Examining the Viability of MINIX 3 as a Consumer Operating System

Joshua C. Loew

March 17, 2016

Abstract

The developers of the MINIX 3 operating system (OS) believe that a computer should work like a television set. You should be able to purchase one, turn it on, and have it work flawlessly for the next ten years [6]. MINIX 3 is a free and open-source microkernel-based operating system. MINIX 3 is still in development, but it is capable of running on x86 and ARM processor architectures. Such processors can be found in computers such as embedded systems, mobile phones, and laptop computers. As a light and simple operating system, MINIX 3 could take the place of the software that many people use every day. As of now, MINIX 3 is not particularly useful to a non-computer scientist. Most interactions with MINIX 3 are done through a command-line interface or an obsolete window manager. Moreover, its tools require some low-level experience with UNIX-like systems to use. This project will examine the viability of MINIX 3 from a performance standpoint to determine whether or not it is relevant to a non-computer scientist. Furthermore, this project attempts to measure how a microkernel-based operating system performs against a traditional monolithic kernel-based OS.

Contents

List of Figures

List of Tables

1 Introduction

Most operating systems such as Linux, OSX, and Windows have monolithic kernels or hybrid implementations. In a monolithic or hybrid kernel-based operating system (OS), either all or nearly everything runs in kernel mode. Kernel mode is one of two modes in a system. User mode allows non-privileged programs to run. These programs include everything outside of the kernel. In kernel mode, the OS can execute instructions and reference any memory address [5]. A microkernel can hide important parts of the OS, such as key kernel data structures, from user space applications. Thus, it prevents those components from being mishandled or corrupted. For example, a bug in a poorly-written printer driver can potentially overwrite important information at the kernel level and crash the entire OS. The advantage of a microkernel-based OS such as MINIX 3 is that it is smaller and much better at bug isolation since fewer programs have kernel privileges (see Figure 4).

It is unreasonable to expect developers to write bug-free code. So, it is beneficial to both developers and users to put fail-safe mechanisms in place wherever possible. MINIX 3 is small relative to other operating systems and therefore contains fewer errors. Its size also makes various components of the OS easier for developers to understand which helps to track and eliminate bugs. It also confines most bugs and crashes to user mode [15].

In decades past, systems developers tried to design and implement fast and efficient operating systems because of the slow hardware that existed in the 70s and 80s [7]. This is why most operating systems found in consumer computers today are monolithic or hybrid-based. Now that hardware has become more powerful and accessible, developers have placed their focus on reliability, which comes as the cost of performance for reasons described below. Since kernel space and user space are separated and more programs exist in user space, communication between user space and kernel space is more frequent and incurs more overhead. This message passing is also known as IPC, or, interprocess communication. In MINIX 3, the IPC mechanism is synchronous and only allows for fixed-length messages. This implementation eliminates some bugs caused by buffer management.

As the title suggests, this project aims to examine the use of MINIX 3 as a consumer operating system replacement. To explore MINIX 3 in this context, this project first attempts to port existing programs from

NetBSD and write drivers that allow MINIX 3 running on a BeagleBone Black to interact with a touchscreen LCD display. Since MINIX 3 is not as mature as other operating systems, it is missing some software that typical users might expect to have on their computer. By attempting to write software for MINIX 3, this project contributes to the goal of creating a more reliable and secure consumer operating system that may introduce another option in consumer computing. Next, this project tests the performance of MINIX 3's implementation by running tests on MINIX 3 against another operating system. This thesis will focus on MINIX 3's performance against the FreeBSD operating system. These tests help conclude whether or not MINIX 3 can still perform at a level expected by consumers. Since the project changed scope between terms, this paper will be broken up into two parts: work on the frame buffer driver and work on performance analysis. The section on the frame buffer driver focuses on the hardware used, technical details of frame buffer drivers and displays, and challenges encountered. The section on performance and analysis will focus on MINIX 3's architecture, how the tests are written, what the results are, and what they mean.

2 Background and Related Work

MINIX 3 is ideal for research projects and education because it is both small and open source. This section will briefly outline some ongoing and completed research projects about MINIX 3 and older versions.

As stated, MINIX 3 is primarily concerned with reliability and dependability. A research study examines how to extend MINIX 3's fault-tolerant filter driver framework (a filter driver manages peripherals connected to the machine) to deal with driver failures in the storage stack [9]. Though not the direct focus of this project, success could be beneficial to non-computer scientists if the project were extended to offer greater reliability in encryption, compression, RAID storage techniques, and other features of an operating system that rely on block-device drivers. The researchers measured the performance of their implementation by examining CPU usage between MINIX 3 with a filter driver and MINIX 3 without a filter driver. The implementation was a success, but yielded up to a 28% increase in overhead. Depending on context, this may be considered a small price to pay for the increase in stability that the developers of MINIX 3 advertise [9].

