often suffered large performance penalties due to complex message passing systems and large, relatively complex code bases. Even optimized versions of these microkernels drug along the performance problems [7].

Surprisingly, after intense examination of first generation microkernel performance, it was realized that the overhead for context switching was not the problem, even though initial intuition suggests the larger number of context switches should be the cause. In fact, the main problem lay in the inherent IPC overhead in first generation microkernels every time a remote procedure call (RPC)[4] was executed. As an example, optimized first generation microkernels execute RPCs roughly eight times *slower* than their Unix system call counterparts [6].

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 [6]. A comparison of the code size between a first and second generation microkernel is shown in Figure 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 [7].

[4]In this context, this is synonymous to doing a system call in a monolithic system. In actuality, however, RPCs allow a client to remotely execute code at a remote location and get results back, all in a transparent fashion.

DRAFT

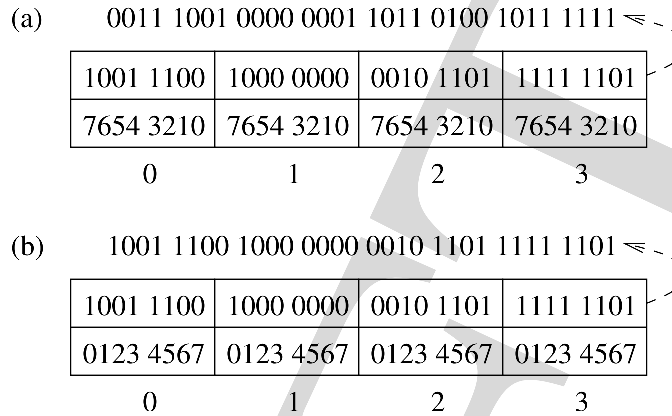


Figure 5: Endianness: (a) little endian and (b) big endian [5].

2.4 Bootstrapping Microkernel Systems

Pure microkernels introduce problems when trying to boot a computer, also known as *bootstrapping*. For example, when the microkernel first loads onto the system, it has no working knowledge of devices and file systems; however, in order for the microkernel to get any work accomplished, by conventional standards, it needs to access file systems on hardware devices in order to load more software. This introduces a chicken-and-the-egg problem whereby the kernel needs to know about devices whose device drivers are separate from the kernel; therefore, something else must be done, such as using external boot-time modules that provide the necessary information.

In general, another problem with bootstrapping is that historically, boot loaders have been written for specific operating systems. These boot loaders are loaded by the Basic Input/Output System (BIOS) and are used to boot the rest of the kernel. However, the Multiboot specifiction and one implementation of it helps solve this problem and others on the Intel IA-32 architecture [3]. Historically, boot loaders have been written for specific operating systems; however, the Multiboot specification and one implementation of it helps solve this problem and others on the Intel IA-32 architecture [3].

2.4.1 Intel 32-bit Architecture

Before getting into the details of bootstrapping, a bit of background on Intel's 32-bit architecture (IA-32) is required.

Intel's address structure is *little endian* [5], meaning each byte's least-significant-bit lays on the right-hand side and most-significant-bit lays on the left-hand side. However, entire bytes are ordered in ascending order, with the least-significant-byte laying on the left-hand side and the most-significant-byte on the right-hand side [5]. For an example, see Figure 5.

All memory addresses in IA-32 are *byte addressable* [5]. What this means is that all 32-bits of an IA-32 CPU can be used to address up to 4 GB of memory: $2^{32}B = 4,294,967,296B * \frac{1\ KB}{1{,}024\ B} * \frac{1\ MB}{1{,}024\ KB} * \frac{1\ GB}{1{,}024\ MB} = 4GB$.

