Analysis of Practicality and Performance Evaluation for Monolithic Kernel and Micro-Kernel Operating Systems

Hui Miao *hui.miao@microchip.com*
Microchip Australia Design Centre
Microchip Technology Inc.
Brisbane, 4108, Australia

Abstract

The microkernel system (as opposite to monolithic systems) has been developed for several years, with the hope that microkernels could solve the problems of other operating systems. However, the evolution of the microkernel systems did not go as many people expected. Because of faultinesses of the design in system structure, the performance of the first generation of microkernel operating systems was disappointing. The overhead of the system was too high to bear for users. However, the second-generation microkernel system uses an improved design architecture that could substantially reduce the overhead in previous microkernel systems.

This project evaluates the system performance of the MINIX3.1.2a and compares the results with the performance of Linux by using Unixbench system evaluating tool. By this way, it could testify whether the microkernel systems could be more flexible, portable and secure than monolithic operating systems. Unixbench could give sufficient statistics on different capacities of MINIX3 and Linux, such as system call overhead, pipe throughput, arithmetic test and so on. The result illustrates MINIX3 has better performance on Shell Scripts running and Arithmetic test and Linux has better performance on other aspects such as system call overhead, process creation and so on. Furthermore, we provide a more detailed analyse on the microkernel Minix 3 system and propose a method that could improve the performance of the MINIX3 system.

Keywords: Monolithic System, Microkernel System, Operating System

1. INTRODUCTION

Kernel controls the critical parts of operating systems. Nowadays, many current operating systems are monolithic kernel operating systems (e.g. Linux). Monolithic kernel operating systems implement most system functionalities such as file management, device drivers, process management and I/O management in kernel mode. Although monolithic kernel operating systems are very popular, they may have some disadvantages. First, the kernel is intensively complex. A kernel with thousands lines of code could be hard and difficult to maintain. Updating one part of the system may result in needing to recompile the whole kernel. Second, a large amount of code means that the operating system could not be ported to different hardware, especially for embedded systems. Third, the monolithic operating system is not reliable; since the kernel's complexity, the possibility of a system crash could be high. A single tiny error in the kernel could lead the whole system to crash. So microkernel operating systems are designed to overcome the disadvantages of the monolithic systems.

The microkernel system excludes several services out of the kernel. One service can run as a user level application out of the kernel. For example, Mach [1] uses an external pager and the file system can also be running out of the kernel. Minix3 is another microkernel-based operating system which is an earlier version of OS inspired the invention of Linux. The kernel of the latest version of Minix3 only has 4000 lines of code [6], much smaller than Linux. The L4 kernel is a developing microkernel system; the latest version of L4 kernel is L4ka::Pistachio 0.4, which can run on a wide variety of hardware [6].

Compared with macrokernel systems, the microkernel approach has following advantages: First, a microkernel system can provide higher reliability than a monolithic system. A microkernel system has much less chance to crash than a system with a huge kernel. Reducing the size of a kernel is a strategy to reduce the problems in the system. Second, a microkernel system with less kernel code could be maintained more easily. Recompiling the kernel is not a huge task for microkernel systems. Last, a microkernel system could be easily ported to simple hardware, especially in embedded systems.

2. RELATED WORKS AND MOTIVATIONS

2.1 First Generation Microkernels

The first pioneering microkernel conception was in Carnegie-Mellon University, the microkernel Mach operating system. Mach minimizes the kernel into a very small module. The kernel of Mach only provides process management; thread management, IPC and I/O service. The file management, which traditionally is in the kernel, is placed out of the kernel. Mach's external pager [1] was the first conceptual breakthrough toward real microkernel. The conceptual foundation of the external pager is that the kernel manages physical and virtual memory, yet the pager is outside of the kernel. As Fig. 1 shows: if a page fault occurs in user applications it will forward the faults to the pager by message passing. The message is handled by the kernel. This technique permits the mapping of files and databases into user address spaces without having to integrate the file/database systems into the kernel [1].

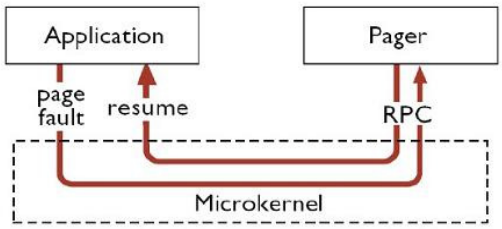

FIGURE 1: Page Fault Processing [1]

The prospect of Mach seems splendid. But it was not as ideal as people expected. The Mach system makes the file system run as a user processes on top of the kernel and uses interprocess communication (IPC) to control this module. IPC contributes a huge overhead to the whole operating system. System calls of traditional operating systems use traps, which are much faster than IPC. Mach needs to create messages, send and switch between processes. As Fig. 2 shows, the overhead is excessive. Chen and Bershad [2] compared applications under Ultrix (a Unix based operating system) and Mach on a DECStation 5-200/200 and found peak degradations of up to 66% on Mach (compared to Ultrix); 66% is really unbearable for user. 75% of the low efficiency is related to IPC. So this first generation microkernel system failed. The Mach system project was abandoned by Carnegie-Mellon University team in 1994.