MINIX 3 goes beyond crash prevention, however. Microkernels are rather useful in applications that

require no reboots or human intervention in order to update. One study creates a proprietary operating system called Proteos based off of MINIX 3. The operating system coins the term *state quiescence*, which is a mechanism used to detect when the system is in an ideal state for a software upgrade [4]. Updates that do not require human intervention will allow users to continue working on their machine uninterrupted by frequent and annoying software updates.

Microkernels have been studied before with regards to their use in mobile devices. One study claims that microkernels have strong structuring concepts, transparent reusability of system components, and are capable of running on devices with different processor architectures [3]. If MINIX 3 is capable of being used on server, desktop, and mobile platforms it can easily be made available to users.

MINIX 3 is not only a promising option for users, but also for students. At only about 12,000 lines of code, it is both portable and easier to understand than some other operating systems that exceed millions of lines of code. Students are able to comprehend vital parts of the system while still being able to see how even minor changes can affect the entire kernel. This makes MINIX 3 ideal for learning about common operating system problems such as installation, system calls, process scheduling, memory and thrashing, filesystems, and so on [10].

3 Part I: The Frame Buffer Driver

Implementing a frame buffer driver for MINIX 3 on ARM would prove that, while a microkernel-based OS may be slower, it is still fast enough to use in a context relevant to the average user. This section describes the details of the frame buffer driver implementation. This includes details about the approach, hardware used, challenges, and a final evaluation of this part of the project.

MINIX 3 is free, open source, and POSIX-conformant [8]. To be POSIX-conformant means that there exists a system call interface that the operating system must support [16, p. 11]. Moreover, MINIX 3 is open source, making it the ideal medium for my senior thesis. Furthermore, the development process is not too far off from that of the systems I have used in the past.

3.1 Outline of Approach

To test the viability of the MINIX 3 microkernel in consumer operating systems, the project will attempt to port existing NetBSD programs and possibly write necessary drivers for a 7-inch resistive touchscreen LCD display attached to a BeagleBone Black running MINIX 3. As illustrated in Figures 1 and 2, the BeagleBone Black and LCD Cape (an expansion board for the BeagleBone Black) contain all of the necessary I/O to complete this task. Furthermore, the BeagleBone Black is a low-cost computer that houses an ARM processor, which is a chip that MINIX 3 supports.

3.2 Hardware and Drivers

The first step of the project was to write a proposal and acquire funding. I have purchased off of `www.sparkfun.com`:

1. One BeagleBone Black - Rev C
2. One 4DCAPE-70T

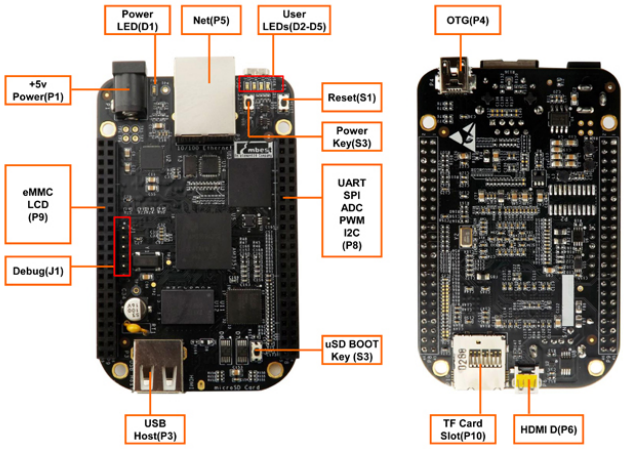

Figure 1: A BeagleBone Black [1].

Figure 2: An LCD display for the BeagleBone Black [19].

The second step of this part of the project was to acquire an adequate knowledge on the implementation of the MINIX 3 operating system. To accomplish this, I have read excerpts of Andrew S. Tennenbaum's book titled *Operating Systems Design and Implementation* as well as official online documentation on the MINIX 3 operating system [16].

The third step was porting and implementing the software and drivers necessary to successfully get the appropriate software running and the LCD Cape to interface with MINIX 3. This will require writing a frame buffer driver to get the LCD display functioning with MINIX 3 on a BeagleBone Black.

The fourth and final step is to evaluate my results as discussed below.

