

Mac OS X Darwin and Ubuntu Linux: Comparison of Some Common Operating System Algorithms

Software Systems Project
Will Clayton, Sharon Talbot, and Nick Zola
May 5, 2005

Abstract

For this paper, we conducted a basic numerical analysis of the Macintosh OS X Darwin kernel and Ubuntu Linux 2.6 kernel by testing several different aspects inherent to the operating system. To exercise file I/O handling we examined the repeated creation of small text files. In addition, we examined disk cache behavior by attempting to overflow the cache with junk data. To test memory duress response, we exposed the kernel to malignant processes and analyzed the resultant data. We tested this particular situation by incrementally consuming memory using `malloc` until it was shut down by the operating system. In conclusion, we analyzed the data we received from the above experiments, in a first order attempt to determine some degree of operating system superiority for the attributes mentioned.

Introduction

Comparing computer operating systems is hardly a new pastime, and UNIX is hardly a new operating system. Nonetheless, we have undertaken the feat of matching key components of two flavors of UNIX: Darwin, the underpinnings of Mac OS X (based on FreeBSD); and Ubuntu Linux, running the typical Linux kernel; both operate on the PowerPC processor. The idea for the project came about from completing group homework assignments in the Software Systems course where one team member performed tests on the UNIX side of the Mac and the second team member performed them on the school's provided distribution of Linux. Differences were apparent in the results, and we wanted to learn whether the differences were hardware or software specific. In other words, does the UNIX of a Macintosh perform equally as well as a Linux OS on the same hardware, or is one fundamentally superior? We decided to explore operating system algorithms for common computer necessities.

Specifically, we ran microbenchmarks to compare I/O performance, virtual memory page replacement, and disk cache replacement in Darwin and Linux. Each of these components are internal choices made by the operating system that the common user is not aware of, but each can have an important effect on the wait time perceived by the user. Since these were areas we had discussed in class, we were relatively confident we could devise means of testing each operating system's efficiency in them.

Of particular note in this experiment is our ability to test the two operating systems on the same piece of hardware, an ideal situation for analyzing algorithms and OS overhead irrespective of hardware power. We did preliminary testing for the experiments on a Blueberry iMac clipping at a brisk 350 MHz with 512MB of RAM, and officially ran the experiments on a brand new Mac mini with a 1.25GHz processor, also with 512MB of RAM. We were running Mac OS X 10.3.7 and Darwin 7.7.1 on the Mac side and Ubuntu Linux 2.6.8 on the Linux side. This computer was recently purchased by the Olin College Advanced Computing Lab, for which we are very grateful.

In order to prepare for this study we also researched what claims were made about each operating system and how they differ, what replacement and scheduling algorithms each operating system was published as using, and any prior comparison data that had been collected in this area. Interestingly, not many resources evaluating Darwin against Linux were found, indicating perhaps that we are breaking new ground. One possible reason for this is Darwin's relatively new appearance in the OS world as a child of FreeBSD, compared to Linux's longtime, widespread adoption.

The rest of this paper is organized as such: the following section is our synopsis of any prior research and comparison data we could find featuring Darwin and Linux. Then we discuss the published differences between Darwin and Linux and any unique OS features related to our experiments. The rest of our paper is spent discussing the specific experiments we ran, their results, and the conclusions we drew from those results.

Prior Research and Related Work

Researching existing comparisons between OS X Darwin and Linux yielded some interesting findings. Though very limited quantitative comparison of the two systems could be found, there was one study comparing their usefulness as server operating systems that was released soon after OS X's initial distribution. As of October 2002, Linux's Virtual Memory and I/O handling were generally much faster than OS X's. During this testing, OS X's I/O performance was 33% worse than Linux's.ⁱ Moshe Bar, an OS researcher who writes for Byte.com, suggests that these characteristics limit the appeal of running OS X on a server. He also suggests that these results were possibly due to Apple's choice of FreeBSD instead of Linux as the foundation of OS Xⁱⁱ, and that since OS X is new to the UNIX market, its speed may improve with time. Now over two years later, we have the opportunity to test that theory. This data, however, is not as relevant as it once was. At the time of the article, the 2.6 kernel was not widely adopted, and was not used for this test. Also, we expect OS X to have improved since its earlier versions.