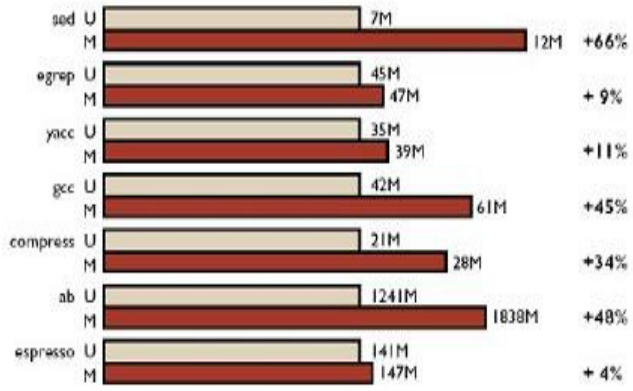

FIGURE 2: Non-idle cycles under Ultrix and Mach [1]

In the case of the failure of first generation, people put forward a compromised way of designing the microkernel system. The main idea is to expand the kernel, put the some file services and device drivers back into the kernel again. In this way, they could reduce the switch time between the user space and kernel. The Chorus operating system [1] uses this design idea. However, the idea of expanding the kernel impairs the original intention of building a small high integrity microkernel. It reduces the extensibility, flexibility, portability and reliability of the operating system. So L4ka appeared.

2.2 Second Generation Microkernels

After the failure of the Mach operating system, scientists began to redesign the structure of the kernel. Prof Dr. Jochen Liedtke invented his first microkernel with low overhead in passing messages, L3. The L3 kernel directly passes the message between processes leaving the process security and authentication to user space servers. This design method greatly reduced massive IPC overhead which occured in Mach system. On the same system where Mach required 114 microseconds for even the smallest of messages, L3 could send the same message for less than 10. The overall time for a system call was less than half the time on Unix, as opposed to Mach where the same system call took five times longer that of Unix [3].

After successfully implementing L3, Liedtke designed L3 more comprehensively. The result was a more flexible kernel, L4. A basic idea of L4 is to support recursive construction of address spaces by user-level servers outside the kernel. The kernel only does three address space operations: Grant, Mapping and Unmapping. Liedtke's design was as follows [3]: the owner of an address space can assign any of its pages to another space, provided that the recipient agrees. Similarly, the owner of an address space can map any of its pages into another address space, provided the recipient agrees. The owner of an address space can also flush any of its pages. The flushed page remains accessible in the flusher's address space, but is removed from all other address spaces which had received the page directly or indirectly from the flusher.

The Mach microkernel had a limitation of implementing the external pager policy outside the kernel. And now, this limitation is largely removed by L4's address space concept. This mechanism implements some protection schemes and physical memory management on the top of the kernel. Grant and map operations need IPC, since they require an agreement between granter/mapper and recipient of the mapping. So cross-address-space communication, also called inter-process communication (IPC), must be supported by the microkernel, which gives extra overhead to Mach system. L4 uses many methods and techniques to reduce the IPC overhead. Liedtke improved the performance of the system and reduced the overhead of IPC by redesigning the kernel. The result is positive. One RPC cost L4 only 10µs, in contrast to 230µs in

Mach and 20µs in Unix. IPC is not a burden to L4 any more. Tests of a Linux kernel ported to run on top of L4 and another ported to run on Mach (MkLinux) and the basic Linux system itself showed clear performance gains with L4. Even in the best case MkLinux was 15% slower than the monolithic kernel, whereas L4 was about 5-10% slower [16].

Minix3 is another second generation microkernel system which is developed by Andrew Tanenbaum. Minix3 was redesigned to be a pure microkernel system from its early version Minix2. Like other microkernel systems, device drivers, process manager, file system and memory manager are all implemented outside the kernel. Recently, there is another radical idea of designing a kernel. In the exokernel concept [7], the operating system only provides manipulating the raw hardware. The kernel only takes charge of securing the hardware and controls it. The application-level libraries and servers can directly implement traditional operating system abstractions. There are already some exokernel operating systems in experimental stage, such as XOK that is implemented by MIT research group and also Nemesis, written by University of Cambridge, University of Glasgow, Citrix Systems, and the Swedish Institute of Computer Science.

2.3 Motivation and Project Aims

The Mach operating system is considered to have poor performance by many people, because of the overheads in IPC. There are many kinds of other microkernel systems that are claimed to boast better performance such as L4ka and Minix3. The project's topic is to use several user application tools such as Unixbench which run on different operating systems to evaluate their performance and to discuss the practicality of new generation microkernel operating systems. The project also uses monolithic operating systems for comparison, because most current operating systems are based on monolithic kernel. Linux is a typical monolithic operating system we can use. Comparing the test results of microkernel-based systems to the test results of Linux could be a good source of discussing the practicality of microkernel operating system.

The comparison has been done by using the benchmark tool Unixbench. Minix3, a pure microkernel operating systems has been used for testing. Minix3 is a developing microkernel system, which could be a good option, and also Minix3 supports POSIX [6]. It is well developed and it can be installed easily. Minix3 has C compiler and Shell like Unix. Therefore, it is easier to implement and run application test programs on Minix3 than implementing test programs on L4. Furthermore Minix3 is written by C and Minix3 is a clearly structured microkernel system, so we could clearly know how microkernel system works after reading the source code of Minix3.

This research is designed to compare and evaluate the performance of two different operating systems by using OS benchmark tool Unixbench. There are many benchmarks Unixbench can provide, such as system call overheads, context switch overhead, file read/write throughput, pipe throughput, arithmetic performance and so on. Even more accurate evaluations were done by writing test programs which could test single IPC time or single system call time. We could analyze the result and discuss the practicality of microkernel from the evaluation results.