Linear address space is a typical hardware phrase indicating the fact that some sort of memory mapping hardware is available to convert a given 1-dimensional address into an N-dimensional cross-section in real memory [4]. Consider the 32-bit address, 0x1000FF98. When the memory mapping hardware receives this address, one possible way to do the mapping is by using a row address strobe[5] followed by a column address strobe[6] to return data to the requester [4]. Under this method, the mapping hardware will use 0x1000 to find a row in real memory and 0xFF98 to find a particular column (i.e. bit) in that row [4]. The important thing to remember is that all memory appears to be logically ordered as a linear set of addresses, while in reality the memory mapping hardware could map the addresses to a foreign set of rules.

2.4.2 Multiboot Specification

To deal with the problem of booting microkernel-based operating systems and GNU Hurd in particular, Erich Boleyn and Brian Ford devised the Multiboot specification [3]. Essentially, the Multiboot specification solves two

[5]Also known as RAS, the address is used to find a particular row in memory matching that address [4].
[6]Also known as CAS, it works almost like RAS, except it works on the row found in RAS to find a particular column [4].

DRAFT

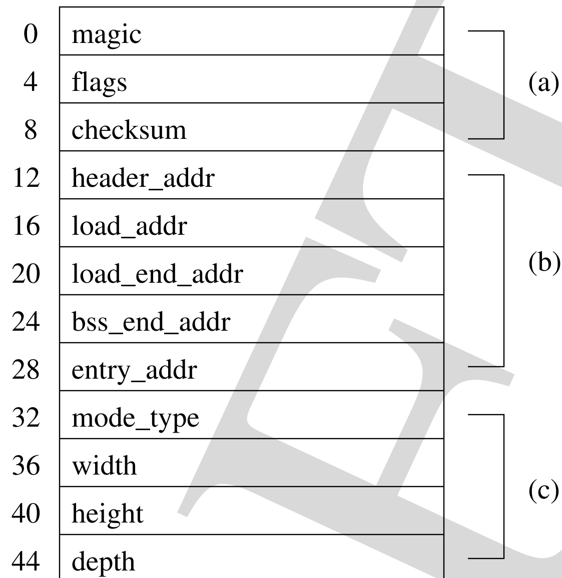


Figure 6: Multiboot header fields: (a) always required, (b) required if *flags[16]* set, and (c) required if *flags[2]* set.

core problems. The first, more classic problem it solves is a consistent bootstrapping protocol for various types of kernels (Linux, BSD, etceteras). The second, more important problem it solves for microkernels is the ability to load a kernel and several kernel servers (modules) at boot time [1].

To solve the bootstrapping protocol problem, the Multiboot specification requires that kernel images contain a special header that gives information about the file layout. This header is known as the *Multiboot header* and is shown in Figure 6. The Multiboot header is a consistent data structure that can be prepended to kernel images to create kernels that Multiboot-compliant boot loaders can understand.

The Multiboot header in Figure 6 is also known as a magic header and eliminates the need for kernel developers to write special purpose boot loaders that are only compatible with specific kernels by specifying three critical fields: *magic*, *flags*, and *checksum* [1]. The *magic*[7] field is located at relative address[8] $0x0$ and defined to be $0x1BADB002$, while *flags* specifies features the operating system requests or requires of the boot loader. Also, *flags* indicates to the boot loader that the Multiboot header contains optional information that should be used [1]. Finally, when the values in *magic* and *flags* are summed with *checksum*, the result should be a 32-bit unsigned zero [1].

For implementors of microkernel-like kernels, the Multiboot specification solves an even greater problem: the loading of several software modules at boot time. This is a critical component to writing microkernel operating systems, because it allows developers to write independent code segments, apart from the microkernel, which are necessary to facilitate the full operation of operating systems. Once a kernel and its modules are in memory, the bootloader notifies the Multiboot-compliant kernel of the modules and their locations in memory via the Multiboot information structure[9] seen in Figure 7 [1].