3.3 Challenges and Strategy

There are useful tutorials on the MINIX 3 developer page pertaining to drivers and adding Cape support. At a high level, developers will first need to create a directory and Makefile for the new driver. Then, they will have to write code, compile it, and edit various system configuration files to add system services and their permissions. MINIX 3 has a feature called the Reincarnation Server which is in place to detect failures

in poorly written drivers and stop them if necessary. Developers are expected to write the necessary code to deal with these messages returned from the driver [12]. The MINIX 3 developer pages also state that implementing support for new BeagleBone capes is relatively straight forward. All that is required is the implementation of the driver (as described above), and the name of the Cape to load drivers and configure the Cape settings. [11]

3.4 Evaluation

The fourth and final step in the project was evaluating my results. There are several practical ways to gauge viability, but I feel that the best way is to ensure that, if the implementation of the necessary drivers is possible, the system does not suffer form a decrease in performance that would render it unusable. A drop in performance may be considered when a user experiences a severe lag when trying to run a program or interact with the LCD display. A previous research study using MINIX 3 measures their performance by comparing the time it takes to complete processes on MINIX 3 to the time it takes to complete the same processes on MINIX 2 which does not have isolated drivers in user mode. The processes that they use consist of ones such as getpid, lseek, open/close, and fork. Overall, the evaluation discovered about an 8% increase in overhead [6]. Since the runtime of similar processes has already been compared across systems with different OS implementations, it may be worth while to examine how long it will take to compile and run similar pieces of software across different operating systems. For example, it may be interesting to examine the compilation and runtime of programs in MINIX 3 versus FreeBSD where many MINIX 3 applications were ported from. Another option to evaluate the success of MINIX 3 is to inject faults in both the software in MINIX 3 and the software in NetBSD from which it was ported to see how the respective OS deals with the fault.

4 Progress

There were some unforeseen obstacles that arose once I got started on the project implementation. These obstacles, however, introduced me to some tools and work environments that I had little to no prior expe-

rience with. This section will start by discussing the initial setup process of MINIX 3 on ARM followed by some technical details about the project and my final progress on this part of the project.

4.1 Compilation and Installation

When making kernel-level modifications to an operating system, the source code is required to make the changes. Once the source is obtained and the changes have been made, the kernel has to be compiled before it can be installed and run on hardware or in a virtual machine.

The very first step in the implementation process was to clone the MINIX 3 source tree form the Git repository (`git://git.minix3.org/minix`). From there, building the image is fairly straight forward when following the official developer's guide. After calling the release script included in the source tree, an .img file is created within the source directory on the development machine as well as an object tree containing developer tools. One of which is a crosscompiler for building executable binaries for MINIX 3 on ARM. The image file can then be coped to a micro SD card using the dd command, run in an emulator/virtual machine such as QEMU and Oracle's VirtualBox, or installed natively on a supported machine depending on the intended use of the image.

4.2 Workflow

Code to be compiled for MINIX 3 can be written from within MINIX 3 itself or cross compiled on a UNIX-like machine such as OSX and various Linux distributions. Compilation is not recommended from within MINIX 3 on ARM, however, because it is quite slow. For this particular project, a Linux box running Xubuntu is used for all code compilation, debugging, and file transfer. The debugging process can vary depending on the target platform (See 4.3 for more details). Once compiled, the executable binaries can be transferred to the target over FTP, serial, or SSH. If transferring files over serial, two command-line utilities are required. The first is PICOCOM, and the second is ZMODEM. PICOCOM establishes the serial connection, and ZMODEM can transfer files over serial via an FTDI cable.

4.3 Debugging

Compilation errors can be sorted by reading error messages produced by the compiler. When using an emulator, GDB can be attached to an instance of the emulator to start and stop the program under certain conditions and examine what has happened upon stopping. Debugging MINIX 3 on ARM is a bit less trivial. The current implementation of MINIX 3 on ARM does not yet have a frame buffer driver implemented. This is because the BeagleBone Black uses a new HDMI chip that MINIX 3 on ARM does not yet support. This means that MINIX 3 running on a BeagleBone Black is incapable of display output. In order to interact with the board, it must be attached to a UNIX-like machine using serial transmission via an FTDI cable. Once connected, a user can perform serial I/O through PICOCOM.

4.4 MINIX 3 Drivers and Compatibility

A significant amount of time this term was spent examining how to write device drivers in MINIX 3. There is documentation on the MINIX 3 website describing how to implement drivers, add Cape support to MINIX 3 on ARM, and work with the I^2C subsystem in MINIX 3 (see below). Drivers for MINIX 3 are written in C, and developers can mostly follow standard procedures for developing in C on other UNIX-like systems (compile programs using a GNU Make-like environment and so on). There is one feature in MINIX 3 that affects development and should be addressed. The Reincarnation Server (RES) is a process that ensures all drivers in MINIX 3 are running properly. Visible in the process list, it is able to send messages to all running services. Should a service not respond to the RES, it will try and restart the broken driver. The protocol for interacting with the RES is exposed to developers. All drivers are expected to respond to the RES upon initialization. If the driver fails to initialize, the RES will not try to restart the driver.