Linux and Minix3 were used in the testing. Minix3 is a mature microkernel based operating system. It is well structured and could easily be installed. Also Minix3 supports POSIX. It is Unix-like operating system. Linux is currently one of the most popular monolithic operating systems; it is open source and also supports POSIX. Linux is used widely in many fields. So Linux is a good option for testing. L4Linux is an operating system which runs with the L4 microkernel on the bottom level, and with Unix-like application runs on top of L4Linux. The reason not to use L4Linux is there are few research documents available for L4Linux. It is hard to get started and installed. Besides that, because the Linux kernel is in the middle between application level and L4 microkernel in L4Linux architecture, the result may not be accurate. The user application program is not implemented directly on L4 microkernel. So accurate results required implementing Unix-

like user application level directly on the L4 microkernel. It is required to work at all levels of abstraction from the bare machine to the application layer, which is a big challenge for researchers. Also, the exokernel is now in the experimental stage, and there are not sufficient documents and resources on the exokernel, and therefore the exokernel is not included in testing.

Second generation microkernels like L4ka had proved that the microkernel system could perform as well as monolithic kernel system. L4Linux on AIM benchmarks report a maximum throughput which is only 5% lower than that of native Linux. However, it is hard to compare pure L4 system with Linux, because it is difficult to implement user level application on L4 kernel. For native Linux, AIM measures a maximum load of 130 jobs per minute. L4Linux achieves 123 jobs per minute, 95% of native Linux. The corresponding numbers for user-mode L4Linux are 81 jobs per minute, 62% of native Linux, and 95 (73%) for the inkernel version. Averaged over all loads, L4Linux is 8.3% slower than native Linux, and 6.8% slower at the maximum load. This is consistent with the 6-7% we measured for recompiling Linux [4]. L4 is designed for optimizing the IPC overhead and context switch between processes, so the user level application implementation is poor. The performance of operating system is not only the kernel performance, but also the application layer performance, which is directly to users. So evaluate the performance of a relatively mature microkernel system is meaningful to microkernel system.

The project designed does not only to make evaluation benchmarks for microkernel system, but also would like to analyze the benchmarks of microkernel system and compare it to monolithic kernel systems. By that, we would like to outline a much clearer performance figure of microkernel system such as Minix3. From previous papers, IPC overhead was complained most in microkernel system, which was seen a biggest flaw affecting microkernel system performance. However, we believe not all the performance differences in microkernel system are due to the heavy IPC overhead. There are many different ways in implementing kernel and user layer between Minix3 and Linux, therefore it is important to find out which part of benchmark differences are due to the different system implementation. From analyzing the benchmarks, we try to separate performance results that are caused by different system implementation from the results that are inherited due to the heavy IPC overhead. The discuss on performance evaluating results are meaningful, because it makes a scrutiny figure on Minix3 microkernel performance and could give advices that which part of system could be improved by tuning the microkernel system.

3. Evaluation Environment and Equipments

3.1 Hardware Environment

The result of the test has to be accurate and correct. The selection of hardware is important. The entire test has to work on the same hardware, thus the hardware selected must be a common one which is supported by all the system kernels (Minix3, Linux). Minix3 supports many kinds of hardware: 386, 486 and Pentium and so on. To install Minix3 requires: Intel 386 or higher with 4 MB of RAM, an IDE hard disk with 100 MB of free disk space and an IDE CD-ROM for booting [5]. Linux also could support IA32 (Pentium). So a computer with Pentium or higher is a good choice. The RAM has to be 256MB or higher. PC must have IDE CD-ROM, VESA compatible VGA, PS/2 keyboard and PS/2 Mouse.

The configurations of the hardware machines are listed as the following:
Central Process: Pentium with 800MHz speed;
Random Access Memory: 256MB RAM;
Hard Disk: IBM DTLA-307020 20GB ATA hard disk;
CD-ROM: 24X IDE CD-ROM
Accessories: VESA compatible VGA, PS/2 keyboard and PS/2 Mouse

3.2 Minix3 Environment

The version of Minix3 used for evaluating is 3.1.2a, which is claimed to be a stable version of the Minix3 operating system [9]. To install, download the compressed CD image of Minix3.1.2a from official server in Minix3 home website, then decompress CD image and burn it into a writable CD. After that, boot computer from CD-ROM. Set Minix3 boot in regular sequence, because we have more than 16MB RAM. The following are the configurations of Minix3 in installation:

Keyboard Standard: US-Keyboard Standard;
Ethernet Chip: None;
Full distribution: Yes (requires 1GB space);
Size of "/home" directory: 2GB;
Data Block Size: 4-KB per block;

For easier implementing testing programs and benchmark tools on Minix3, extra software packages should be installed on Minix3 system. These software packages can be downloaded from the Minix3 home website or installed directly from Minix3.1.2a installation CD. Because Linux uses the compiler GCC to compile test programs and the benchmark tool, the GCC software package was installed in Minix3. Moreover, because Linux uses the Bash shell to execute programs, the shell Minix3 used should be identical with Bash. Consequently, the Bash 3.0 software package was also installed in Minix3 system. If Minix3 used its compiler CC compiler to compile programs and the default shell ash to execute test programs, it will make an inaccurate performance benchmarks. Linux uses compiler GCC to compile programs and uses Bash shell to execute test programs. There are sufficient hard disk spaces for storing, so both the package software binary distributions and their source codes are install in Minix3.

Minix3 has a version of the X window software package (X11 R6.8.2) which provides a window display for Minix3. However, the Minix3 X window system was not installed in Minix3 in the testing, because the X windows software is not a crucial part for system performance evaluating. Furthermore, due to the way Minix3 memory management works, running X window could lead a program to fail because it runs of out of memory. "chmem" command should be used to provide sufficient stack space for the program. The memory of X window binary usually set to a very large number, which often could result in X window not starting. The hardware has 256MB memory which is not sufficient for running the X windows as default setting. So "chmem" should be used for giving the sufficient stack spaces. The higher memory X window consumed the less free memory will be available for other application programs [10]. That means the system performance of Minix3 could be greatly deteriorated if running X window software on the system.