2.4.3 GNU GRUB

GNU GRUB (GRUB) is the first boot loader ever created with the Multiboot specification in mind [3]. It was conceived by Erich Boleyn in 1995 while he was attempting to modify FreeBSD's boot loader to conform to the Multiboot specification; he was doing this in an attempt to get GNU Hurd on University of Utah's Mach 4 microkernel to boot. However, modifying FreeBSD's boot loader proved to be an unreasonble amount of work, and Erich decided to begin GRUB. Over time, Erich's priorities changed, and GRUB became a GNU project in 1999, being adopted by Gordon Matzigkeit and Yoshinori Okuji [3].

[7] Magic values provide information about the rest of the program, such as its linking type.

[8] If a program located at physical memory location 0xAE98 tries to access relative memory location 0x0040, it will actually access physical memory address $0xAE98 + 0x0040 = 0xAED8$.

[9] On IA-32, the Multiboot-compliant kernel knows where the Multiboot information structure is via the physical address set in the EBX register.

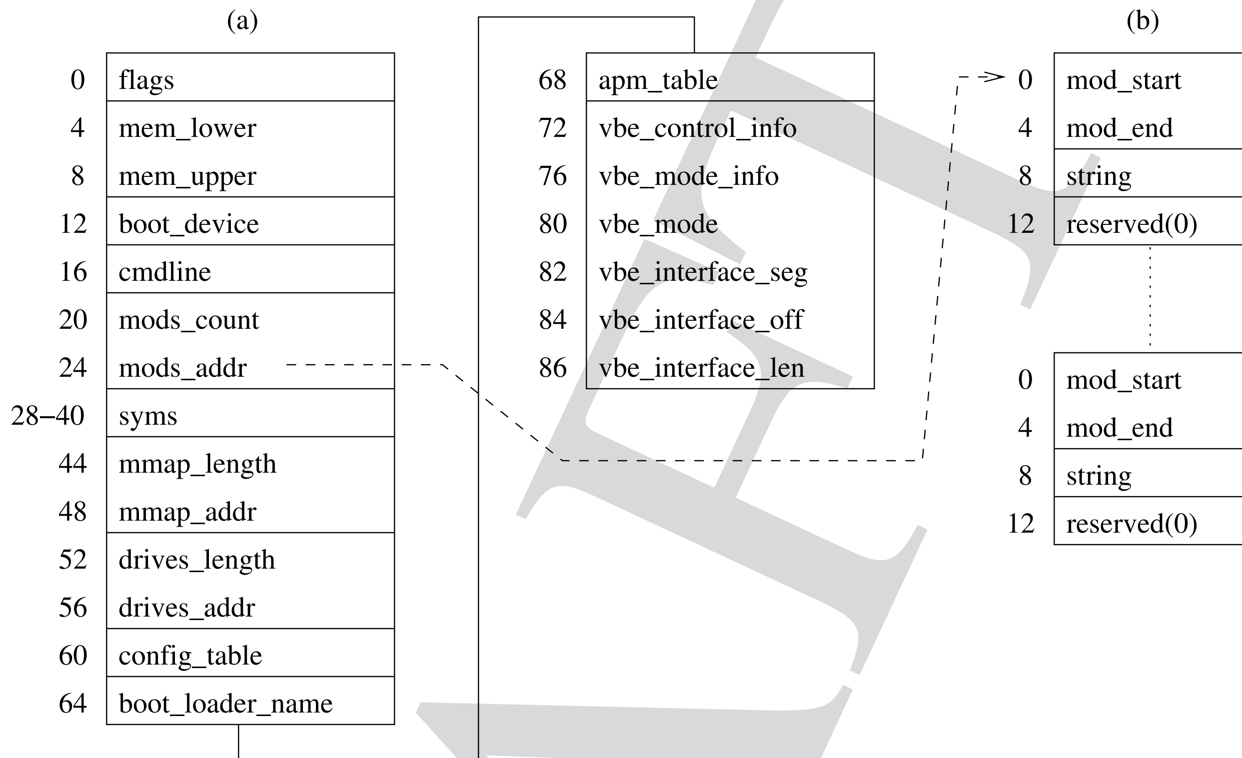


Figure 7: (a) Multiboot information structure and (b) array of Multiboot module structure(s) that exists if *flags[3]* set and *mods_count* $\geq$ 0.