In addition to acquiring an adequate knowledge on drivers in MINIX 3, an understanding of the I^2C interface is also important to writing a frame buffer driver. I^2C stands for Inter-Integrated Circuit and is a bus type that is commonly used in embedded systems. With I^2C, an embedded system can transfer data between a processor and external integrated circuits. An I^2C device driver is specific to each device. In other words, there is one instance per device. The BeagleBone Black there are three I^2C buses. Thus, there

should be three instances of the bus driver started. There is a small utility imported from NetBSD called i2cscan that probes each I^2C bus on the BeagleBone Black. The program simply probes the I^2C buses and reads data from them [14].

4.5 The Frame Buffer Driver

Despite the hours examining existing code, reading up on frame buffer drivers for other operating systems, and attempting to compile an existing partial implementation of one for MINIX 3 on ARM, I have not been able to produce a functional frame buffer driver. There were many obstacles that I had trouble resolving such as compiling the image, interfacing with MINIX 3, and most of all, comprehending the complex frame buffer driver to port from NetBSD. The code not only required a deep understanding of the C programming language, but also how the display hardware works, and how certain POSIX system calls work.

In MINIX 3, a device driver will start, block, and wait for a message from the interrupt handler after registering with the interrupt table. Because of MINIX 3's architecture, there are levels of I/O access that might be needed by a user space driver. For example, the file system lives in user space and will need permissions to read and write to disk. Permissions such as this are handled by the kernel calls and the system task. All MINIX 3 divers are their own process and have their own state, registers, and so on. Of course, there is an efficiency loss in these user space implementations. The book *MINIX 3: A highly reliable, self-repairing operating system* states that this efficiency loss is roughly 510% [13]. In MINIX 3 for the BeagleBone Black, hardware drivers communicate with peripherals via the I2C interface (which is consistent between MINIX 3 and NetBSD).

A functioning frame buffer driver is capable of lighting up the display, displaying colors, rotating the display, and drawing a cursor, text, and pictures. At a high-level, the frame buffer should first read in display information data (the manufacturer, resolution, and so on). From there, it should read the source of the item to be displayed. This could be something like an image or console output. Using the display information, the frame buffer driver parses that information and powers the appropriate pixels on the screen.

Systems developer Thomas Cort produced a broken port from NetBSD for the BeagleBone Black equipped

with a similar Cape two years ago as part of his Google Summer of Code project [2]. The code is able to light up the display on the screen and display a single non-black pixel. That port is available on GitHub, but it seems as though it has used an old compilation scheme and is deprecated and unable to build an image from scratch. After many hours spent trying to get the frame buffer driver working, I have been unsuccessful in getting it to function. It seemed that the task was beyond the scope of my ability at the time of the project.

5 Part II: Performance in MINIX 3

Moving away from the frame buffer driver, the remainder of my project focuses on measuring MINIX 3's performance in relation to another POSIX-conformant monolithic kernel-based operating system. I have chosen FreeBSD to use for comparison because its source code is small and it maintains the same system call interface as MINIX 3. Since both operating systems are POSIX-conformant, I should be able to develop a test suite comprised of C programs using the standard C library and run them on both operating systems. I plan to use the statistical profiling tool for MINIX 3 and DTrace for FreeBSD. The profiling tools are capable of measuring the number of system calls that each program invokes, and the amount of CPU time spent executing those system calls.

So far, the MINIX 3 developers have not release performance numbers. In one document, they briefly discuss the performance of MINIX 3 when compared to MINIX 2. The paper explains how they do not think comparing MINIX 3 to Linux or FreeBSD is reasonable since there are different implementations of different parts of each OS [6].

Since my project is examining the performance of a consumer operating systems, I will compare MINIX 3 to the FreeBSD operating system. This thesis will focus solely on performance rather that reliability given the time frame of the project. For now, we will take the developer's claim, that microkernel-based operating systems re far more reliable than microkernel-based operating systems, at face value. This section will focus on implementation details of MINIX 3, followed by experimental methodology, presentation of data, and an analysis of the results.

5.1 Porting User Space Programs from NetBSD

In an attempt to get acquainted with the implementation of processes in MINIX 3, I ported biff, a mail notification program from NetBSD into MINIX 3. The process was trivial and only required moving code from the NetBSD source tree to MINIX 3's and some simple configuration file modifications.