3.3 Linux Environment

Fedora core 6.0 and FreeBSD 6.0 were used in performance evaluation at the initial stage of the research. The ISO images of Fedora core and FreeBSD could be downloaded from AARNet. Fedora core is an RPM-based Linux distribution. It is well developed and widely used around world, which is a typical monolithic kernel operating system and POSIX-compatible with 7000 software packages. Therefore, Fedora core 6.0 is a suitable Linux distribution system to be used for system evaluation. The Fedora core involved in research was installed with X window software.

FreeBSD is also an Unix-like operating system. It is similar to Linux and also is a typical monolithic kernel operating system. Many software packages are identical with those of Linux. FreeBSD is totally free for the user, and it also provides binary compatibility with other Unix-like operating systems, including Linux, which means programs running on Linux could also run well on FreeBSD without any modifications. It is as reliable and robust as Linux. In this research, FreeBSD 6 was installed for testing.

At the initial stage of the research, Linux, FreeBSD and Minix3.1.2a were used in the system performance evaluation. As the research went along, we found the performance results were very similar between Linux and FreeBSD. There was only 5%-10% difference between Linux and FreeBSD in system call overhead. There is about a 5% difference between Linux and FreeBSD on pipe throughput overhead. The shell and software packages are almost the same in Linux and FreeBSD. At the middle stage of the project, we decided to stop evaluate the performance of FreeBSD. The following reasons are why we did that:

1. The purpose of the project is to obtain benchmarks of microkernel system and monolithic kernel system and discuss the practicality of microkernel system to see whether it could be more reliable and sophisticate as well as current monolithic kernel system. Performance comparison between Linux and FreeBSD does not make any sense for the intention of the project. Furthermore, there are already many benchmarks and benchmark tools for the performance evaluation between those Unix-like monolithic kernel systems. Many documents about benchmarks and performance evaluations could be found in the Internet.

2. The performance evaluation results are very similar in Linux and FreeBSD. There is only 5%-10% difference between Linux and FreeBSD in system call overhead. It is about 5% difference between Linux and FreeBSD on pipe throughput overhead. The structure of the kernels in Linux and FreeBSD are similar. Both of the FreeBSD and Linux are designed as monolithic kernel system, so performance measurement between two monolithic kernel systems certainly will give a similar result.

3.4 Benchmark Tool

The selection criteria of benchmark tool were not complex. The benchmark tool should be able to run correctly on Minix3 and Linux, and gives accurate benchmarks for the system. After carefully reviewing through benchmark tools such as LMbench, Unixbench and Ubench, we finally decided to use Unixbench 4.0.1 [11] to test the system performance of Minix3 and Linux. The reason not use LMbench is that LMbench requires specific header files at time of configuration. The header files only could be found in Linux system or other mature Unix-like system. Minix3 does not have these header files; therefore it will cause compiling failure during install LMbench on Minix3.

Unixbench is another system benchmark tool like LMbench. Unixbench gives performance benchmarks on many aspects of operating systems. Unixbench is a simple portable and POSIX microbenchmarks tool. Unixbench can give operating system benchmarks such as Dhrystone, system call overhead, file system performance on Write/Read/Copy, pipe throughput, context switch, shell script running, arithmetic test, compiler performance and so on. Thus, Unixbench can give a comprehensive figure of system performance on Minix3 and Linux. The main idea Unixbench use to evaluate performance of operating system as follows: On each system, a constant running time is given. Then, an infinite loop running test program is started, and a global variable is used to record the number of loops. When the time is up, a signal interrupts the loop and records how many times the test program runs. Obviously, an operating system which has higher efficiency could run more loops than that with lower performance. The more loops run the better system performed. For example, we use Unixbench to evaluate process creation on Linux:

1. At start, the test program is set to run 10 seconds. “signal(SIGALRM, func)” is used for setting an interrupt function handler. “alarm(10)” will sign a signal after 10 seconds.
2. Use “while (1)” to run an infinite loop. In the loop, “fork()” and “wait(&status)” are used for creating process, and “iter++” counts the times program run during 10 seconds.
3. After 10 seconds, the infinite loop is interrupted by a signal sent by the “alarm(10)” system call. In the end, record the value of variable “iter”. The bigger “iter” is the better system performs in process creation.

The Unixbench only gives the number of loops run to present the performance of the system. What if people want more direct performance benchmarks? For example: microsecond time values on each test program run are wanted. Therefore, in this project we also used another way to measure the performance of Minix3 and Linux. The system call "gettimeofday()" returns the time in seconds and microseconds since epoch in GMT. So we could run "gettimeofday()" system call at the beginning and the end of test program and subtract the returned values in order to get the microseconds used in running test program. Also, taking the process creation test as an example, the following code determines the execution time of one process creation:

```
gettimeofday(time, tzone); /*get the time befor running */
        if ((slave = fork()) == 0) {
                        exit(0);
                } else if (slave < 0) {
                        exit(2);
                } else
                        wait(&status);
                if (status != 0) {
                        exit(2);
                }
gettimeofday(time1, tzone1); /*record finish time */
```

From the value of structure "time1" and "time" we could calculate the execution time of process creation for once.