Currently, GRUB only runs on the IA-32 PC architecture; though, it is likely GRUB will be ported to other architectures in the future (especially when the Hurd extends beyond IA-32). Moreover, it is easiest to itemize the major list of GRUB features, as shown in Figure 8 [3].

The list of GRUB features is large, providing a powerful boot loader implementation that is extremely useful to microkernels because of its Multiboot compliance in loading kernel modules. However, GRUB does lack ability in two key areas: it is currently bound to the IA-32 PC architecture, and it can only load kernels at memory addresses $\geq 1\ MB$ [3].

On modern PC systems, it is unlikely that any kernel would be less than 32-bit addressable. However, GRUB forces kernels to support at least 32-bit memory addressing, because it can only place kernels at the 1 MB (0xFA000) or larger memory address [3]. The reason for this is because on IA-32, the linear address space is byte addressable, meaning a 16-bit kernel can only reach 64 KB (0xFFFF) of main memory; thus, a kernel must boot into IA-32's protected mode at a minimum, which is a 32-bit mode, giving the kernel direct access to 4 GB of linear address space [5]. Fortunately, GRUB takes care of the special boot code that places the CPU in 32-bit protected mode, greatly easing the kernel implementor's job and freeing them from creating specialized boot loaders [3].

3 Microkernel Examples

3.1 L4Ka::Pistachio

The L4Ka::Pistachio (L4Ka) kernel is an implementation of the original L4 design by [........]. L4Ka's main features are simplicity, speed and security.

L4Ka has a straightforward API consisting of eleven system calls and twelve data types. These system calls and data types handle the most basic and central operating of microkernels: the kernel interface page and API information, threading, timing and thread scheduling[10], address spaces and mapping, IPC, exception handling, and

[10]Does not refer to typical scheduling algorithms for simulating parallel processing. Instead, it allows threads to schedule a thread of lower or equal priority to run immediately.

1. Implement full Multiboot specification compliance.
2. Achieve a straight-forward interface to end-users.
3. Employ rich interface for kernel experts.
4. Maintain backward boot compatibility for BSD, Linux, and various proprietary kernels, such as Microsoft©Windows™.
5. Recognize several executable formats, such as a.out variants and the Executable and Linking Format (ELF).
6. Use human-readable configuration files with preset boot information.
7. Provide menu of configuration presets when booting.
8. Provide a boot-time command-line shell for on-the-fly boot customization before booting a kernel.
9. Support several common filesystem types to make GRUB filesystem independent.
10. Support drive access from any drive recognized by the BIOS and work independently of drive geometry translations.
11. Detect all installed RAM.
12. Support large disks using logical block address (LBA) mode.
13. Support booting kernels via network protocols (e.g. TFTP).
14. Support local access via remote terminals (e.g. serial line access).

Figure 8: Major list of GNU GRUB features.

memory and processor control. All other services normally seen in monolithic kernels are considered second class, to be handled by user space task servers (threads).

Along with L4Ka's number of system calls and data types being small, its memory footprint is as well, being only 12 KB. Thus, L4Ka usually resides in the CPU cache, and it exchanges most data via CPU registers. Therefore, both of these attributes are its primary source of increased performance.

3.2 GNU Hurd-L4

GNU Hurd-L4 aims to run on an implementation of the L4 microkernel. Though, recent developments at the GNU Hurd-L4 mailing lists indicates that their focus may be shifting towards the Coyotos kernel (another second generation microkernel written by Dr. Jonathan Shapiro).

The reason for the Hurd-L4's shift in focus is not to further delay releasing the project, but it is because the Hurd-L4 project is not sure whether or not the L4 primitives[11] will be enough to support a complex operating system such as the Hurd-L4.