Darwin and Mac OS X

Mac OS X, although the product of a well-established computer company, is a relatively new OS (first released in 2001). Apple claims in the boilerplate of all of its press-releases that it "ignited the personal computer revolution" with the Apple II and then "reinvented" it with the Macintosh. Apple may not actually be "leading the industry in innovation," but it has made some very brave steps with the release of Mac OS X. Its distinguishing feature is being a UNIX-based OS with an open-source kernel on a commercial operating system, referred to as Darwin in the Macintosh community. Darwin is a variant of FreeBSD 5.0, on which its Mach 3.0 microkernel operating system services are based. Apples touts that Darwin is "rock-solid" and "engineered for stability, reliability, and performance."ⁱⁱⁱ It's important to note that with the debut of Mac OS X Apple rendered all of its previous OS versions (Mac OS 9 and below) and applications incompatible, forcing an entire new set of software and drivers to be developed from the ground up.^{iv} This was a large price to pay when there was already a limited market share and user base, and programmers were hesitant to write new Mac software. But Apple has rarely been afraid to take risks in their perceived direction of the future. They claim to be the "first major computer company to make open source development a key part of its ongoing operating system strategy."^v Corroborating Apple

claims is not this paper's focus (despite what a revealing paper that might be), but it is clear that Apple considered the disadvantages of rewriting their operating system a small price to pay for the advantages of having an open-source, UNIX-based kernel that was founded on stability and reliability to work with for the future.

Mach is the microkernel at the center of Darwin, managing processor resources such as CPU usage and memory, handling scheduling, and enforcing protected memory. With preemptive multitasking, Mach ensures processes share the CPU efficiently, prioritizes tasks, and keeps activity levels at a maximum.

Mach controls the virtual memory, giving each 32-bit application its own 4GB virtual address space. In Mac OS X virtual memory is always on; it cannot be turned off, as in previous versions of Mac OS. Additionally, unlike other UNIX operating systems, Mac OS X uses all the available space on the hard drive's boot partition for virtual memory rather than a preallocated swap partition. The page size is 4K in both Mac OS 9 and X. According to Apple Mach augments page-replacement semantics with the abstraction of virtual memory objects. Each region of the virtual address space is associated with a virtual memory object, used to track the resident and nonresident pages of that region. Memory objects enable one task to map a range of memory, unmap it, and send it to another task.^{vi,vii}

Mach boasts a protected memory feature that keeps individual processes from bringing down the whole system if errors occur, a significant improvement over the previous Mac OS. Mach also enables cooperative multitasking, preemptive threading, and cooperative threading, among other features.^{viii} Mach was developed at Carnegie-Mellon University between 1985 and 1994 as a UNIX-based system capable of inter-process communications.^{ix}

Incorporated in Mach is FreeBSD 5.0, a customized version of the BSD operating system. While Mach handles memory and scheduling, BSD runs the file system and networking of Mac OS X, as well as programming interfaces such as basic security features, the process model (process IDs, signals, etc.), and threading. BSD includes much of the POSIX API.^x

Mac OS X features the I/O Kit for device driver support, written in a restricted subset of C++.

Linux and Ubuntu

Linux is a long-standing, well-established operating system, the history of which is probably more commonly known than Mac OS X. Originally developed by Linus Torvalds as an extension of Minix, Linux is best known for its open-source model. The first version was released to the internet in 1991, and has since been modified and updated by thousands of developers worldwide. Ubuntu is running the Linux kernel 2.6.8 under its version label Hoary.^{xi}

Linux's virtual memory management principle is page aging. Pages age down and are deactivated when they reach zero, but processes age pages up (make them younger) by accessing them. When pages are deactivated they are added to the inactive list and kept relatively in LRU order, though the kernel won't waste too much time on this. The kernel will take the first clean, inactive page available when a page is needed. Pages age down each pass more than they age up by a single process access, thus processes must access pages about twice as frequently as they are aged down in order to

keep them active.^{xii} New additions to the 2.6 kernel include reverse mapping of pages to facilitate page table updating before swapping, larger page sizes to reduce the number of TLB misses, and storing page tables in high memory to save low memory for kernel use.

Going into the experiments, we did not have good background information on file I/O or disk caching algorithms in Linux. Hence the experiments we ran were designed to fill this gap in our knowledge, or at least to allow us to make comparisons between the two operating systems, if not to divine the algorithms themselves. We felt that these tests would be of value in our comparison of Linux to Mac OS, as these operations are done on a very regular basis in typical computer use.