4. Evaluation Results and Analysis

4.1 Unixbench Evaluation Benchmarks

At the beginning of the evaluation, we installed Unixbench on Linux and Minix3 separately first. First problem was how to transfer data into Minix3. Minix3 was not developed as well as Windows and Linux. Linux could automatically mount and unmount many file devices such as USB and CD-ROM. With X window, Linux could easily transfer Unixbench program from mobile devices (USB, CD) to hard disk. However, Minix3 could not obtain data sophisticatedly from outside devices. MINIX's primary purpose is to illustrate operating system principles. Keeping MINIX small enough to fit into a student's head during a semester- or year-long course has required keeping it simple. In particular, the MINIX file system supports mounting only media containing MINIX file systems [9]. In this research, we used command "isoread" to read the content of Unixbench from ISO-9660 CD-ROM and copied the content to local hard disk. We wrote Unixbench install program into a writable CD and use "isoread /dev/c0d2/ Unixbench.zip > /home/Unixbench/Unixbench.zip" command to copy Unixbench.zip into /home/unixbench/ directory. After that, to use GCC, we have to change the PATH to add "/usr/gnu/bin" into PATH of Minix3 system. At last, using "make" command to compile and link all the programs of Unixbench then we used "./run" to run Unixbench to get benchmarks.

The total evaluation procedure lasted for 52 minutes. Unixbench gave system evaluation benchmarks on Dhrystone, system call overhead, file system performance on Write/Read/Copy, pipe throughput, context switch, shell script running (with 8 and 16 concurrent users), arithmetic test, C compiler throughput and process creation. Because Linux uses Bash shell, which is different from Minix3's ash shell, so we installed Bash 3.0 shell on Minix3 and run Unixbench on Minix3 with Bash 3.0. The following table Table 1 in next page are benchmarks of Unixbench on Minix3 with Bash 3.0, Minix3 and Linux. Here are the terminologies for the table:

lps: loops per second
lmp: loops per minute

KBps: Kbytes per second

From table.1, we could find that performance of Minix3 with Bash shell is similar to Minix3 with the default shell. Therefore, we only compare performance of Minix3 with performance of Linux instead of comparing those three benchmarks between Minix3, Minix3 with Bash and Linux. From the table, we could see Minix3 behaves little worse than Linux in some benchmarks such as in Dhrystone, File Write, Shell scripts and Recursion Test. In some benchmarks Minix3 could perform as well as Linux did, such as Arithmetic Test in short integer. In some cases like System Call overhead, Pipe throughput, Pipe-based context switching, Process Creation, Exce throughput and Arithmetic test in float and double type. Minix3 is far slower than Linux. For example, Minix3 is about 182 times slower than Linux. Minix3 could give better performance on Shell Script running with 8 or 16 concurrent users and Arithmetic test in integer and long integer. We used bar charts (Fig 3, Fig 4, Fig. 5 and Fig. 6) to illustrate the benchmarks. The bar charts of the performance benchmarks of Minix3 and Linux are after Table. 1 (Linux as 100 marks).