If GNU Hurd-L4 switches to the Coyotos kernel, it will actually receive an extra feature: persistence. Support for operating system persistence is built into Coyotos and conceptually allows swap space and file systems to act as one single-memory-store. Thus, a user would no longer need to save files, because the mere act of writing anything to memory would save it during an operating system checkpoint[12] or swap to the backing store.

Whatever the Hurd-L4 project chooses as their kernel, one of the primary goals retained from the original GNU Hurd project is the ability for multiple computers to act as a single cluster (collective). Thus, the ability to serialize[13] data must be provided by the operating system. Coyotos already has such features while L4 does not.

[11] Basic API.
[12] Periodically, checkpoints are created by the operating system to save the entire operating system state to the backing store.
[13] Convert complex data structures to binary strings.

4 Related Research

In this proposal, I would like to design and implement a device driver framework for microkernel operating systems, building off of past work and initially targeting the GNU Hurd running on a second-generation microkernel. Work on past $2-nd$-generation Hurd device driver frameworks has been done in two forms: deva/fabrica and Antrik's POSIX proposal.

4.1 Deva/Fabrica

This is the first real device driver framework proposal written and implemented for GNU Hurd-L4. It makes many practical decisions; however, it never reached maturity and suffers from some design limitations as well.

Deva/fabrica is the first device driver framework DDF implemented for GNU Hurd-L4. It makes many practical decisions but never reached maturity and suffers from some design limitations. The goals of this DDF are to be high speed, to be portable, to be flexible, to have a convenient interface, to be consistent, and to be fault tolerant of critical device driver failures. To help meet the goals set forth by the deva/fabrica project, it has three primary components: bus drivers, device drivers, and service servers.

The bus drivers watch I/O busses for insertion 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 it's attached, the IRQ numbers, the DMA channel IDs, etceteras. Also, bus drivers provide a set of communication primitives for use between the drivers and devices, such as send and receive. Support for rescanning busses to check for devices whose drivers are not loaded can be handled; 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.

The DDF defines nine device classes, including character, block, human input, packet switched network, circuit switched network, framebuffer, streaming audio, streaming video, and solid state storage.

XXX: I have more to the last paragraph and this section, but I haven't done so yet.

4.2 Deva/Fabrica Design Analysis

Deva/fabrica is a complex device driver framework that overengineers what a framework should be, answering problems that do not really exist. For example, there is no need for nine device classes, because they can all be simplified to just two classes: character devices and block devices. Another clear example of how complicated the DDF design is is shown in Figure **??**.

XXX: I will add more to this part.

5 Proposed Goals

The device driver framework has three major stages that need to be implemented. The first stage includes the implementation of abstraction libraries that provide common methods for the device driver framework to interact with the rest of the operating system and vice-versa. The second stage consists of a device driver manager (DDM) that monitors and polices device drivers that register themselves with the DDM, and the final stage is the implementation of device drivers that take advantage of the previous stages. Figure 9 shows an overview of the device driver framework.

Also, the device driver framework will not take into account higher level operating system interfaces such as POSIX. This is due to the Hurd-L4 project not yet having POSIX implemented, and because these types of interfaces do not afford efficient ways to get at devices for communication-intensive interaction such as direct memory access (DMA).

5.1 Device Classes

There will be two major classes of devices: externally scheduled and internally scheduled. The former indicates the use of the DDM to schedule access to device driver handles, while the latter indicates that the scheduling should be left to the driver itself.