Experiments

We attempted to test both operating systems under their best conditions; thus we used the following control methods:

Experimental Conditions – OS X

Our original desire was to eliminate as much of the Mac OS X overhead as possible in order to test purely the operating system level algorithms without the distractions of the Aqua GUI or the other Darwin application daemons. The Mac OS does not provide an effective way to eliminate the Finder (the file system abstraction) or some of the other natively running applications. Our solution, so we thought, was to start the computer in single-user mode, which loads directly into the command line and is normally reserved for troubleshooting and repairing the disk. In fact, the disk begins mounted as read-only, and we were forced to run a series of complicated commands in order to turn on write privileges and start the basic *init* processes. Once we thought we had everything in order, we ran our memory experiment first, only to discover after much frustration that in single-user mode there is no default memory manager. In other words, the pager is not turned on and any process that runs out of pages will hang forever because no other processes will be willing (or know how) to give up their pages. Essentially we learned that it is unfair to call single user mode an operating system without the GUI; they are not meant to be separated and any testing should include the whole system. Thus we resolved to run our experiments in the normal Mac OS X environment, as this is what the typical user is doing anyway.

Experimental Conditions – Linux

The Mac mini's Linux partition was running Linux Hoary with 2.6.8 kernel. We killed a few unnecessary daemons, such as cups, and ran it in a terminal without X11.

Memory Exercises Regarding Hostile Processes

Experiment:

The memory exercises that we ran were designed to find out the responses of the operating systems in question to an excess of memory usage. We ran two main functions concurrently, one that would use memory in a sensible manner (that is, it would request memory from the operating system, use it, and then render it invalid again), and also a process that we considered to use memory in a more voracious manner. This process, dubbed the “evil” side, would request more and more memory, in regular intervals with the function we dubbed “good.” The interaction of these two functions was intended to

help us discover the performance of the memory management routines under duress.

Results:

The Linux side of the memory performed as we expected. That is, when the operating system ran out of both physical and virtual memory available to the process it killed the process in question without killing any system imperative processes. We referred to this as the Linux “eye-for-an-eye” policy. In Mac OS, however, the operating system was, in a way, much kinder to the process in question. Rather than simply killing the process when the system ran out of available memory, the operating system simply denied it any additional memory. This did not appear to be terribly useful to us, as the process no longer actually continued to be practically active, since it was denied additional memory every time it requested and thus could not crash as was expected.

The operating system did not appear to schedule the process any less frequently. We referred to the Mac OS policy as the “golden rule” policy. That is, it does not actually kill any process. Even if it doesn't “like” the process, it still treats it as if it has full “friend” status. These very differing policies were fascinating to examine, although they did cause havoc with our data gathering methods.

The results from the memory experiment in Linux are shown in Figure 1. Along the x axis is the growing amount of requested memory for allocation by the process. Along the y axis is the actual memory in the system. Total memory is a constant 512 MB; used and active memory increase while free memory decreases; and inactive memory increases slightly but is relatively constant.

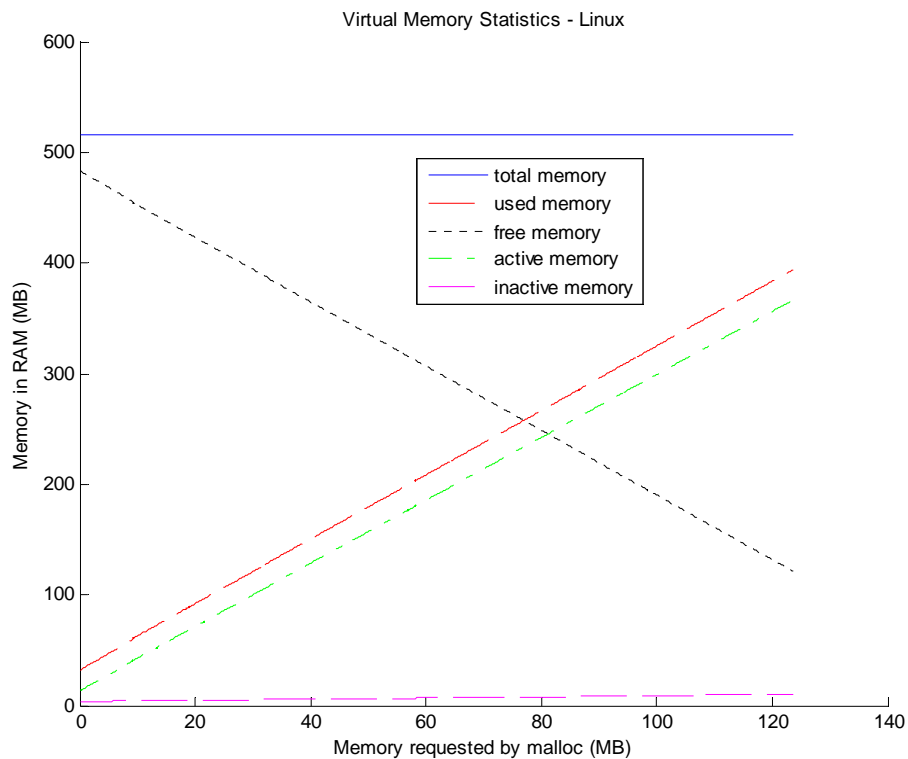


Figure 1 – The virtual memory statistics from the memory test run in Linux.