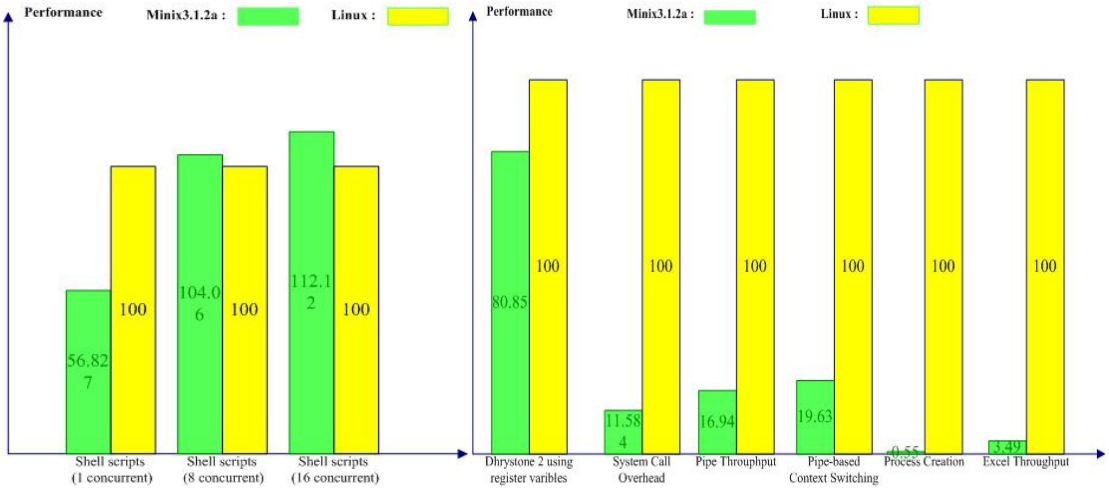

FIGURE 3: and **FIGURE 4:** Benchmarks in Shell Script Running

Benchmarks	Minix 3.1.2a running Bash-3.0	Minix 3.1.2a	Linux
Dhrystone 2 using register varibles	1944025.1 loop/s	1892225 lps	2340428.8 lps
System Call Overhead	60340.9 lps	60366 lps	521107.6 lps
Pipe Throughput	42795.4 lps	42584.1 lps	251399.8 lps
Pipe-based Context Swithing	16668.3 lps	17506.2 lps	89164.7 lps
Process Creation	30 lps	24.6 lps	4479 lps
Excel Throughput	36.7 lps	36.7 lps	1051.9 lps
File Read 1024 bufsize 2000 maxblocks	81011 KBps	80693 KBps	220582 KBps
File Write 1024 bufsize 2000 maxblocks	75644 KBps	75511 KBps	99276 KBps
File Copy 1024 bufsize 2000 maxblocks	39228 KBps	39165 KBps	65518 KBps
File Read 256 bufsize 500 maxblocks	24833 KBps	24721 KBps	81638 KBps
File Write 256 bufsize 500 maxblocks	26400 KBps	26372 KBps	42460 KBps
File Copy 256 bufsize 500 maxblocks	12674 KBps	12664 KBps	25413 KBps
File Read 4096 bufsize 8000 maxblocks	30002 KBps	29997 KBps	392921 KBps
File Write 4096 bufsize 8000 maxblocks	32800 KBps	32799 KBps	158023 KBps
File Copy 4096 bufsize 8000 maxblocks	14064 KBps	14014 KBps	109613 KBps
Shell scripts (1 concurrent)/ (8 concurrent)/ (16 concurrent)	922.5 lps / 212 lps / 112.3 lps	825.6 lps / 207 lps / 111 lps	1452.8 lps / 199 lps / 99 lps
Arithmetic Test (double) / (float)	3656 lps / 7066 lps	3656 lps / 7067.4 lps	250048.8 lps / 259010.4 lps
Arithmetic Test (short)	249705 lps	249774 lps	252042 lps
Arithmetic Test (int) / (long)	261163.6 lps / 261175.5 lps	261163.9 lps / 261159.6 lps	258079.3 lps / 258180.7 lps
Arithoh	4597523.4 lps	4598553 lps	140482502 lps
C Compiler Troughput	43.8 lpm	43.6 lpm	358.7 lmp
Dc: sqrt(2) to 99 decimal places	1721.8 lps	1424.7 lps	38247.5 lps
Recursion Test-Tower of Hanoi	27023.3 lps	27149.8 lps	37328.8 lps