DRAFT

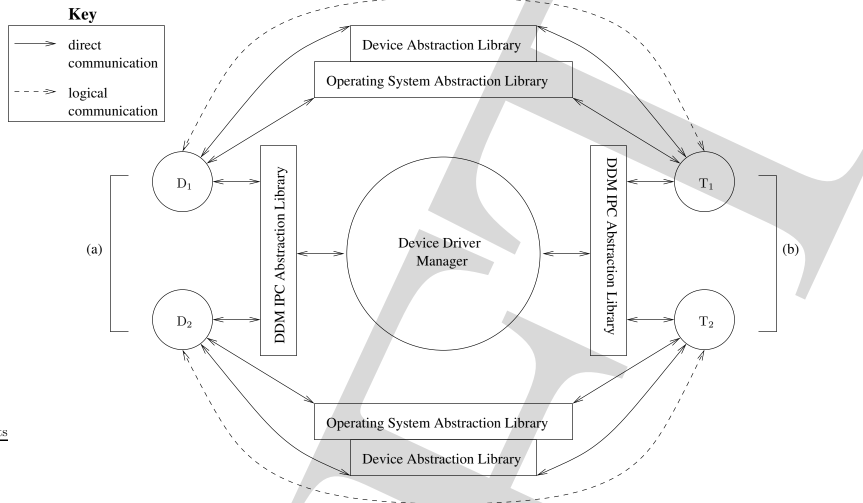


Figure 9: Device driver framework layout: (a) device driver servers and (b) task servers.

5.1.1 Externally Scheduled Devices

Externally scheduled devices assume that the DDM will control the flow of traffic to their device driver by controlling who can get a device driver handle. The primary motivation behind this class of device drivers is to make sure the device conforms to a standard security policy of allowing only one task to control the device at a time. This eases driver writer's task when writing security-sensitive or resource-sensitive drivers by placing the burden of controlling access to the device entirely on the DDM. Some examples of devices that need the tasks to be externally scheduled are printers and security-card readers.

5.1.2 Internally Scheduled Devices

Internally scheduled devices are more open in their security policy than externally scheduled devices by placing the burden of controlling device access on the device driver writer. This may be preferable for certain devices where multiplexed access is within the range of use of the device. Some devices that can have their data multiplexed to several applications at a time are input devices, network cards, and most other hardware devices.

5.2 Framework Abstraction

In this design, abstraction libraries are the core software needed to enable communication between the operating system, applications, device drivers, DDM, and hardware. The libraries need to be efficient; otherwise, the IPC and computational overhead could get in the way of time-critical device-application communication. Therefore, the aim will be to have mostly primitives that can be used to pass raw data quickly; though, the framework will be object-oriented.

At this time, the assumption is that abstraction libraries will be needed for handling two-way communication amongst these services: devices, device drivers, DDM, and operating system. The libraries identified as being able to provide this communication are libos, libdev, and libddm.

5.2.1 Operating System

Libos will be the abstraction of the operating system and may be used by the device drivers and the DDM to be portable across platforms. For example, in GNU Hurd, libos would provide the necessary backend support to be able to acquire memory capabilities; however, on another operating system, capabilities may not be supported and something else will be needed. In general, libos will act as a slab allocator, threading mechanism, and security context handling using the native operating system's functionality to implement them.

DRAFT

5.2.2 Device Abstraction

Libdev will provide the application programming interface (API) for all software to communicate with device drivers in a consistent manner. It will provide a standard set of primitives for communication between device drivers and applications. Instances of libdev will have their interfaces copied directly into tasks to decrease IPC delays using an *eventual-consistency model*.

5.2.3 DDM Communication Abstraction

Libddm will provide a way for the operating system and device drivers to communicate with the DDM. The operating system will likely use it when requesting access to a device driver for a particular device, requesting information about currently loaded drivers, or anything else. However, the device drivers will use it as a means of registering themselves with the DDM and asking the device driver manager for I/O memory capabilities to the device.

5.3 Device Driver Manager

The device driver manager is a task server that manages registered device drivers and allows other tasks to get device capabilities. The DDM is the core item to the entire device driver framework; due to this fact, the DDM must be resilient to failures of devices or their drivers and must provide server-replacement capabilities so other DDM servers can replace the currently running server, such as during an upgrade.

5.3.1 Bootstrap Protocol

Bootstrapping is a recurring theme with microkernel-based operating systems; however, it is doubly difficult with device driver managers.