File I/O

Experimental Design

We wrote a C program to test the ability of both operating systems to handle a large numbers of file I/O operations. The program code is included in Appendix I. Of specific interest was the amount of operating system overhead involved in creating a large number of identical text files and writing them to the hard disk.

The program creates a file and writes "I love PPC" to it, then syncs the disk to write the file; this action was performed 10 times. Following this sync, the next iteration of the process performs this operation 25 times sequentially, re-writing the file after each sync. Again, 10 trials were conducted prior to incrementing the number of files operations by another 25; this process continued iterating until the test file was re-written 975 times. We ran the experiment from 0 to 975 numbers of files, in increments of 25, to verify whether the operating system's I/O handling behaved differently for differing amounts of files to be written.

Results

The data from this experiment, shown in Figure 2, are very telling. Darwin performed consistently slower than Linux, on identical hardware. Operating system differences are clearly the cause of this variation. Other than slow or inefficient code somewhere within the I/O file handler, other possible explanations include the base system processes for Darwin taking up significantly more processor cycles than the Linux ones. This is likely because Darwin simply cannot run at as low a level as Linux can. It appears to be a more graphically-oriented operating system in general, thus requiring the extra processor time to keep all its superfluous pixels in line.

Linux won this test hands down. The most interesting implication of this test is that it suggests Linux is significantly faster than Darwin when a large number of small I/O operations are involved, as is the case with many database operations.

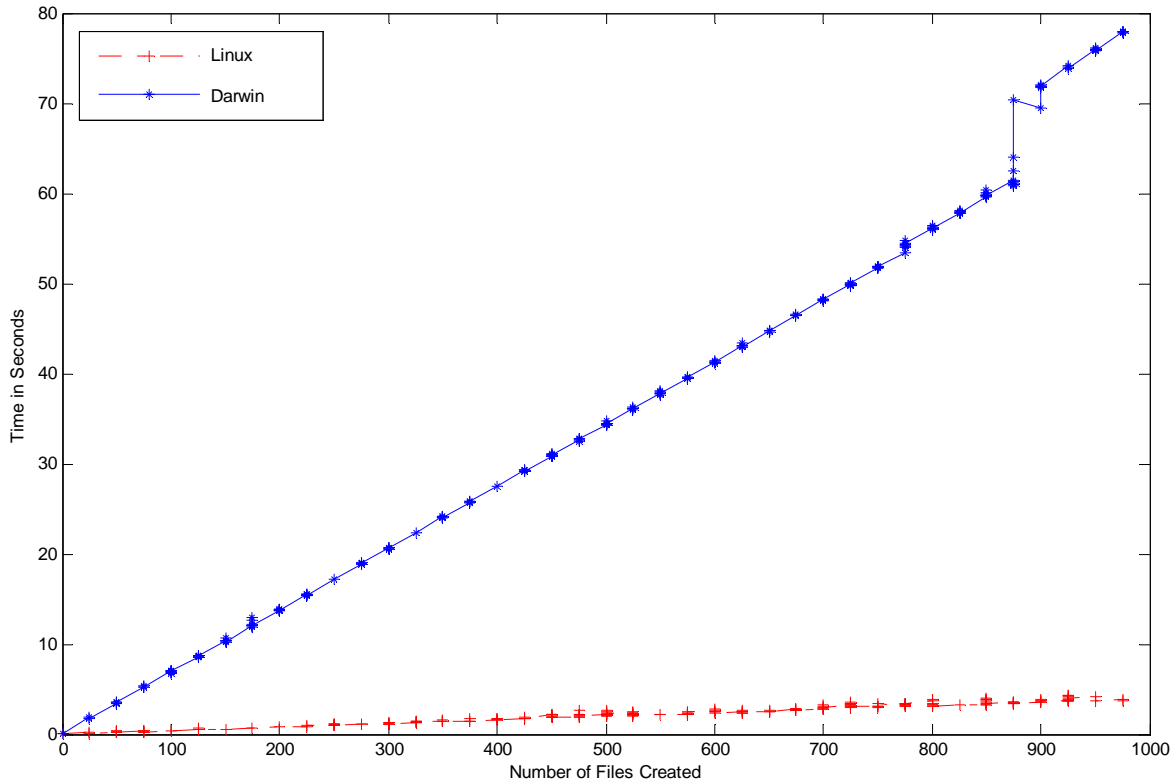


Figure 2 – Time Needed for a Process to Created a Given Number of Small Files and Flush the Buffer

Screen I/O

Experimental Design

To test screen input output speed, we designed a C program that would exercise handling abilities of both Darwin and Linux. This program tested the amount of time the OS required to print a given phrase, such as “I love PPC”, to the console several thousand times. An alternative version of this code printed an incrementing integer several thousand times.