TABLE 1: Unixbench Benchmarks in Minix3 and Linux

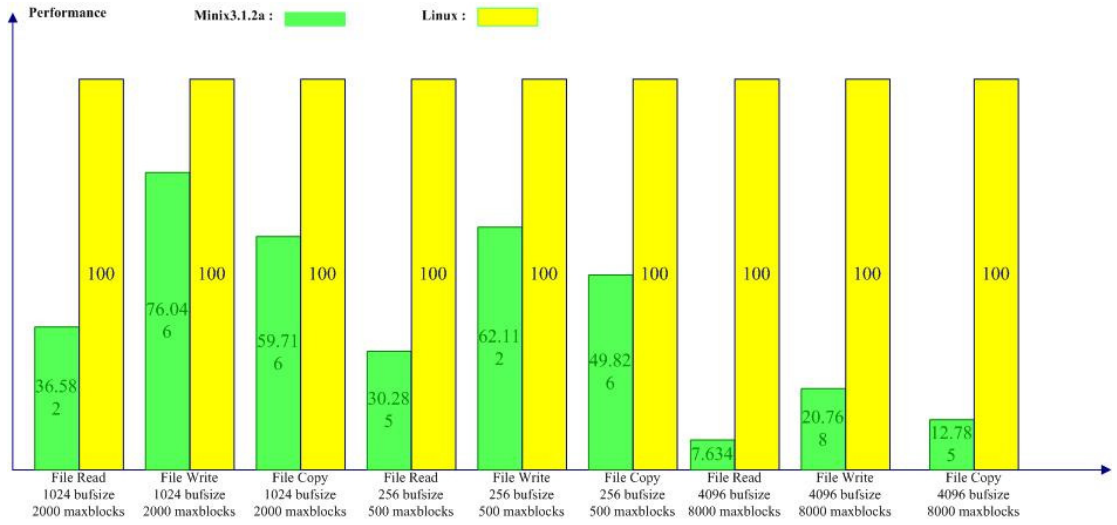

FIGURE 5: File System Benchmarks

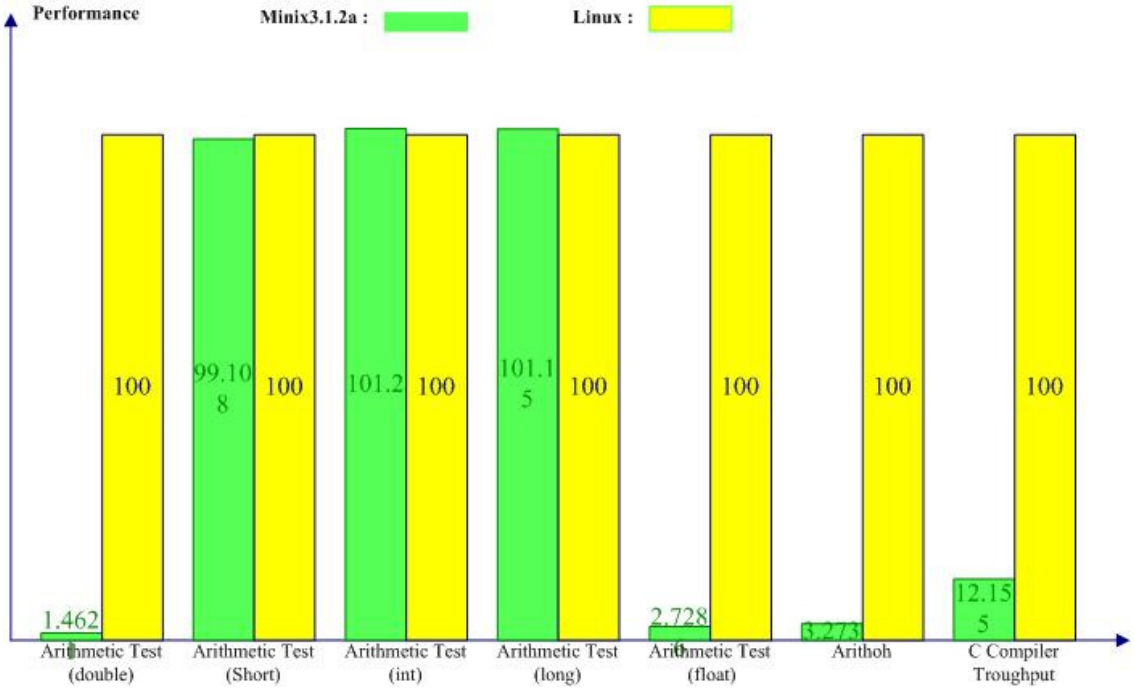

FIGURE 6: Arithmetic Benchmarks

3.2 Microsecond Level Evaluation Benchmarks

As stated in last chapter, Unixbench starts an infinite loop running a test program and uses a global variable to record the number of loops. When the time is up, it uses a signal to interrupt the loop and records how many times the test program ran. The more loops run the better system has performed. Next we want a more precise evaluation result at the microsecond level. So we ran the system call function “gettimeofday()” at the beginning and the end of test program and subtracted the returned values in order to get the microseconds used in running test program. In Unixbench, the main program to evaluate system call overhead is as the following:

```
while (1) {
        close(dup(0));
        getpid();
        getuid();
        umask(022);
        iter++;
}
```

When the time is up, the signal sent from kernel will interrupt the infinite loop and the value of iter will be recorded. For testing the real execution time on system call overhead, we run "gettimeofday()" system call at the beginning and the end of the program. So the main program codes are the following:

```
gettimeofday(time, tzone); /*get the time befor running */
for ( i =0 ; i<1; i++)  /* do test program for just once */
{                       /* copy from Unixbench */
        close(dup(0));
        getpid();
        getuid();
        umask(022);
}
gettimeofday(time1, tzone1); /*record finish time */
```

Ran the test program on Minix3 and Linux separately. When the test program ran for one iteration, the result could not be measured on Minix3 because the system is too fast. So we increased the running loops of test program. When we run test program for 1000 times, the execution time was get in Minix3. The following table are the evaluation results:

	Minix3					Linux				
Iterations (i)	1	10	100	1000	10000	1	10	100	1000	10000
Time (Microsecs)	None	None	None	15,333	166,666	23	45	220	2,000	22,000

TABLE 2 : Benchmarks on microsecond level

From Table 2, we could see running system call test program for 1000 iterations cost Minix3 15333 microseconds and cost Linux 2000 microseconds. In 10000 iterations, Minix3 spent 166666 microseconds, whereas Linux use 22000 microseconds. In 1000 iterations case, Minix3 is about 15333/2000 = 7.6 times slower than Linux. In 10000 iterations case, Minix3 is 166666/22000 = 7.58 times slower than Linux. We back to the evaluation result Unixbench got; Minix3 is about 521107.6 lps/60366 lps = 8.6 times slower than Linux. The results of Unixbench and the results of microsecond level evaluation are similar. The purpose of the project is to evaluate the performance of microkernel operating system, which could define whether microkernel operating systems are as practical as current monolithic operating systems. Then the approach was to analyze the benchmarks and separate performance results that are caused by different system implementation from the results that are inherited due to the heavy IPC overhead. Therefore, we could only consider the benchmarks Unixbench gave, if the microsecond level results are similar to the benchmarks Unixbench gave.

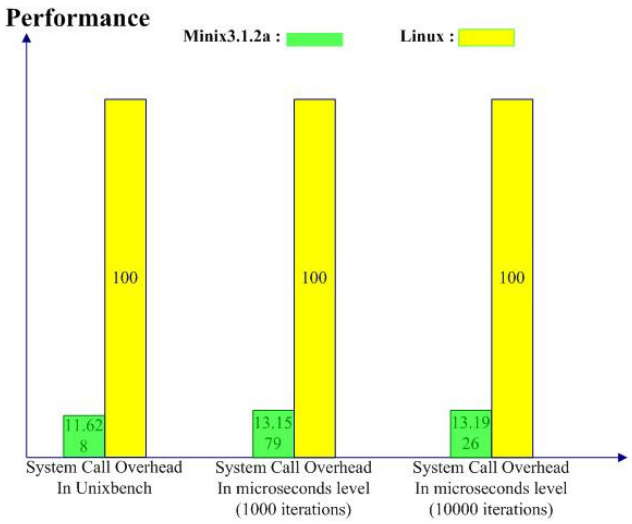

FIGURE 7: Similar Results in Unixbench and Microseconds Level

5. SUMMARY AND CONCLUSION

5.1 Summary of the Conclusion

The project used micro-benchmark tool Unixbench to measure the overall system performance of Minix3.1.2a and the performance of Linux (Fedora Core 6.0). The evaluation test was done in user application layer of the systems. In this way that we could testify the second generation microkernel operating systems whether could be as flexible, portable and secure as monolithic operating systems like Linux. Unixbench gave many benchmarks on MINIX3 and Linux, such as system call overhead, pipe throughput, arithmetic test and so on. The result shows MINIX3 has better performance on Shell Scripts running and Arithmetic test and Linux has better performance on other aspects such as system call overhead, process creation and so on. Linux gave a better performance than Minix3 on the overview of benchmarks. However, after analyzing the benchmarks Unixbench gave, we realized that many benchmarks such as process creation, floating point arithmetic test and Arithoh test in Minix3 could be optimized by system tuning. The following list is the summary of the Unixbench benchmarks discussions:

Process Creation: The benchmark shows that Minix3 is about 182 times slower than Linux. However, COW (Copy-On-Write) technique is a main reason that causes the big performance difference in process creation benchmarks.

Floating Point Arithmetic Test: The benchmark shows that Minix3 is about 68 times and 36 times slower than Linux in double and float data type respectively while Minix3 perform as well as Linux in integer arithmetic test. The reason that causes the poor performance on floating point arithmetic test is Minix3 does not support Floating Point Unit (FPU) that is integrated into CPU. Minix3 used software to emulation the floating point operations, which is much slower than hardware floating point operation supported by Linux.