5.2 Processes and Communication in MINIX 3

IPC is a mechanism through which processes can communicate without the need for interrupts. Message passing uses two primitives: send and receive. These primitives ensure that messages are reliably transferred, and that no two process can communicate unless authorized. Processes in MINIX 3 use pipes to send and receive messages. A pipe is 'a first in first out communication channel that may be used for one-way IPC. MINIX 3 is organized in four layers:

1. The kernel is the bottom layer which schedules processes and manages the transitions between ready, running, and blocked states for processes. The kernel also handles messages sent and received between processes. The kernel checks for legal message destinations, locates send and receive buffers, and copies bytes between the sender and receiver. So, IPC requires trapping into the kernel.

2. The layer directly above the kernel contains the processes that have the most special privileges. These are mostly device drivers who have access to input and output ports. They request the system task, which performs system services for the file system, to do this for them (The system task is one of the few programs that resides in kernel space).

3. The next layer up contains servers and processes for user programs. Two of these processes are essential: The process manager and the file system. The process manager carries out all of the MINIX 3 system calls that involve starting and stopping all process execution (such as fork, exec, and exit). The filesystem carries out all filesystem calls (such as read, mount, and chdir).

4. The top layer of MINIX 3 just contains user programs and utilities (such as grep).

Messages are passed in MINIX 3 using the rendezvous principle. When a process does a send, the lowest level of the kernel checks to see if the file destination is waiting for a message from sender. If so, then the

message is copied form sender's buffer to receiver's buffer. If receiver is not waiting, then sender is blocked in a queue of other processes waiting to send to receiver. When a process does a receive, the kernel checks the queue to see if there is a process trying to send. Once the message has been sent and received, both processes are marked as runnable [16]. Because processes are implemented in MINIX 3 the same way that they are in UNIX-like operating systems, they behave the same way in FreeBSD. The difference between the two is that MINIX 3's scheduler lives outside of the kernel.

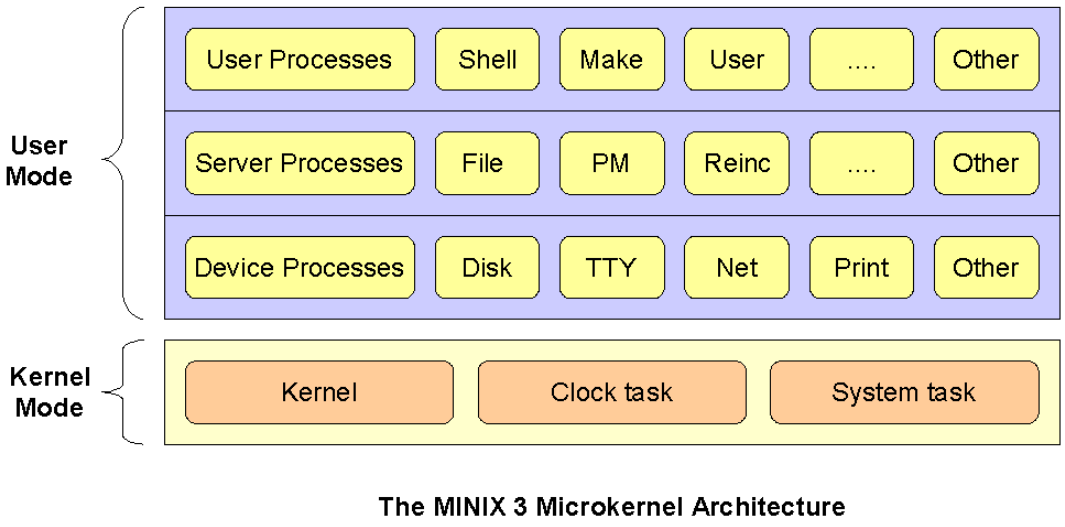

Figure 3: All non-kernel processes take place in user mode [18].

5.3 Test Plan

In order to test how IPC can influence system performance, this project will examine the performance of the filesystem and message passing around a ring. Originally, the tests were planned for execution on an IBM ThinkCentre with 2GB or RAM and an Intel vPro processor. The setup process for this was a but tedious because MINIX 3 and FreeBSD use different boot loaders. After some configuration, GRUB2 was able to boot into MINIX 3 and FreeBSD. Unfortunately, MINIX 3.2.1 does not support USB sticks, nor does it have support for the ethernet card in the development machine. So, the tests had to be performed in a virtual

environment. In other words, the tests were performed in a program, VirtualBox, that emulate hardware for an operating system to be installed on. By running the tests in VirtualBox, I had better control over the state of each OS and could restore the hard disk image if either system crashed. Both the MINIX 3 and FreeBSD machine ran with 2048 MB of RAM and a single CPU core (the host machine has a 2.3 GHz Intel Core i7).