Results

This testing was only conducted in Linux and, as should have been anticipated, did not yield useful data and so was not formally tested on the Mac mini. Once a call is made to an output function, e.g. `printf()`, then I/O controllers are given power over the completion of the operation in question. The calling function, in this case a custom-built C program, hands complete control over this operation to the I/O controller and the operation joins the I/O queue without any distinct concept of where it originated. It was therefore not possible to accurately track the amount of time needed for each batch of screen I/O operations; the OS largely abandons control over input/output operations once they are initially called by the active process.

Disk Cache Replacement

Experimental Design

We choose to focus on disk cache replacement because memory and CPU cache replacement policies are controlled by hardware, not software. Brief evaluation of OS X

features and other research suggests that Darwin operates with a disk cache of roughly 10MB.^{xiii}

The relevant C code is included in Appendix II. The program operated by opening a small fixed-size text file (`fileyay.txt` opened using `Tester.c` code), opening a number of identically sized MP3 files (managed by `memclear.c` code), and finally reopening the text file. The objective was to discern the number of intervening files that would need to be loaded before the text file was paged out to disk and removed from the disk cache. The number of incrementing files used to knock the main file out of cache ranged from 1 to 1000; a loop began by reading one file and continued incrementing upwards by 1 until 1000 files were read. Three trials were made of this looping. Each file was 1.22MB. Data was gathered using the UNIX `time` command, which was called on the `fileIOoperation` process that is described in the appendix. It was this process that actually opened, read, and closed `fileyay.txt`.

Results

The user and system times provided by `time` were generally consistent throughout all testing for both Linux and Darwin. However, the real time bore a notable relationship to the size of files read prior to reading the `fileyay.txt`. Therefore, when we refer to “access time” below we refer to the actual amount of time that passed while `fileIOoperation` was running.

Figure 3 shows the access time required to run our `Tester` program after a given number of junk files had been read, averaged over 5 data points. For a low number of files, both Linux and Darwin performed relatively consistently with Linux averaging 2 millisecond access times and Darwin averaging roughly 7 millisecond access times. At slightly more than 350MB worth of files, Darwin jumps abruptly to more than 65 milliseconds per access. It then fluctuates between roughly 65 milliseconds and 85 milliseconds throughout the remaining 870MB of files.

In contrast, Linux continues with the 2 millisecond average access time until slightly more than 475MB of files are read, then it begins to slowly climb towards a 35 millisecond access time at roughly 595MB at which point it spikes upward rapidly to more than 70 milliseconds and then continues to steadily increase over the remaining 600MB of file read, reaching highs of more than 100 milliseconds near 1100MB.

Again, Linux won these tests, but the results were slightly more mixed than the I/O testing discussed above. Because we know that main memory on the Mac mini was 512MB, we hypothesize that the Darwin performance drop at 350MB was likely due to the larger number of associated system files that needed to be kept in main memory for the operating system and its processes to function properly; this value was likely much higher for Darwin compared to Linux because, as discussed above, Darwin continued to operate its GUI mode throughout testing. It is also possible that even if the kernel were not actively holding such a large chunk of main memory for its own use, it may have been limiting the amount of main memory that could be occupied by caching for a single process.

Though Linux did retain very high performance throughout a larger testing band, access times under Linux continued to increase at a reasonably fast rate as the amount of files being read increased after 600MB. Therefore, for a small range of operations which involve large information transfers and then reading uncached data, Darwin appears to be

a better option than Linux because it is relatively consistent in the time required for such operations.