Shell Script Running: In Unixbench's benchmarks, Minix3 performed better than Linux in shell script running in 8 and 16 concurrent users cases. Because Minix3 does not have virtual file system, so the buffer cache implementation in the systems would be different from Minix3 to Linux. Linux with virtual memory system may require more CPU cycle on page allocation than Minix3.

5.2 Future Works

At the middle of this project, a new experimental version of Minix3 was released. Minix3.1.3, an experimental version of Minix3 which was released at 13th April 2007. A few changes were made

in Minix3.1.3, such as enlarged file system, adding virtual file system, new boot procedure and so on. Notice that virtual file system was imported to Minix3; therefore there must be a big change in file system in Minix3. So we should measure the performance of new Minix3 virtual system again using same benchmark tool and benchmarks in Linux. Measurement on process creation may give us a different benchmark compare with the current benchmark. Another benchmark we should focus on is the shell script running. In current benchmarks, Minix3 gave better performance on shell script running with 8 and 16 concurrent users. We guess in this report that the reason that causes better performance in Minix3 in shell script running is Minix3 does not implement virtual file system. In Minix3.1.3, virtual file system was ported on Minix3. Therefore, the benchmarks of shell script running in Minix3.1.3 could give crucial information that whether the virtual memory system is the reason causes Linux slower than Minix3 in shell script running benchmark.

Form the benchmarks in chapter 3, we could see that there are many benchmarks such as system call overheads, pipe throughput, context switch overheads, excel overheads and C compiler throughput which we did not discuss yet. Research on those benchmarks need to be done in the future. By doing that, we could make a very clear figure that on Minix3 microkernel performance and could find out that which part of system could be improved by tuning the microkernel system. Then try to optimize the system performance by tuning the microkernel system if we know exactly why Minix3 behaved such a low performance in the benchmark.

From the Minix3 official web site, we also found that some future works should be done by researchers [9]:
1. Testing MINIX 3 on different platforms
2. Porting programs and applications to MINIX 3
3. Porting drivers to MINIX 3
4. Building a driver framework to use FreeBSD or Linux drivers
5. Porting MINIX 3 to different architectures

6. REFERENCES

[l] Jochen Liedtke, "*toward real microkernels*". Communications of ACM September 1996. Vo139, No. 9.

[2] Chen, J.B. and Bershad, B.N. "*The impact of operating system structure on memory system performance*". In Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP) (Asheville, N.C., Dec. 1993). ACM Press, 1993, pp. 120—133.

[3] Liedtke, J. "*On microkernel construction*". In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP) (Copper Mountain Resort, Cob., Dec. 1995). ACM Press, New York, 1995, pp. 23 7-250.

[4] Liedtke, J. "*Improving the IPC by design Kernel*". 14th ACM Symposium on Operating System Principles (SOSP) Asheville. 1993, pp. 10-11.

[5] Andrew S Tanenbaum & Albert S Woodhull. "*Operating System Design and Implementation*" (3rd Edition). Prentice Hall Software Series. 2006.

[6] L4 Kickstart < http://www.l4ka.org/projects/pistachio/kickstart.php >. Edited by University of Karlsruhe. 2000-2006. Viewed on 24th Mar. 2010.

[7] D. R. Engler, M. F. Kaashoek, J. O'Toole. "*Exokernel: an operation system architecture for application-level resource management*". ACM SIGOPS Operating Systems Review , Proceedings of the fifteenth ACM symposium on Operating systems principles SOSP '95, Volume 29 Issue 5.

[8] Lmbench home website <http://www.bitmover.com/lmbench/ >LMbench - Tools for Performance Analysis. Viewed on 24th Mar. 2010.

[9] Minix3 home website < http://www.minix3.org/doc/environ.html> MiniFAQ about MINIX 3 Programming. Viewed on 14th May 2010.

[10] Minixtip website < http://www.minixtips.com/> Tips For Running the Minix OS Version 3. Viewed on 14th May 2010.

[11] FTP of Unixbench <http://www.tux.org/pub/tux/benchmarks/System/unixbench> Viewed on 13th Mar 2010.

[12] Daniel P. Bovet & Macro Cesati. "*Understanding the Linux Kernel*". O'REILLY Press, Nov 2005.

[13] Floating Point Unit, <http://en.wikipedia.org/wiki/Floating_point_unit>, From Wikipedia, the free encyclopaedia. Viewed on 24th Mar 2010

[14] Comparing Linux and Minix, <http://lwn.net/Articles/220255/>, LWN.net article, Viewed on 16th May 2010.

[15] Hbench-OS Operating system Benchmarks <http://www.eecs.harvard.edu/vino/perf/hbench/index.html>, Viewed on 16th May 2010.

[16] H. Hartig, M. Hohmuth, J. Liedtke, S. Schänberg, J. Wolter, "*The Performance of µ-Kernel-based Systems*", 16th SOSP TU Dresden, Fakultat Informatik, Heft Jan 1997.

[17] Ben Leslie, Carl van Schaik and Gernot Heiser, "*Wombat: a portable user-mode Linux for embedded systems*"**,** Proceedings of the 6th Linux Conference Australia, Canberra, April, 2005.

[18] ERTOS Website <http://www.ertos.nicta.com.au/research/l4/performance.pml>, National ICT Australia United, Viewed on 16th May 2010.

[19] Release Notes of MINIX 3.1.3 - Developer's Interim Release, <http://www.minix3.org/download/releasenotes-3.1.3.html>, Minix3 Home Website, Viewed on 17th May 2010.