The reason why the file system was selected for testing is because of its dependence on IPC (see Section 5.2). To test the files system, I wrote a C program that takes command-line arguments to create a specified number of files and write the contents of an array to them. After file creation the program reads the files 10, 100, and 1000 times. The program uses the read, write, and open system calls.

In order to test IPC, I wrote a C program to create a ring of processes the size of a command-line argument. A ring of processes is a group of n processes where the first process sends a message to the second process, the second to the third, and so on up until the nth process sends a message back to the first process. Then, the program passes the summed up PID number of all the processes around a ring as many times specified by a second command-line argument. This test uses the fork, read, and write system calls.

5.4 Profiling Tools

There exist two types of system profilers: static and dynamic. Static profiling tools are somewhat rudimentary, and not as comprehensive as dynamic profilers. Static profilers wrap themselves around the execution and have the ability pause at certain points during execution to record the state of the system. The kernel will write profiling samples based on the process name and program counter. Static profilers only provide a subset of executions taking long enough to be seen by the profiler. These profilers interrupt the program during execution, so, to combat this, the interrupt frequency can be changed. Setting a higher frequencies slows the execution time of a program which can affect the results from the profiler.

Dynamic profilers use binary instrumentation and probing to profile the kernel. This means that dynamic profilers can inject code into the executable program and record certain data about the state of the OS when certain requirements are met. Dynamic profilers inject analysis routines into arbitrary locations and can deliver detailed information about a program's execution. DTrace is a dynamic profiler available

on many UNIX-like operating systems, but not MINIX 3. It is a powerful profiling tool that allows users can write scripts in the D programming language. These scripts specify probes that can fire when certain conditions are met to retrieve specific information about the kernel's performance. Dynamic profiling tools rely on binary-level alterations to gather statistical information, so they often require kernel recompilations since they must execute in kernel space. Static profilers can require kernel recompilations as well, but they typically only execute in user space. The default DTrace toolkit comes with a variety of scripts to use that activate certain probes to retrieve data [17].

Unfortunately, all profiling was broken in MINIX 3.3.0. A post that I made in the Google group brought the bug to the developer's attention. They fixed static profiling in a subsequent release, but did not fix kernel profiling. As a result, I was unable to use the kernel profiling utility in MINIX 3 to obtain useful data about the operating system's performance. The user space profiler did work, but I decided not to use it when I also began to experience issues with DTrace on FreeBSD. With DTrace, I was experiencing dynamic variable drops, so decided not to use the tool. A dynamic variable drop occurs when DTrace runs out of space to create new variables to record data to. I suspect that the tool was trying to create a variable to profile each read and write system call and ran out of memory because of the large number of calls. The memory available for these variables can be set by changing a configuration file. This would likely remedy the issue of dynamic variable drops. In the end, I decided not to use profiling tools at all since using a dynamic profiling tool on one machine but not another would lead to inconsistencies since the profiling tools influence execution time. In order to make the testing as fair as possible, the programs were run against the "time" command, which measures the time from process configuration to process termination.

Though the time utility is available on both MINIX 3 and FreeBSD, they implement different POSIX standards which is a minor difference. Time in MINIX 3 writes wall clock time spent on executing, in seconds, to the standard error stream. Time in FreeBSD does the same, but MINIX 3 supports fewer arguments and prints one fewer significant figure than FreeBSD. Both implementations simply write the time elapsed from process creation to termination. One pitfall of the time command is that granularity of seconds on microprocessors is crude and can result in some inaccuracies when reporting times.

5.5 Results & Analysis

The tests were executed on identical virtual machines with the same resources available to them. The three tests below show the performance of MINIX 3 versus FreeBSD creating 10, 100, and 1,000 files. Then, the C program iterates over the files and opens them for reading 10, 100, and 1,000 times, respectively.

Table 1: Creating and reading files in MINIX 3 and FreeBSD

Files Created	Times Read	MINIX 3 Time (s)	FreeBSD Time (s)
10	10	0.00	0.001
	100	0.01	0.002
	1000	0.13	0.027
100	10	0.03	0.005
	100	0.15	0.039
	1000	1.45	0.253
1000	10	00.40	0.055
	100	02.25	0.356
	1000	20.56	2.300

It is worth noting that MINIX 3 was consistently slower than FreeBSD during every single test. What we can draw from these conclusions is that MINIX 3's file system is vastly outperformed by FreeBSD file system. Without the proper profiling tools, it is difficult to see exactly where the performance bottlenecks exist. It is worth noting, however, that some of these tests resemble the 510% overhead mentioned by the developers.