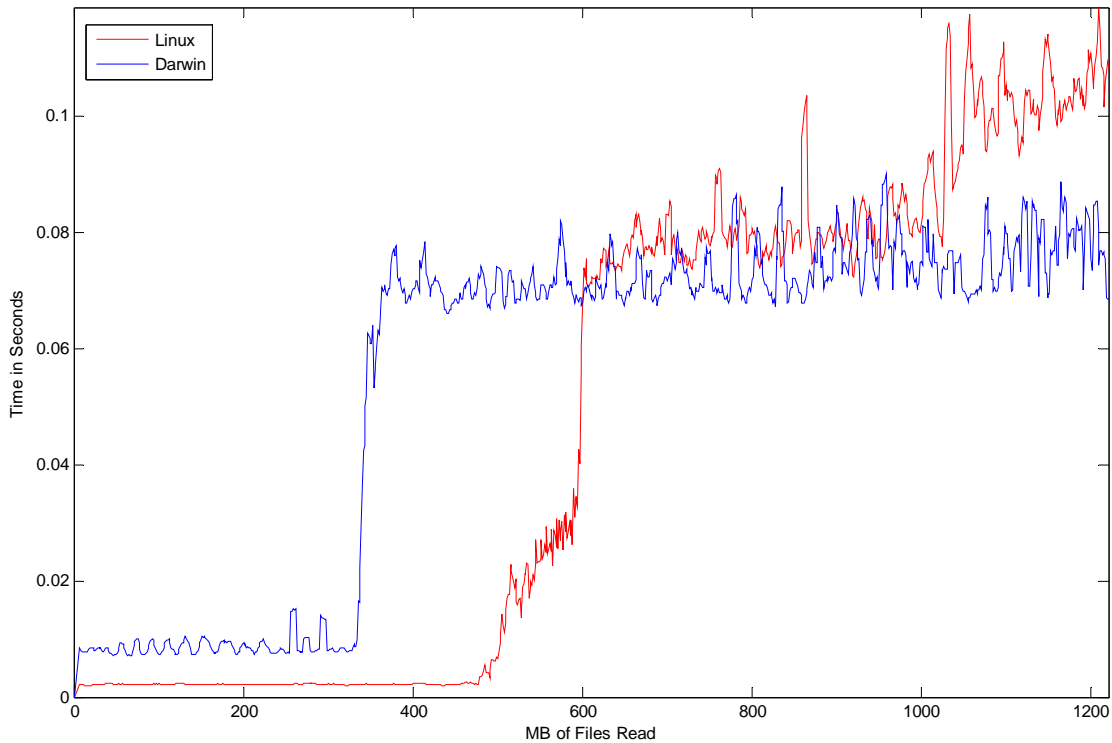


Figure 3 – Averaged Access Time for Testing File After A Given Amount of Other Content Was Read

Because of the way that our `Tester.c` code was structured, we also gathered data for the amount of time required to access `fileyay.txt` after it had just been previously accessed and thus should have been stored in the cache. Figure 4 shows this access time for both Darwin and Linux. Though Darwin appears to experience misses more frequently than Linux, its misses add only 1 millisecond to total access time while Linux's misses add 2 milliseconds. Despite this potential advantage, as was the case when less than 350MB of files were read, Darwin generally averages slightly more than 7 millisecond access times and Linux averages slightly more than 2 millisecond access times. Therefore, Linux's operating system overhead appears to be less than 30% of Darwin's under our testing conditions for basic file access from the cache.