5.3.2 Upgrade Protocol

By allowing the DDM server to be replaced by another DDM server, users gain the ability of on-the-fly updates to the DDM server. For example, assume a user downloads and builds a newer set of DDM server code; this user can now inform the currently running DDM server that a new DDM server is about to take its place. The currently running DDM server can transfer its capabilities and device driver metainformation to the new DDM server, relinquish control and shut down. This replacement technology greatly enhances user flexibility without sacrificing system uptime.

5.3.3 Event Notification Protocol

The DDM should inform clients when device or I/O bus events occur to the devices or busses about which the clients know.

5.3.4 Fault Tolerance

The DDM must be resilient to benign (non-Byzantine) failures in the system. One possible way is to ensure this is to log the error to a memory page, to load a new device driver manager using information from the device state table, and to reload device drivers from the backing store, if possible.

Clients (tasks) should trust the server (DDM) that is serving them. Therefore, a fault tolerant system can be established whereby long operations do not cause the client to assume the server has crashed or suffered other failures such as omission errors. To enable this, a session could be established between the client and the server that has a dedicated communication path that can be used to occassionally check the status of an operation initiated by the client to ensure that it is still executing, and hopefully we can tell if it is still executing properly.

When data is sent from the server back to the client, the same trust cannot be achieved; so, the server must eventually timeout (or provide some other means of terminating an IPC) when the client doesn't acknowledge receipts quick enough. Therefore, the client must be very fast at saving returned data so that the server does not give up on the client.

For two DDMs, D_1 and D_2, that create a 2-way trust relationship using an authentication server, we can optimize fault tolerance against benign faults by providing state information to each DDM about the other DDM on request.

5.3.5 Device Security Policy

The device driver manager will need to have full access to the I/O memory busses in order to be able to guarantee security to devices to prevent snooping, protection against concurrent access to devices that only support serialization from one task at a time (i.e. printers, card readers, and most other devices), and signalling of events that occur on the I/O busses (i.e. adding and removing USB devices) to registered tasks. XXX: This information needs to go in a separate section, and we need to talk about the bus mastering technique known as DMA.

6 Performance Analysis

This section contains a listing of all the tests that will be performed to ensure the proper evaluation of raw performance and usability in a way that does not place bias towards any aspect of the device driver framework.

6.1 Testing Implementation

6.1.1 Unit Testing

In parallel development with the abstraction libraries will be the device driver management server. Initially to be used to unit test the abstraction libraries' progress, it will expand to allow for the (un)loading of device drivers, management of device drivers, and recognition and signalling of events on the I/O busses.

6.2 Conclusion

This section to be completed after final implementation and testing.

7 Future Research

Some possible areas of future research that could be done after completing the implementation of this proposal include: This section outlines some areas of future research that could be pursued after the completion of the "Proposed Work" section. Some of the goals listed are repetitious to those in the "Potential Goals" section, but they are listed here for completeness in the event that the "Potential Goals" are never realized during this project.

- Interface Definition Language (IDL) that eases the driver developer's workload when creating new device drivers.
- Scalable, Transparent, Internetwork Device Access

References

[1] *Multiboot Specification*, 0.6.93 edition.

[2] Marcus Brinkmann, Gordon Matzigkeit, Gibran Hasnaoui, Robert V. Baron, Richard P. Draves, Mary R. Thompson, and Joseph S. Barrera. *The GNU Mach Reference Manual*. Free Software Foundation, Inc., 0.4 edition, September 2001.

[3] GNU. *GRUB Manual*, 0.94 edition.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 3 edition, 2003.

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

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

[7] William Stallings. *Operating Systems*. Prentice Hall, Upper Saddle River, New Jersey 07458, 5 edition, 2005.

DRAFT

[8] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, New Jersey 07458, 1 edition, 2002.

[9] System Architecture Group: L4Ka Team. *L4 eXperimental Kernel Reference Manual*. Universität Karlsruhe, 6 edition, June 2005.

DRAFT