In a UNIX system, processes have two parts: one that lives in user space and one that lives in kernel space. In this model, the user space part can call the kernel space part. This approach is more efficient than a microkernel because procedure calls are much faster than sending messages. For example, the figure below shows how communication between the components of the file system can be slowed by the message passing overhead. The model on the left is implemented in MINIX 3, whereas UNIX uses a model similar to the one on the right.

A user may not notice a lack in performance when copying several smaller files to different folders or external disks, but when putting stress on the file system as the tests did, it becomes clear that MINIX 3 suffers from more performance issues than FreeBSD. For this reason, the file system will perform noticeably

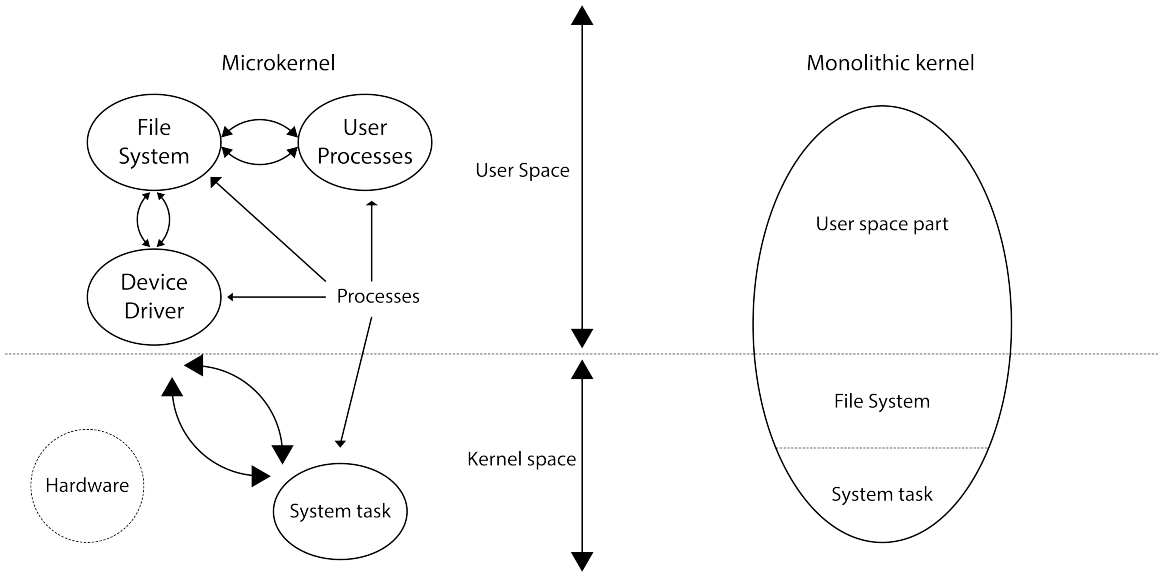

Figure 4: A microkernel versus monolithic kernel representation of the FS [16, p. 156].

worse for large file open, read, and write jobs.

In order to test IPC more directly, I created a test program to fork 10, 100, and 200 processes. Then, a token is passed around the ring of processes 10, 100, and 1000 times, respectively.

Table 2: Creating a ring of processes and passing a token in MINIX 3 and FreeBSD

Ring Size	Times Passed	MINIX 3 Time (s)	FreeBSD Time (s)
10	10	0.01	0.01
	100	0.23	0.01
	1000	2.30	0.01
100	10	0.11	0.10
	100	0.31	0.07
	1000	2.38	0.15
200	10	0.20	0.013
	100	0.43	0.012
	1000	2.50	0.019

What we can draw from this data is that, MINIX 3's IPC incurs more overhead than FreeBSD's. Again, it is difficult to draw detailed conclusions without the proper profiling tools, but it is likely that IPC is neither slower nor faster in one operating system. Either way, when the tasks in MINIX 3 require more IPC, the

degradation in performance will become evident as illustrated by the file system diagram above. It is clear that its implementation is slower than FreeBSD, a traditional monolithic kernel.

6 Conclusion

As a microkernel-based OS, MINIX 3 is better structured and more modular than a monolithic kernel-based OS. It has cleaner interfaces between the pieces, but the UNIX approach is more efficient, because procedure calls are much faster than sending messages. MINIX 3 was split into many processes because the developers believe that with increasingly powerful personal computers available, cleaner software structure was worth making the system slightly slower[16, p. 157].