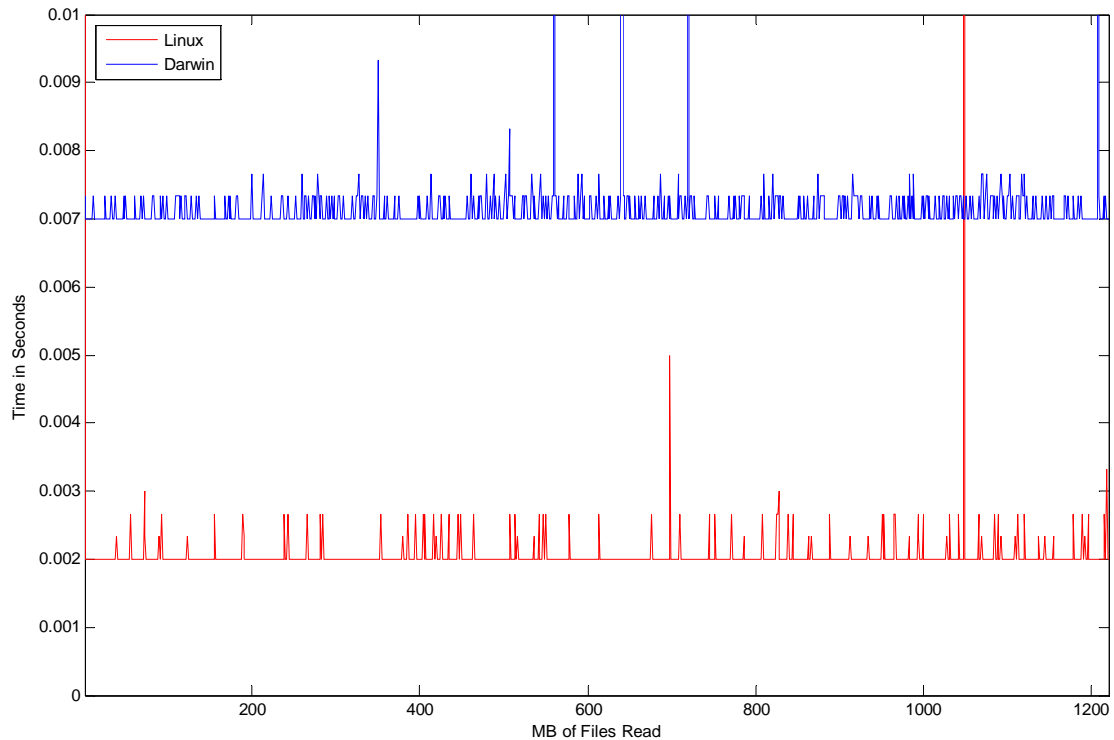


Figure 4 – Access Time for Testing File After it Was Just Accessed

Conclusions

For the majority of tests that we conducted, Ubuntu Linux was faster. We believe that our tests are at least qualitatively valid, and that more close attention to the code itself would probably show that the initial conclusions are accurate. More testing in various different edge cases of all of these hypothetical situations are necessary, as for some of them, we only tested what we thought would likely be the most important or compelling feature. The graphs for file I/O access times were particularly compelling in comparing raw speed of the operating systems. We naturally hesitate to say from our data alone that Linux is a superior operating system, as in order to claim that, many more factors would need to be taken into account, but we believe on a purely numerical basis, Linux holds a clear advantage for users who would use their system extensively at the edge limits of its capabilities.

Appendices:

Appendix I - File I/O Code:

`fileIO.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>
```

```

int main ( void)
{
    int i,j;
    char command[100];

    for(i=0;i<1000;i = i + 25)
        {
            for(j=0; j<10; j++)
                {
                    sprintf(command, "time ./fileIOoperation %d", i);
                    system(command);
                }
        }
    return(1);
}

```

fileIOoperation.c

```

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>

int main(int argc, char * argv[])
{
    int iterations;
    FILE* output_file=NULL;
    FILE* temp_file=NULL;
    clock_t startTime, end;
    double cpu_time_used;
    int numIterations;

    if(argc==2){
        numIterations = atoi(argv[1]);
    }
    else{
        printf("Proper Usage: fileIOoperation [Number of Files to
Create]\n");
        exit(-1);
    }
    for (iterations = 0; iterations < numIterations; iterations++)
        {
            if ((temp_file = fopen("tempfileIO.txt", "w+")) == NULL)
                {
                    printf("Error opening file\n");
                    exit(1);
                }
            fprintf(temp_file, "I love PPC\n");
            fclose(temp_file);
            sync();
        }
    return(1);
}

```

Appendix II - Disk Cache:

memclear.c

```

#include <stdio.h>

```

```

#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>
#include <string.h>

#define MAXSTRING 100

int main(int argc, char * argv[])
{
    FILE* temp;
    char command[MAXSTRING];
    int c, name_length, i;
    char filename[MAXSTRING];
    FILE *ifp;
    char junk[MAXSTRING];
    char filepath[MAXSTRING];
    char *to = filepath;
    int MinNumFilesToUse;
    int MaxNumFilesToUse;
    int NumFilesUsed;

    if(argc==3){
        MinNumFilesToUse = atoi(argv[1]);
        MaxNumFilesToUse = atoi(argv[2]);
    }
    else{
        printf("Proper Usage: memclear [Minimum Number of Files Read]
        [Maximum number of Files you Want to Use]\n");
        exit(-1);
    }
    printf("\nRange of Files: %d to %d", MinNumFilesToUse,
MaxNumFilesToUse);

    sprintf(command, "ls mp3s/ > temp/temp.txt");
    system(command);

    to = strcpy(to, "mp3s/");

    for(i = MinNumFilesToUse; i<=MaxNumFilesToUse; i++)
    {
        NumFilesUsed =0;
        system("time ./tester");
        ifp = fopen("temp/temp.txt", "r");
        while ((fgets(filename, MAXSTRING, ifp)) != NULL) &&
            (NumFilesUsed <i)
        {
            name_length = strlen(filename);
            filename[name_length-1] = 0;
            to = strcpy (to, filename);
            if ((temp = fopen(filepath,"r"))!=NULL) // exit(1);
            {
                while ((fgets(junk, MAXSTRING, temp)) != NULL)
                {
                    //      printf("%s", junk);
                }
                fclose(temp);
                NumFilesUsed++;
            }
            else printf("File did not open correctly");
        }
    };
}

```

```

        fclose(ifp);
        system("time ./tester");
    }
    return(0);
}

```

Tester.c

```

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <math.h>
#include <sys/times.h>
#include <time.h>

static struct tms _start; /* Stores the starting time*/
static struct tms _stop; /* Stores the ending time*/

#define MAXSTRING 100

void start()
{
    times(&_start);
}

void stop()
{
    times(&_stop);
}

unsigned long report()
{
    return (_stop.tms_utime - _start.tms_utime);
}

int main (void)
{
    FILE* yay;
    FILE* temp;
    char MAXSIZE[102400];
    char command[MAXSTRING];
    //FILE *tmp filename;
    int c, name_length, i;
    char filename[MAXSTRING];
    FILE *ifp;
    FILE *output_file;
    char junk[MAXSTRING];
    char filepath[MAXSTRING];
    char *to = filepath;

    if ((output_file = fopen("temp/data.txt","w"))==NULL)
        exit(1);
    // printf("Data file opened\n");

    start();
    if ((yay = fopen("temp/fileyay.txt","r"))==NULL)
        exit(1);
    // printf("Happy File opened\n");

    while (!feof(yay))
    {
        fgets(MAXSIZE, sizeof(MAXSIZE), yay);
    };
}

```

```

fclose(yay);

stop();
fprintf(output_file, "Cycles for core file #1\t%ld\n", report());
//printf("yay was read.\n");
fclose(output_file);
return(0);
}

```

Appendix III – Memory Exercises Regarding Hostile Processes

Memory.c

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <math.h>
#define MAXSTRING 100

int main(void)
{
    int i, iterations=10;
    for (i=0; i<iterations; i++)
    {
        //good(1,10000);
        stats();
        bad(100);
        good(1,2500);
    }
}

int bad (int iterations)
{
    int counter = 0;
    while (counter < iterations)
    {
        double *space = (double *) malloc (sizeof (double));
        counter++;
    }
}

int good (int iterations, int size)
{
    int i, j;
    int myarray[size];

    //printf("int * is %2d bytes \n", sizeof(int));
    for(i=0; i<iterations; i++)
    {
        for(j=0; j<size; j++)
        {
            myarray[j] = j;
        }
    }
    //printf("I did a good thing!\n");
    return(0);
}

int stats (void)
{
    char command[MAXSTRING];

```

```
    sprintf(command, "vmstat -s >> stats.txt");
    system(command);
    //printf("wrote vmstat to file\n");

    return(0);
}
```

ⁱ Bar, Moshe. Comparing Apples and Penguins. Byte.com. Published on October 22, 2002. Available from http://www.byte.com/documents/s=2288/byt1035828368066/1028_bar.html?temp=05xdw-u5lG.

ⁱⁱ Bar, Moshe. OS X: Should I Stay or Should I Go? Byte.com. Published on February 23, 2003. Available from

http://www.byte.com/documents/s=7865/byt1046092712069/0224_bar.html?temp=IyjNpeC23r

ⁱⁱⁱ <http://developer.apple.com/darwin/>

^{iv} This is not precisely true. Mac OS X includes what is referred to as the Classic environment, an emulation shell capable of running Mac OS 9 from within Mac OS X, and thusly any Mac OS 9 native applications. Although still available, the need for Classic has been progressively phased out since Mac OS X's debut.

^v http://developer.apple.com/documentation/MacOSX/Conceptual/OSX_Technology_Overview/

^{vi} <http://developer.apple.com/documentation/Performance/Conceptual/ManagingMemory/index.html>

^{vii} <http://developer.apple.com/documentation/Performance/Conceptual/ManagingMemory/index.html>

^{viii} http://developer.apple.com/documentation/MacOSX/Conceptual/OSX_Technology_Overview/

^{ix} http://en.wikipedia.org/wiki/Mach_kernel

^x http://developer.apple.com/documentation/MacOSX/Conceptual/OSX_Technology_Overview/

^{xi} <http://en.wikipedia.org/wiki/Linux>

^{xii} <http://home.earthlink.net/~jknappa/linux-mm/vmpolicy.html>

^{xiii} <http://www.macworld.com/2005/01/features/preventmacdisasters/index.php>