Above all, MINIX 3 is a tool developed as a research project. As of now, the lack of key software, (such as a reliable window manager and internet connection) make MINIX 3 inaccessible to the average user. The first term of this project focused on developing tools for the MINIX 3 operating system that would help bring it to a state where it was more usable by ta non-computer scientist. Due to various obstacles and my current ability in systems programming, I was unable to accomplish this. My project shifted scope after the first term and set out to examine the performance that comes with a microkernel-based operating system. The lack of kernel profiling, USB support, and underdeveloped networking drivers made comprehensive testing difficult. During the second term of the project, it became clear that the operating system suffers from some under tested software and performance issues. While conducting these tests, I discovered that MINIX 3 also less stable than a mature operating system such as FreeBSD or Linux.

The results produced form the tests corresponded with the developer's claim that MINIX 3's implementation is roughly 510% slower than monolithic kernel-based OSs. This does not mean that all microkernels suffer from issues like this. Though MINIX 3 is both slower and, in my experience, less reliable than FreeBSD, it is still in development and has yet to mature to the state of existing consumer operating systems.

7 Future Plans

One of the fist steps forward will be to get the frame buffer driver finally up an running for MINIX 3 on ARM. This would prove that, though MINIX 3 is slower because of its microkernel-based implementation, it is still capable of running in a way that typical users would expect on a modern computer.

Second, profiling should be done using dynamic, kernel-level tools. This would allow for comprehensive test results that would provide detailed information about the kernel's performance and help identify where the bottlenecks exist in MINIX 3's microkernel. The lack of support for these in MINIX 3 make it difficult to obtain comprehensive results and thus impossible to draw meaningful conclusions about possible bottlenecks introduced by the implementation.

8 Acknowledgements

I would like to thank Professor Matthew Anderson for his guidance, Tom Yanuklis, John Peterson, and Frank Chiarulli for assistance in setting up hardware and software, and the MINIX 3 community for their willingness to help me overcome issues that I encountered with MINIX 3.

References

[1] Alibaba. Beaglebone black bb-black ti cortex-a8 am3359. `http://i01.i.aliimg.com/img/pb/173/754/718/718754173_314.jpg`. [Online; accessed May 27, 2015].

[2] Thomas Cort. Beagleboard.org gsoc 2013 : Minix I2C drivers for the BeagleBone Black. `http://minix-i2c.blogspot.com/`, May 2013.

[3] Michel Gien. Micro-kernel architecture key to modern operating systems design. *Unix Review*, 8(11):58–60, 1990.

[4] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Safe and automatic live update for operating systems. *ACM SIGPLAN Notices*, 48(4):279–292, 2013.

[5] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Construction of a highly dependable operating system. In *Dependable Computing Conference, 2006. EDCC'06. Sixth European*, pages 3–12. IEEE, 2006.

[6] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[7] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Modular system programming in MINIX 3. *login: The USENIX Magazine*, 31(2), 2006.

[8] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Reorganizing Unix for reliability. In *Advances in Computer Systems Architecture*, pages 81–94. Springer, 2006.

[9] Jorrit N Herder, David C Van Moolenbroek, Raja Appuswamy, Bingzheng Wu, Ben Gras, and Andrew S Tanenbaum. Dealing with driver failures in the storage stack. In *Dependable Computing, 2009. LADC'09. Fourth Latin-American Symposium on*, pages 119–126. IEEE, 2009.

[10] James Howatt. Operating systems projects: Minix revisited. *ACM SIGCSE Bulletin*, 34(4):109–111, 2002.

[11] Minix. Beaglebone. `http://wiki.minix3.org/doku.php?id=developersguide:bbcapes`, 2014.

[12] Minix. Programming device drivers in MINIX. `http://wiki.minix3.org/doku.php?id=developersguide:driverprogramming`, 2014.

[13] KS Ramesh. Design and development of minix distributed operating system. In *Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, page 685. ACM, 1988.

[14] Lionel Sambuc. Minix3. `http://wiki.minix3.org/doku.php?id=developersguide:i2cinternals`, Nov 2014.

[15] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[16] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 2006.

[17] Justin Thiel. An overview of software performance analysis tools and techniques: From gprof to dtrace. *Washington University in St. Louis, Tech. Rep*, 2006.

[18] Wikipedia, the free encyclopedia. The minix 3 microkernel architecture. `http://upload.wikimedia.org/wikipedia/commons/7/7d/The_MINIX_3_Microkernel_Architecture.png`, 2015. [Online; accessed May 27, 2015].

[19] Wvshare. Lcd cape 7-inch. `http://www.wvshare.com/img/devkit/accBoard/LCD-CAPE-7inch/LCD-CAPE-7inch-4.jpg`. [Online; accessed May 27, 2015].